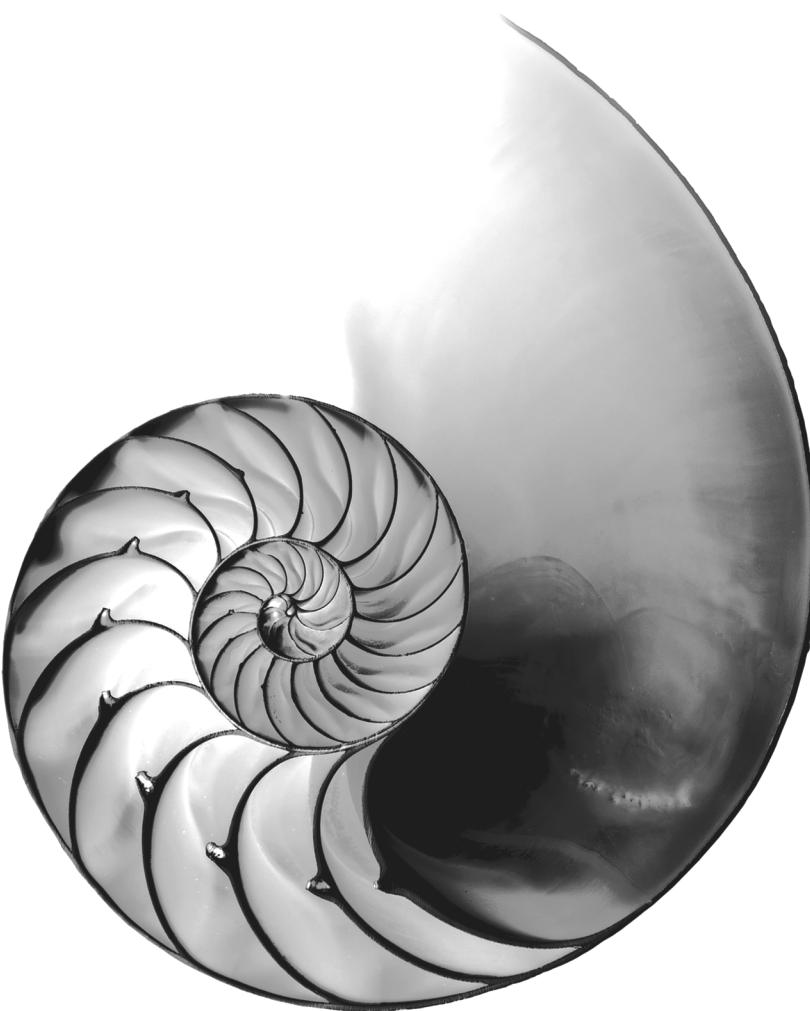


Introduction to JShell: Java 9's REPL for Interactive Java

25



Objectives

In this chapter you'll:

- See how using JShell can enhance the learning and software development processes by enabling you to explore, discover and experiment with Java language and API features.
- Start a JShell session.
- Execute code snippets.
- Declare variables explicitly.
- Evaluate expressions.
- Edit existing code snippets.
- Declare and use a class.
- Save snippets to a file.
- Open a file of JShell snippets and evaluate them.
- Auto-complete code and JShell commands.
- Display method parameters and overloads.
- Discover and explore with the Java API documentation in JShell.
- Declare and use methods.
- Forward reference a method that has not yet been declared.
- See how JShell wraps exceptions.
- Import custom packages for use in a JShell session.
- Control JShell's feedback level.



Outline

25.1	Introduction	25.7.4	Viewing a <code>public</code> Field's Documentation
25.2	Installing JDK 9	25.7.5	Viewing a Class's Documentation
25.3	Introduction to JShell	25.7.6	Viewing Method Overloads
25.3.1	Starting a JShell Session	25.7.7	Exploring Members of a Specific Object
25.3.2	Executing Statements	25.8 Declaring Methods	
25.3.3	Declaring Variables Explicitly	25.8.1	Forward Referencing an Undeclared Method—Declaring Method <code>displayCubes</code>
25.3.4	Listing and Executing Prior Snippets	25.8.2	Declaring a Previously Undeclared Method
25.3.5	Evaluating Expressions and Declaring Variables Implicitly	25.8.3	Testing <code>cube</code> and Replacing Its Declaration
25.3.6	Using Implicitly Declared Variables	25.8.4	Testing Updated Method <code>cube</code> and Method <code>displayCubes</code>
25.3.7	Viewing a Variable's Value	25.9 Exceptions	
25.3.8	Resetting a JShell Session	25.10 Importing Classes and Adding Packages to the <code>CLASSPATH</code>	
25.3.9	Writing Multiline Statements	25.11 Using an External Editor	
25.3.10	Editing Code Snippets	25.12 Summary of JShell Commands	
25.3.11	Exiting JShell	25.12.1	Getting Help in JShell
25.4	Command-Line Input in JShell	25.12.2	<code>/edit</code> Command: Additional Features
25.5	Declaring and Using Classes	25.12.3	<code>/reload</code> Command
25.5.1	Creating a Class in JShell	25.12.4	<code>/drop</code> Command
25.5.2	Explicitly Declaring Reference-Type Variables	25.12.5	Feedback Modes
25.5.3	Creating Objects	25.12.6	Other JShell Features Configurable with <code>/set</code>
25.5.4	Manipulating Objects	25.13 Keyboard Shortcuts for Snippet Editing	
25.5.5	Creating a Meaningful Variable Name for an Expression	25.14 How JShell Reinterprets Java for Interactive Use	
25.5.6	Saving and Opening Code-Snippet Files	25.15 IDE JShell Support	
25.6	Discovery with JShell Auto-Completion	25.16 Wrap-Up	
25.6.1	Auto-Completing Identifiers		
25.6.2	Auto-Completing JShell Commands		
25.7	Exploring a Class's Members and Viewing Documentation		
25.7.1	Listing Class Math's <code>static</code> Members		
25.7.2	Viewing a Method's Parameters		
25.7.3	Viewing a Method's Documentation		

Self-Review Exercises | Answers to Self-Review Exercises

25.1 Introduction

As educators, it's a joy to write this chapter on what may be the most important pedagogic improvement in Java since its inception more than two decades ago. The Java community—by far the largest programming language community in the world—has grown to more than 10 million developers. But along the way, not much has been done to improve the learning process for novice programmers. That changes dramatically in Java 9 with the introduction of **JShell**—Java's **REPL** (*read-evaluate-print loop*).¹

9

1. We'd like to thank Robert Field at Oracle—the head of the JShell/REPL effort. We interacted with Mr. Field extensively as we developed Chapter 25. He answered our many questions. We reported JShell bugs and made suggestions for improvement.

Instructors have indicated a preference in introductory programming courses for languages with REPLs—and now Java has a rich REPL implementation. And with the new JShell APIs, third parties will build JShell and related interactive-development tools into the major IDEs like Eclipse, IntelliJ, NetBeans and others. Java 9 and JShell are evolving rapidly, so we've placed all our Java 9 content online—we'll keep it up-to-date as Java 9 evolves.

What is JShell?

What's the magic? It's simple. JShell provides a fast and friendly environment that enables you to quickly explore, discover and experiment with Java language features and its extensive libraries. REPLs like the one in JShell have been around for decades. In the 1960s, one of the earliest REPLs made convenient interactive development possible in the LISP programming language. Students of that era, like one of your authors, Harvey Deitel, found it fast and fun to use.

JShell replaces the tedious cycle of editing, compiling and executing with its read-evaluate-print loop. Rather than complete programs, you write **JShell commands** and Java code snippets. When you enter a snippet, JShell *immediately reads* it, *evaluates* it and *prints* the results that help you see the effects of your code. Then it *loops* to perform this process again for the next snippet. As you work through Chapter 25's scores of examples and exercises, you'll see how JShell and its instant feedback keep your attention, enhance your performance and speed the learning and software development processes.

Code Comes Alive

As you know, we emphasize the value of the live-code teaching approach in our books, focusing on *complete*, working programs. JShell brings this right down to the individual snippet level. Your code literally comes alive as you enter each line. Of course, you'll still make occasional errors as you enter your snippets. JShell reports compilation errors to you on a snippet-by-snippet basis. You can use this capability, for example, to test the items in our Common Programming Error tips and see the errors as they occur.

Kinds of Snippets

Snippets can be expressions, individual statements, multi-line statements and larger entities, like methods and classes. JShell supports all but a few Java features, but there are some differences designed to facilitate JShell's explore–discover–experiment capabilities. In JShell, methods do not need to be in classes, expressions and statements do not need to be in methods, and you do not need a `main` method (other differences are in Section 25.14). Eliminating this infrastructure saves you considerable time, especially compared to the lengthy repeated edit, compile and execute cycles of complete programs. And because JShell automatically displays the results of evaluating your expressions and statements, you do not need as many `print` statements as we use throughout this book's traditional Java code examples.

Discovery with Auto-Completion

We include a detailed treatment of **auto-completion**—a key discovery feature that speeds the coding process. After you type a portion of a name (class, method, variable, etc.) and press the *Tab* key, JShell completes the name for you or provides a list of all possible names that begin with what you've typed so far. You can then easily display method parameters and even the documentation that describes those methods.

Rapid Prototyping

Professional developers will commonly use JShell for rapid prototyping but not for full-out software development. Once you develop and test a small chunk of code, you can then paste it in to your larger project.

How This Chapter Is Organized

Chapter 25 is optional. For those who want to use JShell, the chapter has been designed as a series of units, paced to certain earlier chapters of the print book. Each unit begins with a statement like: “This section may be read after Chapter 2.” So you’d begin by reading through Chapter 2, then read the corresponding section of this chapter—and similarly for subsequent chapters.

The Chapter 2 JShell Exercises

As you work your way through this chapter, execute each snippet and command in JShell to confirm that the features work as advertised. Sections 25.3–25.4 are designed to be read after Chapter 2. Once you read these sections, we recommend that you do Chapter 25’s dozens of self-review exercises. JShell encourages you to “learn by doing,” so the exercises have you write and test code snippets that exercise many of Chapter 2’s Java features.

The self-review exercises are small and to the point, and the answers are provided to help you quickly get comfortable with JShell’s capabilities. When you’re done you’ll have a great sense of what JShell is all about. Please tell us what you think of this new Java tool. Thanks!

Instead of rambling on about the advantages of JShell, we’re going to let JShell itself convince you. If you have any questions as you work through the following examples and exercises, just write to us at deitel@deitel.com and we’ll always respond promptly.

9 25.2 Installing JDK 9

Java 9 and its JShell are early access technologies that are still under development. This introduction to JShell is based on the JDK 9 Developer Preview (early access build 163). To use JShell, you must first install JDK 9, which is available in early access form at

<https://jdk9.java.net/download/>

The Before You Begin section that follows the Preface discusses the JDK version numbering schemes, then shows how to manage multiple JDK installations on your particular platform.

25.3 Introduction to JShell

[*Note:* This section may be read after studying Chapter 2, Introduction to Java Applications; Input/Output and Operators.]

In Chapter 2, to create a Java application, you:

1. created a class containing a `main` method.
2. declared in `main` the statements that will execute when you run the program.
3. compiled the program and fixed any compilation errors that occurred. This step had to be repeated until the program compiled without errors.
4. ran the program to see the results.

By automatically compiling and executing code as you complete each expression or statement, JShell eliminates the overhead of

- creating a class containing the code you wish to test,
- compiling the class and
- executing the class.

Instead, you can focus on interactively discovering and experimenting with Java's language and API features. If you enter code that does not compile, JShell immediately reports the errors. You can then use JShell's editing features to quickly fix and re-execute the code.

25.3.1 Starting a JShell Session

To start a JShell session in:

- Microsoft Windows, open a **Command Prompt** then type **jshell** and press *Enter*.
- macOS (formerly OS X), open a **Terminal** window then type the following command and press *Enter*.

```
$JAVA_HOME/bin/jshell
```

- Linux, open a shell window then type **jshell** and press *Enter*.

The preceding commands execute a new JShell session and display the following message and the **jshell> prompt**:

```
| Welcome to JShell -- Version 9-ea  
| For an introduction type: /help intro  
  
jshell>
```

In the first line above, "Version 9-ea" indicates that you're using the ea (that is, early access) version of JDK 9. JShell precedes informational messages with vertical bars (|). You are now ready to enter Java code or JShell commands.

9

25.3.2 Executing Statements

[*Note:* As you work through this chapter, type the same code and JShell commands that we show at each **jshell>** prompt to ensure that what you see on your screen will match what we show in the sample outputs.]

JShell has two input types:

- Java code (which the JShell documentation refers to as **snippets**) and
- JShell commands.

In this section and Section 25.3.3, we begin with Java code snippets. Subsequent sections introduce JShell commands.

You can type any expression or statement at the **jshell>** prompt then press *Enter* to execute the code and see its results immediately. Consider the program of Fig. 2.1, which we show again in Fig. 25.1. To demonstrate how `System.out.println` works, this program required many lines of code and comments, which you had to write, compile and execute. Even without the comments, five code lines were still required (lines 4 and 9–9).

```

1 // Fig. 25.1: Welcome1.java
2 // Text-printing program.
3
4 public class Welcome1 {
5     // main method begins execution of Java application
6     public static void main(String[] args) {
7         System.out.println("Welcome to Java Programming!");
8     } // end method main
9 } // end class Welcome1

```

Welcome to Java Programming!

Fig. 25.1 | Text-printing program.

In JShell, you can execute the statement in line 7 without creating all the infrastructure of class `Welcome1` and its `main` method:

```
jshell> System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

In this case, JShell displays the snippet’s command-line output below the initial `jshell>` prompt and the statement you entered. Per our convention, we show user inputs in bold.

Notice that we did not enter the preceding statement’s semicolon (;). JShell adds *only* terminating semicolons.² You need to add a semicolon if the end of the statement is not the end of the line—for example, if the statement is inside braces ({ and }). Also, if there is more than one statement on a line then you need a semicolon between statements, but not after the last statement.

The blank line before the second `jshell>` prompt is the result of the newline displayed by method `println` and the newline that JShell always displays before each `jshell>` prompt. Using `print` rather than `println` eliminates the blank line:

```
jshell> System.out.print("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

JShell keeps track of everything you type, which can be useful for re-executing prior statements and modifying statements to update the tasks they perform.

25.3.3 Declaring Variables Explicitly

Almost anything you can declare in a typical Java source-code file also can be declared in JShell (Section 25.14 discusses some of the features you cannot use). For example, you can explicitly declare a variable as follows:

```
jshell> int number1
number1 ==> 0

jshell>
```

2. Not requiring semicolons is one example of how JShell reinterprets standard Java for convenient interactive use. We discuss several of these throughout the chapter and summarize them in Section 25.14.

When you enter a variable declaration, JShell displays the variable's name (in this case, `number1`) followed by `==>` (which means, “has the value”) and the variable's initial value (0). If you do not specify an initial value explicitly, the variable is initialized to its type's default value—in this case, 0 for an `int` variable.

A variable can be initialized in its declaration—let's redeclare `number1`:

```
jshell> int number1 = 30
number1 ==> 30

jshell>
```

JShell displays

```
number1 ==> 30
```

to indicate that `number1` now has the value 30. When you declare a new variable with the *same name* as another variable in the current JShell session, JShell replaces the first declaration with the new one.³ Because `number1` was declared previously, we could have simply assigned `number1` a value, as in

```
jshell> number1 = 45
number1 ==> 45

jshell>
```

Compilation Errors in JShell

You must declare variables before using them in JShell. The following declaration of `int` variable `sum` attempts to use a variable named `number2` that we have not yet declared, so JShell reports a compilation error, indicating that the compiler was unable to find a variable named `number2`:

```
jshell> int sum = number1 + number2
| Error:
| cannot find symbol
|   symbol:  variable number2
|   int sum = number1 + number2;
|                                ^----^

jshell>
```

The error message uses the notation `^----^` to highlight the error in the statement. No error is reported for the previously declared variable `number1`. Because this snippet has a compilation error, it's invalid. However, JShell still maintains the snippet as part of the JShell session's history, which includes valid snippets, invalid snippets and commands that you've typed. As you'll soon see, you can recall this invalid snippet and execute it again later. JShell's `/history` command displays the current session's history—that is, *everything* you've typed:

3. Redeclaring an existing variable is another example of how JShell reinterprets standard Java for interactive use. This behavior is different from how the Java compiler handles a new declaration of an existing variable—such a “double declaration” generates a compilation error.

```
jshell> /history

System.out.println("Welcome to Java Programming!")
System.out.print("Welcome to Java Programming!")
int number1
int number1 = 45
number1 = 45
int sum = number1 + number2
/history

jshell>
```

Fixing the Error

JShell makes it easy to fix a prior error and re-execute a snippet. Let's fix the preceding error by first declaring `number2` with the value 72:

```
jshell> int number2 = 72
number2 ==> 72

jshell>
```

Subsequent snippets can now use `number2`—in a moment, you'll re-execute the snippet that declared and initialized `sum` with `number1 + number2`.

Recalling and Re-executing a Previous Snippet

Now that both `number1` and `number2` are declared, we can declare the `int` variable `sum`. You can use the up and down arrow keys to navigate backward and forward through the snippets and JShell commands you've entered previously. Rather than retyping `sum`'s declaration, you can press the up arrow key three times to recall the declaration that failed previously. JShell recalls your prior inputs in reverse order—the last line of text you typed is recalled first. So, the first time you press the up arrow key, the following appears at the `jshell>` prompt:

```
jshell> int number2 = 72
```

The second time you press the up arrow key, the `/history` command appears:

```
jshell> /history
```

The third time you press the up arrow key, `sum`'s prior declaration appears:

```
jshell> int sum = number1 + number2
```

Now you can press *Enter* to re-execute the snippet that declares and initializes `sum`:

```
jshell> int sum = number1 + number2
sum ==> 117

jshell>
```

JShell adds the values of `number1` (45) and `number2` (72), stores the result in the new `sum` variable, then shows `sum`'s value (117).

25.3.4 Listing and Executing Prior Snippets

You can view a list of all previous valid Java code snippets with JShell's `/List` command—JShell displays the snippets in the order you entered them:

```
jshell> /list

1 : System.out.println("Welcome to Java Programming!")
2 : System.out.print("Welcome to Java Programming!")
4 : int number1 = 30;
5 : number1 = 45
6 : int number2 = 72;
7 : int sum = number1 + number2;

jshell>
```

Each valid snippet is identified by a sequential **snippet ID**. The snippet with ID 3 is *missing* above, because we replaced that original snippet

```
int number1
```

with the one that has the ID 4 in the preceding `/list`. Note that `/list` may not display everything that `/history` does. As you recall, if you omit a terminating semicolon, JShell inserts it for you behind the scenes. When you say `/list`, *only* the declarations (snippets 4, 6 and 7) actually show the semicolons that JShell inserted.

Snippet 1 above is just an expression. If we type it with a terminating semicolon, it's an **expression statement**.

Executing Snippets By ID Number

You can execute any prior snippet by typing `/id`, where *id* is the snippet's ID. For example, when you enter `/1`:

```
jshell> /1
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!
```

```
jshell>
```

JShell displays the first snippet we entered, executes it and shows the result.⁴ You can re-execute the last snippet you typed (whether it was valid or invalid) with `/?!`:

```
jshell> /!
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!
```

```
jshell>
```

JShell assigns an ID to every valid snippet you execute, so even though

```
System.out.println("Welcome to Java Programming!")
```

already exists in this session as snippet 1, JShell creates a new snippet with the next ID in sequence (in this case, 8 and 9 for the last two snippets). Executing the `/list` command shows that snippets 1, 8 and 9 are identical:

4. At the time of this writing, you cannot use the `/id` command to execute a *range* of previous snippets; however, the JShell command `/reload` can re-execute *all* existing snippets (Section 25.12.3).

```
jshell> /list

1 : System.out.println("Welcome to Java Programming!")
2 : System.out.print("Welcome to Java Programming!")
4 : int number1 = 30;
5 : number1 = 45
6 : int number2 = 72;
7 : int sum = number1 + number2;
8 : System.out.println("Welcome to Java Programming!")
9 : System.out.println("Welcome to Java Programming!")

jshell>
```

25.3.5 Evaluating Expressions and Declaring Variables Implicitly

When you enter an expression in JShell, it evaluates the expression, implicitly creates a variable and assigns the expression's value to the variable. **Implicit variables** are named `$#`, where `#` is the new snippet's ID.⁵ For example:

```
jshell> 11 + 5
$10 ==> 16

jshell>
```

evaluates the expression `11 + 5` and assigns the resulting value (16) to the implicitly declared variable `$10`, because there were nine prior valid snippets (even though one was deleted because we redeclared the variable `number1`). JShell *infers* that the type of `$10` is `int`, because the expression `11 + 5` adds two `int` values, producing an `int`. Expressions may also include one or more method calls. The list of snippets is now:

```
jshell> /list

1 : System.out.println("Welcome to Java Programming!")
2 : System.out.print("Welcome to Java Programming!")
4 : int number1 = 30;
5 : number1 = 45
6 : int number2 = 72;
7 : int sum = number1 + number2;
8 : System.out.println("Welcome to Java Programming!")
9 : System.out.println("Welcome to Java Programming!")
10 : 11 + 5

jshell>
```

Note that the implicitly declared variable `$10` appears in the list simply as `10` without the `$`.

25.3.6 Using Implicitly Declared Variables

Like any other declared variable, you can use an implicitly declared variable in an expression. For example, the following assigns to the *existing* variable `sum` the result of adding `number1` (45) and `$10` (16):

-
5. Implicitly declared variables are another example of how JShell reinterprets standard Java for interactive use. In regular Java programs you must explicitly declare *every* variable.

```
jshell> sum = number1 + $10
sum ==> 61

jshell>
```

The list of snippets is now:

```
jshell> /list

1 : System.out.println("Welcome to Java Programming!")
2 : System.out.print("Welcome to Java Programming!")
4 : int number1 = 30;
5 : number1 = 45
6 : int number2 = 72;
7 : int sum = number1 + number2;
8 : System.out.println("Welcome to Java Programming!")
9 : System.out.println("Welcome to Java Programming!")
10: 11 + 5
11: sum = number1 + $10

jshell>
```

25.3.7 Viewing a Variable's Value

You can view a variable's value at any time simply by typing its name and pressing *Enter*:

```
jshell> sum
sum ==> 61

jshell>
```

JShell treats the variable name as an expression and simply evaluates its value.

25.3.8 Resetting a JShell Session

You can remove all prior code from a JShell session by entering the **/reset** command:

```
jshell> /reset
| Resetting state.

jshell> /list

jshell>
```

The subsequent **/list** command shows that all prior snippets were removed. Confirmation messages displayed by JShell, such as

```
| Resetting state.
```

are helpful when you're first becoming familiar with JShell. In Section 25.12.5, we'll show how you can change the JShell *feedback mode*, making it more or less verbose.

25.3.9 Writing Multiline Statements

Next, we write an **if** statement that determines whether 45 is less than 72. First, let's store 45 and 72 in implicitly declared variables, as in:

```
jshell> 45
$1 ==> 45

jshell> 72
$2 ==> 72

jshell>
```

Next, begin typing the `if` statement:

```
jshell> if ($1 < $2) {
    ...>
```

JShell knows that the `if` statement is incomplete, because we typed the opening left brace, but did not provide a body or a closing right brace. So, JShell displays the **continuation prompt** `...>` at which you can enter more of the control statement. The following completes and evaluates the `if` statement:

```
jshell> if ($1 < $2) {
    ...>     System.out.printf("%d < %d%n", $1, $2);
    ...> }
45 < 72

jshell>
```

In this case, a second continuation prompt appeared because the `if` statement was still missing its terminating right brace `}`. Note that the statement-terminating semicolon `(;)` at the end of the `System.out.printf` statement in the `if`'s body is required. We manually indented the `if`'s body statement—JShell does *not* add spacing or braces for you as IDEs generally do. Also, JShell assigns each multiline code snippet—such as an `if` statement—only one snippet ID. The list of snippets is now:

```
jshell> /list

1 : 45
2 : 72
3 : if ($1 < $2) {
    System.out.printf("%d < %d%n", $1, $2);
}

jshell>
```

25.3.10 Editing Code Snippets

Sometimes you might want to create a new snippet, based on an existing snippet in the current JShell session. For example, suppose you want to create an `if` statement that determines whether `$1` is *greater than* `$2`. The statement that performs this task

```
if ($1 > $2) {
    System.out.printf("%d > %d%n", $1, $2);
}
```

is nearly identical to the `if` statement in Section 25.3.9, so it would be easier to edit the existing statement rather than typing the new one from scratch. When you edit a snippet, JShell saves the edited version as a new snippet with the next snippet ID in sequence.

Editing a Single-Line Snippet

To edit a single-line snippet, locate it with the up-arrow key, make your changes within the snippet then press *Enter* to evaluate it. See Section 25.13 for some keyboard shortcuts that can help you edit single-line snippets.

Editing a Multiline Snippet

For a larger snippet that's spread over several lines—such as a `if` statement that contains one or more statements—you can edit the entire snippet by using JShell's `/edit` command to open the snippet in the **JShell Edit Pad** (Fig. 25.2). The command

```
/edit
```

opens **JShell Edit Pad** and displays *all* valid code snippets you've entered so far. To edit a specific snippet, include the snippet's ID, as in

```
/edit id
```

So, the command:

```
/edit 3
```

displays the `if` statement from Section 25.3.9 in **JShell Edit Pad** (Fig. 25.2)—no snippet IDs are shown in this window. **JShell Edit Pad**'s window is *modal*—that is, while it's open, you cannot enter code snippets or commands at the JShell prompt.

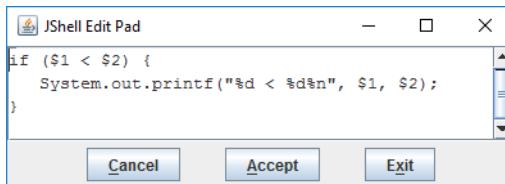


Fig. 25.2 | **JShell Edit Pad** showing the `if` statement from Section 25.3.9.

JShell Edit Pad supports only basic editing capabilities. You can:

- click to insert the cursor at a specific position to begin typing,
- move the cursor via the arrow keys on your keyboard,
- drag the mouse to select text,
- use the *Delete (Backspace)* key to delete text,
- cut, copy and paste text using your operating system's keyboard shortcuts, and
- enter text, including new snippets separate from the one(s) you're editing.

In the first and second lines of the `if` statement, select each less than operator (`<`) and change it to a greater than operator (`>`), then click **Accept** to create a new `if` statement containing the edited code. When you click **Accept**, JShell also immediately evaluates the new `if` statement and displays its results (if any)—because `$1` (45) is *not* greater than `$2` (72) the `System.out.printf` statement does not execute,⁶ so no additional output is shown in JShell.

6. We could have made this an `if...else` statement to show output when the condition is *false*, but this section is meant to be used with Chapter 2 where we introduce only the single-selection `if` statement.

If you want to return immediately to the JShell prompt, rather than clicking **Accept**, you could click **Exit** to execute the edited snippet and close **JShell Edit Pad**. Clicking **Cancel** closes **JShell Edit Pad** and discards any changes you made since the last time you clicked **Accept**, or since **JShell Edit Pad** was launched if have not yet clicked **Accept**.

When you change or create multiple snippets then click **Accept** or **Exit**, JShell compares the **JShell Edit Pad** contents with the previously saved snippets. It then executes every modified or new snippet.

Adding a New Snippet Via JShell Edit Pad

To show that **JShell Edit Pad** does, in fact, execute snippets immediately when you click **Accept**, let's change \$1's value to 100 by entering the following statement following the **if** statement after the other code in **JShell Edit Pad**:

```
$1 = 100
```

and clicking **Accept** (Fig. 25.3). Each time you modify a variable's value, JShell immediately displays the variable's name and new value:

```
jshell> /edit 3
$1 ==> 100
```

Click **Exit** to close **JShell Edit Pad** and return to the **jshell>** prompt.

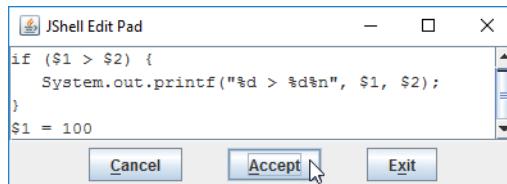


Fig. 25.3 | Entering a new statement following the **if** statement in **JShell Edit Pad**.

The following lists the current snippets—notice that each multiline **if** statement has only one ID:

```
jshell> /list

1 : 45
2 : 72
3 : if ($1 < $2) {
    System.out.printf("%d < %d\n", $1, $2);
}
4 : if ($1 > $2) {
    System.out.printf("%d > %d\n", $1, $2);
}
5 : $1 = 100

jshell>
```

*Executing the New **if** Statement Again*

The following re-executes the new **if** statement (ID 4) with the updated \$1 value:

```
jshell> /4
if ($1 > $2) {
    System.out.printf("%d > %d%n", $1, $2);
}
100 > 72

jshell>
```

The condition `$1 > $2` is now `true`, so the `if` statement's body executes. The list of snippets is now

```
jshell> /list

1 : 45
2 : 72
3 : if ($1 < $2) {
    System.out.printf("%d < %d%n", $1, $2);
}
4 : if ($1 > $2) {
    System.out.printf("%d > %d%n", $1, $2);
}
5 : $1 = 100
6 : if ($1 > $2) {
    System.out.printf("%d > %d%n", $1, $2);
}

jshell>
```

25.3.11 Exiting JShell

To terminate the current JShell session, use the `/exit` command or type the keyboard shortcut `Ctrl + d` (or `control + d`). This returns you to the command-line prompt in your **Command Prompt** (in Windows), **Terminal** (in macOS) or **shell** (in Linux—sometimes called **Terminal**, depending on your Linux distribution).

25.4 Command-Line Input in JShell

[*Note:* This section may be read after studying Chapter 2, Introduction to Java Applications; Input/Output and Operators and the preceding sections in this chapter.]

In Chapter 2, we showed command-line input using a `Scanner` object:

```
Scanner input = new Scanner(System.in);

System.out.print("Enter first integer: ");
int number1 = input.nextInt();
```

We created a `Scanner`, prompted the user for input, then used `Scanner` method `nextInt` to read a value. Recall that the program then waited for you to type an integer and press *Enter* before proceeding to the next statement. The on-screen interaction appeared as:

```
Enter first integer: 45
```

This section shows what that interaction looks like in JShell.

Creating a Scanner

Start a new JShell session or /reset the current one, then create a Scanner object:

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+]
\E[infinity string=\Q\E]

jshell>
```

You do not need to import Scanner. JShell automatically imports the java.util package and several others—we show the complete list in Section 25.10. When you create an object, JShell displays its text representation. The notation to the right of `input ==>` is the Scanner’s text representation (which you can simply ignore).

Prompting for Input and Reading a Value

Next, prompt the user for input:

```
jshell> System.out.print("Enter first integer: ")
Enter first integer:
jshell>
```

The statement’s output is displayed immediately, followed by the next `jshell>` prompt. Now enter the input statement:

```
jshell> int number1 = input.nextInt()
-
```

At this point, JShell waits for your input. The input cursor is positioned below the `jshell>` prompt and snippet you just entered—indicated by the underscore (`_`) above—rather than next to the prompt "Enter first integer:" as it was in Chapter 2. Now type an integer and press *Enter* to assign it to `number1`—the last snippet’s execution is now complete, so the next `jshell>` prompt appears.:

```
jshell> int number1 = input.nextInt()
45
number1 ==> 45

jshell>
```

Though you can use Scanner for command-line input in JShell, in most cases it’s unnecessary. The goal of the preceding interactions was simply to store an integer value in the variable `number1`. You can accomplish that in JShell with the simple assignment

```
jshell> int number1 = 45
number1 ==> 45

jshell>
```

For this reason, you’ll typically use assignments, rather than command-line input in JShell. We introduced Scanner here, because sometimes you’ll want to copy code you developed in JShell into a conventional Java program.

25.5 Declaring and Using Classes

[*Note:* This section may be read after studying Chapter 7, Introduction to Classes and Objects.]

In Section 25.3, we demonstrated basic JShell capabilities. In this section, we create a class and manipulate an object of that class. We'll use the version of class `Account` presented in Fig. 7.1.

25.5.1 Creating a Class in JShell

Start a new JShell session (or `/reset` the current one), then declare class `Account`—we ignored the comments from Fig. 7.1:

```
jshell> public class Account {  
...>     private String name;  
...>  
...>     public void setName(String name) {  
...>         this.name = name;  
...>     }  
...>  
...>     public String getName() {  
...>         return name;  
...>     }  
...> }  
| created class Account  
  
jshell>
```

JShell recognizes when you enter the class's closing brace—then displays

```
| created class Account
```

and issues the next `jshell>` prompt. Note that the semicolons throughout class `Account`'s body are required.

To save time, rather than typing a class's code as shown above, you can load an existing source code file into JShell, as shown in Section 25.5.6. Though you can specify access modifiers like `public` on your classes (and other types), JShell ignores all access modifiers on the top-level types except for `abstract` (discussed in Chapter 10).

Viewing Declared Classes

To view the names of the classes you've declared so far, enter the `/types` command:⁷

```
jshell> /types  
|   class Account  
  
jshell>
```

25.5.2 Explicitly Declaring Reference-Type Variables

The following creates the `Account` variable `account`:

```
jshell> Account account  
account ==> null  
  
jshell>
```

The default value of a reference-type variable is `null`.

7. `/types` actually displays all types you declare, including classes, interfaces and `enums`.

25.5.3 Creating Objects

You can create new objects. The following creates an `Account` variable named `account` and initializes it with a new object:

```
jshell> account = new Account()
account ==> Account@56ef9176

jshell>
```

The strange notation

```
Account@56ef9176
```

is the default text representation of the new `Account` object. If a class provides a custom text representation, you'll see that instead. We show how to provide a custom text representation for objects of a class in Section 7.6. We discuss the default text representation of objects in Section 9.6. The value after the @ symbol is the object's *hashcode*. We discuss hashcodes in Section 16.10.

Declaring an Implicit Account Variable Initialized with an Account Object

If you create an object with only the expression `new Account()`, JShell assigns the object to an implicit variable of type `Account`, as in:

```
jshell> new Account()
$4 ==> Account@1ed4004b

jshell>
```

Note that this object's hashcode (`1ed4004b`) is different from the prior `Account` object's hashcode (`56ef9176`)—these typically are different, but that's not guaranteed.

Viewing Declared Variables

You can view all the variables you've declared so far with the JShell `/vars` command:

```
jshell> /vars
|   Account account = Account@56ef9176
|   Account $4 = Account@1ed4004b

jshell>
```

For each variable, JShell shows the type and variable name followed by an equal sign and the variable's text representation.

25.5.4 Manipulating Objects

Once you have an object, you can call its methods. In fact, you already did this with the `System.out` object by calling its `println`, `print` and `printf` methods in earlier snippets.

The following sets the `account` object's name:

```
jshell> account.setName("Amanda")
jshell>
```

The method `setName` has the return type `void`, so it does not return a value and JShell does not show any additional output.

The following gets the `account` object's name:

```
jshell> account.getName()  
$6 ==> "Amanda"  
  
jshell>
```

Method `getName` returns a `String`. When you invoke a method that returns a value, JShell stores the value in an implicitly declared variable. In this case, `$6`'s type is *inferred* to be `String`. Of course, you could have assigned the result of the preceding method call to an explicitly declared variable.

Using the Return Value of a Method in a Statement

If you invoke a method as part of a larger statement, the return value is used in that statement, rather than stored. For example, the following uses `println` to display the `account` object's name:

```
jshell> System.out.println(account.getName())  
Amanda  
  
jshell>
```

25.5.5 Creating a Meaningful Variable Name for an Expression

You can give a meaningful variable name to a value that JShell previously assigned to an implicit variable. For example, with the following snippet recalled

```
jshell> account.getName()  
type  
Shift + Tab v
```

The `+ v` notation means that you should press *both* the `Shift` and `Tab` keys together, then release those keys and press `v`. JShell infers the expression's type and begins a variable declaration for you—`account.getName()` returns a `String`, so JShell inserts `String` and an equal sign (`=`) before the expression, as in

```
jshell> account.getName()  
jshell> String _= account.getName()
```

JShell also positions the cursor (indicated by the `_` above) immediately before the `=` so you can simply type the variable name, as in

```
jshell> String name = account.getName()  
name ==> "Amanda"  
  
jshell>
```

When you press *Enter*, JShell evaluates the new snippet and stores the value in the specified variable.

25.5.6 Saving and Opening Code-Snippet Files

You can save all of a session's valid code snippets to a file, which you can then load into a JShell session as needed.

Saving Snippets to a File

To save just the *valid* snippets, use the `/save` command, as in:

```
/save filename
```

By default, the file is created in the folder from which you launched JShell. To store the file in a different location, specify the complete path of the file.

Loading Snippets from a File

Once you save your snippets, they can be reloaded with the `/open` command:

```
/open filename
```

which executes each snippet in the file.

Using /open to Load Java Source-Code Files

You also can open existing Java source code files using `/open`. For example, let's assume you'd like to experiment with class `Account` from Fig. 7.1 (as you did in Section 25.5.1). Rather than typing its code into JShell, you can save time by loading the class from the source file `Account.java`. In a command window, you'd change to the folder containing `Account.java`, execute JShell, then use the following command to load the class declaration into JShell:

```
/open Account.java
```

To load a file from another folder, you can specify the full pathname of the file to open. In Section 25.10, we'll show how to use existing compiled classes in JShell.

25.6 Discovery with JShell Auto-Completion

[*Note:* This section may be read after studying Chapter 3, Control Statements: Part 1; Assignment, `++` and `--` Operators, and completing Section 25.5.]

JShell can help you write code. When you partially type the name of an existing class, variable or method then press the *Tab* key, JShell does one of the following:

- If no other name matches what you've typed so far, JShell enters the rest of the name for you.
- If there are multiple names that begin with the same letters, JShell displays a list of those names to help you decide what to type next—then you can type the next letter(s) and press *Tab* again to complete the name.
- If no names match what you typed so far, JShell does nothing and your operating system's alert sound plays as feedback.

Auto-completion is normally an IDE feature, but with JShell it's IDE independent.

Let's first list the snippets we've entered since the last `/reset` (from Section 25.5):

```
jshell> /list
1 : public class Account {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
2 : Account account;
3 : account = new Account()
4 : new Account()
5 : account.setName("Amanda")
6 : account.getName()
7 : System.out.println(account.getName())
8 : String name = account.getName();

jshell>
```

25.6.1 Auto-Completing Identifiers

The only variable declared so far that begins with lowercase "a" is `account`, which was declared in snippet 2. Auto-completion is case sensitive, so "a" does not match the class name `Account`. If you type "a" at the `jshell>` prompt:

```
jshell> a
```

then press *Tab*, JShell auto-completes the name:

```
jshell> account
```

If you then enter a dot:

```
jshell> account.
```

then press *Tab*, JShell does not know what method you want to call, so it displays a list of everything—in this case, all the methods—that can appear to the right of the dot:

```
jshell> account.
equals(      getClass()      getName()      hashCode()      notify()
notifyAll()  setName(       toString()      wait(
```



```
jshell> account.
```

and follows the list with a new `jshell>` prompt that includes what you've typed so far. The list includes the methods we declared in class `Account` (snippet 1) *and* several methods that all Java classes have (as we discuss in Chapter 9). In the list of method names

- those followed by "`()`" are methods that do not require arguments and
- those followed only by "`(`" are methods that either require at least one argument or that are so-called *overloaded methods*—multiple methods with the same name, but different parameter lists (discussed in Section 5.12).

Let's assume you want to use `Account`'s `setName` method to change the name stored in the `account` object to "John". There's only one method that begins with "s", so you can type s then *Tab* to auto-complete `setName`:

```
jshell> account.setName(
```

JShell automatically inserts the method call's opening left parenthesis. Now you can complete the snippet as in:

```
jshell> account.setName("John")
```

```
jshell>
```

25.6.2 Auto-Completing JShell Commands

Auto-completion also works for JShell commands. If you type / then press *Tab*, JShell displays the list of JShell commands:

```
jshell> /
!          /?          /drop      /edit      /env       /exit
/help      /history    /imports   /list      /methods   /open
/reload    /reset     /save      /set       /types    /vars
<press tab again to see synopsis>
jshell> /
```

If you then type h and press *Tab*, JShell displays only the commands that start with /h:

```
jshell> /h
/help      /history
<press tab again to see synopsis>
jshell> /h
```

Finally, if you type "i" and press *Tab*, JShell auto-completes /history. Similarly, if you type /l then press *Tab*, JShell auto-completes the command as /list, because only that command starts with /l.

25.7 Exploring a Class's Members and Viewing Documentation

[*Note:* This section may be read after studying Chapter 5, Methods, and the preceding portions of Chapter 25.]

The preceding section introduced basic auto-completion capabilities. When using JShell for experimentation and discovery, you'll often want to learn more about a class before using it. In this section, we'll show you how to:

- view the parameters required by a method so that you can call it correctly
- view the documentation for a method
- view the documentation for a field of a class
- view the documentation for a class, and
- view the list of overloads for a given method.

To demonstrate these features, let's explore class `Math`. Start a new JShell session or `/reset` the current one.

25.7.1 Listing Class `Math`'s static Members

As we discussed in Chapter 5, class `Math` contains only `static` members—`static` methods for various mathematical calculations and the `static` constants `PI` and `E`. To view a complete list, type "`Math.`" then press `Tab`:

```
jshell> Math.  
E           IEEEremainder()   PI           abs()  
acos()      addExact()       asin()       atan()  
atan2()     cbrt()          ceil()       class  
copySign()  cos()           cosh()       decrementExact()  
exp()       expm1()         floor()      floorDiv()  
floorMod()  fma()          getExponent() hypot()  
incrementExact() log()        log10()      log1p()  
max()       min()          multiplyExact() multiplyFull()  
multiplyHigh() negateExact() nextAfter()  nextDown()  
nextUp()    pow()           random()    rint()  
round()     scalb()         signum()    sin()  
sinh()      sqrt()         subtractExact() tan()  
tanh()      toDegrees()    toIntExact() toRadians()  
ulp()
```

```
jshell> Math.
```

As you know, JShell auto-completion displays a list of everything that can appear to the right of the dot (.). Here we typed a class name and a dot (.), so JShell shows only the class's `static` members. The names that are not followed by any parentheses (`E` and `PI`) are the class's `static` variables. All the other names are the class's `static` methods:

- Any method names followed by `()`—only `random` in this case—do not require any arguments.
- Any method names followed by only an opening left parenthesis, `(`, require at least one argument or are overloaded.

You can easily view the value of the constants `PI` and `E`:

```
jshell> Math.PI  
$1 ==> 3.141592653589793  
  
jshell> Math.E  
$2 ==> 2.718281828459045  
  
jshell>
```

25.7.2 Viewing a Method's Parameters

Let's assume you wish to test `Math`'s `pow` method (introduced in Section 4.4.2), but you do not know the parameters it requires. You can type

```
Math.p
```

then press `Tab` to auto-complete the name `pow`:

```
jshell> Math.pow(
```

Since there are no other methods that begin with "pow", JShell also inserts the left parenthesis to indicate the beginning of a method call. Next, you can type *Tab* to view the method's parameters:

```
jshell> Math.pow<
double Math.pow(double a, double b)
<press tab again to see documentation>
jshell> Math.pow<
```

JShell displays the method's return type, name and complete parameter list followed by the next `jshell>` prompt containing what you've typed so far. As you can see, the method requires two `double` parameters.

25.7.3 Viewing a Method's Documentation

JShell integrates the Java API documentation so you can view documentation conveniently in JShell, rather than requiring you to use a separate web browser. Suppose you'd like to learn more about `pow` before completing your code snippet. You can press *Tab* again to view the method's Java documentation (known as its javadoc)—we cut out some of the documentation text and replaced it with a vertical ellipsis (...) to save space (try the steps in your own JShell session to see the complete text):

```
jshell> Math.pow<
double Math.pow(double a, double b)
>Returns the value of the first argument raised to the power of the
second argument. Special cases:
* If the second argument is positive or negative zero, then the
  result is 1.0.
...
<press tab again to see next page>
```

For long documentation, JShell displays part of it, then shows the message

```
<press tab again to see next page>
```

You can press *Tab* to view the next page of documentation. The next `jshell>` prompt shows the portion of the snippet you've typed so far:

```
jshell> Math.pow<
```

25.7.4 Viewing a `public` Field's Documentation

You can use the *Tab* feature to learn more about a class's `public` fields. For example, if you enter `Math.PI` followed by *Tab*, JShell displays

```
jshell> Math.PI
PI

Signatures:
Math.PI:double

<press tab again to see documentation>
```

which shows `Math.PI`'s type and indicates that you can use *Tab* again to view the documentation. Doing so displays:

```
jshell> Math.PI  
Math.PI:double  
The double value that is closer than any other to pi, the ratio of  
the circumference of a circle to its diameter.  
jshell> Math.PI
```

and the next `jshell>` prompt shows the portion of the snippet you've typed so far.

25.7.5 Viewing a Class's Documentation

You also can type a class name then *Tab* to view the class's fully qualified name. For example, typing `Math` then *Tab* shows:

```
jshell> Math  
Math          MathContext  
  
Signatures:  
java.lang.Math  
  
<press tab again to see documentation>  
jshell> Math
```

indicating that class `Math` is in the package `java.lang`. Typing *Tab* again shows the beginning of the class's documentation:

```
jshell> Math  
java.lang.Math  
The class Math contains methods for performing basic numeric opera-  
tions such as the elementary exponential, logarithm, square root,  
and trigonometric functions. Unlike some of the numeric methods of  
...  
  
<press tab again to see next page>
```

In this case, there is more documentation to view, so you can press *Tab* to view it. Whether or not you view the remaining documentation, the `jshell>` prompt shows the portion of the snippet you've typed so far:

```
jshell> Math
```

25.7.6 Viewing Method Overloads

Many classes have *overloaded* methods. When you press *Tab* to view an overloaded method's parameters, JShell displays the complete list of overloads, showing the parameters for every overload. For example, method `Math.abs` has four overloads:

```
jshell> Math.abs(  
$1  $2  
  
Signatures:  
int Math.abs(int a)  
long Math.abs(long a)  
float Math.abs(float a)  
double Math.abs(double a)  
  
<press tab again to see documentation>  
jshell> Math.abs(
```

When you press *Tab* again to view the documentation, JShell shows you the *first* overload's documentation:

```
jshell> Math.abs(
int Math.abs(int a)
Returns the absolute value of an int value. If the argument is not
negative, the argument is returned. If the argument is negative,
the negation of the argument is returned.
...
<press tab again to see next page>
```

You can then press *Tab* to view the documentation for the next overload in the list. Again, whether or not you view the remaining documentation, the `jshell>` prompt shows the portion of the snippet you've typed so far.

25.7.7 Exploring Members of a Specific Object

The exploration features shown in Sections 25.7.1–25.7.6 also apply to the members of a specific object. Let's create and explore a `String` object:

```
jshell> String dayName = "Monday"
dayName ==> "Monday"

jshell>
```

To view the methods you can call on the `dayName` object, type "`dayName.`" and press *Tab*:

```
jshell> dayName.
charAt()           chars()           codePointAt()
codePointBefore() codePointCount()   codePoints()
compareTo()       compareIgnoreCase() concat()
contains()        contentEquals()   endsWith()
equals()          equalsIgnoreCase() getBytes()
getChars()        getClass()       hashCode()
indexOf()         intern()         isEmpty()
lastIndexOf()    length()         matches()
notify()          notifyAll()      offsetByCodePoints()
regionMatches()   replace()        replaceAll()
replaceFirst()    split()          startsWith()
subSequence()    substring()     toCharArray()
toLowerCase()    toString()      toUpperCase()
trim()           wait()
```

Exploring `toUpperCase`

Let's investigate the `toUpperCase` method. Continue by typing "toU" and pressing *Tab* to auto-complete its name:

```
jshell> dayName.toUpperCase(
toUpperCase()

jshell> dayName.toUpperCase(
```

Then, type *Tab* to view its parameters:

```
jshell> dayName.toUpperCase()  
Signatures:  
String String.toUpperCase(Locale locale)  
String String.toUpperCase()  
<press tab again to see documentation>  
jshell> dayName.toUpperCase()
```

This method has two overloads. You can now use *Tab* to read about each overload, or simply choose the one you wish to use, by specifying the appropriate arguments (if any). In this case, we'll use the no-argument version to create a new `String` containing `MONDAY`, so we simply enter the closing right parenthesis of the method call and press *Enter*:

```
jshell> dayName.toUpperCase()  
$2 ==> "MONDAY"  
jshell>
```

Exploring substring

Let's assume you want to create the new `String` "DAY"—a subset of the implicit variable `$2`'s characters. For this purpose class `String` provides the overloaded method `substring`. First type "`$2.substring`" and press *Tab* to auto-complete its the method's name:

```
jshell> $2.substring()  
substring()  
jshell>
```

Next, use *Tab* to view the method's overloads:

```
jshell> $2.substring()  
Signatures:  
String String.substring(int beginIndex)  
String String.substring(int beginIndex, int endIndex)  
<press tab again to see documentation>  
jshell> $2.substring()
```

Next, use *Tab* again to view the first overload's documentation:

```
jshell> $2.substring()  
String String.substring(int beginIndex)  
Returns a string that is a substring of this string. The substring  
begins with the character at the specified index and extends to the  
end of this string.  
...  
<press tab again to see next page>
```

As you can see from the documentation, this overload of the method enables you to obtain a substring starting from a specific character index (that is, position) and continuing through the end of the `String`. The first character in the `String` is at index 0. This is the version of the method we wish to use to obtain "DAY" from "MONDAY", so we can return to our code snippet at the `jshell>` prompt:

```
jshell> $2.substring()
```

Finally, we can complete our call to `substring` and press *Enter* to view the results:

```
jshell> $2.substring(3)
$3 ==> "DAY"

jshell>
```

25.8 Declaring Methods

[*Note:* This section may be read after studying Chapter 5, Methods, and the preceding portions of Chapter 25.]

You can use JShell to prototype methods. For example, let's assume we'd like to write code that displays the cubes of the values from 1 through 10. For the purpose of this discussion, we're going to define two methods:

- Method `displayCubes` will iterate 10 times, calling method `cube` each time.
- Method `cube` will receive one int value and return the cube of that value.

25.8.1 Forward Referencing an Undeclared Method—Declaring Method `displayCubes`

Let's begin with method `displayCubes`. Start a new JShell session or `/reset` the current one, then enter the following method declaration:

```
void displayCubes() {
    for (int i = 1; i <= 10; i++) {
        System.out.println("Cube of " + i + " is " + cube(i));
    }
}
```

When you complete the method declaration, JShell displays:

```
| created method displayCubes(), however, it cannot be invoked
until method cube(int) is declared

jshell>
```

Again, we *manually* added the indentation. Note that after you type the method body's opening left brace, JShell displays continuation prompts (`...>`) before each subsequent line until you complete the method declaration by entering its closing right brace. Also, although JShell says "created method `displayCubes()`", it indicates that you cannot call this method until "`cube(int)` is declared". This is *not* fatal in JShell—it recognizes that `displayCubes` depends on an undeclared method (`cube`)—this is known as **forward referencing** an undeclared method. Once you define `cube`, you can call `displayCubes`.

25.8.2 Declaring a Previously Undeclared Method

Next, let's declare method `cube`, but *purposely make a logic error* by returning the square rather than the cube of its argument:

```
jshell> int cube(int x) {  
...>     return x * x;  
...> }  
| created method cube(int)  
  
jshell>
```

At this point, you can use JShell's `/methods` command to see the complete list of methods that are declared in the current JShell session:

```
jshell> /methods  
| void displayCubes()  
| int cube(int)  
  
jshell>
```

Note that JShell displays each method's return type to the right of the parameter list.

25.8.3 Testing cube and Replacing Its Declaration

Now that method `cube` is declared, let's test it with the argument 2:

```
jshell> cube(2)  
$3 ==> 4  
  
jshell>
```

The method correctly returns the 4 (that is, $2 * 2$), based on how the method is implemented. However, our the method's purpose was to calculate the cube of the argument, so the result should have been 8 ($2 * 2 * 2$). You can edit `cube`'s snippet to correct the problem. Because `cube` was declared as a multiline snippet, the easiest way to edit the declaration is using **JShell Edit Pad**. You could use `/list` to determine `cube`'s snippet ID then use `/edit` followed by the ID to open the snippet. You also edit the method by specifying its name, as in:

```
jshell> /edit cube
```

In the **JShell Edit Pad** window, change `cube`'s body to:

```
return x * x * x;
```

then press **Exit**. JShell displays:

```
jshell> /edit cube  
| modified method cube(int)  
  
jshell>
```

25.8.4 Testing Updated Method cube and Method displayCubes

Now that method `cube` is properly declared, let's test it again with the arguments 2 and 10:

```
jshell> cube(2)  
$5 ==> 8  
  
jshell> cube(10)  
$6 ==> 1000  
  
jshell>
```

The method properly returns the cubes of 2 (that is, 8) and 10 (that is, 1000), and stores the results in the implicit variables \$5 and \$6.

Now let's test `displayCubes`. If you type "di" and press *Tab*, JShell auto-completes the name, including the parentheses of the method call, because `displayCubes` receives no parameters. The following shows the results of the call:

```
jshell> displayCubes()
Cube of 1 is 1
Cube of 2 is 8
Cube of 3 is 27
Cube of 4 is 64
Cube of 5 is 125
Cube of 6 is 216
Cube of 7 is 343
Cube of 8 is 512
Cube of 9 is 729
Cube of 10 is 1000

jshell>
```

25.9 Exceptions

[*Note*: This section may be read after studying Chapter 6 and the preceding sections of Chapter 25.]

In Section 6.6, we introduced Java's exception-handling mechanism, showing how to catch an exception that occurred when we attempted to use an out-of-bounds array index. In JShell, catching exceptions is not required—it automatically catches each exception and displays information about it, then displays the next JShell prompt, so you can continue your session. This is particularly important for *checked exceptions* (Section 11.5) that are required to be caught in regular Java programs—as you know, catching an exception requires wrapping the code in a `try...catch` statement. By automatically, catching all exceptions, JShell makes it easier for you to *experiment* with methods that throw checked exceptions.

In the following new JShell session, we declare an array of `int` values, then demonstrate both valid and invalid array-access expressions:

```
jshell> int[] values = {10, 20, 30}
values ==> int[3] { 10, 20, 30 }

jshell> values[1]
$2 ==> 20

jshell> values[10]
|   java.lang.ArrayIndexOutOfBoundsException thrown: 10
|       at (#3:1)

jshell>
```

The snippet `values[10]` attempts to access an out-of-bounds element—recall that this results in an `ArrayIndexOutOfBoundsException`. Even though we did not wrap the code in a `try...catch`, JShell catches the exception and displays its `String` representation. This

includes the exception's type and an error message (in this case, the invalid index 10), followed by a so-called stack trace indicating where the problem occurred. The notation

```
| at (#3:1)
```

indicates that the exception occurred at line 1 of the code snippet with the ID 3. Section 5.6 discussed the method-call stack. A stack trace indicates the methods that were on the method-call stack at the time the exception occurred. A typical stack trace contains several "at" lines like the one shown here—one per stack frame. After displaying the stack trace, JShell shows the next `jshell>` prompt. Chapter 11 discusses stack traces in detail.

25.10 Importing Classes and Adding Packages to the CLASSPATH

[*Note:* This section may be read after studying Chapter 21, Custom Generic Data Structures and the preceding sections of Chapter 25.]

When working in JShell, you can import types from Java 9's packages. In fact, several packages are so commonly used by Java developers that JShell automatically imports them for you. (You can change this with JShell's `/set start` command—see Section 25.12.)

You can use JShell's `/imports` command to see the current session's list of `import` declarations. The following listing shows the packages that are auto-imported when you begin a new JShell session:

```
jshell> /imports
| import java.io.*
| import java.math.*
| import java.net.*
| import java.nio.file.*
| import java.util.*
| import java.util.concurrent.*
| import java.util.function.*
| import java.util.prefs.*
| import java.util.regex.*
| import java.util.stream.*
```

The `java.lang` package's contents are always available in JShell, just as in any Java source-code file.

In addition to the Java API's packages, you can import your own or third-party packages to use their types in JShell. First, you use JShell's `/env -c class-path` command to add the packages to JShell's CLASSPATH, which specifies where the additional packages are located. You can then use `import` declarations to experiment with the packages' contents in JShell.

Using Our Custom Generic List Class

In Chapter 21, we declared a custom generic `List` data structure and placed it in the package `com.deitel.datastructures`. Here, we'll add that package to JShell's CLASSPATH, import our `List` class, then use it in JShell. If you have a current JShell session open, use `/exit` to terminate it. Then, change directories to the `ch21` examples folder and start a new JShell session.

Adding the Location of a Package to the CLASSPATH

The ch21 folder contains a folder named com, which is the first of a nested set of folders that represent the compiled classes in our package com.deitel.datastructures. The following uses adds this package to the CLASSPATH:

```
jshell> /env -class-path .
| Setting new options and restoring state.

jshell>
```

The dot (.) indicates the current folder from which you launched JShell. You also can specify complete paths to other folders on your system or the paths of JAR (Java archive) files that contain packages of compiled classes.

Importing a Class from the Package

Now, you can import the List class for use in JShell. The following shows importing our List class and the complete list of imports in the current session:

```
jshell> import com.deitel.datastructures.List

jshell> /imports
| import java.io.*
| import java.math.*
| import java.net.*
| import java.nio.file.*
| import java.util.*
| import java.util.concurrent.*
| import java.util.function.*
| import java.util.prefs.*
| import java.util.regex.*
| import java.util.stream.*
| import com.deitel.datastructures.List

jshell>
```

Using the Imported Class

Finally, you can use class List. Below we create a List<String> and show that JShell's auto-complete capability can display the list of available methods. Then we insert two Strings into the List, displaying its contents after each insertAtFront operation:

```
jshell> List<String> list = new List<>()
list ==> com.deitel.datastructures.List@31610302

jshell> list.
equals(           getClass()          hashCode()          insertAtBack(
insertAtFront(   isEmpty()            notify()            notifyAll()
print()          removeFromBack()    removeFromFront()  toString()
wait(          

jshell> list.insertAtFront("red")

jshell> list.print()
The list is: red
```

```
jshell> list.insertAtFront("blue")
jshell> list.print()
The list is: blue red
jshell>
```

A Note Regarding imports

As you saw at the beginning of this section, JShell imports the entire `java.util` package—which contains the `List` interface (Section 16.6)—in every JShell session. The Java compiler gives precedence to an explicit type `import` for our class `List` like

```
import com.deitel.datastructures.List;
```

vs. an `import-on-demand` like

```
import java.util.*;
```

Had we used the following `import-on-demand`:

```
import com.deitel.datastructures.*;
```

then we would have had to refer to our `List` class by its fully qualified name (that is, `com.deitel.datastructures.List`) to differentiate it from `java.util.List`.

25.11 Using an External Editor

Section 25.3.10 demonstrated **JShell Edit Pad** for editing code snippets. This tool provides only simple editing functionality. Many programmers prefer to use more powerful text editors. Using JShell's `/set editor` command, you can specify your preferred text editor. For example, we have a text editor named `EditPlus`, located on our Windows system at

```
C:\Program Files>EditPlus\editplus.exe
```

The JShell command

```
jshell> /set editor C:\Program Files>EditPlus\editplus.exe
|   Editor set to: C:\Program Files>EditPlus\editplus.exe
jshell>
```

sets `EditPlus` as the snippet editor for the current JShell session. The `/set editor` command's argument is *operating-system specific*. For example, on Ubuntu Linux, you can use the built-in `gedit` text editor with the command

```
/set editor gedit
```

and on macOS,⁸ you can use the built-in `TextEdit` application with the command

```
/set editor -wait open -aTextEdit
```

Editing Snippets with a Custom Text Editor

When you're using a custom editor, each time you save snippet edits JShell immediately re-evaluates any snippets that have changed and shows their results (but not the snippets

8. On macOS, the `-wait` option is required so that JShell does not simply open the external editor, then return immediately to the next `jshell>` prompt.

themselves) in the JShell output. The following shows a new JShell session in which we set a custom editor, then performed JShell interactions—we explain momentarily the two lines of output that follow the `/edit` command:

```
jshell> /set editor C:\Program Files>EditPlus\editplus.exe
| Editor set to: C:\Program Files>EditPlus\editplus.exe

jshell> int x = 10
x ==> 10

jshell> int y = 10
y ==> 20

jshell> /edit
y ==> 20
10 + 20 = 30
jshell> /list

1 : int x = 10;
3 : int y = 20;
4 : System.out.print(x + " + " + y + " = " + (x + y))

jshell>
```

First we declared the `int` variables `x` and `y`, then we launched the external editor to edit our snippets. Initially, the editor shows the snippets that declare `x` and `y` (Fig. 25.4).

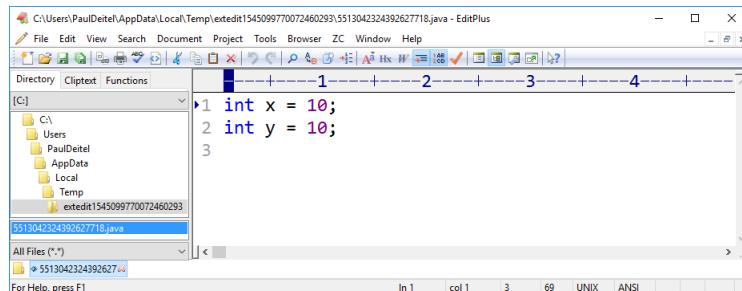


Fig. 25.4 | External editor showing code snippets to edit.

Next, we edited `y`'s declaration, giving it the new value 20, then we added a new snippet to display both values and their sum (Fig. 25.5).

When we saved the edits in our text editor, JShell replaced `y`'s original declaration with the updated one and showed

```
y ==> 20
```

to indicate that `y`'s value changed. Then, JShell executed the new `System.out.print` snippet and showed its results

```
10 + 20 = 30
```

Finally, when we closed the external editor and pressed *Enter* in the command window, JShell displayed the next `jshell>` prompt.

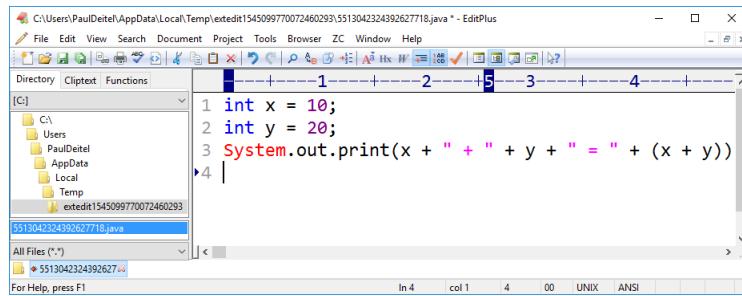


Fig. 25.5 | External editor showing code snippets to edit.

Retaining the Editor Setting

You can retain your editor setting for future JShell sessions as follows:

```
/set editor -retain commandToLaunchYourEditor
```

Restoring the JShell Edit Pad As the Default Editor

If you do not retain your custom editor, subsequent JShell sessions will use **JShell Edit Pad**. If you do retain the custom editor, you can restore **JShell Edit Pad** as the default with

```
/set editor -retain -default
```

25.12 Summary of JShell Commands

Figure 25.6 shows the basic JShell commands. Many of these commands have been presented throughout this chapter. Others are discussed in this section.

Command	Description
/help or /?	Displays JShell's list of commands.
/help intro	Displays a brief introduction to JShell.
/help shortcuts	Displays a description of several JShell shortcut keys.
/list	By default, lists the valid snippets you've entered in the current session. To list all snippets, use <code>/list -all</code> .
!*	Recalls and re-evaluates the last snippet.
/id	Recalls and re-evaluates the snippet with the specified <i>id</i> .
/-n	Recalls and re-evaluates a prior snippet—for <i>n</i> , 1 is the last snippet, 2 is the second-to-last, etc.
/edit	By default, opens a JShell Edit Pad window containing the valid snippets you've entered in the current session. See Section 25.11 to learn how to configure an external editor.
/save	Saves the current session's valid snippets to a specified file.

Fig. 25.6 | Jshell commands. (Part 1 of 2.)

Command	Description
/open	Opens a specified file of code snippets, loads the snippets into the current session and evaluates the loaded snippets.
/vars	Displays the current session's variables and their corresponding values.
/methods	Displays the signatures of the current session's declared methods.
/types	Displays types declared in the current session.
/imports	Displays the current session's import declarations.
/exit	Terminates the current JShell session.
/reset	Resets the current JShell session, deleting all code snippets.
/reload	Reloads a JShell session and executes the valid snippets (Section 25.12.3).
/drop	Deletes a specified snippet from the current session (Section 25.12.4).
/env	Makes changes to the JShell environment, such as adding packages or modules so you can use their types in JShell.
/history	Lists everything you've typed in the current JShell session, including all snippets (valid, invalid or overwritten) and JShell commands—the /list command shows only snippets, not JShell commands.
/set	Sets various JShell configuration options, such as the editor used in response to the /edit command, the text used for the JShell prompts, the imports to specify when a session starts, etc. (Sections 25.12.5–25.12.6).

Fig. 25.6 | Jshell commands. (Part 2 of 2.)

25.12.1 Getting Help in JShell

JShell's help documentation is incorporated directly via the `/help` or `/?` commands—`/?` is simply a shorthand for `/help`. For a quick introduction to JShell, type:

```
/help intro
```

To display JShell's list of commands, type

```
/help
```

For more information on a given command's options, type

```
/help command
```

For example

```
/help /list
```

displays the `/list` command's more detailed help documentation. Similarly

```
/help /set start
```

displays more detailed help documentation for the `/set` command's `start` option. For a list of the shortcut key combinations in JShell, type

```
/help shortcuts
```

25.12.2 /edit Command: Additional Features

We've discussed using `/edit` to load all valid snippets, a snippet with a specified ID or a method with a specified name into **JShell Edit Pad**. You can specify the identifier for any variable, method or type declaration that you'd like to edit. For example, if the current JShell session contains the declaration of a class named `Account`, the following loads that class into **JShell Edit Pad**:

```
/edit Account
```

25.12.3 /reload Command

At the time of this writing, you cannot use the `/id` command to execute a range of previous snippets. However, JShell's **/reload** command can re-execute all valid snippets in the current session. Consider the session from Sections 25.3.9–25.3.10:

```
jshell> /list

1 : 45
2 : 72
3 : if ($1 < $2) {
        System.out.printf("%d < %d%n", $1, $2);
    }
4 : if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }
5 : $1 = 100;
6 : if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }

jshell>
```

The following reloads that session one snippet at a time:

```
jshell> /reload
| Restarting and restoring state.
-: 45
-: 72
-: if ($1 < $2) {
        System.out.printf("%d < %d%n", $1, $2);
    }
45 < 72
-: if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }
-: $1 = 100
-: if ($1 > $2) {
        System.out.printf("%d > %d%n", $1, $2);
    }
100 > 72

jshell>
```

Each reloaded snippet is preceded by -: and in the case of the `if` statements, the output (if any) is shown immediately following each `if` statement. If you prefer not to see the snippets as they reload, you can use the `/reload` command's `-quiet` option:

```
jshell> /reload -quiet
| Restarting and restoring state.
45 < 72
100 > 72

jshell>
```

In this case, only the results of output statements are displayed. Then, you can view the snippets that were reloaded with the `/list` command.

25.12.4 /drop Command

You can eliminate a snippet from the current session with JShell's `/drop` command followed by a snippet ID or an identifier. The following new JShell session declares a variable `x` and a method `cube`, then drops `x` via its snippet ID and drops `cube` via its identifier:

```
jshell> int x = 10
x ==> 10

jshell> int cube(int y) {return y * y * y;}
| created method cube(int)

jshell> /list

1 : int x = 10;
2 : int cube(int y) {return y * y * y;}

jshell> /drop 1
| dropped variable x

jshell> /drop cube
| dropped method cube(int)

jshell> /list

jshell>
```

25.12.5 Feedback Modes

JShell has several feedback modes that determine what gets displayed after each interaction. To change the feedback mode, use JShell's `/set feedback` command:

```
/set feedback mode
```

where `mode` is `concise`, `normal` (the default), `silent` or `verbose`. All of the prior JShell interactions in this chapter used the `normal` mode.

Feedback Mode verbose

Below is a new JShell session in which we used `verbose` mode, which beginning programmers might prefer:

```
jshell> /set feedback verbose
| Feedback mode: verbose

jshell> int x = 10
x ==> 10
| created variable x : int

jshell> int cube(int y) {return y * y * y;}
| created method cube(int)

jshell> cube(x)
$3 ==> 1000
| created scratch variable $3 : int

jshell> x = 5
x ==> 5
| assigned to x : int

jshell> cube(x)
$5 ==> 125
| created scratch variable $5 : int

jshell>
```

Notice the additional feedback indicating that

- variable `x` was created,
- variable `$3` was created on the first call to `cube`—JShell refers to the implicit variable as a *scratch variable*,
- an `int` was assigned to the variable `x`, and
- scratch variable `$5` was created on the second call to `cube`.

Feedback Mode concise

Next, we `/reset` the session then set the feedback mode to `concise` and repeat the preceding session:

```
jshell> /reset
jshell> /set feedback concise
jshell> int x = 10
jshell> int cube(int y) {return y * y * y;}
jshell> cube(x)
$3 ==> 1000
jshell> x = 5
jshell> cube(x)
$5 ==> 125
jshell>
```

As you can see, the only feedback displayed is the result of each call to `cube`. If an error occurs, its feedback also will be displayed.

Feedback Mode silent

Next, we `/reset` the session then set the feedback mode to `silent` and repeat the preceding session:

```
jshell> /set feedback silent
-> int x = 10
-> int cube(int y) {return y * y * y;}
-> cube(x)
-> x = 5
-> cube(x)
-> /set feedback normal
| Feedback mode: normal

jshell>
```

In this case, the `jshell>` prompt becomes `->` and only error feedback will be displayed. You might use this mode if you've copied code from a Java source file and want to paste it into JShell, but do not want to see the feedback for each line.

25.12.6 Other JShell Features Configurable with `/set`

So far, we've demonstrated the `/set` command's capabilities for setting an external snippet editor and setting feedback modes. The `/set` command provides extensive capabilities for creating custom feedback modes via the commands:

- `/set mode`
- `/set prompt`
- `/set truncation`
- `/set format`

The `/set mode` command creates a user-defined custom feedback mode. Then you can use the other three commands to customize all aspects of JShell's feedback. The details of these commands are beyond the scope of this chapter. For more information, see JShell's help documentation for each of the preceding commands.

Customizing JShell Startup

Section 25.10 showed the set of common packages JShell imports at the start of each session. Using JShell's `/set start` command

```
/set start filename
```

you can provide a file of Java snippets and JShell commands that will be used in the current session when it restarts due to a `/reset` or `/reload` command. You can also remove all startup snippets with

```
/set start -none
```

or return to the default startup snippets with

```
/set start -default
```

In all three cases, the setting applies only to the current session unless you also include the `-retain` option. For example, the following command indicates that all subsequent JShell sessions should load the specified file of startup snippets and commands:

```
/set start -retain filename
```

You can restore the defaults for future sessions with

```
/set start -retain -default
```

25.13 Keyboard Shortcuts for Snippet Editing

In addition to the commands in Fig. 25.6, JShell supports many keyboard shortcuts for editing code, such as quickly jumping to the beginning or end of a line, or jumping between words in a line. JShell's command-line features are implemented by a library named JLine 2, which provides command-line editing and history capabilities. Figure 25.7 shows a sample of the shortcuts available.

Shortcut	Description
<i>Ctrl + a</i>	Move cursor to beginning of line.
<i>Ctrl + e</i>	Move cursor to end of line.
<i>Alt + b</i>	Move the cursor backwards by one word.
<i>Alt + f</i>	Move the cursor forwards by one word.
<i>Ctrl + r</i>	Perform a search for the last command or snippet containing the characters you type after typing <i>Ctrl + r</i> .
<i>Ctrl + t</i>	Reverse the two characters to the left of the cursor.
<i>Ctrl + k</i>	Cut everything from the cursor to the end of the line.
<i>Ctrl + u</i>	Cut everything from the beginning of the line up to, but not including the character at the cursor position.
<i>Ctrl + w</i>	Cut the word before the cursor.
<i>Alt + d</i>	Cut the word after the cursor.

Fig. 25.7 | Some keyboard shortcuts for editing the current snippet at the `jshell>` prompt.

25.14 How JShell Reinterprets Java for Interactive Use

In JShell:

- A `main` method is not required.
- Semicolons are not required on standalone statements.
- Variables do not need to be declared in classes or in methods.
- Methods do not need to be declared inside a class's body.
- Statements do not need to be written inside methods.
- Redeclaring a variable, method or type simply drops the prior declaration and replaces it with the new one, whereas the Java compiler normally would report an error.
- You do not need to catch exceptions, though you can if you need to test exception handling.
- JShell ignores top-level access modifiers (`public`, `private`, `protected`, `static`, `final`)—only `abstract` (Chapter 10) is allowed as a class modifier.
- The `synchronized` keyword (Chapter 23, Concurrency) is ignored.
- `package` statements and Java 9 `module` statements are not allowed.

25.15 IDE JShell Support

At the time of this writing, work is just beginning on JShell support in popular IDEs such as NetBeans, IntelliJ IDEA and Eclipse. NetBeans currently has an early access plug-in that enables you to work with JShell in both Java 8 and Java 9—even though JShell is a Java 9 feature. Some vendors will use JShell’s APIs to provide developers with JShell environments that show both the code users type and the results of running that code side-by-side. Some features you might see in IDE JShell support include:

- Source-code syntax coloring for better code readability.
- Automatic source-code indentation and insertion of closing braces {}, parentheses () and brackets [] to save programmers time.
- Debugger integration.
- Project integration, such as being able to automatically use classes in the same project from a JShell session.

25.16 Wrap-Up

In this chapter, you used JShell—Java 9’s new interactive REPL for exploration, discovery and experimentation. We showed how to start a JShell session and work with various types of code snippets, including statements, variables, expressions, methods and classes—all without having to declare a class containing a `main` method to execute the code.

You saw that you can list the valid snippets in the current session, and recall and execute prior snippets and commands using the up and down arrow keys. You also saw that you can list the current session’s variables, methods, types and `imports`. We showed how to clear the current JShell session to remove all existing snippets and how to save snippets to a file then reload them.

We demonstrated JShell’s auto-completion capabilities for code and commands, and showed how you can explore a class’s members and view documentation directly in JShell. We explored class `Math`, demonstrating how to list its `static` members, how to view a method’s parameters and overloads, view a method’s documentation and view a `public` field’s documentation. We also explored the methods of a `String` object.

You declared methods and forward referenced an undeclared method that you declared later in the session, then saw that you could go back and execute the first method. We also showed that you can replace a method declaration with a new method—in fact, you can replace any declaration of a variable, method or type.

We showed that JShell catches all exceptions and simply displays a stack trace followed by the next `jshell>` prompt, so you can continue the session. You imported an existing compiled class from a package, then used that class in a JShell session.

Next, we summarized and demonstrated various other JShell commands. We showed how to configure a custom snippet editor, view JShell’s help documentation, reload a session, drop snippets from a session, configure feedback modes and more. We listed some additional keyboard shortcuts for editing the current snippet at the `jshell>` prompt. Finally, we discussed how JShell reinterprets Java for interactive use and IDE support for JShell.

Self-Review Exercises

We encourage you to use JShell to do Exercises 25.1–25.43 after reading Sections 25.3–25.4. We've included the answers for all these exercises to help you get comfortable with JShell/REPL quickly.

25.1 Confirm that when you use `System.out.println` to display a `String` literal, such as "Happy Birthday!", the quotes ("") are not displayed. End your statement with a semicolon.

25.2 Repeat Exercise 25.1, but remove the semicolon at the end of your statement to demonstrate that semicolons in this position are optional in JShell.

25.3 Confirm that JShell does not execute a // end-of-line comment.

25.4 Show that an executable statement enclosed in a multiline comment—delimited by /* and */—does not execute.

25.5 Show what happens when the following code is entered in JShell:

```
/* incomplete multi-line comment
System.out.println("Welcome to Java Programming!")
/* complete multi-line
comment */
```

25.6 Show that indenting code with spaces does not affect statement execution.

25.7 Declare each of the following variables as type `int` in JShell to determine which are valid and which are invalid?

- a) first
- b) first number
- c) first1
- d) 1first

25.8 Show that braces do not have to occur in matching pairs inside a string literal.

25.9 Show what happens when you type each of the following code snippets into JShell:

- a) `System.out.println("seems OK")`
- b) `System.out.println("missing something?")`
- c) `System.out.println"missing something else?"`

25.10 Demonstrate that after a `System.out.print` the next print results appear on the same line right after the previous one's. [Hint: To demonstrate this, reset the current session, enter two `System.out.print` statements, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

25.11 Demonstrate that after a `System.out.println`, the next text that prints displays its text at the left of the next line. [Hint: To demonstrate this, reset the current session, enter a `System.out.println` statement followed by another print statement, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

25.12 Demonstrate that you can reset a JShell session to remove all prior snippets and start from scratch without having to exit JShell and start a new session.

- 25.13** Using `System.out.println`, demonstrate that the escape sequence `\n` causes a newline to be issued to the output. Use the string

```
"Welcome\n to\n JShell!"
```

- 25.14** Demonstrate that the escape sequence `\t` causes a tab to be issued to the output. Note that your output will depend on how tabs are set on your system. Use the string

```
"before\t after\n before\t after"
```

- 25.15** Demonstrate what happens when you include a single backslash (`\`) in a string. Be sure that the character after the backslash does not create a valid escape sequence.

- 25.16** Display a string containing `\\\\"` (recall that `\\"` is an escape sequence for a backslash). How many backslashes are displayed?

- 25.17** Use the escape sequence `\"` to display a quoted string.

- 25.18** What happens when the following code executes in JShell:

```
System.out.println("Happy Birthday!\rSunny")
```

- 25.19** Consider the following statement

```
System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")
```

Make the following intentional errors (separately) to see what happens.

- Omit the parentheses around the argument list.
- Omit the commas.
- Omit one of the `%s%n` sequences.
- Omit one of the strings (i.e., the second or the third argument).
- Replace the first `%s` with `%d`.
- Replace the string "Welcome to " with the integer 23.

- 25.20** What happens when you enter the `/imports` command in a new JShell session?

- 25.21** Import class `Scanner` then create a `Scanner` object `input` for reading from `System.in`. What happens when you execute the statement:

```
int number = input.nextInt()
```

and the user enters the string "hello"?

- 25.22** In a new or `/reset` JShell session, repeat Exercise 25.21 without importing class `Scanner` to demonstrate that the package `java.util` is already imported in JShell.

- 25.23** Demonstrate what happens when you don't precede a `Scanner` input operation with a meaningful prompting message telling the user what to input. Enter the following statements:

```
Scanner input = new Scanner(System.in)
int value = input.nextInt()
```

- 25.24** Demonstrate that you can't place an `import` statement in a class.

- 25.25** Demonstrate that identifiers are case sensitive by declaring variables `id` and `ID` of types `String` and `int`, respectively. Also use the `/list` command to show the two snippets representing the separate variables.

- 25.26** Demonstrate that initialization statements like

```
String month = "April"
int age = 65
```

indeed initialize their variables with the indicated values.

- 25.27** Demonstrate what happens when you:

- Add 1 to the largest possible `int` value 2,147,483,647.
- Subtract 1 from the smallest possible integer -2,147,483,648.

25.28 Demonstrate that large integers like 1234567890 are equivalent to their counterparts with the underscore separators, namely 1_234_567_890:

- 1234567890 == 1_234_567_890
- Print each of these values and show that you get the same result.
- Divide each of these values by 2 and show that you get the same result.

25.29 Placing spaces around operators in an arithmetic expression does not affect the value of that expression. In particular, the following expressions are equivalent:

17+23
17 + 23

Demonstrate this with an if statement using the condition

(17+23) == (17 + 23)

25.30 Demonstrate that the parentheses around the argument number1 + number2 in the following statement are unnecessary:

`System.out.printf("Sum is %d%n", (number1 + number2))`

25.31 Declare the int variable x and initialize it to 14, then demonstrate that the subsequent assignment x = 27 is destructive.

25.32 Demonstrate that printing the value of the following variable is non-destructive:

`int y = 29`

25.33 Using the declarations:

`int b = 7
int m = 9`

- Demonstrate that attempting to do algebraic multiplication by placing the variable names next to one another as in `bm` doesn't work in Java.
- Demonstrate that the Java expression `b * m` indeed multiplies the two operands.

25.34 Use the following expressions to demonstrate that integer division yields an integer result:

- `8 / 4`
- `7 / 5`

25.35 Demonstrate what happens when you attempt each of the following integer divisions:

- `0 / 1`
- `1 / 0`
- `0 / 0`

25.36 Demonstrate that the values of the following expressions:

- `(3 + 4 + 5) / 5`
- `3 + 4 + 5 / 5`

are different and thus the parentheses in the first expression are required if you want to divide the entire quantity $3 + 4 + 5$ by 5.

25.37 Calculate the value of the following expression:

`5 / 2 * 2 + 4 % 3 + 9 - 3`

manually being careful to observe the rules of operator precedence. Confirm the result in JShell.

25.38 Test each of the two equality and four relational operators on the two values 7 and 7. For example, `7 == 7`, `7 < 7`, etc.

25.39 Repeat Exercise 25.38 using the values 7 and 9.

25.40 Repeat Exercise 25.38 using the values 11 and 9.

- 25.41** Demonstrate that accidentally placing a semicolon after the right parenthesis of the condition in an if statement can be a logic error.

```
if (3 == 5); {
    System.out.println("3 is equal to 5");
}
```

- 25.42** Given the following declarations:

```
int x = 1
int y = 2
int z = 3
int a
```

what are the values of a, x, y and z after the following statement executes?

```
a = x = y = z = 10
```

- 25.43** Manually determine the value of the following expression then use JShell to check your work:

```
(3 * 9 * (3 + (9 * 3 / (3))))
```

Answers to Self-Review Exercises

25.1

```
jshell> System.out.println("Happy Birthday!");
Happy Birthday!
```

```
jshell>
```

25.2

```
jshell> System.out.println("Happy Birthday!")
Happy Birthday!
```

```
jshell>
```

25.3

```
jshell> // comments are not executable
```

```
jshell>
```

25.4

```
jshell> /* opening line of multi-line comment
...>     System.out.println("Welcome to Java Programming!")
...>     closing line of multi-line comment */
```

```
jshell>
```

- 25.5** There is no compilation error, because the second /* is considered to be part of the first multi-line comment.

```
jshell> /* incomplete multi-line comment
...>     System.out.println("Welcome to Java Programming!")
...>     /* complete multi-line
...>     comment */
```

```
jshell>
```

25.6

```
jshell> System.out.println("A")
A

jshell>     System.out.println("A") // indented 3 spaces
A

jshell>         System.out.println("A") // indented 6 spaces
A

jshell>
```

- 25.7** a) valid. b) invalid (space not allowed). c) valid. d) invalid (can't begin with a digit).

```
jshell> int first
first ==> 0

jshell> int first number
| Error:
| ';' expected
| int first number
|           ^
| 

jshell> int first1
first1 ==> 0

jshell> int 1first
| Error:
| '.class' expected
| int 1first
|       ^
| Error:
| not a statement
| int 1first
|       ^--^
| Error:
| unexpected type
|   required: value
|   found:   class
| int 1first
| ^--^
| Error:
| missing return statement
| int 1first
| > ^-----^

jshell>
```

25.8

```
jshell> "Unmatched brace { in a string is OK"
$1 ==> "Unmatched brace { in a string is OK"

jshell>
```

25.9

```
jshell> System.out.println("seems OK")
seems OK

jshell> System.out.println("missing something?")
| Error:
| unclosed string literal
| System.out.println("missing something?")
|           ^
| 

jshell> System.out.println"missing something else?"
| Error:
| ';' expected
| System.out.println"missing something else?"
|           ^
| Error:
| cannot find symbol
|   symbol:   variable println
| System.out.println"missing something else?"
| ^-----^
```

jshell>

25.10

```
jshell> System.out.print("Happy ")
Happy
jshell> System.out.print("Birthday")
Birthday
jshell> /save mysession

jshell> /open mysession
Happy Birthday
jshell>
```

25.11

```
jshell> System.out.println("Happy ")
Happy
jshell> System.out.println("Birthday")
Birthday
jshell> /save mysession

jshell> /open mysession
Happy
Birthday
jshell>
```

25.12

```
jshell> int x = 10
x ==> 10

jshell> int y = 20
y ==> 20
```

(continued...)

```
jshell> x + y  
$3 ==> 30  
  
jshell> /reset  
| Resetting state.  
  
jshell> /list  
  
jshell>
```

25.13

```
jshell> System.out.println("Welcome\n to\n JShell!")
Welcome
 to
JShell!
```

25.14

25.15

```
jshell> System.out.println("Bad escap\ e")
|  Error:
|  illegal escape character
|  System.out.println("Bad escap\ e")
|                                ^
jshell>
```

25.16 Two.

```
jshell> System.out.println("Displaying backslashes \\\\")  
Displaying backslashes \\  
  
jshell>
```

25.17

```
jshell> System.out.println("\"This is a string in quotes\"")  
"This is a string in quotes"  
  
jshell>
```

25.18

```
jshell> System.out.println("Happy Birthday!\rSunny")
Sunny Birthday!

jshell>
```

25.19 a)

```
jshell> System.out.printf("%s%n%s%n", "Welcome to ", "Java
Programming!")
| Error:
| ';' expected
| System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")
|           ^
| Error:
| cannot find symbol
|   symbol:   variable printf
| System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")
| ^-----^
```

jshell>

b)

```
jshell> System.out.printf("%s%n%s%n" "Welcome to " "Java
Programming!")
| Error:
| ')' expected
| System.out.printf("%s%n%s%n" "Welcome to " "Java Programming!")
|           ^
```

jshell>

c)

```
jshell> System.out.printf("%s%n", "Welcome to ", "Java Programming!")
Welcome to
$6 ==> java.io.PrintStream@6d4b1c02
```

jshell>

d)

```
jshell> System.out.printf("%s%n%s%n", "Welcome to ")
Welcome to
| java.util.MissingFormatArgumentException thrown: Format
specifier '%'
|     at Formatter.format (Formatter.java:2524)
|     at PrintStream.format (PrintStream.java:974)
|     at PrintStream.printf (PrintStream.java:870)
|     at (#7:1)
```

jshell>

e)

```
jshell> System.out.printf("%d%n%s%n", "Welcome to ", "Java
Programming!")
| java.util.IllegalFormatConversionException thrown: d !=
java.lang.String
|     at Formatter$FormatSpecifier.failConversion
(Formatter.java:4275)
|     at Formatter$FormatSpecifier.printInteger
(Formatter.java:2790)
|     at Formatter$FormatSpecifier.print (Formatter.java:2744)
(continued...)
```

```
|      at Formatter.format (Formatter.java:2525)
|      at PrintStream.format (PrintStream.java:974)
|      at PrintStream.printf (PrintStream.java:870)
|      at (#8:1)

jshell>
f)

jshell> System.out.printf("%s%n%s%n", 23, "Java Programming!")
23
Java Programming!
$9 ==> java.io.PrintStream@6d4b1c02

jshell>
```

25.20

```
jshell> /imports
|      import java.io.*
|      import java.math.*
|      import java.net.*
|      import java.nio.file.*
|      import java.util.*
|      import java.util.concurrent.*
|      import java.util.function.*
|      import java.util.prefs.*
|      import java.util.regex.*
|      import java.util.stream.*

jshell>
```

25.21

```
jshell> import java.util.Scanner

jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\] [infinity string=\Q\] ]]

jshell> int number = input.nextInt()
hello
|  java.util.InputMismatchException thrown:
|      at Scanner.throwFor (Scanner.java:860)
|      at Scanner.next (Scanner.java:1497)
|      at Scanner.nextInt (Scanner.java:2161)
|      at Scanner.nextInt (Scanner.java:2115)
|      at (#2:1)

jshell>
```

25.22

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\] [infinity string=\Q\] ]]
```

(continued...)

```
jshell> int number = input.nextInt()
hello
|  java.util.InputMismatchException thrown:
|      at Scanner.throwFor (Scanner.java:860)
|      at Scanner.next (Scanner.java:1497)
|      at Scanner.nextInt (Scanner.java:2161)
|      at Scanner.nextInt (Scanner.java:2115)
|      at (#2:1)

jshell>
```

25.23 JShell appears to hang while it waits for the user to type a value and press *Enter*.

25.24

```
jshell> class Demonstration {  
|     ...>     import java.util.Scanner;  
|     ...> }  
| Error:  
| illegal start of type  
|     import java.util.Scanner;  
|     ^  
| Error:  
| <identifier> expected  
|     import java.util.Scanner;  
|     ^  
|  
jshell> import java.util.Scanner  
  
jshell> class Demonstration {  
|     ...> }  
| created class Demonstration  
  
jshell>
```

25,25

```
jshell> /reset
| Resetting state.

jshell> String id = "Natasha"
id ==> "Natasha"

jshell> int ID = 413
ID ==> 413

jshell> /list
```

(continued...)

```
1 : String id = "Natasha";
2 : int ID = 413;
```

```
jshell>
```

25.26

```
jshell> String month = "April"
month ==> "April"

jshell> System.out.println(month)
April

jshell> int age = 65
age ==> 65

jshell> System.out.println(age)
65

jshell>
```

25.27

```
jshell> 2147483647 + 1
$9 ==> -2147483648

jshell> -2147483648 - 1
$10 ==> 2147483647

jshell>
```

25.28

```
jshell> 1234567890 == 1_234_567_890
$4 ==> true

jshell> System.out.println(1234567890)
1234567890

jshell> System.out.println(1_234_567_890)
1234567890

jshell> 1234567890 / 2
$5 ==> 617283945

jshell> 1_234_567_890 / 2
$6 ==> 617283945

jshell>
```

25.29

```
jshell> (17+23) == (17 + 23)
$7 ==> true

jshell>
```

25.30

```
jshell> int number1 = 10
number1 ==> 10

jshell> int number2 = 20
number2 ==> 20

jshell> System.out.printf("Sum is %d%n", (number1 + number2))
Sum is 30
$10 ==> java.io.PrintStream@1794d431

jshell> System.out.printf("Sum is %d%n", number1 + number2)
Sum is 30
$11 ==> java.io.PrintStream@1794d431

jshell>
```

25.31

```
jshell> int x = 14
x ==> 14

jshell> x = 27
x ==> 27

jshell>
```

25.32

```
jshell> int y = 29
y ==> 29

jshell> System.out.println(y)
29

jshell> y
y ==> 29
```

25.33

```
jshell> int b = 7
b ==> 7

jshell> int m = 9
m ==> 9

jshell> bm
| Error:
| cannot find symbol
|   symbol:   variable bm
|   bm
|   ^
|   ^

jshell> b * m
$3 ==> 63

jshell>
```

25.34 a) 2. b) 1.

```
jshell> 8 / 4  
$4 ==> 2  
  
jshell> 7 / 5  
$5 ==> 1  
  
jshell>
```

25.35

```
jshell> 0 / 1  
$6 ==> 0  
  
jshell> 1 / 0  
|  java.lang.ArithmException thrown: / by zero  
|      at (#7:1)  
  
jshell> 0 / 0  
|  java.lang.ArithmException thrown: / by zero  
|      at (#8:1)  
  
jshell>
```

25.36

```
jshell> (3 + 4 + 5) / 5  
$9 ==> 2  
  
jshell> 3 + 4 + 5 / 5  
$10 ==> 8  
  
jshell>
```

25.37

```
jshell> 5 / 2 * 2 + 4 % 3 + 9 - 3  
$11 ==> 11  
  
jshell>
```

25.38

```
jshell> 7 == 7  
$12 ==> true  
  
jshell> 7 != 7  
$13 ==> false  
  
jshell> 7 < 7  
$14 ==> false  
  
jshell> 7 <= 7  
$15 ==> true  
  
jshell> 7 > 7  
$16 ==> false
```

(continued...)

```
jshell> 7 >= 7
$17 ==> true

jshell>
```

25.39

```
jshell> 7 == 9
$18 ==> false

jshell> 7 != 9
$19 ==> true

jshell> 7 < 9
$20 ==> true

jshell> 7 <= 9
$21 ==> true

jshell> 7 > 9
$22 ==> false

jshell> 7 >= 9
$23 ==> false

jshell>
```

25.40

```
jshell> 11 == 9
$24 ==> false

jshell> 11 != 9
$25 ==> true

jshell> 11 < 9
$26 ==> false

jshell> 11 <= 9
$27 ==> false

jshell> 11 > 9
$28 ==> true

jshell> 11 >= 9
$29 ==> true

jshell>
```

25.41

```
jshell> if (3 == 5); {
    ...>     System.out.println("3 is equal to 5");
    ...> }
3 is equal to 5

jshell>
```

25.42

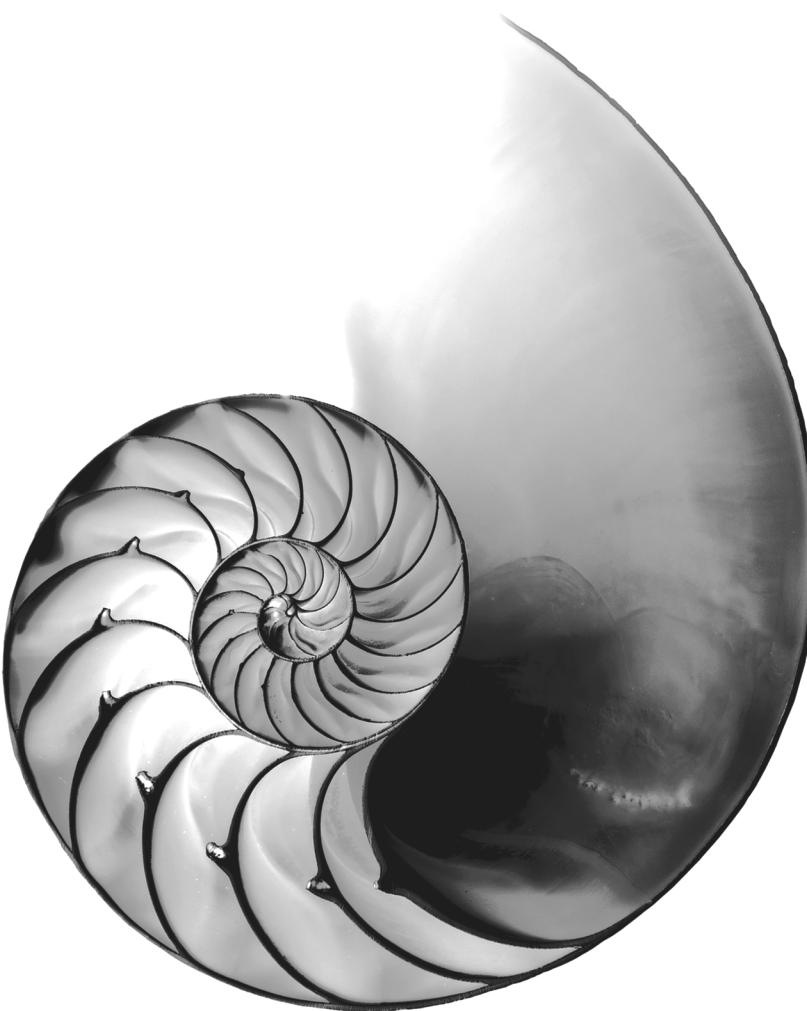
```
jshell> int x = 1  
x ==> 1  
  
jshell> int y = 2  
y ==> 2  
  
jshell> int z = 3  
z ==> 3  
  
jshell> int a  
a ==> 0  
  
jshell> a = x = y = z = 10  
a ==> 10  
  
jshell> x  
x ==> 10  
  
jshell> y  
y ==> 10  
  
jshell> z  
z ==> 10  
  
jshell>
```

25.43

```
jshell> (3 * 9 * (3 + (9 * 3 / (3))))  
$42 ==> 324  
  
jshell>
```

Swing GUI Components: Part I

26



Objectives

In this chapter you'll:

- Learn how to use the Nimbus look-and-feel.
- Build GUIs and handle events generated by user interactions with GUIs.
- Understand the packages containing GUI components, event-handling classes and interfaces.
- Create and manipulate buttons, labels, lists, text fields and panels.
- Handle mouse events and keyboard events.
- Use layout managers to arrange GUI components.

Outline

- 26.1** Introduction
- 26.2** Java's Nimbus Look-and-Feel
- 26.3** Simple GUI-Based Input/Output with JOptionPane
- 26.4** Overview of Swing Components
- 26.5** Displaying Text and Images in a Window
- 26.6** Text Fields and an Introduction to Event Handling with Nested Classes
- 26.7** Common GUI Event Types and Listener Interfaces
- 26.8** How Event Handling Works
- 26.9** JButton
- 26.10** Buttons That Maintain State
 - 26.10.1 JCheckBox
 - 26.10.2 JRadioButton
- 26.11** JComboBox; Using an Anonymous Inner Class for Event Handling
- 26.12** JList
- 26.13** Multiple-Selection Lists
- 26.14** Mouse Event Handling
- 26.15** Adapter Classes
- 26.16** JPanel Subclass for Drawing with the Mouse
- 26.17** Key Event Handling
- 26.18** Introduction to Layout Managers
 - 26.18.1 FlowLayout
 - 26.18.2 BorderLayout
 - 26.18.3 GridLayout
- 26.19** Using Panels to Manage More Complex Layouts
- 26.20** JTextArea
- 26.21** Wrap-Up

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#) | [Making a Difference](#)

26.1 Introduction

[Note: JavaFX (Chapters 12, 13 and 22) is Java's GUI, graphics and multimedia API of the future. We provide this chapter and Chapters 27 and 35 *as is* from this book's Tenth Edition for those still interested in Swing GUIs.]

A **graphical user interface (GUI)** presents a user-friendly mechanism for interacting with an application. A GUI (pronounced “GOO-ee”) gives an application a distinctive “look-and-feel.” GUIs are built from **GUI components**. These are sometimes called *controls* or *widgets*—short for window gadgets. A GUI component is an object with which the user *interacts* via the mouse, the keyboard or another form of input, such as voice recognition. In this chapter and Chapter 35, Swing GUI Components: Part 2, you’ll learn about many of Java’s so-called **Swing GUI components** from the **javax.swing** package. We cover other GUI components as they’re needed throughout the book. In Chapter 12 and two online chapters, you’ll learn about JavaFX—Java’s latest APIs for GUI, graphics and multimedia.



Look-and-Feel Observation 26.1

Providing different applications with consistent, intuitive user-interface components gives users a sense of familiarity with a new application, so that they can learn it more quickly and use it more productively.

IDE Support for GUI Design

Many IDEs provide GUI design tools with which you can specify a component’s *size*, *location* and other attributes in a visual manner by using the mouse, the keyboard and drag-and-drop. The IDEs generate the GUI code for you. This greatly simplifies creating GUIs,

but each IDE generates this code differently. For this reason, we wrote the GUI code by hand, as you'll see in the source-code files for this chapter's examples. We encourage you to build each GUI visually using your preferred IDE(s).

Sample GUI: The SwingSet3 Demo Application

As an example of a GUI, consider Fig. 26.1, which shows the **SwingSet3** demo application from the JDK demos and samples download at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. This application is a nice way for you to browse through the various GUI components provided by Java's Swing GUI APIs. Simply click a component name (e.g., `JFrame`, `JTabbedPane`, etc.) in the **GUI Components** area at the left of the window to see a demonstration of the GUI component in the right side of the window. The source code for each demo is shown in the text area at the bottom of the window. We've labeled a few of the GUI components in the application. At the top of the window is a **title bar** that contains the window's title. Below that is a **menu bar** containing **menus** (`File` and `View`). In the top-right region of the window is a set of **buttons**—typically, users click buttons to perform tasks. In the **GUI Components** area of the window is a **combo box**; the user can click the down arrow at the right side of the box to select from a list of items. The menus, buttons and combo box are part of the application's GUI. They enable you to interact with the application.

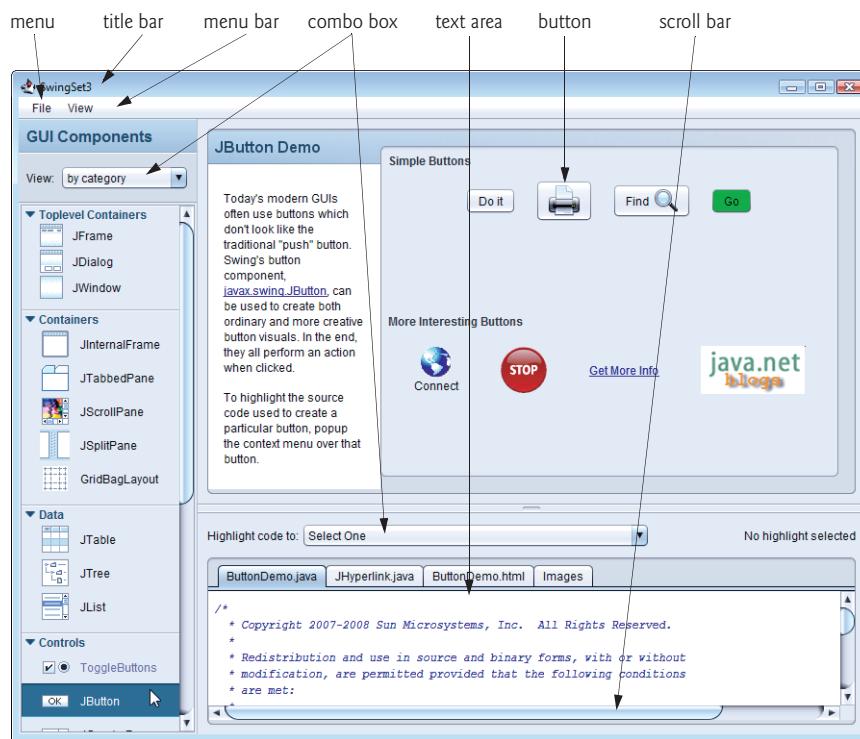


Fig. 26.1 | SwingSet3 application demonstrates many of Java's Swing GUI components.

26.2 Java's Nimbus Look-and-Feel

A GUI's look consists of its visual aspects, such as its colors and fonts, and its feel consists of the components you use to interact with the GUI, such as buttons and menus. Together these are known as the GUI's look-and-feel. Swing has a cross-platform look-and-feel known as **Nimbus**. For GUI screen captures like Fig. 26.1, we've configured our systems to use Nimbus as the default look-and-feel. There are three ways that you can use Nimbus:

1. Set it as the default for all Java applications that run on your computer.
2. Set it as the look-and-feel at the time that you launch an application by passing a command-line argument to the `java` command.
3. Set it as the look-and-feel programatically in your application (see Section 35.6).

To set Nimbus as the default for all Java applications, you must create a text file named `swing.properties` in the `lib` folder of both your JDK installation folder and your JRE installation folder. Place the following line of code in the file:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

In addition to the standalone JRE, there is a JRE nested in your JDK's installation folder. If you're using an IDE that depends on the JDK, you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.

If you prefer to select Nimbus on an application-by-application basis, place the following command-line argument after the `java` command and before the application's name when you run the application:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

26.3 Simple GUI-Based Input/Output with JOptionPane

The applications in Chapters 2–10 display text in the command window and obtain input from the command window. Most applications you use on a daily basis use windows or **dialog boxes** (also called **dialogs**) to interact with the user. For example, an e-mail program allows you to type and read messages in a window the program provides. Dialog boxes are windows in which programs display important messages to the user or obtain information from the user. Java's **JOptionPane** class (package `javax.swing`) provides prebuilt dialog boxes for both input and output. These are displayed by invoking static `JOptionPane` methods. Figure 26.2 presents a simple addition application that uses two **input dialogs** to obtain integers from the user and a **message dialog** to display the sum of the integers the user enters.

```

1 // Fig. 26.2: Addition.java
2 // Addition program that uses JOptionPane for input and output.
3 import javax.swing.JOptionPane;
4
5 public class Addition
6 {

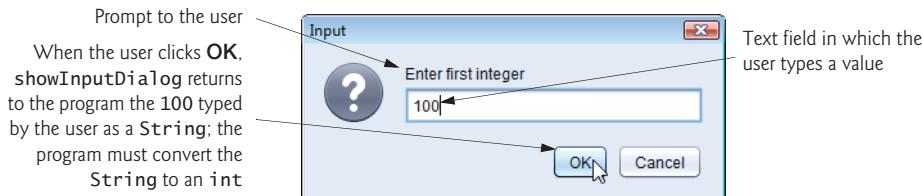
```

Fig. 26.2 | Addition program that uses `JOptionPane` for input and output. (Part I of 2.)

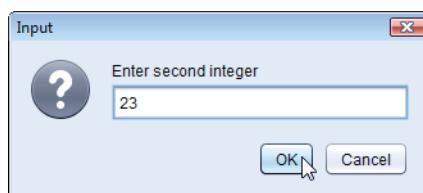
```

7  public static void main(String[] args)
8  {
9      // obtain user input from JOptionPane input dialogs
10     String firstNumber =
11         JOptionPane.showInputDialog("Enter first integer");
12     String secondNumber =
13         JOptionPane.showInputDialog("Enter second integer");
14
15     // convert String inputs to int values for use in a calculation
16     int number1 = Integer.parseInt(firstNumber);
17     int number2 = Integer.parseInt(secondNumber);
18
19     int sum = number1 + number2;
20
21     // display result in a JOptionPane message dialog
22     JOptionPane.showMessageDialog(null, "The sum is " + sum,
23         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE);
24 }
25 }
```

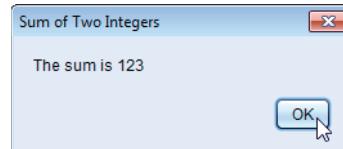
(a) Input dialog displayed by lines 10–11



(b) Input dialog displayed by lines 12–13



(c) Message dialog displayed by lines 22–23—When the user clicks OK, the message dialog is dismissed (removed from the screen)

**Fig. 26.2** | Addition program that uses JOptionPane for input and output. (Part 2 of 2.)

Input Dialogs

Line 3 imports class JOptionPane. Lines 10–11 declare the local String variable `firstNumber` and assign it the result of the call to JOptionPane static method `showInputDialog`. This method displays an input dialog (see the screen capture in Fig. 26.2(a)), using the method's String argument ("Enter first integer") as a prompt.



Look-and-Feel Observation 26.2

The prompt in an input dialog typically uses *sentence-style capitalization*—a style that capitalizes only the first letter of the first word in the text unless the word is a proper noun (for example, *Jones*).

The user types characters in the text field, then clicks **OK** or presses the *Enter* key to submit the **String** to the program. Clicking **OK** also **dismisses (hides) the dialog**. [Note: If you type in the text field and nothing appears, activate the text field by clicking it with the mouse.] Unlike **Scanner**, which can be used to input values of *several* types from the user at the keyboard, *an input dialog can input only Strings*. This is typical of most GUI components. The user can type *any* characters in the input dialog's text field. Our program assumes that the user enters a *valid* integer. If the user clicks **Cancel**, **showInputDialog** returns **null**. If the user either types a noninteger value or clicks the **Cancel** button in the input dialog, an exception will occur and the program will not operate correctly. Lines 12–13 display another input dialog that prompts the user to enter the second integer. Each **JOptionPane** dialog that you display is a so called **modal dialog**—while the dialog is on the screen, the user *cannot* interact with the rest of the application.



Look-and-Feel Observation 26.3

Do not overuse modal dialogs, as they can reduce the usability of your applications. Use a modal dialog only when it's necessary to prevent users from interacting with the rest of an application until they dismiss the dialog.

Converting **Strings** to **int** Values

To perform the calculation, we convert the **Strings** that the user entered to **int** values. Recall that the **Integer** class's **static** method **parseInt** converts its **String** argument to an **int** value and might throw a **NumberFormatException**. Lines 16–17 assign the converted values to local variables **number1** and **number2**, and line 19 sums these values.

Message Dialogs

Lines 22–23 use **JOptionPane** **static** method **showMessageDialog** to display a message dialog (the last screen of Fig. 26.2) containing the sum. The first argument helps the Java application determine where to *position* the dialog box. A dialog is typically displayed from a GUI application with its own window. The first argument refers to that window (known as the *parent window*) and causes the dialog to appear centered over the parent (as we'll do in Section 26.9). If the first argument is **null**, the dialog box is displayed at the *center* of your screen. The second argument is the *message* to display—in this case, the result of concatenating the **String** "The sum is " and the value of **sum**. The third argument—"Sum of Two Integers"—is the **String** that should appear in the *title bar* at the top of the dialog. The fourth argument—**JOptionPane.PLAIN_MESSAGE**—is the *type of message dialog to display*. A **PLAIN_MESSAGE** dialog does *not* display an *icon* to the left of the message. Class **JOptionPane** provides several overloaded versions of methods **showInputDialog** and **showMessageDialog**, as well as methods that display other dialog types. For complete information, visit <http://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html>.



Look-and-Feel Observation 26.4

*The title bar of a window typically uses **book-title capitalization**—a style that capitalizes the first letter of each significant word in the text and does not end with any punctuation (for example, Capitalization in a Book Title).*

JOptionPane Message Dialog Constants

The constants that represent the message dialog types are shown in Fig. 26.3. All message dialog types except PLAIN_MESSAGE display an icon to the *left* of the message. These icons provide a visual indication of the message's importance to the user. A QUESTION_MESSAGE icon is the *default icon* for an input dialog box (see Fig. 26.2).

Message dialog type	Icon	Description
ERROR_MESSAGE		Indicates an error.
INFORMATION_MESSAGE		Indicates an informational message.
WARNING_MESSAGE		Warns of a potential problem.
QUESTION_MESSAGE		Poses a question. This dialog normally requires a response, such as clicking a Yes or a No button.
PLAIN_MESSAGE	no icon	A dialog that contains a message, but no icon.

Fig. 26.3 | JOptionPane static constants for message dialogs.

26.4 Overview of Swing Components

Though it's possible to perform input and output using the JOptionPane dialogs, most GUI applications require more elaborate user interfaces. The remainder of this chapter discusses many GUI components that enable application developers to create robust GUIs. Figure 26.4 lists several basic Swing GUI components that we discuss.

Component	Description
JLabel	Displays <i>uneditable text</i> and/or icons.
JTextField	Typically receives <i>input</i> from the user.
JButton	Triggers an event when clicked with the mouse.
JCheckBox	Specifies an option that can be <i>selected</i> or <i>not selected</i> .
JComboBox	A <i>drop-down list of items</i> from which the <i>user</i> can make a <i>selection</i> .
JList	A <i>list of items</i> from which the <i>user</i> can make a <i>selection</i> by <i>clicking</i> on <i>any one</i> of them. <i>Multiple elements</i> can be selected.
JPanel	An area in which <i>components</i> can be <i>placed</i> and <i>organized</i> .

Fig. 26.4 | Some basic Swing GUI components.

Swing vs. AWT

There are actually *two* sets of Java GUI components. In Java's early days, GUIs were built with components from the **Abstract Window Toolkit (AWT)** in package `java.awt`.

These look like the native GUI components of the platform on which a Java program executes. For example, a `Button` object displayed in a Java program running on Microsoft Windows looks like those in other *Windows* applications. On Apple macOS, the `Button` looks like those in other *Mac* applications. Sometimes, even the manner in which a user can interact with an AWT component *differs between platforms*. The component's appearance and the way in which the user interacts with it are known as its **look-and-feel**.



Look-and-Feel Observation 26.5

Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel.

Lightweight vs. Heavyweight GUI Components

Most Swing components are **lightweight components**—they're written, manipulated and displayed completely in Java. AWT components are **heavyweight components**, because they rely on the local platform's **windowing system** to determine their functionality and their look-and-feel. Several Swing components are *heavyweight* components.

Superclasses of Swing's Lightweight GUI Components

The UML class diagram of Fig. 26.5 shows an *inheritance hierarchy* of classes from which lightweight Swing components inherit their common attributes and behaviors.

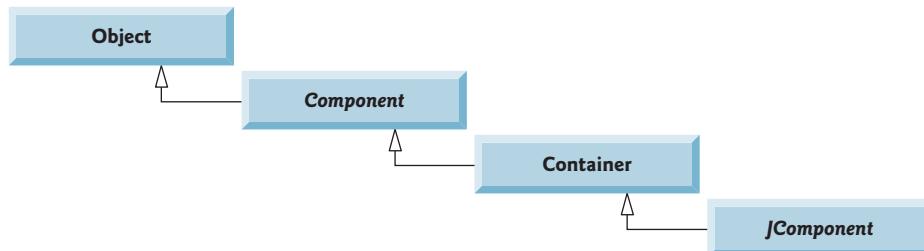


Fig. 26.5 | Common superclasses of the lightweight Swing components.

Class **Component** (package `java.awt`) is a superclass that declares the common features of GUI components in packages `java.awt` and `javax.swing`. Any object that is a **Container** (package `java.awt`) can be used to organize **Components** by *attaching* the **Components** to the **Container**. **Containers** can be placed in other **Containers** to organize a GUI.

Class **JComponent** (package `javax.swing`) is a subclass of **Container**. **JComponent** is the superclass of all *lightweight* Swing components and declares their common attributes and behaviors. Because **JComponent** is a subclass of **Container**, all lightweight Swing components are also **Containers**. Some common features supported by **JComponent** include:

1. A **pluggable look-and-feel** for *customizing* the appearance of components (e.g., for use on particular platforms). You'll see an example of this in Section 35.6.
2. Shortcut keys (called **mnemonics**) for direct access to GUI components through the keyboard. You'll see an example of this in Section 35.4.

3. Brief descriptions of a GUI component's purpose (called **tool tips**) that are displayed when the *mouse cursor is positioned over the component* for a short time. You'll see an example of this in the next section.
4. Support for *accessibility*, such as braille screen readers for the visually impaired.
5. Support for user-interface **localization**—that is, customizing the user interface to display in different languages and use local cultural conventions.

26.5 Displaying Text and Images in a Window

Our next example introduces a framework for building GUI applications. Several concepts in this framework will appear in many of our GUI applications. This is our first example in which the application appears in its own window. Most windows you'll create that can contain Swing GUI components are instances of class `JFrame` or a subclass of `JFrame`. `JFrame` is an *indirect* subclass of class `java.awt.Window` that provides the basic attributes and behaviors of a window—a *title bar* at the top, and *buttons* to *minimize*, *maximize* and *close* the window. Since an application's GUI is typically specific to the application, most of our examples will consist of *two* classes—a subclass of `JFrame` that helps us demonstrate new GUI concepts and an application class in which `main` creates and displays the application's primary window.

Labeling GUI Components

A typical GUI consists of many components. GUI designers often provide text stating the purpose of each. Such text is known as a **label** and is created with a `JLabel`—a subclass of `JComponent`. A `JLabel` displays read-only text, an image, or both text and an image. Applications rarely change a label's contents after creating it.



Look-and-Feel Observation 26.6

Text in a `JLabel` normally uses sentence-style capitalization.

The application of Figs. 26.6–26.7 demonstrates several `JLabel` features and presents the framework we use in most of our GUI examples. We did not highlight the code in this example, since most of it is new. [Note: There are many more features for each GUI component than we can cover in our examples. To learn the complete details of each GUI component, visit its page in the online documentation. For class `JLabel`, visit <http://docs.oracle.com/javase/8/docs/api/javax/swing/JLabel.html>.]

```
1 // Fig. 26.6: LabelFrame.java
2 // JLabels with text and icons.
3 import java.awt.FlowLayout; // specifies how components are arranged
4 import javax.swing.JFrame; // provides basic window features
5 import javax.swing.JLabel; // displays text and images
6 import javax.swing.SwingConstants; // common constants used with Swing
7 import javax.swing.Icon; // interface used to manipulate images
8 import javax.swing.ImageIcon; // loads images
9
```

Fig. 26.6 | `JLabels` with text and icons. (Part I of 2.)

```
10 public class LabelFrame extends JFrame
11 {
12     private final JLabel label1; // JLabel with just text
13     private final JLabel label2; // JLabel constructed with text and icon
14     private final JLabel label3; // JLabel with added text and icon
15
16     // LabelFrame constructor adds JLabels to JFrame
17     public LabelFrame()
18     {
19         super("Testing JLabel");
20         setLayout(new FlowLayout()); // set frame layout
21
22         // JLabel constructor with a string argument
23         label1 = new JLabel("Label with text");
24         label1.setToolTipText("This is label1");
25         add(label1); // add label1 to JFrame
26
27         // JLabel constructor with string, Icon and alignment arguments
28         Icon bug = new ImageIcon(getClass().getResource("bug1.png"));
29         label2 = new JLabel("Label with text and icon", bug,
30             SwingConstants.LEFT);
31         label2.setToolTipText("This is label2");
32         add(label2); // add label2 to JFrame
33
34         label3 = new JLabel(); // JLabel constructor no arguments
35         label3.setText("Label with icon and text at bottom");
36         label3.setIcon(bug); // add icon to JLabel
37         label3.setHorizontalTextPosition(SwingConstants.CENTER);
38         label3.setVerticalTextPosition(SwingConstants.BOTTOM);
39         label3.setToolTipText("This is label3");
40         add(label3); // add label3 to JFrame
41     }
42 }
```

Fig. 26.6 | JLabels with text and icons. (Part 2 of 2.)

```
1 // Fig. 26.7: LabelTest.java
2 // Testing LabelFrame.
3 import javax.swing.JFrame;
4
5 public class LabelTest
6 {
7     public static void main(String[] args)
8     {
9         LabelFrame labelFrame = new LabelFrame();
10        labelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        labelFrame.setSize(260, 180);
12        labelFrame.setVisible(true);
13    }
14 }
```

Fig. 26.7 | Testing LabelFrame. (Part 1 of 2.)

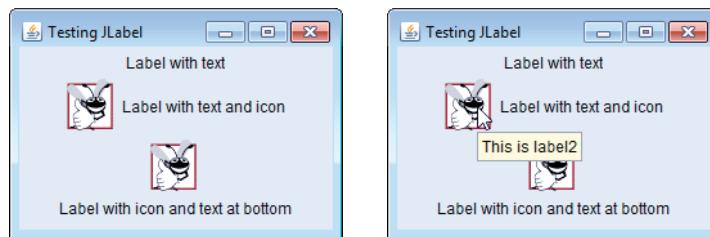


Fig. 26.7 | Testing `JLabelFrame`. (Part 2 of 2.)

Class `JLabelFrame` (Fig. 26.6) extends `JFrame` to inherit the features of a window. We'll use an instance of class `JLabelFrame` to display a window containing three `JLabel`s. Lines 12–14 declare the three `JLabel` instance variables that are instantiated in the `JLabelFrame` constructor (lines 17–41). Typically, the `JFrame` subclass's constructor builds the GUI that's displayed in the window when the application executes. Line 19 invokes superclass `JFrame`'s constructor with the argument "Testing `JLabel`". `JFrame`'s constructor uses this `String` as the text in the window's title bar.

Specifying the Layout

When building a GUI, you must attach each GUI component to a container, such as a window created with a `JFrame`. Also, you typically must decide where to *position* each GUI component—known as *specifying the layout*. Java provides several **layout managers** that can help you position components, as you'll learn later in this chapter and in Chapter 35.

Many IDEs provide GUI design tools in which you can specify components' exact *sizes* and *locations* in a visual manner by using the mouse; then the IDE will generate the GUI code for you. Such IDEs can greatly simplify GUI creation.

To ensure that our GUIs can be used with *any* IDE, we did *not* use an IDE to create the GUI code. We use Java's layout managers to *size* and *position* components. With the **FlowLayout** layout manager, components are placed in a *container* from left to right in the order in which they're added. When no more components can fit on the current line, they continue to display left to right on the next line. If the container is *resized*, a `FlowLayout` *reflows* the components, possibly with fewer or more rows based on the new container width. Every container has a *default layout*, which we're changing for `JLabelFrame` to a `FlowLayout` (line 20). Method `setLayout` is inherited into class `JLabelFrame` indirectly from class `Container`. The argument to the method must be an object of a class that implements the `LayoutManager` interface (e.g., `FlowLayout`). Line 20 creates a new `FlowLayout` object and passes its reference as the argument to `setLayout`.

Creating and Attaching Label1

Now that we've specified the window's layout, we can begin creating and attaching GUI components to the window. Line 23 creates a `JLabel` object and passes "Label with text" to the constructor. The `JLabel` displays this text on the screen. Line 24 uses method `setToolTipText` (inherited by `JLabel` from `JComponent`) to specify the tool tip that's displayed when the user positions the mouse cursor over the `JLabel` in the GUI. You can see a sample tool tip in the second screen capture of Fig. 26.7. When you execute this appli-

cation, hover the mouse pointer over each `JLabel` to see its tool tip. Line 25 (Fig. 26.6) attaches `label11` to the `LabelFrame` by passing `label11` to the `add` method, which is inherited indirectly from class `Container`.



Common Programming Error 26.1

If you do not explicitly add a GUI component to a container, the GUI component will not be displayed when the container appears on the screen.



Look-and-Feel Observation 26.7

Use tool tips to add descriptive text to your GUI components. This text helps the user determine the GUI component's purpose in the user interface.

The Icon Interface and Class `ImageIcon`

Icons are a popular way to enhance the look-and-feel of an application and are also commonly used to indicate functionality. For example, the same icon is used to play most of today's media on devices like DVD players and MP3 players. Several Swing components can display images. An icon is normally specified with an `Icon` (package `javax.swing`) argument to a constructor or to the component's `setIcon` method. Class `ImageIcon` supports several image formats, including Graphics Interchange Format (GIF), Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG).

Line 28 declares an `ImageIcon`. The file `bug1.png` contains the image to load and store in the `ImageIcon` object. This image is included in the directory for this example. The `ImageIcon` object is assigned to `Icon` reference `bug`.

Loading an Image Resource

In line 28, the expression `getClass().getResource("bug1.png")` invokes method `getClass` (inherited indirectly from class `Object`) to retrieve a reference to the `Class` object that represents the `LabelFrame` class declaration. That reference is then used to invoke `Class` method `getResource`, which returns the location of the image as a URL. The `ImageIcon` constructor uses the URL to locate the image, then loads it into memory. As we discussed in Chapter 1, the JVM loads class declarations into memory, using a class loader. The class loader knows where each class it loads is located on disk. Method `getResource` uses the `Class` object's class loader to determine the *location* of a resource, such as an image file. In this example, the image file is stored in the same location as the `LabelFrame.class` file. The techniques described here enable an application to load image files from locations that are relative to the class file's location.

Creating and Attaching `label12`

Lines 29–30 use another `JLabel` constructor to create a `JLabel` that displays the text "Label with text and icon" and the `Icon` `bug` created in line 28. The last constructor argument indicates that the label's contents are left justified, or left aligned (i.e., the icon and text are at the left side of the label's area on the screen). Interface `SwingConstants` (package `javax.swing`) declares a set of common integer constants (such as `SwingConstants.LEFT`, `SwingConstants.CENTER` and `SwingConstants.RIGHT`) that are used with many Swing components. By default, the text appears to the right of the image when a label contains both text and an image. The horizontal and vertical alignments of a `JLabel` can be set with

methods `setHorizontalAlignment` and `setVerticalAlignment`, respectively. Line 31 specifies the tool-tip text for `label12`, and line 32 adds `label12` to the `JFrame`.

Creating and Attaching `label13`

Class `JLabel` provides methods to change a `JLabel`'s appearance after it's been instantiated. Line 34 creates an empty `JLabel` with the no-argument constructor. Line 35 uses `JLabel` method `setText` to set the text displayed on the label. Method `getText` can be used to retrieve the `JLabel`'s current text. Line 36 uses `JLabel` method `setIcon` to specify the icon to display. Method `getIcon` can be used to retrieve the current icon displayed on a label. Lines 37–38 use `JLabel` methods `setHorizontalTextPosition` and `setVerticalTextPosition` to specify the text position in the label. In this case, the text will be centered horizontally and will appear at the *bottom* of the label. Thus, the icon will appear *above* the text. The horizontal-position constants in `SwingConstants` are `LEFT`, `CENTER` and `RIGHT` (Fig. 26.8). The vertical-position constants in `SwingConstants` are `TOP`, `CENTER` and `BOTTOM` (Fig. 26.8). Line 39 (Fig. 26.6) sets the tool-tip text for `label13`. Line 40 adds `label13` to the `JFrame`.

Constant	Description	Constant	Description
<i>Horizontal-position constants</i>			
<code>LEFT</code>	Place text on the left	<code>TOP</code>	Place text at the top
<code>CENTER</code>	Place text in the center	<code>CENTER</code>	Place text in the center
<code>RIGHT</code>	Place text on the right	<code>BOTTOM</code>	Place text at the bottom
<i>Vertical-position constants</i>			

Fig. 26.8 | Positioning constants (static members of interface `SwingConstants`).

Creating and Displaying a `LabelFrame` Window

Class `LabelTest` (Fig. 26.7) creates an object of class `LabelFrame` (line 9), then specifies the default close operation for the window. By default, closing a window simply *hides* the window. However, when the user closes the `LabelFrame` window, we would like the application to *terminate*. Line 10 invokes `LabelFrame`'s `setDefaultCloseOperation` method (inherited from class `JFrame`) with constant `JFrame.EXIT_ON_CLOSE` as the argument to indicate that the program should *terminate* when the window is closed by the user. This line is important. Without it the application will *not* terminate when the user closes the window. Next, line 11 invokes `LabelFrame`'s `setSize` method to specify the *width* and *height* of the window in *pixels*. Finally, line 12 invokes `LabelFrame`'s `setVisible` method with the argument `true` to display the window on the screen. Try resizing the window to see how the `FlowLayout` changes the `JLabel` positions as the window width changes.

26.6 Text Fields and an Introduction to Event Handling with Nested Classes

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the `Send` button tells the application to send the e-mail to the specified e-mail addresses. GUIs are **event driven**. When the user interacts with a GUI component, the in-

teraction—known as an **event**—drives the program to perform a task. Some common user interactions that cause an application to perform a task include *clicking* a button, *typing* in a text field, *selecting* an item from a menu, *closing* a window and *moving* the mouse. The code that performs a task in response to an event is called an **event handler**, and the process of responding to events is known as **event handling**.

Let's consider two other GUI components that can generate events—**JTextFields** and **JPasswordFields** (package `javax.swing`). Class **JTextField** extends class **JTextComponent** (package `javax.swing.text`), which provides many features common to Swing's text-based components. Class **JPasswordField** extends **JTextField** and adds methods that are specific to processing passwords. Each of these components is a single-line area in which the user can enter text via the keyboard. Applications can also display text in a **JTextField** (see the output of Fig. 26.10). A **JPasswordField** shows that characters are being typed as the user enters them, but hides the actual characters with an **echo character**, assuming that they represent a password that should remain known only to the user.

When the user types in a **JTextField** or a **JPasswordField**, then presses *Enter*, an *event* occurs. Our next example demonstrates how a program can perform a task *in response* to that event. The techniques shown here are applicable to all GUI components that generate events.

The application of Figs. 26.9–26.10 uses classes **JTextField** and **JPasswordField** to create and manipulate four text fields. When the user types in one of the text fields, then presses *Enter*, the application displays a message dialog box containing the text the user typed. You can type only in the text field that's “in **focus**.” When you *click* a component, it *receives the focus*. This is important, because the text field with the focus is the one that generates an event when you press *Enter*. In this example, when you press *Enter* in the **JPasswordField**, the password is revealed. We begin by discussing the setup of the GUI, then discuss the event-handling code.

```

1 // Fig. 26.9: TextFieldFrame.java
2 // JTextFields and JPasswordField.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextField;
8 import javax.swing.JPasswordField;
9 import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13     private final JTextField textField1; // text field with set size
14     private final JTextField textField2; // text field with text
15     private final JTextField textField3; // text field with text and size
16     private final JPasswordField passwordField; // password field with text
17
18     // TextFieldFrame constructor adds JTextFields to JFrame
19     public TextFieldFrame()
20     {

```

Fig. 26.9 | **JTextFields and JPasswordField.** (Part I of 3.)

```
21     super("Testing JTextField and JPasswordField");
22     setLayout(new FlowLayout());
23
24     // construct text field with 10 columns
25     textField1 = new JTextField(10);
26     add(textField1); // add textField1 to JFrame
27
28     // construct text field with default text
29     textField2 = new JTextField("Enter text here");
30     add(textField2); // add textField2 to JFrame
31
32     // construct text field with default text and 21 columns
33     textField3 = new JTextField("Uneditable text field", 21);
34     textField3.setEditable(false); // disable editing
35     add(textField3); // add textField3 to JFrame
36
37     // construct password field with default text
38     passwordField = new JPasswordField("Hidden text");
39     add(passwordField); // add passwordField to JFrame
40
41     // register event handlers
42     TextFieldHandler handler = new TextFieldHandler();
43     textField1.addActionListener(handler);
44     textField2.addActionListener(handler);
45     textField3.addActionListener(handler);
46     passwordField.addActionListener(handler);
47 }
48
49 // private inner class for event handling
50 private class TextFieldHandler implements ActionListener
51 {
52     // process text field events
53     @Override
54     public void actionPerformed(ActionEvent event)
55     {
56         String string = "";
57
58         // user pressed Enter in JTextField textField1
59         if (event.getSource() == textField1)
60             string = String.format("textField1: %s",
61                 event.getActionCommand());
62
63         // user pressed Enter in JTextField textField2
64         else if (event.getSource() == textField2)
65             string = String.format("textField2: %s",
66                 event.getActionCommand());
67
68         // user pressed Enter in JTextField textField3
69         else if (event.getSource() == textField3)
70             string = String.format("textField3: %s",
71                 event.getActionCommand());
72 }
```

Fig. 26.9 | JTextFields and JPasswordFields. (Part 2 of 3.)

```

73         // user pressed Enter in JTextField passwordField
74     else if (event.getSource() == passwordField)
75         string = String.format("passwordField: %s",
76             event.getActionCommand());
77
78     // display JTextField content
79     JOptionPane.showMessageDialog(null, string);
80 }
81 } // end private inner class TextFieldHandler
82 }
```

Fig. 26.9 | JTextFields and JPasswordFields. (Part 3 of 3.)

Class `TextFieldFrame` extends `JFrame` and declares three `JTextField` variables and a `JPasswordField` variable (lines 13–16). Each of the corresponding text fields is instantiated and attached to the `TextFieldFrame` in the constructor (lines 19–47).

Creating the GUI

Line 22 sets the `TextFieldFrame`'s layout to `FlowLayout`. Line 25 creates `textField1` with 10 columns of text. A text column's width in *pixels* is determined by the average width of a character in the text field's current font. When text is displayed in a text field and the text is wider than the field itself, a portion of the text at the right side is not visible. If you're typing in a text field and the cursor reaches the right edge, the text at the left edge is pushed off the left side of the field and is no longer visible. Users can use the left and right arrow keys to move through the complete text. Line 26 adds `textField1` to the `JFrame`.

Line 29 creates `textField2` with the initial text "Enter text here" to display in the text field. The width of the field is determined by the width of the default text specified in the constructor. Line 30 adds `textField2` to the `JFrame`.

Line 33 creates `textField3` and calls the `JTextField` constructor with two arguments—the default text "Uneditable text field" to display and the text field's width in columns (21). Line 34 uses method `setEditable` (inherited by `JTextField` from class `JTextComponent`) to make the text field *uneditable*—i.e., the user cannot modify the text in the field. Line 35 adds `textField3` to the `JFrame`.

Line 38 creates `passwordField` with the text "Hidden text" to display in the text field. The width of the field is determined by the width of the default text. When you execute the application, notice that the text is displayed as a string of asterisks. Line 39 adds `passwordField` to the `JFrame`.

Steps Required to Set Up Event Handling for a GUI Component

This example should display a message dialog containing the text from a text field when the user presses *Enter* in that text field. Before an application can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of the class from *Step 1* should be notified when the event occurs—known as **registering the event handler**.

Using a Nested Class to Implement an Event Handler

All the classes discussed so far were so-called **top-level classes**—that is, they *were* not declared *inside* another class. Java allows you to declare classes *inside* other classes—these are called **nested classes**. Nested classes can be **static** or non-**static**. Non-**static** nested classes are called **inner classes** and are frequently used to implement *event handlers*.

An inner-class object must be created by an object of the top-level class that contains the inner class. Each inner-class object *implicitly* has a reference to an object of its top-level class. The inner-class object is allowed to use this implicit reference to directly access all the variables and methods of the top-level class. A nested class that's **static** does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class. As you'll see in Chapter 27, Graphics and Java 2D, the Java 2D graphics API uses **static** nested classes extensively.

Inner Class `TextFieldHandler`

The event handling in this example is performed by an object of the **private inner class** `TextFieldHandler` (lines 50–81). This class is **private** because it will be used only to create event handlers for the text fields in top-level class `TextFieldFrame`. As with other class members, *inner classes* can be declared **public**, **protected** or **private**. Since event handlers tend to be specific to the application in which they're defined, they're often implemented as **private inner classes** or as *anonymous inner classes* (Section 26.11).

GUI components can generate many events in response to user interactions. Each event is represented by a class and can be processed only by the appropriate type of event handler. Normally, a component's supported events are described in the Java API documentation for that component's class and its superclasses. When the user presses *Enter* in a `JTextField` or `JPasswordField`, an **ActionEvent** (package `java.awt.event`) occurs. Such an event is processed by an object that implements the interface **ActionListener** (package `java.awt.event`). The information discussed here is available in the Java API documentation for classes `JTextField` and `ActionEvent`. Since `JPasswordField` is a sub-class of `JTextField`, `JPasswordField` supports the same events.

To prepare to handle the events in this example, inner class `TextFieldHandler` implements interface `ActionListener` and declares the only method in that interface—`actionPerformed` (lines 53–80). This method specifies the tasks to perform when an `ActionEvent` occurs. So, inner class `TextFieldHandler` satisfies *Step 1* listed earlier in this section. We'll discuss the details of method `actionPerformed` shortly.

Registering the Event Handler for Each Text Field

In the `TextFieldFrame` constructor, line 42 creates a `TextFieldHandler` object and assigns it to variable `handler`. This object's `actionPerformed` method will be called automatically when the user presses *Enter* in any of the GUI's text fields. However, before this can occur, the program must register this object as the event handler for each text field. Lines 43–46 are the *event-registration* statements that specify `handler` as the event handler for the three `JTextFields` and the `JPasswordField`. The application calls `JTextField` method `addActionListener` to register the event handler for each component. This method receives as its argument an `ActionListener` object, which can be an object of any class that implements `ActionListener`. The object `handler` *is an ActionListener*, because class `TextFieldHandler` implements `ActionListener`. After lines 43–46 execute, the object `handler` **listens for events**. Now, when the user presses *Enter* in any of these four text

fields, method `actionPerformed` (line 53–80) in class `TextFieldHandler` is called to handle the event. If an event handler is *not* registered for a particular text field, the event that occurs when the user presses *Enter* in that text field is **consumed**—i.e., it's simply *ignored* by the application.



Software Engineering Observation 26.1

The event listener for an event must implement the appropriate event-listener interface.



Common Programming Error 26.2

If you forget to register an event-handler object for a particular GUI component's event type, events of that type will be ignored.

Details of Class `TextFieldHandler`'s `actionPerformed` Method

In this example, we use one event-handling object's `actionPerformed` method (lines 53–80) to handle the events generated by four text fields. Since we'd like to output the name of each text field's instance variable for demonstration purposes, we must determine *which* text field generated the event each time `actionPerformed` is called. The **event source** is the component with which the user interacted. When the user presses *Enter* while a text field or the password field *has the focus*, the system creates a unique `ActionEvent` object that contains information about the event that just occurred, such as the event source and the text in the text field. The system passes this `ActionEvent` object to the event listener's `actionPerformed` method. Line 56 declares the `String` that will be displayed. The variable is initialized with the **empty string**—a `String` containing no characters. The compiler requires the variable to be initialized in case none of the branches of the nested `if` in lines 59–76 executes.

`ActionEvent` method `getSource` (called in lines 59, 64, 69 and 74) returns a reference to the event source. The condition in line 59 asks, “Is the event source `textField1`?” This condition compares references with the `==` operator to determine if they refer to the same object. If they *both* refer to `textField1`, the user pressed *Enter* in `textField1`. Then, lines 60–61 create a `String` containing the message that line 79 displays in a message dialog. Line 61 uses `ActionEvent` method `getActionCommand` to obtain the text the user typed in the text field that generated the event.

In this example, we display the text of the password in the `JPasswordField` when the user presses *Enter* in that field. Sometimes it's necessary to programmatically process the characters in a password. Class `JPasswordField` method `getPassword` returns the password's characters as an array of type `char`.

Class `TextFieldTest`

Class `TextFieldTest` (Fig. 26.10) contains the `main` method that executes this application and displays an object of class `TextFieldFrame`. When you execute the application, even the uneditable `JTextField` (`textField3`) can generate an `ActionEvent`. To test this, click the text field to give it the focus, then press *Enter*. Also, the actual text of the password is displayed when you press *Enter* in the `JPasswordField`. Of course, you would normally not display the password!

This application used a single object of class `TextFieldHandler` as the event listener for four text fields. Starting in Section 26.10, you'll see that it's possible to declare several

```
1 // Fig. 26.10: TextFieldTest.java
2 // Testing JTextFieldFrame.
3 import javax.swing.JFrame;
4
5 public class JTextFieldTest
6 {
7     public static void main(String[] args)
8     {
9         JTextFieldFrame textFieldFrame = new JTextFieldFrame();
10        textFieldFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        textFieldFrame.setSize(350, 100);
12        textFieldFrame.setVisible(true);
13    }
14 }
```

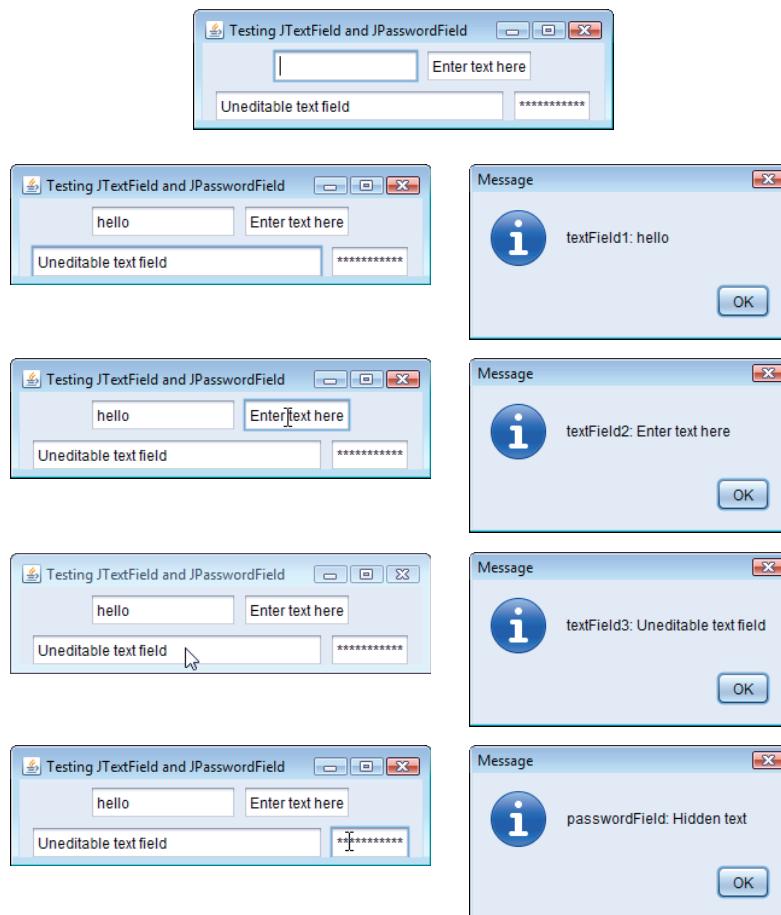


Fig. 26.10 | Testing JTextFieldFrame.

event-listener objects of the same type and register each object for a separate GUI component's event. This technique enables us to eliminate the `if...else` logic used in this example's event handler by providing separate event handlers for each component's events.

8 Java SE 8: Implementing Event Listeners with Lambdas

Recall that interfaces like `ActionListener` that have only one abstract method are functional interfaces in Java SE 8. In Section 17.16, we showed how to use lambdas to implement such event-listener interfaces.

26.7 Common GUI Event Types and Listener Interfaces

In Section 26.6, you learned that information about the event that occurs when the user presses `Enter` in a text field is stored in an `ActionEvent` object. Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends `AWTEvent` (from package `java.awt`). Figure 26.11 illustrates a hierarchy containing many event classes from the package `java.awt.event`. Some of these are discussed in this chapter and Chapter 35. These event types are used with both AWT and Swing components. Additional event types that are specific to Swing GUI components are declared in package `javax.swing.event`.

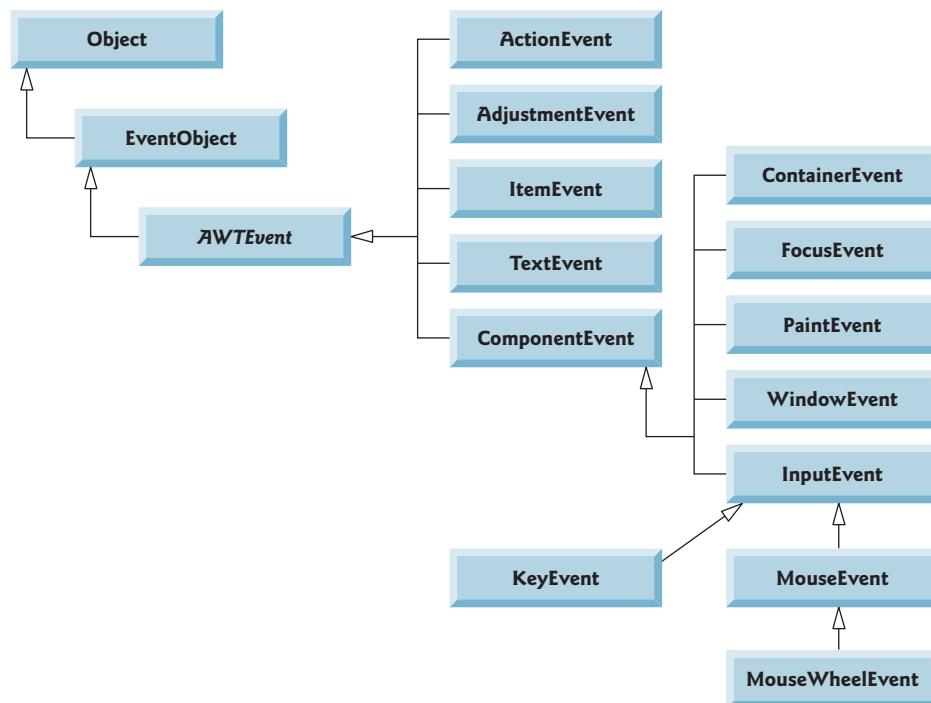


Fig. 26.11 | Some event classes of package `java.awt.event`.

Let's summarize the three parts to the event-handling mechanism that you saw in Section 26.6—the *event source*, the *event object* and the *event listener*. The event source is

the GUI component with which the user interacts. The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that's notified by the event source when an event occurs; in effect, it "listens" for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event's processing is delegated to an object (the event listener) in the application.

For each event-object type, there's typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from packages `java.awt.event` and `javax.swing.event`. Many of the event-listener types are common to both Swing and AWT components. Such types are declared in package `java.awt.event`, and some of them are shown in Fig. 26.12. Additional event-listener types that are specific to Swing components are declared in package `javax.swing.event`.

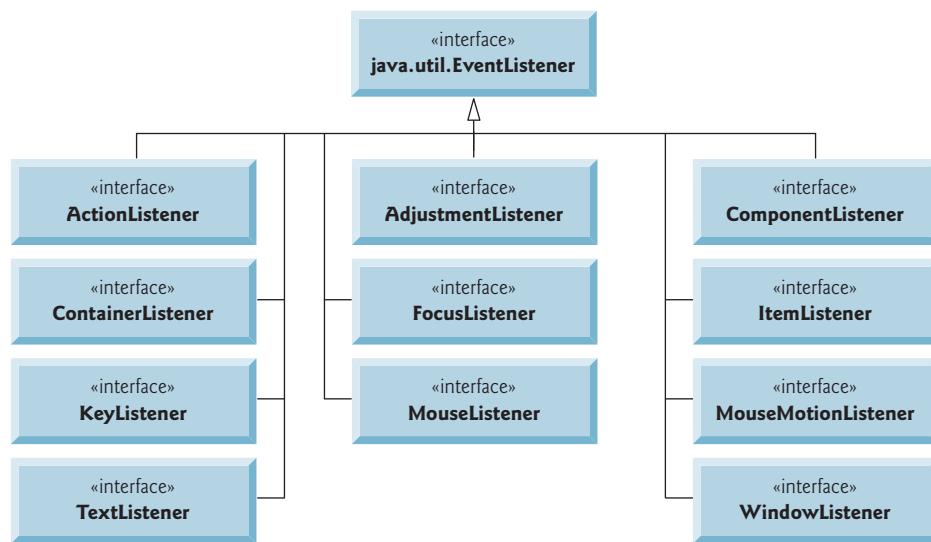


Fig. 26.12 | Some common event-listener interfaces of package `java.awt.event`.

Each event-listener interface specifies one or more event-handling methods that *must* be declared in the class that implements the interface. Recall from Section 10.9 that any class which implements an interface must declare *all* the abstract methods of that interface; otherwise, the class is an **abstract** class and cannot be used to create objects.

When an event occurs, the GUI component with which the user interacted notifies its *registered listeners* by calling each listener's appropriate *event-handling method*. For example, when the user presses the *Enter* key in a `JTextField`, the registered listener's `actionPerformed` method is called. In the next section, we complete our discussion of how the event handling works in the preceding example.

26.8 How Event Handling Works

Let's illustrate how the event-handling mechanism works, using `textField1` from the example of Fig. 26.9. We have two remaining open questions from Section 26.7:

1. How did the *event handler* get registered?
2. How does the GUI component know to call `actionPerformed` rather than some other event-handling method?

The first question is answered by the event registration performed in lines 43–46 of Fig. 26.9. Figure 26.13 diagrams `JTextField` variable `textField1`, `TextFieldHandler` variable `handler` and the objects to which they refer.

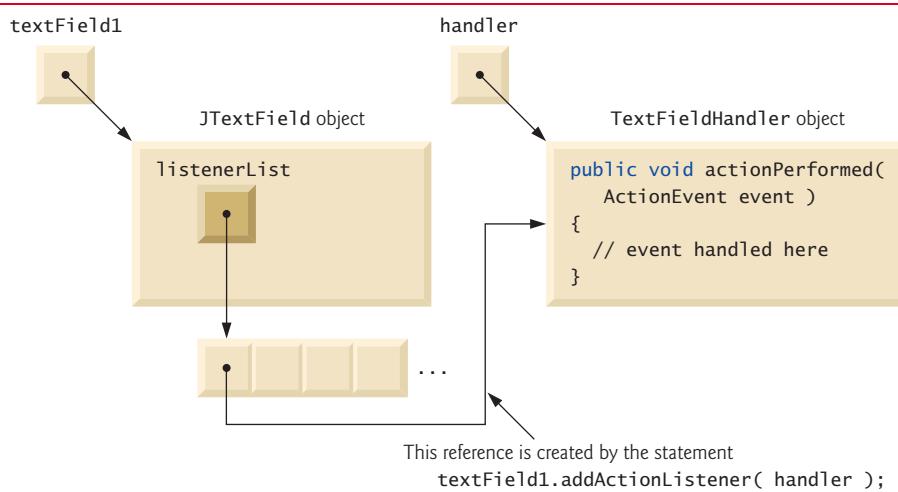


Fig. 26.13 | Event registration for `JTextField` `textField1`.

Registering Events

Every `JComponent` has an instance variable called `listenerList` that refers to an object of class `EventListenerList` (package `javax.swing.event`). Each object of a `JComponent` subclass maintains references to its *registered listeners* in the `listenerList`. For simplicity, we've diagrammed `listenerList` as an array below the `JTextField` object in Fig. 26.13.

When the following statement (line 43 of Fig. 26.9) executes

```
textField1.addActionListener(handler);
```

a new entry containing a reference to the `TextFieldHandler` object is placed in `textField1`'s `listenerList`. Although not shown in the diagram, this new entry also includes the listener's type (`ActionListener`). Using this mechanism, each lightweight Swing component maintains its own list of *listeners* that were *registered* to *handle* the component's *events*.

Event-Handler Invocation

The event-listener type is important in answering the second question: How does the GUI component know to call `actionPerformed` rather than another method? Every GUI component supports several *event types*, including **mouse events**, **key events** and others. When

an event occurs, the event is **dispatched** only to the *event listeners* of the appropriate type. Dispatching is simply the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the event type that occurred.

Each *event type* has one or more corresponding *event-listener interfaces*. For example, **ActionEvents** are handled by **ActionListeners**, **MouseEvents** by **MouseListener**s and **MouseMotionListeners**, and **KeyEvents** by **KeyListeners**. When an event occurs, the GUI component receives (from the JVM) a unique *event ID* specifying the event type. The GUI component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object. For an **ActionEvent**, the event is dispatched to *every* registered **ActionListener**'s **actionPerformed** method (the only method in interface **ActionListener**). For a **MouseEvent**, the event is dispatched to *every* registered **MouseListener** or **MouseMotionListener**, depending on the mouse event that occurs. The **MouseEvent**'s event ID determines which of the several mouse event-handling methods are called. All these decisions are handled for you by the GUI components. All you need to do is register an event handler for the particular event type that your application requires, and the GUI component will ensure that the event handler's appropriate method gets called when the event occurs. We discuss other event types and event-listener interfaces as they're needed with each new component we introduce.



Performance Tip 26.1

GUIs should always remain responsive to the user. Performing a long-running task in an event handler prevents the user from interacting with the GUI until that task completes. Section 23.11 demonstrates techniques prevent such problems.

26.9 JButton

A **button** is a component the user clicks to trigger a specific action. A Java application can use several types of buttons, including **command buttons**, **checkboxes**, **toggle buttons** and **radio buttons**. Figure 26.14 shows the inheritance hierarchy of the Swing buttons we cover in this chapter. As you can see, all the button types are subclasses of **AbstractButton** (package `javax.swing`), which declares the common features of Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command.

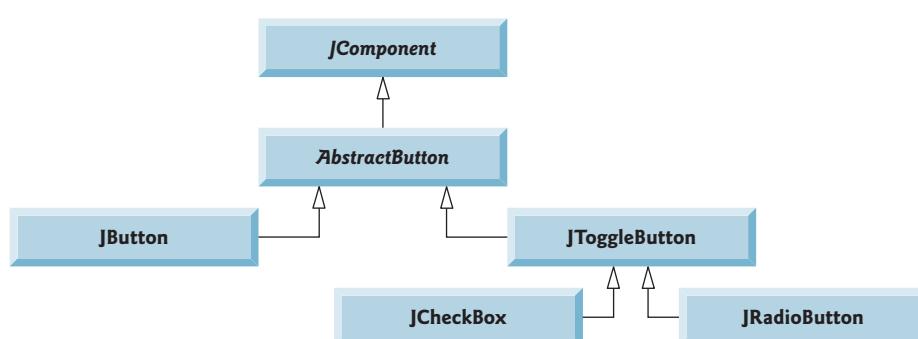


Fig. 26.14 | Swing button hierarchy.

A *command button* (see Fig. 26.16's output) generates an `ActionEvent` when the user clicks it. Command buttons are created with class `JButton`. The text on the face of a `JButton` is called a **button label**.



Look-and-Feel Observation 26.8

The text on buttons typically uses book-title capitalization.



Look-and-Feel Observation 26.9

A GUI can have many `JButtons`, but each button label should be unique in the portion of the GUI that's currently displayed. Having more than one `JButton` with the same label makes the `JButtons` ambiguous to the user.

The application of Figs. 26.15 and 26.16 creates two `JButtons` and demonstrates that `JButtons` can display `Icons`. Event handling for the buttons is performed by a single instance of *inner class* `ButtonHandler` (Fig. 26.15, lines 39–48).

```

1 // Fig. 26.15: ButtonFrame.java
2 // Command buttons and action events.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JButton;
8 import javax.swing.Icon;
9 import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14     private final JButton plainJButton; // button with just text
15     private final JButton fancyJButton; // button with icons
16
17     // ButtonFrame adds JButtons to JFrame
18     public ButtonFrame()
19     {
20         super("Testing Buttons");
21         setLayout(new FlowLayout());
22
23         plainJButton = new JButton("Plain Button"); // button with text
24         add(plainJButton); // add plainJButton to JFrame
25
26         Icon bug1 = new ImageIcon(getClass().getResource("bug1.gif"));
27         Icon bug2 = new ImageIcon(getClass().getResource("bug2.gif"));
28         fancyJButton = new JButton("Fancy Button", bug1); // set image
29         fancyJButton.setRolloverIcon(bug2); // set rollover image
30         add(fancyJButton); // add fancyJButton to JFrame
31
32         // create new ButtonHandler for button event handling
33         ButtonHandler handler = new ButtonHandler();

```

Fig. 26.15 | Command buttons and action events. (Part I of 2.)

```

34     fancyJButton.addActionListener(handler);
35     plainJButton.addActionListener(handler);
36 }
37
38 // inner class for button event handling
39 private class ButtonHandler implements ActionListener
40 {
41     // handle button event
42     @Override
43     public void actionPerformed(ActionEvent event)
44     {
45         JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
46             "You pressed: %s", event.getActionCommand()));
47     }
48 }
49 }
```

Fig. 26.15 | Command buttons and action events. (Part 2 of 2.)

Lines 14–15 declare JButton variables plainJButton and fancyJButton. The corresponding objects are instantiated in the constructor. Line 23 creates plainJButton with the button label "Plain Button". Line 24 adds the JButton to the JFrame.

A JButton can display an Icon. To provide the user with an extra level of visual interaction with the GUI, a JButton can also have a **rollover Icon**—an Icon that's displayed when the user positions the mouse over the JButton. The icon on the JButton changes as the mouse moves in and out of the JButton's area on the screen. Lines 26–27 create two ImageIcon objects that represent the default Icon and rollover Icon for the JButton created at line 28. Both statements assume that the image files are stored in the *same* directory as the application. Images are commonly placed in the *same* directory as the application or a subdirectory like `images`. These image files have been provided for you with the example.

Line 28 creates fancyButton with the text "Fancy Button" and the icon bug1. By default, the text is displayed to the *right* of the icon. Line 29 uses `setRolloverIcon` (inherited from class AbstractButton) to specify the image displayed on the JButton when the user positions the mouse over it. Line 30 adds the JButton to the JFrame.



Look-and-Feel Observation 26.10

Because class AbstractButton supports displaying text and images on a button, all subclasses of AbstractButton also support displaying text and images.



Look-and-Feel Observation 26.11

Rollover icons provide visual feedback indicating that an action will occur when a JButton is clicked.

JButtons, like JTextFields, generate ActionEvents that can be processed by any ActionListener object. Lines 33–35 create an object of private *inner class* ButtonHandler and use `addActionListener` to register it as the *event handler* for each JButton. Class ButtonHandler (lines 39–48) declares `actionPerformed` to display a message dialog box

containing the label for the button the user pressed. For a JButton event, ActionEvent method `getActionCommand` returns the label on the JButton.

```

1 // Fig. 26.16: ButtonTest.java
2 // Testing ButtonFrame.
3 import javax.swing.JFrame;
4
5 public class ButtonTest
6 {
7     public static void main(String[] args)
8     {
9         ButtonFrame buttonFrame = new ButtonFrame();
10        buttonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        buttonFrame.setSize(275, 110);
12        buttonFrame.setVisible(true);
13    }
14 }
```

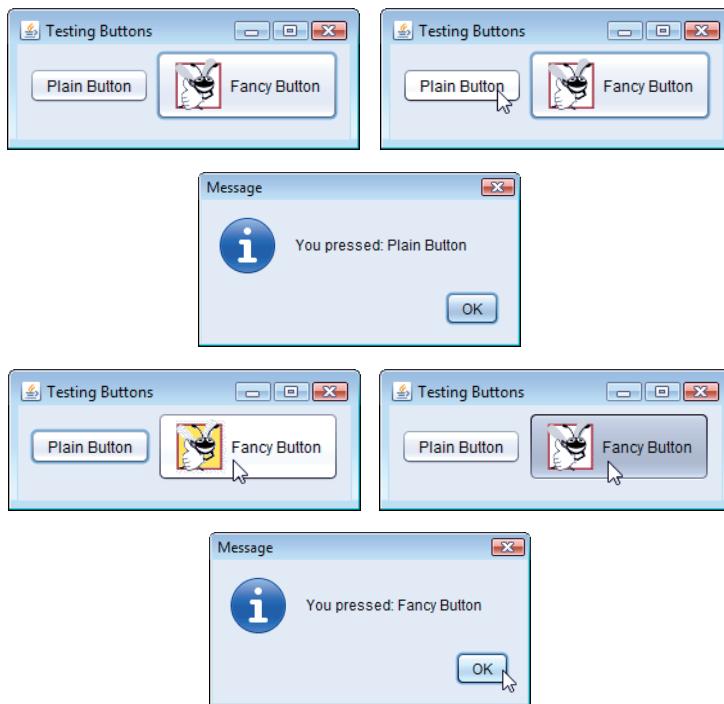


Fig. 26.16 | Testing ButtonFrame.

Accessing the `this` Reference in an Object of a Top-Level Class from an Inner Class

When you execute this application and click one of its buttons, notice that the message dialog that appears is centered over the application's window. This occurs because the call to JOptionPane method `showMessageDialog` (lines 45–46) uses `ButtonFrame.this` rather than `null` as the first argument. When this argument is not `null`, it represents the so-called

parent GUI component of the message dialog (in this case the application window is the parent component) and enables the dialog to be centered over that component when the dialog is displayed. `ButtonFrame.this` represents the `this` reference of the object of top-level class `ButtonFrame`.



Software Engineering Observation 26.2

When used in an inner class, keyword `this` refers to the current inner-class object being manipulated. An inner-class method can use its outer-class object's `this` by preceding `this` with the outer-class name and a dot (.) separator, as in `ButtonFrame.this`.

26.10 Buttons That Maintain State

The Swing GUI components contain three types of **state buttons**—`JToggleButton`, `JCheckBox` and `JRadioButton`—that have on/off or true/false values. Classes `JCheckBox` and `JRadioButton` are subclasses of `JToggleButton` (Fig. 26.14). A `JRadioButton` is different from a `JCheckBox` in that normally several `JRadioButtons` are grouped together and are *mutually exclusive*—only *one* in the group can be selected at any time, just like the buttons on a car radio. We first discuss class `JCheckBox`.

26.10.1 JCheckBox

The application of Figs. 26.17–26.18 uses two `JCheckBoxes` to select the desired font style of the text displayed in a `JTextField`. When selected, one applies a bold style and the other an italic style. If *both* are selected, the style is bold *and* italic. When the application initially executes, neither `JCheckBox` is checked (i.e., they're both `false`), so the font is plain. Class `CheckBoxTest` (Fig. 26.18) contains the `main` method that executes this application.

```

1 // Fig. 26.17: CheckBoxFrame.java
2 // JCheckboxes and item events.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13     private final JTextField textField; // displays text in changing fonts
14     private final JCheckBox boldJCheckBox; // to select/deselect bold
15     private final JCheckBox italicJCheckBox; // to select/deselect italic
16
17     // CheckBoxFrame constructor adds JCheckboxes to JFrame
18     public CheckBoxFrame()
19     {
20         super("JCheckBox Test");
21         setLayout(new FlowLayout());
22     }

```

Fig. 26.17 | `JCheckboxes` and item events. (Part 1 of 2.)

```

23      // set up JTextField and set its font
24      textField = new JTextField("Watch the font style change", 20);
25      textField.setFont(new Font("Serif", Font.PLAIN, 14));
26      add(textField); // add textField to JFrame
27
28      boldJCheckBox = new JCheckBox("Bold");
29      italicJCheckBox = new JCheckBox("Italic");
30      add(boldJCheckBox); // add bold checkbox to JFrame
31      add(italicJCheckBox); // add italic checkbox to JFrame
32
33      // register listeners for JCheckboxes
34      CheckBoxHandler handler = new CheckBoxHandler();
35      boldJCheckBox.addItemListener(handler);
36      italicJCheckBox.addItemListener(handler);
37  }
38
39  // private inner class for ItemListener event handling
40  private class CheckBoxHandler implements ItemListener
41  {
42      // respond to checkbox events
43      @Override
44      public void itemStateChanged(ItemEvent event)
45      {
46          Font font = null; // stores the new Font
47
48          // determine which Checkboxes are checked and create Font
49          if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected())
50              font = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
51          else if (boldJCheckBox.isSelected())
52              font = new Font("Serif", Font.BOLD, 14);
53          else if (italicJCheckBox.isSelected())
54              font = new Font("Serif", Font.ITALIC, 14);
55          else
56              font = new Font("Serif", Font.PLAIN, 14);
57
58          textField.setFont(font);
59      }
60  }
61 }
```

Fig. 26.17 | JCheckboxes and item events. (Part 2 of 2.)

```

1  // Fig. 26.18: CheckBoxTest.java
2  // Testing CheckBoxFrame.
3  import javax.swing.JFrame;
4
5  public class CheckBoxTest
6  {
7      public static void main(String[] args)
8      {
9          CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
10         checkBoxFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Fig. 26.18 | Testing CheckBoxFrame. (Part 1 of 2.)

```

11     checkBoxFrame.setSize(275, 100);
12     checkBoxFrame.setVisible(true);
13 }
14 }
```

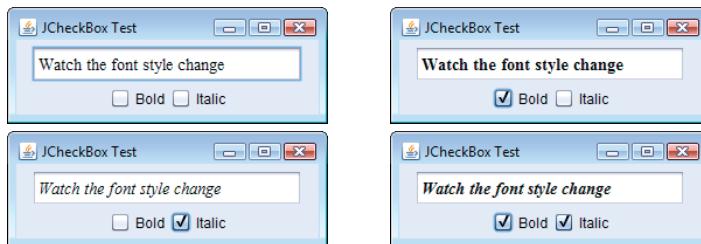


Fig. 26.18 | Testing CheckBoxFrame. (Part 2 of 2.)

After the JTextField is created and initialized (Fig. 26.17, line 24), line 25 uses method `setFont` (inherited by JTextField indirectly from class Component) to set the font of the JTextField to a new object of class `Font` (package `java.awt`). The new Font is initialized with "Serif" (a generic font name that represents a font such as Times and is supported on all Java platforms), `Font.PLAIN` style and 14-point size. Next, lines 28–29 create two JCheckBox objects. The String passed to the JCheckBox constructor is the `checkbox label` that appears to the right of the JCheckBox by default.

When the user clicks a JCheckBox, an `ItemEvent` occurs. This event can be handled by an `ItemListener` object, which *must* implement method `itemStateChanged`. In this example, the event handling is performed by an instance of private *inner class* `CheckBoxHandler` (lines 40–60). Lines 34–36 create an instance of class `CheckBoxHandler` and register it with method `addItemListener` as the listener for both the JCheckBox objects.

`CheckBoxHandler` method `itemStateChanged` (lines 43–59) is called when the user clicks the `boldJCheckBox` or `italicJCheckBox`. In this example, we do not determine which JCheckBox was clicked—we use both states to determine the font to display. Line 49 uses JCheckBox method `isSelected` to determine if both JCheckboxes are selected. If so, line 50 creates a bold italic font by adding the `Font` constants `Font.BOLD` and `Font.ITALIC` for the font-style argument of the `Font` constructor. Line 51 determines whether the `boldJCheckBox` is selected, and if so line 52 creates a bold font. Line 53 determines whether the `italicJCheckBox` is selected, and if so line 54 creates an italic font. If none of the preceding conditions are true, line 56 creates a plain font using the `Font` constant `Font.PLAIN`. Finally, line 58 sets `textField`'s new font, which changes the font in the JTextField on the screen.

Relationship Between an Inner Class and Its Top-Level Class

Class `CheckBoxHandler` used variables `boldJCheckBox` (lines 49 and 51), `italicJCheckBox` (lines 49 and 53) and `textField` (line 58) even though they are *not* declared in the inner class. Recall that an *inner class* has a special relationship with its *top-level class*—it's allowed to access *all* the variables and methods of the top-level class. `CheckBoxHandler` method `itemStateChanged` (line 43–59) uses this relationship to determine which JCheckboxes are checked and to set the font on the JTextField. Notice that none of the code in inner class `CheckBoxHandler` requires an explicit reference to the top-level class object.

26.10.2 JRadioButton

Radio buttons (declared with class `JRadioButton`) are similar to checkboxes in that they have two states—*selected* and *not selected* (also called *deselected*). However, radio buttons normally appear as a **group** in which only *one* button can be selected at a time (see the output of Fig. 26.20). Radio buttons are used to represent **mutually exclusive options** (i.e., multiple options in the group *cannot* be selected at the same time). The logical relationship between radio buttons is maintained by a **ButtonGroup** object (package `javax.swing`), which itself is *not* a GUI component. A **ButtonGroup** object organizes a group of buttons and is *not* itself displayed in a user interface. Rather, the individual `JRadioButton` objects from the group are displayed in the GUI.

The application of Figs. 26.19–26.20 is similar to that of Figs. 26.17–26.18. The user can alter the font style of a `JTextField`’s text. The application uses radio buttons that permit only a single font style in the group to be selected at a time. Class `RadioButtonTest` (Fig. 26.20) contains the `main` method that executes this application.

```

1 // Fig. 26.19: RadioButtonFrame.java
2 // Creating radio buttons using ButtonGroup and JRadioButton.
3 import java.awt.FlowLayout;
4 import java.awt.Font;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JTextField;
9 import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14     private final JTextField textField; // used to display font changes
15     private final Font plainFont; // font for plain text
16     private final Font boldFont; // font for bold text
17     private final Font italicFont; // font for italic text
18     private final Font boldItalicFont; // font for bold and italic text
19     private final JRadioButton plainJRadioButton; // selects plain text
20     private final JRadioButton boldJRadioButton; // selects bold text
21     private final JRadioButton italicJRadioButton; // selects italic text
22     private final JRadioButton boldItalicJRadioButton; // bold and italic
23     private final ButtonGroup radioGroup; // holds radio buttons
24
25     // RadioButtonFrame constructor adds JRadioButtons to JFrame
26     public RadioButtonFrame()
27     {
28         super("RadioButton Test");
29         setLayout(new FlowLayout());
30
31         textField = new JTextField("Watch the font style change", 25);
32         add(textField); // add textField to JFrame
33
34         // create radio buttons
35         plainJRadioButton = new JRadioButton("Plain", true);

```

Fig. 26.19 | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part I of 2.)

```
36     boldJRadioButton = new JRadioButton("Bold", false);
37     italicJRadioButton = new JRadioButton("Italic", false);
38     boldItalicJRadioButton = new JRadioButton("Bold/Italic", false);
39     add(plainJRadioButton); // add plain button to JFrame
40     add(boldJRadioButton); // add bold button to JFrame
41     add(italicJRadioButton); // add italic button to JFrame
42     add(boldItalicJRadioButton); // add bold and italic button
43
44     // create logical relationship between JRadioButtons
45     radioGroup = new ButtonGroup(); // create ButtonGroup
46     radioGroup.add(plainJRadioButton); // add plain to group
47     radioGroup.add(boldJRadioButton); // add bold to group
48     radioGroup.add(italicJRadioButton); // add italic to group
49     radioGroup.add(boldItalicJRadioButton); // add bold and italic
50
51     // create font objects
52     plainFont = new Font("Serif", Font.PLAIN, 14);
53     boldFont = new Font("Serif", Font.BOLD, 14);
54     italicFont = new Font("Serif", Font.ITALIC, 14);
55     boldItalicFont = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
56     textField.setFont(plainFont);
57
58     // register events for JRadioButtons
59     plainJRadioButton.addItemListener(
60         new RadioButtonHandler(plainFont));
61     boldJRadioButton.addItemListener(
62         new RadioButtonHandler(boldFont));
63     italicJRadioButton.addItemListener(
64         new RadioButtonHandler(italicFont));
65     boldItalicJRadioButton.addItemListener(
66         new RadioButtonHandler(boldItalicFont));
67 }
68
69     // private inner class to handle radio button events
70     private class RadioButtonHandler implements ItemListener
71     {
72         private Font font; // font associated with this listener
73
74         public RadioButtonHandler(Font f)
75         {
76             font = f;
77         }
78
79         // handle radio button events
80         @Override
81         public void itemStateChanged(ItemEvent event)
82         {
83             textField.setFont(font);
84         }
85     }
86 }
```

Fig. 26.19 | Creating radio buttons using `ButtonGroup` and `JRadioButton`. (Part 2 of 2.)

```

1 // Fig. 26.20: RadioButtonTest.java
2 // Testing RadioButtonFrame.
3 import javax.swing.JFrame;
4
5 public class RadioButtonTest
6 {
7     public static void main(String[] args)
8     {
9         RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
10        radioButtonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        radioButtonFrame.setSize(300, 100);
12        radioButtonFrame.setVisible(true);
13    }
14}

```

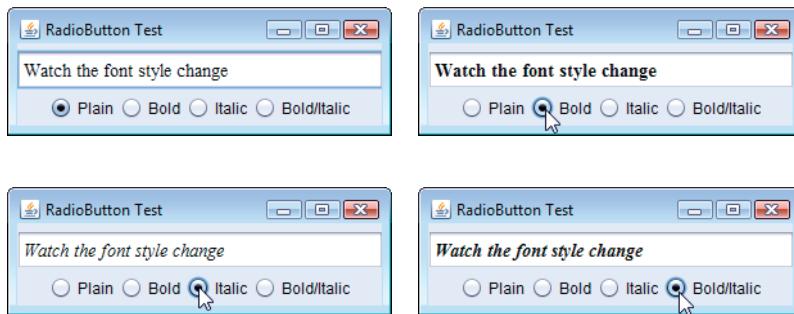


Fig. 26.20 | Testing RadioButtonFrame.

Lines 35–42 (Fig. 26.19) in the constructor create four `JRadioButton` objects and add them to the `JFrame`. Each `JRadioButton` is created with a constructor call like that in line 35. This constructor specifies the label that appears to the right of the `JRadioButton` by default and the initial state of the `JRadioButton`. A `true` second argument indicates that the `JRadioButton` should appear *selected* when it's displayed.

Line 45 instantiates `ButtonGroup` object `radioGroup`. This object is the “glue” that forms the logical relationship between the four `JRadioButton` objects and allows only one of the four to be selected at a time. It's possible that no `JRadioButtons` in a `ButtonGroup` are selected, but this can occur *only* if *no* preselected `JRadioButtons` are added to the `ButtonGroup` and the user has *not* selected a `JRadioButton` yet. Lines 46–49 use `ButtonGroup` method `add` to associate each of the `JRadioButtons` with `radioGroup`. If more than one selected `JRadioButton` object is added to the group, the selected one that was added *first* will be selected when the GUI is displayed.

`JRadioButtons`, like `JCheckboxes`, generate `ItemEvents` when they're *clicked*. Lines 59–66 create four instances of inner class `RadioButtonHandler` (declared at lines 70–85). In this example, each event-listener object is registered to handle the `ItemEvent` generated when the user clicks a particular `JRadioButton`. Notice that each `RadioButtonHandler` object is initialized with a particular `Font` object (created in lines 52–55).

Class `RadioButtonHandler` (line 70–85) implements interface `ItemListener` so it can handle `ItemEvents` generated by the `JRadioButtons`. The constructor stores the `Font`

object it receives as an argument in the event-listener object's instance variable `font` (declared at line 72). When the user clicks a `JRadioButton`, `radioGroup` turns off the previously selected `JRadioButton`, and method `itemStateChanged` (lines 80–84) sets the font in the `JTextField` to the `Font` stored in the `JRadioButton`'s corresponding event-listener object. Notice that line 83 of inner class `RadioButtonHandler` uses the top-level class's `textField` instance variable to set the font.

26.11 JComboBox; Using an Anonymous Inner Class for Event Handling

A combo box (sometimes called a **drop-down list**) enables the user to select *one* item from a list (Fig. 26.22). Combo boxes are implemented with class `JComboBox`, which extends class `JComponent`. `JComboBox` is a generic class, like the class `ArrayList` (Chapter 7). When you create a `JComboBox`, you specify the type of the objects that it manages—the `JComboBox` then displays a `String` representation of each object.

```
1 // Fig. 26.21: ComboBoxFrame.java
2 // JComboBox that displays a list of image names.
3 import java.awt.FlowLayout;
4 import java.awt.event.ItemListener;
5 import java.awt.event.ItemEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8 import javax.swing.JComboBox;
9 import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14     private final JComboBox<String> imagesJComboBox; // holds icon names
15     private final JLabel label; // displays selected icon
16
17     private static final String[] names =
18         {"bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif"};
19     private final Icon[] icons = {
20         new ImageIcon(getClass().getResource(names[0])),
21         new ImageIcon(getClass().getResource(names[1])),
22         new ImageIcon(getClass().getResource(names[2])),
23         new ImageIcon(getClass().getResource(names[3]))};
24
25     // ComboBoxFrame constructor adds JComboBox to JFrame
26     public ComboBoxFrame()
27     {
28         super("Testing JComboBox");
29         setLayout(new FlowLayout()); // set frame layout
30
31         imagesJComboBox = new JComboBox<String>(names); // set up JComboBox
32         imagesJComboBox.setMaximumRowCount(3); // display three rows
33     }
}
```

Fig. 26.21 | JComboBox that displays a list of image names. (Part 1 of 2.)

```

34     imagesJComboBox.addItemListener(
35         new ItemListener() // anonymous inner class
36     {
37         // handle JComboBox event
38         @Override
39         public void itemStateChanged(ItemEvent event)
40         {
41             // determine whether item selected
42             if (event.getStateChange() == ItemEvent.SELECTED)
43                 label.setIcon(icons[
44                     imagesJComboBox.getSelectedIndex()]);
45         }
46     } // end anonymous inner class
47 ); // end call to addItemListener
48
49     add(imagesJComboBox); // add combo box to JFrame
50     label = new JLabel(icons[0]); // display first icon
51     add(label); // add label to JFrame
52 }
53 }
```

Fig. 26.21 | JComboBox that displays a list of image names. (Part 2 of 2.)

JComboBoxes generate ItemEvents just as JCheckButtons and JRadioButtons do. This example also demonstrates a special form of inner class that's used frequently in event handling. The application (Figs. 26.21–26.22) uses a JComboBox to provide a list of four image filenames from which the user can select one image to display. When the user selects a name, the application displays the corresponding image as an Icon on a JLabel. Class ComboBoxTest (Fig. 26.22) contains the main method that executes this application. The screen captures for this application show the JComboBox list after the selection was made to illustrate which image filename was selected.

Lines 19–23 (Fig. 26.21) declare and initialize array icons with four new ImageIcon objects. String array names (lines 17–18) contains the names of the four image files that are stored in the same directory as the application.

```

1 // Fig. 26.22: ComboBoxTest.java
2 // Testing ComboBoxFrame.
3 import javax.swing.JFrame;
4
5 public class ComboBoxTest
6 {
7     public static void main(String[] args)
8     {
9         ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
10        comboBoxFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        comboBoxFrame.setSize(350, 150);
12        comboBoxFrame.setVisible(true);
13    }
14 }
```

Fig. 26.22 | Testing ComboBoxFrame. (Part I of 2.)

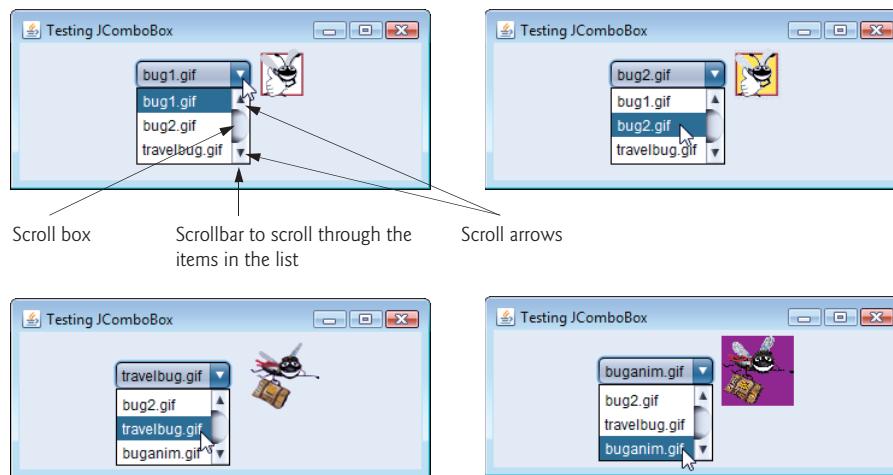


Fig. 26.22 | Testing ComboBoxFrame. (Part 2 of 2.)

At line 31, the constructor initializes a JComboBox object with the Strings in array names as the elements in the list. Each item in the list has an index. The first item is added at index 0, the next at index 1 and so forth. The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed. Other items are selected by clicking the JComboBox, then selecting an item from the list that appears.

Line 32 uses JComboBox method `setMaximumRowCount` to set the maximum number of elements that are displayed when the user clicks the JComboBox. If there are additional items, the JComboBox provides a scrollbar (see the first screen) that allows the user to scroll through all the elements in the list. The user can click the scroll arrows at the top and bottom of the scrollbar to move up and down through the list one element at a time, or else drag the scroll box in the middle of the scrollbar up and down. To drag the scroll box, position the mouse cursor on it, hold the mouse button down and move the mouse. In this example, the drop-down list is too short to drag the scroll box, so you can click the up and down arrows or use your mouse's wheel to scroll through the four items in the list. Line 49 attaches the JComboBox to the ComboBoxFrame's FlowLayout (set in line 29). Line 50 creates the JLabel that displays ImageIcon and initializes it with the first ImageIcon in array icons. Line 51 attaches the JLabel to the ComboBoxFrame's FlowLayout.



Look-and-Feel Observation 26.12

Set the maximum row count for a JComboBox to a number of rows that prevents the list from expanding outside the bounds of the window in which it's used.

Using an Anonymous Inner Class for Event Handling

Lines 34–46 are one statement that declares the event listener's class, creates an object of that class and registers it as imagesJComboBox's ItemEvent listener. This event-listener object is an instance of an **anonymous inner class**—a class that's declared without a name and typically appears inside a method declaration. As with other inner classes, an anonymous

inner class can access its top-level class's members. However, an anonymous inner class has limited access to the local variables of the method in which it's declared. Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared (starting at line 35).



Software Engineering Observation 26.3

An anonymous inner class declared in a method can access the instance variables and methods of the top-level class object that declared it, as well as the method's final local variables, but cannot access the method's non-final local variables. As of Java SE 8, anonymous inner classes may also access a methods "effectively final" local variables—see Chapter 17 for more information.

8

Lines 34–47 are a call to `imagesJComboBox`'s `addItemListener` method. The argument to this method must be an object that *is an ItemListener* (i.e., any object of a class that implements `ItemListener`). Lines 35–46 are a class-instance creation expression that declares an anonymous inner class and creates one object of that class. A reference to that object is then passed as the argument to `addItemListener`. The syntax `ItemListener()` after `new` begins the declaration of an anonymous inner class that implements interface `ItemListener`. This is similar to beginning a class declaration with

```
public class MyHandler implements ItemListener
```

The opening left brace at line 36 and the closing right brace at line 46 delimit the body of the anonymous inner class. Lines 38–45 declare the `ItemListener`'s `itemStateChanged` method. When the user makes a selection from `imagesJComboBox`, this method sets `label`'s `Icon`. The `Icon` is selected from array `icons` by determining the index of the selected item in the `JComboBox` with method `getSelectedIndex` in line 44. For each item selected from a `JComboBox`, another item is first deselected—so two `ItemEvents` occur when an item is selected. We wish to display only the icon for the item the user just selected. For this reason, line 42 determines whether `ItemEvent` method `getStateChange` returns `ItemEvent.SELECTED`. If so, lines 43–44 set `label`'s `icon`.



Software Engineering Observation 26.4

Like any other class, when an anonymous inner class implements an interface, the class must implement every abstract method in the interface.

The syntax shown in lines 35–46 for creating an event handler with an anonymous inner class is similar to the code that would be generated by a Java integrated development environment (IDE). Typically, an IDE enables you to design a GUI visually, then it generates code that implements the GUI. You simply insert statements in the event-handling methods that declare how to handle each event.

26.12 JList

A list displays a series of items from which the user may *select one or more items* (see the output of Fig. 26.24). Lists are created with class `JList`, which directly extends class `JComponent`. Class `JList`—which like `JComboBox` is a generic class—supports **single-selection lists** (which allow only one item to be selected at a time) and **multiple-selection lists** (which allow any number of items to be selected). In this section, we discuss single-selection lists.

The application of Figs. 26.23–26.24 creates a `JList` containing 13 color names. When a color name is clicked in the `JList`, a `ListSelectionEvent` occurs and the application changes the background color of the application window to the selected color. Class `ListTest` (Fig. 26.24) contains the `main` method that executes this application.

Line 29 (Fig. 26.23) creates `JList` object `colorJList`. The argument to the `JList` constructor is the array of `Objects` (in this case `Strings`) to display in the list. Line 30 uses `JList` method `setVisibleRowCount` to determine the number of items *visible* in the list.

Line 33 uses `JList` method `setSelectionMode` to specify the list's **selection mode**. Class `ListSelectionMode1` (of package `javax.swing`) declares three constants that specify a `JList`'s selection mode—`SINGLE_SELECTION` (which allows only one item to be selected at a time), `SINGLE_INTERVAL_SELECTION` (for a multiple-selection list that allows selection of several contiguous items) and `MULTIPLE_INTERVAL_SELECTION` (for a multiple-selection list that does not restrict the items that can be selected).

```

1 // Fig. 26.23: ListFrame.java
2 // JList that displays a list of colors.
3 import java.awt.FlowLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JList;
7 import javax.swing.JScrollPane;
8 import javax.swing.event.ListSelectionListener;
9 import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class ListFrame extends JFrame
13 {
14     private final JList<String> colorJList; // list to display colors
15     private static final String[] colorNames = {"Black", "Blue", "Cyan",
16         "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
17         "Orange", "Pink", "Red", "White", "Yellow"};
18     private static final Color[] colors = {Color.BLACK, Color.BLUE,
19         Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20         Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21         Color.RED, Color.WHITE, Color.YELLOW};
22
23     // ListFrame constructor add JScrollPane containing JList to JFrame
24     public ListFrame()
25     {
26         super("List Test");
27         setLayout(new FlowLayout());
28
29         colorJList = new JList<String>(colorNames); // list of colorNames
30         colorJList.setVisibleRowCount(5); // display five rows at once
31
32         // do not allow multiple selections
33         colorJList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
34
35         // add a JScrollPane containing JList to frame
36         add(new JScrollPane(colorJList));

```

Fig. 26.23 | `JList` that displays a list of colors. (Part I of 2.)

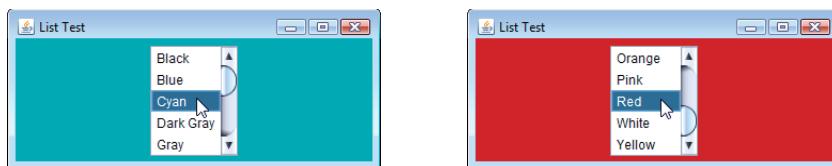
```

37
38     colorJList.addListSelectionListener(
39         new ListSelectionListener() // anonymous inner class
40     {
41         // handle list selection events
42         @Override
43         public void valueChanged(ListSelectionEvent event)
44         {
45             getContentPane().setBackground(
46                 colors[colorJList.getSelectedIndex()]);
47         }
48     });
49 }
50 }
51 }
```

Fig. 26.23 | *JList* that displays a list of colors. (Part 2 of 2.)

```

1 // Fig. 26.24: ListTest.java
2 // Selecting colors from a JList.
3 import javax.swing.JFrame;
4
5 public class ListTest
6 {
7     public static void main(String[] args)
8     {
9         ListFrame listFrame = new ListFrame(); // create ListFrame
10        listFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        listFrame.setSize(350, 150);
12        listFrame.setVisible(true);
13    }
14 }
```

**Fig. 26.24** | Selecting colors from a *JList*.

Unlike a *JComboBox*, a *JList* *does not provide a scrollbar* if there are more items in the list than the number of visible rows. In this case, a **JScrollPane** object is used to provide the scrolling capability. Line 36 adds a new instance of class **JScrollPane** to the **JFrame**. The **JScrollPane** constructor receives as its argument the **JComponent** that needs scrolling functionality (in this case, **colorJList**). Notice in the screen captures that a scrollbar created by the **JScrollPane** appears at the right side of the *JList*. By default, the scrollbar appears only when the number of items in the *JList* exceeds the number of visible items.

Lines 38–49 use *JList* method **addListSelectionListener** to register an object that implements **ListSelectionListener** (package `javax.swing.event`) as the listener for the

JList's selection events. Once again, we use an instance of an anonymous inner class (lines 39–48) as the listener. In this example, when the user makes a selection from colorJList, method **valueChanged** (line 42–47) should change the background color of the ListFrame to the selected color. This is accomplished in lines 45–46. Note the use of JFrame method **getContentPane** in line 45. Each JFrame actually consists of *three layers*—the *background*, the *content pane* and the *glass pane*. The content pane appears in front of the background and is where the GUI components in the JFrame are displayed. The glass pane is used to display tool tips and other items that should appear in front of the GUI components on the screen. The content pane completely hides the background of the JFrame; thus, to change the background color behind the GUI components, you must change the content pane's background color. Method **getContentPane** returns a reference to the JFrame's content pane (an object of class **Container**). In line 45, we then use that reference to call method **setBackground**, which sets the content pane's background color to an element in the colors array. The color is selected from the array by using the selected item's index. JList method **getSelectedIndex** returns the selected item's index. As with arrays and JComboBoxes, JList indexing is zero based.

26.13 Multiple-Selection Lists

A **multiple-selection list** enables the user to select many items from a JList (see the output of Fig. 26.26). A **SINGLE_INTERVAL_SELECTION** list allows selecting a contiguous range of items. To do so, click the first item, then press and hold the *Shift* key while clicking the last item in the range. A **MULTIPLE_INTERVAL_SELECTION** list (the default) allows continuous range selection as described for a **SINGLE_INTERVAL_SELECTION** list. Such a list also allows miscellaneous items to be selected by pressing and holding the *Ctrl* key while clicking each item to select. To *deselect* an item, press and hold the *Ctrl* key while clicking the item a second time.

The application of Figs. 26.25–26.26 uses multiple-selection lists to copy items from one JList to another. One list is a **MULTIPLE_INTERVAL_SELECTION** list and the other is a **SINGLE_INTERVAL_SELECTION** list. When you execute the application, try using the selection techniques described previously to select items in both lists.

```
1 // Fig. 26.25: MultipleSelectionFrame.java
2 // JList that allows multiple selections.
3 import java.awt.FlowLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JList;
8 import javax.swing.JButton;
9 import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14     private final JList<String> colorJList; // list to hold color names
15     private final JList<String> copyJList; // list to hold copied names
```

Fig. 26.25 | JList that allows multiple selections. (Part I of 2.)

```
16    private JButton copyJButton; // button to copy selected names
17    private static final String[] colorNames = {"Black", "Blue", "Cyan",
18        "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19        "Pink", "Red", "White", "Yellow"};
20
21    // MultipleSelectionFrame constructor
22    public MultipleSelectionFrame()
23    {
24        super("Multiple Selection Lists");
25        setLayout(new FlowLayout());
26
27        colorJList = new JList<String>(colorNames); // list of color names
28        colorJList.setVisibleRowCount(5); // show five rows
29        colorJList.setSelectionMode(
30            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
31        add(new JScrollPane(colorJList)); // add list with scrollpane
32
33        copyJButton = new JButton("Copy >>>");
34        copyJButton.addActionListener(
35            new ActionListener() // anonymous inner class
36            {
37                // handle button event
38                @Override
39                public void actionPerformed(ActionEvent event)
40                {
41                    // place selected values in copyJList
42                    copyJList.setListData(
43                        colorJList.getSelectedValuesList().toArray(
44                            new String[0]));
45                }
46            }
47        );
48
49        add(copyJButton); // add copy button to JFrame
50
51        copyJList = new JList<String>(); // list to hold copied color names
52        copyJList.setVisibleRowCount(5); // show 5 rows
53        copyJList.setFixedCellWidth(100); // set width
54        copyJList.setFixedCellHeight(15); // set height
55        copyJList.setSelectionMode(
56            ListSelectionModel.SINGLE_INTERVAL_SELECTION);
57        add(new JScrollPane(copyJList)); // add list with scrollpane
58    }
59 }
```

Fig. 26.25 | `JList` that allows multiple selections. (Part 2 of 2.)

```
1 // Fig. 26.26: MultipleSelectionTest.java
2 // Testing MultipleSelectionFrame.
3 import javax.swing.JFrame;
4
```

Fig. 26.26 | Testing `MultipleSelectionFrame`. (Part 1 of 2.)

```

5  public class MultipleSelectionTest
6  {
7      public static void main(String[] args)
8      {
9          MultipleSelectionFrame multipleSelectionFrame =
10             new MultipleSelectionFrame();
11             multipleSelectionFrame.setDefaultCloseOperation(
12                 JFrame.EXIT_ON_CLOSE);
13             multipleSelectionFrame.setSize(350, 150);
14             multipleSelectionFrame.setVisible(true);
15     }
16 }

```

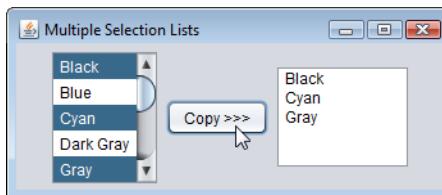


Fig. 26.26 | Testing `MultipleSelectionFrame`. (Part 2 of 2.)

Line 27 of Fig. 26.25 creates `JList colorJList` and initializes it with the `Strings` in the array `colorNames`. Line 28 sets the number of visible rows in `colorJList` to 5. Lines 29–30 specify that `colorJList` is a `MULTIPLE_INTERVAL_SELECTION` list. Line 31 adds a new `JScrollPane` containing `colorJList` to the `JFrame`. Lines 51–57 perform similar tasks for `copyJList`, which is declared as a `SINGLE_INTERVAL_SELECTION` list. If a `JList` does not contain items, it will not display in a `FlowLayout`. For this reason, lines 53–54 use `JList` methods `setFixedCellWidth` and `setFixedCellHeight` to set `copyJList`'s width to 100 pixels and the height of each item in the `JList` to 15 pixels, respectively.

Normally, an event generated by another GUI component (known as an **external event**) specifies when the multiple selections in a `JList` should be processed. In this example, the user clicks the `JButton` called `copyJButton` to trigger the event that copies the selected items in `colorJList` to `copyJList`.

Lines 34–47 declare, create and register an `ActionListener` for the `copyJButton`. When the user clicks `copyJButton`, method `actionPerformed` (lines 38–45) uses `JList` method `setListData` to set the items displayed in `copyJList`. Lines 43–44 call `colorJList`'s method `getSelectedValuesList`, which returns a `List<String>` (because the `JList` was created as a `JList<String>`) representing the selected items in `colorJList`. We call the `List<String>`'s `toArray` method to convert this into an array of `Strings` that can be passed as the argument to `copyJList`'s `setListData` method. `List` method `toArray` receives as its argument an array representing the type of array that the method will return. You'll learn more about `List` and `toArray` in Chapter 16.

You might be wondering why `copyJList` can be used in line 42 even though the application does not create the object to which it refers until line 49. Remember that method `actionPerformed` (lines 38–45) does not execute until the user presses the `copyJButton`, which cannot occur until after the constructor completes execution and the

application displays the GUI. At that point in the application's execution, `copyJList` is already initialized with a new `JList` object.

26.14 Mouse Event Handling

This section presents the `MouseListener` and `MouseMotionListener` event-listener interfaces for handling **mouse events**. Mouse events can be processed for any GUI component that derives from `java.awt.Component`. The methods of interfaces `MouseListener` and `MouseMotionListener` are summarized in Figure 26.27. Package `javax.swing.event` contains interface `MouseInputListener`, which extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all the `MouseListener` and `MouseMotionListener` methods. The `MouseListener` and `MouseMotionListener` methods are called when the mouse interacts with a `Component` if appropriate event-listener objects are registered for that `Component`.

MouseListener and MouseMotionListener interface methods

Methods of interface MouseListener

```
public void mousePressed(MouseEvent event)
```

Called when a mouse button is *pressed* while the mouse cursor is on a component.

```
public void mouseClicked(MouseEvent event)
```

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. Always preceded by a call to `mousePressed` and `mouseReleased`.

```
public void mouseReleased(MouseEvent event)
```

Called when a mouse button is *released after being pressed*. Always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

```
public void mouseEntered(MouseEvent event)
```

Called when the mouse cursor *enters* the bounds of a component.

```
public void mouseExited(MouseEvent event)
```

Called when the mouse cursor *leaves* the bounds of a component.

Methods of interface MouseMotionListener

```
public void mouseDragged(MouseEvent event)
```

Called when the mouse button is *pressed* while the mouse cursor is on a component and the mouse is *moved* while the mouse button *remains pressed*. Always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.

```
public void mouseMoved(MouseEvent event)
```

Called when the mouse is *moved* (with no mouse buttons pressed) when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

Fig. 26.27 | `MouseListener` and `MouseMotionListener` interface methods.

Each of the mouse event-handling methods receives as an argument a `MouseEvent` object that contains information about the mouse event that occurred, including the *x*- and *y*-coordinates of its location. These coordinates are measured from the *upper-left corner*

of the GUI component on which the event occurred. The *x*-coordinates start at 0 and *increase from left to right*. The *y*-coordinates start at 0 and *increase from top to bottom*. The methods and constants of class **InputEvent** (`MouseEvent`'s superclass) enable you to determine which mouse button the user clicked.



Software Engineering Observation 26.5

Calls to `mouseDragged` are sent to the `MouseMotionListener` for the Component on which the drag started. Similarly, the `mouseReleased` call at the end of a drag operation is sent to the `MouseListener` for the Component on which the drag operation started.

Java also provides interface **MouseWheelListener** to enable applications to respond to the *rotation of a mouse wheel*. This interface declares method **mouseWheelMoved**, which receives a **MouseWheelEvent** as its argument. Class `MouseWheelEvent` (a subclass of `MouseEvent`) contains methods that enable the event handler to obtain information about the amount of wheel rotation.

Tracking Mouse Events on a JPanel

The `MouseTracker` application (Figs. 26.28–26.29) demonstrates the `MouseListener` and `MouseMotionListener` interface methods. The event-handler class (lines 36–97 of Fig. 26.28) implements both interfaces. You *must* declare all seven methods from these two interfaces when your class implements them both. Each mouse event in this example displays a `String` in the `JLabel` called `statusBar` that is attached to the bottom of the window.

```

1 // Fig. 26.28: MouseTrackerFrame.java
2 // Mouse event handling.
3 import java.awt.Color;
4 import java.awt.BorderLayout;
5 import java.awt.event.MouseListener;
6 import java.awt.event.MouseMotionListener;
7 import java.awt.event.MouseEvent;
8 import javax.swing.JFrame;
9 import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14     private final JPanel mousePanel; // panel in which mouse events occur
15     private final JLabel statusBar; // displays event information
16
17     // MouseTrackerFrame constructor sets up GUI and
18     // registers mouse event handlers
19     public MouseTrackerFrame()
20     {
21         super("Demonstrating Mouse Events");
22
23         mousePanel = new JPanel();
24         mousePanel.setBackground(Color.WHITE);
25         add(mousePanel, BorderLayout.CENTER); // add panel to JFrame
26

```

Fig. 26.28 | Mouse event handling. (Part I of 3.)

```
27     statusBar = new JLabel("Mouse outside JPanel");
28     add(statusBar, BorderLayout.SOUTH); // add Label to JFrame
29
30     // create and register listener for mouse and mouse motion events
31     MouseHandler handler = new MouseHandler();
32     mousePanel.addMouseListener(handler);
33     mousePanel.addMouseMotionListener(handler);
34 }
35
36 private class MouseHandler implements MouseListener,
37     MouseMotionListener
38 {
39     // MouseListener event handlers
40     // handle event when mouse released immediately after press
41     @Override
42     public void mouseClicked(MouseEvent event)
43     {
44         statusBar.setText(String.format("Clicked at [%d, %d]",
45             event.getX(), event.getY()));
46     }
47
48     // handle event when mouse pressed
49     @Override
50     public void mousePressed(MouseEvent event)
51     {
52         statusBar.setText(String.format("Pressed at [%d, %d]",
53             event.getX(), event.getY()));
54     }
55
56     // handle event when mouse released
57     @Override
58     public void mouseReleased(MouseEvent event)
59     {
60         statusBar.setText(String.format("Released at [%d, %d]",
61             event.getX(), event.getY()));
62     }
63
64     // handle event when mouse enters area
65     @Override
66     public void mouseEntered(MouseEvent event)
67     {
68         statusBar.setText(String.format("Mouse entered at [%d, %d]",
69             event.getX(), event.getY()));
70         mousePanel.setBackground(Color.GREEN);
71     }
72
73     // handle event when mouse exits area
74     @Override
75     public void mouseExited(MouseEvent event)
76     {
77         statusBar.setText("Mouse outside JPanel");
78         mousePanel.setBackground(Color.WHITE);
79     }
}
```

Fig. 26.28 | Mouse event handling. (Part 2 of 3.)

```

80
81     // MouseMotionListener event handlers
82     // handle event when user drags mouse with button pressed
83     @Override
84     public void mouseDragged(MouseEvent event)
85     {
86         statusBar.setText(String.format("Dragged at [%d, %d]",
87             event.getX(), event.getY()));
88     }
89
90     // handle event when user moves mouse
91     @Override
92     public void mouseMoved(MouseEvent event)
93     {
94         statusBar.setText(String.format("Moved at [%d, %d]",
95             event.getX(), event.getY()));
96     }
97 } // end inner class MouseHandler
98 }
```

Fig. 26.28 | Mouse event handling. (Part 3 of 3.)

```

1 // Fig. 26.29: MouseTrackerFrame.java
2 // Testing MouseTrackerFrame.
3 import javax.swing.JFrame;
4
5 public class MouseTracker
6 {
7     public static void main(String[] args)
8     {
9         MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
10        mouseTrackerFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        mouseTrackerFrame.setSize(300, 100);
12        mouseTrackerFrame.setVisible(true);
13    }
14 }
```

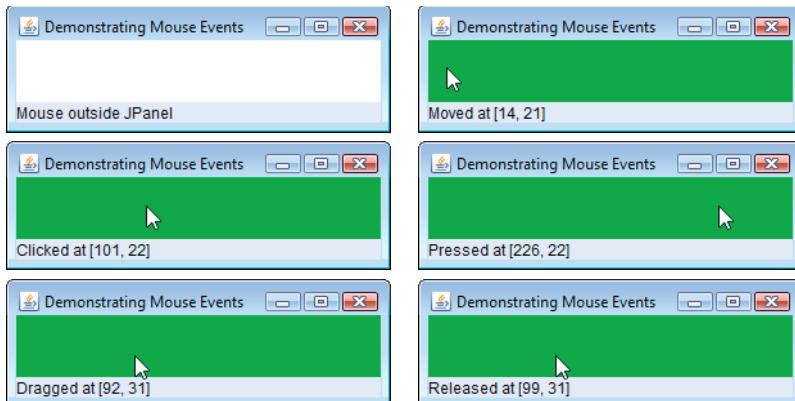


Fig. 26.29 | Testing MouseTrackerFrame.

Line 23 creates `JPanel mousePanel`. This `JPanel`'s mouse events are tracked by the app. Line 24 sets `mousePanel`'s background color to white. When the user moves the mouse into the `mousePanel`, the application will change `mousePanel`'s background color to green. When the user moves the mouse out of the `mousePanel`, the application will change the background color back to white. Line 25 attaches `mousePanel` to the `JFrame`. As you've learned, you typically must specify the layout of the GUI components in a `JFrame`. In that section, we introduced the layout manager `FlowLayout`. Here we use the default layout of a `JFrame`'s content pane—**BorderLayout**, which arranges component **NORTH**, **SOUTH**, **EAST**, **WEST** and **CENTER** regions. **NORTH** corresponds to the container's top. This example uses the **CENTER** and **SOUTH** regions. Line 25 uses a two-argument version of method `add` to place `mousePanel` in the **CENTER** region. The `BorderLayout` automatically sizes the component in the **CENTER** to use all the space in the `JFrame` that is not occupied by components in the other regions. Section 26.18.2 discusses `BorderLayout` in more detail.

Lines 27–28 in the constructor declare `JLabel statusBar` and attach it to the `JFrame`'s **SOUTH** region. This `JLabel` occupies the width of the `JFrame`. The region's height is determined by the `JLabel`.

Line 31 creates an instance of inner class `MouseHandler` (lines 36–97) called `handler` that responds to mouse events. Lines 32–33 register `handler` as the listener for `mousePanel`'s mouse events. Methods `addMouseListener` and `addMouseMotionListener` are inherited indirectly from class `Component` and can be used to register `MouseListener`s and `MouseMotionListener`s, respectively. A `MouseHandler` object is a `MouseListener` and is a `MouseMotionListener` because the class implements both interfaces. We chose to implement both interfaces here to demonstrate a class that implements more than one interface, but we could have implemented interface `MouseListener` instead.

When the mouse enters and exits `mousePanel`'s area, methods `mouseEntered` (lines 65–71) and `mouseExited` (lines 74–79) are called, respectively. Method `mouseEntered` displays a message in the `statusBar` indicating that the mouse entered the `JPanel` and changes the background color to green. Method `mouseExited` displays a message in the `statusBar` indicating that the mouse is outside the `JPanel` (see the first sample output window) and changes the background color to white.

The other five events display a string in the `statusBar` that includes the event and the coordinates at which it occurred. `MouseEvent` methods `getX` and `getY` return the *x*- and *y*-coordinates, respectively, of the mouse at the time the event occurred.

26.15 Adapter Classes

Many event-listener interfaces, such as `MouseListener` and `MouseMotionListener`, contain multiple methods. It's not always desirable to declare every method in an event-listener interface. For example, an application may need only the `mouseClicked` handler from `MouseListener` or the `mouseDragged` handler from `MouseMotionListener`. Interface `WindowListener` specifies seven window event-handling methods. For many of the listener interfaces that have multiple methods, packages `java.awt.event` and `javax.swing.event` provide event-listener adapter classes. An **adapter class** implements an interface and provides a default implementation (with an empty method body) of each method in the interface. Figure 26.30 shows several `java.awt.event` adapter classes and the interfaces they

implement. You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.



Software Engineering Observation 26.6

When a class implements an interface, the class has an is-a relationship with that interface. All direct and indirect subclasses of that class inherit this interface. Thus, an object of a class that extends an event-adapter class is an object of the corresponding event-listener type (e.g., an object of a subclass of MouseAdapter is a MouseListener).

Event-adapter class in <code>java.awt.event</code>	Implements interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Fig. 26.30 | Event-adapter classes and the interfaces they implement.

Extending MouseAdapter

The application of Figs. 26.31–26.32 demonstrates how to determine the number of mouse clicks (i.e., the click count) and how to distinguish between the different mouse buttons. The event listener in this application is an object of inner class `MouseClickHandler` (Fig. 26.31, lines 25–46) that extends `MouseAdapter`, so we can declare just the `mouseClicked` method we need in this example.



Common Programming Error 26.3

If you extend an adapter class and misspell the name of the method you're overriding, and you do not declare the method with `@Override`, your method simply becomes another method in the class. This is a logic error that is difficult to detect, since the program will call the empty version of the method inherited from the adapter class.

```

1 // Fig. 26.31: MouseDetailsFrame.java
2 // Demonstrating mouse clicks and distinguishing between mouse buttons.
3 import java.awt.BorderLayout;
4 import java.awt.event.MouseAdapter;
5 import java.awt.event.MouseEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8
9 public class MouseDetailsFrame extends JFrame
10 {
11     private String details; // String displayed in the statusBar
12     private final JLabel statusBar; // JLabel at bottom of window
13 }
```

Fig. 26.31 | Demonstrating mouse clicks and distinguishing between mouse buttons. (Part 1 of 2.)

```

14 // constructor sets title bar String and register mouse listener
15 public MouseDetailsFrame()
16 {
17     super("Mouse clicks and buttons");
18
19     statusBar = new JLabel("Click the mouse");
20     add(statusBar, BorderLayout.SOUTH);
21     addMouseListener(new MouseClickHandler()); // add handler
22 }
23
24 // inner class to handle mouse events
25 private class MouseClickHandler extends MouseAdapter
26 {
27     // handle mouse-click event and determine which button was pressed
28     @Override
29     public void mouseClicked(MouseEvent event)
30     {
31         int xPos = event.getX(); // get x-position of mouse
32         int yPos = event.getY(); // get y-position of mouse
33
34         details = String.format("Clicked %d time(s)",
35             event.getClickCount());
36
37         if (event.isMetaDown()) // right mouse button
38             details += " with right mouse button";
39         else if (event.isAltDown()) // middle mouse button
40             details += " with center mouse button";
41         else // left mouse button
42             details += " with left mouse button";
43
44         statusBar.setText(details); // display message in statusBar
45     }
46 }
47 }

```

Fig. 26.31 | Demonstrating mouse clicks and distinguishing between mouse buttons. (Part 2 of 2.)

```

1 // Fig. 26.32: MouseDetails.java
2 // Testing MouseDetailsFrame.
3 import javax.swing.JFrame;
4
5 public class MouseDetails
6 {
7     public static void main(String[] args)
8     {
9         MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10        mouseDetailsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        mouseDetailsFrame.setSize(400, 150);
12        mouseDetailsFrame.setVisible(true);
13    }
14 }

```

Fig. 26.32 | Testing MouseDetailsFrame. (Part 1 of 2.)

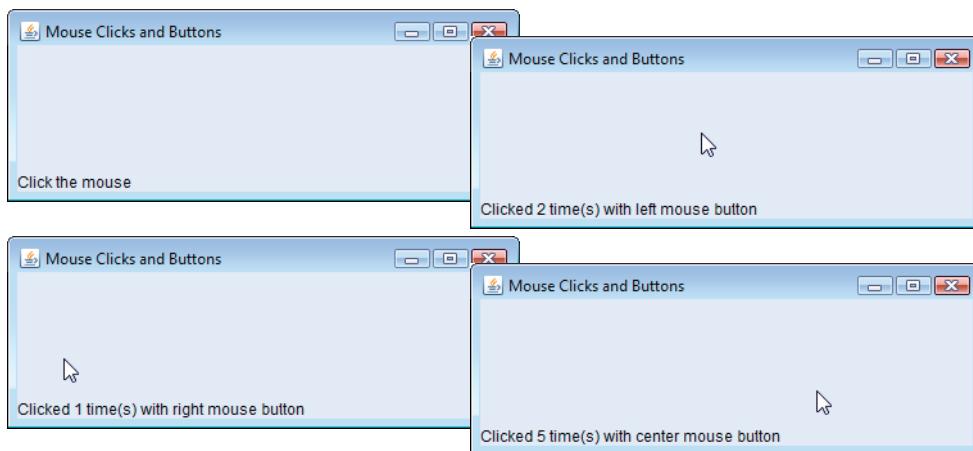


Fig. 26.32 | Testing `MouseDetailsFrame`. (Part 2 of 2.)

A user of a Java application may be on a system with a one-, two- or three-button mouse. Class `MouseEvent` inherits several methods from class `InputEvent` that can distinguish among mouse buttons on a multibutton mouse or can mimic a multibutton mouse with a combined keystroke and mouse-button click. Figure 26.33 shows the `InputEvent` methods used to distinguish among mouse-button clicks. Java assumes that every mouse contains a left mouse button. Thus, it's simple to test for a left-mouse-button click. However, users with a one- or two-button mouse must use a combination of keystrokes and mouse-button clicks at the same time to simulate the missing buttons on the mouse. In the case of a one- or two-button mouse, a Java application assumes that the center mouse button is clicked if the user holds down the *Alt* key and clicks the left mouse button on a two-button mouse or the only mouse button on a one-button mouse. In the case of a one-button mouse, a Java application assumes that the right mouse button is clicked if the user holds down the *Meta* key (sometimes called the *Command* key or the “Apple” key on a Mac) and clicks the mouse button.

InputEvent method	Description
<code>isMetaDown()</code>	Returns <code>true</code> when the user clicks the <i>right mouse button</i> on a mouse with two or three buttons. To simulate a right-mouse-button click on a one-button mouse, the user can hold down the <i>Meta</i> key on the keyboard and click the mouse button.
<code>isAltDown()</code>	Returns <code>true</code> when the user clicks the <i>middle mouse button</i> on a mouse with three buttons. To simulate a middle-mouse-button click on a one- or two-button mouse, the user can press the <i>Alt</i> key and click the only or left mouse button, respectively.

Fig. 26.33 | `InputEvent` methods that help determine whether the right or center mouse button was clicked.

Line 21 of Fig. 26.31 registers a `MouseListener` for the `MouseDetailsFrame`. The event listener is an object of class `MouseClickHandler`, which extends `MouseAdapter`. This enables us to declare only method `mouseClicked` (lines 28–45). This method first captures the coordinates where the event occurred and stores them in local variables `xPos` and `yPos` (lines 31–32). Lines 34–35 create a `String` called `details` containing the number of consecutive mouse clicks, which is returned by `MouseEvent` method `getClickCount` at line 35. Lines 37–42 use methods `isMetaDown` and `isAltDown` to determine which mouse button the user clicked and append an appropriate `String` to `details` in each case. The resulting `String` is displayed in the `statusBar`. Class `MouseDetails` (Fig. 26.32) contains the `main` method that executes the application. Try clicking with each of your mouse's buttons repeatedly to see the click count increment.

26.16 JPanel Subclass for Drawing with the Mouse

Section 26.14 showed how to track mouse events in a `JPanel`. In this section, we use a `JPanel` as a **dedicated drawing area** in which the user can draw by dragging the mouse. In addition, this section demonstrates an event listener that extends an adapter class.

Method `paintComponent`

Lightweight Swing components that extend class `JComponent` (such as `JPanel`) contain method `paintComponent`, which is called when a lightweight Swing component is displayed. By overriding this method, you can specify how to draw shapes using Java's graphics capabilities. When customizing a `JPanel` for use as a dedicated drawing area, the subclass should override method `paintComponent` and call the superclass version of `paintComponent` as the first statement in the body of the overridden method to ensure that the component displays correctly. The reason is that subclasses of `JComponent` support **transparency**. To display a component correctly, the program must determine whether the component is transparent. The code that determines this is in superclass `JComponent`'s `paintComponent` implementation. When a component is transparent, `paintComponent` will not clear its background when the program displays the component. When a component is **opaque**, `paintComponent` clears the component's background before the component is displayed. The transparency of a Swing lightweight component can be set with method `setOpaque` (a `false` argument indicates that the component is transparent).



Error-Prevention Tip 26.1

In a `JComponent` subclass's `paintComponent` method, the first statement should always call the superclass's `paintComponent` method to ensure that an object of the subclass displays correctly.



Common Programming Error 26.4

If an overridden `paintComponent` method does not call the superclass's version, the subclass component may not display properly. If an overridden `paintComponent` method calls the superclass's version after other drawing is performed, the drawing will be erased.

Defining the Custom Drawing Area

The Painter application of Figs. 26.34–26.35 demonstrates a customized subclass of `JPanel` that's used to create a dedicated drawing area. The application uses the `mouse-`

Dragged event handler to create a simple drawing application. The user can draw pictures by dragging the mouse on the JPanel. This example does not use method `mouseMoved`, so our *event-listener class* (the *anonymous inner class* at lines 20–29 of Fig. 26.34) extends `MouseMotionAdapter`. Since this class already declares both `mouseMoved` and `mouseDragged`, we can simply override `mouseDragged` to provide the event handling this application requires.

```
1 // Fig. 26.34: PaintPanel.java
2 // Adapter class used to implement event handlers.
3 import java.awt.Point;
4 import java.awt.Graphics;
5 import java.awt.event.MouseEvent;
6 import java.awt.event.MouseMotionAdapter;
7 import java.util.ArrayList;
8 import javax.swing.JPanel;
9
10 public class PaintPanel extends JPanel
11 {
12     // list of Point references
13     private final ArrayList<Point> points = new ArrayList<>();
14
15     // set up GUI and register mouse event handler
16     public PaintPanel()
17     {
18         // handle frame mouse motion event
19         addMouseMotionListener(
20             new MouseMotionAdapter() // anonymous inner class
21             {
22                 // store drag coordinates and repaint
23                 @Override
24                 public void mouseDragged(MouseEvent event)
25                 {
26                     points.add(event.getPoint());
27                     repaint(); // repaint JFrame
28                 }
29             }
30         );
31     }
32
33     // draw ovals in a 4-by-4 bounding box at specified locations on window
34     @Override
35     public void paintComponent(Graphics g)
36     {
37         super.paintComponent(g); // clears drawing area
38
39         // draw all points
40         for (Point point : points)
41             g.fillOval(point.x, point.y, 4, 4);
42     }
43 }
```

Fig. 26.34 | Adapter class used to implement event handlers.

Class `PaintPanel` (Fig. 26.34) extends `JPanel` to create the dedicated drawing area. Class `Point` (package `java.awt`) represents an *x-y* coordinate. We use objects of this class to store the coordinates of each mouse drag event. Class `Graphics` is used to draw. In this example, we use an `ArrayList` of `Points` (line 13) to store the location at which each mouse drag event occurs. As you'll see, method `paintComponent` uses these `Points` to draw.

Lines 19–30 register a `MouseMotionListener` to listen for the `PaintPanel`'s mouse motion events. Lines 20–29 create an object of an anonymous inner class that extends the adapter class `MouseMotionAdapter`. Recall that `MouseMotionAdapter` implements `MouseMotionListener`, so the *anonymous inner class* object is a `MouseMotionListener`. The anonymous inner class inherits default `mouseMoved` and `mouseDragged` implementations, so it already implements all the interface's methods. However, the default implementations do nothing when they're called. So, we override method `mouseDragged` at lines 23–28 to capture the coordinates of a mouse drag event and store them as a `Point` object. Line 26 invokes the `MouseEvent`'s `getPoint` method to obtain the `Point` where the event occurred and stores it in the `ArrayList`. Line 27 calls method `repaint` (inherited indirectly from class `Component`) to indicate that the `PaintPanel` should be refreshed on the screen as soon as possible with a call to the `PaintPanel`'s `paintComponent` method.

Method `paintComponent` (lines 34–42), which receives a `Graphics` parameter, is called automatically any time the `PaintPanel` needs to be displayed on the screen—such as when the GUI is first displayed—or refreshed on the screen—such as when method `repaint` is called or when the GUI component has been *hidden* by another window on the screen and subsequently becomes visible again.



Look-and-Feel Observation 26.13

Calling `repaint` for a Swing GUI component indicates that the component should be refreshed on the screen as soon as possible. The component's background is cleared only if the component is opaque. `JComponent` method `setOpaque` can be passed a boolean argument indicating whether the component is opaque (true) or transparent (false).

Line 37 invokes the superclass version of `paintComponent` to clear the `PaintPanel`'s background (`JPanels` are opaque by default). Lines 40–41 draw an oval at the location specified by each `Point` in the `ArrayList`. `Graphics` method `fillOval` draws a solid oval. The method's four parameters represent a rectangular area (called the *bounding box*) in which the oval is displayed. The first two parameters are the upper-left *x*-coordinate and the upper-left *y*-coordinate of the rectangular area. The last two coordinates represent the rectangular area's width and height. Method `fillOval` draws the oval so it touches the middle of each side of the rectangular area. In line 41, the first two arguments are specified by using class `Point`'s two `public` instance variables—*x* and *y*. You'll learn more `Graphics` features in Chapter 27.



Look-and-Feel Observation 26.14

Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component, not the upper-left corner of the screen.

Using the Custom `JPanel` in an Application

Class `Painter` (Fig. 26.35) contains the `main` method that executes this application. Line 14 creates a `PaintPanel` object on which the user can drag the mouse to draw. Line 15 attaches the `PaintPanel` to the `JFrame`.

```
1 // Fig. 26.35: Painter.java
2 // Testing PaintPanel.
3 import java.awt.BorderLayout;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6
7 public class Painter
8 {
9     public static void main(String[] args)
10    {
11        // create JFrame
12        JFrame application = new JFrame("A simple paint program");
13
14        PaintPanel paintPanel = new PaintPanel();
15        application.add(paintPanel, BorderLayout.CENTER);
16
17        // create a label and place it in SOUTH of BorderLayout
18        application.add(new JLabel("Drag the mouse to draw"),
19                        BorderLayout.SOUTH);
20
21        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22        application.setSize(400, 200);
23        application.setVisible(true);
24    }
25 }
```

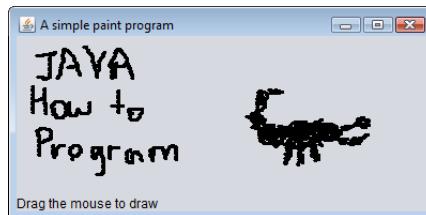


Fig. 26.35 | Testing PaintPanel.

26.17 Key Event Handling

This section presents the `KeyListener` interface for handling **key events**. Key events are generated when keys on the keyboard are pressed and released. A class that implements `KeyListener` must provide declarations for methods `keyPressed`, `keyReleased` and `keyTyped`, each of which receives a `KeyEvent` as its argument. Class `KeyEvent` is a subclass of `InputEvent`. Method `keyPressed` is called in response to pressing any key. Method `keyTyped` is called in response to pressing any key that is not an **action key**. (The action keys are any arrow key, *Home*, *End*, *Page Up*, *Page Down*, any function key, etc.) Method `keyReleased` is called when the key is released after any `keyPressed` or `keyTyped` event.

The application of Figs. 26.36–26.37 demonstrates the `KeyListener` methods. Class `KeyDemoFrame` implements the `KeyListener` interface, so all three methods are declared in the application. The constructor (Fig. 26.36, lines 17–28) registers the application to handle its own key events by using method `addKeyListener` at line 27. Method `addKey-`

Listener is declared in class Component, so every subclass of Component can notify KeyListener objects of key events for that Component.

```
1 // Fig. 26.36: KeyDemoFrame.java
2 // Key event handling.
3 import java.awt.Color;
4 import java.awt.event.KeyListener;
5 import java.awt.event.KeyEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;
8
9 public class KeyDemoFrame extends JFrame implements KeyListener
10 {
11     private final String line1 = ""; // first line of text area
12     private final String line2 = ""; // second line of text area
13     private final String line3 = ""; // third line of text area
14     private final JTextArea textArea; // text area to display output
15
16     // KeyDemoFrame constructor
17     public KeyDemoFrame()
18     {
19         super("Demonstrating Keystroke Events");
20
21         textArea = new JTextArea(10, 15); // set up JTextArea
22         textArea.setText("Press any key on the keyboard...");
23         textArea.setEnabled(false);
24         textArea.setDisabledTextColor(Color.BLACK);
25         add(textArea); // add text area to JFrame
26
27         addKeyListener(this); // allow frame to process key events
28     }
29
30     // handle press of any key
31     @Override
32     public void keyPressed(KeyEvent event)
33     {
34         line1 = String.format("Key pressed: %s",
35             KeyEvent.getKeyText(event.getKeyCode())); // show pressed key
36         setLines2and3(event); // set output lines two and three
37     }
38
39     // handle release of any key
40     @Override
41     public void keyReleased(KeyEvent event)
42     {
43         line1 = String.format("Key released: %s",
44             KeyEvent.getKeyText(event.getKeyCode())); // show released key
45         setLines2and3(event); // set output lines two and three
46     }
47 }
```

Fig. 26.36 | Key event handling. (Part I of 2.)

```

48     // handle press of an action key
49     @Override
50     public void keyTyped(KeyEvent event)
51     {
52         line1 = String.format("Key typed: %s", event.getKeyChar());
53         setLines2and3(event); // set output lines two and three
54     }
55
56     // set second and third lines of output
57     private void setLines2and3(KeyEvent event)
58     {
59         line2 = String.format("This key is %san action key",
60             (event.isActionKey() ? "" : "not "));
61
62         String temp = KeyEvent.getKeyModifiersText(event.getModifiers());
63
64         line3 = String.format("Modifier keys pressed: %s",
65             (temp.equals("") ? "none" : temp)); // output modifiers
66
67         textArea.setText(String.format("%s\n%s\n%s\n",
68             line1, line2, line3)); // output three lines of text
69     }
70 }
```

Fig. 26.36 | Key event handling. (Part 2 of 2.)

At line 25, the constructor adds the `JTextArea` `textArea` (where the application's output is displayed) to the `JFrame`. A `JTextArea` is a *multiline area* in which you can display text. (We discuss `JTextAreas` in more detail in Section 26.20.) Notice in the screen captures that `textArea` occupies the *entire window*. This is due to the `JFrame`'s default `BorderLayout` (discussed in Section 26.18.2 and demonstrated in Fig. 26.41). When a single Component is added to a `BorderLayout`, the Component occupies the *entire Container*. Line 23 disables the `JTextArea` so the user cannot type in it. This causes the text in the `JTextArea` to become gray. Line 24 uses method `setDisabledTextColor` to change the text color in the `JTextArea` to black for readability.

```

1 // Fig. 26.37: KeyDemo.java
2 // Testing KeyDemoFrame.
3 import javax.swing.JFrame;
4
5 public class KeyDemo
6 {
7     public static void main(String[] args)
8     {
9         KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
10        keyDemoFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        keyDemoFrame.setSize(350, 100);
12        keyDemoFrame.setVisible(true);
13    }
14 }
```

Fig. 26.37 | Testing `KeyDemoFrame`. (Part 1 of 2.)

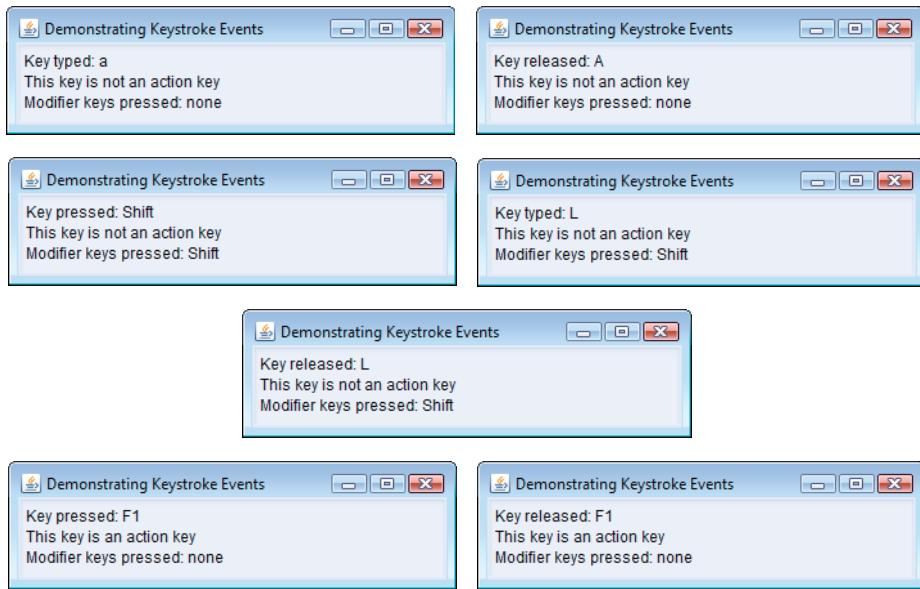


Fig. 26.37 | Testing KeyDemoFrame. (Part 2 of 2.)

Methods `keyPressed` (lines 31–37) and `keyReleased` (lines 40–46) use `KeyEvent` method `getKeyCode` to get the **virtual key code** of the pressed key. Class `KeyEvent` contains virtual key-code constants that represent every key on the keyboard. These constants can be compared with `getKeyCode`'s return value to test for individual keys on the keyboard. The value returned by `getKeyCode` is passed to static `KeyEvent` method `getKeyText`, which returns a string containing the name of the key that was pressed. For a complete list of virtual key constants, see the online documentation for class `KeyEvent` (package `java.awt.event`). Method `keyTyped` (lines 49–54) uses `KeyEvent` method `getKeyChar` (which returns a `char`) to get the Unicode value of the character typed.

All three event-handling methods finish by calling method `setLines2and3` (lines 57–69) and passing it the `KeyEvent` object. This method uses `KeyEvent` method `isActionKey` (line 60) to determine whether the key in the event was an action key. Also, `InputEvent` method `getModifiers` is called (line 62) to determine whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred. The result of this method is passed to static `KeyEvent` method `getKeyModifiersText`, which produces a `String` containing the names of the pressed modifier keys.

[*Note:* If you need to test for a specific key on the keyboard, class `KeyEvent` provides a **key constant** for each one. These constants can be used from the key event handlers to determine whether a particular key was pressed. Also, to determine whether the *Alt*, *Ctrl*, *Meta* and *Shift* keys are pressed individually, `InputEvent` methods `isAltDown`, `isControlDown`, `isMetaDown` and `isShiftDown` each return a `boolean` indicating whether the particular key was pressed during the key event.]

26.18 Introduction to Layout Managers

Layout managers arrange GUI components in a container for presentation purposes. You can use the layout managers for basic layout capabilities instead of determining every GUI component's exact position and size. This functionality enables you to concentrate on the basic look-and-feel and lets the layout managers process most of the layout details. All layout managers implement the interface **LayoutManager** (in package `java.awt`). Class `Container`'s `setLayout` method takes an object that implements the `LayoutManager` interface as an argument. There are basically three ways for you to arrange components in a GUI:

1. *Absolute positioning*: This provides the greatest level of control over a GUI's appearance. By setting a Container's layout to `null`, you can specify the *absolute position of each GUI component* with respect to the upper-left corner of the Container by using Component methods `setSize` and `setLocation` or `setBounds`. If you do this, you also must specify each GUI component's size. Programming a GUI with absolute positioning can be tedious, unless you have an integrated development environment (IDE) that can generate the code for you.
2. *Layout managers*: Using layout managers to position elements can be simpler and faster than creating a GUI with absolute positioning, and makes your GUIs more resizable, but you lose some control over the size and the precise positioning of each component.
3. *Visual programming in an IDE*: IDEs provide tools that make it easy to create GUIs. Each IDE typically provides a **GUI design tool** that allows you to drag and drop GUI components from a tool box onto a design area. You can then position, size and align GUI components as you like. The IDE generates the Java code that creates the GUI. In addition, you can typically add event-handling code for a particular component by double-clicking the component. Some design tools also allow you to use the layout managers described in this chapter and in Chapter 35.



Look-and-Feel Observation 26.15

Most Java IDEs provide GUI design tools for visually designing a GUI; the tools then write Java code that creates the GUI. Such tools often provide greater control over the size, position and alignment of GUI components than do the built-in layout managers.



Look-and-Feel Observation 26.16

It's possible to set a Container's layout to `null`, which indicates that no layout manager should be used. In a Container without a layout manager, you must position and size the components and take care that, on resize events, all components are repositioned as necessary. A component's resize events can be processed by a `ComponentListener`.

Figure 26.38 summarizes the layout managers presented in this chapter. A couple of additional layout managers are discussed in Chapter 35.

Layout manager	Description
FlowLayout	Default for javax.swing.JPanel. Places components <i>sequentially, left to right</i> , in the order they were added. It's also possible to specify the order of the components by using the Container method add, which takes a Component and an integer index position as arguments.
BorderLayout	Default for JFrames (and other windows). Arranges the components into five areas: NORTH, SOUTH, EAST, WEST and CENTER.
GridLayout	Arranges the components into rows and columns.

Fig. 26.38 | Layout managers.

26.18.1 FlowLayout

FlowLayout is the *simpler* layout manager. GUI components are placed in a container from left to right in the order in which they're added to the container. When the edge of the container is reached, components continue to display on the next line. Class FlowLayout allows GUI components to be *left aligned, centered* (the default) and *right aligned*.

The application of Figs. 26.39–26.40 creates three JButton objects and adds them to the application, using a FlowLayout. The components are center aligned by default. When the user clicks **Left**, the FlowLayout's alignment is changed to left aligned. When the user clicks **Right**, the FlowLayout's alignment is changed to right aligned. When the user clicks **Center**, the FlowLayout's alignment is changed to center aligned. The sample output windows show each alignment. The last sample output shows the centered alignment after the window has been resized to a smaller width so that the button **Right** flows onto a new line.

As seen previously, a container's layout is set with method `setLayout` of class Container. Line 25 (Fig. 26.39) sets the layout manager to the FlowLayout declared at line 23. Normally, the layout is set before any GUI components are added to a container.



Look-and-Feel Observation 26.17

Each individual container can have only one layout manager, but multiple containers in the same application can each use different layout managers.

```

1 // Fig. 26.39: FlowLayoutFrame.java
2 // FlowLayout allows components to flow over multiple lines.
3 import java.awt.FlowLayout;
4 import java.awt.Container;
5 import java.awt.event.ActionListener;
6 import java.awt.event.ActionEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12     private final JButton left JButton; // button to set alignment left
13     private final JButton center JButton; // button to set alignment center

```

Fig. 26.39 | FlowLayout allows components to flow over multiple lines. (Part I of 3.)

```
14    private final JButton rightJButton; // button to set alignment right
15    private final FlowLayout layout; // Layout object
16    private final Container container; // container to set layout
17
18    // set up GUI and register button listeners
19    public FlowLayoutFrame()
20    {
21        super("FlowLayout Demo");
22
23        layout = new FlowLayout();
24        container = getContentPane(); // get container to layout
25        setLayout(layout);
26
27        // set up leftJButton and register listener
28        leftJButton = new JButton("Left");
29        add(leftJButton); // add Left button to frame
30        leftJButton.addActionListener(
31            new ActionListener() // anonymous inner class
32            {
33                // process leftJButton event
34                @Override
35                public void actionPerformed(ActionEvent event)
36                {
37                    layout.setAlignment(FlowLayout.LEFT);
38
39                    // realign attached components
40                    layout.layoutContainer(container);
41                }
42            }
43        );
44
45        // set up centerJButton and register listener
46        centerJButton = new JButton("Center");
47        add(centerJButton); // add Center button to frame
48        centerJButton.addActionListener(
49            new ActionListener() // anonymous inner class
50            {
51                // process centerJButton event
52                @Override
53                public void actionPerformed(ActionEvent event)
54                {
55                    layout.setAlignment(FlowLayout.CENTER);
56
57                    // realign attached components
58                    layout.layoutContainer(container);
59                }
60            }
61        );
62
63        // set up rightJButton and register listener
64        rightJButton = new JButton("Right");
65        add(rightJButton); // add Right button to frame
```

Fig. 26.39 | FlowLayout allows components to flow over multiple lines. (Part 2 of 3.)

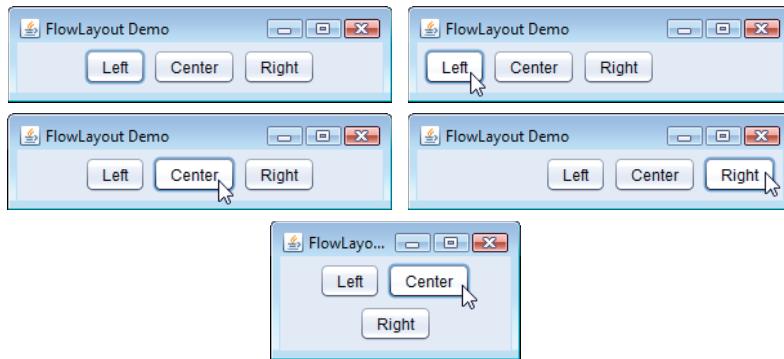
```

66     rightJButton.addActionListener(
67         new ActionListener() // anonymous inner class
68     {
69         // process rightJButton event
70         @Override
71         public void actionPerformed(ActionEvent event)
72         {
73             layout.setAlignment(FlowLayout.RIGHT);
74
75             // realign attached components
76             layout.layoutContainer(container);
77         }
78     });
79 );
80 } // end FlowLayoutFrame constructor
81 }
```

Fig. 26.39 | FlowLayout allows components to flow over multiple lines. (Part 3 of 3.)

```

1 // Fig. 26.40: FlowLayoutDemo.java
2 // Testing FlowLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class FlowLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
10        flowLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        flowLayoutFrame.setSize(300, 75);
12        flowLayoutFrame.setVisible(true);
13    }
14 }
```

**Fig. 26.40** | Testing FlowLayoutFrame.

Each button's event handler is specified with a separate anonymous inner-class object (lines 30–43, 48–61 and 66–79, respectively), and method `actionPerformed` in each case

executes two statements. For example, line 37 in the event handler for `leftJButton` uses `FlowLayout` method `setAlignment` to change the alignment for the `FlowLayout` to a left-aligned (`FlowLayout.LEFT`) `FlowLayout`. Line 40 uses `LayoutManager` interface method `layoutContainer` (which is inherited by all layout managers) to specify that the `JFrame` should be rearranged based on the adjusted layout. According to which button was clicked, the `actionPerformed` method for each button sets the `FlowLayout`'s alignment to `FlowLayout.LEFT` (line 37), `FlowLayout.CENTER` (line 55) or `FlowLayout.RIGHT` (line 73).

26.18.2 BorderLayout

The `BorderLayout` layout manager (the default layout manager for a `JFrame`) arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER. NORTH corresponds to the top of the container. Class `BorderLayout` extends `Object` and implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).

A `BorderLayout` limits a `Container` to containing *at most five components*—one in each region. The component placed in each region can be a container to which other components are attached. The components placed in the NORTH and SOUTH regions extend horizontally to the sides of the container and are as tall as the components placed in those regions. The EAST and WEST regions expand vertically between the NORTH and SOUTH regions and are as wide as the components placed in those regions. The component placed in the CENTER region *expands to fill all remaining space in the layout* (which is the reason the `JTextArea` in Fig. 26.37 occupies the entire window). If all five regions are occupied, the entire container's space is covered by GUI components. If the NORTH or SOUTH region is not occupied, the GUI components in the EAST, CENTER and WEST regions expand vertically to fill the remaining space. If the EAST or WEST region is not occupied, the GUI component in the CENTER region *expands horizontally to fill the remaining space*. If the CENTER region is *not occupied*, the area is left *empty*—the other GUI components do *not* expand to fill the remaining space. The application of Figs. 26.41–26.42 demonstrates the `BorderLayout` layout manager by using five `JButtons`.

```
1 // Fig. 26.41: BorderLayoutFrame.java
2 // BorderLayout containing five buttons.
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionListener;
5 import java.awt.event.ActionEvent;
6 import javax.swing.JFrame;
7 import javax.swing.JButton;
8
9 public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11     private final JButton[] buttons; // array of buttons to hide portions
12     private static final String[] names = {"Hide North", "Hide South",
13                                         "Hide East", "Hide West", "Hide Center"};
14     private final BorderLayout layout;
15 }
```

Fig. 26.41 | `BorderLayout` containing five buttons. (Part I of 2.)

```
16 // set up GUI and event handling
17 public BorderLayoutFrame()
18 {
19     super("BorderLayout Demo");
20
21     layout = new BorderLayout(5, 5); // 5 pixel gaps
22     setLayout(layout);
23     buttons = new JButton[names.length];
24
25     // create JButtons and register listeners for them
26     for (int count = 0; count < names.length; count++)
27     {
28         buttons[count] = new JButton(names[count]);
29         buttons[count].addActionListener(this);
30     }
31
32     add(buttons[0], BorderLayout.NORTH);
33     add(buttons[1], BorderLayout.SOUTH);
34     add(buttons[2], BorderLayout.EAST);
35     add(buttons[3], BorderLayout.WEST);
36     add(buttons[4], BorderLayout.CENTER);
37 }
38
39 // handle button events
40 @Override
41 public void actionPerformed(ActionEvent event)
42 {
43     // check event source and lay out content pane correspondingly
44     for (JButton button : buttons)
45     {
46         if (event.getSource() == button)
47             button.setVisible(false); // hide the button that was clicked
48         else
49             button.setVisible(true); // show other buttons
50     }
51
52     layout.layoutContainer(getContentPane()); // Lay out content pane
53 }
54 }
```

Fig. 26.41 | BorderLayout containing five buttons. (Part 2 of 2.)

Line 21 of Fig. 26.41 creates a `BorderLayout`. The constructor arguments specify the number of pixels between components that are arranged horizontally (**horizontal gap space**) and between components that are arranged vertically (**vertical gap space**), respectively. The default is one pixel of gap space horizontally and vertically. Line 22 uses method `setLayout` to set the content pane's layout to `layout`.

We add Components to a `BorderLayout` with another version of `Container` method `add` that takes two arguments—the Component to add and the region in which the Component should appear. For example, line 32 specifies that `buttons[0]` should appear in the `NORTH` region. The components can be added in *any* order, but only *one* component should be added to each region.



Look-and-Feel Observation 26.18

If no region is specified when adding a Component to a BorderLayout, the layout manager assumes that the Component should be added to region BorderLayout.CENTER.



Common Programming Error 26.5

When more than one component is added to a region in a BorderLayout, only the last component added to that region will be displayed. There's no error that indicates this problem.

Class BorderLayoutFrame implements ActionListener directly in this example, so the BorderLayoutFrame will handle the events of the JButtons. For this reason, line 29 passes the `this` reference to the `addActionListener` method of each JButton. When the user clicks a particular JButton in the layout, method `actionPerformed` (lines 40–53) executes. The enhanced `for` statement at lines 44–50 uses an `if...else` to hide the particular JButton that generated the event. Method `setVisible` (inherited into JButton from class Component) is called with a `false` argument (line 47) to hide the JButton. If the current JButton in the array is not the one that generated the event, method `setVisible` is called with a `true` argument (line 49) to ensure that the JButton is displayed on the screen. Line 52 uses LayoutManager method `layoutContainer` to recalculate the layout of the content pane. Notice in the screen captures of Fig. 26.42 that certain regions in the BorderLayout change shape as JButton are *hidden* and displayed in other regions. Try resizing the application window to see how the various regions resize based on the window's width and height. *For more complex layouts, group components in JPanel s, each with a separate layout manager.* Place the JPanel s on the JFrame using either the default BorderLayout or some other layout.

```
1 // Fig. 26.42: BorderLayoutDemo.java
2 // Testing BorderLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class BorderLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10        borderLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        borderLayoutFrame.setSize(300, 200);
12        borderLayoutFrame.setVisible(true);
13    }
14 }
```

Fig. 26.42 | Testing BorderLayoutFrame. (Part I of 2.)

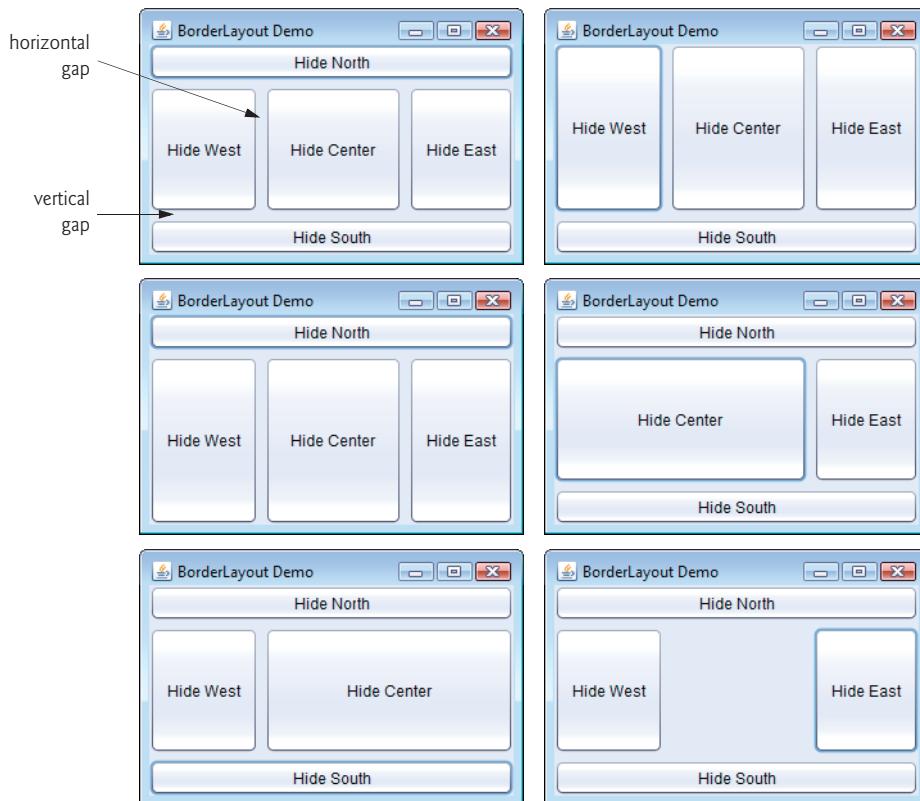


Fig. 26.42 | Testing BorderLayoutFrame. (Part 2 of 2.)

26.18.3 GridLayout

The **GridLayout** layout manager divides the container into a *grid* so that components can be placed in *rows* and *columns*. Class **GridLayout** inherits directly from class **Object** and implements interface **LayoutManager**. Every **Component** in a **GridLayout** has the *same* width and height. Components are added to a **GridLayout** starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on. The application of Figs. 26.43–26.44 demonstrates the **GridLayout** layout manager by using six **JButton**s.

```

1 // Fig. 26.43: GridLayoutFrame.java
2 // GridLayout containing six buttons.
3 import java.awt.GridLayout;
4 import java.awt.Container;
5 import java.awt.event.ActionListener;
6 import java.awt.event.ActionEvent;
7 import javax.swing.JFrame;
8 import javax.swing.JButton;
```

Fig. 26.43 | GridLayout containing six buttons. (Part 1 of 2.)

```

 9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12     private final JButton[] buttons; // array of buttons
13     private static final String[] names =
14         { "one", "two", "three", "four", "five", "six" };
15     private boolean toggle = true; // toggle between two layouts
16     private final Container container; // frame container
17     private final GridLayout gridLayout1; // first GridLayout
18     private final GridLayout gridLayout2; // second GridLayout
19
20     // no-argument constructor
21     public GridLayoutFrame()
22     {
23         super("GridLayout Demo");
24         gridLayout1 = new GridLayout(2, 3, 5, 5); // 2 by 3; gaps of 5
25         gridLayout2 = new GridLayout(3, 2); // 3 by 2; no gaps
26         container = getContentPane();
27         setLayout(gridLayout1);
28         buttons = new JButton[names.length];
29
30         for (int count = 0; count < names.length; count++)
31         {
32             buttons[count] = new JButton(names[count]);
33             buttons[count].addActionListener(this); // register listener
34             add(buttons[count]); // add button to JFrame
35         }
36     }
37
38     // handle button events by toggling between layouts
39     @Override
40     public void actionPerformed(ActionEvent event)
41     {
42         if (toggle) // set layout based on toggle
43             container.setLayout(gridLayout2);
44         else
45             container.setLayout(gridLayout1);
46
47         toggle = !toggle;
48         container.validate(); // re-lay out container
49     }
50 }

```

Fig. 26.43 | GridLayout containing six buttons. (Part 2 of 2.)

```

1 // Fig. 26.44: GridLayoutDemo.java
2 // Testing GridLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class GridLayoutDemo
6 {

```

Fig. 26.44 | Testing GridLayoutFrame. (Part 1 of 2.)

```

7   public static void main(String[] args)
8   {
9       GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10      gridLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11      gridLayoutFrame.setSize(300, 200);
12      gridLayoutFrame.setVisible(true);
13  }
14 }
```

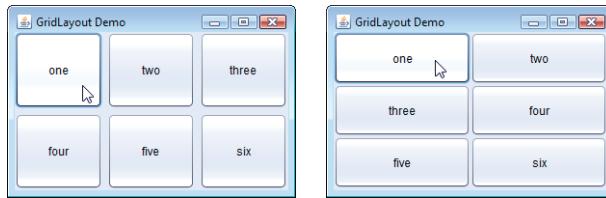


Fig. 26.44 | Testing GridLayoutFrame. (Part 2 of 2.)

Lines 24–25 (Fig. 26.43) create two `GridLayout` objects. The `GridLayout` constructor used at line 24 specifies a `GridLayout` with 2 rows, 3 columns, 5 pixels of horizontal-gap space between Components in the grid and 5 pixels of vertical-gap space between Components in the grid. The `GridLayout` constructor used at line 25 specifies a `GridLayout` with 3 rows and 2 columns that uses the default gap space (1 pixel).

The `JButton` objects in this example initially are arranged using `gridLayout1` (set for the content pane at line 27 with method `setLayout`). The first component is added to the first column of the first row. The next component is added to the second column of the first row, and so on. When a `JButton` is pressed, method `actionPerformed` (lines 39–49) is called. Every call to `actionPerformed` toggles the layout between `gridLayout2` and `gridLayout1`, using boolean variable `toggle` to determine the next layout to set.

Line 48 shows another way to reformat a container for which the layout has changed. Container method `validate` recomputes the container's layout based on the current layout manager for the Container and the current set of displayed GUI components.

26.19 Using Panels to Manage More Complex Layouts

Complex GUIs (like Fig. 26.1) often require that each component be placed in an exact location. They often consist of multiple panels, with each panel's components arranged in a specific layout. Class `JPanel` extends `JComponent` and `JComponent` extends class `Container`, so every `JPanel` is a `Container`. Thus, every `JPanel` may have components, including other panels, attached to it with `Container` method `add`. The application of Figs. 26.45–26.46 demonstrates how a `JPanel` can be used to create a more complex layout in which several `JButtons` are placed in the `SOUTH` region of a `BorderLayout`.

```
1 // Fig. 26.45: PanelFrame.java
2 // Using a JPanel to help lay out components.
3 import java.awt.GridLayout;
4 import java.awt.BorderLayout;
5 import javax.swing.JFrame;
6 import javax.swing.JPanel;
7 import javax.swing.JButton;
8
9 public class PanelFrame extends JFrame
10 {
11     private final JPanel buttonJPanel; // panel to hold buttons
12     private final JButton[] buttons;
13
14     // no-argument constructor
15     public PanelFrame()
16     {
17         super("Panel Demo");
18         buttons = new JButton[5];
19         buttonJPanel = new JPanel();
20         buttonJPanel.setLayout(new GridLayout(1, buttons.length));
21
22         // create and add buttons
23         for (int count = 0; count < buttons.length; count++)
24         {
25             buttons[count] = new JButton("Button " + (count + 1));
26             buttonJPanel.add(buttons[count]); // add button to panel
27         }
28
29         add(buttonJPanel, BorderLayout.SOUTH); // add panel to JFrame
30     }
31 }
```

Fig. 26.45 | JPanel with five JButtons in a GridLayout attached to the SOUTH region of a BorderLayout.

```
1 // Fig. 26.46: PanelDemo.java
2 // Testing PanelFrame.
3 import javax.swing.JFrame;
4
5 public class PanelDemo extends JFrame
6 {
7     public static void main(String[] args)
8     {
9         PanelFrame panelFrame = new PanelFrame();
10        panelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        panelFrame.setSize(450, 200);
12        panelFrame.setVisible(true);
13    }
14 }
```

Fig. 26.46 | Testing PanelFrame. (Part 1 of 2.)

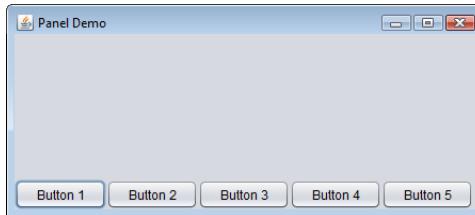


Fig. 26.46 | Testing JPanelFrame. (Part 2 of 2.)

After `JPanel buttonJPanel` is declared (line 11 of Fig. 26.45) and created (line 19), line 20 sets `buttonJPanel`'s layout to a `GridLayout` of one row and five columns (there are five `JButtons` in array `buttons`). Lines 23–27 add the `JButtons` in the array to the `JPanel`. Line 26 adds the buttons directly to the `JPanel`—class `JPanel` does not have a content pane, unlike a `JFrame`. Line 29 uses the `JFrame`'s default `BorderLayout` to add `buttonJPanel` to the `SOUTH` region. The `SOUTH` region is as tall as the buttons on `buttonJPanel`. A `JPanel` is sized to the components it contains. As more components are added, the `JPanel` grows (according to the restrictions of its layout manager) to accommodate the components. Resize the window to see how the layout manager affects the size of the `JButtons`.

26.20 JTextArea

A `JTextArea` provides an area for *manipulating multiple lines of text*. Like class `JTextField`, `JTextArea` is a subclass of `JTextComponent`, which declares common methods for `JTextFields`, `JTextAreas` and several other text-based GUI components.

The application in Figs. 26.47–26.48 demonstrates `JTextAreas`. One `JTextArea` displays text that the user can select. The other is uneditable by the user and is used to display the text the user selected in the first `JTextArea`. Unlike `JTextFields`, `JTextAreas` do not have action events—when you press *Enter* while typing in a `JTextArea`, the cursor simply moves to the next line. As with multiple-selection `JLists` (Section 26.13), an external event from another GUI component indicates when to process the text in a `JTextArea`. For example, when typing an e-mail message, you normally click a `Send` button to send the text of the message to the recipient. Similarly, when editing a document in a word processor, you normally save the file by selecting a `Save` or `Save As...` menu item. In this program, the button `Copy >>>` generates the external event that copies the selected text in the left `JTextArea` and displays it in the right `JTextArea`.

```

1 // Fig. 26.47: TextAreaFrame.java
2 // Copying selected text from one JText area to another.
3 import java.awt.event.ActionListener;
4 import java.awt.event.ActionEvent;
5 import javax.swing.Box;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;

```

Fig. 26.47 | Copying selected text from one `JTextArea` to another. (Part 1 of 2.)

```

8 import javax.swing.JButton;
9 import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13     private final JTextArea textArea1; // displays demo string
14     private final JTextArea textArea2; // highlighted text is copied here
15     private final JButton copyJButton; // initiates copying of text
16
17     // no-argument constructor
18     public TextAreaFrame()
19     {
20         super("TextArea Demo");
21         Box box = Box.createHorizontalBox(); // create box
22         String demo = "This is a demo string to\n" +
23             "illustrate copying text\nfrom one textarea to \n" +
24             "another textarea using an\nexternal event\n";
25
26         textArea1 = new JTextArea(demo, 10, 15);
27         box.add(new JScrollPane(textArea1)); // add scrollpane
28
29         copyJButton = new JButton("Copy >>"); // create copy button
30         box.add(copyJButton); // add copy button to box
31         copyJButton.addActionListener(
32             new ActionListener() // anonymous inner class
33             {
34                 // set text in textArea2 to selected text from textArea1
35                 @Override
36                 public void actionPerformed(ActionEvent event)
37                 {
38                     textArea2.setText(textArea1.getSelectedText());
39                 }
40             }
41         );
42
43         textArea2 = new JTextArea(10, 15);
44         textArea2.setEditable(false);
45         box.add(new JScrollPane(textArea2)); // add scrollpane
46
47         add(box); // add box to frame
48     }
49 }
```

Fig. 26.47 | Copying selected text from one JTextArea to another. (Part 2 of 2.)

```

1 // Fig. 26.48: TextAreaDemo.java
2 // Testing TextAreaFrame.
3 import javax.swing.JFrame;
4
5 public class TextAreaDemo
6 {
```

Fig. 26.48 | Testing TextAreaFrame. (Part I of 2.)

```

7     public static void main(String[] args)
8     {
9         TextAreaFrame textAreaFrame = new TextAreaFrame();
10        textAreaFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        textAreaFrame.setSize(425, 200);
12        textAreaFrame.setVisible(true);
13    }
14 }
```

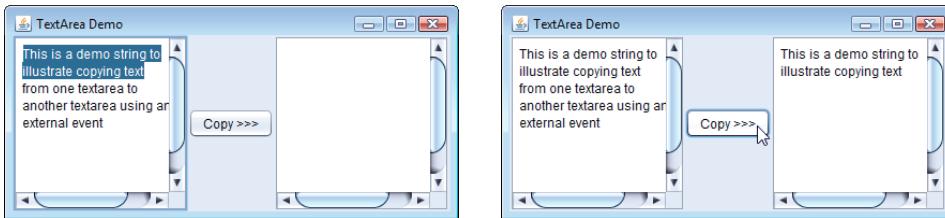


Fig. 26.48 | Testing TextAreaFrame. (Part 2 of 2.)

In the constructor (lines 18–48), line 21 creates a **Box** container (package `javax.swing`) to organize the GUI components. **Box** is a subclass of **Container** that uses a **BoxLayout** layout manager (discussed in detail in Section 35.9) to arrange the GUI components either horizontally or vertically. **Box**'s static method `createHorizontalBox` creates a **Box** that arranges components from left to right in the order that they're attached.

Lines 26 and 43 create **JTextAreas** `textArea1` and `textArea2`. Line 26 uses **JTextArea**'s three-argument constructor, which takes a **String** representing the initial text and two **ints** specifying that the **JTextArea** has 10 rows and 15 columns. Line 43 uses **JTextArea**'s two-argument constructor, specifying that the **JTextArea** has 10 rows and 15 columns. Line 26 specifies that `demo` should be displayed as the default **JTextArea** content. A **JTextArea** does not provide scrollbars if it cannot display its complete contents. So, line 27 creates a **JScrollPane** object, initializes it with `textArea1` and attaches it to container `box`. By default, horizontal and vertical scrollbars appear as necessary in a **JScrollPane**.

Lines 29–41 create **JButton** object `copyJButton` with the label "Copy >>", add `copyJButton` to container `box` and register the event handler for `copyJButton`'s **ActionEvent**. This button provides the external event that determines when the program should copy the selected text in `textArea1` to `textArea2`. When the user clicks `copyJButton`, line 38 in `actionPerformed` indicates that method `getSelectedText` (inherited into **JTextArea** from **JTextComponent**) should return the selected text from `textArea1`. The user selects text by dragging the mouse over the desired text to highlight it. Method `setText` changes the text in `textArea2` to the string returned by `getSelectedText`.

Lines 43–45 create `textArea2`, set its `editable` property to `false` and add it to container `box`. Line 47 adds `box` to the **JFrame**. Recall from Section 26.18.2 that the default layout of a **JFrame** is a **BorderLayout** and that the `add` method by default attaches its argument to the **CENTER** of the **BorderLayout**.

When text reaches the right edge of a `JTextArea` the text can wrap to the next line. This is referred to as **line wrapping**. By default, `JTextArea` does *not* wrap lines.



Look-and-Feel Observation 26.19

To provide line wrapping functionality for a `JTextArea`, invoke `JTextArea` method `setLineWrap` with a `true` argument.

JScrollPane Scrollbar Policies

This example uses a `JScrollPane` to provide scrolling for a `JTextArea`. By default, `JScrollPane` displays scrollbars *only* if they're required. You can set the horizontal and vertical **scrollbar policies** of a `JScrollPane` when it's constructed. If a program has a reference to a `JScrollPane`, the program can use `JScrollPane` methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy` to change the scrollbar policies at any time. Class `JScrollPane` declares the constants

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS  
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

to indicate that *a scrollbar should always appear*, constants

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED  
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

to indicate that *a scrollbar should appear only if necessary* (the defaults) and constants

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER  
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

to indicate that *a scrollbar should never appear*. If the horizontal scrollbar policy is set to `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, a `JTextArea` attached to the `JScrollPane` will automatically wrap lines.

26.21 Wrap-Up

In this chapter, you learned many GUI components and how to handle their events. You also learned about nested classes, inner classes and anonymous inner classes. You saw the special relationship between an inner-class object and an object of its top-level class. You learned how to use `JOptionPane` dialogs to obtain text input from the user and how to display messages to the user. You also learned how to create applications that execute in their own windows. We discussed class `JFrame` and components that enable a user to interact with an application. We also showed you how to display text and images to the user. You learned how to customize `JPanels` to create custom drawing areas, which you'll use extensively in the next chapter. You saw how to organize components on a window using layout managers and how to creating more complex GUIs by using `JPanels` to organize components. Finally, you learned about the `JTextArea` component in which a user can enter text and an application can display text. In Chapter 35, you'll learn about more advanced GUI components, such as sliders, menus and more complex layout managers. In the next chapter, you'll learn how to add graphics to your GUI application. Graphics allow you to draw shapes and text with colors and styles.

Summary

Section 26.1 Introduction

- A graphical user interface (GUI; p. 2) presents a user-friendly mechanism for interacting with an application. A GUI gives an application a distinctive look-and-feel (p. 2).
- Providing different applications with consistent, intuitive user-interface components gives users a sense of familiarity with a new application, so that they can learn it more quickly.
- GUIs are built from GUI components (p. 2)—sometimes called controls or widgets.

Section 26.2 Java's Nimbus Look-and-Feel

- As of Java SE 6 update 10, Java comes bundled with a new, elegant, cross-platform look-and-feel known as Nimbus (p. 4).
- To set Nimbus as the default for all Java applications, create a `swing.properties` text file in the `\lib` folder of your JDK and JRE installation folders. Place the following line of code in the file:
`swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel`
- To select Nimbus on an application-by-application basis, place the following command-line argument after the `java` command and before the application's name when you run the application:
`-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel`

Section 26.3 Simple GUI-Based Input/Output with JOptionPane

- Most applications use windows or dialog boxes (p. 4) to interact with the user.
- Class `JOptionPane` (p. 4) of package `javax.swing` provides prebuilt dialog boxes for both input and output. `JOptionPane` static method `showInputDialog` (p. 5) displays an input dialog (p. 4).
- A prompt typically uses sentence-style capitalization—capitalizing only the first letter of the first word in the text unless the word is a proper noun.
- An input dialog can input only input `Strings`. This is typical of most GUI components.
- `JOptionPane` static method `showMessageDialog` (p. 6) displays a message dialog (p. 4).

Section 26.4 Overview of Swing Components

- Most Swing GUI components (p. 2) are located in package `javax.swing`.
- Together, the appearance and the way in which the user interacts with the application are known as that application's look-and-feel. Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel.
- Lightweight Swing components are not tied to actual GUI components supported by the underlying platform on which an application executes.
- Several Swing components are heavyweight components (p. 8) that require direct interaction with the local windowing system (p. 8), which may restrict their appearance and functionality.
- Class `Component` (p. 8) of package `java.awt` declares many of the attributes and behaviors common to the GUI components in packages `java.awt` (p. 7) and `javax.swing`.
- Class `Container` (p. 8) of package `java.awt` is a subclass of `Component`. Components are attached to `Containers` so the `Components` can be organized and displayed on the screen.
- Class `JComponent` (p. 8) of package `javax.swing` is a subclass of `Container`. `JComponent` is the superclass of all lightweight Swing components and declares their common attributes and behaviors.

- Some common `JComponent` features include a pluggable look-and-feel (p. 8), shortcut keys called mnemonics (p. 8), tool tips (p. 9), support for assistive technologies and support for user-interface localization (p. 9).

Section 26.5 Displaying Text and Images in a Window

- Class `JFrame` provides the basic attributes and behaviors of a window.
- A `JLabel` (p. 9) displays read-only text, an image, or both text and an image. Text in a `JLabel` normally uses sentence-style capitalization.
- Each GUI component must be attached to a container, such as a window created with a `JFrame`.
- Many IDEs provide GUI design tools in which you can specify the exact size and location of a component by using the mouse; then the IDE will generate the GUI code for you.
- `JComponent` method `setToolTipText` (p. 11) specifies the tool tip that's displayed when the user positions the mouse cursor over a lightweight component (p. 8).
- `Container` method `add` attaches a GUI component to a `Container`.
- Class `ImageIcon` (p. 12) supports several image formats, including GIF, PNG and JPEG.
- Method `getClass` of class `Object` (p. 12) retrieves a reference to the `Class` object that represents the class declaration for the object on which the method is called.
- `Class` method `getResource` (p. 12) returns the location of its argument as a URL. The method `getResource` uses the `Class` object's class loader to determine the location of the resource.
- The horizontal and vertical alignments of a `JLabel` can be set with methods `setHorizontalAlignment` (p. 13) and `setVerticalAlignment` (p. 13), respectively.
- `JLabel` methods `setText` (p. 13) and `getText` (p. 13) set and get the text displayed on a label.
- `JLabel` methods `setIcon` (p. 13) and `getIcon` (p. 13) set and get the `Icon` (p. 12) on a label.
- `JLabel` methods `setHorizontalTextPosition` (p. 13) and `setVerticalTextPosition` (p. 13) specify the text position in the label.
- `JFrame` method `setDefaultCloseOperation` (p. 13) with constant `JFrame.EXIT_ON_CLOSE` as the argument indicates that the program should terminate when the window is closed by the user.
- `Component` method `setSize` (p. 13) specifies the width and height of a component.
- `Component` method `setVisible` (p. 13) with the argument `true` displays a `JFrame` on the screen.

Section 26.6 Text Fields and an Introduction to Event Handling with Nested Classes

- GUIs are event driven—when the user interacts with a GUI component, events (p. 13) drive the program to perform tasks.
- An event handler (p. 14) performs a task in response to an event.
- Class `JTextField` (p. 14) extends `JTextComponent` (p. 14) of package `javax.swing.text`, which provides common text-based component features. Class `JPasswordField` (p. 14) extends `JTextField` and adds several methods that are specific to processing passwords.
- A `JPasswordField` shows that characters are being typed as the user enters them, but hides the actual characters with echo characters (p. 14).
- A component receives the focus (p. 14) when the user clicks the component.
- `JTextComponent` method `setEditable` (p. 16) can be used to make a text field uneditable.
- To respond to an event for a particular GUI component, you must create a class that represents the event handler and implements an appropriate event-listener interface (p. 16), then register an object of the event-handling class as the event handler (p. 16).

- Non-static nested classes (p. 17) are called inner classes and are frequently used for event handling.
- An object of a non-static inner class (p. 17) must be created by an object of the top-level class (p. 17) that contains the inner class.
- An inner-class object can directly access the instance variables and methods of its top-level class.
- A nested class that's static does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class.
- Pressing *Enter* in a JTextField or JPasswordField generates an ActionEvent (p. 17) that can be handled by an ActionListener (p. 17) of package java.awt.event.
- JTextField method addActionListener (p. 17) registers an event handler for a text field's ActionEvent.
- The GUI component with which the user interacts is the event source (p. 18).
- An ActionEvent object contains information about the event that just occurred, such as the event source and the text in the text field.
- ActionEvent method getSource returns a reference to the event source. ActionEvent method getActionCommand (p. 18) returns the text the user typed in a text field or the label on a JButton.
- JPasswordField method getPassword (p. 18) returns the password the user typed.

Section 26.7 Common GUI Event Types and Listener Interfaces

- Each event-object type typically has a corresponding event-listener interface that specifies one or more event-handling methods, which must be declared in the class that implements the interface.

Section 26.8 How Event Handling Works

- When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener's appropriate event-handling method.
- Every GUI component supports several event types. When an event occurs, the event is dispatched (p. 23) only to the event listeners of the appropriate type.

Section 26.9 JButton

- A button is a component the user clicks to trigger an action. All the button types are subclasses of AbstractButton (p. 23; package javax.swing). Button labels (p. 24) typically use book-title capitalization.
- Command buttons (p. 23) are created with class JButton.
- A JButton can display an Icon. A JButton can also have a rollover Icon (p. 25)—an Icon that's displayed when the user positions the mouse over the button.
- Method setRolloverIcon (p. 25) of class AbstractButton specifies the image displayed on a button when the user positions the mouse over it.

Section 26.10 Buttons That Maintain State

- There are three Swing state button types—JToggleButton (p. 27), JCheckBox (p. 27) and JRadioButton (p. 27).
- Classes JCheckBox and JRadioButton are subclasses of JToggleButton.
- Component method setFont (p. 29) sets the component's font to a new Font object (p. 29) of package java.awt.

- Clicking a `JCheckBox` causes an `ItemEvent` (p. 29) that can be handled by an `ItemListener` (p. 29) which defines method `itemStateChanged` (p. 29). Method `addItemListener` registers the listener for the `ItemEvent` of a `JCheckBox` or `JRadioButton` object.
- `JCheckBox` method `isSelected` determines whether a `JCheckBox` is selected.
- `JRadioButtons` have two states—selected and not selected. Radio buttons (p. 23) normally appear as a group (p. 30) in which only one button can be selected at a time.
- `JRadioButtons` are used to represent mutually exclusive options (p. 30).
- The logical relationship between `JRadioButtons` is maintained by a `ButtonGroup` object (p. 30).
- `ButtonGroup` method `add` (p. 32) associates each `JRadioButton` with a `ButtonGroup`. If more than one selected `JRadioButton` object is added to a group, the selected one that was added first will be selected when the GUI is displayed.
- `JRadioButtons` generate `ItemEvents` when they're clicked.

Section 26.11 JComboBox; Using an Anonymous Inner Class for Event Handling

- A `JComboBox` (p. 33) provides a list of items from which the user can make a single selection. `JComboBoxes` generate `ItemEvents`.
- Each item in a `JComboBox` has an index (p. 35). The first item added to a `JComboBox` appears as the currently selected item when the `JComboBox` is displayed.
- `JComboBox` method `setMaximumRowCount` (p. 35) sets the maximum number of elements that are displayed when the user clicks the `JComboBox`.
- An anonymous inner class (p. 35) is a class without a name and typically appears inside a method declaration. One object of the anonymous inner class must be created when the class is declared.
- `JComboBox` method `getSelectedIndex` (p. 36) returns the index of the selected item.

Section 26.12 JList

- A `JList` displays a series of items from which the user may select one or more items. Class `JList` supports single-selection lists (p. 36) and multiple-selection lists.
- When the user clicks an item in a `JList`, a `ListSelectionEvent` (p. 37) occurs. `JList` method `addListSelectionListener` (p. 38) registers a `ListSelectionListener` (p. 38) for a `JList`'s selection events. A `ListSelectionListener` of package `javax.swing.event` must implement method `valueChanged`.
- `JList` method `setVisibleRowCount` (p. 37) specifies the number of visible items in the list.
- `JList` method `setSelectionMode` (p. 37) specifies a list's selection mode.
- A `JList` can be attached to a `JScrollPane` (p. 38) to provide a scrollbar for the `JList`.
- `JFrame` method `getContentPane` (p. 39) returns a reference to the `JFrame`'s content pane where GUI components are displayed.
- `JList` method `getSelectedIndex` (p. 39) returns the selected item's index.

Section 26.13 Multiple-Selection Lists

- A multiple-selection list enables the user to select many items from a `JList`.
- `JList` method `setFixedCellWidth` (p. 41) sets a `JList`'s width. Method `setFixedCellHeight` (p. 41) sets the height of each item in a `JList`.
- Normally, an external event (p. 41) generated by another GUI component (such as a `JButton`) specifies when the multiple selections in a `JList` should be processed.

- `JList` method `setListData` (p. 41) sets the items displayed in a `JList`. `JList` method `getSelectedValues` (p. 41) returns an array of `Objects` representing the selected items in a `JList`.

Section 26.14 Mouse Event Handling

- The `MouseListener` (p. 23) and `MouseMotionListener` event-listener interfaces are used to handle mouse events. Mouse events can be trapped for any GUI component that extends `Component`.
- Interface `MouseInputListener` of package `javax.swing.event` extends interfaces `MouseListener` and `MouseMotionListener` to create a single interface containing all their methods.
- Each mouse event-handling method receives a `MouseEvent` object (p. 23) that contains information about the event, including the *x*- and *y*-coordinates where the event occurred. Coordinates are measured from the upper-left corner of the GUI component on which the event occurred.
- The methods and constants of class `InputEvent` (`MouseEvent`'s superclass) enable an application to determine which mouse button the user clicked.
- Interface `MouseWheelListener` enables applications to respond to mouse-wheel events.

Section 26.15 Adapter Classes

- An adapter class (p. 46) implements an interface and provides default implementations of its methods. When you extend an adapter class, you can override just the method(s) you need.
- `MouseEvent` method `getClickCount` (p. 50) returns the number of consecutive mouse-button clicks. Methods `isMetaDown` (p. 50) and `isAltDown` (p. 50) determine which button was clicked.

Section 26.16 JPanel Subclass for Drawing with the Mouse

- `JComponents` method `paintComponent` (p. 50) is called when a lightweight Swing component is displayed. Override this method to specify how to draw shapes using Java's graphics capabilities.
- When overriding `paintComponent`, call the superclass version as the first statement in the body.
- Subclasses of `JComponent` support transparency. When a component is opaque (p. 50), `paintComponent` clears its background before the component is displayed.
- The transparency of a Swing lightweight component can be set with method `setOpaque` (p. 50; a `false` argument indicates that the component is transparent).
- Class `Point` (p. 52) package `java.awt` represents an *x-y* coordinate.
- Class `Graphics` (p. 52) is used to draw.
- `MouseEvent` method `getPoint` (p. 52) obtains the `Point` where a mouse event occurred.
- Method `repaint` (p. 52), inherited indirectly from class `Component`, indicates that a component should be refreshed on the screen as soon as possible.
- Method `paintComponent` receives a `Graphics` parameter and is called automatically whenever a lightweight component needs to be displayed on the screen.
- `Graphics` method `fillOval` (p. 52) draws a solid oval. The first two arguments are the upper-left *x-y* coordinate of the bounding box, and the last two are the bounding box's width and height.

Section 26.17 Key Event Handling

- Interface `KeyListener` is used to handle key events that are generated when keys on the keyboard are pressed and released. Method `addKeyListener` of class `Component` (p. 53) registers a `KeyListener`.
- `KeyEvent` method `getKeyCode` (p. 56) gets the virtual key code (p. 56) of the pressed key. Class `KeyEvent` contains virtual key-code constants that represent every key on the keyboard.
- `KeyEvent` method `getKeyText` (p. 56) returns a string containing the name of the pressed key.
- `KeyEvent` method `getKeyChar` (p. 56) gets the Unicode value of the character typed.

- `KeyEvent` method `isActionKey` (p. 56) determines whether the key in an event was an action key (p. 53).
- `InputEvent` method `getModifiers` (p. 56) determines whether any modifier keys (such as *Shift*, *Alt* and *Ctrl*) were pressed when the key event occurred.
- `KeyEvent` method `getKeyModifiersText` (p. 56) returns a string containing the pressed modifier keys.

Section 26.18 Introduction to Layout Managers

- Layout managers (p. 11) arrange GUI components in a container for presentation purposes.
- All layout managers implement the interface `LayoutManager` of package `java.awt`.
- `Container` method `setLayout` specifies the layout of a container.
- `FlowLayout` places components left to right in the order in which they're added to the container. When the container's edge is reached, components continue to display on the next line. `FlowLayout` allows GUI components to be left aligned, centered (the default) and right aligned.
- `FlowLayout` method `setAlignment` changes the alignment for a `FlowLayout`.
- `BorderLayout` (the default for a `JFrame`) arranges components into five regions: NORTH, SOUTH, EAST, WEST and CENTER. NORTH corresponds to the top of the container.
- A `BorderLayout` limits a `Container` to containing at most five components—one in each region.
- `GridLayout` (p. 64) divides a container into a grid of rows and columns.
- `Container` method `validate` (p. 66) recomputes a container's layout based on the current layout manager for the `Container` and the current set of displayed GUI components.

Section 26.19 Using Panels to Manage More Complex Layouts

- Complex GUIs often consist of multiple panels with different layouts. Every `JPanel` may have components, including other panels, attached to it with `Container` method `add`.

Section 26.20 JTextArea

- A `JTextArea` (p. 68)—a subclass of `JTextComponent`—may contain multiple lines of text.
- Class `Box` (p. 70) is a subclass of `Container` that uses a `BoxLayout` layout manager (p. 70) to arrange the GUI components either horizontally or vertically.
- `Box` static method `createHorizontalBox` (p. 70) creates a `Box` that arranges components from left to right in the order that they're attached.
- Method `getSelectedText` (p. 70) returns the selected text from a `JTextArea`.
- You can set the horizontal and vertical scrollbar policies (p. 71) of a `JScrollPane` when it's constructed. `JScrollPane` methods `setHorizontalScrollBarPolicy` (p. 71) and `setVerticalScrollBarPolicy` (p. 71) can be used to change the scrollbar policies at any time.

Self-Review Exercises

26.1 Fill in the blanks in each of the following statements:

- Method _____ is called when the mouse is moved with no buttons pressed and an event listener is registered to handle the event.
- Text that cannot be modified by the user is called _____ text.
- A(n) _____ arranges GUI components in a `Container`.
- The `add` method for attaching GUI components is a method of class _____.
- GUI is an acronym for _____.
- Method _____ is used to specify the layout manager for a container.

- g) A `mouseDragged` method call is preceded by a(n) _____ method call and followed by a(n) _____ method call.
 - h) Class _____ contains methods that display message dialogs and input dialogs.
 - i) An input dialog capable of receiving input from the user is displayed with method _____ of class _____.
 - j) A dialog capable of displaying a message to the user is displayed with method _____ of class _____.
 - k) Both `JTextFields` and `JTextAreas` directly extend class _____.
- 26.2** Determine whether each statement is *true* or *false*. If *false*, explain why.
- a) `BorderLayout` is the default layout manager for a `JFrame`'s content pane.
 - b) When the mouse cursor is moved into the bounds of a GUI component, method `mouseOver` is called.
 - c) A `JPanel` cannot be added to another `JPanel`.
 - d) In a `BorderLayout`, two buttons added to the `NORTH` region will be placed side by side.
 - e) A maximum of five components can be added to a `BorderLayout`.
 - f) Inner classes are not allowed to access the members of the enclosing class.
 - g) A `JTextArea`'s text is always read-only.
 - h) Class `JTextArea` is a direct subclass of class `Component`.

- 26.3** Find the error(s) in each of the following statements, and explain how to correct it (them):

```
a) buttonName = JButton("Caption");  
b) JLabel aLabel, JLabel;  
c) txtField = new JTextField(50, "Default Text");  
d) setLayout(new BorderLayout());  
    button1 = new JButton("North Star");  
    button2 = new JButton("South Pole");  
    add(button1);  
    add(button2);
```

Answers to Self-Review Exercises

- 26.1** a) `mouseMoved`. b) uneditable (read-only). c) layout manager. d) `Container`. e) graphical user interface. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.

- 26.2** Answers for a) through h):

- a) True.
- b) False. Method `mouseEntered` is called.
- c) False. A `JPanel` can be added to another `JPanel`, because `JPanel` is an indirect subclass of `Component`. So, a `JPanel` is a `Component`. Any `Component` can be added to a `Container`.
- d) False. Only the last button added will be displayed. Remember that only one component should be added to each region in a `BorderLayout`.
- e) True. [Note: Panels containing multiple components can be added to each region.]
- f) False. Inner classes have access to all members of the enclosing class declaration.
- g) False. `JTextAreas` are editable by default.
- h) False. `JTextArea` derives from class `JTextComponent`.

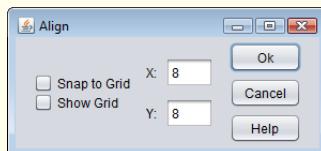
- 26.3** Answers for a) through d):

- a) `new` is needed to create an object.
- b) `JLabel` is a class name and cannot be used as a variable name.
- c) The arguments passed to the constructor are reversed. The `String` must be passed first.

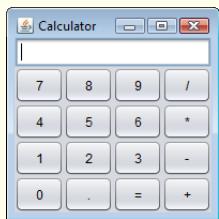
- d) `BorderLayout` has been set, and components are being added without specifying the region, so both are added to the center region. Proper add statements might be
`add(button1, BorderLayout.NORTH);`
`add(button2, BorderLayout.SOUTH);`

Exercises

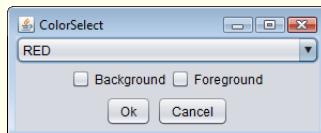
- 26.4** Fill in the blanks in each of the following statements:
- The `JTextField` class directly extends class _____.
 - Container method _____ attaches a GUI component to a container.
 - Method _____ is called when a mouse button is released (without moving the mouse).
 - The _____ class is used to create a group of `JRadioButtons`.
- 26.5** Determine whether each statement is *true* or *false*. If *false*, explain why.
- Only one layout manager can be used per `Container`.
 - GUI components can be added to a `Container` in any order in a `BorderLayout`.
 - `JRadioButtons` provide a series of mutually exclusive options (i.e., only one can be *true* at a time).
 - `Graphics` method `setFont` is used to set the font for text fields.
 - A `JList` displays a scrollbar if there are more items in the list than can be displayed.
 - A `Mouse` object has a method called `mouseDragged`.
- 26.6** Determine whether each statement is *true* or *false*. If *false*, explain why.
- A `JPanel` is a `JComponent`.
 - A `JPanel` is a `Component`.
 - A `JLabel` is a `Container`.
 - A `JList` is a `JPanel`.
 - An `AbstractButton` is a `JButton`.
 - A `JTextField` is an `Object`.
 - `ButtonGroup` is a subclass of `JComponent`.
- 26.7** Find any errors in each of the following lines of code, and explain how to correct them.
- `import javax.swing.JFrame`
 - `panelObject.GridLayout(8, 8);`
 - `container.setLayout(new FlowLayout(FlowLayout.DEFAULT));`
 - `container.add(eastButton, EAST);`
- 26.8** Create the following GUI. You do not have to provide any functionality.



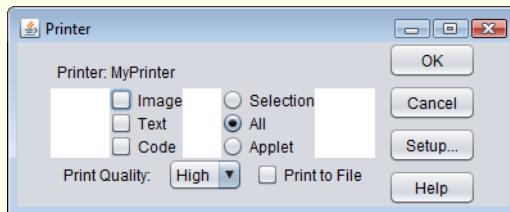
- 26.9** Create the following GUI. You do not have to provide any functionality.



- 26.10** Create the following GUI. You do not have to provide any functionality.



- 26.11** Create the following GUI. You do not have to provide any functionality.



- 26.12** (*Temperature Conversion*) Write a temperature-conversion application that converts from Fahrenheit to Celsius. The Fahrenheit temperature should be entered from the keyboard (via a JTextField). A JLabel should be used to display the converted temperature. Use the following formula for the conversion:

$$\text{Celsius} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

- 26.13** (*Temperature-Conversion Modification*) Enhance the temperature-conversion application of Exercise 26.12 by adding the Kelvin temperature scale. The application should also allow the user to make conversions between any two scales. Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise 26.12):

$$\text{Kelvin} = \text{Celsius} + 273.15$$

- 26.14** (*Guess-the-Number Game*) Write an application that plays “guess the number” as follows: Your application chooses the number to be guessed by selecting an integer at random in the range 1–1000. The application then displays the following in a label:

I have a number between 1 and 1000. Can you guess my number?
Please enter your first guess.

A JTextField should be used to input the guess. As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting “warmer,” and blue, “colder.” A JLabel should display either “Too High” or “Too Low” to help the user zero in. When the user gets the correct answer, “Correct!” should be displayed, and the JTextField used for input should be changed to be uneditable. A JButton should be provided to allow the user to play the game again. When the JButton is clicked, a new random number should be generated and the input JTextField changed to be editable.

- 26.15** (*Displaying Events*) It’s often useful to display the events that occur during the execution of an application. This can help you understand when the events occur and how they’re generated. Write an application that enables the user to generate and process every event discussed in this chapter. The application should provide methods from the ActionListener, ItemListener, ListSelectionListener, MouseListener, MouseMotionListener and KeyListener interfaces to display messages when the events occur. Use method `toString` to convert the event objects received in each

event handler into `Strings` that can be displayed. Method `toString` creates a `String` containing all the information in the event object.

26.16 (GUI-Based Craps Game) Modify the application of Section 6.10 to provide a GUI that enables the user to click a `JButton` to roll the dice. The application should also display four `JLabels` and four `JTextFields`, with one `JLabel` for each `JTextField`. The `JTextFields` should be used to display the values of each die and the sum of the dice after each roll. The point should be displayed in the fourth `JTextField` when the user does not win or lose on the first roll and should continue to be displayed until the game is lost.

(Optional) GUI and Graphics Case Study Exercise: Expanding the Interface

26.17 (Interactive Drawing Application) In this exercise, you'll implement a GUI application that uses the `MyShape` hierarchy from GUI and Graphics Case Study Exercise 10.2 to create an interactive drawing application. You'll create two classes for the GUI and provide a test class that launches the application. The classes of the `MyShape` hierarchy require no additional changes.

The first class to create is a subclass of `JPanel` called `DrawPanel`, which represents the area on which the user draws the shapes. Class `DrawPanel` should have the following instance variables:

- An array `shapes` of type `MyShape` that will store all the shapes the user draws.
- An integer `shapeCount` that counts the number of shapes in the array.
- An integer `shapeType` that determines the type of shape to draw.
- A `MyShape currentShape` that represents the current shape the user is drawing.
- A `Color currentColor` that represents the current drawing color.
- A boolean `filledShape` that determines whether to draw a filled shape.
- A `JLabel statusLabel` that represents the status bar. The status bar will display the coordinates of the current mouse position.

Class `DrawPanel` should also declare the following methods:

- Overridden method `paintComponent` that draws the shapes in the array. Use instance variable `shapeCount` to determine how many shapes to draw. Method `paintComponent` should also call `currentShape`'s `draw` method, provided that `currentShape` is not `null`.
- Set methods for the `shapeType`, `currentColor` and `filledShape`.
- Method `clearLastShape` should clear the last shape drawn by decrementing instance variable `shapeCount`. Ensure that `shapeCount` is never less than zero.
- Method `clearDrawing` should remove all the shapes in the current drawing by setting `shapeCount` to zero.

Methods `clearLastShape` and `clearDrawing` should call `repaint` (inherited from `JPanel`) to refresh the drawing on the `DrawPanel` by indicating that the system should call method `paintComponent`.

Class `DrawPanel` should also provide event handling to enable the user to draw with the mouse. Create a single inner class that both extends `MouseAdapter` and implements `MouseMotionListener` to handle all mouse events in one class.

In the inner class, override method `mousePressed` so that it assigns `currentShape` a new shape of the type specified by `shapeType` and initializes both points to the mouse position. Next, override method `mouseReleased` to finish drawing the current shape and place it in the array. Set the second point of `currentShape` to the current mouse position and add `currentShape` to the array. Instance variable `shapeCount` determines the insertion index. Set `currentShape` to `null` and call method `repaint` to update the drawing with the new shape.

Override method `mouseMoved` to set the text of the `statusLabel` so that it displays the mouse coordinates—this will update the label with the coordinates every time the user moves (but does not drag) the mouse within the `DrawPanel`. Next, override method `mouseDragged` so that it sets the second point of the `currentShape` to the current mouse position and calls method `repaint`. This will allow the user to see the shape while dragging the mouse. Also, update the `JLabel` in mouse-

Dragged with the current position of the mouse.

Create a constructor for `DrawPanel` that has a single `JLabel` parameter. In the constructor, initialize `statusLabel` with the value passed to the parameter. Also initialize array `shapes` with 100 entries, `shapeCount` to 0, `shapeType` to the value that represents a line, `currentShape` to `null` and `currentColor` to `Color.BLACK`. The constructor should then set the background color of the `DrawPanel` to `Color.WHITE` and register the `MouseListener` and `MouseMotionListener` so the `JPanel` properly handles mouse events.

Next, create a `JFrame` subclass called `DrawFrame` that provides a GUI that enables the user to control various aspects of drawing. For the layout of the `DrawFrame`, we recommend a `BorderLayout`, with the components in the `NORTH` region, the main drawing panel in the `CENTER` region, and a status bar in the `SOUTH` region, as in Fig. 26.49. In the top panel, create the components listed below. Each component's event handler should call the appropriate method in class `DrawPanel`.

- A button to undo the last shape drawn.
- A button to clear all shapes from the drawing.
- A combo box for selecting the color from the 13 predefined colors.
- A combo box for selecting the shape to draw.
- A checkbox that specifies whether a shape should be filled or unfilled.

Declare and create the interface components in `DrawFrame`'s constructor. You'll need to create the status bar `JLabel` before you create the `DrawPanel`, so you can pass the `JLabel` as an argument to `DrawPanel`'s constructor. Finally, create a test class that initializes and displays the `DrawFrame` to execute the application.

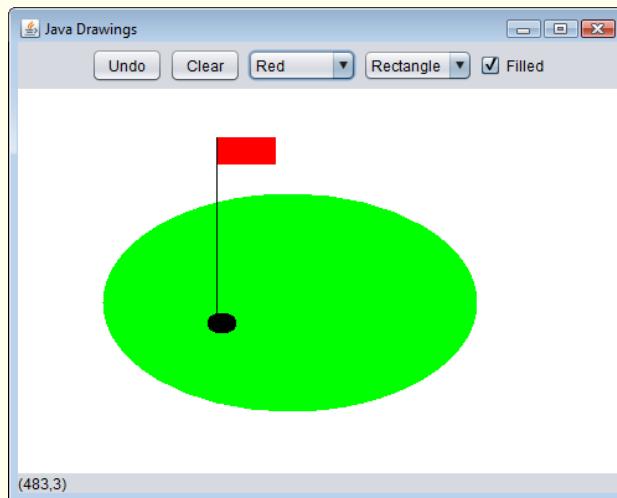


Fig. 26.49 | Interface for drawing shapes.

26.18 (GUI-Based Version of the ATM Case Study) Reimplement the Optional ATM Case Study of Chapters 33–34 as a GUI-based application. Use GUI components to approximate the ATM user interface shown in Fig. 33.1. For the cash dispenser and the deposit slot use `JButtons` labeled `Remove Cash` and `Insert Envelope`. This will enable the application to receive events indicating when the user takes the cash and inserts a deposit envelope, respectively.

Making a Difference

26.19 (Ecofont) Ecofont (<http://www.ecofont.com/en/products/green/font/download-the-ink-saving-font.html>)—developed by SPRANQ (a Netherlands-based company)—is a free, open-source computer font designed to reduce by as much as 20% the amount of ink used for printing, thus reducing also the number of ink cartridges used and the environmental impact of the manufacturing and shipping processes (using less energy, less fuel for shipping, and so on). The font, based on sans-serif Verdana, has small circular “holes” in the letters that are not visible in smaller sizes—such as the 9- or 10-point type frequently used. Download Ecofont, then install the font file Spranq_eco_sans_regular.ttf using the instructions from the Ecofont website. Next, develop a GUI-based program that allows you to type in a text string to be displayed in the Ecofont. Create **Increase Font Size** and **Decrease Font Size** buttons that allow you to scale up or down by one point at a time. Start with a default font size of 9 points. As you scale up, you’ll be able to see the holes in the letters more clearly. As you scale down, the holes will be less apparent. What is the smallest font size at which you begin to notice the holes?

26.20 (Typing Tutor: Tuning a Crucial Skill in the Computer Age) Typing quickly and correctly is an essential skill for working effectively with computers and the Internet. In this exercise, you’ll build a GUI application that can help users learn to “touch type” (i.e., type correctly without looking at the keyboard). The application should display a *virtual keyboard* (Fig. 26.50) and should allow the user to watch what he or she is typing on the screen without looking at the *actual keyboard*. Use **JButtons** to represent the keys. As the user presses each key, the application highlights the corresponding **JButton** on the GUI and adds the character to a **JTextArea** that shows what the user has typed so far. [Hint: To highlight a **JButton**, use its **setBackgroundColor** method to change its background color. When the key is released, reset its original background color. You can obtain the **JButton**’s original background color with the **getBackground** method before you change its color.]

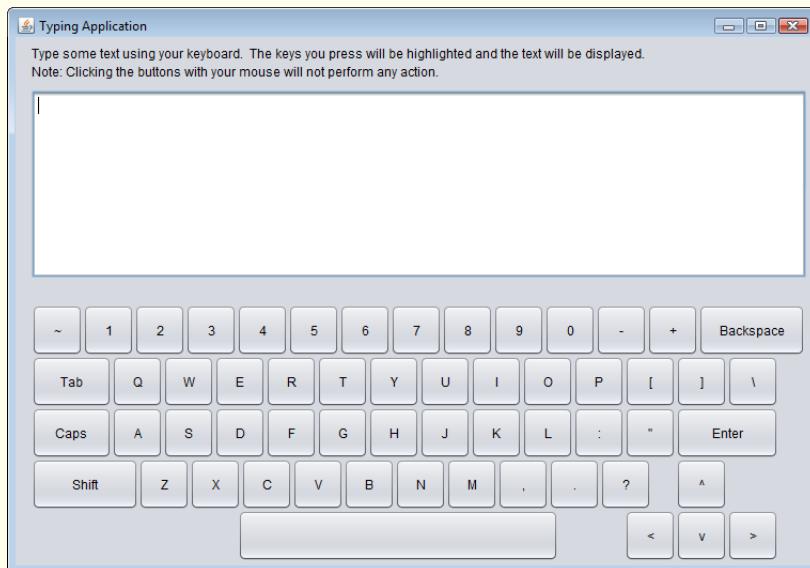


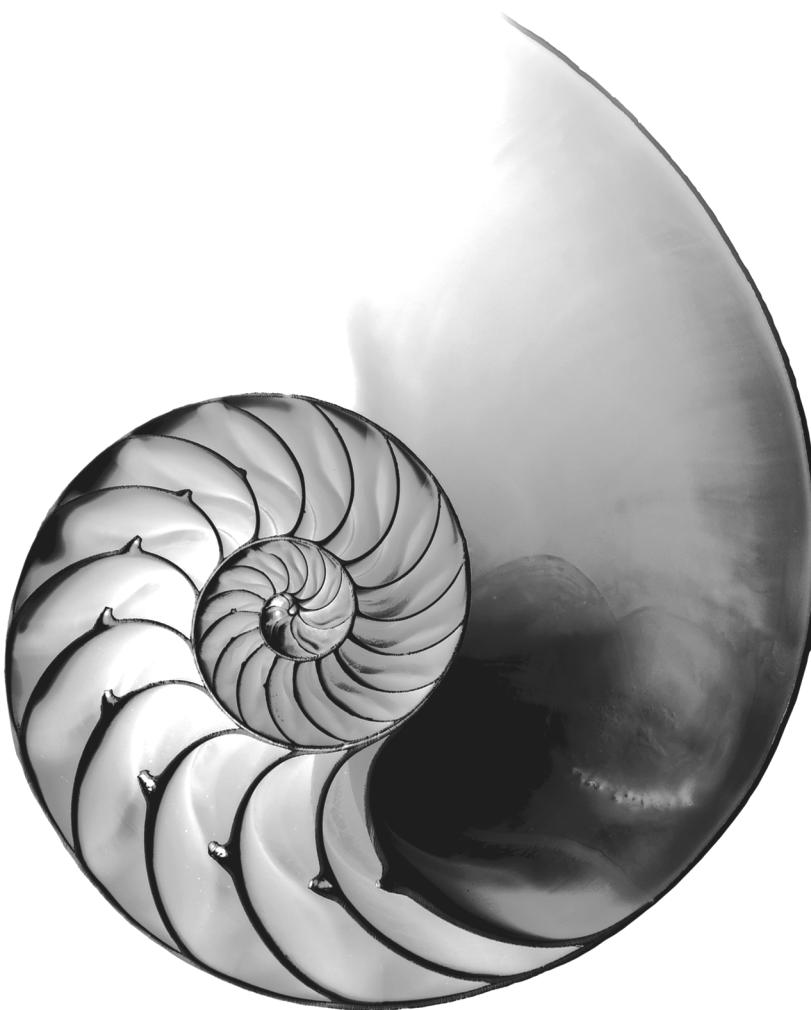
Fig. 26.50 | Typing tutor.

You can test your program by typing a pangram—a phrase that contains every letter of the alphabet at least once—such as “The quick brown fox jumped over a lazy dog.” You can find other pangrams on the web.

To make the program more interesting you could monitor the user’s accuracy. You could have the user type specific phrases that you’ve prestored in your program and that you display on the screen above the virtual keyboard. You could keep track of how many keystrokes the user types correctly and how many are typed incorrectly. You could also keep track of which keys the user is having difficulty with and display a report showing those keys.

27

Graphics and Java 2D



Objectives

In this chapter you'll:

- Understand graphics contexts and graphics objects.
- Manipulate colors and fonts.
- Use methods of class `Graphics` to draw various shapes.
- Use methods of class `Graphics2D` from the Java 2D API to draw various shapes.
- Specify `Paint` and `Stroke` characteristics of shapes displayed with `Graphics2D`.

Outline

- | | |
|-----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 27.1 Introduction
27.2 Graphics Contexts and Graphics Objects
27.3 Color Control
27.4 Manipulating Fonts | 27.5 Drawing Lines, Rectangles and Ovals
27.6 Drawing Arcs
27.7 Drawing Polygons and Polylines
27.8 Java 2D API
27.9 Wrap-Up |
|-----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

27.1 Introduction

[Note: JavaFX (Chapters 12, 13 and 22) is Java's GUI, graphics and multimedia API of the future. This chapter is provided *as is* for those still interested in Swing GUIs.]

In this chapter, we overview several of Java's capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts. Part of Java's initial appeal was its support for graphics that enabled programmers to visually enhance their applications. Java contains more sophisticated drawing capabilities as part of the Java 2D API (presented in this chapter) and its successor technology JavaFX (presented in Chapter 12 and two online chapters). This chapter begins by introducing many of Java's original drawing capabilities. Next we present several of the more powerful Java 2D capabilities, such as controlling the *style* of lines used to draw shapes and the way shapes are *filled* with *colors* and *patterns*. The classes that were part of Java's original graphics capabilities are now considered to be part of the Java 2D API.

Figure 27.1 shows a portion of the class hierarchy that includes various graphics classes and Java 2D API classes and interfaces covered in this chapter. Class **Color** contains methods and constants for manipulating colors. Class **JComponent** contains method **paintComponent**, which is used to draw graphics on a component. Class **Font** contains methods and constants for manipulating fonts. Class **FontMetrics** contains methods for obtaining *font* information. Class **Graphics** contains methods for drawing strings, lines, rectangles and other shapes. Class **Graphics2D**, which extends class **Graphics**, is used for drawing with the Java 2D API. Class **Polygon** contains methods for creating *polygons*. The bottom half of the figure lists several classes and interfaces from the Java 2D API. Class **BasicStroke** helps specify the drawing characteristics of *lines*. Classes **GradientPaint** and **TexturePaint** help specify the characteristics for filling *shapes* with *colors* or *patterns*. Classes **GeneralPath**, **Line2D**, **Arc2D**, **Ellipse2D**, **Rectangle2D** and **RoundRectangle2D** represent several Java 2D shapes.

To begin drawing in Java, we must first understand Java's **coordinate system** (Fig. 27.2), which is a scheme for identifying every *point* on the screen. By default, the *upper-left corner* of a GUI component (e.g., a window) has the coordinates (0, 0). A coordinate pair is composed of an **x-coordinate** (the **horizontal coordinate**) and a **y-coordinate** (the **vertical coordinate**). The **x-coordinate** is the horizontal distance moving *right* from the left edge of the screen. The **y-coordinate** is the vertical distance moving *down* from the *top* of the screen. The **x-axis** describes every horizontal coordinate, and the **y-axis** every vertical coordinate. The coordinates are used to indicate where graphics should be displayed on a screen. Coordinate units are measured in **pixels** (which stands for "picture elements"). A pixel is a display monitor's *smallest unit of resolution*.

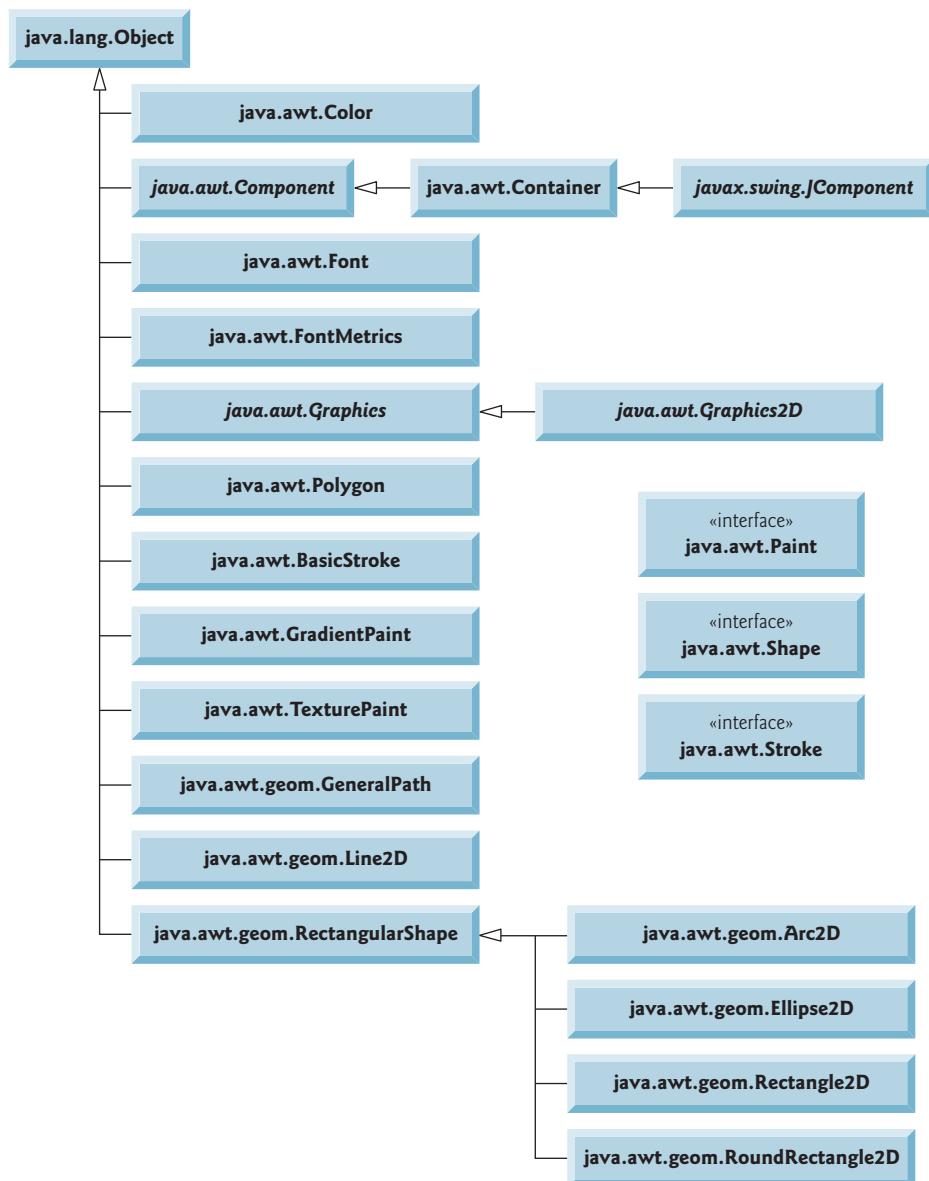


Fig. 27.1 | Classes and interfaces used in this chapter from Java's original graphics capabilities and from the Java 2D API.



Portability Tip 27.1

Different display monitors have different resolutions (i.e., the density of the pixels varies). This can cause graphics to appear in different sizes on different monitors or on the same monitor with different settings.

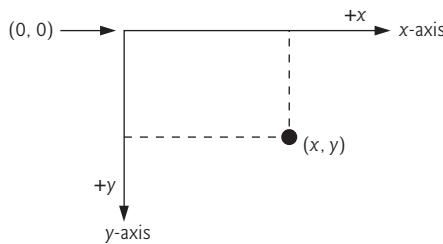


Fig. 27.2 | Java coordinate system. Units are measured in pixels.

27.2 Graphics Contexts and Graphics Objects

A **graphics context** enables drawing on the screen. A **Graphics** object manages a graphics context and draws pixels on the screen that represent *text* and other graphical objects (e.g., *lines*, *ellipses*, *rectangles* and other *polygons*). **Graphics** objects contain methods for *drawing*, *font manipulation*, *color manipulation* and the like.

Class **Graphics** is an abstract class (i.e., you cannot instantiate **Graphics** objects). This contributes to Java's portability. Because drawing is performed *differently* on every platform that supports Java, there cannot be only one implementation of the drawing capabilities across all systems. When Java is implemented on a particular platform, a sub-class of **Graphics** is created that implements the drawing capabilities. This implementation is hidden by class **Graphics**, which supplies the interface that enables us to use graphics in a *platform-independent* manner.

Recall from Chapter 26 that class **Component** is the *superclass* for many of the classes in package `java.awt`. Class **JComponent** (package `javax.swing`), which inherits indirectly from class **Component**, contains a `paintComponent` method that can be used to draw graphics. Method `paintComponent` takes a **Graphics** object as an argument. This object is passed to the `paintComponent` method by the system when a lightweight Swing component needs to be repainted. The header for the `paintComponent` method is

```
public void paintComponent(Graphics g)
```

Parameter `g` receives a reference to an instance of the system-specific subclass of **Graphics**. The preceding method header should look familiar to you—it's the same one we used in some of the applications in Chapter 26. Actually, class **JComponent** is a *superclass* of **JPanel**. Many capabilities of class **JPanel** are inherited from class **JComponent**.

You seldom call method `paintComponent` directly, because drawing graphics is an *event-driven* process. As we mentioned in Chapter 11, Java uses a *multithreaded* model of program execution. Each thread is a *parallel* activity. Each program can have many threads. When you create a GUI-based application, one of those threads is known as the **event-dispatch thread (EDT)**—it's used to process all GUI events. All manipulation of the on-screen GUI components must be performed in that thread. When a GUI application executes, the application container calls method `paintComponent` (in the event-dispatch thread) for each lightweight component as the GUI is displayed. For `paintComponent` to be called again, an event must occur (such as *covering* and *uncovering* the component with another window).

If you need `paintComponent` to execute (i.e., if you want to update the graphics drawn on a Swing component), you can call method `repaint`, which returns `void`, takes no arguments and is inherited by all `JComponents` indirectly from class `Component` (package `java.awt`).

27.3 Color Control

Class `Color` declares methods and constants for manipulating colors in a Java program. The predeclared color constants are summarized in Fig. 27.3, and several color methods and constructors are summarized in Fig. 27.4. Two of the methods in Fig. 27.4 are `Graphics` methods that are specific to colors.

Color constant	RGB value
<code>public static final Color RED</code>	255, 0, 0
<code>public static final Color GREEN</code>	0, 255, 0
<code>public static final Color BLUE</code>	0, 0, 255
<code>public static final Color ORANGE</code>	255, 200, 0
<code>public static final Color PINK</code>	255, 175, 175
<code>public static final Color CYAN</code>	0, 255, 255
<code>public static final Color MAGENTA</code>	255, 0, 255
<code>public static final Color YELLOW</code>	255, 255, 0
<code>public static final Color BLACK</code>	0, 0, 0
<code>public static final Color WHITE</code>	255, 255, 255
<code>public static final Color GRAY</code>	128, 128, 128
<code>public static final Color LIGHT_GRAY</code>	192, 192, 192
<code>public static final Color DARK_GRAY</code>	64, 64, 64

Fig. 27.3 | Color constants and their RGB values.

Method	Description
<i>Color constructors and methods</i>	
<code>public Color(int r, int g, int b)</code>	Creates a color based on red, green and blue components expressed as integers from 0 to 255.
<code>public Color(float r, float g, float b)</code>	Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0.
<code>public int getRed()</code>	Returns a value between 0 and 255 representing the red content.

Fig. 27.4 | Color methods and color-related `Graphics` methods. (Part 1 of 2.)

Method	Description
<code>public int getGreen()</code>	Returns a value between 0 and 255 representing the green content.
<code>public int getBlue()</code>	Returns a value between 0 and 255 representing the blue content.
<i>Graphics methods for manipulating Colors</i>	
<code>public Color getColor()</code>	Returns <code>Color</code> object representing current color for the graphics context.
<code>public void setColor(Color c)</code>	Sets the current color for drawing with the graphics context.

Fig. 27.4 | Color methods and color-related `Graphics` methods. (Part 2 of 2.)

Every color is created from a red, a green and a blue value. Together these are called **RGB values**. All three RGB components can be integers in the range from 0 to 255, or all three can be floating-point values in the range 0.0 to 1.0. The first RGB component specifies the amount of red, the second the amount of green and the third the amount of blue. The larger the value, the greater the amount of that particular color. Java enables you to choose from $256 \times 256 \times 256$ (approximately 16.7 million) colors. Not all computers are capable of displaying all these colors. The screen will display the closest color it can.

Two of class `Color`'s constructors are shown in Fig. 27.4—one that takes three `int` arguments and one that takes three `float` arguments, with each argument specifying the amount of red, green and blue. The `int` values must be in the range 0–255 and the `float` values in the range 0.0–1.0. The new `Color` object will have the specified amounts of red, green and blue. `Color` methods `getRed`, `getGreen` and `getBlue` return integer values from 0 to 255 representing the amounts of red, green and blue, respectively. `Graphics` method `getColor` returns a `Color` object representing the `Graphics` object's current drawing color. `Graphics` method `setColor` sets the current drawing color.

Drawing in Different Colors

Figures 27.5–27.6 demonstrate several methods from Fig. 27.4 by drawing *filled rectangles* and `String`s in several different colors. When the application begins execution, class `ColorJPanel`'s `paintComponent` method (lines 10–37 of Fig. 27.5) is called to paint the window. Line 17 uses `Graphics` method `setColor` to set the drawing color. Method `setColor` receives a `Color` object. The expression `new Color(255, 0, 0)` creates a new `Color` object that represents red (red value 255, and 0 for the green and blue values). Line 18 uses `Graphics` method `fillRect` to draw a *filled rectangle* in the current color. Method `fillRect` draws a rectangle based on its four arguments. The first two integer values represent the upper-left *x*-coordinate and upper-left *y*-coordinate, where the `Graphics` object begins drawing the rectangle. The third and fourth arguments are nonnegative integers that represent the width and the height of the rectangle in pixels, respectively. A rectangle drawn using method `fillRect` is filled by the current color of the `Graphics` object.

Line 19 uses `Graphics` method `drawString` to draw a `String` in the current color. The expression `g.getColor()` retrieves the current color from the `Graphics` object. We

```

1 // Fig. 13.5: ColorJPanel.java
2 // Changing drawing colors.
3 import java.awt.Graphics;
4 import java.awt.Color;
5 import javax.swing.JPanel;
6
7 public class ColorJPanel extends JPanel
8 {
9     // draw rectangles and Strings in different colors
10    @Override
11    public void paintComponent(Graphics g)
12    {
13        super.paintComponent(g);
14        this.setBackground(Color.WHITE);
15
16        // set new drawing color using integers
17        g.setColor(new Color(255, 0, 0));
18        g.fillRect(15, 25, 100, 20);
19        g.drawString("Current RGB: " + g.getColor(), 130, 40);
20
21        // set new drawing color using floats
22        g.setColor(new Color(0.50f, 0.75f, 0.0f));
23        g.fillRect(15, 50, 100, 20);
24        g.drawString("Current RGB: " + g.getColor(), 130, 65);
25
26        // set new drawing color using static Color objects
27        g.setColor(Color.BLUE);
28        g.fillRect(15, 75, 100, 20);
29        g.drawString("Current RGB: " + g.getColor(), 130, 90);
30
31        // display individual RGB values
32        Color color = Color.MAGENTA;
33        g.setColor(color);
34        g.fillRect(15, 100, 100, 20);
35        g.drawString("RGB values: " + color.getRed() + ", " +
36                     color.getGreen() + ", " + color.getBlue(), 130, 115);
37    }
38 } // end class ColorJPanel

```

Fig. 27.5 | Changing drawing colors.

```

1 // Fig. 13.6: ShowColors.java
2 // Demonstrating Colors.
3 import javax.swing.JFrame;
4
5 public class ShowColors
6 {
7     // execute application
8     public static void main(String[] args)
9     {
10         // create frame for ColorJPanel
11         JFrame frame = new JFrame("Using colors");

```

Fig. 27.6 | Demonstrating Colors. (Part I of 2.)

```

12     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14     ColorJPanel colorJPanel = new ColorJPanel();
15     frame.add(colorJPanel);
16     frame.setSize(400, 180);
17     frame.setVisible(true);
18 }
19 } // end class ShowColors

```

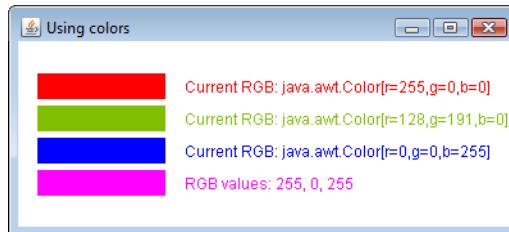


Fig. 27.6 | Demonstrating Colors. (Part 2 of 2.)

then concatenate the `Color` with string "Current RGB: ", resulting in an *implicit* call to class `Color`'s `toString` method. The String representation of a `Color` contains the class name and package (`java.awt.Color`) and the red, green and blue values.



Look-and-Feel Observation 27.1

People perceive colors differently. Choose your colors carefully to ensure that your application is readable, both for people who can perceive color and for those who are color blind. Try to avoid using many different colors in close proximity.

Lines 22–24 and 27–29 perform the same tasks again. Line 22 uses the `Color` constructor with three `float` arguments to create a dark green color (0.50f for red, 0.75f for green and 0.0f for blue). Note the syntax of the values. The letter `f` appended to a floating-point literal indicates that the literal should be treated as type `float`. Recall that by default, floating-point literals are treated as type `double`.

Line 27 sets the current drawing color to one of the predeclared `Color` constants (`Color.BLUE`). The `Color` constants are `static`, so they're created when class `Color` is loaded into memory at execution time.

The statement in lines 35–36 makes calls to `Color` methods `getRed`, `getGreen` and `getBlue` on the predeclared `Color.MAGENTA` constant. Method `main` of class `ShowColors` (lines 8–18 of Fig. 27.6) creates the `JFrame` that will contain a `ColorJPanel` object where the colors will be displayed.



Software Engineering Observation 27.1

To change the color, you must create a new `Color` object (or use one of the predeclared `Color` constants). Like `String` objects, `Color` objects are immutable (not modifiable).

The `JColorChooser` component (package `javax.swing`) enables application users to select colors. Figures 27.7–27.8 demonstrate a `JColorChooser` dialog. When you click the

Change Color button, a JColorChooser dialog appears. When you select a color and press the dialog's OK button, the background color of the application window changes.

```
1 // Fig. 13.7: ShowColors2JFrame.java
2 // Choosing colors with JColorChooser.
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import java.awt.event.ActionEvent;
6 import java.awt.event.ActionListener;
7 import javax.swing.JButton;
8 import javax.swing.JFrame;
9 import javax.swing.JColorChooser;
10 import javax.swing.JPanel;
11
12 public class ShowColors2JFrame extends JFrame
13 {
14     private final JButton changeColor JButton;
15     private Color color = Color.LIGHT_GRAY;
16     private final JPanel color JPanel;
17
18     // set up GUI
19     public ShowColors2JFrame()
20     {
21         super("Using JColorChooser");
22
23         // create JPanel for display color
24         color JPanel = new JPanel();
25         color JPanel.setBackground(color);
26
27         // set up changeColor JButton and register its event handler
28         changeColor JButton = new JButton("Change Color");
29         changeColor JButton.addActionListener(
30             new ActionListener() // anonymous inner class
31             {
32                 // display JColorChooser when user clicks button
33                 @Override
34                 public void actionPerformed(ActionEvent event)
35                 {
36                     color = JColorChooser.showDialog(
37                         ShowColors2JFrame.this, "Choose a color", color);
38
39                     // set default color, if no color is returned
40                     if (color == null)
41                         color = Color.LIGHT_GRAY;
42
43                     // change content pane's background color
44                     color JPanel.setBackground(color);
45                 } // end method actionPerformed
46             } // end anonymous inner class
47         ); // end call to addActionListener
48
49         add(color JPanel, BorderLayout.CENTER);
```

Fig. 27.7 | Choosing colors with JColorChooser. (Part I of 2.)

```

50     add(changeColorJButton, BorderLayout.SOUTH);
51
52     setSize(400, 130);
53     setVisible(true);
54 } // end ShowColor2JFrame constructor
55 } // end class ShowColors2JFrame

```

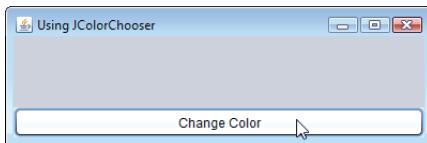
Fig. 27.7 | Choosing colors with JColorChooser. (Part 2 of 2.)

```

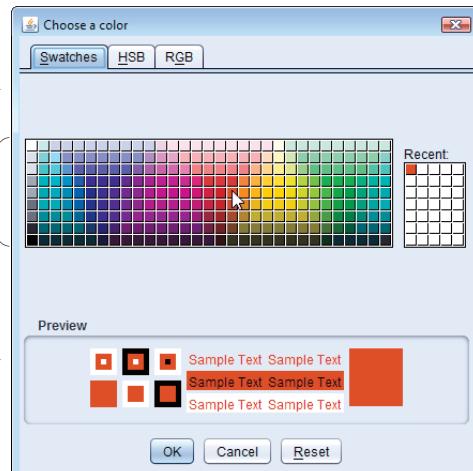
1 // Fig. 13.8: ShowColors2.java
2 // Choosing colors with JColorChooser.
3 import javax.swing.JFrame;
4
5 public class ShowColors2
6 {
7     // execute application
8     public static void main(String[] args)
9     {
10         ShowColors2JFrame application = new ShowColors2JFrame();
11         application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12     }
13 } // end class ShowColors2

```

(a) Initial application window



(b) JColorChooser window



(c) Application window after changing JPanel's background color



Fig. 27.8 | Choosing colors with JColorChooser.

Class `JColorChooser` provides static method `showDialog`, which creates a `JColorChooser` object, attaches it to a dialog box and displays the dialog. Lines 36–37 of Fig. 27.7 invoke this method to display the color-chooser dialog. Method `showDialog` returns the selected `Color` object, or `null` if the user presses `Cancel` or closes the dialog without pressing `OK`. The method takes three arguments—a reference to its parent `Component`, a

`String` to display in the title bar of the dialog and the initial selected `Color` for the dialog. The parent component is a reference to the window from which the dialog is displayed (in this case the `JFrame`, with the reference name `frame`). The dialog will be centered on the parent. If the parent is `null`, the dialog is centered on the screen. While the color-chooser dialog is on the screen, the user cannot interact with the parent component until the dialog is dismissed. This type of dialog is called a modal dialog.

After the user selects a color, lines 40–41 determine whether `color` is `null`, and, if so, set `color` to `Color.LIGHT_GRAY`. Line 44 invokes method `setBackground` to change the background color of the `JPanel1`. Method `setBackground` is one of the many `Component` methods that can be used on most GUI components. The user can continue to use the **Change Color** button to change the background color of the application. Figure 27.8 contains method `main`, which executes the program.

Figure 27.8(b) shows the default `JColorChooser` dialog that allows the user to select a color from a variety of **color swatches**. There are three tabs across the top of the dialog—**Swatches**, **HSB** and **RGB**. These represent three different ways to select a color. The **HSB** tab allows you to select a color based on **hue**, **saturation** and **brightness**—values that are used to define the amount of light in a color. Visit http://en.wikipedia.org/wiki/HSL_and_HSV for more information on HSB. The **RGB** tab allows you to select a color by using sliders to select the red, green and blue components. The **HSB** and **RGB** tabs are shown in Fig. 27.9.

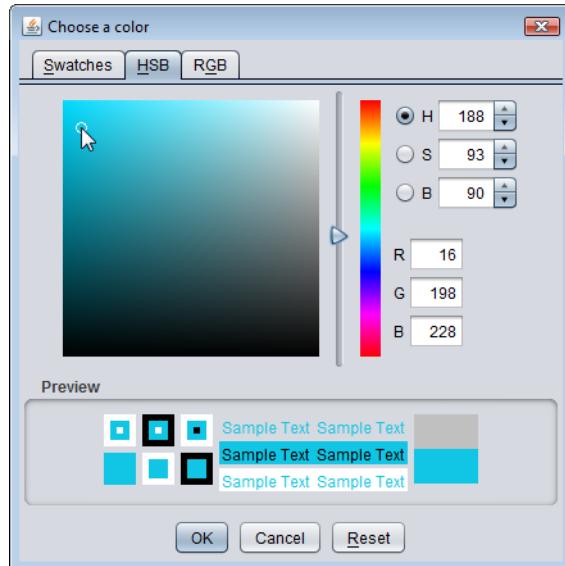


Fig. 27.9 | HSB and RGB tabs of the `JColorChooser` dialog. (Part I of 2.)

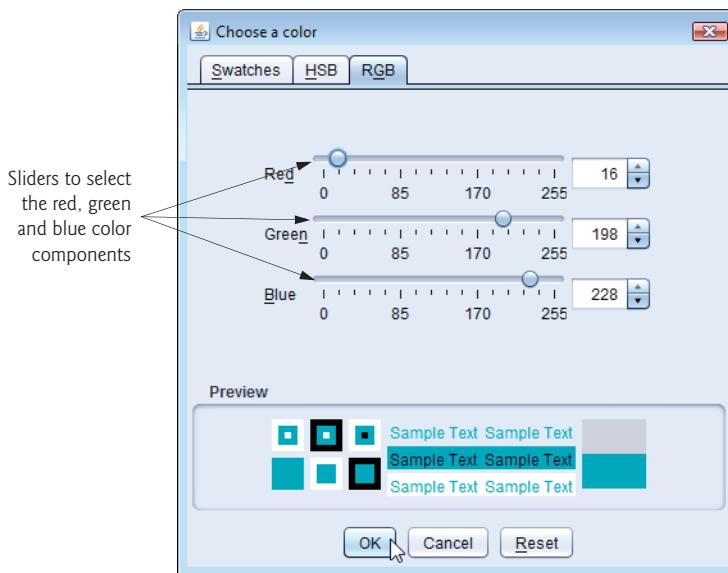


Fig. 27.9 | HSB and RGB tabs of the JColorChooser dialog. (Part 2 of 2.)

27.4 Manipulating Fonts

This section introduces methods and constants for manipulating fonts. Most font methods and font constants are part of class `Font`. Some constructors, methods and constants of class `Font` and class `Graphics` are summarized in Fig. 27.10.

Method or constant	Description
<i>Font constants, constructors and methods</i>	
<code>public static final int PLAIN</code>	A constant representing a plain font style.
<code>public static final int BOLD</code>	A constant representing a bold font style.
<code>public static final int ITALIC</code>	A constant representing an italic font style.
<code>public Font(String name, int style, int size)</code>	Creates a <code>Font</code> object with the specified font name, style and size.
<code>public int getStyle()</code>	Returns an <code>int</code> indicating the current font style.
<code>public int getSize()</code>	Returns an <code>int</code> indicating the current font size.
<code>public String getName()</code>	Returns the current font name as a string.
<code>public String getFamily()</code>	Returns the font's family name as a string.
<code>public boolean isPlain()</code>	Returns <code>true</code> if the font is plain, else <code>false</code> .
<code>public boolean isBold()</code>	Returns <code>true</code> if the font is bold, else <code>false</code> .
<code>public boolean isItalic()</code>	Returns <code>true</code> if the font is italic, else <code>false</code> .

Fig. 27.10 | Font-related methods and constants. (Part 1 of 2.)

Method or constant	Description
<i>Graphics methods for manipulating Fonts</i>	
<code>public Font getFont()</code>	Returns a Font object reference representing the current font.
<code>public void setFont(Font f)</code>	Sets the current font to the font, style and size specified by the Font object reference f.

Fig. 27.10 | Font-related methods and constants. (Part 2 of 2.)

Class `Font`'s constructor takes three arguments—the **font name**, **font style** and **font size**. The font name is any font currently supported by the system on which the program is running, such as standard Java fonts `Monospaced`, `SansSerif` and `Serif`. The font style is `Font.PLAIN`, `Font.ITALIC` or `Font.BOLD` (each is a `static` field of class `Font`). Font styles can be used in combination (e.g., `Font.ITALIC + Font.BOLD`). The font size is measured in points. A **point** is $1/72$ of an inch. `Graphics` method `setFont` sets the current drawing font—the font in which text will be displayed—to its `Font` argument.



Portability Tip 27.2

The number of fonts varies across systems. Java provides five font names—`Serif`, `Monospaced`, `SansSerif`, `Dialog` and `DialogInput`—that can be used on all Java platforms. The Java runtime environment (JRE) on each platform maps these logical font names to actual fonts installed on the platform. The actual fonts used may vary by platform.

The application of Figs. 27.11–27.12 displays text in four different fonts, with each font in a different size. Figure 27.11 uses the `Font` constructor to initialize `Font` objects (in lines 17, 21, 25 and 30) that are each passed to `Graphics` method `setFont` to change the drawing font. Each call to the `Font` constructor passes a font name (`Serif`, `Monospaced` or `SansSerif`) as a string, a font style (`Font.PLAIN`, `Font.ITALIC` or `Font.BOLD`) and a font size. Once `Graphics` method `setFont` is invoked, all text displayed following the call will appear in the new font until the font is changed. Each font's information is displayed in lines 18, 22, 26 and 31–32 using method `drawString`. The coordinates passed to `drawString` correspond to the lower-left corner of the baseline of the font. Line 29 changes the drawing color to red, so the next string displayed appears in red. Lines 31–32 display information about the final `Font` object. Method `getFont` of class `Graphics` returns a `Font` object representing the current font. Method `getName` returns the current font name as a string. Method `getSize` returns the font size in points.



Software Engineering Observation 27.2

To change the font, you must create a new `Font` object. `Font` objects are immutable—class `Font` has no set methods to change the characteristics of the current font.

```

1 // Fig. 13.11: FontJPanel.java
2 // Display strings in different fonts and colors.
3 import java.awt.Font;

```

Fig. 27.11 | Display strings in different fonts and colors. (Part 1 of 2.)

```
4 import java.awt.Color;
5 import java.awt.Graphics;
6 import javax.swing.JPanel;
7
8 public class Font JPanel extends JPanel
9 {
10    // display Strings in different fonts and colors
11    @Override
12    public void paintComponent(Graphics g)
13    {
14        super.paintComponent(g);
15
16        // set font to Serif (Times), bold, 12pt and draw a string
17        g.setFont(new Font("Serif", Font.BOLD, 12));
18        g.drawString("Serif 12 point bold.", 20, 30);
19
20        // set font to Monospaced (Courier), italic, 24pt and draw a string
21        g.setFont(new Font("Monospaced", Font.ITALIC, 24));
22        g.drawString("Monospaced 24 point italic.", 20, 50);
23
24        // set font to SansSerif (Helvetica), plain, 14pt and draw a string
25        g.setFont(new Font("SansSerif", Font.PLAIN, 14));
26        g.drawString("SansSerif 14 point plain.", 20, 70);
27
28        // set font to Serif (Times), bold/italic, 18pt and draw a string
29        g.setColor(Color.RED);
30        g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 18));
31        g.drawString(g.getFont().getName() + " " + g.getFont().getSize() +
32            " point bold italic.", 20, 90);
33    }
34 } // end class Font JPanel
```

Fig. 27.11 | Display strings in different fonts and colors. (Part 2 of 2.)

Figure 27.12 contains the `main` method, which creates a `JFrame` to display a `Font JPanel`. We add a `Font JPanel` object to this `JFrame` (line 15), which displays the graphics created in Fig. 27.11.

```
1 // Fig. 13.12: Fonts.java
2 // Using fonts.
3 import javax.swing.JFrame;
4
5 public class Fonts
6 {
7     // execute application
8     public static void main(String[] args)
9     {
10        // create frame for Font JPanel
11        JFrame frame = new JFrame("Using fonts");
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13    }
}
```

Fig. 27.12 | Using fonts. (Part 1 of 2.)

```

14     JPanel fontJPanel = new JPanel();
15     frame.add(fontJPanel);
16     frame.setSize(420, 150);
17     frame.setVisible(true);
18 }
19 } // end class Fonts

```

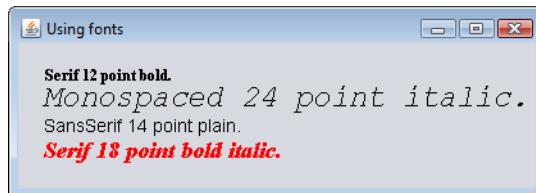


Fig. 27.12 | Using fonts. (Part 2 of 2.)

Font Metrics

Sometimes it's necessary to get information about the current drawing font, such as its name, style and size. Several Font methods used to get font information are summarized in Fig. 27.10. Method `getStyle` returns an integer value representing the current style. The integer value returned is either `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` or the combination of `Font.ITALIC` and `Font.BOLD`. Method `getFamily` returns the name of the font family to which the current font belongs. The name of the font family is platform specific. Font methods are also available to test the style of the current font, and these too are summarized in Fig. 27.10. Methods `isPlain`, `isBold` and `isItalic` return `true` if the current font style is plain, bold or italic, respectively.

Figure 27.13 illustrates some of the common **font metrics**, which provide precise information about a font, such as **height**, **descent** (the amount a character dips below the baseline), **ascent** (the amount a character rises above the baseline) and **leading** (the difference between the descent of one line of text and the ascent of the line of text below it—that is, the interline spacing).

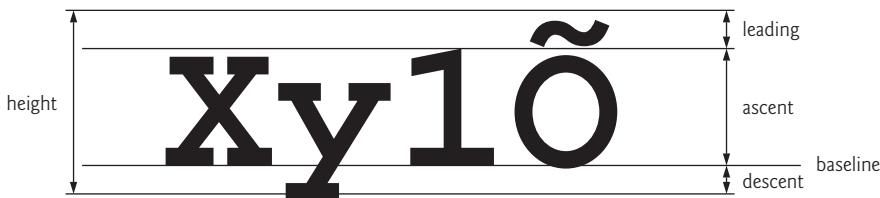


Fig. 27.13 | Font metrics.

Class `FontMetrics` declares several methods for obtaining font metrics. These methods and `Graphics` method `getFontMetrics` are summarized in Fig. 27.14. The application of Figs. 27.15–27.16 uses the methods of Fig. 27.14 to obtain font metric information for two fonts.

Method	Description
<i>FontMetrics methods</i>	
<code>public int getAscent()</code>	Returns the ascent of a font in points.
<code>public int getDescent()</code>	Returns the descent of a font in points.
<code>public int getLeading()</code>	Returns the leading of a font in points.
<code>public int getHeight()</code>	Returns the height of a font in points.
<i>Graphics methods for getting a Font's FontMetrics</i>	
<code>public FontMetrics getFontMetrics()</code>	Returns the <code>FontMetrics</code> object for the current drawing <code>Font</code> .
<code>public FontMetrics getFontMetrics(Font f)</code>	Returns the <code>FontMetrics</code> object for the specified <code>Font</code> argument.

Fig. 27.14 | `FontMetrics` and `Graphics` methods for obtaining font metrics.

```

1 // Fig. 13.15: MetricsJPanel.java
2 // FontMetrics and Graphics methods useful for obtaining font metrics.
3 import java.awt.Font;
4 import java.awt.FontMetrics;
5 import java.awt.Graphics;
6 import javax.swing.JPanel;
7
8 public class MetricsJPanel extends JPanel
9 {
10    // display font metrics
11    @Override
12    public void paintComponent(Graphics g)
13    {
14        super.paintComponent(g);
15
16        g.setFont(new Font("SansSerif", Font.BOLD, 12));
17        FontMetrics metrics = g.getFontMetrics();
18        g.drawString("Current font: " + g.getFont(), 10, 30);
19        g.drawString("Ascent: " + metrics.getAscent(), 10, 45);
20        g.drawString("Descent: " + metrics.getDescent(), 10, 60);
21        g.drawString("Height: " + metrics.getHeight(), 10, 75);
22        g.drawString("Leading: " + metrics.getLeading(), 10, 90);
23
24        Font font = new Font("Serif", Font.ITALIC, 14);
25        metrics = g.getFontMetrics(font);
26        g.setFont(font);
27        g.drawString("Current font: " + font, 10, 120);
28        g.drawString("Ascent: " + metrics.getAscent(), 10, 135);
29        g.drawString("Descent: " + metrics.getDescent(), 10, 150);
30        g.drawString("Height: " + metrics.getHeight(), 10, 165);
31        g.drawString("Leading: " + metrics.getLeading(), 10, 180);
32    }
33 } // end class MetricsJPanel

```

Fig. 27.15 | `FontMetrics` and `Graphics` methods useful for obtaining font metrics.

```
1 // Fig. 13.16: Metrics.java
2 // Displaying font metrics.
3 import javax.swing.JFrame;
4
5 public class Metrics
6 {
7     // execute application
8     public static void main(String[] args)
9     {
10         // create frame for MetricsJPanel
11         JFrame frame = new JFrame("Demonstrating FontMetrics");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14         MetricsJPanel metricsJPanel = new MetricsJPanel();
15         frame.add(metricsJPanel);
16         frame.setSize(510, 240);
17         frame.setVisible(true);
18     }
19 } // end class Metrics
```

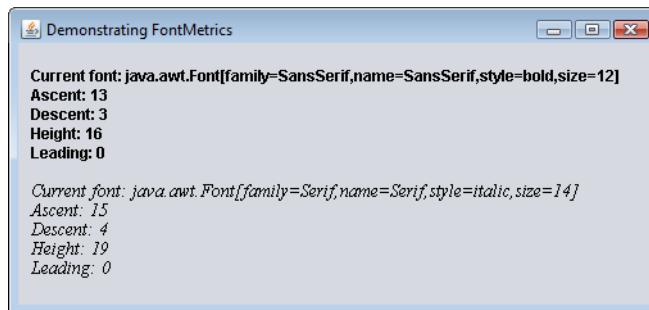


Fig. 27.16 | Displaying font metrics.

Line 16 of Fig. 27.15 creates and sets the current drawing font to a SansSerif, bold, 12-point font. Line 17 uses `Graphics` method `getFontMetrics` to obtain the `FontMetrics` object for the current font. Line 18 outputs the `String` representation of the `Font` returned by `g.getFont()`. Lines 19–22 use `FontMetric` methods to obtain the ascent, descent, height and leading for the font.

Line 24 creates a new Serif, italic, 14-point font. Line 25 uses a second version of `Graphics` method `getFontMetrics`, which accepts a `Font` argument and returns a corresponding `FontMetrics` object. Lines 28–31 obtain the ascent, descent, height and leading for the font. The font metrics are slightly different for the two fonts.

27.5 Drawing Lines, Rectangles and Ovals

This section presents `Graphics` methods for drawing lines, rectangles and ovals. The methods and their parameters are summarized in Fig. 27.17. For each drawing method that requires a `width` and `height` parameter, the `width` and `height` must be nonnegative values. Otherwise, the shape will not display.

Method	Description
<code>public void drawLine(int x1, int y1, int x2, int y2)</code>	Draws a line between the point (x1, y1) and the point (x2, y2).
<code>public void drawRect(int x, int y, int width, int height)</code>	Draws a rectangle of the specified width and height. The rectangle's top-left corner is located at (x, y). Only the outline of the rectangle is drawn using the Graphics object's color—the body of the rectangle is not filled with this color.
<code>public void fillRect(int x, int y, int width, int height)</code>	Draws a filled rectangle in the current color with the specified width and height. The rectangle's top-left corner is located at (x, y).
<code>public void clearRect(int x, int y, int width, int height)</code>	Draws a filled rectangle with the specified width and height in the current background color. The rectangle's <i>top-left</i> corner is located at (x, y). This method is useful if you want to remove a portion of an image.
<code>public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Draws a rectangle with rounded corners in the current color with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners (see Fig. 27.20). Only the outline of the shape is drawn.
<code>public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</code>	Draws a filled rectangle in the current color with rounded corners with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners (see Fig. 27.20).
<code>public void draw3DRect(int x, int y, int width, int height, boolean b)</code>	Draws a three-dimensional rectangle in the current color with the specified width and height. The rectangle's <i>top-left</i> corner is located at (x, y). The rectangle appears raised when b is true and lowered when b is false. Only the outline of the shape is drawn.
<code>public void fill3DRect(int x, int y, int width, int height, boolean b)</code>	Draws a filled three-dimensional rectangle in the current color with the specified width and height. The rectangle's <i>top-left</i> corner is located at (x, y). The rectangle appears raised when b is true and lowered when b is false.
<code>public void drawOval(int x, int y, int width, int height)</code>	Draws an oval in the current color with the specified width and height. The bounding rectangle's <i>top-left</i> corner is located at (x, y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 27.21). Only the outline of the shape is drawn.
<code>public void fillOval(int x, int y, int width, int height)</code>	Draws a filled oval in the current color with the specified width and height. The bounding rectangle's <i>top-left</i> corner is located at (x, y). The oval touches the center of all four sides of the bounding rectangle (see Fig. 27.21).

Fig. 27.17 | Graphics methods that draw lines, rectangles and ovals.

The application of Figs. 27.18–27.19 demonstrates drawing a variety of lines, rectangles, three-dimensional rectangles, rounded rectangles and ovals. In Fig. 27.18, line 17 draws a red line, line 20 draws an empty blue rectangle and line 21 draws a filled blue rectangle. Methods `fillRoundRect` (line 24) and `drawRoundRect` (line 25) draw rectangles with rounded corners. Their first two arguments specify the coordinates of the upper-left corner of the **bounding rectangle**—the area in which the rounded rectangle will be drawn. The upper-left corner coordinates are *not* the edge of the rounded rectangle, but the coordinates where the edge would be if the rectangle had square corners. The third and fourth arguments specify the width and height of the rectangle. The last two arguments determine the horizontal and vertical diameters of the arc (i.e., the arc width and arc height) used to represent the corners.

```
1 // Fig. 13.18: LinesRectsOvalsJPanel.java
2 // Drawing lines, rectangles and ovals.
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import javax.swing.JPanel;
6
7 public class LinesRectsOvalsJPanel extends JPanel
8 {
9     // display various lines, rectangles and ovals
10    @Override
11    public void paintComponent(Graphics g)
12    {
13        super.paintComponent(g);
14        this.setBackground(Color.WHITE);
15
16        g.setColor(Color.RED);
17        g.drawLine(5, 30, 380, 30);
18
19        g.setColor(Color.BLUE);
20        g.drawRect(5, 40, 90, 55);
21        g.fillRect(100, 40, 90, 55);
22
23        g.setColor(Color.CYAN);
24        g.fillRoundRect(195, 40, 90, 55, 50, 50);
25        g.drawRoundRect(290, 40, 90, 55, 20, 20);
26
27        g.setColor(Color.GREEN);
28        g.draw3DRect(5, 100, 90, 55, true);
29        g.fill3DRect(100, 100, 90, 55, false);
30
31        g.setColor(Color.MAGENTA);
32        g.drawOval(195, 100, 90, 55);
33        g.fillOval(290, 100, 90, 55);
34    }
35 } // end class LinesRectsOvalsJPanel
```

Fig. 27.18 | Drawing lines, rectangles and ovals.

```

1 // Fig. 13.19: LinesRectsOvals.java
2 // Testing LinesRectsOvalsJPanel.
3 import java.awt.Color;
4 import javax.swing.JFrame;
5
6 public class LinesRectsOvals
7 {
8     // execute application
9     public static void main(String[] args)
10    {
11        // create frame for LinesRectsOvalsJPanel
12        JFrame frame =
13            new JFrame("Drawing lines, rectangles and ovals");
14        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16        LinesRectsOvalsJPanel linesRectsOvalsJPanel =
17            new LinesRectsOvalsJPanel();
18        linesRectsOvalsJPanel.setBackground(Color.WHITE);
19        frame.add(linesRectsOvalsJPanel);
20        frame.setSize(400, 210);
21        frame.setVisible(true);
22    }
23 } // end class LinesRectsOvals

```

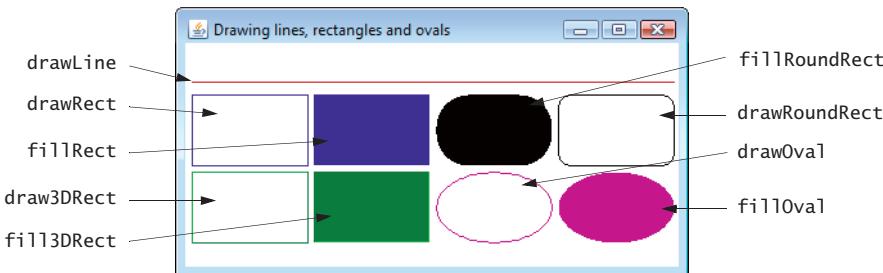


Fig. 27.19 | Testing LinesRectsOvalsJPanel.

Figure 27.20 labels the arc width, arc height, width and height of a rounded rectangle. Using the same value for the arc width and arc height produces a quarter-circle at each corner. When the arc width, arc height, width and height have the same values, the result is a circle. If the values for width and height are the same and the values of `arcWidth` and `arcHeight` are 0, the result is a square.

Methods `draw3DRect` (Fig. 27.18, line 28) and `fill3DRect` (line 29) take the same arguments. The first two specify the *top-left* corner of the rectangle. The next two arguments specify the width and height of the rectangle, respectively. The last argument determines whether the rectangle is *raised* (`true`) or *lowered* (`false`). The three-dimensional effect of `draw3DRect` appears as two edges of the rectangle in the original color and two edges in a slightly darker color. The three-dimensional effect of `fill3DRect` appears as two edges of the rectangle in the original drawing color and the fill and other two edges in a

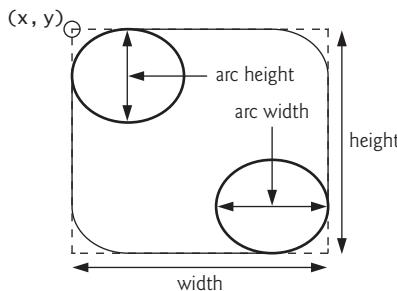


Fig. 27.20 | Arc width and arc height for rounded rectangles.

slightly darker color. Raised rectangles have the original drawing color edges at the top and left of the rectangle. Lowered rectangles have the original drawing color edges at the bottom and right of the rectangle. The three-dimensional effect is difficult to see in some colors.

Methods `drawOval` and `fillOval` (lines 32–33) take the same four arguments. The first two specify the top-left coordinate of the bounding rectangle that contains the oval. The last two specify the width and height of the bounding rectangle, respectively. Figure 27.21 shows an oval bounded by a rectangle. The oval touches the *center* of all four sides of the bounding rectangle. (The bounding rectangle is *not* displayed on the screen.)

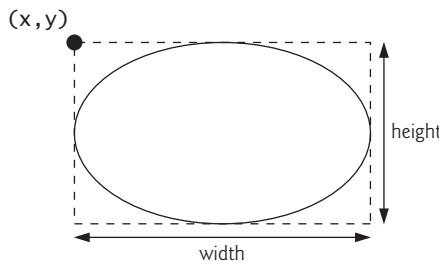


Fig. 27.21 | Oval bounded by a rectangle.

27.6 Drawing Arcs

An **arc** is drawn as a portion of an oval. Arc angles are measured in degrees. Arcs **sweep** (i.e., move along a curve) from a **starting angle** through the number of degrees specified by their **arc angle**. The starting angle indicates in degrees where the arc begins. The arc angle specifies the total number of degrees through which the arc sweeps. Figure 27.22 illustrates two arcs. The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a *counterclockwise* direction are measured in **positive degrees**. The set of axes on the right shows an arc sweeping from zero degrees to approximately –110 degrees. Arcs that sweep in a *clockwise* direction are measured in **negative degrees**. Note the dashed boxes around the arcs in Fig. 27.22. When drawing an arc,

we specify a bounding rectangle for an oval. The arc will sweep along part of the oval. `Graphics` methods `drawArc` and `fillArc` for drawing arcs are summarized in Fig. 27.23.



Fig. 27.22 | Positive and negative arc angles.

Method	Description
<code>public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)</code>	Draws an arc relative to the bounding rectangle's top-left x- and y-coordinates with the specified <code>width</code> and <code>height</code> . The arc segment is drawn starting at <code>startAngle</code> and sweeps <code>arcAngle</code> degrees.
<code>public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)</code>	Draws a filled arc (i.e., a sector) relative to the bounding rectangle's top-left x- and y-coordinates with the specified <code>width</code> and <code>height</code> . The arc segment is drawn starting at <code>startAngle</code> and sweeps <code>arcAngle</code> degrees.

Fig. 27.23 | `Graphics` methods for drawing arcs.

Figures 27.24–27.25 demonstrate the arc methods of Fig. 27.23. The application draws six arcs (three unfilled and three filled). To illustrate the bounding rectangle that helps determine where the arc appears, the first three arcs are displayed inside a red rectangle that has the same `x`, `y`, `width` and `height` arguments as the arcs.

```

1 // Fig. 13.24: Arcs JPanel.java
2 // Arcs displayed with drawArc and fillArc.
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import javax.swing.JPanel;
6
7 public class ArcsJPanel extends JPanel
8 {

```

Fig. 27.24 | Arcs displayed with `drawArc` and `fillArc`. (Part I of 2.)

```

9   // draw rectangles and arcs
10  @Override
11  public void paintComponent(Graphics g)
12  {
13      super.paintComponent(g);
14
15      // start at 0 and sweep 360 degrees
16      g.setColor(Color.RED);
17      g.drawRect(15, 35, 80, 80);
18      g.setColor(Color.BLACK);
19      g.drawArc(15, 35, 80, 80, 0, 360);
20
21      // start at 0 and sweep 110 degrees
22      g.setColor(Color.RED);
23      g.drawRect(100, 35, 80, 80);
24      g.setColor(Color.BLACK);
25      g.drawArc(100, 35, 80, 80, 0, 110);
26
27      // start at 0 and sweep -270 degrees
28      g.setColor(Color.RED);
29      g.drawRect(185, 35, 80, 80);
30      g.setColor(Color.BLACK);
31      g.drawArc(185, 35, 80, 80, 0, -270);
32
33      // start at 0 and sweep 360 degrees
34      g.fillArc(15, 120, 80, 40, 0, 360);
35
36      // start at 270 and sweep -90 degrees
37      g.fillArc(100, 120, 80, 40, 270, -90);
38
39      // start at 0 and sweep -270 degrees
40      g.fillArc(185, 120, 80, 40, 0, -270);
41  }
42 } // end class ArcsJPanel

```

Fig. 27.24 | Arcs displayed with `drawArc` and `fillArc`. (Part 2 of 2.)

```

1 // Fig. 13.25: DrawArcs.java
2 // Drawing arcs.
3 import javax.swing.JFrame;
4
5 public class DrawArcs
6 {
7     // execute application
8     public static void main(String[] args)
9     {
10         // create frame for ArcsJPanel
11         JFrame frame = new JFrame("Drawing Arcs");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13

```

Fig. 27.25 | Drawing arcs. (Part 1 of 2.)

```

14     ArcsJPanel arcsJPanel = new ArcsJPanel();
15     frame.add(arcsJPanel);
16     frame.setSize(300, 210);
17     frame.setVisible(true);
18 }
19 } // end class DrawArcs

```

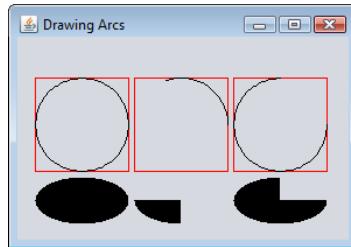


Fig. 27.25 | Drawing arcs. (Part 2 of 2.)

27.7 Drawing Polygons and Polylines

Polygons are *closed multisided shapes* composed of straight-line segments. **Polylines** are *sequences of connected points*. Figure 27.26 discusses methods for drawing polygons and polylines. Some methods require a **Polygon** object (package `java.awt`). Class **Polygon**'s constructors are also described in Fig. 27.26. The application of Figs. 27.27–27.28 draws polygons and polylines.

Method	Description
<i>Graphics methods for drawing polygons</i>	
<code>public void drawPolygon(int[] xPoints, int[] yPoints, int points)</code>	Draws a polygon. The <i>x</i> -coordinate of each point is specified in the <i>xPoints</i> array and the <i>y</i> -coordinate of each point in the <i>yPoints</i> array. The last argument specifies the number of points. This method draws a <i>closed polygon</i> . If the last point is different from the first, the polygon is <i>closed</i> by a line that connects the last point to the first.
<code>public void drawPolyline(int[] xPoints, int[] yPoints, int points)</code>	Draws a sequence of connected lines. The <i>x</i> -coordinate of each point is specified in the <i>xPoints</i> array and the <i>y</i> -coordinate of each point in the <i>yPoints</i> array. The last argument specifies the number of points. If the last point is different from the first, the polyline is <i>not closed</i> .
<code>public void drawPolygon(Polygon p)</code>	Draws the specified polygon.

Fig. 27.26 | Graphics methods for polygons and class Polygon methods. (Part I of 2.)

Method	Description
<code>public void fillPolygon(int[] xPoints, int[] yPoints, int points)</code>	Draws a <i>filled</i> polygon. The <i>x</i> -coordinate of each point is specified in the <code>xPoints</code> array and the <i>y</i> -coordinate of each point in the <code>yPoints</code> array. The last argument specifies the number of <code>points</code> . This method draws a <i>closed</i> polygon. If the last point is different from the first, the polygon is <i>closed</i> by a line that connects the last point to the first.
<code>public void fillPolygon(Polygon p)</code>	Draws the specified <i>filled</i> polygon. The polygon is <i>closed</i> .
<i>Polygon constructors and methods</i>	
<code>public Polygon()</code>	Constructs a new polygon object. The polygon does not contain any points.
<code>public Polygon(int[] xValues, int[] yValues, int numberOfPoints)</code>	Constructs a new polygon object. The polygon has <code>numberOfPoints</code> sides, with each point consisting of an <i>x</i> -coordinate from <code>xValues</code> and a <i>y</i> -coordinate from <code>yValues</code> .
<code>public void addPoint(int x, int y)</code>	Adds pairs of <i>x</i> - and <i>y</i> -coordinates to the <code>Polygon</code> .

Fig. 27.26 | Graphics methods for polygons and class `Polygon` methods. (Part 2 of 2.)

```

1 // Fig. 13.27: PolygonsJPanel.java
2 // Drawing polygons.
3 import java.awt.Graphics;
4 import java.awt.Polygon;
5 import javax.swing.JPanel;
6
7 public class PolygonsJPanel extends JPanel
8 {
9     // draw polygons and polylines
10    @Override
11    public void paintComponent(Graphics g)
12    {
13        super.paintComponent(g);
14
15        // draw polygon with Polygon object
16        int[] xValues = {20, 40, 50, 30, 20, 15};
17        int[] yValues = {50, 50, 60, 80, 80, 60};
18        Polygon polygon1 = new Polygon(xValues, yValues, 6);
19        g.drawPolygon(polygon1);
20
21        // draw polylines with two arrays
22        int[] xValues2 = {70, 90, 100, 80, 70, 65, 60};
23        int[] yValues2 = {100, 100, 110, 110, 130, 110, 90};
24        g.drawPolyline(xValues2, yValues2, 7);

```

Fig. 27.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 1 of 2.)

```
25
26     // fill polygon with two arrays
27     int[] xValues3 = {120, 140, 150, 190};
28     int[] yValues3 = {40, 70, 80, 60};
29     g.fillPolygon(xValues3, yValues3, 4);
30
31     // draw filled polygon with Polygon object
32     Polygon polygon2 = new Polygon();
33     polygon2.addPoint(165, 135);
34     polygon2.addPoint(175, 150);
35     polygon2.addPoint(270, 200);
36     polygon2.addPoint(200, 220);
37     polygon2.addPoint(130, 180);
38     g.fillPolygon(polygon2);
39 }
40 } // end class PolygonsJPanel
```

Fig. 27.27 | Polygons displayed with `drawPolygon` and `fillPolygon`. (Part 2 of 2.)

Lines 16–17 of Fig. 27.27 create two `int` arrays and use them to specify the points for `Polygon` `polygon1`. The `Polygon` constructor call in line 18 receives array `xValues`, which contains the *x*-coordinate of each point; array `yValues`, which contains the *y*-coordinate of each point; and 6 (the number of points in the polygon). Line 19 displays `polygon1` by passing it as an argument to `Graphics` method `drawPolygon`.

Lines 22–23 create two `int` arrays and use them to specify the points for a series of connected lines. Array `xValues2` contains the *x*-coordinate of each point and array `yValues2` the *y*-coordinate of each point. Line 24 uses `Graphics` method `drawPolyline` to display the series of connected lines specified with the arguments `xValues2`, `yValues2` and 7 (the number of points).

Lines 27–28 create two `int` arrays and use them to specify the points of a polygon. Array `xValues3` contains the *x*-coordinate of each point and array `yValues3` the *y*-coordinate of each point. Line 29 displays a polygon by passing to `Graphics` method `fillPolygon` the two arrays (`xValues3` and `yValues3`) and the number of points to draw (4).



Common Programming Error 27.1

An `ArrayIndexOutOfBoundsException` is thrown if the number of points specified in the third argument to method `drawPolygon` or method `fillPolygon` is greater than the number of elements in the arrays of coordinates that specify the polygon to display.

Line 32 creates `Polygon` `polygon2` with no points. Lines 33–37 use `Polygon` method `addPoint` to add pairs of *x*- and *y*-coordinates to the `Polygon`. Line 38 displays `Polygon` `polygon2` by passing it to `Graphics` method `fillPolygon`.

```
1 // Fig. 13.28: DrawPolygons.java
2 // Drawing polygons.
3 import javax.swing.JFrame;
4
```

Fig. 27.28 | Drawing polygons. (Part 1 of 2.)

```

5  public class DrawPolygons
6  {
7      // execute application
8      public static void main(String[] args)
9      {
10          // create frame for PolygonsJPanel
11          JFrame frame = new JFrame("Drawing Polygons");
12          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14          PolygonsJPanel polygonsJPanel = new PolygonsJPanel();
15          frame.add(polygonsJPanel);
16          frame.setSize(280, 270);
17          frame.setVisible(true);
18      }
19  } // end class DrawPolygons

```

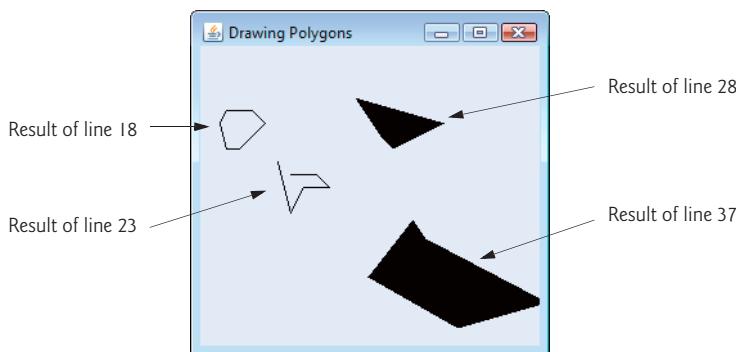


Fig. 27.28 | Drawing polygons. (Part 2 of 2.)

27.8 Java 2D API

The **Java 2D API** provides advanced two-dimensional graphics capabilities for programmers who require detailed and complex graphical manipulations. The API includes features for processing line art, text and images in packages `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` and `java.awt.image.renderable`. The capabilities of the API are far too broad to cover in this textbook. For an overview, visit <http://docs.oracle.com/javase/8/docs/technotes/guides/2d/>. In this section, we overview several Java 2D capabilities.

Drawing with the Java 2D API is accomplished with a **Graphics2D** reference (package `java.awt`). **Graphics2D** is an *abstract subclass* of class **Graphics**, so it has all the graphics capabilities demonstrated earlier in this chapter. In fact, the actual object used to draw in every `paintComponent` method is an instance of a *subclass* of **Graphics2D** that is passed to method `paintComponent` and accessed via the *superclass* **Graphics**. To access **Graphics2D** capabilities, we must cast the **Graphics** reference (`g`) passed to `paintComponent` into a **Graphics2D** reference with a statement such as

```
Graphics2D g2d = (Graphics2D) g;
```

The next two examples use this technique.

Lines, Rectangles, Round Rectangles, Arcs and Ellipses

This example demonstrates several Java 2D shapes from package `java.awt.geom`, including `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double`, `Arc2D.Double` and `Ellipse2D.Double`. Note the syntax of each class name. Each class represents a shape with dimensions specified as `double` values. There's a *separate* version of each represented with `float` values (e.g., `Ellipse2D.Float`). In each case, `Double` is a `public static` nested class of the class specified to the left of the dot (e.g., `Ellipse2D`). To use the `static` nested class, we simply qualify its name with the outer-class name.

In Figs. 27.29–27.30, we draw Java 2D shapes and modify their drawing characteristics, such as changing line thickness, filling shapes with patterns and drawing dashed lines. These are just a few of the many capabilities provided by Java 2D. Line 25 of Fig. 27.29 casts the `Graphics` reference received by `paintComponent` to a `Graphics2D` reference and assigns it to `g2d` to allow access to the Java 2D features.

```

1 // Fig. 13.29: Shapes JPanel.java
2 // Demonstrating some Java 2D shapes.
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import java.awt.BasicStroke;
6 import java.awt.GradientPaint;
7 import java.awt.TexturePaint;
8 import java.awt.Rectangle;
9 import java.awt.Graphics2D;
10 import java.awt.geom.Ellipse2D;
11 import java.awt.geom.Rectangle2D;
12 import java.awt.geom.RoundRectangle2D;
13 import java.awt.geom.Arc2D;
14 import java.awt.geom.Line2D;
15 import java.awt.image.BufferedImage;
16 import javax.swing.JPanel;
17
18 public class Shapes JPanel extends JPanel
19 {
20     // draw shapes with Java 2D API
21     @Override
22     public void paintComponent(Graphics g)
23     {
24         super.paintComponent(g);
25         Graphics2D g2d = (Graphics2D) g; // cast g to Graphics2D
26
27         // draw 2D ellipse filled with a blue-yellow gradient
28         g2d.setPaint(new GradientPaint(5, 30, Color.BLUE, 35, 100,
29             Color.YELLOW, true));
30         g2d.fill(new Ellipse2D.Double(5, 30, 65, 100));
31
32         // draw 2D rectangle in red
33         g2d.setPaint(Color.RED);
34         g2d.setStroke(new BasicStroke(10.0f));
35         g2d.draw(new Rectangle2D.Double(80, 30, 65, 100));
36

```

Fig. 27.29 | Demonstrating some Java 2D shapes. (Part I of 2.)

```

37     // draw 2D rounded rectangle with a buffered background
38     BufferedImage buffImage = new BufferedImage(10, 10,
39             BufferedImage.TYPE_INT_RGB);
40
41     // obtain Graphics2D from buffImage and draw on it
42     Graphics2D gg = buffImage.createGraphics();
43     gg.setColor(Color.YELLOW);
44     gg.fillRect(0, 0, 10, 10);
45     gg.setColor(Color.BLACK);
46     gg.drawRect(1, 1, 6, 6);
47     gg.setColor(Color.BLUE);
48     gg.fillRect(1, 1, 3, 3);
49     gg.setColor(Color.RED);
50     gg.fillRect(4, 4, 3, 3); // draw a filled rectangle
51
52     // paint buffImage onto the JFrame
53     g2d.setPaint(new TexturePaint(buffImage,
54             new Rectangle(10, 10)));
55     g2d.fill(
56             new RoundRectangle2D.Double(155, 30, 75, 100, 50, 50));
57
58     // draw 2D pie-shaped arc in white
59     g2d.setPaint(Color.WHITE);
60     g2d.setStroke(new BasicStroke(6.0f));
61     g2d.draw(
62             new Arc2D.Double(240, 30, 75, 100, 0, 270, Arc2D.PIE));
63
64     // draw 2D lines in green and yellow
65     g2d.setPaint(Color.GREEN);
66     g2d.draw(new Line2D.Double(395, 30, 320, 150));
67
68     // draw 2D line using stroke
69     float[] dashes = {10}; // specify dash pattern
70     g2d.setPaint(Color.YELLOW);
71     g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
72             BasicStroke.JOIN_ROUND, 10, dashes, 0));
73     g2d.draw(new Line2D.Double(320, 30, 395, 150));
74 }
75 } // end class ShapesJPanel

```

Fig. 27.29 | Demonstrating some Java 2D shapes. (Part 2 of 2.)

```

1 // Fig. 13.30: Shapes.java
2 // Testing ShapesJPanel.
3 import javax.swing.JFrame;
4
5 public class Shapes
6 {
7     // execute application
8     public static void main(String[] args)
9     {

```

Fig. 27.30 | Testing ShapesJPanel. (Part 1 of 2.)

```
10     // create frame for ShapesJPanel
11     JFrame frame = new JFrame("Drawing 2D shapes");
12     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14     // create ShapesJPanel
15     ShapesJPanel shapesJPanel = new ShapesJPanel();
16
17     frame.add(shapesJPanel);
18     frame.setSize(425, 200);
19     frame.setVisible(true);
20 }
21 } // end class Shapes
```

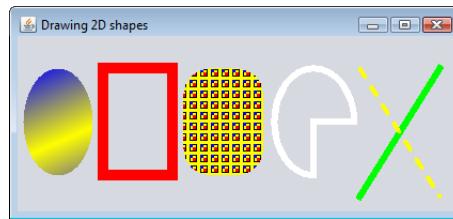


Fig. 27.30 | Testing ShapesJPanel. (Part 2 of 2.)

Ovals, Gradient Fills and Paint Objects

The first shape we draw is an *oval filled with gradually changing colors*. Lines 28–29 invoke `Graphics2D` method `setPaint` to set the `Paint` object that determines the color for the shape to display. A `Paint` object implements interface `java.awt.Paint`. It can be something as simple as one of the predeclared `Color` objects introduced in Section 27.3 (class `Color` implements `Paint`), or it can be an instance of the Java 2D API's `GradientPaint`, `SystemColor`, `TexturePaint`, `LinearGradientPaint` or `RadialGradientPaint` classes. In this case, we use a `GradientPaint` object.

Class `GradientPaint` helps draw a shape in *gradually changing colors*—called a **gradient**. The `GradientPaint` constructor used here requires seven arguments. The first two specify the starting coordinates for the gradient. The third specifies the starting `Color` for the gradient. The fourth and fifth specify the ending coordinates for the gradient. The sixth specifies the ending `Color` for the gradient. The last argument specifies whether the gradient is `cyclic` (`true`) or `acyclic` (`false`). The two sets of coordinates determine the direction of the gradient. Because the second coordinate (35, 100) is down and to the right of the first coordinate (5, 30), the gradient goes down and to the right at an angle. Because this gradient is cyclic (`true`), the color starts with blue, gradually becomes yellow, then gradually returns to blue. If the gradient is acyclic, the color transitions from the first color specified (e.g., blue) to the second color (e.g., yellow).

Line 30 uses `Graphics2D` method `fill` to draw a filled `Shape` object—an object that implements interface `Shape` (package `java.awt`). In this case, we display an `Ellipse2D.Double` object. The `Ellipse2D.Double` constructor receives four arguments specifying the *bounding rectangle* for the ellipse to display.

Rectangles, Strokes

Next we draw a red rectangle with a thick border. Line 33 invokes `setPaint` to set the `Paint` object to `Color.RED`. Line 34 uses `Graphics2D` method `setStroke` to set the characteristics of the rectangle's border (or the lines for any other shape). Method `setStroke` requires as its argument an object that implements interface `Stroke` (package `java.awt`). In this case, we use an instance of class `BasicStroke`. Class `BasicStroke` provides several constructors to specify the width of the line, how the line ends (called the `end caps`), how lines join together (called `line joins`) and the dash attributes of the line (if it's a dashed line). The constructor here specifies that the line should be 10 pixels wide.

Line 35 uses `Graphics2D` method `draw` to draw a `Shape` object—in this case, a `Rectangle2D.Double`. The `Rectangle2D.Double` constructor receives arguments specifying the rectangle's *upper-left x*-coordinate, upper-left *y*-coordinate, width and height.

Rounded Rectangles, `BufferedImage` and `TexturePaint` Objects

Next we draw a rounded rectangle filled with a pattern created in a `BufferedImage` (package `java.awt.image`) object. Lines 38–39 create the `BufferedImage` object. Class `BufferedImage` can be used to produce images in color and grayscale. This particular `BufferedImage` is 10 pixels wide and 10 pixels tall (as specified by the first two arguments of the constructor). The third argument `BufferedImage.TYPE_INT_RGB` indicates that the image is stored in color using the RGB color scheme.

To create the rounded rectangle's fill pattern, we must first draw into the `BufferedImage`. Line 42 creates a `Graphics2D` object (by calling `BufferedImage` method `createGraphics`) that can be used to draw into the `BufferedImage`. Lines 43–50 use methods `setColor`, `fillRect` and `drawRect` to create the pattern.

Lines 53–54 set the `Paint` object to a new `TexturePaint` (package `java.awt`) object. A `TexturePaint` object uses the image stored in its associated `BufferedImage` (the first constructor argument) as the fill texture for a filled-in shape. The second argument specifies the `Rectangle` area from the `BufferedImage` that will be replicated through the texture. In this case, the `Rectangle` is the same size as the `BufferedImage`. However, a smaller portion of the `BufferedImage` can be used.

Lines 55–56 use `Graphics2D` method `fill` to draw a filled `Shape` object—in this case, a `RoundRectangle2D.Double`. The constructor for class `RoundRectangle2D.Double` receives six arguments specifying the rectangle dimensions and the arc width and arc height used to determine the rounding of the corners.

Arcs

Next we draw a pie-shaped arc with a thick white line. Line 59 sets the `Paint` object to `Color.WHITE`. Line 60 sets the `Stroke` object to a new `BasicStroke` for a line 6 pixels wide. Lines 61–62 use `Graphics2D` method `draw` to draw a `Shape` object—in this case, an `Arc2D.Double`. The `Arc2D.Double` constructor's first four arguments specify the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height of the bounding rectangle for the arc. The fifth argument specifies the start angle. The sixth argument specifies the arc angle. The last argument specifies how the arc is *closed*. Constant `Arc2D.PIE` indicates that the arc is *closed* by drawing two lines—one line from the arc's starting point to the center of the bounding rectangle and one line from the center of the bounding rectangle to the ending point. Class `Arc2D` provides two other static constants for specifying how the arc is

closed. Constant **Arc2D.CHORD** draws a line from the starting point to the ending point. Constant **Arc2D.OPEN** specifies that the arc should *not* be *closed*.

Lines

Finally, we draw two lines using **Line2D** objects—one solid and one dashed. Line 65 sets the **Paint** object to **Color.GREEN**. Line 66 uses **Graphics2D** method **draw** to draw a **Shape** object—in this case, an instance of class **Line2D.Double**. The **Line2D.Double** constructor's arguments specify the starting coordinates and ending coordinates of the line.

Line 69 declares a one-element **float** array containing the value 10. This array describes the dashes in the dashed line. In this case, each dash will be 10 pixels long. To create dashes of different lengths in a pattern, simply provide the length of each dash as an element in the array. Line 70 sets the **Paint** object to **Color.YELLOW**. Lines 71–72 set the **Stroke** object to a new **BasicStroke**. The line will be 4 pixels wide and will have rounded ends (**BasicStroke.CAP_ROUND**). If lines join together (as in a rectangle at the corners), their joining will be rounded (**BasicStroke.JOIN_ROUND**). The **dashes** argument specifies the dash lengths for the line. The last argument indicates the starting index in the **dashes** array for the first dash in the pattern. Line 73 then draws a line with the current **Stroke**.

Creating Your Own Shapes with General Paths

Next we present a **general path**—a shape constructed from straight lines and complex curves. A general path is represented with an object of class **GeneralPath** (package **java.awt.geom**). The application of Figs. 27.31 and 27.32 demonstrates drawing a general path in the shape of a five-pointed star.

```

1 // Fig. 13.31: Shapes2JPanel.java
2 // Demonstrating a general path.
3 import java.awt.Color;
4 import java.awt.Graphics;
5 import java.awt.Graphics2D;
6 import java.awt.geom.GeneralPath;
7 import java.security.SecureRandom;
8 import javax.swing.JPanel;
9
10 public class Shapes2JPanel extends JPanel
11 {
12     // draw general paths
13     @Override
14     public void paintComponent(Graphics g)
15     {
16         super.paintComponent(g);
17         SecureRandom random = new SecureRandom();
18
19         int[] xPoints = {55, 67, 109, 73, 83, 55, 27, 37, 1, 43};
20         int[] yPoints = {0, 36, 36, 54, 96, 72, 96, 54, 36, 36};
21
22         Graphics2D g2d = (Graphics2D) g;
23         GeneralPath star = new GeneralPath();
24

```

Fig. 27.31 | Java 2D general paths. (Part I of 2.)

```

25     // set the initial coordinate of the General Path
26     star.moveTo(xPoints[0], yPoints[0]);
27
28     // create the star--this does not draw the star
29     for (int count = 1; count < xPoints.length; count++)
30         star.lineTo(xPoints[count], yPoints[count]);
31
32     star.closePath(); // close the shape
33
34     g2d.translate(150, 150); // translate the origin to (150, 150)
35
36     // rotate around origin and draw stars in random colors
37     for (int count = 1; count <= 20; count++)
38     {
39         g2d.rotate(Math.PI / 10.0); // rotate coordinate system
40
41         // set random drawing color
42         g2d.setColor(new Color(random.nextInt(256),
43             random.nextInt(256), random.nextInt(256)));
44
45         g2d.fill(star); // draw filled star
46     }
47 }
48 } // end class Shapes2JPanel

```

Fig. 27.31 | Java 2D general paths. (Part 2 of 2.)

```

1 // Fig. 13.32: Shapes2.java
2 // Demonstrating a general path.
3 import java.awt.Color;
4 import javax.swing.JFrame;
5
6 public class Shapes2
7 {
8     // execute application
9     public static void main(String[] args)
10    {
11        // create frame for Shapes2JPanel
12        JFrame frame = new JFrame("Drawing 2D Shapes");
13        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15        Shapes2JPanel shapes2JPanel = new Shapes2JPanel();
16        frame.add(shapes2JPanel);
17        frame.setBackground(Color.WHITE);
18        frame.setSize(315, 330);
19        frame.setVisible(true);
20    }
21 } // end class Shapes2

```

Fig. 27.32 | Demonstrating a general path. (Part 1 of 2.)

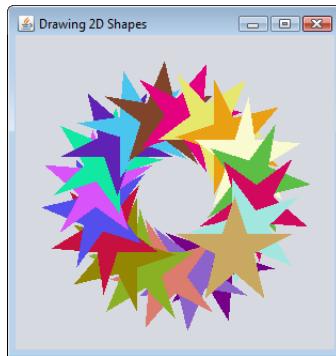


Fig. 27.32 | Demonstrating a general path. (Part 2 of 2.)

Lines 19–20 (Fig. 27.31) declare two `int` arrays representing the *x*- and *y*-coordinates of the points in the star. Line 23 creates `GeneralPath` object `star`. Line 26 uses `GeneralPath` method `moveTo` to specify the first point in the star. The `for` statement in lines 29–30 uses `GeneralPath` method `lineTo` to draw a line to the next point in the star. Each new call to `lineTo` draws a line from the previous point to the current point. Line 32 uses `GeneralPath` method `closePath` to draw a line from the last point to the point specified in the last call to `moveTo`. This completes the general path.

Line 34 uses `Graphics2D` method `translate` to move the drawing origin to location (150, 150). All drawing operations now use location (150, 150) as (0, 0).

The `for` statement in lines 37–46 draws the star 20 times by rotating it around the new origin point. Line 39 uses `Graphics2D` method `rotate` to rotate the next displayed shape. The argument specifies the rotation angle in radians (with $360^\circ = 2\pi$ radians). Line 45 uses `Graphics2D` method `fill` to draw a filled version of the star.

27.9 Wrap-Up

In this chapter, you learned how to use Java's graphics capabilities to produce colorful drawings. You learned how to specify the location of an object using Java's coordinate system, and how to draw on a window using the `paintComponent` method. You were introduced to class `Color`, and you learned how to use this class to specify different colors using their RGB components. You used the `JColorChooser` dialog to allow users to select colors in a program. You then learned how to work with fonts when drawing text on a window. You learned how to create a `Font` object from a font name, style and size, as well as how to access the metrics of a font. From there, you learned how to draw various shapes on a window, such as rectangles (regular, rounded and 3D), ovals and polygons, as well as lines and arcs. You then used the Java 2D API to create more complex shapes and to fill them with gradients or patterns. The chapter concluded with a discussion of general paths, used to construct shapes from straight lines and complex curves.

Summary

Section 27.1 Introduction

- Java's coordinate system (p. 2) is a scheme for identifying every point (p. 13) on the screen.
- A coordinate pair (p. 2) has an *x*-coordinate (horizontal) and a *y*-coordinate (vertical).
- Coordinates are used to indicate where graphics should be displayed on a screen.
- Coordinate units are measured in pixels (p. 2). A pixel is a display monitor's smallest unit of resolution.

Section 27.2 Graphics Contexts and Graphics Objects

- A Java graphics context (p. 4) enables drawing on the screen.
- Class `Graphics` (p. 4) contains methods for drawing strings, lines, rectangles and other shapes. Methods are also included for font manipulation and color manipulation.
- A `Graphics` object manages a graphics context and draws pixels on the screen that represent text and other graphical objects, e.g., lines, ellipses, rectangles and other polygons (p. 4).
- Class `Graphics` is an abstract class. Each Java implementation has a `Graphics` subclass that provides drawing capabilities. This implementation is hidden from us by class `Graphics`, which supplies the interface that enables us to use graphics in a platform-independent manner.
- Method `paintComponent` can be used to draw graphics in any `JComponent` component.
- Method `paintComponent` receives a `Graphics` object that is passed to the method by the system when a lightweight Swing component needs to be repainted.
- When an application executes, the application container calls method `paintComponent`. For `paintComponent` to be called again, an event must occur.
- When a `JComponent` is displayed, its `paintComponent` method is called.
- Calling method `repaint` (p. 5) on a component updates the graphics drawn on that component.

Section 27.3 Color Control

- Class `Color` (p. 5) declares methods and constants for manipulating colors in a Java program.
- Every color is created from a red, a green and a blue component. Together these components are called RGB values (p. 6). The RGB components specify the amount of red, green and blue in a color, respectively. The larger the value, the greater the amount of that particular color.
- `Color` methods `getRed`, `getGreen` and `getBlue` (p. 6) return `int` values from 0 to 255 representing the amount of red, green and blue, respectively.
- `Graphics` method `getColor` (p. 6) returns a `Color` object with the current drawing color.
- `Graphics` method `setColor` (p. 6) sets the current drawing color.
- `Graphics` method `fillRect` (p. 6) draws a rectangle filled by the `Graphics` object's current color.
- `Graphics` method `drawString` (p. 6) draws a `String` in the current color.
- The `JColorChooser` GUI component (p. 8) enables application users to select colors.
- `JColorChooser` static method `showDialog` (p. 10) displays a modal `JColorChooser` dialog.

Section 27.4 Manipulating Fonts

- Class `Font` (p. 12) contains methods and constants for manipulating fonts.
- Class `Font`'s constructor takes three arguments—the font name (p. 13), font style and font size.
- A `Font`'s font style can be `Font.PLAIN`, `Font.ITALIC` or `Font.BOLD` (each is a `static` field of class `Font`). Font styles can be used in combination (e.g., `Font.ITALIC + Font.BOLD`).

- The font size is measured in points. A point is 1/72 of an inch.
- `Graphics` method `setFont` (p. 13) sets the drawing font in which text will be displayed.
- `Font` method `getSize` (p. 13) returns the font size in points.
- `Font` method `getName` (p. 13) returns the current font name as a string.
- `Font` method `getStyle` (p. 15) returns an integer value representing the current `Font`'s style.
- `Font` method `getFamily` (p. 15) returns the name of the font family to which the current font belongs. The name of the font family is platform specific.
- Class `FontMetrics` (p. 2) contains methods for obtaining font information.
- Font metrics (p. 15) include height, descent and leading.

Section 27.5 Drawing Lines, Rectangles and Ovals

- `Graphics` methods `fillRoundRect` (p. 19) and `drawRoundRect` (p. 19) draw rectangles with rounded corners.
- `Graphics` methods `draw3DRect` (p. 20) and `fill3DRect` (p. 20) draw three-dimensional rectangles.
- `Graphics` methods `drawOval` (p. 21) and `fillOval` (p. 21) draw ovals.

Section 27.6 Drawing Arcs

- An arc (p. 21) is drawn as a portion of an oval.
- Arcs sweep from a starting angle by the number of degrees specified by their arc angle (p. 21).
- `Graphics` methods `drawArc` (p. 22) and `fillArc` (p. 22) are used for drawing arcs.

Section 27.7 Drawing Polygons and Polylines

- Class `Polygon` contains methods for creating polygons.
- Polygons are closed multisided shapes composed of straight-line segments.
- Polylines (p. 24) are sequences of connected points.
- `Graphics` method `drawPolyline` (p. 26) displays a series of connected lines.
- `Graphics` methods `drawPolygon` (p. 26) and `fillPolygon` (p. 26) are used to draw polygons.
- `Polygon` method `addPoint` (p. 26) adds pairs of *x*- and *y*-coordinates to the `Polygon`.

Section 27.8 Java 2D API

- The Java 2D API (p. 27) provides advanced two-dimensional graphics capabilities.
- Class `Graphics2D` (p. 27)—a subclass of `Graphics`—is used for drawing with the Java 2D API.
- The Java 2D API's classes for drawing shapes include `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double`, `Arc2D.Double` and `Ellipse2D.Double` (p. 28).
- Class `GradientPaint` (p. 30) helps draw a shape in gradually changing colors—called a gradient (p. 30).
- `Graphics2D` method `fill` (p. 30) draws a filled object of any type that implements interface `Shape` (p. 30).
- Class `BasicStroke` (p. 30) helps specify the drawing characteristics of lines.
- `Graphics2D` method `draw` (p. 31) is used to draw a `Shape` object.
- Classes `GradientPaint` (p. 31) and `TexturePaint` (p. 31) help specify the characteristics for filling shapes with colors or patterns.
- A general path (p. 32) is a shape constructed from straight lines and complex curves and is represented with an object of class `GeneralPath` (p. 32).

- GeneralPath method `moveTo` (p. 34) specifies the first point in a general path.
- GeneralPath method `lineTo` (p. 34) draws a line to the next point in the path. Each new call to `lineTo` draws a line from the previous point to the current point.
- GeneralPath method `closePath` (p. 34) draws a line from the last point to the point specified in the last call to `moveTo`. This completes the general path.
- Graphics2D method `translate` (p. 34) is used to move the drawing origin to a new location.
- Graphics2D method `rotate` (p. 34) is used to rotate the next displayed shape.

Self-Review Exercises

27.1 Fill in the blanks in each of the following statements:

- In Java 2D, method _____ of class _____ sets the characteristics of a stroke used to draw a shape.
- Class _____ helps specify the fill for a shape such that the fill gradually changes from one color to another.
- The _____ method of class `Graphics` draws a line between two points.
- RGB is short for _____, _____ and _____.
- Font sizes are measured in units called _____.
- Class _____ helps specify the fill for a shape using a pattern drawn in a `BufferedImage`.

27.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- The first two arguments of `Graphics` method `drawOval` specify the center coordinate of the oval.
- In the Java coordinate system, *x*-coordinates increase from left to right and *y*-coordinates from top to bottom.
- `Graphics` method `fillPolygon` draws a filled polygon in the current color.
- `Graphics` method `drawArc` allows negative angles.
- `Graphics` method `getSize` returns the size of the current font in centimeters.
- Pixel coordinate (0, 0) is located at the exact center of the monitor.

27.3 Find the error(s) in each of the following and explain how to correct them. Assume that `g` is a `Graphics` object.

- `g.setFont("SansSerif");`
- `g.erase(x, y, w, h); // clear rectangle at (x, y)`
- `Font f = new Font("Serif", Font.BOLDITALIC, 12);`
- `g.setColor(255, 255, 0); // change color to yellow`

Answers to Self-Review Exercises

27.1 a) `setStroke`, `Graphics2D`. b) `GradientPaint`. c) `drawLine`. d) red, green, blue. e) points. f) `TexturePaint`.

27.2 Answers for a) through f):

- False. The first two arguments specify the upper-left corner of the bounding rectangle.
- True.
- True.
- True.
- False. Font sizes are measured in points.
- False. The coordinate (0,0) corresponds to the upper-left corner of a GUI component on which drawing occurs.

27.3 Answers for a) through d):

- The `setFont` method takes a `Font` object as an argument—not a `String`.
- The `Graphics` class does not have an `erase` method. The `clearRect` method should be used.
- `Font.BOLDITALIC` is not a valid font style. To get a bold italic font, use `Font.BOLD + Font.ITALIC`.
- Method `setColor` takes a `Color` object as an argument, not three integers.

Exercises

27.4 Fill in the blanks in each of the following statements:

- Class _____ of the Java 2D API is used to draw ovals.
- Methods `draw` and `fill` of class `Graphics2D` require an object of type _____ as their argument.
- The three constants that specify font style are _____, _____ and _____.
- `Graphics2D` method _____ sets the painting color for Java 2D shapes.

27.5 State whether each of the following is *true* or *false*. If *false*, explain why.

- `Graphics` method `drawPolygon` automatically connects the endpoints of the polygon.
- `Graphics` method `drawLine` draws a line between two points.
- `Graphics` method `fillArc` uses degrees to specify the angle.
- In the Java coordinate system, values on the *y*-axis increase from left to right.
- `Graphics` inherits directly from class `Object`.
- `Graphics` is an *abstract* class.
- The `Font` class inherits directly from class `Graphics`.

27.6 (*Concentric Circles Using Method `drawArc`*) Write an application that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use `Graphics` method `drawArc`.

27.7 (*Concentric Circles Using Class `Ellipse2D.Double`*) Modify your solution to Exercise 27.6 to draw the ovals by using class `Ellipse2D.Double` and method `draw` of class `Graphics2D`.

27.8 (*Random Lines Using Class `Line2D.Double`*) Modify your solution to Exercise 27.7 to draw random lines in random colors and random thicknesses. Use class `Line2D.Double` and method `draw` of class `Graphics2D` to draw the lines.

27.9 (*Random Triangles*) Write an application that displays randomly generated triangles in different colors. Each triangle should be filled with a different color. Use class `GeneralPath` and method `fill` of class `Graphics2D` to draw the triangles.

27.10 (*Random Characters*) Write an application that randomly draws characters in different fonts, sizes and colors.

27.11 (*Grid Using Method `drawLine`*) Write an application that draws an 8-by-8 grid. Use `Graphics` method `drawLine`.

27.12 (*Grid Using Class `Line2D.Double`*) Modify your solution to Exercise 27.11 to draw the grid using instances of class `Line2D.Double` and method `draw` of class `Graphics2D`.

27.13 (*Grid Using Method `drawRect`*) Write an application that draws a 10-by-10 grid. Use the `Graphics` method `drawRect`.

27.14 (*Grid Using Class `Rectangle2D.Double`*) Modify your solution to Exercise 27.13 to draw the grid by using class `Rectangle2D.Double` and method `draw` of class `Graphics2D`.

27.15 (*Drawing Tetrahedrons*) Write an application that draws a tetrahedron (a three-dimensional shape with four triangular faces). Use class `GeneralPath` and method `draw` of class `Graphics2D`.

27.16 (Drawing Cubes) Write an application that draws a cube. Use class `GeneralPath` and method `draw` of class `Graphics2D`.

27.17 (Circles Using Class `Ellipse2D.Double`) Write an application that asks the user to input the radius of a circle as a floating-point number and draws the circle, as well as the values of the circle's diameter, circumference and area. Use the value 3.14159 for π . [Note: You may also use the predefined constant `Math.PI` for the value of π . This constant is more precise than the value 3.14159. Class `Math` is declared in the `java.lang` package, so you need not `import` it.] Use the following formulas (r is the radius):

$$\begin{aligned} \text{diameter} &= 2r \\ \text{circumference} &= 2\pi r \\ \text{area} &= \pi r^2 \end{aligned}$$

The user should also be prompted for a set of coordinates in addition to the radius. Then draw the circle and display its diameter, circumference and area, using an `Ellipse2D.Double` object to represent the circle and method `draw` of class `Graphics2D` to display it.

27.18 (Screen Saver) Write an application that simulates a screen saver. The application should randomly draw lines using method `drawLine` of class `Graphics`. After drawing 100 lines, the application should clear itself and start drawing lines again. To allow the program to draw continuously, place a call to `repaint` as the last line in method `paintComponent`. Do you notice any problems with this on your system?

27.19 (Screen Saver Using Timer) Package `javax.swing` contains a class called `Timer` that is capable of calling method `actionPerformed` of interface `ActionListener` at a fixed time interval (specified in milliseconds). Modify your solution to Exercise 27.18 to remove the call to `repaint` from method `paintComponent`. Declare your class to implement `ActionListener`. (The `actionPerformed` method should simply call `repaint`.) Declare an instance variable of type `Timer` called `timer` in your class. In the constructor for your class, write the following statements:

```
timer = new Timer(1000, this);
timer.start();
```

This creates an instance of class `Timer` that will call `this` object's `actionPerformed` method every 1000 milliseconds (i.e., every second).

27.20 (Screen Saver for a Random Number of Lines) Modify your solution to Exercise 27.19 to enable the user to enter the number of random lines that should be drawn before the application clears itself and starts drawing lines again. Use a `JTextField` to obtain the value. The user should be able to type a new number into the `JTextField` at any time during the program's execution. Use an inner class to perform event handling for the `JTextField`.

27.21 (Screen Saver with Shapes) Modify your solution to Exercise 27.20 such that it uses random-number generation to choose different shapes to display. Use methods of class `Graphics`.

27.22 (Screen Saver Using the Java 2D API) Modify your solution to Exercise 27.21 to use classes and drawing capabilities of the Java 2D API. Draw shapes like rectangles and ellipses, with randomly generated gradients. Use class `GradientPaint` to generate the gradient.

27.23 (Turtle Graphics) Modify your solution to Exercise 7.21—*Turtle Graphics*—to add a graphical user interface using `JTextFields` and `JButtons`. Draw lines rather than asterisks (*). When the turtle graphics program specifies a move, translate the number of positions into a number of pixels on the screen by multiplying the number of positions by 10 (or any value you choose). Implement the drawing with Java 2D API features.

27.24 (Knight's Tour) Produce a graphical version of the Knight's Tour problem (Exercise 7.22, Exercise 7.23 and Exercise 7.26). As each move is made, the appropriate cell of the chessboard

should be updated with the proper move number. If the result of the program is a *full tour* or a *closed tour*, the program should display an appropriate message. If you like, use class `Timer` (see Exercise 27.19) to help animate the Knight's Tour.

27.25 (Tortoise and Hare) Produce a graphical version of the *Tortoise and Hare* simulation (Exercise 7.28). Simulate the mountain by drawing an arc that extends from the bottom-left corner of the window to the top-right corner. The tortoise and the hare should race up the mountain. Implement the graphical output to actually print the tortoise and the hare on the arc for every move. [Hint: Extend the length of the race from 70 to 300 to allow yourself a larger graphics area.]

27.26 (Drawing Spirals) Write an application that uses `Graphics` method `drawPolyline` to draw a spiral similar to the one shown in Fig. 27.33.

27.27 (Pie Chart) Write a program that inputs four numbers and graphs them as a pie chart. Use class `Arc2D.Double` and method `fill` of class `Graphics2D` to perform the drawing. Draw each piece of the pie in a separate color.

27.28 (Selecting Shapes) Write an application that allows the user to select a shape from a `JComboBox` and draws it 20 times with random locations and dimensions in method `paintComponent`. The first item in the `JComboBox` should be the default shape that is displayed the first time `paintComponent` is called.

27.29 (Random Colors) Modify Exercise 27.28 to draw each of the 20 randomly sized shapes in a randomly selected color. Use all 13 predefined `Color` objects in an array of `Colors`.

27.30 (JColorChooser Dialog) Modify Exercise 27.28 to allow the user to select the color in which shapes should be drawn from a `JColorChooser` dialog.

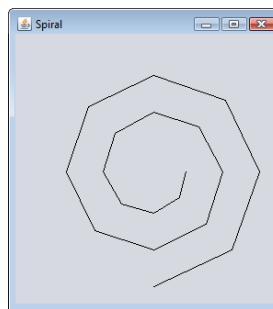


Fig. 27.33 | Spiral drawn using method `drawPolyline`.

(Optional) GUI and Graphics Case Study Exercise: Adding Java 2D

27.31 Java 2D introduces many new capabilities for creating unique and impressive graphics. We'll add a small subset of these features to the drawing application you created in Exercise 26.17. In this version, you'll enable the user to specify gradients for filling shapes and to change stroke characteristics for drawing lines and outlines of shapes. The user will be able to choose which colors compose the gradient and set the width and dash length of the stroke.

First, you must update the `MyShape` hierarchy to support Java 2D functionality. Make the following changes in class `MyShape`:

- Change abstract method `draw`'s parameter type from `Graphics` to `Graphics2D`.
- Change all variables of type `Color` to type `Paint` to enable support for gradients. [Note: Recall that class `Color` implements interface `Paint`.]

- c) Add an instance variable of type `Stroke` in class `MyShape` and a `Stroke` parameter in the constructor to initialize the new instance variable. The default stroke should be an instance of class `BasicStroke`.

Classes `MyLine`, `MyBoundedShape`, `MyOval` and `MyRectangle` should each add a `Stroke` parameter to their constructors. In the `draw` methods, each shape should set the `Paint` and the `Stroke` before drawing or filling a shape. Since `Graphics2D` is a subclass of `Graphics`, we can continue to use `Graphics` methods `drawLine`, `drawOval`, `fillOval`, and so on to draw the shapes. When these methods are called, they'll draw the appropriate shape using the specified `Paint` and `Stroke` settings.

Next, you'll update the `DrawPanel` to handle the Java 2D features. Change all `Color` variables to `Paint` variables. Declare an instance variable `currentStroke` of type `Stroke` and provide a `set` method for it. Update the calls to the individual shape constructors to include the `Paint` and `Stroke` arguments. In method `paintComponent`, cast the `Graphics` reference to type `Graphics2D` and use the `Graphics2D` reference in each call to `MyShape` method `draw`.

Next, make the Java 2D features accessible from the GUI. Create a `JPanel` of GUI components for setting the Java 2D options. Add these components at the top of the `DrawFrame` below the panel that currently contains the standard shape controls (see Fig. 27.34). These GUI components should include:

- a) A checkbox to specify whether to paint using a gradient.
- b) Two `JButtons` that each show a `JColorChooser` dialog to allow the user to choose the first and second color in the gradient. (These will replace the `JComboBox` used for choosing the color in Exercise 26.17.)
- c) A text field for entering the `Stroke` width.
- d) A text field for entering the `Stroke` dash length.
- e) A checkbox for selecting whether to draw a dashed or solid line.

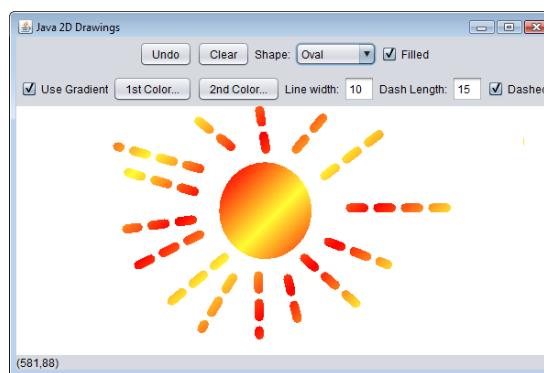


Fig. 27.34 | Drawing with Java 2D.

If the user selects to draw with a gradient, set the `Paint` on the `DrawPanel` to be a gradient of the two colors chosen by the user. The expression

```
new GradientPaint(0, 0, color1, 50, 50, color2, true))
```

creates a `GradientPaint` that cycles diagonally from the upper-left to the bottom-right every 50 pixels. Variables `color1` and `color2` represent the colors chosen by the user. If the user does not select to use a gradient, then simply set the `Paint` on the `DrawPanel` to be the first `Color` chosen by the user.

For strokes, if the user chooses a solid line, then create the `Stroke` with the expression

```
new BasicStroke(width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)
```

where variable `width` is the width specified by the user in the line-width text field. If the user chooses a dashed line, then create the `Stroke` with the expression

```
new BasicStroke(width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND,
    10, dashes, 0)
```

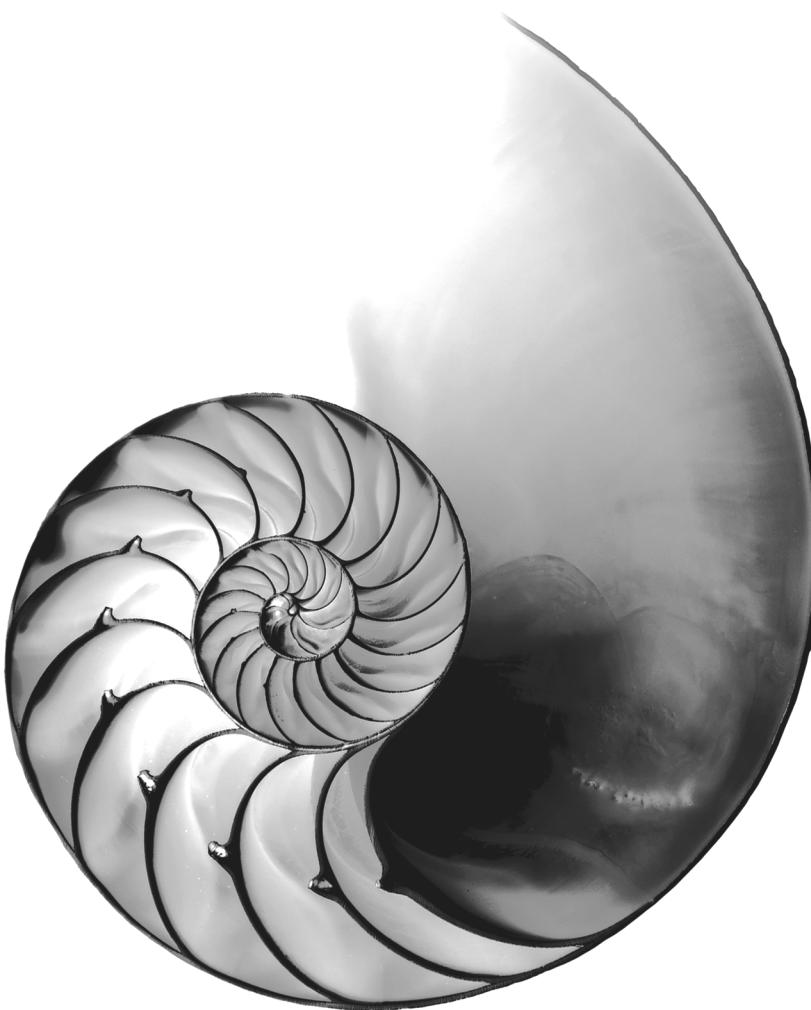
where `width` again is the width in the line-width field, and `dashes` is an array with one element whose value is the length specified in the dash-length field. The `Panel` and `Stroke` objects should be passed to the shape object's constructor when the shape is created in `DrawPanel`.

Making a Difference

27.32 (*Large-Type Displays for People with Low Vision*) The accessibility of computers and the Internet to all people, regardless of disabilities, is becoming more important as these tools play increasing roles in our personal and business lives. According to a recent estimate by the World Health Organization (<http://www.who.int/mediacentre/factsheets/fs282/en/>), 246 million people worldwide have low vision. To learn more about low vision, check out the GUI-based low-vision simulation at <http://webaim.org/simulations/lowlvision>. People with low vision might prefer to choose a font and/or a larger font size when reading electronic documents and web pages. Java has five built-in “logical” fonts that are guaranteed to be available in any Java implementation, including `Serif`, `Sans-serif` and `Monospaced`. Write a GUI application that provides a `JTextArea` in which the user can type text. Allow the user to select `Serif`, `Sans-serif` or `Monospaced` from a `JComboBox`. Provide a `Bold` `JCheckBox`, which, if checked, makes the text bold. Include `Increase Font Size` and `Decrease Font Size` `JButtons` that allow the user to scale the size of the font up or down, respectively, by one point at a time. Start with a font size of 18 points. For the purposes of this exercise, set the font size on the `JComboBox`, `JButtons` and `JCheckBox` to 20 points so that a person with low vision will be able to read the text on them.

28

Networking



Objectives

In this chapter you'll:

- Implement Java networking applications by using sockets and datagrams.
- Implement Java clients and servers that communicate with one another.
- Implement network-based collaborative applications.

Outline

-
- | | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 28.1 Introduction
28.2 Reading a File on a Web Server
28.3 Establishing a Simple Server Using Stream Sockets
28.4 Establishing a Simple Client Using Stream Sockets
28.5 Client/Server Interaction with Stream Socket Connections | 28.6 Datagrams: Connectionless Client/Server Interaction
28.7 Client/Server Tic-Tac-Toe Using a Multithreaded Server
28.8 Optional Online Case Study: <i>DeitelMessenger</i>
28.9 Wrap-Up |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
-

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

28.1 Introduction¹

Java provides a number of built-in networking capabilities that make it easy to develop Internet-based and web-based applications. Java can enable programs to search the world for information and to collaborate with programs running on other computers internationally, nationally or just within an organization (subject to security constraints).

Java's fundamental networking capabilities are declared by the classes and interfaces of package **java.net**, through which Java offers **stream-based communications** that enable applications to view networking as streams of data. The classes and interfaces of package **java.net** also offer **packet-based communications** for transmitting individual **packets** of information—commonly used to transmit data images, audio and video over the Internet. In this chapter, we show how to communicate with packets and streams of data.

We focus on both sides of the **client/server relationship**. The **client** *requests* that some action be performed, and the **server** performs the action and *responds* to the client. A common implementation of the *request-response model* is between web browsers and web servers. When a user selects a website to browse through a browser (the client application), a request is sent to the appropriate web server (the server application). The server normally responds to the client by sending an appropriate web page to be rendered by the browser.

We introduce Java's **socket-based communications**, which enable applications to view networking as if it were *file I/O*—a program can read from a **socket** or write to a socket as simply as reading from a file or writing to a file. The socket is simply a software construct that represents one endpoint of a connection. We show how to create and manipulate *stream sockets* and *datagram sockets*. With **stream sockets**, a process establishes a **connection** to another process. While the connection is in place, data flows between the processes in continuous **streams**. Stream sockets are said to provide a **connection-oriented service**. The protocol used for transmission is the popular **TCP (Transmission Control Protocol)**.

With **datagram sockets**, individual **packets** of information are transmitted. The protocol used—**UDP, the User Datagram Protocol**—is a **connectionless service** and does *not* guarantee that packets arrive in any particular *order*. With UDP, packets can even be *lost* or *duplicated*. Significant extra programming is required on your part to deal with these problems (if you choose to do so). UDP is most appropriate for network applications that do not require the error checking and reliability of TCP. Stream sockets and the TCP protocol will be more desirable for the vast majority of Java networking applications.

1. This is a legacy chapter posted as is from the book's 10th edition.



Performance Tip 28.1

Connectionless services generally offer greater performance but less reliability than connection-oriented services.



Portability Tip 28.1

TCP, UDP and related protocols enable heterogeneous computer systems (i.e., those with different processors and different operating systems) to intercommunicate.

For interested readers, we provide at

<http://www.deitel.com/books/jhttp11>

a case study from an older edition of this book. In the case study, we implement a client/server chat application using **multicasting**, in which a server can publish information and *many* clients can *subscribe* to it. When the server publishes information, *all* subscribers receive it.

28.2 Reading a File on a Web Server

The application in Fig. 28.1 uses Swing GUI component **JEditorPane** (from package `javax.swing`) to display the contents of a file on a web server. The user enters a URL in the **JTextField** at the top of the window, and the application displays the corresponding document (if it exists) in the **JEditorPane**. Class **JEditorPane** is able to render both plain text and basic HTML-formatted text, as illustrated in the two screen captures (Fig. 28.2), so this application acts as a simple web browser. The application also demonstrates how to process **HyperlinkEvents** when the user clicks a hyperlink in the HTML document.

```
1 // Fig. 28.1: ReadServerFile.java
2 // Reading a file by opening a connection through a URL.
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.io.IOException;
7 import javax.swing.JEditorPane;
8 import javax.swing.JFrame;
9 import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextField;
12 import javax.swing.event.HyperlinkEvent;
13 import javax.swing.event.HyperlinkListener;
14
15 public class ReadServerFile extends JFrame
16 {
17     private JTextField enterField; // JTextField to enter site name
18     private JEditorPane contentsArea; // to display website
19
20     // set up GUI
21     public ReadServerFile()
22     {
```

Fig. 28.1 | Reading a file by opening a connection through a URL. (Part 1 of 2.)

```
23     super("Simple Web Browser");
24
25     // create enterField and register its listener
26     enterField = new JTextField("Enter file URL here");
27     enterField.addActionListener(
28         new ActionListener()
29     {
30         // get document specified by user
31         public void actionPerformed(ActionEvent event)
32         {
33             getThePage(event.getActionCommand());
34         }
35     }
36 );
37
38     add(enterField, BorderLayout.NORTH);
39
40     contentsArea = new JEditorPane(); // create contentsArea
41     contentsArea.setEditable(false);
42     contentsArea.addHyperlinkListener(
43         new HyperlinkListener()
44     {
45         // if user clicked hyperlink, go to specified page
46         public void hyperlinkUpdate(HyperlinkEvent event)
47         {
48             if (event.getEventType() ==
49                 HyperlinkEvent.EventType.ACTIVATED)
50                 getThePage(event.getURL().toString());
51         }
52     }
53 );
54
55     add(new JScrollPane(contentsArea), BorderLayout.CENTER);
56     setSize(400, 300); // set size of window
57     setVisible(true); // show window
58 }
59
60 // Load document
61 private void getThePage(String location)
62 {
63     try // Load document and display location
64     {
65         contentsArea.setPage(location); // set the page
66         enterField.setText(location); // set the text
67     }
68     catch (IOException ioException)
69     {
70         JOptionPane.showMessageDialog(this,
71             "Error retrieving specified URL", "Bad URL",
72             JOptionPane.ERROR_MESSAGE);
73     }
74 }
75 }
```

Fig. 28.1 | Reading a file by opening a connection through a URL. (Part 2 of 2.)

```
1 // Fig. 28.2: ReadServerFileTest.java
2 // Create and start a ReadServerFile.
3 import javax.swing.JFrame;
4
5 public class ReadServerFileTest
6 {
7     public static void main(String[] args)
8     {
9         ReadServerFile application = new ReadServerFile();
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11    }
12 }
```



Fig. 28.2 | Test class for ReadServerFile.

The application class `ReadServerFile` contains `JTextField enterField`, in which the user enters the URL of the file to read and `JEditorPane contentsArea` to display the file's contents. When the user presses the *Enter* key in `enterField`, the application calls method `actionPerformed` (lines 31–34). Line 33 uses `ActionEvent` method `getActionCommand` to get the `String` the user input in the `JTextField` and passes the `String` to utility method `getThePage` (lines 61–74).

Line 65 invokes `JEditorPane` method `setPage` to download the document specified by `location` and display it in the `JEditorPane`. If there's an error downloading the document, method `setPage` throws an `IOException`. Also, if an invalid URL is specified, a `MalformedURLException` (a subclass of `IOException`) occurs. If the document loads successfully, line 66 displays the current location in `enterField`.

Typically, an HTML document contains **hyperlinks** that, when clicked, provide quick access to another document on the web. If a `JEditorPane` contains an HTML document and the user clicks a hyperlink, the `JEditorPane` generates a `HyperlinkEvent` (package `javax.swing.event`) and notifies all registered `HyperlinkListeners` (package `javax.swing.event`) of that event. Lines 42–53 register a `HyperlinkListener` to handle `HyperlinkEvents`. When a `HyperlinkEvent` occurs, the program calls method `hyperlinkUpdate` (lines 46–51). Lines 48–49 use `HyperlinkEvent` method `getEventType` to determine the type of the `HyperlinkEvent`. Class `HyperlinkEvent` contains a public nested class called `EventType` that declares three `static EventType` objects, which represent the hyperlink event types. **ACTIVATED** indicates that the user clicked a hyperlink to change web pages, **ENTERED** indicates that the user moved the mouse over a hyperlink and **EXITED** indicates that the user moved the mouse away from a hyperlink. If a hyperlink was **ACTIVATED**, line 50 uses `HyperlinkEvent` method `getURL` to obtain the URL represented by the hyperlink. Method `toString` converts the returned URL to a `String` that can be passed to utility method `getThePage`.



Look-and-Feel Observation 28.1

A JEditorPane generates HyperlinkEvents only if it's uneditable.

28.3 Establishing a Simple Server Using Stream Sockets

The two examples discussed so far use *high-level* Java networking capabilities to communicate between applications. In the examples, it was not your responsibility to establish the connection between a client and a server. The first program relied on the web browser to communicate with a web server. The second program relied on a JEditorPane to perform the connection. This section begins our discussion of creating your own applications that can communicate with one another.

Step 1: Create a ServerSocket

Establishing a simple server in Java requires five steps. *Step 1* is to create a **ServerSocket** object. A call to the ServerSocket constructor, such as

```
ServerSocket server = new ServerSocket(portNumber, queueLength);
```

registers an available TCP **port number** and specifies the maximum number of clients that can wait to connect to the server (i.e., the **queue length**). The port number is used by clients to locate the server application on the server computer. This is often called the **handshake point**. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients—a process known as **binding the server to the port**. Each client will ask to connect to the server on this **port**. Only one application at a time can be bound to a specific port on the server.



Software Engineering Observation 28.1

Port numbers can be between 0 and 65,535. Most operating systems reserve port numbers below 1024 for system services (e.g., e-mail and World Wide Web servers). Generally, these ports should not be specified as connection ports in user programs. In fact, some operating systems require special access privileges to bind to port numbers below 1024.

Step 2: Wait for a Connection

Programs manage each client connection with a **Socket** object. In *Step 2*, the server listens indefinitely (or **blocks**) for an attempt by a client to connect. To listen for a client connection, the program calls ServerSocket method **accept**, as in

```
Socket connection = server.accept();
```

which returns a **Socket** when a connection with a client is established. The **Socket** allows the server to interact with the client. The interactions with the client actually occur at a different server port from the *handshake point*. This allows the port specified in *Step 1* to be used again in a multithreaded server to accept another client connection. We demonstrate this concept in Section 28.7.

Step 3: Get the Socket's I/O Streams

Step 3 is to get the **OutputStream** and **InputStream** objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to

the client via an `OutputStream` and receives information from the client via an `InputStream`. The server invokes method `getOutputStream` on the `Socket` to get a reference to the `Socket`'s `OutputStream` and invokes method `getInputStream` on the `Socket` to get a reference to the `Socket`'s `InputStream`.

The stream objects can be used to send or receive individual bytes or sequences of bytes with the `OutputStream`'s method `write` and the `InputStream`'s method `read`, respectively. Often it's useful to send or receive values of primitive types (e.g., `int` and `double`) or `Serializable` objects (e.g., `Strings` or other serializable types) rather than sending bytes. In this case, we can use the techniques discussed in Chapter 15 to wrap other stream types (e.g., `ObjectOutputStream` and `ObjectInputStream`) around the `OutputStream` and `InputStream` associated with the `Socket`. For example,

```
ObjectInputStream input =
    new ObjectInputStream(connection.getInputStream());
ObjectOutputStream output =
    new ObjectOutputStream(connection.getOutputStream());
```

The beauty of establishing these relationships is that whatever the server writes to the `ObjectOutputStream` is sent via the `OutputStream` and is available at the client's `InputStream`, and whatever the client writes to its `OutputStream` (with a corresponding `ObjectOutputStream`) is available via the server's `InputStream`. The transmission of the data over the network is seamless and is handled completely by Java.

Step 4: Perform the Processing

Step 4 is the *processing* phase, in which the server and the client communicate via the `OutputStream` and `InputStream` objects.

Step 5: Close the Connection

In *Step 5*, when the transmission is complete, the server closes the connection by invoking the `close` method on the streams and on the `Socket`.



Software Engineering Observation 28.2

With sockets, network I/O appears to Java programs to be similar to sequential file I/O. Sockets hide much of the complexity of network programming.



Software Engineering Observation 28.3

A multithreaded server can take the `Socket` returned by each call to `accept` and create a new thread that manages network I/O across that `Socket`. Alternatively, a multithreaded server can maintain a pool of threads (a set of already existing threads) ready to manage network I/O across the new `Sockets` as they're created. These techniques enable multithreaded servers to manage many simultaneous client connections.



Performance Tip 28.2

In high-performance systems in which memory is abundant, a multithreaded server can create a pool of threads that can be assigned quickly to handle network I/O for new `Sockets` as they're created. Thus, when the server receives a connection, it need not incur thread-creation overhead. When the connection is closed, the thread is returned to the pool for reuse.

28.4 Establishing a Simple Client Using Stream Sockets

Establishing a simple client in Java requires four steps.

Step 1: Create a Socket to Connect to the Server

In *Step 1*, we create a `Socket` to connect to the server. The `Socket` constructor establishes the connection. For example, the statement

```
Socket connection = new Socket(serverAddress, port);
```

uses the `Socket` constructor with two arguments—the server's address (`serverAddress`) and the `port` number. If the connection attempt is successful, this statement returns a `Socket`. A connection attempt that fails throws an instance of a subclass of `IOException`, so many programs simply catch `IOException`. An `UnknownHostException` occurs specifically when the system is unable to resolve the server name specified in the call to the `Socket` constructor to a corresponding IP address.

Step 2: Get the Socket's I/O Streams

In *Step 2*, the client uses `Socket` methods `getInputStream` and `getOutputStream` to obtain references to the `Socket`'s `InputStream` and `OutputStream`. As we mentioned in the preceding section, we can use the techniques of Chapter 15 to wrap other stream types around the `InputStream` and `OutputStream` associated with the `Socket`. If the server is sending information in the form of actual types, the client should receive the information in the same format. Thus, if the server sends values with an `ObjectOutputStream`, the client should read those values with an `ObjectInputStream`.

Step 3: Perform the Processing

Step 3 is the processing phase in which the client and the server communicate via the `InputStream` and `OutputStream` objects.

Step 4: Close the Connection

In *Step 4*, the client closes the connection when the transmission is complete by invoking the `close` method on the streams and on the `Socket`. The client must determine when the server is finished sending information so that it can call `close` to close the `Socket` connection. For example, the `InputStream` method `read` returns the value `-1` when it detects end-of-stream (also called EOF—end-of-file). If an `ObjectInputStream` reads information from the server, an `EOFException` occurs when the client attempts to read a value from a stream on which end-of-stream is detected.

28.5 Client/Server Interaction with Stream Socket Connections

Figures 28.3 and 28.5 use stream sockets, `ObjectInputStream` and `ObjectOutputStream` to demonstrate a simple **client/server chat application**. The server waits for a client connection attempt. When a client connects to the server, the server application sends the client a `String` object (recall that `Strings` are `Serializable` objects) indicating that the connection was successful. Then the client displays the message. The client and server applications each provide text fields that allow the user to type a message and send it to the other application. When the client or the server sends the `String` "TERMINATE", the con-

nection terminates. Then the server waits for the next client to connect. The declaration of class `Server` appears in Fig. 28.3. The declaration of class `Client` appears in Fig. 28.5. The screen captures showing the execution between the client and the server are shown in Fig. 28.6.

Server Class

`Server`'s constructor (Fig. 28.3, lines 30–55) creates the server's GUI, which contains a `JTextField` and a `JTextArea`. `Server` displays its output in the `JTextArea`. When the main method (lines 6–11 of Fig. 28.4) executes, it creates a `Server` object, specifies the window's default close operation and calls method `runServer` (Fig. 28.3, lines 57–86).

```
1 // Fig. 28.3: Server.java
2 // Server portion of a client/server stream-socket connection.
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9 import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Server extends JFrame
19 {
20     private JTextField enterField; // inputs message from user
21     private JTextArea displayArea; // display information to user
22     private ObjectOutputStream output; // output stream to client
23     private ObjectInputStream input; // input stream from client
24     private ServerSocket server; // server socket
25     private Socket connection; // connection to client
26     private int counter = 1; // counter of number of connections
27
28     // set up GUI
29     public Server()
30     {
31         super("Server");
32
33         enterField = new JTextField(); // create enterField
34         enterField.setEditable(false);
35         enterField.addActionListener(
36             new ActionListener()
37             {
38                 // send message to client
39                 public void actionPerformed(ActionEvent event)
40                 {
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part I of 4.)

```
41             sendData(event.getActionCommand());
42             enterField.setText("");
43         }
44     }
45 );
46
47 add(enterField, BorderLayout.NORTH);
48
49 displayArea = new JTextArea(); // create displayArea
50 add(new JScrollPane(displayArea), BorderLayout.CENTER);
51
52 setSize(300, 150); // set size of window
53 setVisible(true); // show window
54 }
55
56 // set up and run server
57 public void runServer()
58 {
59     try // set up server to receive connections; process connections
60     {
61         server = new ServerSocket(12345, 100); // create ServerSocket
62
63         while (true)
64         {
65             try
66             {
67                 waitForConnection(); // wait for a connection
68                 getStreams(); // get input & output streams
69                 processConnection(); // process connection
70             }
71             catch (EOFException eofException)
72             {
73                 displayMessage("\nServer terminated connection");
74             }
75             finally
76             {
77                 closeConnection(); // close connection
78                 ++counter;
79             }
80         }
81     }
82     catch (IOException ioException)
83     {
84         ioException.printStackTrace();
85     }
86 }
87
88 // wait for connection to arrive, then display connection info
89 private void waitForConnection() throws IOException
90 {
91     displayMessage("Waiting for connection\n");
92     connection = server.accept(); // allow server to accept connection
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 2 of 4.)

```
93     displayMessage("Connection " + counter + " received from: " +
94         connection.getInetAddress().getHostName());
95 }
96
97 // get streams to send and receive data
98 private void getStreams() throws IOException
99 {
100     // set up output stream for objects
101     output = new ObjectOutputStream(connection.getOutputStream());
102     output.flush(); // Flush output buffer to send header information
103
104     // set up input stream for objects
105     input = new ObjectInputStream(connection.getInputStream());
106
107     displayMessage("\nGot I/O streams\n");
108 }
109
110 // process connection with client
111 private void processConnection() throws IOException
112 {
113     String message = "Connection successful";
114     sendData(message); // send connection successful message
115
116     // enable enterField so server user can send messages
117     setTextFieldEditable(true);
118
119     do // process messages sent from client
120     {
121         try // read message and display it
122         {
123             message = (String) input.readObject(); // read new message
124             displayMessage("\n" + message); // display message
125         }
126         catch (ClassNotFoundException classNotFoundException)
127         {
128             displayMessage("\nUnknown object type received");
129         }
130
131     } while (!message.equals("CLIENT>>> TERMINATE"));
132 }
133
134 // close streams and socket
135 private void closeConnection()
136 {
137     displayMessage("\nTerminating connection\n");
138     setTextFieldEditable(false); // disable enterField
139
140     try
141     {
142         output.close(); // close output stream
143         input.close(); // close input stream
144         connection.close(); // close socket
145     }
146 }
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 3 of 4.)

```
146     catch (IOException ioException)
147     {
148         ioException.printStackTrace();
149     }
150 }
151
152 // send message to client
153 private void sendData(String message)
154 {
155     try // send object to client
156     {
157         output.writeObject("SERVER>>> " + message);
158         output.flush(); // flush output to client
159         displayMessage("\nSERVER>>> " + message);
160     }
161     catch (IOException ioException)
162     {
163         displayArea.append("\nError writing object");
164     }
165 }
166
167 // manipulates displayArea in the event-dispatch thread
168 private void displayMessage(final String messageToDisplay)
169 {
170     SwingUtilities.invokeLater(
171         new Runnable()
172         {
173             public void run() // updates displayArea
174             {
175                 displayArea.append(messageToDisplay); // append message
176             }
177         }
178     );
179 }
180
181 // manipulates enterField in the event-dispatch thread
182 private void setTextFieldEditable(final boolean editable)
183 {
184     SwingUtilities.invokeLater(
185         new Runnable()
186         {
187             public void run() // sets enterField's editability
188             {
189                 enterField.setEditable(editable);
190             }
191         }
192     );
193 }
194 }
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 4 of 4.)

```
1 // Fig. 28.4: ServerTest.java
2 // Test the Server application.
3 import javax.swing.JFrame;
4
5 public class ServerTest
6 {
7     public static void main(String[] args)
8     {
9         Server application = new Server(); // create server
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.runServer(); // run server application
12    }
13 }
```

Fig. 28.4 | Test class for Server.

Method `runServer`

Method `runServer` (Fig. 28.3, lines 57–86) sets up the server to receive a connection and processes one connection at a time. Line 61 creates a `ServerSocket` called `server` to wait for connections. The `ServerSocket` listens for a connection from a client at port 12345. The second argument to the constructor is the number of connections that can wait in a queue to connect to the server (100 in this example). If the queue is full when a client attempts to connect, the server refuses the connection.



Common Programming Error 28.1

Specifying a port that's already in use or specifying an invalid port number when creating a `ServerSocket` results in a `BindException`.

Line 67 calls method `waitForConnection` (declared at lines 89–95) to wait for a client connection. After the connection is established, line 68 calls method `getStreams` (declared at lines 98–108) to obtain references to the connection's streams. Line 69 calls method `processConnection` (declared at lines 111–132) to send the initial connection message to the client and to process all messages received from the client. The `finally` block (lines 75–79) terminates the client connection by calling method `closeConnection` (lines 135–150), even if an exception occurs. These methods call `displayMessage` (lines 168–179), which uses the event-dispatch thread to display messages in the application's `JTextArea`. `SwingUtilities` method `invokeLater` receives a `Runnable` object as its argument and places it into the event-dispatch thread for execution. This ensures that we don't modify a GUI component from a thread other than the event-dispatch thread, which is important since *Swing GUI components are not thread safe*. We use a similar technique in method `setTextFieldEditable` (lines 182–193), to set the editability of `enterField`. For more information on interface `Runnable`, see Chapter 23.

Method `waitForConnection`

Method `waitForConnection` (lines 89–95) uses `ServerSocket` method `accept` (line 92) to wait for a connection from a client. When a connection occurs, the resulting `Socket` is assigned to `connection`. Method `accept` blocks until a connection is received (i.e., the thread in which `accept` is called stops executing until a client connects). Lines 93–94 output the host name of the computer that made the connection. `Socket` method `getInet-`

Address returns an **InetAddress** (package `java.net`) containing information about the client computer. **InetAddress** method **getHostName** returns the host name of the client computer. For example, a special IP address (**127.0.0.1**) and host name (**localhost**) are useful for testing networking applications on your local computer (this is also known as the **loopback address**). If **getHostName** is called on an **InetAddress** containing **127.0.0.1**, the corresponding host name returned by the method will be **localhost**.

Method **getStreams**

Method **getStreams** (lines 98–108) obtains the **Socket**'s streams and uses them to initialize an **ObjectOutputStream** (line 101) and an **ObjectInputStream** (line 105), respectively. Note the call to **ObjectOutputStream** method **flush** at line 102. This statement causes the **ObjectOutputStream** on the server to send a **stream header** to the corresponding client's **ObjectInputStream**. The stream header contains such information as the version of object serialization being used to send objects. This information is required by the **ObjectInputStream** so that it can prepare to receive those objects correctly.



Software Engineering Observation 28.4

*When using **ObjectOutputStream** and **ObjectInputStream** to send and receive data over a network connection, always create the **ObjectOutputStream** first and **flush** the stream so that the client's **ObjectInputStream** can prepare to receive the data. This is required for networking applications that communicate using **ObjectOutputStream** and **ObjectInputStream**.*



Performance Tip 28.3

A computer's I/O components are typically much slower than its memory. Output buffers are used to increase the efficiency of an application by sending larger amounts of data fewer times, reducing the number of times an application accesses the computer's I/O components.

Method **processConnection**

Line 114 of method **processConnection** (lines 111–132) calls method **sendData** to send "SERVER>>> Connection successful" as a **String** to the client. The loop at lines 119–131 executes until the server receives the message "CLIENT>>> TERMINATE". Line 123 uses **ObjectInputStream** method **readObject** to read a **String** from the client. Line 124 invokes method **displayMessage** to append the message to the **JTextArea**.

Method **closeConnection**

When the transmission is complete, method **processConnection** returns, and the program calls method **closeConnection** (lines 135–150) to close the streams associated with the **Socket** and close the **Socket**. Then the server waits for the next connection attempt from a client by continuing with line 67 at the beginning of the **while** loop.

Server receives a connection, processes it, closes it and waits for the next connection. A more likely scenario would be a Server that receives a connection, sets it up to be processed as a separate thread of execution, then immediately waits for new connections. The separate threads that process existing connections can continue to execute while the Server concentrates on new connection requests. This makes the server more efficient, because multiple client requests can be processed concurrently. We demonstrate a *multi-threaded server* in Section 28.7.

Processing User Interactions

When the user of the server application enters a `String` in the text field and presses the `Enter` key, the program calls method `actionPerformed` (lines 39–43), which reads the `String` from the text field and calls utility method `sendData` (lines 153–165) to send the `String` to the client. Method `sendData` writes the object, flushes the output buffer and appends the same `String` to the text area in the server window. It's not necessary to invoke `displayMessage` to modify the text area here, because method `sendData` is called from an event handler—thus, `sendData` executes as part of the *event-dispatch thread*.

Client Class

Like class `Server`, class `Client`'s constructor (Fig. 28.5, lines 29–56) creates the GUI of the application (a `JTextField` and a `JTextArea`). `Client` displays its output in the text area. When method `main` (lines 7–19 of Fig. 28.6) executes, it creates an instance of class `Client`, specifies the window's default close operation and calls method `runClient` (Fig. 28.5, lines 59–79). In this example, you can execute the client from any computer on the Internet and specify the IP address or host name of the server computer as a command-line argument to the program. For example, the command

```
java Client 192.168.1.15
```

attempts to connect to the Server on the computer with IP address 192.168.1.15.

```
1 // Fig. 28.5: Client.java
2 // Client portion of a stream-socket connection between client and server.
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.InetAddress;
8 import java.net.Socket;
9 import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Client extends JFrame
19 {
20     private JTextField enterField; // enters information from user
21     private JTextArea displayArea; // display information to user
22     private ObjectOutputStream output; // output stream to server
23     private ObjectInputStream input; // input stream from server
24     private String message = ""; // message from server
25     private String chatServer; // host server for this application
26     private Socket client; // socket to communicate with server
27 }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part I of 5.)

```
28 // initialize chatServer and set up GUI
29 public Client(String host)
30 {
31     super("Client");
32
33     chatServer = host; // set server to which this client connects
34
35     enterField = new JTextField(); // create enterField
36     enterField.setEditable(false);
37     enterField.addActionListener(
38         new ActionListener()
39     {
40         // send message to server
41         public void actionPerformed(ActionEvent event)
42         {
43             sendData(event.getActionCommand());
44             enterField.setText("");
45         }
46     }
47 );
48
49     add(enterField, BorderLayout.NORTH);
50
51     displayArea = new JTextArea(); // create displayArea
52     add(new JScrollPane(displayArea), BorderLayout.CENTER);
53
54     setSize(300, 150); // set size of window
55     setVisible(true); // show window
56 }
57
58 // connect to server and process messages from server
59 public void runClient()
60 {
61     try // connect to server, get streams, process connection
62     {
63         connectToServer(); // create a Socket to make connection
64         getStreams(); // get the input and output streams
65         processConnection(); // process connection
66     }
67     catch (EOFException eofException)
68     {
69         displayMessage("\nClient terminated connection");
70     }
71     catch (IOException ioException)
72     {
73         ioException.printStackTrace();
74     }
75     finally
76     {
77         closeConnection(); // close connection
78     }
79 }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 2 of 5.)

```
80 // connect to server
81 private void connectToServer() throws IOException
82 {
83     displayMessage("Attempting connection\n");
84
85     // create Socket to make connection to server
86     client = new Socket(InetAddress.getByName(chatServer), 12345);
87
88     // display connection information
89     displayMessage("Connected to: " +
90                     client.getInetAddress().getHostName());
91 }
92
93
94 // get streams to send and receive data
95 private void getStreams() throws IOException
96 {
97     // set up output stream for objects
98     output = new ObjectOutputStream(client.getOutputStream());
99     output.flush(); // flush output buffer to send header information
100
101    // set up input stream for objects
102    input = new ObjectInputStream(client.getInputStream());
103
104    displayMessage("\nGot I/O streams\n");
105 }
106
107 // process connection with server
108 private void processConnection() throws IOException
109 {
110     // enable enterField so client user can send messages
111     setTextFieldEditable(true);
112
113     do // process messages sent from server
114     {
115         try // read message and display it
116         {
117             message = (String) input.readObject(); // read new message
118             displayMessage("\n" + message); // display message
119         }
120         catch (ClassNotFoundException classNotFoundException)
121         {
122             displayMessage("\nUnknown object type received");
123         }
124
125     } while (!message.equals("SERVER>>> TERMINATE"));
126 }
127
128 // close streams and socket
129 private void closeConnection()
130 {
131     displayMessage("\nClosing connection");
132     setTextFieldEditable(false); // disable enterField
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 3 of 5.)

```
133
134     try
135     {
136         output.close(); // close output stream
137         input.close(); // close input stream 1
138         client.close(); // close socket
139     }
140     catch (IOException ioException)
141     {
142         ioException.printStackTrace();
143     }
144 }
145
146 // send message to server
147 private void sendData(String message)
148 {
149     try // send object to server
150     {
151         output.writeObject("CLIENT>> " + message);
152         output.flush(); // flush data to output
153         displayMessage("\nCLIENT>> " + message);
154     }
155     catch (IOException ioException)
156     {
157         displayArea.append("\nError writing object");
158     }
159 }
160
161 // manipulates displayArea in the event-dispatch thread
162 private void displayMessage(final String messageToDisplay)
163 {
164     SwingUtilities.invokeLater(
165         new Runnable()
166         {
167             public void run() // updates displayArea
168             {
169                 displayArea.append(messageToDisplay);
170             }
171         }
172     );
173 }
174
175 // manipulates enterField in the event-dispatch thread
176 private void setTextFieldEditable(final boolean editable)
177 {
178     SwingUtilities.invokeLater(
179         new Runnable()
180         {
181             public void run() // sets enterField's editability
182             {
183                 enterField.setEditable(editable);
184             }
185         }
186     );
187 }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 4 of 5.)

```

186         );
187     }
188 }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 5 of 5.)

```

1 // Fig. 28.6: ClientTest.java
2 // Class that tests the Client.
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main(String[] args)
8     {
9         Client application; // declare client application
10
11         // if no command line args
12         if (args.length == 0)
13             application = new Client("127.0.0.1"); // connect to localhost
14         else
15             application = new Client(args[0]); // use args to connect
16
17         application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         application.runClient(); // run client application
19     }
20 }
```

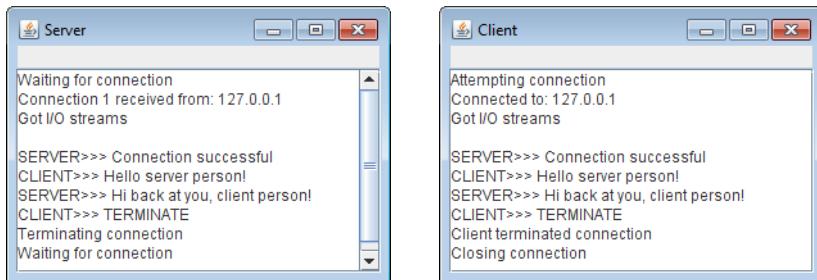


Fig. 28.6 | Class that tests the Client.

Method `runClient`

Client method `runClient` (Fig. 28.5, lines 59–79) sets up the connection to the server, processes messages received from the server and closes the connection when communication is complete. Line 63 calls method `connectToServer` (declared at lines 82–92) to perform the connection. After connecting, line 64 calls method `getStreams` (declared at lines 95–105) to obtain references to the `Socket`'s stream objects. Then line 65 calls method `processConnection` (declared at lines 108–126) to receive and display messages sent from the server. The `finally` block (lines 75–78) calls `closeConnection` (lines 129–144) to close the streams and the `Socket` even if an exception occurred. Method `displayMessage` (lines 162–173) is called from these methods to use the event-dispatch thread to display messages in the application's text area.

Method connectToServer

Method `connectToServer` (lines 82–92) creates a `Socket` called `client` (line 87) to establish a connection. The arguments to the `Socket` constructor are the IP address of the server computer and the port number (12345) where the server application is awaiting client connections. In the first argument, `InetAddress` static method `getByName` returns an `InetAddress` object containing the IP address specified as a command-line argument to the application (or 127.0.0.1 if none was specified). Method `getByName` can receive a `String` containing either the actual IP address or the host name of the server. The first argument also could have been written other ways. For the `localhost` address 127.0.0.1, the first argument could be specified with either of the following expressions:

```
InetAddress.getByName("localhost")
InetAddress.getLocalHost()
```

Other versions of the `Socket` constructor receive the IP address or host name as a `String`. The first argument could have been specified as the IP address "127.0.0.1" or the host name "localhost". We chose to demonstrate the client/server relationship by connecting between applications on the same computer (`localhost`). Normally, this first argument would be the IP address of another computer. The `InetAddress` object for another computer can be obtained by specifying the computer's IP address or host name as the argument to `InetAddress` method `getByName`. The `Socket` constructor's second argument is the server port number. This *must* match the port number at which the server is waiting for connections (called the *handshake point*). Once the connection is made, lines 90–91 display a message in the text area indicating the name of the server computer to which the client has connected.

The `Client` uses an `ObjectOutputStream` to send data to the server and an `ObjectInputStream` to receive data from the server. Method `getStreams` (lines 95–105) creates the `ObjectOutputStream` and `ObjectInputStream` objects that use the streams associated with the `client` socket.

Methods processConnection and closeConnection

Method `processConnection` (lines 108–126) contains a loop that executes until the client receives the message "SERVER>>> TERMINATE". Line 117 reads a `String` object from the server. Line 118 invokes `displayMessage` to append the message to the text area. When the transmission is complete, method `closeConnection` (lines 129–144) closes the streams and the `Socket`.

Processing User Interactions

When the client application user enters a `String` in the text field and presses *Enter*, the program calls method `actionPerformed` (lines 41–45) to read the `String`, then invokes utility method `sendData` (147–159) to send the `String` to the server. Method `sendData` writes the object, flushes the output buffer and appends the same `String` to the client window's `JTextArea`. Once again, it's not necessary to invoke utility method `displayMessage` to modify the text area here, because method `sendData` is called from an event handler.

28.6 Datagrams: Connectionless Client/Server Interaction

We've been discussing connection-oriented, streams-based transmission. Now we consider **connectionless transmission with datagrams**.

Connection-oriented transmission is like the telephone system in which you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for your phone call, *even when you're not talking*.

Connectionless transmission with datagrams is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate pieces that you place in sequentially numbered envelopes. All of the letters are then mailed at once. The letters could arrive *in order*, *out of order* or *not at all* (the last case is rare). The person at the receiving end *reassembles* the pieces into sequential order before attempting to make sense of the message.

If your message is small enough to fit in one envelope, you need not worry about the “out-of-sequence” problem, but it’s still possible that your message might not arrive. One advantage of datagrams over postal mail is that duplicates of datagrams can arrive at the receiving computer.

Figures 28.7–28.10 use datagrams to send packets of information via the User Datagram Protocol (UDP) between a client application and a server application. In the Client application (Fig. 28.9), the user types a message into a text field and presses *Enter*. The program converts the message into a byte array and places it in a datagram packet that’s sent to the server. The Server (Figs. 28.7–28.8) receives the packet and displays the information in it, then *echoes* the packet back to the client. Upon receiving the packet, the client displays the information it contains.

Server Class

Class Server (Fig. 28.7) declares two **DatagramPackets** that the server uses to send and receive information and one **DatagramSocket** that sends and receives the packets. The constructor (lines 19–37), which is called from `main` (Fig. 28.8, lines 7–12), creates the GUI in which the packets of information will be displayed. Line 30 creates the `DatagramSocket` in a `try` block. Line 30 in Fig. 28.7 uses the `DatagramSocket` constructor that takes an integer port-number argument (5000 in this example) to bind the server to a port where it can receive packets from clients. Clients sending packets to this `Server` specify the same port number in the packets they send. A **SocketException** is thrown if the `DatagramSocket` constructor fails to bind the `DatagramSocket` to the specified port.



Common Programming Error 28.2

Specifying a port that’s already in use or specifying an invalid port number when creating a `DatagramSocket` results in a `SocketException`.

```
1 // Fig. 28.7: Server.java
2 // Server side of connectionless client/server computing with datagrams.
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.SocketException;
7 import java.awt.BorderLayout;
8 import javax.swing.JFrame;
9 import javax.swing.JScrollPane;
10 import javax.swing.JTextArea;
```

Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 1 of 3.)

```
11 import javax.swing.SwingUtilities;
12
13 public class Server extends JFrame
14 {
15     private JTextArea displayArea; // displays packets received
16     private DatagramSocket socket; // socket to connect to client
17
18     // set up GUI and DatagramSocket
19     public Server()
20     {
21         super("Server");
22
23         displayArea = new JTextArea(); // create displayArea
24         add(new JScrollPane(displayArea), BorderLayout.CENTER);
25         setSize(400, 300); // set size of window
26         setVisible(true); // show window
27
28         try // create DatagramSocket for sending and receiving packets
29         {
30             socket = new DatagramSocket(5000);
31         }
32         catch (SocketException socketException)
33         {
34             socketException.printStackTrace();
35             System.exit(1);
36         }
37     }
38
39     // wait for packets to arrive, display data and echo packet to client
40     public void waitForPackets()
41     {
42         while (true)
43         {
44             try // receive packet, display contents, return copy to client
45             {
46                 byte[] data = new byte[100]; // set up packet
47                 DatagramPacket receivePacket =
48                     new DatagramPacket(data, data.length);
49
50                 socket.receive(receivePacket); // wait to receive packet
51
52                 // display information from received packet
53                 displayMessage("\nPacket received:" +
54                     "\nFrom host: " + receivePacket.getAddress() +
55                     "\nHost port: " + receivePacket.getPort() +
56                     "\nLength: " + receivePacket.getLength() +
57                     "\nContaining:\n\t" + new String(receivePacket.getData(),
58                         0, receivePacket.getLength()));
59
60                 sendPacketToClient(receivePacket); // send packet to client
61             }
62             catch (IOException ioException)
63             {
```

Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 2 of 3.)

```

64             displayMessage(ioException + "\n");
65         }
66     }
67 }
68 }
69
70 // echo packet to client
71 private void sendPacketToClient(DatagramPacket receivePacket)
72 throws IOException
73 {
74     displayMessage("\n\nEcho data to client...");
75
76     // create packet to send
77     DatagramPacket sendPacket = new DatagramPacket(
78         receivePacket.getData(), receivePacket.getLength(),
79         receivePacket.getAddress(), receivePacket.getPort());
80
81     socket.send(sendPacket); // send packet to client
82     displayMessage("Packet sent\n");
83 }
84
85 // manipulates displayArea in the event-dispatch thread
86 private void displayMessage(final String messageToDisplay)
87 {
88     SwingUtilities.invokeLater(
89         new Runnable()
90     {
91         public void run() // updates displayArea
92         {
93             displayArea.append(messageToDisplay); // display message
94         }
95     });
96 }
97 }
98 }
```

Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 3 of 3.)

```

1 // Fig. 28.8: ServerTest.java
2 // Class that tests the Server.
3 import javax.swing.JFrame;
4
5 public class ServerTest
6 {
7     public static void main(String[] args)
8     {
9         Server application = new Server(); // create server
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.waitForPackets(); // run server application
12    }
13 }
```

Fig. 28.8 | Class that tests the Server. (Part 1 of 2.)

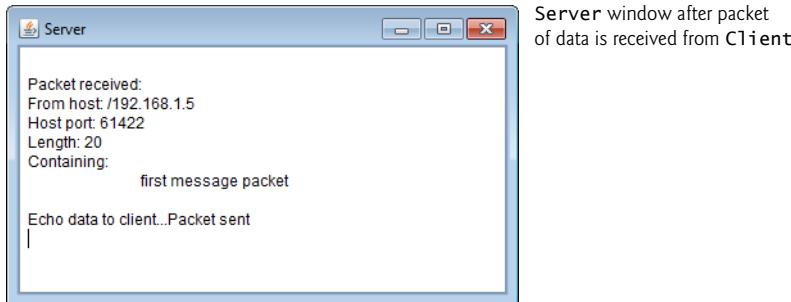


Fig. 28.8 | Class that tests the Server. (Part 2 of 2.)

Method waitForPackets

Server method `waitForPackets` (Fig. 28.7, lines 40–68) uses an infinite loop to wait for packets to arrive at the Server. Lines 47–48 create a `DatagramPacket` in which a received packet of information can be stored. The `DatagramPacket` constructor for this purpose receives two arguments—a byte array in which the data will be stored and the length of the array. Line 50 uses `DatagramSocket` method `receive` to wait for a packet to arrive at the Server. Method `receive` blocks until a packet arrives, then stores the packet in its `DatagramPacket` argument. The method throws an `IOException` if an error occurs while receiving a packet.

Method displayMessage

When a packet arrives, lines 53–58 call method `displayMessage` (declared at lines 86–97) to append the packet’s contents to the text area. `DatagramPacket` method `getAddress` (line 54) returns an `InetAddress` object containing the IP address of the computer from which the packet was sent. Method `getPort` (line 55) returns an integer specifying the port number through which the client computer sent the packet. Method `getLength` (line 56) returns an integer representing the number of bytes of data received. Method `getData` (line 57) returns a byte array containing the data. Lines 57–58 initialize a `String` object using a three-argument constructor that takes a byte array, the offset and the length. This `String` is then appended to the text to display.

Method sendPacketToClient

After displaying a packet, line 60 calls method `sendPacketToClient` (declared at lines 71–83) to create a new packet and send it to the client. Lines 77–79 create a `DatagramPacket` and pass four arguments to its constructor. The first argument specifies the byte array to send. The second argument specifies the number of bytes to send. The third argument specifies the client computer’s IP address, to which the packet will be sent. The fourth argument specifies the port where the client is waiting to receive packets. Line 81 sends the packet over the network. Method `send` of `DatagramSocket` throws an `IOException` if an error occurs while sending a packet.

Client Class

The Client (Figs. 28.9–28.10) works similarly to class `Server`, except that the `Client` sends packets only when the user types a message in a text field and presses the *Enter* key.

When this occurs, the program calls method `actionPerformed` (Fig. 28.9, lines 32–57), which converts the `String` the user entered into a byte array (line 41). Lines 44–45 create a `DatagramPacket` and initialize it with the byte array, the length of the `String` that was entered by the user, the IP address to which the packet is to be sent (`InetAddress.getLocalHost()` in this example) and the port number at which the Server is waiting for packets (5000 in this example). Line 47 sends the packet. The client in this example must know that the server is receiving packets at port 5000—otherwise, the server will *not* receive the packets.

The `DatagramSocket` constructor call (Fig. 28.9, line 71) in this application does not specify any arguments. This no-argument constructor allows the computer to select the next available port number for the `DatagramSocket`. The client does not need a specific port number, because the server receives the client's port number as part of each `DatagramPacket` sent by the client. Thus, the server can send packets back to the same computer and port number from which it receives a packet of information.

```
1 // Fig. 28.9: Client.java
2 // Client side of connectionless client/server computing with datagrams.
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.InetAddress;
7 import java.net.SocketException;
8 import java.awt.BorderLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Client extends JFrame
18 {
19     private JTextField enterField; // for entering messages
20     private JTextArea displayArea; // for displaying messages
21     private DatagramSocket socket; // socket to connect to server
22
23     // set up GUI and DatagramSocket
24     public Client()
25     {
26         super("Client");
27
28         enterField = new JTextField("Type message here");
29         enterField.addActionListener(
30             new ActionListener()
31             {
32                 public void actionPerformed(ActionEvent event)
33                 {
34                     try // create and send packet
35                     {
```

Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 1 of 3.)

```
36         // get message from textfield
37         String message = event.getActionCommand();
38         displayArea.append("\nSending packet containing: " +
39                         message + "\n");
40
41         byte[] data = message.getBytes(); // convert to bytes
42
43         // create sendPacket
44         DatagramPacket sendPacket = new DatagramPacket(data,
45                         data.length, InetAddress.getLocalHost(), 5000);
46
47         socket.send(sendPacket); // send packet
48         displayArea.append("Packet sent\n");
49         displayArea.setCaretPosition(
50                         displayArea.getText().length());
51     }
52     catch (IOException ioException)
53     {
54         displayMessage(ioException + "\n");
55         ioException.printStackTrace();
56     }
57 }
58 );
59
60 add(enterField, BorderLayout.NORTH);
61
62 displayArea = new JTextArea();
63 add(new JScrollPane(displayArea), BorderLayout.CENTER);
64
65 setSize(400, 300); // set window size
66 setVisible(true); // show window
67
68 try // create DatagramSocket for sending and receiving packets
69 {
70     socket = new DatagramSocket();
71 }
72 catch (SocketException socketException)
73 {
74     socketException.printStackTrace();
75     System.exit(1);
76 }
77 }
78
79 // wait for packets to arrive from Server, display packet contents
80 public void waitForPackets()
81 {
82     while (true)
83     {
84         try // receive packet and display contents
85         {
86             byte[] data = new byte[100]; // set up packet
```

Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 2 of 3.)

```

88         DatagramPacket receivePacket = new DatagramPacket(
89             data, data.length);
90
91         socket.receive(receivePacket); // wait for packet
92
93         // display packet contents
94         displayMessage("\nPacket received:" +
95             "\nFrom host: " + receivePacket.getAddress() +
96             "\nHost port: " + receivePacket.getPort() +
97             "\nLength: " + receivePacket.getLength() +
98             "\nContaining:\n\t" + new String(receivePacket.getData(),
99             0, receivePacket.getLength()));
100    }
101    catch (IOException exception)
102    {
103        displayMessage(exception + "\n");
104        exception.printStackTrace();
105    }
106}
107}
108
109 // manipulates displayArea in the event-dispatch thread
110 private void displayMessage(final String messageToDisplay)
111{
112    SwingUtilities.invokeLater(
113        new Runnable()
114        {
115            public void run() // updates displayArea
116            {
117                displayArea.append(messageToDisplay);
118            }
119        }
120    );
121}
122}

```

Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 3 of 3.)

```

1 // Fig. 28.10: ClientTest.java
2 // Tests the Client class.
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main(String[] args)
8     {
9         Client application = new Client(); // create client
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.waitForPackets(); // run client application
12    }
13}

```

Fig. 28.10 | Class that tests the Client. (Part 1 of 2.)

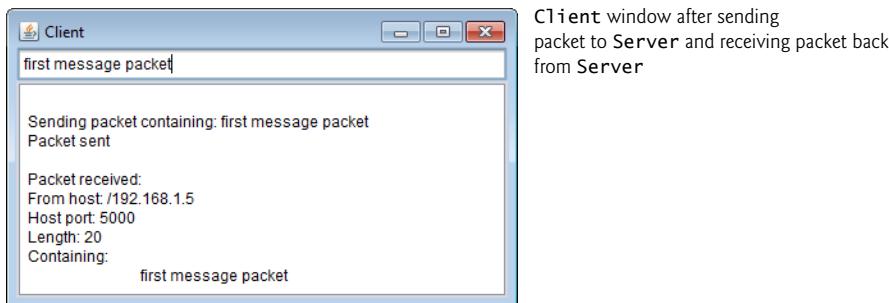


Fig. 28.10 | Class that tests the Client. (Part 2 of 2.)

Method `waitForPackets`

Client method `waitForPackets` (lines 81–107) uses an infinite loop to wait for packets from the server. Line 91 blocks until a packet arrives. This does not prevent the user from sending a packet, because the *GUI events are handled in the event-dispatch thread*. It only prevents the `while` loop from continuing until a packet arrives at the Client. When a packet arrives, line 91 stores it in `receivePacket`, and lines 94–99 call method `displayMessage` (declared at lines 110–121) to display the packet's contents in the text area.

28.7 Client/Server Tic-Tac-Toe Using a Multithreaded Server

This section presents the popular game Tic-Tac-Toe implemented by using client/server techniques with stream sockets. The program consists of a `TicTacToeServer` application (Figs. 28.11–28.12) that allows two `TicTacToeClient` applications (Figs. 28.13–28.14) to connect to the server and play Tic-Tac-Toe. Sample outputs are shown in Fig. 28.15.

`TicTacToeServer` Class

As the `TicTacToeServer` receives each client connection, it creates an instance of inner-class `Player` (Fig. 28.11, lines 182–304) to process the client in a *separate thread*. These threads enable the clients to play the game independently. The first client to connect to the server is player X and the second is player O. Player X makes the first move. The server maintains the information about the board so it can determine if a player's move is valid.

```

1 // Fig. 28.11: TicTacToeServer.java
2 // Server side of client/server Tic-Tac-Toe program.
3 import java.awt.BorderLayout;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6 import java.io.IOException;
7 import java.util.Formatter;
8 import java.util.Scanner;
9 import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 1 of 7.)

```
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13 import java.util.concurrent.locks.Condition;
14 import javax.swing.JFrame;
15 import javax.swing.JTextArea;
16 import javax.swing.SwingUtilities;
17
18 public class TicTacToeServer extends JFrame
19 {
20     private String[] board = new String[9]; // tic-tac-toe board
21     private JTextArea outputArea; // for outputting moves
22     private Player[] players; // array of Players
23     private ServerSocket server; // server socket to connect with clients
24     private int currentPlayer; // keeps track of player with current move
25     private final static int PLAYER_X = 0; // constant for first player
26     private final static int PLAYER_O = 1; // constant for second player
27     private final static String[] MARKS = { "X", "O" }; // array of marks
28     private ExecutorService runGame; // will run players
29     private Lock gameLock; // to lock game for synchronization
30     private Condition otherPlayerConnected; // to wait for other player
31     private Condition otherPlayerTurn; // to wait for other player's turn
32
33     // set up tic-tac-toe server and GUI that displays messages
34     public TicTacToeServer()
35     {
36         super("Tic-Tac-Toe Server"); // set title of window
37
38         // create ExecutorService with a thread for each player
39         runGame = Executors.newFixedThreadPool(2);
40         gameLock = new ReentrantLock(); // create lock for game
41
42         // condition variable for both players being connected
43         otherPlayerConnected = gameLock.newCondition();
44
45         // condition variable for the other player's turn
46         otherPlayerTurn = gameLock.newCondition();
47
48         for (int i = 0; i < 9;
49             board[i] = new String(""); // create tic-tac-toe board
50         players = new Player[2]; // create array of players
51         currentPlayer = PLAYER_X; // set current player to first player
52
53         try
54         {
55             server = new ServerSocket(12345, 2); // set up ServerSocket
56         }
57         catch (IOException ioException)
58         {
59             ioException.printStackTrace();
60             System.exit(1);
61         }
62
63         outputArea = new JTextArea(); // create JTextArea for output
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 2 of 7.)

```
64     add(outputArea, BorderLayout.CENTER);
65     outputArea.setText("Server awaiting connections\n");
66
67     setSize(300, 300); // set size of window
68     setVisible(true); // show window
69 }
70
71 // wait for two connections so game can be played
72 public void execute()
73 {
74     // wait for each client to connect
75     for (int i = 0; i < players.length; i++)
76     {
77         try // wait for connection, create Player, start runnable
78         {
79             players[i] = new Player(server.accept(), i);
80             runGame.execute(players[i]); // execute player runnable
81         }
82         catch (IOException ioException)
83         {
84             ioException.printStackTrace();
85             System.exit(1);
86         }
87     }
88
89     gameLock.lock(); // lock game to signal player X's thread
90
91     try
92     {
93         players[PLAYER_X].setSuspended(false); // resume player X
94         otherPlayerConnected.signal(); // wake up player X's thread
95     }
96     finally
97     {
98         gameLock.unlock(); // unlock game after signalling player X
99     }
100 }
101
102 // display message in outputArea
103 private void displayMessage(final String messageToDisplay)
104 {
105     // display message from event-dispatch thread of execution
106     SwingUtilities.invokeLater(
107         new Runnable()
108         {
109             public void run() // updates outputArea
110             {
111                 outputArea.append(messageToDisplay); // add message
112             }
113         }
114     );
115 }
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 3 of 7.)

```
116 // determine if move is valid
117 public boolean validateAndMove(int location, int player)
118 {
119     // while not current player, must wait for turn
120     while (player != currentPlayer)
121     {
122         gameLock.lock(); // lock game to wait for other player to go
123
124         try
125         {
126             otherPlayerTurn.await(); // wait for player's turn
127         }
128         catch (InterruptedException exception)
129         {
130             exception.printStackTrace();
131         }
132         finally
133         {
134             gameLock.unlock(); // unlock game after waiting
135         }
136     }
137
138     // if location not occupied, make move
139     if (!isOccupied(location))
140     {
141         board[location] = MARKS[currentPlayer]; // set move on board
142         currentPlayer = (currentPlayer + 1) % 2; // change player
143
144         // let new current player know that move occurred
145         players[currentPlayer].otherPlayerMoved(location);
146
147         gameLock.lock(); // lock game to signal other player to go
148
149         try
150         {
151             otherPlayerTurn.signal(); // signal other player to continue
152         }
153         finally
154         {
155             gameLock.unlock(); // unlock game after signaling
156         }
157
158         return true; // notify player that move was valid
159     }
160     else // move was not valid
161     {
162         return false; // notify player that move was invalid
163     }
164
165     // determine whether location is occupied
166     public boolean isOccupied(int location)
167     {
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 4 of 7.)

```
168     if (board[location].equals(MARKS[PLAYER_X]) ||  
169         board [location].equals(MARKS[PLAYER_O]))  
170         return true; // location is occupied  
171     else  
172         return false; // location is not occupied  
173 }  
174  
175 // place code in this method to determine whether game over  
176 public boolean isGameOver()  
177 {  
178     return false; // this is left as an exercise  
179 }  
180  
181 // private inner class Player manages each Player as a runnable  
182 private class Player implements Runnable  
183 {  
184     private Socket connection; // connection to client  
185     private Scanner input; // input from client  
186     private Formatter output; // output to client  
187     private int playerNumber; // tracks which player this is  
188     private String mark; // mark for this player  
189     private boolean suspended = true; // whether thread is suspended  
190  
191     // set up Player thread  
192     public Player(Socket socket, int number)  
193     {  
194         playerNumber = number; // store this player's number  
195         mark = MARKS[playerNumber]; // specify player's mark  
196         connection = socket; // store socket for client  
197  
198         try // obtain streams from Socket  
199         {  
200             input = new Scanner(connection.getInputStream());  
201             output = new Formatter(connection.getOutputStream());  
202         }  
203         catch (IOException ioException)  
204         {  
205             ioException.printStackTrace();  
206             System.exit(1);  
207         }  
208     }  
209  
210     // send message that other player moved  
211     public void otherPlayerMoved(int location)  
212     {  
213         output.format("Opponent moved\n");  
214         output.format("%d\n", location); // send location of move  
215         output.flush(); // flush output  
216     }  
217 }
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 5 of 7.)

```
218     // control thread's execution
219     public void run()
220     {
221         // send client its mark (X or O), process messages from client
222         try
223         {
224             displayMessage("Player " + mark + " connected\n");
225             output.format("%s\n", mark); // send player's mark
226             output.flush(); // flush output
227
228             // if player X, wait for another player to arrive
229             if (playerNumber == PLAYER_X)
230             {
231                 output.format("%s\n%s", "Player X connected",
232                             "Waiting for another player\n");
233                 output.flush(); // flush output
234
235                 gameLock.lock(); // lock game to wait for second player
236
237                 try
238                 {
239                     while(suspended)
240                     {
241                         otherPlayerConnected.await(); // wait for player O
242                     }
243                 }
244                 catch (InterruptedException exception)
245                 {
246                     exception.printStackTrace();
247                 }
248                 finally
249                 {
250                     gameLock.unlock(); // unlock game after second player
251                 }
252
253                 // send message that other player connected
254                 output.format("Other player connected. Your move.\n");
255                 output.flush(); // flush output
256             }
257             else
258             {
259                 output.format("Player O connected, please wait\n");
260                 output.flush(); // flush output
261             }
262
263             // while game not over
264             while (!isGameOver())
265             {
266                 int location = 0; // initialize move location
267
268                 if (input.hasNext())
269                     location = input.nextInt(); // get move location
270             }
271         }
272     }
273 }
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 6 of 7.)

```
271         // check for valid move
272         if (validateAndMove(location, playerNumber))
273         {
274             displayMessage("\nlocation: " + location);
275             output.format("Valid move.\n"); // notify client
276             output.flush(); // flush output
277         }
278         else // move was invalid
279         {
280             output.format("Invalid move, try again\n");
281             output.flush(); // flush output
282         }
283     }
284 }
285 finally
286 {
287     try
288     {
289         connection.close(); // close connection to client
290     }
291     catch (IOException ioException)
292     {
293         ioException.printStackTrace();
294         System.exit(1);
295     }
296 }
297 }
298
299 // set whether or not thread is suspended
300 public void setSuspended(boolean status)
301 {
302     suspended = status; // set value of suspended
303 }
304 }
305 }
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 7 of 7.)

```
1 // Fig. 28.12: TicTacToeServerTest.java
2 // Class that tests Tic-Tac-Toe server.
3 import javax.swing.JFrame;
4
5 public class TicTacToeServerTest
6 {
7     public static void main(String[] args)
8     {
9         TicTacToeServer application = new TicTacToeServer();
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.execute();
12    }
13 }
```

Fig. 28.12 | Class that tests Tic-Tac-Toe server. (Part 1 of 2.)

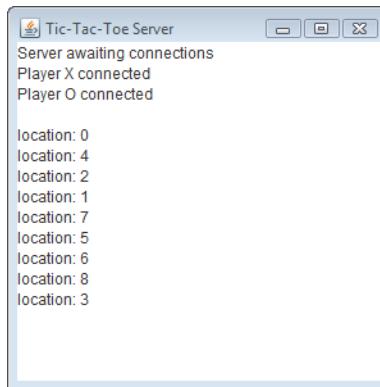


Fig. 28.12 | Class that tests Tic-Tac-Toe server. (Part 2 of 2.)

We begin with a discussion of the server side of the Tic-Tac-Toe game. When the `TicTacToeServer` application executes, the `main` method (lines 7–12 of Fig. 28.12) creates a `TicTacToeServer` object called `application`. The constructor (Fig. 28.11, lines 34–69) attempts to set up a `ServerSocket`. If successful, the program displays the server window, then `main` invokes the `TicTacToeServer` method `execute` (lines 72–100). Method `execute` loops twice, blocking at line 79 each time while waiting for a client connection. When a client connects, line 79 creates a new `Player` object to manage the connection as a separate thread, and line 80 executes the `Player` in the `runGame` thread pool.

When the `TicTacToeServer` creates a `Player`, the `Player` constructor (lines 192–208) receives the `Socket` object representing the connection to the client and gets the associated input and output streams. Line 201 creates a `Formatter` (see Chapter 15) by wrapping it around the output stream of the socket. The `Player`'s `run` method (lines 219–297) controls the information that's sent to and received from the client. First, it passes to the client the character that the client will place on the board when a move is made (line 225). Line 226 calls `Formatter` method `flush` to force this output to the client. Line 241 suspends player X's thread as it starts executing, because player X can move only after player O connects.

When player O connects, the game can be played, and the `run` method begins executing its `while` statement (lines 264–283). Each iteration of this loop reads an integer (line 269) representing the location where the client wants to place a mark (blocking to wait for input, if necessary), and line 272 invokes the `TicTacToeServer` method `validateAndMove` (declared at lines 118–163) to check the move. If the move is valid, line 275 sends a message to the client to this effect. If not, line 280 sends a message indicating that the move was invalid. The program maintains board locations as numbers from 0 to 8 (0 through 2 for the first row, 3 through 5 for the second row and 6 through 8 for the third row).

Method `validateAndMove` (lines 118–163 in class `TicTacToeServer`) allows only one player at a time to move, thereby preventing them from modifying the state information of the game simultaneously. If the `Player` attempting to validate a move is *not* the current player (i.e., the one allowed to make a move), it's placed in a `wait` state until its turn to move. If the position for the move being validated is already occupied on the board,

`validMove` returns `false`. Otherwise, the server places a mark for the player in its local representation of the board (line 142), notifies the other `Player` object (line 146) that a move has been made (so that the client can be sent a message), invokes method `signal` (line 152) so that the waiting `Player` (if there is one) can validate a move and returns `true` (line 159) to indicate that the move is valid.

TicTacToeClient Class

Each `TicTacToeClient` application (Figs. 28.13–28.14; sample outputs in Fig. 28.15) maintains its own GUI version of the Tic-Tac-Toe board on which it displays the state of the game. The clients can place a mark only in an empty square. Inner class `Square` (Fig. 28.13, lines 205–261) implements each of the nine squares on the board. When a `TicTacToeClient` begins execution, it creates a `JTextArea` in which messages from the server and a representation of the board using nine `Square` objects are displayed. The `startClient` method (lines 80–100) opens a connection to the server and gets the associated input and output streams from the `Socket` object. Lines 85–86 make a connection to the server. Class `TicTacToeClient` implements interface `Runnable` so that a separate thread can read messages from the server. This approach enables the user to interact with the board (in the event-dispatch thread) while waiting for messages from the server. After establishing the connection to the server, line 99 executes the client with the `worker ExecutorService`. The `run` method (lines 103–126) controls the separate thread of execution. The method first reads the mark character (X or O) from the server (line 105), then loops continuously (lines 121–125) and reads messages from the server (line 124). Each message is passed to the `processMessage` method (lines 129–156) for processing.

```

1 // Fig. 28.13: TicTacToeClient.java
2 // Client side of client/server Tic-Tac-Toe program.
3 import java.awt.BorderLayout;
4 import java.awt.Dimension;
5 import java.awt.Graphics;
6 import java.awt.GridLayout;
7 import java.awt.event.MouseAdapter;
8 import java.awt.event.MouseEvent;
9 import java.net.Socket;
10 import java.net.InetAddress;
11 import java.io.IOException;
12 import javax.swing.JFrame;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18 import java.util.Formatter;
19 import java.util.Scanner;
20 import java.util.concurrent.Executors;
21 import java.util.concurrent.ExecutorService;
22
23 public class TicTacToeClient extends JFrame implements Runnable
24 {

```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part I of 6.)

```
25  private JTextField idField; // textfield to display player's mark
26  private JTextArea displayArea; // JTextArea to display output
27  private JPanel boardPanel; // panel for tic-tac-toe board
28  private JPanel panel2; // panel to hold board
29  private Square[][] board; // tic-tac-toe board
30  private Square currentSquare; // current square
31  private Socket connection; // connection to server
32  private Scanner input; // input from server
33  private Formatter output; // output to server
34  private String ticTacToeHost; // host name for server
35  private String myMark; // this client's mark
36  private boolean myTurn; // determines which client's turn it is
37  private final String X_MARK = "X"; // mark for first client
38  private final String O_MARK = "O"; // mark for second client
39
40  // set up user-interface and board
41  public TicTacToeClient(String host)
42  {
43      ticTacToeHost = host; // set name of server
44      displayArea = new JTextArea(4, 30); // set up JTextArea
45      displayArea.setEditable(false);
46      add(new JScrollPane(displayArea), BorderLayout.SOUTH);
47
48      boardPanel = new JPanel(); // set up panel for squares in board
49      boardPanel.setLayout(new GridLayout(3, 3, 0, 0));
50
51      board = new Square[3][3]; // create board
52
53      // loop over the rows in the board
54      for (int row = 0; row < board.length; row++)
55      {
56          // loop over the columns in the board
57          for (int column = 0; column < board[row].length; column++)
58          {
59              // create square
60              board[row][column] = new Square(' ', row * 3 + column);
61              boardPanel.add(board[row][column]); // add square
62          }
63      }
64
65      idField = new JTextField(); // set up textfield
66      idField.setEditable(false);
67      add(idField, BorderLayout.NORTH);
68
69      panel2 = new JPanel(); // set up panel to contain boardPanel
70      panel2.add(boardPanel, BorderLayout.CENTER); // add board panel
71      add(panel2, BorderLayout.CENTER); // add container panel
72
73      setSize(300, 225); // set size of window
74      setVisible(true); // show window
75
76      startClient();
77  }
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 2 of 6.)

```
78
79 // start the client thread
80 public void startClient()
81 {
82     try // connect to server and get streams
83     {
84         // make connection to server
85         connection = new Socket(
86             InetAddress.getByName(ticTacToeHost), 12345);
87
88         // get streams for input and output
89         input = new Scanner(connection.getInputStream());
90         output = new Formatter(connection.getOutputStream());
91     }
92     catch (IOException ioException)
93     {
94         ioException.printStackTrace();
95     }
96
97     // create and start worker thread for this client
98     ExecutorService worker = Executors.newFixedThreadPool(1);
99     worker.execute(this); // execute client
100 }
101
102 // control thread that allows continuous update of displayArea
103 public void run()
104 {
105     myMark = input.nextLine(); // get player's mark (X or O)
106
107     SwingUtilities.invokeLater(
108         new Runnable()
109         {
110             public void run()
111             {
112                 // display player's mark
113                 idField.setText("You are player \\" + myMark + "\\");
114             }
115         }
116     );
117
118     myTurn = (myMark.equals(X_MARK)); // determine if client's turn
119
120     // receive messages sent to client and output them
121     while (true)
122     {
123         if (input.hasNextLine())
124             processMessage(input.nextLine());
125     }
126 }
127
128 // process messages received by client
129 private void processMessage(String message)
130 {
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 3 of 6.)

```
131     // valid move occurred
132     if (message.equals("Valid move."))
133     {
134         displayMessage("Valid move, please wait.\n");
135         setMark(currentSquare, myMark); // set mark in square
136     }
137     else if (message.equals("Invalid move, try again"))
138     {
139         displayMessage(message + "\n"); // display invalid move
140         myTurn = true; // still this client's turn
141     }
142     else if (message.equals("Opponent moved"))
143     {
144         int location = input.nextInt(); // get move location
145         input.nextLine(); // skip newline after int location
146         int row = location / 3; // calculate row
147         int column = location % 3; // calculate column
148
149         setMark( board[row][column],
150                 (myMark.equals(X_MARK) ? O_MARK : X_MARK)); // mark move
151         displayMessage("Opponent moved. Your turn.\n");
152         myTurn = true; // now this client's turn
153     }
154     else
155         displayMessage(message + "\n"); // display the message
156 }
157
158 // manipulate displayArea in event-dispatch thread
159 private void displayMessage(final String messageToDisplay)
160 {
161     SwingUtilities.invokeLater(
162         new Runnable()
163         {
164             public void run()
165             {
166                 displayArea.append(messageToDisplay); // updates output
167             }
168         }
169     );
170 }
171
172 // utility method to set mark on board in event-dispatch thread
173 private void setMark(final Square squareToMark, final String mark)
174 {
175     SwingUtilities.invokeLater(
176         new Runnable()
177         {
178             public void run()
179             {
180                 squareToMark.setMark(mark); // set mark in square
181             }
182         }
183     );
184 }
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 4 of 6.)

```
182         }
183     );
184 }
185
186 // send message to server indicating clicked square
187 public void sendClickedSquare(int location)
188 {
189     // if it is my turn
190     if (myTurn)
191     {
192         output.format("%d\n", location); // send location to server
193         output.flush();
194         myTurn = false; // not my turn any more
195     }
196 }
197
198 // set current Square
199 public void setCurrentSquare(Square square)
200 {
201     currentSquare = square; // set current square to argument
202 }
203
204 // private inner class for the squares on the board
205 private class Square extends JPanel
206 {
207     private String mark; // mark to be drawn in this square
208     private int location; // location of square
209
210     public Square(String squareMark, int squareLocation)
211     {
212         mark = squareMark; // set mark for this square
213         location = squareLocation; // set location of this square
214
215         addMouseListener(
216             new MouseAdapter()
217             {
218                 public void mouseReleased(MouseEvent e)
219                 {
220                     setCurrentSquare(Square.this); // set current square
221
222                     // send location of this square
223                     sendClickedSquare(getSquareLocation());
224                 }
225             }
226         );
227     }
228
229     // return preferred size of Square
230     public Dimension getPreferredSize()
231     {
232         return new Dimension(30, 30); // return preferred size
233     }

```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 5 of 6.)

```
234      // return minimum size of Square
235      public Dimension getPreferredSize()
236      {
237          return getPreferredSize(); // return preferred size
238      }
239
240      // set mark for Square
241      public void setMark(String newMark)
242      {
243          mark = newMark; // set mark of square
244          repaint(); // repaint square
245      }
246
247      // return Square location
248      public int getSquareLocation()
249      {
250          return location; // return location of square
251      }
252
253      // draw Square
254      public void paintComponent(Graphics g)
255      {
256          super.paintComponent(g);
257
258          g.drawRect(0, 0, 29, 29); // draw square
259          g.drawString(mark, 11, 20); // draw mark
260      }
261  }
262 }
263 }
```

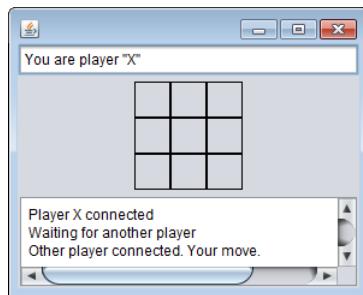
Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 6 of 6.)

```
1  // Fig. 28.14: TicTacToeClientTest.java
2  // Test class for Tic-Tac-Toe client.
3  import javax.swing.JFrame;
4
5  public class TicTacToeClientTest
6  {
7      public static void main(String[] args)
8      {
9          TicTacToeClient application; // declare client application
10
11         // if no command line args
12         if (args.length == 0)
13             application = new TicTacToeClient("127.0.0.1"); // localhost
14         else
15             application = new TicTacToeClient(args[0]); // use args
16
17         application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     }
19 }
```

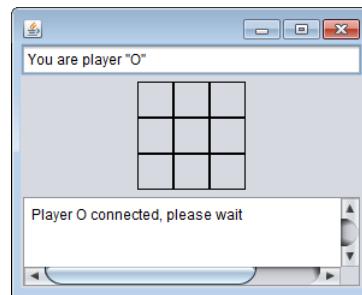
Fig. 28.14 | Test class for Tic-Tac-Toe client.

If the message received is "Valid move.", lines 134–135 display the message "Valid move, please wait." and call method `setMark` (lines 173–184) to set the client's mark in the current square (the one in which the user clicked), using `SwingUtilities` method `invokeLater` to ensure that the GUI updates occur in the event-dispatch thread. If the message received is "Invalid move, try again.", line 139 displays the message so that the user can click a different square. If the message received is "Opponent moved.", line 144 reads an integer from the server indicating where the opponent moved, and lines 149–150 place a mark in that square of the board (again using `SwingUtilities` method `invokeLater` to ensure that the GUI updates occur in the event-dispatch thread). If any other message is received, line 155 simply displays the message.

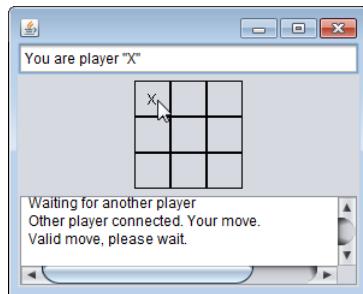
a) Player X connected to server.



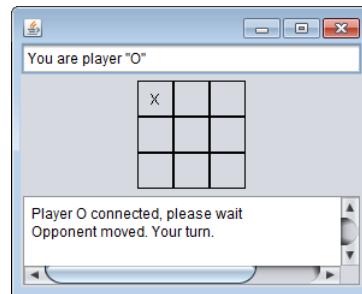
b) Player O connected to server.



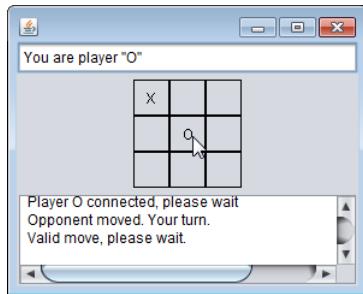
c) Player X moved.



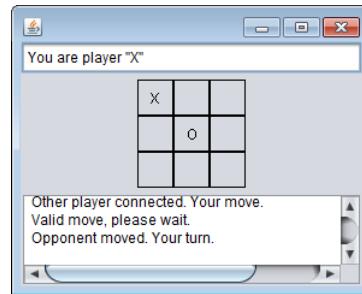
d) Player O sees Player X's move.



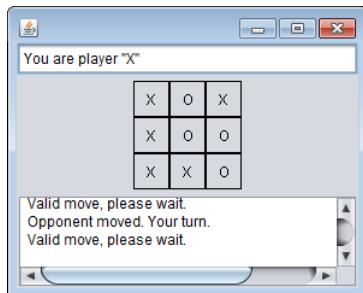
e) Player O moved.



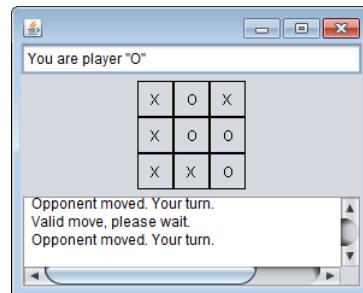
f) Player X sees Player O's move.

**Fig. 28.15** | Sample outputs from the client/server Tic-Tac-Toe program. (Part 1 of 2.)

g) Player X moved.



h) Player O sees Player X's last move.

**Fig. 28.15** | Sample outputs from the client/server Tic-Tac-Toe program. (Part 2 of 2.)

28.8 Optional Online Case Study: DeitelMessenger²

This case study is available at <http://www.deitel.com/books/jhtp11>. Chat rooms provide a central location where users can chat with each other via short text messages. Each participant can see all the messages that the other users post, and each user can post messages. This case study integrates many of the Java networking, multithreading and Swing GUI features you've learned thus far to build an online chat system. We also introduce **multicasting**, which enables an application to send **DatagramPackets** to *groups* of clients.

The DeitelMessenger case study is a significant application that uses many intermediate Java features, such as networking with Sockets, DatagramPackets and Multicast-Sockets, multithreading and Swing GUI. The case study also demonstrates good software engineering practices by separating interface from implementation and enabling developers to support different network protocols and provide different user interfaces. After reading this case study, you'll be able to build more significant networking applications.

28.9 Wrap-Up

In this chapter, you learned the basics of network programming in Java. You learned two different methods of sending data over a network—streams-based networking using TCP/IP and datagrams-based networking using UDP. We showed how to build simple client/server chat programs using both streams-based and datagram-based networking. You then saw a client/server Tic-Tac-Toe game that enables two clients to play by interacting with a multithreaded server that maintains the game's state and logic. In the next chapter, you'll learn basic database concepts, how to interact with data in a database using SQL and how to use JDBC to allow Java applications to manipulate database data.

2. This case study is from the Seventh Edition of this book and is provided as is. We no longer provide support for it.

Summary

Section 28.1 Introduction

- Java provides stream sockets and datagram sockets (p. 2). With stream sockets (p. 2), a process establishes a connection (p. 2) to another process. While the connection is in place, data flows between the processes in streams. Stream sockets are said to provide a connection-oriented service (p. 2). The protocol used for transmission is the popular TCP (Transmission Control Protocol; p. 2).
- With datagram sockets (datagram socket), individual packets of information are transmitted. UDP (User Datagram Protocol; p. 2) is a connectionless service that does not guarantee that packets will not be lost, duplicated or arrive out of sequence.

Section 28.2 Reading a File on a Web Server

- JEditorPane (p. 3) method setPage (p. 5) downloads the document specified by its argument and displays it.
- Typically, an HTML document contains hyperlinks that link to other documents on the web. If an HTML document is displayed in an uneditable JEditorPane and the user clicks a hyperlink (p. 5), a HyperlinkEvent (p. 5) occurs and the HyperlinkListeners are notified.
- HyperlinkEvent method getEventType (p. 5) determines the event type. HyperlinkEvent contains nested class EventType (p. 5), which declares event types ACTIVATED, ENTERED and EXITED. HyperlinkEvent method getURL (p. 5) obtains the URL represented by the hyperlink.

Section 28.3 Establishing a Simple Server Using Stream Sockets

- Stream-based connections (p. 2) are managed with Socket objects (p. 6).
- A ServerSocket object (p. 6) establishes the port (p. 6) where a server (p. 2) waits for connections from clients (p. 2). ServerSocket method accept (p. 6) waits indefinitely for a connection from a client and returns a Socket object when a connection is established.
- Socket methods getOutputStream and getInputStream (p. 7) get references to a Socket's OutputStream and InputStream, respectively. Method close (p. 7) terminates a connection.

Section 28.4 Establishing a Simple Client Using Stream Sockets

- A server name and port number (p. 6) are specified when creating a Socket object to enable it to connect a client to the server. A failed connection attempt throws an IOException.
- InetAddress method getByName (p. 20) returns an InetAddress object (p. 14) containing the IP address of the specified computer. InetAddress method getLocalHost (p. 20) returns an InetAddress object containing the IP address of the local computer executing the program.

Section 28.6 Datagrams: Connectionless Client/Server Interaction

- Connection-oriented transmission is like the telephone system—you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you aren't talking.
- Connectionless transmission (p. 20) with datagrams is similar to mail carried via the postal service. A large message that will not fit in one envelope can be broken into separate message pieces that are placed in separate, sequentially numbered envelopes. All the letters are then mailed at once. They could arrive in order, out of order or not at all.
- DatagramPacket objects store packets of data that are to be sent or that are received by an application. DatagramSockets send and receive DatagramPackets.

- The `DatagramSocket` constructor that takes no arguments binds the `DatagramSocket` to a port chosen by the computer executing the program. The one that takes an integer port-number argument binds the `DatagramSocket` to the specified port. If a `DatagramSocket` constructor fails to bind the `DatagramSocket` to a port, a `SocketException` occurs (p. 21). `DatagramSocket` method `receive` (p. 24) blocks (waits) until a packet arrives, then stores the packet in its argument.
- `DatagramPacket` method `getAddress` (p. 24) returns an `InetAddress` object containing information about the computer from or to which the packet was sent. Method `getPort` (p. 24) returns an integer specifying the port number (p. 6) through which the `DatagramPacket` was sent or received. Method `getLength` (`getLength`) returns the number of bytes of data in a `DatagramPacket`. Method `getData` (p. 24) returns a byte array containing the data.
- The `DatagramPacket` constructor for a packet to be sent takes four arguments—the byte array to be sent, the number of bytes to be sent, the client address to which the packet will be sent and the port number where the client is waiting to receive packets.
- `DatagramSocket` method `send` (p. 24) sends a `DatagramPacket` out over the network.
- If an error occurs when receiving or sending a `DatagramPacket`, an `IOException` occurs.

Self-Review Exercises

28.1 Fill in the blanks in each of the following statements:

- Exception _____ occurs when an input/output error occurs when closing a socket.
- Exception _____ occurs when a hostname indicated by a client cannot be resolved to an address.
- If a `DatagramSocket` constructor fails to set up a `DatagramSocket` properly, an exception of type _____ occurs.
- Many of Java's networking classes are contained in package _____.
- Class _____ binds the application to a port for datagram transmission.
- An object of class _____ contains an IP address.
- The two types of sockets we discussed in this chapter are _____ and _____.
- Method `getLocalHost` returns a(n) _____ object containing the local IP address of the computer on which the program is executing.
- The `URL` constructor determines whether its `String` argument is a valid URL. If so, the `URL` object is initialized with that location. If not, a(n) _____ exception occurs.

28.2 State whether each of the following is *true or false*. If *false*, explain why.

- UDP is a connection-oriented protocol.
- With stream sockets a process establishes a connection to another process.
- A server waits at a port for connections from a client.
- Datagram packet transmission over a network is reliable—packets are guaranteed to arrive in sequence.

Answers to Self-Review Exercises

28.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) stream sockets, datagram sockets. h) `InetAddress`. i) `MalformedURLException`.

28.2 a) False; UDP is a connectionless protocol and TCP is a connection-oriented protocol.
b) True. c) True. d) False; packets can be lost, arrive out of order or be duplicated.

Exercises

- 28.3** Distinguish between connection-oriented and connectionless network services.
- 28.4** How does a client determine the hostname of the client computer?
- 28.5** Under what circumstances would a `SocketException` be thrown?
- 28.6** How can a client get a line of text from a server?
- 28.7** Describe how a client connects to a server.
- 28.8** Describe how a server sends data to a client.
- 28.9** Describe how to prepare a server to receive a stream-based connection from a single client.
- 28.10** How does a server listen for streams-based socket connections at a port?
- 28.11** What determines how many connect requests from clients can wait in a queue to connect to a server?
- 28.12** As described in the text, what reasons might cause a server to refuse a connection request from a client?
- 28.13** Use a socket connection to allow a client to specify a filename of a text file and have the server send the contents of the file or indicate that the file does not exist.
- 28.14** Modify Exercise 28.13 to allow the client to modify the contents of the file and send the file back to the server for storage. The user can edit the file in a `JTextArea`, then click a *save changes* button to send the file back to the server.
- 28.15** (*Multithreaded Server*) Multithreaded servers are quite popular today, especially because of the increasing use of multi-core servers. Modify the simple server application presented in Section 28.5 to be a multithreaded server. Then use several client applications and have each of them connect to the server simultaneously. Use an `ArrayList` to store the client threads. `ArrayList` provides several methods to use in this exercise. Method `size` determines the number of elements in an `ArrayList`. Method `get` returns the element in the location specified by its argument. Method `add` places its argument at the end of the `ArrayList`. Method `remove` deletes its argument from the `ArrayList`.
- 28.16** (*Checkers Game*) In the text, we presented a Tic-Tac-Toe program controlled by a multi-threaded server. Develop a checkers program modeled after the Tic-Tac-Toe program. The two users should alternate making moves. Your program should mediate the players' moves, determining whose turn it is and allowing only valid moves. The players themselves will determine when the game is over.
- 28.17** (*Chess Game*) Develop a chess-playing program modeled after Exercise 28.16.
- 28.18** (*Blackjack Game*) Develop a blackjack card game program in which the server application deals cards to each of the clients. The server should deal additional cards (per the rules of the game) to each player as requested.
- 28.19** (*Poker Game*) Develop a poker game in which the server application deals cards to each client. The server should deal additional cards (per the rules of the game) to each player as requested.
- 28.20** (*Modifications to the Multithreaded Tic-Tac-Toe Program*) The programs in Figs. 28.11 and 28.13 implemented a multithreaded, client/server version of the game of Tic-Tac-Toe. Our goal in developing this game was to demonstrate a multithreaded server that could process multiple connections from clients at the same time. The server in the example is really a mediator between the two clients—it makes sure that each move is valid and that each client moves in the proper order. The server does not determine who won or lost or whether there was a draw. Also, there's no capability to allow a new game to be played or to terminate an existing game.

The following is a list of suggested modifications to Figs. 28.11 and 28.13:

- Modify the `TicTacToeServer` class to test for a win, loss or draw after each move. Send a message to each client that indicates the result of the game when the game is over.
- Modify the `TicTacToeClient` class to display a button that when clicked allows the client to play another game. The button should be enabled only when a game completes. Both class `TicTacToeClient` and class `TicTacToeServer` must be modified to reset the board and all state information. Also, the other `TicTacToeClient` should be notified that a new game is about to begin so that its board and state can be reset.
- Modify the `TicTacToeClient` class to provide a button that allows a client to terminate the program at any time. When the user clicks the button, the server and the other client should be notified. The server should then wait for a connection from another client so that a new game can begin.
- Modify the `TicTacToeClient` class and the `TicTacToeServer` class so that the winner of a game can choose game piece X or O for the next game. Remember: X always goes first.
- If you'd like to be ambitious, allow a client to play against the server while the server waits for a connection from another client.

28.21 (3-D Multithreaded Tic-Tac-Toe) Modify the multithreaded, client/server Tic-Tac-Toe program to implement a three-dimensional 4-by-4-by-4 version of the game. Implement the server application to mediate between the two clients. Display the three-dimensional board as four boards containing four rows and four columns each. If you're ambitious, try the following modifications:

- Draw the board in a three-dimensional manner.
- Allow the server to test for a win, loss or draw. Beware! There are many possible ways to win on a 4-by-4-by-4 board!

28.22 (Networked Morse Code) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (e.g., period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems. Separation between words is indicated by a space or, simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short time during which no sound is transmitted. The international version of the Morse code appears in Fig. 28.16.

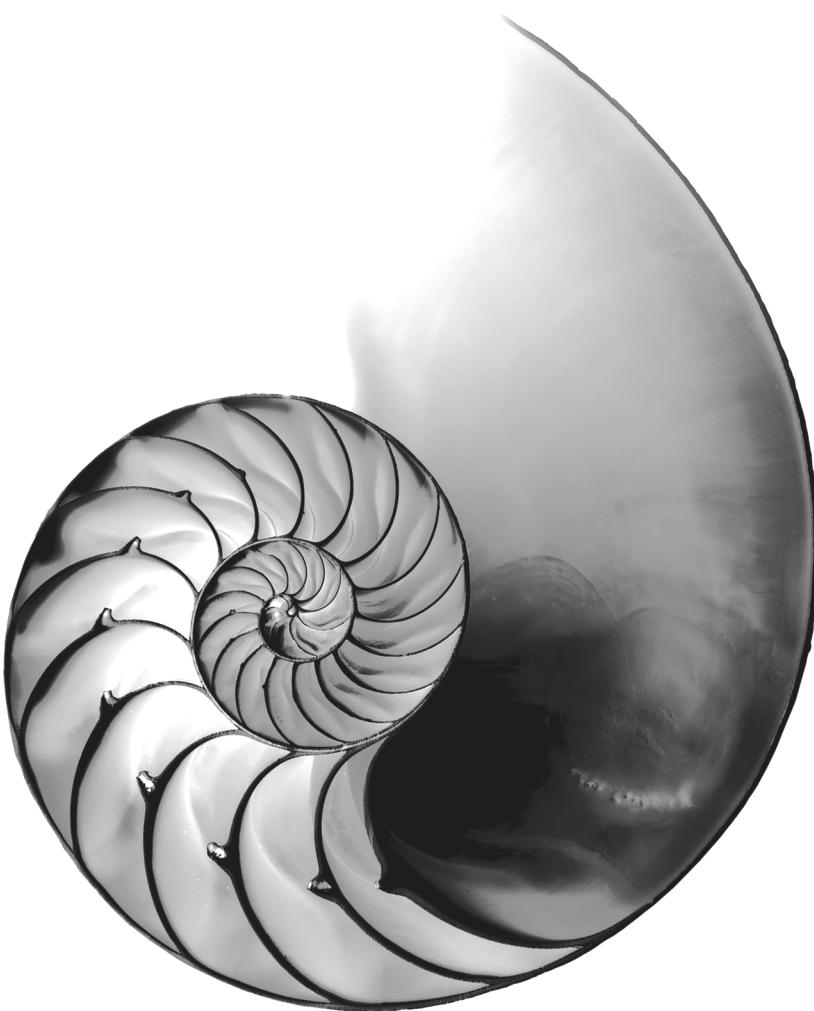
Character	Code	Character	Code	Character	Code	Character	Code
A	..-	J	.---	S	...	1	-----
B	----	K	-.-	T	-	2-
C	---.	L	.---	U	...-	3--
D	--.	M	--	V	...-	4-
E	.	N	-.	W	--	5
F	---.	O	---	X	-..-	6	-....
G	P	.---	Y	-.--	7	-----
H	...-	Q	---.	Z	--..	8	----..
I	..	R	.-.			9	----.
						0	-----

Fig. 28.16 | Letters and digits in international Morse code.

Write a client/server application in which two clients can send Morse-code messages to each other through a multithreaded server application. The client application should allow the user to type English-language phrases in a `JTextArea`. When the user sends the message, the client application encodes the text into Morse code and sends the coded message through the server to the other

client. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word. When messages are received, they should be decoded and displayed as normal characters and as Morse code. The client should have one JTextField for typing and one JTextArea for displaying the other client's messages.

Java Persistence API (JPA)



Objectives

In this chapter you'll:

- Learn the fundamentals of JPA.
- Use classes, interfaces and annotations from the `javax.persistence` package.
- Use the NetBeans IDE's tools to create a Java DB database.
- Use the NetBeans IDE's object-relational-mapping tools to autogenerated JPA entity classes.
- Use autogenerated entity classes to query databases and access data from multiple database tables.
- Use JPA transaction processing capabilities to modify database data.
- Use Java 8 lambdas and streams to manipulate the results of JPA queries.

Outline

29.1	Introduction	29.4.1	Using a Named Query to Get the List of Authors, then Display the Authors with Their ISBNs
29.2	JPA Technology Overview	29.4.2	Using a Named Query to Get the List of Titles, then Display Each with Its Authors
29.2.1	Generated Entity Classes	29.5	Address Book: Using JPA and Transactions to Modify a Database
29.2.2	Relationships Between Tables in the Entity Classes	29.5.1	Transaction Processing
29.2.3	The javax.persistence Package	29.5.2	Creating the AddressBook Database, Project and Persistence Unit
29.3	Querying a Database with JPA	29.5.3	Addresses Entity Class
29.3.1	Creating the Java DB Database	29.5.4	AddressBookController Class
29.3.2	Populating the books Database with Sample Data	29.5.5	Other JPA Operations
29.3.3	Creating the Java Project	29.6	Web Resources
29.3.4	Adding the JPA and Java DB Libraries	29.7	Wrap-Up
29.3.5	Creating the Persistence Unit for the books Database		
29.3.6	Querying the Authors Table		
29.3.7	JPA Features of Autogenerated Class Authors		
29.4	Named Queries; Accessing Data from Multiple Tables		

29.1 Introduction

Chapter 24 used JDBC to connect to relational databases and Structured Query Language (SQL) to query and manipulate relational databases. Recall that we created Strings containing the SQL for every query, insert, update and delete operation. We also created our own classes for managing interactions with databases. If you’re not already familiar with relational databases, SQL and JDBC, you should read Chapter 24 first as this chapter assumes you’re already familiar with the concepts we presented there. This chapter uses the Java EE version of NetBeans 8.2. You can download the current NetBeans versions¹ from

<https://netbeans.org/downloads/>

In this chapter, we introduce the Java Persistence API (JPA). One of JPA’s key capabilities is mapping Java classes to relational database tables and objects of those classes to rows in the tables. This is known as **object-relational mapping**. You’ll use the NetBeans IDE’s object-relational mapping tools to select a database and autogenerate classes that use JPA to interact with that database. Your programs can then use those classes to query the database, insert new records, update existing records and delete records. You will not have to create mappings between your Java code and database tables (as you did with JDBC), and you’ll be able to perform complex database manipulations directly in Java.

Though you’ll manipulate Java DB databases in this chapter, the JPA can be used with any database management system that supports JDBC. At the end of the chapter, we provide links to online JPA resources where you can learn more.

1. As NetBeans and Java EE evolve, the steps in this chapter may change. NetBeans.org provides prior NetBeans versions for download at <http://services.netbeans.org/downloads/dev.php>.

29.2 JPA Technology Overview

When using JPA in this chapter, you'll interact with an existing database via classes that the NetBeans IDE generates from the database's schema. Though we do not do so in this chapter, it's also possible for you to create such classes from scratch and use JPA annotations that enable those classes to create corresponding tables in a database.

29.2.1 Generated Entity Classes

In Section 29.3.5, you'll use the **NetBeans Entity Classes from Database...** option to add to your project classes that represent the database tables. Together, these classes and the corresponding settings are known as a **persistence unit**. The discussion in this section is based on the books database that we introduced in Section 24.3.

For the books database, the NetBeans IDE's object-relational mapping tools create two classes in the data model—**Authors** and **Titles**. Each class—known as an **entity class**—represents the corresponding table in the database and objects of these classes—known as **entities**—represent the rows in the corresponding tables. These classes contain:

- Instance variables representing the table's columns—These are named with all lowercase letters by default and have Java types that are compatible with their database types. Each instance variable is preceded by JPA annotations with information about the corresponding database column, such as whether the instance variable is the table's primary key, whether the column's value in the table is auto-generated, whether the column's value is optional and the column's name.
- Constructors for initializing objects of the class—The resulting entity objects represent rows in the corresponding table. Programs can use entity objects to manipulate the corresponding data in the database.
- *Set* and *get* methods that enable client code to access each instance variable.
- Overridden methods of class **Object**—`hashCode`, `equals` and `toString`.

29.2.2 Relationships Between Tables in the Entity Classes

We did not mention the books database's **AuthorISBN** table. Recall from Section 24.3 that this table links:

- each author in the **Authors** table to that author's books in the **Titles** table, and
- each book in the **Titles** table to the book's authors in the **Authors** table.

This is known as a **join table**, because it's used to join information from multiple other tables. The object-relational mapping tools do *not* create a class for the **AuthorISBN** table. Instead, relationships between tables are taken into account by the generated entity classes:

- The **Authors** class contains the **titlesList** instance variable—a **List** of **Title** objects representing books written by that author.
- The **Titles** class contains the **authorsList** instance variable—a **List** of **Author** objects representing that book's authors.

Like the other instance variables, these **List** variable declarations are preceded by JPA annotations, such as the join table's name, the **Authors** and **AuthorISBN** columns that link

authors to their books, the `Titles` and `AuthorISBN` columns that link titles to their authors, and the type of the relationship. In the book's database there is a *many-to-many relationship*, because each author can write many books and each book can have many authors. We'll show key features of these autogenerated classes later in the chapter. Section 29.4 demonstrates queries that use the relationships among the books database's tables to display joined data.

29.2.3 The `javax.persistence` Package

The package `javax.persistence` contains the JPA interfaces and classes used to interact with the databases in this chapter.

EntityManager Interface

An object that implements the `EntityManager` interface manages the interactions between the program and the database. In Sections 29.3–29.4, you'll use an `EntityManager` to create query objects for obtaining entities from the books database. In Section 29.5, you'll use an `EntityManager` to both query the addressbook database and to create transactions for inserting new entities into the database.

EntityManagerFactory Interface and the Persistence Class

To obtain an `EntityManager` for a given database, you'll use an object that implements the `EntityManagerFactory` interface. As you'll see, the `Persistence` class's `static` method `createEntityManagerFactory` returns an `EntityManagerFactory` for the persistence unit you specify as a `String` argument.

In this chapter, you'll use application-managed `EntityManagers`—that is, ones you obtain from an `EntityManagerFactory` in your app. When you use JPA in Java EE apps, you'll obtain container-managed `EntityManagers` from the Java EE server (i.e., the container) on which your app executes.

TypedQuery Class, Dynamic Queries and Named Queries

An object that implements the `TypedQuery` generic interface performs queries and returns a collection of matching entities—in this chapter, you'll specify that the queries should return `List` objects, though you can choose `Collection`, `List` or `Set` when you generate the entity classes.

To create queries, you'll use `EntityManager` methods. In Section 29.3, you'll create a query with the `EntityManager`'s `createQuery` method. This method's first argument is a `String` written in the `Java Persistence Query Language (JPQL)`—as you'll see, JPQL is similar to SQL (Section 24.4). JPQL queries entity objects, rather than relational database tables. When you define a query in your own code, it's known as a *dynamic query*. In Sections 29.4–29.5, you'll use autogenerated *named queries* that you can access via the `EntityManager` method `createNamedQuery`.

29.3 Querying a Database with JPA

In this section, we demonstrate how to create the books database's JPA entity classes, then use JPA and those classes to *connect* to the books database, *query* it and *display* the results of the query. As you'll see, NetBeans provides tools that simplify accessing data via JPA.

This section's example performs a simple query that retrieves the books database's Authors table. We then use lambdas and streams to display the table's contents. The steps you'll perform are:

- Create a Java DB database and populate it from the `books.sql` file provided with this chapter's examples.
- Create the Java project.
- Add the JPA reference implementation's libraries to the project.
- Add the Java DB library to the project so that the app can access the driver required to connect to the Java DB database over a network—though we'll use the network-capable version of Java DB here, the database will still reside on your local computer.
- Create the persistence unit containing the entity classes for querying the database.
- Create the Java app that uses JPA to obtain the Authors table's data.

29.3.1 Creating the Java DB Database

In this section, you'll use the SQL script (`books.sql`) provided with this chapter's examples to create the books database in NetBeans. Chapter 24 demonstrated several database apps that used the embedded version of Java DB. This chapter's examples use the network server version.

Creating the Database

Perform the following steps to create the books database:

1. In the upper-left corner of the NetBeans IDE, click the **Services** tab. (If the Services tab is not displayed, select **Services** from the **Window** menu.)
2. Expand the **Databases** node then right click **Java DB**. If **Java DB** is not already running the **Start Server** option will be enabled. In this case, Select **Start Server** to launch the Java DB server. You may need to wait a moment for the server to begin executing.²
3. Right click the **Java DB** node, then select **Create Database....**
4. In the **Create Java DB Database** dialog, set **Database Name** to `books`, **User Name** to `deitel`, and **Password** and **Confirm Password** to `deitel`.³
5. Click **OK**.

The preceding steps create the database using Java DB's *server version* that can receive database connections over a network. A new node named

```
jdbc:derby://localhost:1527/books
```

-
2. If the **Start Server** option is disabled, select **Properties...** and ensure that the **Java DB Installation** option is set to the JDK's `db` folder location.
 3. We used `deitel` as the user name and password for simplicity—ensure that you use secure passwords in real applications.

appears in the **Services** tab's **Database** node. This is the JDBC URL that's used to connect to the database.

29.3.2 Populating the books Database with Sample Data

You'll now populate the database with sample data using the `books.sql` script that's provided with this chapter's examples. To do so, perform the following steps:

1. Select **File > Open File...** to display the **Open** dialog.
2. Navigate to this chapter's examples folder, select `books.sql` and click **Open**.
3. In NetBeans, right click in the SQL script and select **Run File**.
4. In the **Select Database Connection** dialog, select the JDBC URL for the database you created in Section 29.3.1 and click **OK**.

The IDE will connect to the database and run the SQL script to populate the database. The SQL script attempts to remove the database's tables if they already exist. If they do not, you'll receive error messages when the three `DROP TABLE` commands in the SQL script execute, but the tables will still be created properly.

You can confirm that the database was populated properly by viewing each table's data in NetBeans. To do so:

1. In the NetBeans **Services** tab, expand the **Databases** node, then expand the node `jdbc:derby://localhost:1527/books`.
2. Expand the **DETEL** node, then the **Tables** node.
3. Right click one of the tables and select **View Data....**

The `books` database is now set up and ready for connections.

29.3.3 Creating the Java Project

For the examples in this section and Section 29.4, we'll create one project that contains the `books` database's JPA entity classes and two Java apps that use them. To create the project:

1. In the upper-left corner of NetBeans, select the **Projects** tab.
2. Select **File > New Project....**
3. In the **New Project** dialog, select the **Java** category, then **Java Application** and click **Next >**.
4. For the **Project Name**, specify `BooksDatabaseExamples`, then choose where you wish to store the project on your computer.
5. Ensure that the **Create Main Class** option is checked. By default, NetBeans uses the project name as the class name and puts the class in a package named `books-databaseexamples` (the project name in all lowercase letters). We changed the class name for this first example to `DisplayAuthors`. Also, to indicate that the classes in this package are from this book's JPA chapter, we replaced the package name with

```
com.deitel.jhttp.jpa
```

6. Click **Finish** to create the project.

29.3.4 Adding the JPA and Java DB Libraries

For certain types of projects (such as server-side Java EE applications), NetBeans automatically includes JPA support, but not for simple **Java Application** projects. In addition, NetBeans projects do not include database drivers by default. In this section, you'll add the JPA libraries and Java DB driver library to the project so that you can use JPA's features to interact with the Java DB database you created in Sections 29.3.1–29.3.2.

EclipseLink—The JPA Reference Implementation

Each Java Enterprise Edition (Java EE) API—such as JPA—has a *reference implementation* that you can use to experiment with the API's features and implement applications. The JPA reference implementation—which is included with the NetBeans Java EE version—is **EclipseLink** (<http://www.eclipse.org/eclipselink>).

Adding Libraries

To add JPA and Java DB support to your project:

1. In the NetBeans Projects tab, expand the **BooksDatabaseExamples** node.
2. Right click the project's **Libraries** node and select **Add Library....**
3. In the **Add Library** dialog, hold the *Ctrl* key—*command* (⌘) in OS X—and select **EclipseLink (JPA 2.1)**, **Java DB Driver** and **Persistence (JPA 2.1)**, then click **Add Library**.

29.3.5 Creating the Persistence Unit for the books Database

In this section, you'll create the persistence unit containing the entity classes **Authors** and **Titles** using the NetBeans object-relational mapping tools. To do so:

1. In the NetBeans Projects tab, right click the **BooksDatabaseExamples** node, then select **New > Entity Classes from Database....**
2. In the **New Entity Classes from Database** dialog's **Database Tables** step, select the books database's URL from the **Database Connection** drop-down list. Then, click the **Add All >>** button and click **Next >**.
3. The **Entity Classes** step enables you to customize the entity class names and the package. Keep the default names, ensure that **Generate Named Query Annotations for Persistent Fields**, **Generate JAXB Annotations** and **Create Persistence Unit** are checked then click **Next >**.
4. In the **Mapping Options** step, change the **Collection Type** to **java.util.List** and keep the other default settings—for queries that return multiple authors or titles, the results will be placed in **List** objects.
5. Click **Finish**.

The IDE creates the persistence unit containing the **Authors** and **Titles** classes and adds their source-code files **Authors.java** and **Titles.java** to the project's package node (**com.deitel.jhttp.jpa**) in the **Source Packages** folder. As part of the persistence unit, the IDE also creates a **META-INF** package in the **Source Packages** folder. This contains the **persistence.xml** file, which specifies persistence unit settings. These include the books database's JDBC URL and the persistence unit's name, which you'll use to obtain an

`EntityManager` to manage the books database interactions. By default, the persistence unit's name is the project name followed by PU—`BooksDatabaseExamplesPU`. It's also possible to have multiple persistence units, but that's beyond this chapter's scope.

29.3.6 Querying the Authors Table

Figure 29.1 performs a simple books database query that retrieves the `Authors` table and displays its data. The program illustrates using JPA to connect to the database and query it. You'll use a dynamic query created in `main` to get the data from the database—in the next example, you'll use auto-generated queries in the persistence unit to perform the same query and others. In Section 29.5, you'll learn how to modify a database through a JPA persistence unit. *Reminder:* Before you run this example, ensure that the Java DB database server is running; otherwise, you'll get runtime exceptions indicating that the app cannot connect to the database server. For details on starting the Java DB server, see Section 29.3.1.

```

1 // Fig. 29.1: DisplayAuthors.java
2 // Displaying the contents of the authors table.
3 package com.deitel.jhttp.jpa;
4
5 import javax.persistence.EntityManager;
6 import javax.persistence.EntityManagerFactory;
7 import javax.persistence.Persistence;
8 import javax.persistence.TypedQuery;
9
10 public class DisplayAuthors
11 {
12     public static void main(String[] args)
13     {
14         // create an EntityManagerFactory for the persistence unit
15         EntityManagerFactory entityManagerFactory =
16             Persistence.createEntityManagerFactory(
17                 "BooksDatabaseExamplesPU");
18
19         // create an EntityManager for interacting with the persistence unit
20         EntityManager entityManager =
21             entityManagerFactory.createEntityManager();
22
23         // create a dynamic TypedQuery<Authors> that selects all authors
24         TypedQuery<Authors> findAllAuthors = entityManager.createQuery(
25             "SELECT author FROM Authors AS author", Authors.class);
26
27         // display List of Authors
28         System.out.printf("Authors Table of Books Database:%n%n");
29         System.out.printf("%-12s%-13s%sn",
30             "Author ID", "First Name", "Last Name");
31
32         // get all authors, create a stream and display each author
33         findAllAuthors.getResultList().stream()
34             .forEach((author) ->
35             {

```

Fig. 29.1 | Displaying contents of the authors table. (Part 1 of 2.)

```
36         System.out.printf("%-12d%-13s%s%n", author.getAuthorid(),
37                           author.getFirstname(), author.getLastname());
38     }
39 }
40 }
41 }
```

Authors Table of Books Database:

Author ID	First Name	Last Name
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Fig. 29.1 | Displaying contents of the authors table. (Part 2 of 2.)

Importing the JPA Interfaces and Class Used in This Example

Lines 5–8 import the JPA interfaces and class from package javax.persistence used in this program:

- EntityManager interface—An object of this type manages the data flow between the program and the database.
- EntityManagerFactory interface—An object of this type creates the persistence unit’s EntityManager.
- Persistence class—A static method of this class creates the specified persistence unit’s EntityManagerFactory.
- TypedQuery interface—The EntityManager returns an object of this type when you create a query. You then execute the query to get data from the database.

Creating the EntityManagerFactory Object

Lines 15–17 create the persistence unit’s EntityManagerFactory object. The Persistence class’s static method `createEntityManagerFactory` receives the persistence unit’s name—BooksDatabaseExamplesPU. In Section 29.3.5, NetBeans created this name in `persistence.xml`, based on the project’s name.

Creating the EntityManager

Lines 20–21 use the EntityManagerFactory’s `createEntityManager` method to create an application-managed EntityManager that handles the interactions between the app and the database. These include querying the database, storing new entities into the database, updating existing entries in the database and removing entities from the database. You’ll use the EntityManager in this example to create a query.

Creating a TypedQuery That Retrieves the Authors Table

Lines 24–25 use EntityManager’s `createQuery` method to create a TypedQuery that returns all of the Authors entities in the Authors table—each Authors entity represents one row in the table. The first argument to `createQuery` is a String written in the Java Per-

sistence Query Language (JPQL). The second argument specifies a `Class` object representing the type of objects the query returns—`Authors.class` is shorthand notation for creating a `Class` object representing `Authors`. Recall that when creating the entity classes, we specified that query results should be returned as `Lists`. When this query executes, it returns a `List<Authors>` that you can then use in your code to manipulate the `Authors` table. You can learn more about JPQL in the Java EE 7 tutorial at:

<https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm>

Displaying the Query Results

Lines 33–39 execute the query and use lambdas and streams to display each `Authors` object. To perform the query created in lines 24–25, line 33 calls its `getResultsList` method, which returns a `List<Authors>`. Next, we create a `Stream` from that `List` and invoke the `Stream`'s `forEach` method to display each `Authors` object in the `List`. The lambda expression passed to `forEach` uses the `Authors` class's autogenerated `get` methods to obtain the author ID, first name and last name from each `Authors` object.

29.3.7 JPA Features of Autogenerated Class Authors

In this section, we overview various JPA annotations that were inserted into the autogenerated entity class `Authors`. Class `Titles` contains similar annotations. You can see the complete list of JPA annotations and their full descriptions at:

<http://docs.oracle.com/javaee/7/api/index.html?javax/persistence/package-summary.html>

JPA Annotations for Class Authors

If you look through the source code for autogenerated class `Authors` (or class `Titles`), you'll notice that the class does not contain any code that interacts with a database. Instead, you'll see various JPA annotations that the NetBeans IDE's object-relational-mapping tools autogenerated. When you compile the entity classes, the compiler looks at the annotations and adds JPA capabilities that help manage the interactions with the database—this is known as *injecting* capabilities. For the entity classes, the annotations include:

- **@Entity**—Specifies that the class is an entity class.
- **@Table**—Specifies the entity class's corresponding database table.
- **@NamedQueries/@NamedQuery**—An `@NamedQueries` annotation specifies a collection of `@NamedQuery` annotations that declare various named queries. You can define your own `@NamedQuery` annotations in addition to the ones that the object-relational-mapping tools can autogenerated.

JPA Annotations for Class Authors' Instance Variables

JPA annotations also specify information about an entity class's instance variables:

- **@Id**—Used to indicate the instance variable that corresponds to the database table's primary key. For composite primary keys, multiple instance variables would be annotated with `@Id`.
- **@GeneratedValue**—Indicates that the column value in the database is autogenerated.

- **@Basic**—Specifies whether the column is optional and whether the corresponding data should load *lazily* (i.e., only when the data is accessed through the entity object) or *eagerly* (i.e., loaded immediately when the entity object is created).
- **@Column**—Specifies the database column to which the instance variable corresponds.
- **@JoinTable/@JoinColumn**—These specify relationships between tables. In the Authors class, this helps JPA determine how to populate an Authors entity's titlesList.
- **@ManyToMany**—Specifies the relationship between entities. For the Authors and Titles entity classes, there is a many-to-many relationship—each author can write many books and each book can have many authors. There are also annotations for **@ManyToOne**, **@OneToMany** and **@OneToOne** relationships.

29.4 Named Queries; Accessing Data from Multiple Tables

The next example demonstrates two named queries that were autogenerated when you created the books database's persistence unit in Section 29.3.5. For discussion purposes we split the program into Figs. 29.2 and 29.3, each showing the corresponding portion of the program's output. Once again, we use lambdas and streams capabilities to display the results. As you'll see, we use the relationships between the Authors and Titles entities to display information from both database tables.

29.4.1 Using a Named Query to Get the List of Authors, then Display the Authors with Their Titles

Figure 29.2 uses the techniques you learned in Section 29.3 to display each author followed by that author's list of titles. To add `DisplayQueryResults.java` to your project:

1. Right click the project's name in the NetBeans **Projects** tab and select **New > Java Class....**
2. In the **New Java Class** dialog, enter `DisplayQueryResults` for the **Class Name**, select `com.deitel.jhttp.jpa` as the **Package** and click **Finish**.

The IDE opens the new file and you can now enter the code in Figs. 29.2 and 29.3. To run this file, right click its name in the project, then select **Run File**. You can also right click the project and select **Properties** then set this class as the **Main Class** in the project's **Run** settings. Then, when you run the project, this file's `main` method will execute.

```
1 // Fig. 29.2: DisplayQueryResults.java
2 // Display the results of various queries.
3
4 package com.deitel.jhttp.jpa;
5
```

Fig. 29.2 | Using a `NamedQuery` to get the list of `Authors`, then display the `Authors` with their titles. (Part 1 of 3.)

```
6 import java.util.Comparator;
7 import javax.persistence.EntityManager;
8 import javax.persistence.EntityManagerFactory;
9 import javax.persistence.Persistence;
10 import javax.persistence.TypedQuery;
11
12 public class DisplayQueryResults
13 {
14     public static void main(String[] args)
15     {
16         // create an EntityManagerFactory for the persistence unit
17         EntityManagerFactory entityManagerFactory =
18             Persistence.createEntityManagerFactory(
19                 "BooksDatabaseExamplesPU");
20
21         // create an EntityManager for interacting with the persistence unit
22         EntityManager entityManager =
23             entityManagerFactory.createEntityManager();
24
25         // TypedQuery that returns all authors
26         TypedQuery<Authors> findAllAuthors =
27             entityManager.createNamedQuery("Authors.findAll", Authors.class);
28
29         // display titles grouped by author
30         System.out.printf("Titles grouped by author:%n");
31
32         // get the List of Authors then display the results
33         findAllAuthors.getResultList().stream()
34             .sorted(Comparator.comparing(Authors::getLastname)
35                 .thenComparing(Authors::getFirstname))
36             .forEach((author) ->
37             {
38                 System.out.printf("%n%s %s:%n",
39                     author.getFirstname(), author.getLastname());
40
41                 for (Titles title : author.getTitlesList())
42                 {
43                     System.out.printf("\t%s%n", title.getTitle());
44                 }
45             });
46     };
47 
```

Titles grouped by author:

Abbey Deitel:

Internet & World Wide Web How to Program
Simply Visual Basic 2010
Visual Basic 2012 How to Program
Android How to Program
Android for Programmers: An App-Driven Approach, 2/e, Volume 1
Android for Programmers: An App-Driven Approach

Fig. 29.2 | Using a NamedQuery to get the list of Authors, then display the Authors with their titles. (Part 2 of 3.)

Harvey Deitel:

Internet & World Wide Web How to Program
Java How to Program
Java How to Program, Late Objects Version
C How to Program
Simply Visual Basic 2010
Visual Basic 2012 How to Program
Visual C# 2012 How to Program
Visual C++ How to Program
C++ How to Program
Android How to Program
Android for Programmers: An App-Driven Approach, 2/e, Volume 1
Android for Programmers: An App-Driven Approach

Paul Deitel:

Internet & World Wide Web How to Program
Java How to Program
Java How to Program, Late Objects Version
C How to Program
Simply Visual Basic 2010
Visual Basic 2012 How to Program
Visual C# 2012 How to Program
Visual C++ How to Program
C++ How to Program
Android How to Program
Android for Programmers: An App-Driven Approach, 2/e, Volume 1
Android for Programmers: An App-Driven Approach

Michael Morgano:

Android for Programmers: An App-Driven Approach

Dan Quirk:

Visual C++ How to Program

Fig. 29.2 | Using a `NamedQuery` to get the list of `Authors`, then display the `Authors` with their titles. (Part 3 of 3.)

Creating a `TypedQuery` That Retrieves the `Authors` Table

One of the default options when you created the books database's persistence unit was **Generate Named Query Annotations for Persistent Fields**—you can view these named queries before the class definitions in `Authors.java` and `Titles.java`. For class `Authors`, the object-relational mapping tool autogenerated the following queries:

- "`Authors.findAll`"—Returns the `List` of all `Authors` entities.
- "`Authors.findByAuthorid`"—Returns the `Authors` entity with the specified `authorid` value.
- "`Authors.findByFirstname`"—Returns the `List` of all `Authors` entities with the specified `firstname` value.
- "`Authors.findByLastname`"—Returns the `List` of all `Authors` entities with the specified `lastname` value.

You'll see how to provide arguments to queries in Section 29.5. Like the dynamic query you defined in Fig. 29.1, each of these queries is defined using the Java Persistence Query Language (JPQL).

Lines 17–23 get the EntityManager for this program, just as we did in Fig. 29.1. Lines 26–27 use EntityManager's createNamedQuery method to create a TypedQuery that returns the result of the "Authors.findAll" query. The first argument is a String containing the query's name and the second is the Class object representing the entity type that the query returns.

Processing the Results

Lines 33–46 execute the query and use Java 8 lambdas and streams to display each Authors entity's name followed by the list of that author's titles. Line 33 calls the TypedQuery's getResultsList method to perform the query. We create a Stream that sorts the Authors entities by last name then first name. Next, we invoke the Stream's forEach method to display each Authors entity's name and list of titles. The lambda expression passed to forEach uses the Authors class's autogenerated get methods to obtain the first name and last name from each Authors entity. Line 41 calls the autogenerated Authors method getTitlesList to get the current author's List<Titles>, then lines 41–44 display the String returned by each Titles entity's autogenerated getTitle method.

29.4.2 Using a Named Query to Get the List of Titles, then Display Each with Its Authors

In Fig. 29.3, lines 49–50 use EntityManager method createNamedQuery to create a TypedQuery that returns the result of the "Titles.findAll" query. Then, lines 56–68 display each title followed by that title's list of author names. Line 56 calls the TypedQuery's getResultsList method to perform the query. We create a Stream that sorts the Titles entities by title. Next, we invoke the Stream's forEach method to display each Titles entity's title and the corresponding list of authors. Once again, the lambda expression uses the autogenerated Titles and Authors methods to access the entity data that's displayed.

```

48     // TypedQuery that returns all titles
49     TypedQuery<Titles> findAllTitles =
50         entityManager.createNamedQuery("Titles.findAll", Titles.class);
51
52     // display titles grouped by author
53     System.out.printf("%nAuthors grouped by title:%n%n");
54
55     // get the List of Titles then display the results
56     findAllTitles.getResultList().stream()
57         .sorted(Comparator.comparing(Titles::getTitle))
58         .forEach((title) ->
59             {
60                 System.out.println(title.getTitle());
61

```

Fig. 29.3 | Using a NamedQuery to get the list of Titles, then display each with its Authors.
(Part 1 of 3.)

```
62             for (Authors author : title.getAuthorsList())
63         {
64             System.out.printf("\t%s %s%n",
65                             author.getFirstname(), author.getLastname());
66         }
67     }
68 }
69 }
70 }
```

Authors grouped by title:

```
Android How to Program
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
Android for Programmers: An App-Driven Approach
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
    Michael Morgano
Android for Programmers: An App-Driven Approach, 2/e, Volume 1
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
C How to Program
    Paul Deitel
    Harvey Deitel
C++ How to Program
    Paul Deitel
    Harvey Deitel
Internet & World Wide Web How to Program
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
Java How to Program
    Paul Deitel
    Harvey Deitel
Java How to Program, Late Objects Version
    Paul Deitel
    Harvey Deitel
Simply Visual Basic 2010
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
Visual Basic 2012 How to Program
    Paul Deitel
    Harvey Deitel
    Abbey Deitel
Visual C# 2012 How to Program
    Paul Deitel
    Harvey Deitel
```

Fig. 29.3 | Using a NamedQuery to get the list of Titles, then display each with its Authors.
(Part 2 of 3.)

Visual C++ How to Program
 Paul Deitel
 Harvey Deitel
 Dan Quirk

Fig. 29.3 | Using a NamedQuery to get the list of Titles, then display each with its Authors.
 (Part 3 of 3.)

29.5 Address Book: Using JPA and Transactions to Modify a Database

We now reimplement the address book app from Section 24.9 using JPA. As before, you can browse existing entries, add new entries and search for entries with a specific last name. Recall that the AddressBook Java DB database contains an Addresses table with the columns addressID, FirstName, LastName, Email and PhoneNumber. The column addressID is an identity column in the Addresses table.

29.5.1 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine whether the transaction is successful. You begin by specifying the source account and the amount you wish to transfer to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank *did not* deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to treat a set of operations as a *single* operation, known as an **atomic operation** or a **transaction**. At the end of a transaction, a decision can be made either to **commit the transaction** or **roll back the transaction**:

- Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation.
- Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

JPA provides transaction processing via methods of interfaces EntityManager and EntityTransaction. EntityManager method **getTransaction** returns an EntityTransaction for managing a transaction. EntityTransaction method **begin** starts a transaction. Next, you perform your database's operations using the EntityManager. If the

operations execute successfully, you call `EntityTransaction` method `commit` to commit the changes to the database. If any operation fails, you call `EntityTransaction` method `rollback` to return the database to its state prior to the transaction. You'll use these techniques in Section 29.5.4. (In a Java EE project, the server can perform these tasks for you.)

29.5.2 Creating the AddressBook Database, Project and Persistence Unit

Use the techniques you learned in Sections 29.3.1–29.3.5 to perform the following steps:

Step 1: Creating the addressbook Database

Using the steps presented in Section 29.3.1, create the addressbook database.

Step 2: Populating the Database

Using the steps presented in Section 29.3.2, populate the addressbook database with the sample data in the `addressbook.sql` file that's provided with this chapter's examples.

Step 3: Creating the AddressBook Project

This app has a JavaFX GUI. For prior JavaFX apps, we created an FXML file that described the app's GUI, a subclass of `Application` that launched the app and a controller class that handled the app's GUI events and provided other app logic. NetBeans provides a **JavaFX FXML Application** project template that creates the FXML file and Java source-code files for the `Application` subclass and controller class. To use this template:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Under **Categories**: select **JavaFX** and under **Projects**: select **JavaFX FXML Application**, then click **Next >**.
3. For the **Project Name** specify **AddressBook**.
4. For the **FXML name**, specify **AddressBook**.
5. In the **Create Application Class** textfield, replace the default package name and class name with `com.deitel.jhttp.jpa.AddressBook`.
6. Click **Finish** to create the project.

NetBeans places in the app's package the files `AddressBook.fxml`, `AddressBook.java` and `AddressBookController.java`. If you double-click the FXML file in NetBeans, it will automatically open in Scene Builder (if you have it installed) so that you can design your GUI.

For this app, rather than recreating `AddressBook` GUI, we replaced the default FXML that NetBeans generated in `AddressBook.fxml` with the contents of `AddressBook.fxml` from Section 24.9's example (right click the FXML file in NetBeans and select **Edit** to view its source code). We then changed the controller class's name from `AddressBookController` to

```
com.deitel.jhttp.jpa.AddressBookController
```

because the controller class in this example is in the package `com.deitel.jhttp.jpa`.

Also, in the autogenerated `AddressBook` subclass of `Application` (located in `AddressBook.java`), we added the following statement to set the stage's title bar `String`:

```
stage.setTitle("Address Book");
```

Step 4: Adding the JPA and Java DB Libraries

Using the steps presented in Section 29.3.4, add the required JPA and JavaDB libraries to the project's **Libraries** folder.

Step 5: Creating the AddressBook Database's Persistence Unit

Using the steps presented in Section 29.3.5, create the AddressBook database's persistence unit, which will be named **AddressBookPU** by default.

29.5.3 Addresses Entity Class

When you created the AddressBook database's persistence unit, NetBeans autogenerated the **Addresses** entity class (in **Addresses.java**) with several named queries. In this app, you'll use the queries:

- "Addresses.findAll"—Returns a `List<Addresses>` containing **Addresses** entities for all the contacts.
- "Addresses.findByLastname"—Returns a `List<Addresses>` containing an **Addresses** entity for each contact with the specified last name.

Ordering the Named Query Results

By default, the JPQL for the autogenerated named queries does not order the query results. In Section 24.9, we used the SQL's ORDER BY clause to arrange query results into ascending order by last name then first name. JPQL also has an ORDER BY clause. To order the query results in this app, we opened **Addresses.java** and added

```
ORDER BY a.lastname, a.firstname
```

to the query strings for the "Addresses.findAll" and "Addresses.findByLastname" named queries—again these are specified in the `@NamedQuery` annotations just before the **Addresses** class's declaration.

ToString Method of Class Addresses

In this app, we use the `List<Addresses>` returned by each query to populate an `ObservableList` that's bound to the app's `ListView`. Recall that, by default, a `ListView`'s cells display the `String` representation of the `ObservableList`'s elements. To ensure that each **Addresses** object in the `ListView` is displayed in the format *Last Name, First Name*, we modified the **Addresses** class's autogenerated `toString` method. To do so, open **Addresses.java** and replace its `return` statement with

```
return getLastname() + ", " + getFirstname();
```

29.5.4 AddressBookController Class

The **AddressBookController** class (Fig. 29.4) uses the persistence unit you created in Section 29.5.2 to interact with `addressbook` database. Much of the code in Fig. 29.4 is identical to the code in Fig. 24.34. For the discussion in this section, we focus on the highlighted JPA features.

```
1 // Fig. 29.4: AddressBookController.java
2 // Controller for a simple address book
3 package com.deitel.jhttp.jpa;
4
5 import java.util.List;
6 import javafx.collections.FXCollections;
7 import javafx.collections.ObservableList;
8 import javafx.event.ActionEvent;
9 import javafx.fxml.FXML;
10 import javafx.scene.control.Alert;
11 import javafx.scene.control.Alert.AlertType;
12 import javafx.scene.control.ListView;
13 import javafx.scene.control.TextField;
14 import javax.persistence.EntityManager;
15 import javax.persistence.EntityManagerFactory;
16 import javax.persistence.EntityTransaction;
17 import javax.persistence.Persistence;
18 import javax.persistence.TypedQuery;
19
20 public class AddressBookController {
21     @FXML private ListView<Addresses> listView;
22     @FXML private TextField firstNameTextField;
23     @FXML private TextField lastNameTextField;
24     @FXML private TextField emailTextField;
25     @FXML private TextField phoneTextField;
26     @FXML private TextField findByLastNameTextField;
27
28     // create an EntityManagerFactory for the persistence unit
29     private final EntityManagerFactory entityManagerFactory =
30         Persistence.createEntityManagerFactory("AddressBookPU");
31
32     // create an EntityManager for interacting with the persistence unit
33     private final EntityManager entityManager =
34         entityManagerFactory.createEntityManager();
35
36     // stores list of Addresses objects that results from a database query
37     private final ObservableList<Addresses> contactList =
38         FXCollections.observableArrayList();
39
40     // populate ListView and set up listener for selection events
41     public void initialize() {
42         listView.setItems(contactList); // bind to contactList
43
44         // when ListView selection changes, display selected person's data
45         listView.getSelectionModel().selectedItemProperty().addListener(
46             (observableValue, oldValue, newValue) -> {
47                 displayContact(newValue);
48             }
49         );
50         getAllEntries(); // populates contactList, which updates ListView
51     }
52 }
```

Fig. 29.4 | A simple address book. (Part I of 5.)

```
53     // get all the entries from the database to populate contactList
54     private void getAllEntries() {
55         // query that returns all contacts
56         TypedQuery<Addresses> findAllAddresses =
57             entityManager.createNamedQuery(
58                 "Addresses.findAll", Addresses.class);
59
60         contactList.setAll(findAllAddresses.getResultList());
61         selectFirstEntry();
62     }
63
64     // select first item in listView
65     private void selectFirstEntry() {
66         listView.getSelectionModel().selectFirst();
67     }
68
69     // display contact information
70     private void displayContact(Addresses contact) {
71         if (contact != null) {
72             firstNameTextField.setText(contact.getFirstname());
73             lastNameTextField.setText(contact.getLastname());
74             emailTextField.setText(contact.getEmail());
75             phoneTextField.setText(contact.getPhonenumber());
76         }
77         else {
78             firstNameTextField.clear();
79             lastNameTextField.clear();
80             emailTextField.clear();
81             phoneTextField.clear();
82         }
83     }
84
85     // add a new entry
86     @FXML
87     void addEntryButtonPressed(ActionEvent event) {
88         Addresses address = new Addresses();
89         address.setFirstname(firstNameTextField.getText());
90         address.setLastname(lastNameTextField.getText());
91         address.setPhonenumber(phoneTextField.getText());
92         address.setEmail(emailTextField.getText());
93
94         // get an EntityTransaction to manage insert operation
95         EntityTransaction transaction = entityManager.getTransaction();
96
97         try
98         {
99             transaction.begin(); // start transaction
100            entityManager.persist(address); // store new entry
101            transaction.commit(); // commit changes to the database
102            displayAlert(AlertType.INFORMATION, "Entry Added",
103                         "New entry successfully added.");
104        }

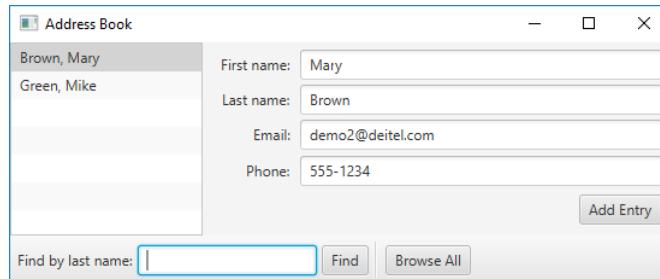
```

Fig. 29.4 | A simple address book. (Part 2 of 5.)

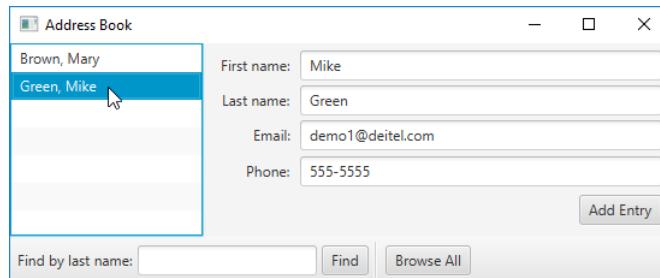
```
105     catch (Exception e) // if transaction failed
106     {
107         transaction.rollback(); // undo database operations
108         displayAlert(AlertType.ERROR, "Entry Not Added",
109                     "Unable to add entry: " + e);
110     }
111
112     getAllEntries();
113 }
114
115 // find entries with the specified last name
116 @FXML
117 void findButtonPressed(ActionEvent event) {
118     // query that returns all contacts
119     TypedQuery<Addresses> findByLastname =
120         entityManager.createNamedQuery(
121             "Addresses.findByLastname", Addresses.class);
122
123     // configure parameter for query
124     findByLastname.setParameter(
125         "lastname", findByLastNameTextField.getText() + "%");
126
127     // get all addresses
128     List<Addresses> people = findByLastname.getResultList();
129
130     if (people.size() > 0) { // display all entries
131         contactList.setAll(people);
132         selectFirstEntry();
133     }
134     else {
135         displayAlert(AlertType.INFORMATION, "Lastname Not Found",
136                     "There are no entries with the specified last name.");
137     }
138 }
139
140 // browse all the entries
141 @FXML
142 void browseAllButtonPressed(ActionEvent event) {
143     getAllEntries();
144 }
145
146 // display an Alert dialog
147 private void displayAlert(
148     AlertType type, String title, String message) {
149     Alert alert = new Alert(type);
150     alert.setTitle(title);
151     alert.setContentText(message);
152     alert.showAndWait();
153 }
154 }
```

Fig. 29.4 | A simple address book. (Part 3 of 5.)

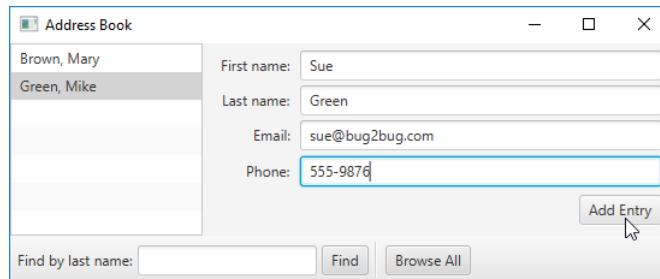
a) Initial Address Book screen showing entries.



b) Viewing the entry for Green, Mike.



c) Adding a new entry for Sue Green.



d) Searching for last names that start with Gr.

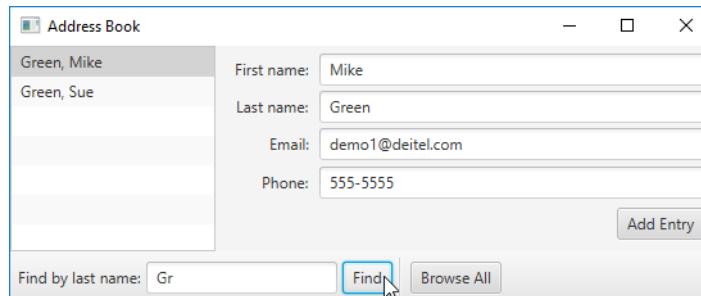


Fig. 29.4 | A simple address book. (Part 4 of 5.)

- e) Returning to the complete list by clicking **Browse All**.

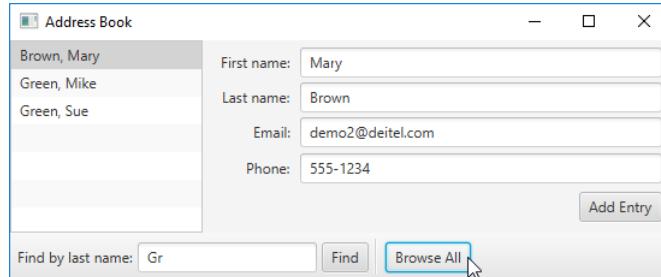


Fig. 29.4 | A simple address book. (Part 5 of 5.)

Obtaining the EntityManager

Lines 29–34 use the techniques you learned in Section 29.3.6 to obtain an `EntityManagerFactory` for the `AddressBook` persistence unit ("AddressBookPU"), then use it to get the `EntityManager` for interacting with the `addressbook` database. Lines 37–38 define an `ObservableList<Addresses>` named `contactList` that's used to bind the app's query results to the `ListView` (line 42 of method `initialize`).

Obtaining the Complete List of Contacts—Method `getAllEntries`

Lines 56–58 in method `getAllEntries` create a `TypedQuery` for the named query "`Addresses.findAll`", which returns a `List<Addresses>` containing all the `Addresses` entities in the database. Line 60 calls the `TypedQuery`'s `getResultList` method and uses the resulting `List<Addresses>` to populate the `contactList`, which was previously bound to the `ListView`. Each time the complete contacts list is loaded, line 61 calls method `selectFirstEntry` to display the first `Addresses` entity's details. Due to the listener registered in lines 45–49, this in turn calls method `displayContact` to display the selected `Addresses` entity if there is one; otherwise, `displayContact` clears the `TextFields` that display a contact's details.

Adding an Entry to the Database—Method `addEntryButtonPressed`

When you enter new data in this app's GUI, then click the `Add Entry` button—a new row should be added to the `Addresses` table in the database. To create a new entity in the database, you must first create an instance of the entity class (line 88) and set its instance variables (lines 89–92), then use a transaction to insert the data in the database (lines 95–110). Notice that we do not specify a value for the `Addresses` entity's `addressid` instance variable—this value is autogenerated by the database when you add a new entry.

Lines 95–110 use the techniques discussed in Section 29.5.1 to perform the insert operation. Line 95 uses `EntityManager` method `getTransaction` to get the `EntityTransaction` used to manage the transaction. In the `try` block, line 99 uses `EntityTransaction` method `begin` to start the transaction. Next, line 100 calls `EntityManager` method `persist` to insert the new entity into the database. If this operation executes successfully, line 101 calls `EntityTransaction` method `commit` to complete the transaction and commit the changes to the database. If the `persist` operation fails, line 107 in the `catch`

block calls `EntityTransaction` method `rollback` to return the database to its state prior to the transaction.⁴

Finding by Last Name—Method `findButtonPressed`

Lines 119–121 in method `findButtonPressed` create a `TypedQuery` for the named query "`Addresses.findByLastname`", which returns a `List<Addresses>` containing all the entities with the specified last name. If you open the autogenerated `Addresses` class in your project, you'll see that the query requires a parameter, as specified in the following JPQL that we copied from the `Addresses.java` file:

```
SELECT a FROM Addresses a WHERE a.lastname = :lastname
```

The notation `:lastname` represents a parameter named `lastname`. The autogenerated query locates only exact matches, as indicated by the JPQL equals (`=`) operator. For this app, we changed `=` to the JPQL `LIKE` operator so we can locate last names that begin with the letters typed by the user in the `findByLastNameTextField`.

Before executing the query, you set arguments for each query parameter by calling `TypedQuery` method `setParameter` (lines 124–125) with the JPQL parameter name as the first argument and the corresponding value as the second argument. As in SQL, line 125 appends `%` to the contents of `findByLastNameTextField` to indicate that we're searching for last names that begin with the user's input, possibly followed by more characters.

When you execute the query (line 128), it returns a `List` containing any matching entities in database. If the number of results is greater than 0, lines 131–132 display the search results in the `ListView` and select the first matching result to display its details. Otherwise, 135–136 display an `Alert` dialog indicating there were no entries with the specified last name.

29.5.5 Other JPA Operations

Though we did not do so in this example, you also can update an existing entity in the database or delete an existing entity from the database.

Updating an Existing Entity

You update an existing entity by modifying its entity object in the context of a transaction. Once you commit the transaction, the changes to the entity are saved to the database.

Deleting an Existing Entity

To remove an entity from the database, call `EntityManager` method `remove` in the context of a transaction, passing the entity object to delete as an argument. When you commit the transaction the entity is deleted from the database. This operation will fail if the entity is referenced elsewhere in the database.

29.6 Web Resources

Here are a few key online JPA resources.

4. For simplicity, we performed this example's database operations on the JavaFX application thread. Any potentially long-running database operations should be performed in separate threads using the techniques in Section 23.11.

<https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm>

The *Introduction to the Java Persistence API* chapter of the *Java EE 7 Tutorial*.

<https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm>

The *Java Persistence Query Language* chapter of the *Java EE 7 Tutorial*.

<http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>

The javax.persistence package documentation.

<https://platform.netbeans.org/tutorials/nbm-crud.html>

A NetBeans tutorial for creating a JPA-based app.

29.7 Wrap-Up

In this chapter, we introduced the Java Persistence API (JPA). We used the NetBeans IDE to create and populate a Java DB database, using Java DB's network server version, rather than the embedded version demonstrated in Chapter 24. We created NetBeans projects and added the libraries for JPA and the Java DB driver. Next, we used the NetBeans object-relational mapping tools to autogenerate entity classes from an existing database's schema. We then used those classes to interact with the database.

We queried the databases with both dynamic queries created in code and named queries that were autogenerated by NetBeans. We used the relationships between JPA entities to access data from multiple database tables.

Next, we used JPA transactions to insert new data in a database. We also discussed other JPA operations that you can perform in the context of transactions, such as updating existing entities in and deleting entities from a database. Finally, we listed several online JPA resources from which you can learn more about JPA. In the next chapter, we begin our two-chapter object-oriented design and implementation case study.



JavaServer™ Faces Web Apps: Part I

30



Objectives

In this chapter you'll learn:

- To create JavaServer Faces web apps.
- To create web apps consisting of multiple pages.
- To validate user input on a web page.
- To maintain user-specific state information throughout a web app with session tracking.

Outline

30_2 Chapter 30 JavaServer™ Faces Web Apps: Part I

- 30.1** Introduction
- 30.2** HyperText Transfer Protocol (HTTP) Transactions
- 30.3** Multitier Application Architecture
- 30.4** Your First JSF Web App
 - 30.4.1 The Default `index.xhtml` Document: Introducing Facelets
 - 30.4.2 Examining the `WebTimeBean` Class
 - 30.4.3 Building the `WebTime` JSF Web App in NetBeans
- 30.5** Model-View-Controller Architecture of JSF Apps
- 30.6** Common JSF Components
- 30.7** Validation Using JSF Standard Validators
- 30.8** Session Tracking
 - 30.8.1 Cookies
 - 30.8.2 Session Tracking with `@SessionScoped` Beans
- 30.9** Wrap-Up

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

30.1 Introduction

In this chapter, we introduce web app development in Java with JavaServer Faces (JSF). Web-based apps create content for web browser clients. This content includes eXtensible HyperText Markup Language (XHTML), JavaScript client-side scripting, Cascading Style Sheets (CSS), images and binary data. XHTML is an XML (eXtensible Markup Language) vocabulary that is based on HTML (HyperText Markup Language). We discuss only the features of these technologies that are required to understand the examples in this chapter. If you'd like more information on XHTML, XML, JavaScript and CSS, please visit our Resource Centers on each of these topics at

www.deitel.com/ResourceCenters.html

where you'll find links to introductions, tutorials and other valuable resources.

This chapter begins with an overview of how interactions between a web browser and web server work. We then present several web apps implemented with JSF. We continue this discussion in Chapter 31 with more advanced web applications.

Java multitier applications are typically implemented using Java Enterprise Edition (Java EE). The technologies we use to develop web apps here and in Chapter 31 are part of Java EE (<http://www.oracle.com/technetwork/java/javaee/overview/index.html>). After you study this chapter and the next, you can learn more about JavaServer Faces in Oracle's extensive Java EE tutorial at <http://docs.oracle.com/javaee/7/tutorial/>.

We focus on the JavaServer Faces 2.0¹ subset of Java EE. JavaServer Faces is a **web-application framework** that enables you to build multitier web apps by extending the framework with your application-specific capabilities. The framework handles the details of receiving client requests and returning responses for you so that you can focus on your application's functionality.

Required Software for This Chapter

To work with and implement the examples in this chapter and Chapters 31–32, you must install the **NetBeans 8** IDE and the **GlassFish** open-source application server. Both are available in a bundle from <http://netbeans.org/downloads/index.html>. You're prob-

1. The JavaServer Faces Specification: <http://bit.ly/JSF20Spec>.

30.2 HyperText Transfer Protocol (HTTP) Transactions

30_3

ably using a computer with the Windows, Linux or Max OS X operating system—installers are provided for each of these platforms. Download and execute the installer for the **Java EE** or **All** version—both include the required **Java Web and EE** and **Glassfish Server Open Source Edition** options. We assume you use the default installation options for your platform. Once you've installed NetBeans, run it. Then, use the **Help** menu's **Check for Updates** option to make sure you have the most up-to-date components.

30.2 HyperText Transfer Protocol (HTTP) Transactions

To learn how JSF web apps work, it's important to understand the basics of what occurs behind the scenes when a user requests a web page in a web browser. If you're already familiar with this and with multitier application architecture, you can skip to Section 30.4.

XHTML Documents

In its simplest form, a web page is nothing more than an XHTML document (also called an XHTML page) that describes content to display in a web browser. HTML documents normally contain *hyperlinks* that link to different pages or to other parts of the same page. When the user clicks a hyperlink, the requested web page loads into the user's web browser. Similarly, the user can type the address of a page into the browser's address field.

URLs

Computers that run **web-server** software make resources available, such as web pages, images, PDF documents and even objects that perform complex tasks such as database lookups and web searches. The HyperText Transfer Protocol (HTTP) is used by web browsers to communicate with web servers, so they can exchange information in a uniform and reliable manner. URLs (Uniform Resource Locators) identify the locations on the Internet of resources, such as those mentioned above. If you know the URL of a publicly available web resource, you can access it through HTTP.

Parts of a URL

When you enter a URL into a web browser, the browser uses the information in the URL to locate the web server that contains the resource and to request that resource from the server. Let's examine the components of the URL

```
http://www.deitel.com/books/downloads.html
```

The `http://` indicates that the resource is to be obtained using the HTTP protocol. The next portion, `www.deitel.com`, is the server's fully qualified **hostname**—the name of the *server* on which the *resource* resides. The computer that houses and maintains resources is usually referred to as the **host**. The hostname `www.deitel.com` is translated into an **IP (Internet Protocol) address**—a unique numerical value that identifies the server, much as a telephone number uniquely defines a particular phone line. This translation is performed by a **domain-name system (DNS) server**—a computer that maintains a database of hostnames and their corresponding IP addresses—and the process is called a **DNS lookup**. To test web apps, you'll often use your computer as the host. This host is referred to using the reserved domain name `localhost`, which translates to the IP address `127.0.0.1`. The fully qualified hostname can be followed by a colon (`:`) and a port number. Web servers typically await requests on port 80 by default; however, many development web servers use a different port number, such as 8080—as you'll see in Section 30.4.3.

30.4 Chapter 30 JavaServer™ Faces Web Apps: Part I

The remainder of the URL (i.e., `/books/downloads.html`) specifies both the name of the requested resource (the HTML document `downloads.html`) and its path, or location (`/books`), on the web server. The path could specify the location of an actual directory on the web server's file system. For security reasons, however, the path normally specifies the location of a **virtual directory**. The server translates the virtual directory into a real location on the server (or on another computer on the server's network), thus hiding the resource's true location. Some resources are created dynamically using other information, such as data from a database.

Making a Request and Receiving a Response

When given a URL, a web browser performs an HTTP transaction to retrieve and display the web page at that address. Figure 30.1 illustrates the transaction, showing the interaction between the web browser (the client) and the web server (the server).

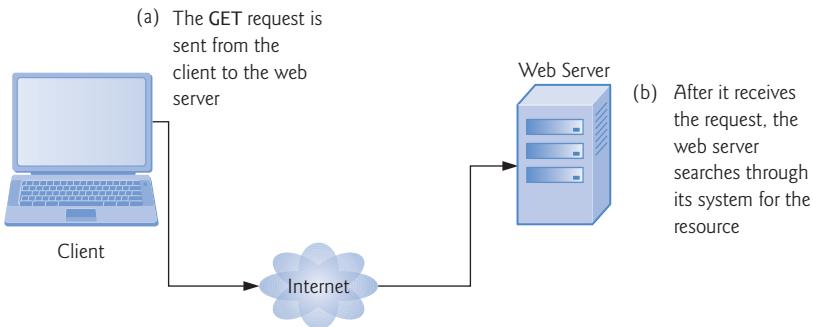


Fig. 30.1 | Client interacting with the web server. *Step 1: The GET request.*

In Fig. 30.1, the web browser sends an HTTP request to the server. Underneath the hood, the request (in its simplest form) is

```
GET /books/downloads.html HTTP/1.1
```

The word **GET** is an **HTTP method** indicating that the client wishes to obtain a resource from the server. The remainder of the request provides the pathname of the resource (e.g., an HTML document) and the protocol's name and version number (`HTTP/1.1`). As part of the client request, the browser also sends other required and optional information, such as the **Host** (which identifies the server computer) or the **User-Agent** (which identifies the web browser type and version number).

Any server that understands HTTP (version 1.1) can translate this request and respond appropriately. Figure 30.2 depicts the server responding to a request.

The server first responds by sending a line of text that indicates the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. For example,

```
HTTP/1.1 200 OK
```

indicates success, whereas

```
HTTP/1.1 404 Not found
```

30.2 HyperText Transfer Protocol (HTTP) Transactions

30_5

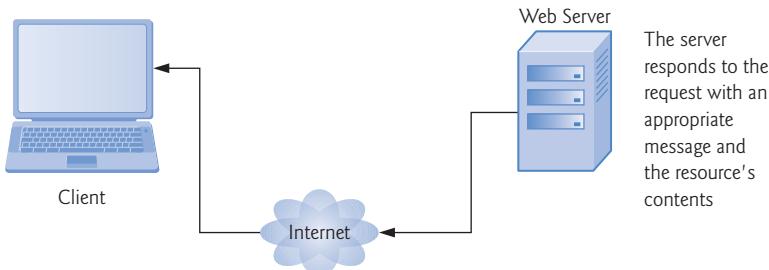


Fig. 30.2 | Client interacting with the web server. Step 2: The HTTP response.

informs the client that the web server could not locate the requested resource. On a successful request, the server appends the requested resource to the HTTP response. A complete list of numeric codes indicating the status of an HTTP transaction can be found at www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

HTTP Headers

The server then sends one or more **HTTP headers**, which provide additional information about the data that will be sent. If the server is sending an HTML text document, one HTTP header would read:

```
Content-type: text/html
```

The information provided in this header specifies the **Multipurpose Internet Mail Extensions (MIME)** type of the content that the server is transmitting to the browser. MIME is an Internet standard that specifies *data formats* so that programs can interpret data correctly. For example, the MIME type `text/plain` indicates that the sent information is text that can be displayed directly, without any interpretation of the content as HTML markup. Similarly, the MIME type `image/jpeg` indicates that the content is a JPEG image. When the browser receives this MIME type, it attempts to display the image. For a list of available MIME types, visit www.w3schools.com/media/media_mimeref.asp.

The header or set of headers is followed by a blank line, which indicates to the client browser that the server is finished sending HTTP headers. The server then sends the contents of the requested resource (such as, `downloads.html`). In the case of an HTML document, the web browser parses the HTML markup it receives and **renders** (or displays) the results.

HTTP GET and POST Requests

The two most common **HTTP request types** (also known as **request methods**) are **GET** and **POST**. A **GET** request typically asks for a resource on a server. Common uses of **GET** requests are to retrieve an HTML document or an image or to fetch search results from a search engine based on a user-submitted search term. A **POST** request typically sends data to a server. Common uses of **POST** requests are to send form data or documents to a server.

When a web page contains an HTML form in which the user can enter data, an HTTP request typically posts that data to a **server-side form handler** for processing. For example, when a user performs a search or participates in a web-based survey, the web server receives the information specified in the form as part of the request.

30_6 Chapter 30 JavaServer™ Faces Web Apps: Part I

GET requests and POST requests can both send form data to a web server, yet each request type sends the information differently. A GET request sends information to the server in the URL, as in `www.google.com/search?q=deitel`. Here, `search` is the name of Google's server-side form handler, `q` is the *name* of a variable in Google's search form and `deitel` is the search term. A `?` separates the **query string** from the rest of the URL in a request. A *name/value* pair is passed to the server with the *name* and the *value* separated by an equals sign (`=`). If more than one *name/value* pair is submitted, each is separated from the next by an ampersand (`&`). The server uses data passed in a query string to retrieve an appropriate resource. The server then sends a **response** to the client. A GET request may be initiated by submitting an HTML form whose `method` attribute is set to "get", by typing the URL (possibly containing a query string) directly into the browser's address bar or through a hyperlink when the user clicks the link.

A POST request sends form data as part of the HTTP message, not as part of the URL. The specification for GET requests does not limit the query string's number of characters, but some web browsers do—for example, Internet Explorer restricts the length to 2083 characters), so it's often necessary to send large pieces of information using POST. Sometimes POST is preferred because it hides the submitted data from the user by embedding it in an HTTP message.



Software Engineering Observation 30.1

The data sent in a POST request is not part of the URL, and the user can't see the data by default. However, tools are available that expose this data, so you should not assume that the data is secure just because a POST request is used.

Client-Side Caching

Browsers often **cache** (save on disk) web pages for quick reloading. If there are no changes between the version stored in the cache and the current version on the web, the browser uses the cached copy to speed up your browsing experience. An HTTP response can indicate the length of time for which the content remains "fresh." If this amount of time has not been reached, the browser can avoid another request to the server. Otherwise, the browser requests the document from the server. Thus, the browser minimizes the amount of data that must be downloaded for you to view a web page. Browsers typically do not cache the server's response to a POST request, because the next POST might not return the same result. For example, in a survey, many users could visit the same web page and answer a question. The survey results could then be displayed for the user. Each new answer changes the survey results.

When you use a web-based search engine, the browser normally supplies the information you specify in an HTML form to the search engine with a GET request. The search engine performs the search, then returns the results to you as a web page. Such pages are sometimes cached by the browser in case you perform the same search again.

30.3 Multitier Application Architecture

Web apps are **multitier applications** (sometimes referred to as **n-tier applications**). Multitier applications divide functionality into separate **tiers** (i.e., logical groupings of functionality). Although tiers can be located on the same computer, the tiers of web apps often reside on separate computers. Figure 30.3 presents the basic structure of a three-tier web app.

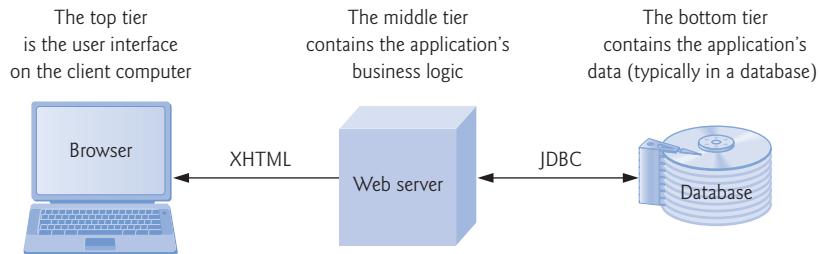


Fig. 30.3 | Three-tier architecture.

The **information tier** (also called the **data tier** or the **bottom tier**) maintains data pertaining to the application. This tier typically stores data in a *relational database management system (RDBMS)*. We discussed RDBMSs in Chapter 24. For example, a retail store might have a database for storing product information, such as descriptions, prices and quantities in stock. The same database also might contain customer information, such as user names, billing addresses and credit card numbers. This tier can contain multiple databases, which together comprise the data needed for our application.

The **middle tier** implements **business logic**, **controller logic** and **presentation logic** to control interactions between the application's clients and the application's data. The middle tier acts as an intermediary between data in the information tier and the application's clients. The middle-tier controller logic processes client requests (such as requests to view a product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client. Web apps typically present data to clients as HTML documents.

Business logic in the middle tier enforces **business rules** and ensures that data is reliable before the server application updates the database or presents the data to users. Business rules dictate how clients can and cannot access application data, and how applications process data. For example, a business rule in the middle tier of a retail store's web app might ensure that all product quantities remain positive. A client request to set a negative quantity in the bottom tier's product-information database would be rejected by the middle tier's business logic.

The **client tier**, or **top tier**, is the application's user interface, which gathers input and displays output. Users interact directly with the application through the user interface (typically viewed in a web browser), keyboard and mouse. In response to user actions (e.g., clicking a hyperlink), the client tier interacts with the middle tier to make requests and to retrieve data from the information tier. The client tier then displays the data retrieved from the middle tier to the user. The client tier never directly interacts with the information tier.

30.4 Your First JSF Web App

Let's begin with a simple example. Figure 30.4 shows the output of our `WebTime` app. When you invoke this app from a web browser, the browser requests the app's default JSF page. The web server receives this request and passes it to the **JSF web-application framework** for processing. This framework is available in any Java EE-compliant application server (such as the GlassFish application server used in this chapter) or any JavaServer

30_8 Chapter 30 JavaServer™ Faces Web Apps: Part I

Faces-compliant container (such as Apache Tomcat). The framework includes the **Faces servlet**—a software component running on the server that processes each requested JSF page so that the server can eventually return a response to the client. In this example, the Faces servlet processes the JSF document in Fig. 30.5 and forms a response containing the text "Current time on the web server:" followed by the web server's local time. We demonstrate this chapter's examples on the GlassFish server that you installed with NetBeans locally on your computer.



Fig. 30.4 | Sample output of the WebTime app.

Executing the WebTime App

To run this example on your own computer, perform the following steps:

1. Open the NetBeans IDE.
2. Select **File > Open Project...** to display the **Open Project** dialog.
3. Navigate to the ch30 folder in the book's examples and select **WebTime**.
4. Click the **Open Project** button.
5. Right click the project's name in the **Projects** tab (in the upper-left corner of the IDE, below the toolbar) and select **Run** from the pop-up menu.

This launches the GlassFish application server (if it isn't already running), installs the web app onto the server, then opens your computer's default web browser which requests the **WebTime** app's default JSF page. The browser should display a web page similar to that in Fig. 30.4.

30.4.1 The Default `index.xhtml` Document: Introducing Facelets

This app contains a single web page and consists of two related files—a JSF document named `index.xhtml` (Fig. 30.5) and a supporting Java source-code file (Fig. 30.6), which we discuss in Section 30.4.2. First we discuss the markup in `index.xhtml` and the supporting source code, then we provide step-by-step instructions for creating this web app in Section 30.4.3. Most of the markup in Fig. 30.5 was generated by NetBeans. We've reformatted the generated code to match our coding conventions used throughout the book.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- JSF page that displays the current time on the web server -->
```

Fig. 30.5 | JSF page that displays the current time on the web server. (Part I of 2.)

```

5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8    xmlns:h="http://java.sun.com/jsf/html">
9    <h:head>
10      <title>WebTime: A Simple Example</title>
11      <meta http-equiv="refresh" content="60"/>
12    </h:head>
13    <h:body>
14      <h1>Current time on the web server: #{webTimeBean.time}</h1>
15    </h:body>
16  </html>
```

Fig. 30.5 | JSF page that displays the current time on the web server. (Part 2 of 2.)

Facelets: XHTML and JSF Markup

You present your web app’s content in JSF using **Facelets**—a combination of XHTML markup and JSF markup. **XHTML**—the **Extensible HyperText Markup Language**—specifies the content of a web page that is displayed in a web browser. XHTML separates the **presentation** of a document (that is, the document’s appearance when rendered by a browser) from the **structure** of the document’s data. A document’s presentation might specify where the browser should place an element in a web page or what fonts and colors should be used to display an element. The XHTML 1.0 Strict Recommendation allows only a document’s structure to appear in a valid XHTML document, and not its presentation. Presentation is specified with Cascading Style Sheets (CSS). JSF uses the XHTML 1.0 Transitional Recommendation by default. Transitional markup may include some non-CSS formatting, but this is not recommended.

XML Declaration, Comments and the DOCTYPE Declaration

With the exception of lines 3–4, 10–11 and 14, the code shown in Fig. 30.5 was generated by NetBeans. Line 1 is an XML declaration, indicating that the JSF document is expressed in XML 1.0 syntax. Lines 3–4 are comments that we added to the document to indicate its filename and purpose. Lines 5–6 are a DOCTYPE declaration indicating the version of XHTML used in the markup. This can be used by a web browser to validate the syntax of the document.

Specifying the XML Namespaces Used in the Document

Line 7 begins the document’s root **html** element, which spans lines 7–16. Each element typically consists of a starting and ending tag. The starting **<html>** tag (lines 7–8) may contain one or more **xmlns** attributes. Each **xmlns** attribute has a **name** and a **value** separated by an equal sign (=), and specifies an XML namespace of elements that are used in the document. Just as Java packages can be used to differentiate class names, XML namespaces can be used to differentiate sets of elements. When there’s a naming conflict, fully qualified tag names can be used to resolve the conflict.

Line 7 specifies a required **xmlns** attribute and its value (<http://www.w3.org/1999/xhtml>) for the **html** element. This indicates that the **html** element and any other unqualified element names are part of the default XML namespace that’s used in this document.



30_10 Chapter 30 JavaServer™ Faces Web Apps: Part I

The `xmlns:h` attribute (line 8) specifies a prefix and a URL for JSF's **HTML Tag Library**, allowing the document to use JSF's elements from that library. A tag library defines a set of elements that can be inserted into the XHTML markup. The elements in the HTML Tag Library generate XHTML elements. Based on line 7, each element we use from the HTML Tag Library must be preceded by the `h:` prefix. This tag library is one of several supported by JSF that can be used to create Facelets pages. We'll discuss others as we use them. For a complete list of JSF tag libraries and their elements and attributes, visit

```
javaserverfaces.java.net/nonav/docs/2.0/pdldocs/facellets/
```

The `h:head` and `h:body` Elements

The `h:head` element (lines 9–12) defines the XHTML page's head element. In this example the head contains an HTML `title` element and a `meta` element. The document's `title` (line 10) typically appears in the browser window's title bar, or a browser tab if you have multiple web pages open in the browser at once. The `title` is also used when search engines index your web pages. The `meta` element (line 11) tells the browser to refresh the page every 60 seconds. This forces the browser to re-request the page once per minute.

The `h:body` element (lines 13–15) represents the page's content. In this example, it contains a XHTML `h1` header element (line 14) that represents the text to display when this document is rendered in the web browser. The `h1` element contains some literal text (`Current time on the web server:`) that's simply placed into the response to the client and a **JSF Expression Language (EL)** expression that obtains a value dynamically and inserts it into the response. The expression

```
#{{webTimeBean.time}}
```

indicates that the web app has an object named `webTimeBean` which contains a property named `time`. The property's value replaces the expression in the response that's sent to the client. We'll discuss this EL expression in more detail shortly.

30.4.2 Examining the `WebTimeBean` Class

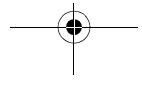
JSF documents typically interact with one or more Java objects to perform the app's tasks. As you saw, this example obtains the time on the server and sends it as part of the response.

JavaBeans

JavaBeans objects are instances of classes that follow certain conventions for class design. Each JavaBean class typically contains data and methods. A JavaBean exposes its data to a JSF document as **properties**. Depending on their use, these properties can be read/write, read-only or write-only. To define a read/write property, a JavaBean class provides `set` and `get` methods for that property. For example, to create a `String` property `firstName`, the class would provide methods with the following first lines:

```
public String getFirstName()  
public void setFirstName(String name)
```

The fact that both method names contain "FirstName" with an uppercase "F" indicates that the class exposes a `firstName` property with a lowercase "F." This naming convention is part of the JavaBeans Specification (available at bit.ly/JavaBeansSpecification). A



read-only property would have only a *get* method and a write-only property only a *set* method. The JavaBeans used in JSF are also **POJOs (plain old Java objects)**, meaning that—unlike prior versions of JSF—you do *not* need to extend a special class to create the beans used in JSF applications. Instead various annotations are used to “inject” functionality into your beans so they can be used easily in JSF applications. The JSF framework is responsible for creating and managing objects of your JavaBean classes for you—you’ll see how to enable this momentarily.

Class WebTimeBean

Figure 30.6 presents the `WebTimeBean` class that allows the JSF document to obtain the web server’s time. You can name your bean classes like any other class. We chose to end the class name with “Bean” to indicate that the class represents a JavaBean. The class contains just a `getTime` method (lines 13–17), which defines the read-only `time` property of the class. Recall that we access this property at line 14 of Fig. 30.5. Lines 15–16 create a `Date` object, then format and return the time as a `String`.

```

1 // WebTimeBean.java
2 // Bean that enables the JSF page to retrieve the time from the server
3 package webtime;
4
5 import java.text.DateFormat;
6 import java.util.Date;
7 import javax.faces.bean.ManagedBean;
8
9 @ManagedBean(name="webTimeBean")
10 public class WebTimeBean
11 {
12     // return the time on the server at which the request was received
13     public String getTime()
14     {
15         return DateFormat.getTimeInstance(DateFormat.LONG).format(
16             new Date());
17     }
18 }
```

Fig. 30.6 | Bean that enables the JSF page to retrieve the time from the server.

The @ManagedBean Annotation

Line 9 uses the **@ManagedBean annotation** (from the package `javax.faces.bean`) to indicate that the JSF framework should create and manage the `WebTimeBean` object(s) used in the application. The parentheses following the annotation contain the optional **name attribute**—in this case, indicating that the bean object created by the JSF framework should be called `webTimeBean`. If you specify the annotation without the parentheses and the `name` attribute, the JSF framework will use the class name with a lowercase first letter (that is, `webTimeBean`) as the default bean name.

Processing the EL Expression

When the Faces servlet encounters an EL expression that accesses a bean property, it automatically invokes the property’s *set* or *get* method based on the context in which the

30_12 Chapter 30 JavaServer™ Faces Web Apps: Part I

property is used. In line 14 of Fig. 30.5, accessing the property `webTimeBean.time` results in a call to the bean's `getTime` method, which returns the web server's time. If this bean object does not yet exist, the JSF framework instantiates it, then calls the `getTime` method on the bean object. The framework can also discard beans that are no longer being used. We discuss only the EL expressions that we use in this chapter. For more EL details, see Chapter 9 of the Java EE 7 tutorial at

<http://docs.oracle.com/javaee/7/tutorial/>

and Chapter 5 of the JSF 2.0 specification, which you can download from

http://download.oracle.com/otndocs/jcp/jsf-2_2-fr-spec/index.html

30.4.3 Building the WebTime JSF Web App in NetBeans

We'll now build the `WebTime` app from scratch using NetBeans.

Creating the JSF Web Application Project

Begin by opening the NetBeans IDE and performing the following steps:

1. Select **File > New Project...** to display the **New Project** dialog. Select **Java Web** in the **Categories** pane, **Web Application** in the **Projects** pane and click **Next >**.
2. In the dialog's **Name and Location** step, specify **WebTime** as the **Project Name**. In the **Project Location** field, specify where you'd like to store the project (or keep the default location). These settings will create a `WebTime` directory to store the project's files in the parent directory you specified. Keep the other default settings and click **Next >**.
3. In the dialog's **Server and Settings** step, specify **GlassFish Server 4.1** as the **Server** and **Java EE 7 Web** as the **Java EE Version** (these may be the default). Keep the default **Context Path** and click **Next >**.
4. In the dialog's **Frameworks** step, select **JavaServer Faces**, then click **Finish** to create the web application project.

Examining the NetBeans Projects Window

Figure 30.7 displays the **Projects** window, which appears in the upper-left corner of the IDE. This window displays the contents of the project. The app's XHTML documents are placed in the **Web Pages** node. NetBeans supplies the default web page `index.xhtml` that will be displayed when a user requests this web app from a browser. When you add Java source code to the project, it will be placed in the **Source Packages** node.

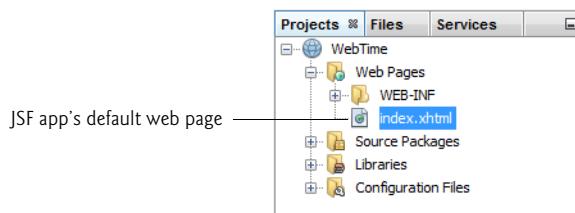
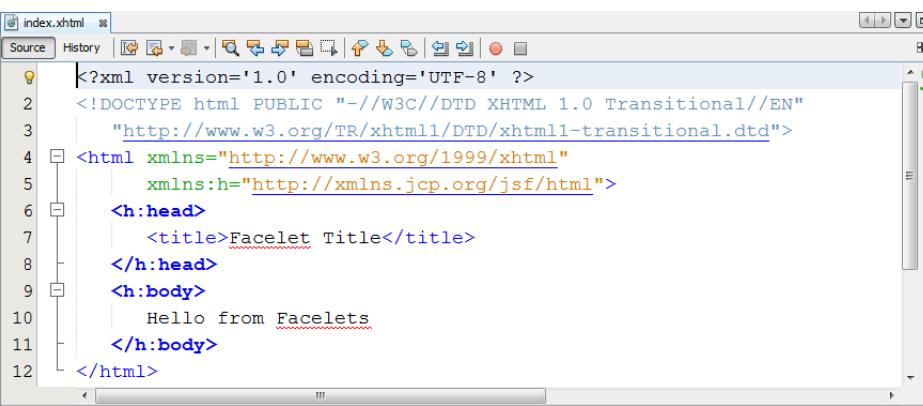


Fig. 30.7 | Projects window for the `WebTime` project.

30.4 Your First JSF Web App 30_13

Examining the Default index.xhtml Page

Figure 30.8 displays `index.xhtml`—the default page that will be displayed when a user requests this web app. We reformatted the code to match our coding conventions. When this file is first created, it contains elements for setting up the page, including linking to the page's style sheet and declaring the JSF libraries that will be used. By default, NetBeans does not show line numbers in the source-code editor. To view the line numbers, select **View > Show Line Numbers**.



```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
<title>Facelet Title</title>
</h:head>
<h:body>
Hello from Facelets
</h:body>
</html>

```

Fig. 30.8 | Default `index.xhtml` page generated by NetBeans for the web app.

Editing the h:head Element's Contents

Modify line 7 of Fig. 30.8 by changing the `title` element's content from "Facelet Title" to "Web Time: A Simple Example". After the closing `</title>` tag, press *Enter*, then insert the `meta` element

```
<meta http-equiv="refresh" content="60"/>
```

which will cause the browser to refresh this page once per minute. As you type, notice that NetBeans provides a code-completion window to help you write your code. For example, after typing "`<meta`" and a space, the IDE displays the code-completion window in Fig. 30.9, which shows the list of valid attributes for the starting tag of a `meta` element. You can then double click an item in the list to insert it into your code. Code-completion support is provided for XHTML elements, JSF elements and Java code.

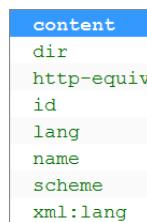


Fig. 30.9 | NetBeans code-completion window.



30_14 Chapter 30 JavaServer™ Faces Web Apps: Part I

Editing the h:body Element's Contents

In the h:body element, replace "Hello from Facelets" with the h1 header element

```
<h1>Current time on the web server: </h1>
```

Don't insert the expression #{webTimeBean.time} yet. After we define the WebTimeBean class, we'll come back to this file and insert this expression to demonstrate that the IDE provides code-completion support for the Java classes you define in your project.

Defining the Page's Logic: Class WebTimeBean

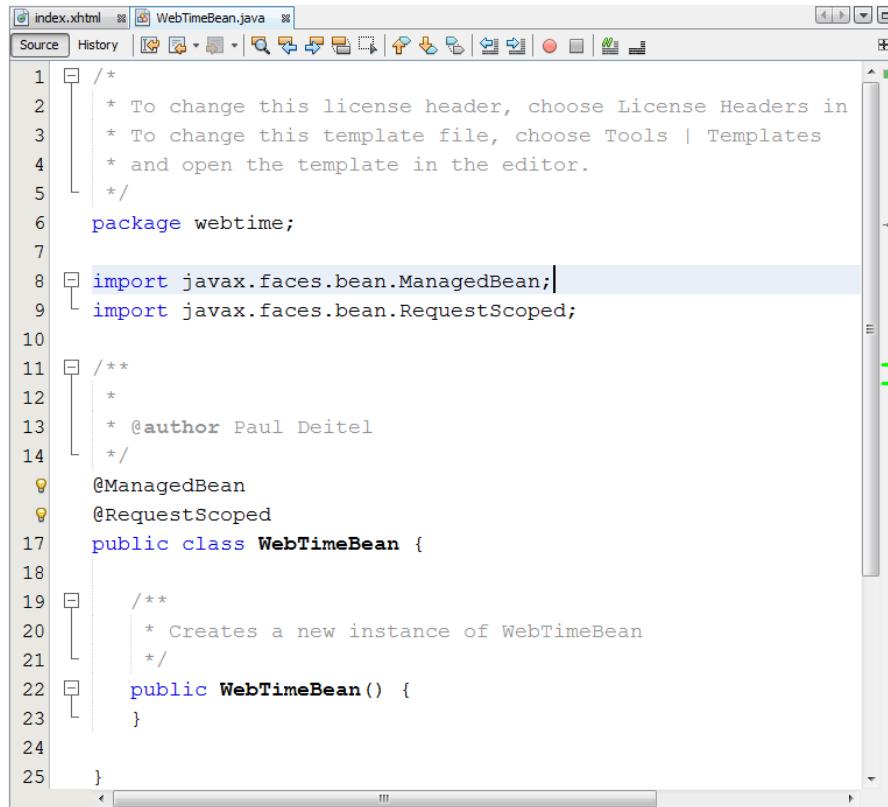
We'll now create the WebTimeBean class—the @ManagedBean class that will allow the JSF page to obtain the web server's time. To create the class, perform the following steps:

1. In the NetBeans **Projects** tab, right click the WebTime project's **Source Packages** node and select **New > Other...** to display the **New File** dialog.
2. In the **Categories** list, select **JavaServer Faces**, then in the **File Types** list select **JSF Managed Bean**. Click **Next >**.
3. In the **Name and Location** step, specify **WebTimeBean** as the **Class Name** and **webtime** as the **Package**, then click **Finish**.

NetBeans creates the **WebTimeBean.java** file and places it within the **webtime** package in the project's **Source Packages** node. Figure 30.10 shows this file's default source code displayed in the IDE. At line 16, notice that NetBeans added the **@RequestScoped annotation** to the class—this indicates that an object of this class exists only for the duration of the request that's being processed. (We'll discuss **@RequestScoped** and other bean scopes in more detail in Section 30.8.) We did not include this annotation in Fig. 30.6, because all JSF beans are request scoped by default. Replace the code in Fig. 30.10 with the code in Fig. 30.6.

30.4 Your First JSF Web App

30_15



```

1  /*
2   * To change this license header, choose License Headers in
3   * Project Properties. To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package webtime;
7
8  import javax.faces.bean.ManagedBean;
9  import javax.faces.bean.RequestScoped;
10
11 /**
12  *
13  * @author Paul Deitel
14  */
15 @ManagedBean
16 @RequestScoped
17 public class WebTimeBean {
18
19 /**
20  * Creates a new instance of WebTimeBean
21  */
22 public WebTimeBean() {
23 }
24
25 }

```

Fig. 30.10 | Default source code for the WebTimeBean class.

Adding the EL Expression to the index.xhtml Page

Now that you've created the WebTimeBean, let's go back to the `index.xhtml` file and add the EL expression that will obtain the time. In the `index.xhtml` file, modify the line

```
<h1>Current time on the web server: </h1>
```

by inserting the expression `#{{webTimeBean.time}}` before the `h1` element's closing tag. After you type the characters `#` and `{`, the IDE automatically inserts the closing `}`, inserts the cursor between the braces and displays the code-completion window. This shows various items that could be placed in the braces of the EL expression, including the `webTimeBean` object (of type `WebTimeBean`). To insert `webTimeBean` in the code, you can type the object's name or double click it in the code-completion window. As you type, the list of items in the code-completion window is filtered by what you've typed so far.

Running the Application

You've now completed the `WebTime` app. To test it, right click the project's name in the **Projects** tab and select **Run** from the pop-up menu. The IDE will compile the code and deploy (that is, install) the `WebTime` app on the GlassFish application server running on your local machine. Then, the IDE will launch your default web browser and request the



30_16 Chapter 30 JavaServer™ Faces Web Apps: Part I

WebTime app's default web page (`index.xhtml`). Because GlassFish is installed on your local computer, the URL displayed in the browser's address bar will be

`http://localhost:8080/WebTime/`

where 8080 is the port number on which the GlassFish server runs by default. Depending on your web browser, the `http://` may not be displayed (Fig. 30.5).

Debugging the Application

If there's a problem with your web app's logic, you can press `<Ctrl> F5` (`⌘ F5` on Mac OS X) to build the application and run it in debug mode—the NetBeans built-in debugger can help you troubleshoot applications. If you press `F6`, the program executes without debugging enabled.

Testing the Application from Other Web Browsers

After deploying your project, you can test it from another web browser on your computer by entering the app's URL into the other browser's address field. Since your application resides on the local file system, GlassFish must be running. If you've already executed the application using one of the techniques above and have not closed NetBeans, GlassFish will still be running. Otherwise, you can start the server from the IDE. To do so, open the **Services** tab (located in the same panel as the **Projects**), expand the **Servers** node, right click **GlassFish Server 4.1** (or whichever version you have installed) and select **Start**. Then you can type the URL in the browser to execute the application.

30.5 Model-View-Controller Architecture of JSF Apps

JSF applications adhere to the **Model-View-Controller (MVC) architecture**, which separates an application's data (contained in the **model**) from the graphical presentation (the **view**) and the processing logic (the **controller**). Figure 30.11 shows the relationships between components in MVC.

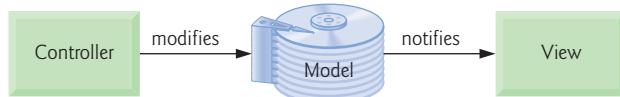


Fig. 30.11 | Model-View-Controller architecture.

In JSF, the controller is the JSF framework and is responsible for coordinating interactions between the view and the model. The model contains the application's data (typically in a database), and the view presents the data stored in the model (typically as web pages). When a user interacts with a JSF web app's view, the framework interacts with the model to store and/or retrieve data. When the model changes, the view is updated with the changed data.

30.6 Common JSF Components

As mentioned in Section 30.4, JSF supports several tag libraries. In this section, we introduce several of the JSF HTML Tag Library's elements and one element from the **JSF Core Tag Library**. Figure 30.12 summarizes elements discussed in this section.

JSF component	Description
<code>h:form</code>	Inserts an XHTML <code>form</code> element into a page.
<code>h:commandButton</code>	Displays a button that triggers an event when clicked. Typically, such a button is used to submit a form's user input to the server for processing.
<code>h:graphicImage</code>	Displays an image (e.g., GIF and JPG).
<code>h:inputText</code>	Displays a text box in which the user can enter input.
<code>h:outputLink</code>	Displays a hyperlink.
<code>h:panelGrid</code>	Displays an XHTML <code>table</code> element.
<code>h:selectOneMenu</code>	Displays a drop-down list of choices from which the user can make a selection.
<code>h:selectOneRadio</code>	Displays a set of radio buttons.
<code>f:selectItem</code>	Specifies an item in an <code>h:selectOneMenu</code> or <code>h:selectOneRadio</code> (and other similar components).

Fig. 30.12 | Commonly used JSF components.

All of these elements are mapped by JSF framework to a combination of XHTML elements and JavaScript code that enables the browser to render the page. JavaScript is a scripting language that's interpreted in all of today's popular web browsers. It can be used to perform tasks that manipulate web-page elements in a web browser and provide interactivity with the user. You can learn more about JavaScript in our JavaScript Resource Center at www.deitel.com/JavaScript/.

Figure 30.13 displays a form for gathering user input. [Note: To create this application from scratch, review the steps in Section 30.4.3 and name the application **WebComponents**.] The `h:form` element (lines 14–55) contains the components with which a user interacts to provide data, such as registration or login information, to a JSF app. This example uses the components summarized in Fig. 30.12. This example does not perform a task when the user clicks the `Register` button. Later, we demonstrate how to add functionality to many of these components.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- Registration form that demonstrates various JSF components -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Fig. 30.13 | Registration form that demonstrates various JSF components. (Part 1 of 3.)

30_18 Chapter 30 JavaServer™ Faces Web Apps: Part I

```

7  <html xmlns="http://www.w3.org/1999/xhtml"
8      xmlns:h="http://java.sun.com/jsf/html"
9      xmlns:f="http://java.sun.com/jsf/core">
10     <h:head>
11         <title>Sample Registration Form</title>
12     </h:head>
13     <h:body>
14         <h:form>
15             <h1>Registration Form</h1>
16             <p>Please fill in all fields and click Register</p>
17             <h:panelGrid columns="4" style="height: 96px; width:456px;">
18                 <h:graphicImage name="fname.png" library="images"/>
19                 <h:inputText id="firstNameInputText"/>
20                 <h:graphicImage name="lname.png" library="images"/>
21                 <h:inputText id="lastNameInputText"/>
22                 <h:graphicImage name="email.png" library="images"/>
23                 <h:inputText id="emailInputText"/>
24                 <h:graphicImage name="phone.png" library="images"/>
25                 <h:inputText id="phoneInputText"/>
26             </h:panelGrid>
27             <p><h:graphicImage name="publications.png" library="images"/>
28                 <br/>Which book would you like information about?</p>
29             <h:selectOneMenu id="booksSelectOneMenu">
30                 <f:selectItem itemValue="CHTP"
31                     itemLabel="C How to Program" />
32                 <f:selectItem itemValue="CPPHTP"
33                     itemLabel="C++ How to Program" />
34                 <f:selectItem itemValue="IW3HTP"
35                     itemLabel="Internet & World Wide Web How to Program" />
36                 <f:selectItem itemValue="JHTP"
37                     itemLabel="Java How to Program" />
38                 <f:selectItem itemValue="VBHTP"
39                     itemLabel="Visual Basic How to Program" />
40                 <f:selectItem itemValue="VCSHTP"
41                     itemLabel="Visual C# How to Program" />
42             </h:selectOneMenu>
43             <p><h:outputLink value="http://www.deitel.com">
44                 Click here to learn more about our books
45             </h:outputLink></p>
46             <h:graphicImage name="os.png" library="images"/>
47             <h:selectOneRadio id="osSelectOneRadio">
48                 <f:selectItem itemValue="WinVista" itemLabel="Windows Vista"/>
49                 <f:selectItem itemValue="Win7" itemLabel="Windows 7"/>
50                 <f:selectItem itemValue="OSX" itemLabel="macOS"/>
51                 <f:selectItem itemValue="Linux" itemLabel="Linux"/>
52                 <f:selectItem itemValue="Other" itemLabel="Other"/>
53             </h:selectOneRadio>
54             <h:commandButton value="Register"/>
55         </h:form>
56     </h:body>
57 </html>
```

Fig. 30.13 | Registration form that demonstrates various JSF components. (Part 2 of 3.)

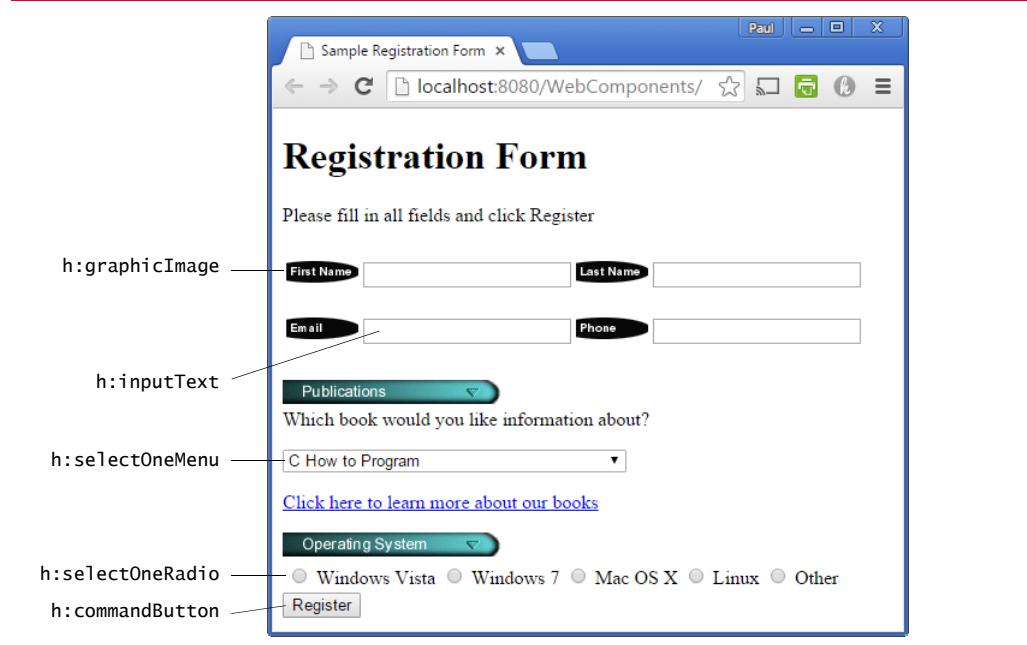


Fig. 30.13 | Registration form that demonstrates various JSF components. (Part 3 of 3.)

h:panelGrid Element

Lines 17–26 define an ***h:panelGrid*** element for organizing elements in the page. This element inserts an XHTML table in the page. The ***h:*** prefix indicates that ***panelGrid*** is from the JSF HTML Tag Library. The ***columns*** attribute specifies the number of columns in the ***table***. The elements between the ***h:panelGrid***'s start tag (line 17) and end tag (line 26) are automatically placed into the ***table***'s columns from left to right in the order they appear in the JSF page. When the number of elements in a row exceeds the number of columns, the ***h:panelGrid*** creates a new row. We use the ***h:panelGrid*** to control the positions of the ***h:graphicImage*** and ***h:inputText*** elements in the user information section of the page. In this case, there are eight elements in the ***h:panelGrid***, so the first four (lines 18–21) are placed in the table's first row and the last four are placed in the second row. The ***h:panelGrid***'s ***style*** attribute specifies the CSS formatting for the ***table***. We use the CSS attributes ***width*** and ***height*** to specify the width and height of the table in pixels (px). The ***h:panelGrid*** contains pairs of ***h:graphicImage*** and ***h:inputText*** elements.

h:graphicImage Element and Resource Libraries

Each ***h:graphicImage*** displays an image in the page. For example, line 18 inserts the image `fname.png`—as specified by the ***name*** attribute. You add resources that are used throughout your app—such as images, CSS files, JavaScript files—to your web apps by placing them in the app's ***resources*** folder within your project's ***Web Pages*** node. Each subfolder of ***resources*** represents a ***resource library***. Typically, images are placed in an ***images*** library and CSS files in a ***css*** library. In line 18, we specify that the image is located in the ***images*** library with the ***library*** attribute. JSF knows that the value of this attribute represents a folder within the ***resources*** folder.



30_20 Chapter 30 JavaServer™ Faces Web Apps: Part I

You can create any library you like in the resources folder. To create this folder:

1. Expand your app's node in the NetBeans Projects tab.
2. Right click the Web Pages node and select New > Folder... to display the New Folder dialog. [Note: If the Folder... option is not available in the popup menu, select Other..., then in the Categories pane select Other and in the File Types pane select Folder and click Next >.]
3. Specify resources as the *Folder Name* and press *Finish*.

Next, right click the resources folder you just created and create an images subfolder. You can then drag the images from your file system onto the images folder to add them as resources. The images in this example are located in the images directory with the chapter's examples.

The `h:graphicImage` in line 18 is a so-called **empty element**—an element that does not have content between its start and end tags. In such an element, data is typically specified as attributes in the start tag, such as the `name` and `library` attributes in line 18. You can close an empty element either by placing a slash immediately preceding the start tag's right angle bracket, as shown in line 18, or by explicitly writing an end tag.

h:inputText Element

Line 19 defines an ***h:inputText*** element in which the user can enter text or the app can display text. For any element that might be accessed by other elements of the page or that might be used in server-side code, you should specify an ***id attribute***. We specified these attributes in this example, even though the app does not provide any functionality. We'll use the `id` attribute starting with the next example.

h:selectOneMenu Element

Lines 29–42 define an ***h:selectOneMenu*** element, which is typically rendered in a web page as a drop-down list. When a user clicks the drop-down list, it expands and displays a list from which the user can make a selection. Each item to display appears between the start and end tags of this element as an ***f:selectItem*** element (lines 30–41). This element is part of the JSF Core Tag Library. The XML namespace for this tag library is specified in the `html` element's start tag at line 9. Each `f:selectItem` has `itemValue` and `itemLabel` attributes. The `itemLabel` is the string that the user will see in the browser, and the `itemValue` is the value that's returned when you programmatically retrieve the user's selection from the drop-down list (as you'll see in a later example).

h:outputLink Element

The ***h:outputLink*** element (lines 43–45) inserts a hyperlink in a web page. Its ***value attribute*** specifies the resource (`http://www.deitel.com` in this case) that's requested when a user clicks the hyperlink. By default, `h:outputLink` elements cause pages to open in the same browser window, but you can set the element's `target` attribute to change this behavior.

h:selectOneMenu Element

Lines 47–53 define an ***h:selectOneRadio*** element, which provides a series of radio buttons from which the user can select only one. Like an `h:selectOneMenu`, an `h:selectOneRadio` displays items that are specified with `f:selectItem` elements.

30.7 Validation Using JSF Standard Validators

30_21

h:commandButton Element

Lines 54 defines an **h:commandButton** element that triggers an action when clicked—in this example, we don't specify the action to trigger, so the default action occurs (re-requesting the same page from the server) when the user clicks this button. An **h:commandButton** typically maps to an XHTML **input** element with its **type** attribute set to "submit". Such elements are often used to submit a form's user input values to the server for processing.

30.7 Validation Using JSF Standard Validators

Validating input is an important step in collecting information from users. **Validation** helps prevent processing errors due to incomplete, incorrect or improperly formatted user input. For example, you may perform validation to ensure that all required fields contain data or that a zip-code field has the correct number of digits. The JSF Core Tag Library provides several standard validator components and allows you to create your own custom validators. Multiple validators can be specified for each input element. The validators are:

- **f:validateLength**—determines whether a field contains an acceptable number of characters.
- **f:validateDoubleRange** and **f:validateLongRange**—determine whether numeric input falls within acceptable ranges of **double** or **long** values, respectively.
- **f:validateRequired**—determines whether a field contains a value.
- **f:validateRegex**—determines whether a field contains a string that matches a specified regular expression pattern.
- **f:validateBean**—allows you to invoke a bean method that performs custom validation.

Validating Form Data in a Web Application

[Note: To create this application from scratch, review the steps in Section 30.4.3 and name the application **Validation**.] The example in this section prompts the user to enter a name, e-mail address and phone number in a form. When the user enters any data and presses the **Submit** button to submit the form's contents to the web server, validation ensures that the user entered a value in each field, that the entered name does not exceed 30 characters, and that the e-mail address and phone-number values are in an acceptable format. In this example, (555) 123-4567, 555-123-4567 and 123-4567 are all considered valid phone numbers. Once valid data is submitted, the JSF framework stores the submitted values in a bean object of class **ValidationBean** (Fig. 30.14), then sends a response back to the web browser. We simply display the validated data in the page to demonstrate that the server received the data. A real business application would typically store the submitted data in a database or in a file on the server.

Class ValidationBean

Class **ValidationBean** (Fig. 30.14) provides the read/write properties **name**, **email** and **phone**, and the read-only property **result**. Each read/write property has an instance variable (lines 11–13) and corresponding **set/get** methods (lines 16–25, 28–37 and 40–49) for manipulating the instance variables. The read-only property **response** has only a **get-**

30_22 Chapter 30 JavaServer™ Faces Web Apps: Part I

Result method (lines 52–60), which returns a paragraph (*p*) element containing the validated data. (You can create the *ValidationBean* managed bean class by using the steps presented in Fig. 30.4.3.)

```

1 // ValidationBean.java
2 // Validating user input.
3 package validation;
4
5 import java.io.Serializable;
6 import javax.faces.bean.ManagedBean;
7
8 @ManagedBean(name="validationBean")
9 public class ValidationBean implements Serializable
10 {
11     private String name;
12     private String email;
13     private String phone;
14
15     // return the name String
16     public String getName()
17     {
18         return name;
19     }
20
21     // set the name String
22     public void setName(String name)
23     {
24         this.name = name;
25     }
26
27     // return the email String
28     public String getEmail()
29     {
30         return email;
31     }
32
33     // set the email String
34     public void setEmail(String email)
35     {
36         this.email = email;
37     }
38
39     // return the phone String
40     public String getPhone()
41     {
42         return phone;
43     }
44
45     // set the phone String
46     public void setPhone(String phone)
47     {

```

Fig. 30.14 | *ValidationBean* stores the validated data, which is then used as part of the response to the client. (Part I of 2.)

30.7 Validation Using JSF Standard Validators

30_23

```

48     this.phone = phone;
49 }
50
51 // returns result for rendering on the client
52 public String getResult()
53 {
54     if (name != null && email != null && phone != null)
55         return "<p style=\"background-color:yellow;width:200px;" +
56             "padding:5px\">Name: " + getName() + "<br>E-Mail: " +
57             getEmail() + "<br>Phone: " + getPhone() + "</p>";
58     else
59         return ""; // request has not yet been made
60 }
61 }
```

Fig. 30.14 | ValidationBean stores the validated data, which is then used as part of the response to the client. (Part 2 of 2.)

index.xhtml

Figure 30.15 shows this app's *index.xhtml* file. The initial request to this web app displays the page shown in Fig. 30.15(a). When this app is initially requested, the beginning of the **JSF application lifecycle** uses this *index.xhtml* document to build the app's facelets view and sends it as the response to the client browser, which displays the form for user input. During this initial request, the EL expressions (lines 22, 30, 39 and 49) are evaluated to obtain the values that should be displayed in various parts of the page. Nothing is displayed initially as a result of these four EL expressions being evaluated, because no default values are specified for the bean's properties. The page's *h:form* element contains an *h:panelGrid* (lines 18–45) with three columns and an *h:commandButton* (line 46), which by default submits the contents of the form's fields to the server.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- Validating user input -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8      xmlns:h="http://java.sun.com/jsf/html"
9      xmlns:f="http://java.sun.com/jsf/core">
10 <h:head>
11     <title>Validating Form Data</title>
12     <h:outputStylesheet name="style.css" library="css"/>
13 </h:head>
14 <h:body>
15     <h:form>
16         <h1>Please fill out the following form:</h1>
17         <p>All fields are required and must contain valid information</p>
```

Fig. 30.15 | Form to demonstrate validating user input. (Part 1 of 3.)

30_24 Chapter 30 JavaServer™ Faces Web Apps: Part I

```

18      <h:panelGrid columns="3">
19          <h:outputText value="Name:"/>
20          <h:inputText id="nameInputText" required="true"
21              requiredMessage="Please enter your name"
22              value="#{validationBean.name}"
23              validatorMessage="Name must be fewer than 30 characters">
24                  <f:validateLength maximum="30" />
25          </h:inputText>
26          <h:message for="nameInputText" styleClass="error"/>
27          <h:outputText value="E-mail:"/>
28          <h:inputText id="emailInputText" required="true"
29              requiredMessage="Please enter a valid e-mail address"
30              value="#{validationBean.email}"
31              validatorMessage="Invalid e-mail address format">
32                  <f:validateRegex pattern=
33                      "\w+([-.\'])*\@\w+([-.\']\w+)*\.\w+([-.\']\w+)*" />
34          </h:inputText>
35          <h:message for="emailInputText" styleClass="error"/>
36          <h:outputText value="Phone:"/>
37          <h:inputText id="phoneInputText" required="true"
38              requiredMessage="Please enter a valid phone number"
39              value="#{validationBean.phone}"
40              validatorMessage="Invalid phone number format">
41                  <f:validateRegex pattern=
42                      "((\d{3})\ ?|(\d{3}-))?\d{3}-\d{4}" />
43          </h:inputText>
44          <h:message for="phoneInputText" styleClass="error"/>
45      </h:panelGrid>
46      <h:commandButton value="Submit" />
47      <h:outputText escape="false" value="#{validationBean.result}" />
48  </h:form>
49  </h:body>
50 </html>

```

a) Submitting the form before entering any information

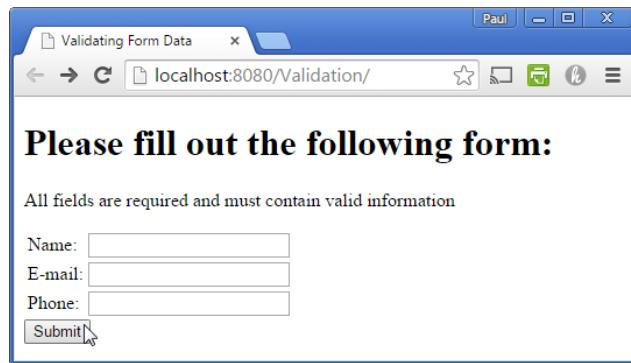


Fig. 30.15 | Form to demonstrate validating user input. (Part 2 of 3.)

b) Error messages displayed after submitting the empty form

c) Error messages displayed after submitting invalid information

d) Successfully submitted form

Fig. 30.15 | Form to demonstrate validating user input. (Part 3 of 3.)

30_26 Chapter 30 JavaServer™ Faces Web Apps: Part I

First Row of the h:panelGrid

In this application, we demonstrate several new elements and attributes. The first new element is the **h:outputText** element (line 19; from the JSF HTML Tag Library), which inserts text in the page. In this case, we insert a literal string ("Name:") that is specified with the element's **value** attribute.

The **h:inputText** element (lines 20–25) displays a text box in which the user can enter a name. We've specified several attributes for this element:

- **id**—This enables other elements or server-side code to reference this element.
- **required**—Ensuring that the user has made a selection or entered some text in a required input element is a basic type of validation. When set to "true", this attribute specifies that the element *must* contain a value.
- **requiredMessage**—This specifies the message that should be displayed if the user submits the form without first providing a value for this required element.
- **value**—This specifies the value to display in the field or to be saved into a bean on the server. In this case, the EL expression indicates the bean property that's associated with this field.
- **validatorMessage**—This specifies the message to display if a validator is associated with this **h:inputText** and the data the user enters is invalid.

The messages specified by the **requiredMessage** and **validatorMessage** attributes are displayed in an associated **h:message** element (line 26) when validation fails. The element's **for** attribute specifies the id of the specific element for which messages will be displayed (`nameInputText`), and the **styleClass** attribute specifies the name of a CSS style class that will format the message. For this example, we defined a CSS style sheet, which was inserted into the document's head element at line 12 using the **h:outputStylesheet** element. We placed the style sheet in the `css` library within the `resources` folder. The style sheet contains the following CSS rule:

```
.error
{
    color:red;
}
```

which creates a style class named `error` (the dot indicates that it's a style class) and specifies that any text to which this is applied, such as the error messages, should be red. We use this CSS style for all the **h:message** elements in this example.

Validating the nameInputText Element's Contents

If the user submits the form without a value in the `nameInputText`, the **requiredMessage** "Please enter your name" is displayed in the corresponding **h:message** element. If the user specifies a value for the `nameInputText`, the JSF framework executes the **f:validateLength** validator that's nested in the **h:inputText** element. Here, we check that the name contains no more than 30 characters—as specified by the validator's **maximum attribute**. This might be useful to ensure that a value will fit within a particular database field.

Users can type as much text in the `nameInputText` as they wish. If the name is too long, the **validatorMessage** is displayed in the **h:message** element after the user submits the form. It's also possible to limit the length of user input in an **h:inputText** without

30.7 Validation Using JSF Standard Validators

30_27

using validation by setting its **maxLength** attribute, in which case the element's cursor will not advance beyond the maximum allowable number of characters. This would prevent the user from submitting data that exceeds the length limit.

Second and Third Rows of the h:panelGrid

The next two rows of the **h:panelGrid** have elements similar to those in the first row. In addition to being required elements, the **h:inputText** elements at lines 28–34 and 37–43 are each validated by **h:validateRegex** validators as described next.

Validating the e-Mail Address

The **h:validateRegex** element at lines 32–33 uses the regular expression

```
\w+([-+.']\w+)*@\w+([-.\w+]*\.|\w+([-.\w+])*
```

which indicates that an e-mail address is valid if the part before the @ symbol contains one or more word characters (that is, alphanumeric characters or underscores), followed by zero or more strings comprised of a hyphen, plus sign, period or apostrophe and additional word characters. After the @ symbol, a valid e-mail address must contain one or more groups of word characters potentially separated by hyphens or periods, followed by a required period and another group of one or more word characters potentially separated by hyphens or periods. For example, bob's-personal.email@white.email.com, bob-white@email.com and bob.white@email.com are all valid e-mail addresses. If the address the user enters has an invalid format, the **validatorMessage** (line 31) will be displayed in the corresponding **h:message** element (line 35).

Validating the Phone Number

The **h:validateRegex** element at lines 41–42 uses the regular expression

```
(((\d{3}\)) | (\d{3}-))?\d{3}-\d{4}
```

which indicates that a phone number can contain a three-digit area code either in parentheses and followed by an optional space or without parentheses and followed by a required hyphen. After an optional area code, a phone number must contain three digits, a hyphen and another four digits. For example, (555) 123-4567, 555-123-4567 and 123-4567 are all valid phone numbers. If the phone number the user enters has an invalid format, the **validatorMessage** (line 40) will be displayed in the corresponding **h:message** element (line 44).

Submitting the Form—More Details of the JSF Lifecycle

As we mentioned earlier in this section, when the app receives the initial request, it returns the page shown in Fig. 30.15(a). When a request does not contain any request values, such as those the user enters in a form, the JSF framework simply creates the view and returns it as the response.

The user submits the form to the server by pressing the **Submit h:commandButton** (defined at line 46). Since we did not specify an **action** attribute for this **h:commandButton**, the **action** is configured by default to perform a **postback**—the browser re-requests the page **index.xhtml** and sends the values of the form's fields to the server for processing. Next, the JSF framework performs the validations of all the form elements. If any of the elements is invalid, the framework renders the appropriate error message as part of the response.

30_28 Chapter 30 JavaServer™ Faces Web Apps: Part I

If the values of all the elements are valid, the framework uses the values of the elements to set the properties of the `validateBean`—as specified in the EL expressions in lines 22, 30 and 39. Each property's `set` method is invoked, passing the value of the corresponding element as an argument. The framework then formulates the response to the client. In the response, the form elements are populated with the values of the `validateBean`'s properties (by calling their `get` methods), and the `h:outputText` element at line 47 is populated with the value of the read-only `result` property. The value of this property is determined by the `getResult` method (lines 52–60 of Fig. 30.14), which uses the submitted form data in the string that it returns.

When you execute this app, try submitting the form with no data (Fig. 30.15(b)), with invalid data (Fig. 30.15(c)) and with valid data (Fig. 30.15(d)).

30.8 Session Tracking

Originally, critics accused the Internet and e-business of failing to provide the customized service typically experienced in “brick-and-mortar” stores. To address this problem, businesses established mechanisms by which they could *personalize* users’ browsing experiences, tailoring content to individual users. They tracked each customer’s movement through the Internet and combined the collected data with information the consumer provided, including billing information, personal preferences, interests and hobbies.

Personalization

Personalization enables businesses to communicate effectively with their customers and also helps users locate desired products and services. Companies that provide content of particular interest to users can establish relationships with customers and build on those relationships over time. Furthermore, by targeting consumers with personal offers, recommendations, advertisements, promotions and services, businesses create customer loyalty. Websites can use sophisticated technology to allow visitors to customize home pages to suit their individual needs and preferences. Similarly, online shopping sites often store personal information for customers, tailoring notifications and special offers to their interests. Such services encourage customers to visit sites more frequently and make purchases more regularly.

Privacy

A trade-off exists between personalized business service and protection of privacy. Some consumers embrace tailored content, but others fear the possible adverse consequences if the info they provide to businesses is released or collected by tracking technologies. Consumers and privacy advocates ask: What if the business to which we give personal data sells or gives that information to another organization without our knowledge? What if we do not want our actions on the Internet—a supposedly anonymous medium—to be tracked and recorded by unknown parties? What if unauthorized parties gain access to sensitive private data, such as credit-card numbers or medical history? These are questions that must be addressed by programmers, consumers, businesses and lawmakers alike.

Recognizing Clients

To provide personalized services, businesses must be able to recognize clients when they request information from a site. As we have discussed, the request/response system on

which the web operates is facilitated by HTTP. Unfortunately, HTTP is a *stateless protocol*—it *does not* provide information that would enable web servers to maintain state information regarding particular clients. This means that web servers cannot determine whether a request comes from a particular client or whether the same or different clients generate a series of requests.

To circumvent this problem, sites can provide mechanisms by which they identify individual clients. A session represents a unique client on a website. If the client leaves a site and then returns later, the client will still be recognized as the same user. When the user closes the browser, the session typically ends. To help the server distinguish among clients, each client must identify itself to the server. Tracking individual clients is known as **session tracking**. One popular session-tracking technique uses cookies (discussed in Section 30.8.1); another uses beans that are marked with the `@SessionScoped annotation` (used in Section 30.8.2). Additional session-tracking techniques are beyond this book's scope.

30.8.1 Cookies

Cookies provide you with a tool for personalizing web pages. A cookie is a piece of data stored by web browsers in a small text file on the user's computer. A cookie maintains information about the client during and between browser sessions. The first time a user visits the website, the user's computer might receive a cookie from the server; this cookie is then reactivated each time the user revisits that site. The collected information is intended to be an anonymous record containing data that is used to personalize the user's future visits to the site. For example, cookies in a shopping application might store unique identifiers for users. When a user adds items to an online shopping cart or performs another task resulting in a request to the web server, the server receives a cookie containing the user's unique identifier. The server then uses the unique identifier to locate the shopping cart and perform any necessary processing.

In addition to identifying users, cookies also can indicate users' shopping preferences. When a Web Form receives a request from a client, the Web Form can examine the cookie(s) it sent to the client during previous communications, identify the user's preferences and immediately display products of interest to the client.

Every HTTP-based interaction between a client and a server includes a header containing information either about the request (when the communication is from the client to the server) or about the response (when the communication is from the server to the client). When a Web Form receives a request, the header includes information such as the request type and any cookies that have been sent previously from the server to be stored on the client machine. When the server formulates its response, the header information contains any cookies the server wants to store on the client computer and other information, such as the MIME type of the response.

The **expiration date** of a cookie determines how long the cookie remains on the client's computer. If you do not set an expiration date for a cookie, the web browser maintains the cookie for the duration of the browsing session. Otherwise, the web browser maintains the cookie until the expiration date occurs. Cookies are deleted by the web browser when they **expire**.



Portability Tip 30.1

Users may disable cookies in their web browsers to help ensure their privacy. Such users will experience difficulty using web applications that depend on cookies to maintain state information.

30.8.2 Session Tracking with @SessionScoped Beans

The previous web applications used `@RequestScoped` beans by default—the beans existed only for the duration of each request. In the next application, we use a `@SessionScoped` bean to maintain selections throughout the user’s session. Such a bean is created when a session begins and exists throughout the entire session. A `@SessionScoped` bean can be accessed by all of the app’s pages during the session, and the app server maintains a separate `@SessionScoped` bean for each user. By default a session expires after 30 minutes of inactivity or when the user closes the browser that was used to begin the session. When the session expires, the server discards the bean associated with that session.



Software Engineering Observation 30.2

`@SessionScoped` beans should implement the `Serializable` interface. Websites with heavy traffic often use groups of servers (sometimes hundreds or thousands of them) to respond to requests. Such groups are known as server farms. Server farms often balance the number of requests being handled on each server by moving some sessions to other servers. Making a bean `Serializable` enables the session to be moved properly among servers.

Test-Driving the App

This example consists of a `SelectionsBean` class that is `@SessionScoped` and two pages (`index.xhtml` and `recommendations.xhtml`) that store data in and retrieve data from a `SelectionsBean` object. To understand how these pieces fit together, let’s walk through a sample execution of the app. When you first execute the app, the `index.xhtml` page is displayed. The user selects a topic from a group of radio buttons and submits the form (Fig. 30.16).

Welcome to Sessions!

You have made 0 selection(s)

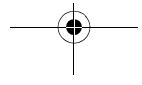
Make a Selection and Press Submit

Java C++ iPhone Android

[Submit](#)

[Click here for book recommendations](#)

Fig. 30.16 | `index.xhtml` after the user has made a selection and is about to submit the form for the first time.



When the form is submitted, the JSF framework creates a `SelectionsBean` object that is specific to this user, stores the selected topic in the bean and returns the `index.xhtml` page. The page now shows how many selections have been made (1) and allows the user to make another selection (Fig. 30.17).

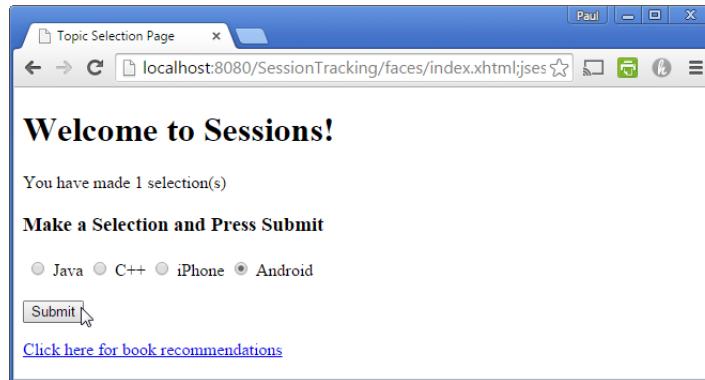


Fig. 30.17 | `index.xhtml` after the user has submitted the form the first time, made another selection and is about to submit the form again.

The user makes a second topic selection and submits the form again. The app stores the selection in this user's existing `SelectionsBean` object and returns the `index.xhtml` page (Fig. 30.18), which shows how many selections have been made so far (2).

At any time, the user can click the link at the bottom of the `index.xhtml` page to open `recommendations.xhtml`, which obtains the information from this user's `SelectionsBean` object and creates a recommended books list (Fig. 30.19) for the user's selected topics.

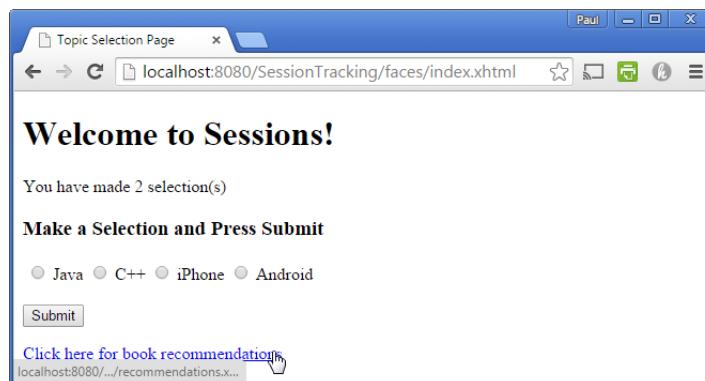


Fig. 30.18 | `index.xhtml` after the user has submitted the form the second time and is about to click the link to the `recommendations.xhtml` page.

30_32 Chapter 30 JavaServer™ Faces Web Apps: Part I

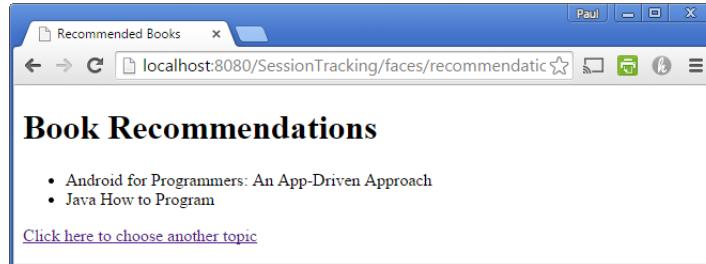


Fig. 30.19 | recommendations.xhtml showing book recommendations for the topic selections made by the user in Figs. 30.17 and 30.18.

@SessionScoped Class SelectionsBean

Class SelectionsBean (Fig. 30.20) uses the @SessionScoped annotation (line 13) to indicate that the server should maintain separate instances of this class for each user session. The class maintains a static HashMap (created at lines 17–18) of topics and their corresponding book titles. We made this object static, because its values can be shared among all SelectionsBean objects. The static initializer block (lines 23–28) specifies the HashMap's key/value pairs. Class SelectionsBean maintains each user's selections in a Set<String> (line 32), which allows only unique keys, so selecting the same topic multiple times does not increase the number of selections.

```

1 // SelectionsBean.java
2 // Manages a user's topic selections
3 package sessiontracking;
4
5 import java.io.Serializable;
6 import java.util.HashMap;
7 import java.util.Set;
8 import java.util.TreeSet;
9 import javax.faces.bean.ManagedBean;
10 import javax.faces.bean.SessionScoped;
11
12 @ManagedBean(name="selectionsBean")
13 @SessionScoped
14 public class SelectionsBean implements Serializable
15 {
16     // map of topics to book titles
17     private static final HashMap< String, String > booksMap =
18         new HashMap< String, String >();
19
20     // initialize booksMap
21     static
22     {
23         booksMap.put("java", "Java How to Program");
24         booksMap.put("cpp", "C++ How to Program");

```

Fig. 30.20 | @SessionScoped SelectionsBean class. (Part I of 2.)

```

25      booksMap.put("iphone",
26          "iPhone for Programmers: An App-Driven Approach");
27      booksMap.put("android",
28          "Android for Programmers: An App-Driven Approach");
29  }
30
31  // stores individual user's selections
32  private Set< String > selections = new TreeSet< String >();
33  private String selection; // stores the current selection
34
35  // return number of selections
36  public int getNumberOfSelections()
37  {
38      return selections.size();
39  }
40
41  // returns the current selection
42  public String getSelection()
43  {
44      return selection;
45  }
46
47  // store user's selection
48  public void setSelection(String topic)
49  {
50      selection = booksMap.get(topic);
51      selections.add(selection);
52  }
53
54  // return the Set of selections
55  public String[] getSelections()
56  {
57      return selections.toArray(new String[selections.size()]);
58  }
59 }
```

Fig. 30.20 | @SessionScoped SelectionsBean class. (Part 2 of 2.)

Methods of Class SelectionsBean

Method `getNumberOfSelections` (lines 36–39) returns the number of topics the user has selected and represents the read-only property `numberOfSelections`. We use this property in the `index.xhtml` document to display the number of selections the user has made so far.

Methods `getSelection` (lines 42–45) and `setSelection` (lines 48–52) represent the read/write `selection` property. When a user makes a selection in `index.xhtml` and submits the form, method `setSelection` looks up the corresponding book title in the `booksMap` (line 50), then stores that title in `selections` (line 51).

Method `getSelections` (lines 55–58) represents the read-only property `selections`, which returns an array of `Strings` containing the book titles for the topics selected by the user so far. When the `recommendations.xhtml` page is requested, it uses the `selections` property to get the list of book titles and display them in the page.

30_34 Chapter 30 JavaServer™ Faces Web Apps: Part I

index.xhtml

The *index.xhtml* document (Fig. 30.21) contains an *h:selectOneRadio* element (lines 19–26) with the options **Java**, **C++**, **iPhone** and **Android**. The user selects a topic by clicking a radio button, then pressing **Submit** to send the selection. As the user makes each selection and submits the form, the *selectionsBean* object's *selection* property is updated and this document is returned. The EL expression at line 15 inserts the number of selections that have been made so far into the page. When the user clicks the *h:outputLink* (lines 29–31) the *recommendations.xhtml* page is requested. The *value* attribute specifies only *recommendations.xhtml*, so the browser assumes that this page is on the same server and at the same location as *index.xhtml*.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- Allow the user to select a topic -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8   xmlns:h="http://java.sun.com/jsf/html"
9   xmlns:f="http://java.sun.com/jsf/core">
10 <h:head>
11   <title>Topic Selection Page</title>
12 </h:head>
13 <h:body>
14   <h1>Welcome to Sessions!</h1>
15   <p>You have made #{selectionsBean.numberOfSelections} selection(s)</p>
16   <h3>Make a Selection and Press Submit</h3>
17   <h:form>
18     <h:selectOneRadio id="topicSelectOneRadio" required="true"
19       requiredMessage="Please choose a topic, then press Submit"
20       value="#{selectionsBean.selection}">
21       <f:selectItem itemValue="java" itemLabel="Java"/>
22       <f:selectItem itemValue="cpp" itemLabel="C++"/>
23       <f:selectItem itemValue="iphone" itemLabel="iPhone"/>
24       <f:selectItem itemValue="android" itemLabel="Android"/>
25     </h:selectOneRadio>
26     <p><h:commandButton value="Submit"/></p>
27   </h:form>
28   <p><h:outputLink value="recommendations.xhtml">
29     Click here for book recommendations
30   </h:outputLink></p>
31   </h:body>
32 </html>
```

Fig. 30.21 | *index.xhtml* allows the user to select a topic.

recommendations.xhtml

When the user clicks the *h:outputLink* in the *index.xhtml* page, the browser requests the *recommendations.xhtml* (Fig. 30.22), which displays book recommendations in an XHTML unordered (bulleted) list (lines 15–19). The *h:outputLink* (lines 20–22) allows the user to return to *index.xhtml* to select additional topics.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- recommendations.xhtml -->
4  <!-- Display recommended books based on the user's selected topics -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8   xmlns:h="http://java.sun.com/jsf/html"
9   xmlns:ui="http://java.sun.com/jsf/facelets">
10 <h:head>
11   <title>Recommended Books</title>
12 </h:head>
13 <h:body>
14   <h1>Book Recommendations</h1>
15   <ul>
16     <ui:repeat value="#{selectionsBean.selections}" var="book">
17       <li>#{book}</li>
18     </ui:repeat>
19   </ul>
20   <p><h:outputLink value="index.xhtml">
21       Click here to choose another topic
22     </h:outputLink></p>
23   </h:body>
24 </html>

```

Fig. 30.22 | recommendations.xhtml displays book recommendations based on the user's selections.

Iterating Through the List of Books

Line 9 enables us to use elements from the **JSF Facelets Tag Library**. This library includes the **ui:repeat** element (lines 16–18), which can be thought of as an enhanced **for** loop that iterates through collections JSF Expression Language. The element inserts its nested element(s) once for each element in a collection. The collection is specified by the **value** attribute's EL expression, which *must* return an array, a **List**, a **java.sql.ResultSet** or an **Object**. If the EL expression does not return an array, a **List** or a **ResultSet**, the **ui:repeat** element inserts its nested element(s) only once for the returned **Object**. In this example, the **ui:repeat** element renders the items returned by the **selectionsBean**'s **selections** property.

The **ui:repeat** element's **var** attribute creates a variable named **book** to which each item in the collection is assigned in sequence. You can use this variable in EL expressions in the nested elements. For example, the expression **#{book}** in line 17 inserts between the **** and **** tags the **String** representation of one item in the collection. You can also use the variable to invoke methods on, or access properties of, the referenced object.

30.9 Wrap-Up

In this chapter, we introduced web application development using JavaServer Faces in NetBeans. We began by discussing the simple HTTP transactions that take place when you request and receive a web page through a web browser. We then discussed the three



30_36 Chapter 30 JavaServer™ Faces Web Apps: Part I

tiers (i.e., the client or top tier, the business logic or middle tier and the information or bottom tier) that comprise most web applications.

You learned how to use NetBeans and the GlassFish Application Server to create, compile and execute web applications. We demonstrated several common JSF components. We also showed how to use validators to ensure that user input satisfies the requirements of an application.

We discussed the benefits of maintaining user information across multiple pages of a website. We then demonstrated how you can include such functionality in a web application using @SessionScoped beans.

In Chapter 31, we continue our discussion of Java web application development with more advanced concepts. You'll learn how to access a database from a JSF web application and how to use AJAX to help web-based applications provide the interactivity and responsiveness that users typically expect of *desktop* applications.

Summary

Section 30.1 Introduction

- Web-based apps create content for web browser clients. This content includes eXtensible HyperText Markup Language (XHTML), JavaScript client-side scripting, Cascading Style Sheets (CSS), images and binary data.
- XHTML is an XML (eXtensible Markup Language) vocabulary that is based on HTML (HyperText Markup Language).
- Java multitier applications are typically implemented using the features of Java Enterprise Edition (Java EE).
- The JavaServer Faces subset of Java EE is a web-application framework (p. 2) for building multitier web apps by extending the framework with your application-specific capabilities. The framework handles the details of receiving client requests and returning responses for you.

Section 30.2 HyperText Transfer Protocol (HTTP) Transactions

- In its simplest form, a web page is nothing more than an XHTML document that describes to a web browser how to display and format the document's information.
- XHTML documents normally contain hyperlinks that link to different pages or to other parts of the same page. When the user clicks a hyperlink, the requested web page loads into the browser.
- Computers that run web-server software (p. 3) make resources available, such as web pages, images, PDF documents and even objects that perform complex tasks.
- The HTTP protocol allows clients and servers to interact and exchange information.
- HTTP uses URLs (Uniform Resource Locators) to locate resources on the Internet.
- A URL contains information that directs a browser to the resource that the user wishes to access.
- The computer that houses and maintains resources is usually referred to as the host (p. 3).
- Host names are translated into IP addresses by domain-name system (DNS) servers (p. 3).
- The path in a URL typically specifies a virtual directory on the server. The server translates the this into a real location, thus hiding a resource's true location.

- When given a URL, a web browser performs an HTTP transaction to retrieve and display the web page at that address.
- HTTP headers (p. 5) provide additional information about the data that will be sent.
- Multipurpose Internet Mail Extensions (MIME; p. 5) is an Internet standard that specifies data formats so that programs can interpret data correctly.
- The two most common HTTP request types are GET and POST (p. 5). A GET request typically asks for a specific resource on a server. A POST request typically posts (or sends) data to a server.
- GET requests and POST requests can both be used to send form data to a web server, yet each request type sends the information differently. A GET request sends information to the server in the URL's query string (p. 6). A POST request sends form data as part of the HTTP message.
- Browsers often cache (p. 6) web pages for quick reloading. If there are no changes between the cached version and the current version on the web, this speeds up your browsing experience.
- An HTTP response can indicate the length of time for which the content remains "fresh." If this amount of time has not been reached, the browser can avoid another request to the server.
- Browsers typically do not cache the server's response to a POST request, because the next POST might not return the same result.

Section 30.3 Multitier Application Architecture

- Web-based applications are multitier (*n*-tier) applications (p. 6) that divide functionality into separate tiers (i.e., logical groupings of functionality). Although tiers can be located on the same computer, the tiers of web-based applications often reside on separate computers.
- The information tier (p. 7) maintains data pertaining to the application.
- The middle tier (p. 7) implements business logic, controller logic and presentation logic to control interactions between the application's clients and the application's data. Business logic in the middle tier enforces business rules and ensures that data is reliable before the server application updates the database or presents the data to users. Business rules dictate how clients can and cannot access application data, and how applications process data.
- The client tier (p. 7) is the application's user interface, which gathers input and displays output. Users interact directly with the application through the user interface. In response to user actions (e.g., clicking a hyperlink), the client tier interacts with the middle tier to make requests and to retrieve data from the information tier.

Section 30.4 Your First JSF Web App

- The JSF web-application framework's Faces servlet (p. 8) processes each requested JSF page so that the server can eventually return a response to the client.

Section 30.4.1 The Default index.xhtml Document: Introducing Facelets

- You present your web app's content in JSF using Facelets (p. 9)—a combination of XHTML markup and JSF markup.
- XHTML (p. 9) specifies the content of a web page that is displayed in a web browser. XHTML separates the presentation of a document from the structure of the document's data.
- Presentation is specified with Cascading Style Sheets (CSS).
- JSF uses the XHTML 1.0 Transitional Recommendation by default. Transitional markup may include some non-CSS formatting, but this is not recommended.
- The starting <html> tag may contain one or more xmlns attributes (p. 9). Each has a name and a value separated by an equal sign (=), and specifies an XML namespace of elements that are used in the document.

30_38 Chapter 30 JavaServer™ Faces Web Apps: Part I

- The attribute `xmlns:h="http://java.sun.com/jsf/html"` specifies a prefix and a URL for JSF's HTML Tag Library (p. 10), allowing the document to use JSF's elements from that library.
- A tag library defines a set of elements that can be inserted into the XHTML markup.
- The elements in the HTML Tag Library generate XHTML elements.
- The `h:head` element (p. 10) defines the XHTML page's head element.
- The document's `title` typically appears in the browser window's title bar, or a browser tab if you have multiple web pages open in the browser at once.
- The `h:body` (p. 10) element represents the page's content.
- A JSF Expression Language (EL; p. 10) expression can interact with a JavaBean to obtain data.

Section 30.4.2 Examining the WebTimeBean Class

- JSF documents typically interact with one or more Java objects to perform the app's tasks.
- JavaBeans objects (p. 10) are instances of classes that follow certain conventions for class design. A JavaBean exposes its data as properties (p. 10). Properties can be read/write, read-only or write-only. To define a read/write property, a JavaBean class provides *set* and *get* methods for that property. A read-only property would have only a *get* method and a write-only property only a *set* method.
- The JavaBeans used in JSF are also POJOs (plain old Java objects; p. 11)
- The JSF framework creates and manages objects of your JavaBean classes for you.
- The `@ManagedBean` annotation (from the package `javax.faces.bean`; p. 11) indicates that the JSF framework should create and manage instances of the class. The parentheses following the annotation contain the optional `name` attribute. If you specify the annotation without the parentheses and the `name` attribute, the JSF framework will use the class name with a lowercase first letter as the default bean name.
- When the Faces servlet encounters an EL expression that accesses a bean property, it automatically invokes the property's *set* or *get* method based on the context in which the property is used.

Section 30.5 Model-View-Controller Architecture of JSF Apps

- JSF applications adhere to the Model-View-Controller (MVC; p. 16) architecture, which separates an application's data (contained in the model) from the graphical presentation (the view) and the processing logic (the controller).
- In JSF, the controller is the JSF framework and is responsible for coordinating interactions between the view and the model. The model contains the application's data (typically in a database), and the view presents the data stored in the model (typically as web pages).

Section 30.6 Common JSF Components

- Elements from the JSF HTML Tag Library are mapped by the JSF framework to a combination of XHTML elements and JavaScript code that enables the browser to render the page.
- The `h:form` element (p. 17) contains the components with which a user interacts to provide data, such as registration or login information, to a JSF app.
- An `h:panelGrid` element (p. 19) organizes elements in an XHTML table. The `columns` attribute specifies the number of columns in the table. The `style` attribute specifies the CSS formatting for the table.
- A `h:graphicImage` (p. 19) displays an image (specified by the `name` attribute) in the page.

- As of JSF 2.0, you add resources (p. 19) that are used throughout your app—such as images, CSS files, JavaScript files—to your web apps by placing them in the app’s **resources** folder within your project’s **Web Pages** node. Each subfolder of resources represents a resource library (p. 19).
- An empty element (p. 20) does not have content between its start and end tags. In such an element, data can be specified as attributes in the start tag. You can close an empty element either by placing a slash immediately preceding the start tag’s right angle bracket or by explicitly writing an end tag.
- An **h:selectOneMenu** element (p. 20) is typically rendered in a web page as a drop-down list. Each item to display appears between the start and end tags of this element as an **f:selectItem** element (from the JSF Core Tag Library; p. 20). An **f:selectItem**’s **itemLabel** is the string that the user will see in the browser, and its **itemValue** is the value that’s returned when you programmatically retrieve the user’s selection from the drop-down list.
- An **h:outputLink** element (p. 20) inserts a hyperlink in a web page. Its **value** attribute specifies the resource that’s requested when a user clicks the hyperlink.
- An **h:selectOneRadio** element (p. 20) provides a series of radio buttons from which the user can select only one.
- An **h:commandButton** element (p. 21) triggers an action when clicked. An **h:commandButton** typically maps to an XHTML **input** element with its **type** attribute set to “**submit**”. Such elements are often used to submit a form’s user input values to the server for processing.

Section 30.7 Validation Using JSF Standard Validators

- Form validation (p. 21) helps prevent processing errors due to incomplete or improperly formatted user input.
- An **f:validateLength** validator (p. 21) determines whether a field contains an acceptable number of characters.
- **f:validateDoubleRange** and **f:validateLongRange** validators (p. 21) determine whether numeric input falls within acceptable ranges.
- An **f:validateRequired** validator (p. 21) determines whether a field contains a value.
- An **f:validateRegex** validator (p. 21) determines whether a field contains a string that matches a specified regular expression pattern.
- An **f:validateBean** validator (p. 21) invokes a bean method that performs custom validation.
- An **h:outputText** element (p. 26) inserts text in a page.
- An input element’s **required** attribute (when set to “**true**”; p. 26) ensures that the user has made a selection or entered some text in a required input element is a basic type of validation.
- An input element’s **requiredMessage** attribute (p. 26) specifies the message that should be displayed if the user submits the form without first providing a value for the required element.
- An input element’s **validatorMessage** attribute (p. 26) specifies the message to display if a validator is associated with the element and the data the user enters is invalid.
- The messages specified by the **requiredMessage** and **validatorMessage** attributes are displayed in an associated **h:message** element (p. 26) when validation fails.
- To limit the length of user input in an **h:inputText**, set its **maxlength** attribute (p. 27)—the element’s cursor will not advance beyond the maximum allowable number of characters.
- In a postback (p. 27), the browser re-requests the page and sends the values of the form’s fields to the server for processing.

30_40 Chapter 30 JavaServer™ Faces Web Apps: Part I

Section 30.8 Session Tracking

- Personalization (p. 28) makes it possible for e-businesses to communicate effectively with their customers and also improves the user's ability to locate desired products and services.
- A trade-off exists between personalized e-business service and protection of privacy. Some consumers embrace the idea of tailored content, but others fear the possible adverse consequences if the information they provide to e-businesses is released or collected by tracking technologies.
- HTTP is a stateless protocol—it does not provide information that would enable web servers to maintain state information regarding particular clients.
- To help the server distinguish among clients, each client must identify itself to the server. Tracking individual clients, known as session tracking, can be achieved in a number of ways. One popular technique uses cookies; another uses the `@SessionScoped` annotation.

Section 30.8.1 Cookies

- A cookie (p. 29) is a piece of data stored in a small text file on the user's computer. A cookie maintains information about the client during and between browser sessions.
- The expiration date (p. 29) of a cookie determines how long the cookie remains on the client's computer. If you do not set an expiration date for a cookie, the web browser maintains the cookie for the duration of the browsing session.

Section 30.8.2 Session Tracking with `@SessionScoped` Beans

- A `@SessionScoped` bean (p. 30) can maintain a user's selections throughout the user's session. Such a bean is created when a session begins and exists throughout the entire session.
- A `@SessionScoped` bean can be accessed by all of the app's pages, and the app server maintains a separate `@SessionScoped` bean for each user.
- By default a session expires after 30 minutes of inactivity or when the user closes the browser that was used to begin the session. When the session expires, the server discards the bean that was associated with that session.
- The `ui:repeat` element (from the JSF Facelets Tag Library; p. 35) inserts its nested element(s) once for each element in a collection. The collection is specified by the `value` attribute's EL expression, which must return an array, a `List`, a `java.sql.ResultSet` or an `Object`.
- The `ui:repeat` element's `var` attribute creates a variable named `book` to which each item in the collection is assigned in sequence.

Self-Review Exercises

30.1 State whether each of the following is *true* or *false*. If *false*, explain why.

- A URL contains information that directs a browser to the resource that the user wishes to access.
- Host names are translated into IP addresses by web servers.
- The path in a URL typically specifies a resource's exact location on the server.
- GET requests and POST requests can both be used to send form data to a web server.
- Browsers typically cache the server's response to a POST request.
- A tag library defines a set of elements that can be inserted into the XHTML markup.
- You must create and manage the JavaBean objects that are used in your JSF web applications.
- When the Faces servlet encounters an EL expression that accesses a bean property, it automatically invokes the property's `set` or `get` method based on the context in which the property is used.

Answers to Self-Review Exercises 30_41

- i) An `h:panelGrid` element organizes elements in an XHTML table.
- j) An `h:selectOneMenu` element is typically rendered in a web page as a set of radio buttons.
- k) The messages specified by an element's `requiredMessage` and `validatorMessage` attributes are displayed in an associated `h:message` element when validation fails.
- l) The HTTP protocol provides information that enables web servers to maintain state information regarding particular clients.
- m) The `ui:repeat` element inserts its nested element(s) once for each element in a collection. The collection can be any `IEnumerable` type.

30.2 Fill in the blanks in each of the following statements:

- a) Java mult-tier applications are typically implemented using the features of _____.
- b) Computers that run _____ software make resources available, such as web pages, images, PDF documents and even objects that perform complex tasks.
- c) The JSF web-application framework's _____ processes each requested JSF page.
- d) A(n) _____ exposes its data as read/write, read-only or write-only properties.
- e) The _____ annotation indicates that the JSF framework should create and manage instances of the class.
- f) A(n) _____ element contains the components with which a user interacts to provide data, such as registration or login information, to a JSF app.
- g) A(n) _____ element triggers an action when clicked.
- h) A(n) _____ validator determines whether a field contains an acceptable number of characters.
- i) A(n) _____ validator determines whether a field contains a string that matches a specified regular expression pattern.
- j) In a(n) _____, the browser re-requests the page and sends the values of the form's fields to the server for processing.
- k) A(n) _____ bean is created when a session begins and exists throughout the entire session.

Answers to Self-Review Exercises

30.1 a) True. b) False. Host names are translated into IP addresses by DNS servers. c) False. The server translates a virtual directory into a real location, thus hiding a resource's true location. d) True. e) False. Browsers typically do not cache the server's response to a POST request, because the next POST might not return the same result. f) True. g) False. The JSF framework creates and manages objects of your JavaBean classes for you. h) True. i) True. j) False. An `h:selectOneRadio` element is rendered as a set of radio buttons. An `h:selectOneMenu` is rendered as a drop-down list. k) True. l) False. HTTP is a stateless protocol that does not provide information that enables web servers to maintain state information regarding particular clients—a separate tracking technology must be used. m) False. A `ui:repeat` element can iterate over only arrays, Lists and ResultSets. For any other object, the elements in a `ui:repeat` element will be inserted once.

30.2 a) Java Enterprise Edition (Java EE). b) web-server. c) Faces servlet. d) JavaBean. e) `@ManagedBean`. f) `h:form`. g) `h:commandButton`. h) `f:validateLength`. i) `f:validateRegex`. j) postback. k) `@SessionScoped`.

Exercises

30.3 (*Registration Form Modification*) Modify the WebComponents application to add functionality to the **Register** button. When the user clicks **Register**, validate all input fields to make sure the user has filled out the form completely and entered a valid email address and phone number. Then,

30_42 Chapter 30 JavaServer™ Faces Web Apps: Part I

JavaServer™ Faces Web Apps: Part 2

31

Objectives

In this chapter you'll learn:

- To access databases from JSF applications.
- The basic principles and advantages of Ajax technology.
- To use Ajax in a JSF web app.



Outline

31_2 Chapter 31 JavaServer™ Faces Web Apps: Part 2

- | | |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| 31.1 Introduction
31.2 Accessing Databases in Web Apps | 31.3 Ajax
31.4 Adding Ajax Functionality to the Validation App
31.5 Wrap-Up |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|

[Summary](#) | [Self-Review Exercise](#) | [Answers to Self-Review Exercise](#) | [Exercises](#)

31.1 Introduction

This chapter continues our discussion of JSF web application development with two additional examples. In the first, we present a simple address book app that retrieves data from and inserts data into a Java DB database. The app allows users to view the existing contacts in the address book and to add new contacts. In the second example, we add so-called *Ajax* capabilities to the *Validation* example from Section 30.7. As you'll learn, Ajax improves application performance and responsiveness. This chapter's examples, like those in Chapter 30, were developed in NetBeans, but similar capabilities are available in other IDEs.

31.2 Accessing Databases in Web Apps

Many web apps access databases to store and retrieve persistent data. In this section, we build an address book web app that uses a Java DB database display contacts from the address book on a web page and to store contacts in the address book. Figure 31.1 shows sample interactions with the *AddressBook* app.

a) Table of addresses displayed when the *AddressBook* app is first requested

First Name	Last Name	Street	City	State	Zip code
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Meg	Gold	1200 Stout St.	Denver	CO	80204
John	Gray	500 South St.	Philadelphia	PA	19147
Bob	Green	5 Bay St.	San Francisco	CA	94133
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
Liz	White	100 5th Ave.	New York	NY	10011

Fig. 31.1 | Sample outputs from the *AddressBook* app. (Part 1 of 2.)

31.2 Accessing Databases in Web Apps

31_3

b) Form for adding an entry

Address Book: Add Entry

First name: Jessica
 Last name: Magenta
 Street: 1 Main Street
 City: SomeCity
 State: FL
 Zipcode: 12345

[Save Address](#) [Return to Addresses](#)

c) Table of addresses updated with the new entry added in Part (b)

Address Book

First Name	Last Name	Street	City	State	Zip code
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Meg	Gold	1200 Stout St.	Denver	CO	80204
John	Gray	500 South St.	Philadelphia	PA	19147
Bob	Green	5 Bay St.	San Francisco	CA	94133
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
Jessica	Magenta	1 Main Street	SomeCity	FL	12345
Liz	White	100 5th Ave.	New York	NY	10011

Fig. 31.1 | Sample outputs from the AddressBook app. (Part 2 of 2.)

If the app's database already contains addresses, the initial request to the app displays those addresses as shown in Fig. 31.1(a)—we populated the database with the sample addresses shown. Clicking **Add Entry** displays the `addentry.xhtml` page for adding an address to the database (Fig. 31.1(b)). Clicking **Save Address** validates the form's fields. If the fields are valid, the JSF app adds the address to the database and returns to the `index.xhtml` page to show the updated list of addresses (Fig. 31.1(c)). This example also introduces the `h:dataTable` element for displaying data in tabular format.

31_4 Chapter 31 JavaServer™ Faces Web Apps: Part 2

The next several sections explain how to build the `AddressBook` application. First, we set up the database (Section 31.2.1). Next, we present class `AddressBean` (Section 31.2.2), which enables the app's Facelets pages to interact with the database. Finally, we present the `index.xhtml` (Section 31.2.3) and `addentry.xhtml` (Section 31.2.4) Facelets pages.

31.2.1 Setting Up the Database

You'll now create the `addressbook` database and populate it with sample data.

Open NetBeans and Ensure that Java DB and GlassFish Are Running

Before you can create the data source in NetBeans, the IDE must be open and the Java DB and GlassFish servers must be running. Perform the following steps:

1. Open the NetBeans IDE.
2. On the **Services** tab, expand the **Databases** node then right click **Java DB**. If **Java DB** is not already running the **Start Server** option will be enabled. In this case, **Select Start** server to launch the Java DB server.
3. On the **Services** tab, expand the **Servers** node then right click **GlassFish Server 4.1** (or the version that was installed with NetBeans). If **GlassFish Server 4.1** is not already running the **Start** option will be enabled. In this case, select **Start** to launch GlassFish.

You may need to wait a few moments for the servers to begin executing.

Creating the Database

In web apps that receive many requests, it's inefficient to create separate database connections for each request. Instead, you use a **connection pool** to allow the server to manage a limited number of database connections and share them among requests. To create a connection pool for this app, perform the following steps:

1. On the **Services** tab, expand the **Databases** node, right click **Java DB** and select **Create Database....** This opens the **Create Java DB Database** dialog.
2. Specify the following values:
 - **Database Name:** addressbook
 - **User Name:** APP
 - **Password:** APP
3. Click **OK** to create the database.

You can specify any **User** name and **Password** you like and should change these as appropriate for real applications. The preceding steps create a new entry in the **Databases** node showing the database's URL (`jdbc:derby://localhost:1527/addressbook`). The database server that provides access to this database resides on the local machine and accepts connections on port 1527.

Populate the addressbook Database with Sample Data

You'll now populate the database with sample data using the `AddressBook.sql` SQL script that's provided with this chapter's examples. NetBeans must be connected to the database to execute SQL statements. If NetBeans is already connected to the database, the icon 

31.2 Accessing Databases in Web Apps 31_5

is displayed next to the database's URL; otherwise, the icon  is displayed. In this case, right click the icon and select **Connect....**

To populate the database with sample data, perform the following steps:

1. Expand the `jdbc:derby://localhost:1527/addressbook` node, then expand the nested **APP** node.
2. Right click the **Tables** node and select **Execute Command...** to open a **SQL** editor tab in NetBeans. In a text editor, open the file `AddressBook.sql` from this chapter's examples folder, then copy the SQL statements and paste them into the **SQL** editor in NetBeans. Next, right click in the **SQL Command** editor and select **Run File**. This will create the **Addresses** table with the sample data in Fig. 31.1(a). [Note: The SQL script attempts to remove the database's **Addresses** table if it already exists. If it doesn't exist, you'll receive an error message in the NetBeans **Output** window, but the table will still be created properly.] Expand the **Tables** node to see the new table. You can view the table's data by right clicking **ADDRESSES** and selecting **View Data....** Notice that we named the columns with all capital letters. We'll be using these names in Section 31.2.3.

31.2.2 Class AddressBean

[Note: To build this app from scratch, use the techniques you learned in Chapter 30 to create a JSF web application named `AddressBook` and add a second Facelets page named `addentry.xhtml` to the app.] Class `AddressBean` (Fig. 31.2) enables the `AddressBook` app to interact with the `addressbook` database. The class provides properties that represent the first name, last name, street, city, state and zip code for an entry in the database. These are used by the `addentry.xhtml` page when adding a new entry to the database. In addition, this class declares a `DataSource` (lines 39–40) for interacting with the database, method `getAddresses` (lines 115–148) for obtaining the list of addresses from the database and method `save` (lines 151–191) for saving a new address into the database. These methods use various JDBC techniques you learned in Chapter 24. [Note: It's also possible to implement this app's data storage using the JPA techniques from Chapter 29.]

```

1 // AddressBean.java
2 // Bean for interacting with the AddressBook database
3 package addressbook;
4
5 import java.io.Serializable;
6 import java.sql.Connection;
7 import java.sql.PreparedStatement;
8 import java.sql.ResultSet;
9 import java.sql.SQLException;
10 import javax.annotation.Resource;
11 import javax.annotation.sql.DataSourceDefinition;
12 import javax.inject.Named;
13 import javax.sql.DataSource;
14 import javax.sql.rowset.CachedRowSet;
15 import javax.sql.rowset.RowSetProvider;
```

Fig. 31.2 | `AddressBean` interacts with a database to store and retrieve addresses. (Part I of 5.)

31.6 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

16 // define the data source
17 @DataSourceDefinition(
18     name = "java:global/jdbc/addressbook",
19     className = "org.apache.derby.jdbc.ClientDataSource",
20     url = "jdbc:derby://localhost:1527/addressbook",
21     databaseName = "addressbook",
22     user = "APP",
23     password = "APP")
24
25
26 @Named("addressBean")
27 @javax.faces.view.ViewScoped
28 public class AddressBean implements Serializable
29 {
30     // instance variables that represent one address
31     private String firstName;
32     private String lastName;
33     private String street;
34     private String city;
35     private String state;
36     private String zipcode;
37
38     // allow the server to inject the DataSource
39     @Resource(lookup="java:global/jdbc/addressbook")
40     DataSource dataSource;
41
42     // get the first name
43     public String getFirstName()
44     {
45         return firstName;
46     }
47
48     // set the first name
49     public void setFirstName(String firstName)
50     {
51         this.firstName = firstName;
52     }
53
54     // get the last name
55     public String getLastname()
56     {
57         return lastName;
58     }
59
60     // set the last name
61     public void setLastName(String lastName)
62     {
63         this.lastName = lastName;
64     }
65
66     // get the street
67     public String getStreet()
68     {

```

Fig. 31.2 | AddressBean interacts with a database to store and retrieve addresses. (Part 2 of 5.)

31.2 Accessing Databases in Web Apps **31_7**

```
69         return street;
70     }
71
72     // set the street
73     public void setStreet(String street)
74     {
75         this.street = street;
76     }
77
78     // get the city
79     public String getCity()
80     {
81         return city;
82     }
83
84     // set the city
85     public void setCity(String city)
86     {
87         this.city = city;
88     }
89
90     // get the state
91     public String getState()
92     {
93         return state;
94     }
95
96     // set the state
97     public void setState(String state)
98     {
99         this.state = state;
100    }
101
102    // get the zipcode
103    public String getZipcode()
104    {
105        return zipcode;
106    }
107
108    // set the zipcode
109    public void setZipcode(String zipcode)
110    {
111        this.zipcode = zipcode;
112    }
113
114    // return a ResultSet of entries
115    public ResultSet getAddresses() throws SQLException
116    {
117        // check whether dataSource was injected by the server
118        if (dataSource == null)
119        {
120            throw new SQLException("Unable to obtain DataSource");
121        }
122    }
123
124    // return a single address
125    public Address getAddress()
126    {
127        Address address = new Address();
128
129        address.setStreet(street);
130        address.setCity(city);
131        address.setState(state);
132        address.setZipcode(zipcode);
133
134        return address;
135    }
136
137    // update the database
138    public void updateAddress()
139    {
140        Connection conn = null;
141        PreparedStatement ps = null;
142
143        try
144        {
145            conn = dataSource.getConnection();
146
147            String sql = "UPDATE address SET street = ? , city = ? , state = ? , zipcode = ? WHERE id = ? ";
148
149            ps = conn.prepareStatement(sql);
150
151            ps.setString(1, street);
152            ps.setString(2, city);
153            ps.setString(3, state);
154            ps.setString(4, zipcode);
155            ps.setInt(5, id);
156
157            int rowsAffected = ps.executeUpdate();
158
159            if (rowsAffected > 0)
160            {
161                System.out.println("Address updated successfully!");
162            }
163        }
164        catch (SQLException e)
165        {
166            e.printStackTrace();
167        }
168        finally
169        {
170            try
171            {
172                if (ps != null)
173                    ps.close();
174
175                if (conn != null)
176                    conn.close();
177            }
178            catch (SQLException e)
179            {
180                e.printStackTrace();
181            }
182        }
183    }
184
185    // insert a new address
186    public void insertAddress()
187    {
188        Connection conn = null;
189        PreparedStatement ps = null;
190
191        try
192        {
193            conn = dataSource.getConnection();
194
195            String sql = "INSERT INTO address (street, city, state, zipcode) VALUES (?, ?, ?, ?)";
196
197            ps = conn.prepareStatement(sql);
198
199            ps.setString(1, street);
200            ps.setString(2, city);
201            ps.setString(3, state);
202            ps.setString(4, zipcode);
203
204            int rowsAffected = ps.executeUpdate();
205
206            if (rowsAffected > 0)
207            {
208                System.out.println("Address inserted successfully!");
209            }
210        }
211        catch (SQLException e)
212        {
213            e.printStackTrace();
214        }
215        finally
216        {
217            try
218            {
219                if (ps != null)
220                    ps.close();
221
222                if (conn != null)
223                    conn.close();
224            }
225            catch (SQLException e)
226            {
227                e.printStackTrace();
228            }
229        }
230    }
231
232    // delete an address
233    public void deleteAddress()
234    {
235        Connection conn = null;
236        PreparedStatement ps = null;
237
238        try
239        {
240            conn = dataSource.getConnection();
241
242            String sql = "DELETE FROM address WHERE id = ? ";
243
244            ps = conn.prepareStatement(sql);
245
246            ps.setInt(1, id);
247
248            int rowsAffected = ps.executeUpdate();
249
250            if (rowsAffected > 0)
251            {
252                System.out.println("Address deleted successfully!");
253            }
254        }
255        catch (SQLException e)
256        {
257            e.printStackTrace();
258        }
259        finally
260        {
261            try
262            {
263                if (ps != null)
264                    ps.close();
265
266                if (conn != null)
267                    conn.close();
268            }
269            catch (SQLException e)
270            {
271                e.printStackTrace();
272            }
273        }
274    }
275
276    // find an address by ID
277    public Address getAddress(int id)
278    {
279        Connection conn = null;
280        PreparedStatement ps = null;
281
282        try
283        {
284            conn = dataSource.getConnection();
285
286            String sql = "SELECT * FROM address WHERE id = ? ";
287
288            ps = conn.prepareStatement(sql);
289
290            ps.setInt(1, id);
291
292            ResultSet rs = ps.executeQuery();
293
294            if (rs.next())
295            {
296                Address address = new Address();
297
298                address.setId(rs.getInt("id"));
299                address.setStreet(rs.getString("street"));
300                address.setCity(rs.getString("city"));
301                address.setState(rs.getString("state"));
302                address.setZipcode(rs.getString("zipcode"));
303
304                return address;
305            }
306        }
307        catch (SQLException e)
308        {
309            e.printStackTrace();
310        }
311        finally
312        {
313            try
314            {
315                if (ps != null)
316                    ps.close();
317
318                if (conn != null)
319                    conn.close();
320            }
321            catch (SQLException e)
322            {
323                e.printStackTrace();
324            }
325        }
326    }
327
328    // find all addresses
329    public List<Address> getAllAddresses()
330    {
331        Connection conn = null;
332        PreparedStatement ps = null;
333
334        try
335        {
336            conn = dataSource.getConnection();
337
338            String sql = "SELECT * FROM address ";
339
340            ps = conn.prepareStatement(sql);
341
342            ResultSet rs = ps.executeQuery();
343
344            List<Address> addresses = new ArrayList<Address>();
345
346            while (rs.next())
347            {
348                Address address = new Address();
349
350                address.setId(rs.getInt("id"));
351                address.setStreet(rs.getString("street"));
352                address.setCity(rs.getString("city"));
353                address.setState(rs.getString("state"));
354                address.setZipcode(rs.getString("zipcode"));
355
356                addresses.add(address);
357            }
358
359            return addresses;
360        }
361        catch (SQLException e)
362        {
363            e.printStackTrace();
364        }
365        finally
366        {
367            try
368            {
369                if (ps != null)
370                    ps.close();
371
372                if (conn != null)
373                    conn.close();
374            }
375            catch (SQLException e)
376            {
377                e.printStackTrace();
378            }
379        }
380    }
381
382    // find an address by street name
383    public Address getAddressByStreet(String street)
384    {
385        Connection conn = null;
386        PreparedStatement ps = null;
387
388        try
389        {
390            conn = dataSource.getConnection();
391
392            String sql = "SELECT * FROM address WHERE street = ? ";
393
394            ps = conn.prepareStatement(sql);
395
396            ps.setString(1, street);
397
398            ResultSet rs = ps.executeQuery();
399
400            if (rs.next())
401            {
402                Address address = new Address();
403
404                address.setId(rs.getInt("id"));
405                address.setStreet(rs.getString("street"));
406                address.setCity(rs.getString("city"));
407                address.setState(rs.getString("state"));
408                address.setZipcode(rs.getString("zipcode"));
409
410                return address;
411            }
412        }
413        catch (SQLException e)
414        {
415            e.printStackTrace();
416        }
417        finally
418        {
419            try
420            {
421                if (ps != null)
422                    ps.close();
423
424                if (conn != null)
425                    conn.close();
426            }
427            catch (SQLException e)
428            {
429                e.printStackTrace();
430            }
431        }
432    }
433
434    // find an address by city name
435    public Address getAddressByCity(String city)
436    {
437        Connection conn = null;
438        PreparedStatement ps = null;
439
440        try
441        {
442            conn = dataSource.getConnection();
443
444            String sql = "SELECT * FROM address WHERE city = ? ";
445
446            ps = conn.prepareStatement(sql);
447
448            ps.setString(1, city);
449
450            ResultSet rs = ps.executeQuery();
451
452            if (rs.next())
453            {
454                Address address = new Address();
455
456                address.setId(rs.getInt("id"));
457                address.setStreet(rs.getString("street"));
458                address.setCity(rs.getString("city"));
459                address.setState(rs.getString("state"));
460                address.setZipcode(rs.getString("zipcode"));
461
462                return address;
463            }
464        }
465        catch (SQLException e)
466        {
467            e.printStackTrace();
468        }
469        finally
470        {
471            try
472            {
473                if (ps != null)
474                    ps.close();
475
476                if (conn != null)
477                    conn.close();
478            }
479            catch (SQLException e)
480            {
481                e.printStackTrace();
482            }
483        }
484    }
485
486    // find an address by state name
487    public Address getAddressByState(String state)
488    {
489        Connection conn = null;
490        PreparedStatement ps = null;
491
492        try
493        {
494            conn = dataSource.getConnection();
495
496            String sql = "SELECT * FROM address WHERE state = ? ";
497
498            ps = conn.prepareStatement(sql);
499
500            ps.setString(1, state);
501
502            ResultSet rs = ps.executeQuery();
503
504            if (rs.next())
505            {
506                Address address = new Address();
507
508                address.setId(rs.getInt("id"));
509                address.setStreet(rs.getString("street"));
510                address.setCity(rs.getString("city"));
511                address.setState(rs.getString("state"));
512                address.setZipcode(rs.getString("zipcode"));
513
514                return address;
515            }
516        }
517        catch (SQLException e)
518        {
519            e.printStackTrace();
520        }
521        finally
522        {
523            try
524            {
525                if (ps != null)
526                    ps.close();
527
528                if (conn != null)
529                    conn.close();
530            }
531            catch (SQLException e)
532            {
533                e.printStackTrace();
534            }
535        }
536    }
537
538    // find an address by zip code
539    public Address getAddressByZipcode(String zipcode)
540    {
541        Connection conn = null;
542        PreparedStatement ps = null;
543
544        try
545        {
546            conn = dataSource.getConnection();
547
548            String sql = "SELECT * FROM address WHERE zipcode = ? ";
549
550            ps = conn.prepareStatement(sql);
551
552            ps.setString(1, zipcode);
553
554            ResultSet rs = ps.executeQuery();
555
556            if (rs.next())
557            {
558                Address address = new Address();
559
560                address.setId(rs.getInt("id"));
561                address.setStreet(rs.getString("street"));
562                address.setCity(rs.getString("city"));
563                address.setState(rs.getString("state"));
564                address.setZipcode(rs.getString("zipcode"));
565
566                return address;
567            }
568        }
569        catch (SQLException e)
570        {
571            e.printStackTrace();
572        }
573        finally
574        {
575            try
576            {
577                if (ps != null)
578                    ps.close();
579
580                if (conn != null)
581                    conn.close();
582            }
583            catch (SQLException e)
584            {
585                e.printStackTrace();
586            }
587        }
588    }
589
590    // find an address by ID and street name
591    public Address getAddressByIdAndStreet(int id, String street)
592    {
593        Connection conn = null;
594        PreparedStatement ps = null;
595
596        try
597        {
598            conn = dataSource.getConnection();
599
600            String sql = "SELECT * FROM address WHERE id = ? AND street = ? ";
601
602            ps = conn.prepareStatement(sql);
603
604            ps.setInt(1, id);
605            ps.setString(2, street);
606
607            ResultSet rs = ps.executeQuery();
608
609            if (rs.next())
610            {
611                Address address = new Address();
612
613                address.setId(rs.getInt("id"));
614                address.setStreet(rs.getString("street"));
615                address.setCity(rs.getString("city"));
616                address.setState(rs.getString("state"));
617                address.setZipcode(rs.getString("zipcode"));
618
619                return address;
620            }
621        }
622        catch (SQLException e)
623        {
624            e.printStackTrace();
625        }
626        finally
627        {
628            try
629            {
630                if (ps != null)
631                    ps.close();
632
633                if (conn != null)
634                    conn.close();
635            }
636            catch (SQLException e)
637            {
638                e.printStackTrace();
639            }
640        }
641    }
642
643    // find an address by ID and city name
644    public Address getAddressByIdAndCity(int id, String city)
645    {
646        Connection conn = null;
647        PreparedStatement ps = null;
648
649        try
650        {
651            conn = dataSource.getConnection();
652
653            String sql = "SELECT * FROM address WHERE id = ? AND city = ? ";
654
655            ps = conn.prepareStatement(sql);
656
657            ps.setInt(1, id);
658            ps.setString(2, city);
659
660            ResultSet rs = ps.executeQuery();
661
662            if (rs.next())
663            {
664                Address address = new Address();
665
666                address.setId(rs.getInt("id"));
667                address.setStreet(rs.getString("street"));
668                address.setCity(rs.getString("city"));
669                address.setState(rs.getString("state"));
670                address.setZipcode(rs.getString("zipcode"));
671
672                return address;
673            }
674        }
675        catch (SQLException e)
676        {
677            e.printStackTrace();
678        }
679        finally
680        {
681            try
682            {
683                if (ps != null)
684                    ps.close();
685
686                if (conn != null)
687                    conn.close();
688            }
689            catch (SQLException e)
690            {
691                e.printStackTrace();
692            }
693        }
694    }
695
696    // find an address by ID and state name
697    public Address getAddressByIdAndState(int id, String state)
698    {
699        Connection conn = null;
700        PreparedStatement ps = null;
701
702        try
703        {
704            conn = dataSource.getConnection();
705
706            String sql = "SELECT * FROM address WHERE id = ? AND state = ? ";
707
708            ps = conn.prepareStatement(sql);
709
710            ps.setInt(1, id);
711            ps.setString(2, state);
712
713            ResultSet rs = ps.executeQuery();
714
715            if (rs.next())
716            {
717                Address address = new Address();
718
719                address.setId(rs.getInt("id"));
720                address.setStreet(rs.getString("street"));
721                address.setCity(rs.getString("city"));
722                address.setState(rs.getString("state"));
723                address.setZipcode(rs.getString("zipcode"));
724
725                return address;
726            }
727        }
728        catch (SQLException e)
729        {
730            e.printStackTrace();
731        }
732        finally
733        {
734            try
735            {
736                if (ps != null)
737                    ps.close();
738
739                if (conn != null)
740                    conn.close();
741            }
742            catch (SQLException e)
743            {
744                e.printStackTrace();
745            }
746        }
747    }
748
749    // find an address by ID and zip code
750    public Address getAddressByIdAndZipcode(int id, String zipcode)
751    {
752        Connection conn = null;
753        PreparedStatement ps = null;
754
755        try
756        {
757            conn = dataSource.getConnection();
758
759            String sql = "SELECT * FROM address WHERE id = ? AND zipcode = ? ";
760
761            ps = conn.prepareStatement(sql);
762
763            ps.setInt(1, id);
764            ps.setString(2, zipcode);
765
766            ResultSet rs = ps.executeQuery();
767
768            if (rs.next())
769            {
770                Address address = new Address();
771
772                address.setId(rs.getInt("id"));
773                address.setStreet(rs.getString("street"));
774                address.setCity(rs.getString("city"));
775                address.setState(rs.getString("state"));
776                address.setZipcode(rs.getString("zipcode"));
777
778                return address;
779            }
780        }
781        catch (SQLException e)
782        {
783            e.printStackTrace();
784        }
785        finally
786        {
787            try
788            {
789                if (ps != null)
790                    ps.close();
791
792                if (conn != null)
793                    conn.close();
794            }
795            catch (SQLException e)
796            {
797                e.printStackTrace();
798            }
799        }
800    }
801
802    // find an address by street name and city name
803    public Address getAddressByStreetAndCity(String street, String city)
804    {
805        Connection conn = null;
806        PreparedStatement ps = null;
807
808        try
809        {
810            conn = dataSource.getConnection();
811
812            String sql = "SELECT * FROM address WHERE street = ? AND city = ? ";
813
814            ps = conn.prepareStatement(sql);
815
816            ps.setString(1, street);
817            ps.setString(2, city);
818
819            ResultSet rs = ps.executeQuery();
820
821            if (rs.next())
822            {
823                Address address = new Address();
824
825                address.setId(rs.getInt("id"));
826                address.setStreet(rs.getString("street"));
827                address.setCity(rs.getString("city"));
828                address.setState(rs.getString("state"));
829                address.setZipcode(rs.getString("zipcode"));
830
831                return address;
832            }
833        }
834        catch (SQLException e)
835        {
836            e.printStackTrace();
837        }
838        finally
839        {
840            try
841            {
842                if (ps != null)
843                    ps.close();
844
845                if (conn != null)
846                    conn.close();
847            }
848            catch (SQLException e)
849            {
850                e.printStackTrace();
851            }
852        }
853    }
854
855    // find an address by street name and state name
856    public Address getAddressByStreetAndState(String street, String state)
857    {
858        Connection conn = null;
859        PreparedStatement ps = null;
860
861        try
862        {
863            conn = dataSource.getConnection();
864
865            String sql = "SELECT * FROM address WHERE street = ? AND state = ? ";
866
867            ps = conn.prepareStatement(sql);
868
869            ps.setString(1, street);
870            ps.setString(2, state);
871
872            ResultSet rs = ps.executeQuery();
873
874            if (rs.next())
875            {
876                Address address = new Address();
877
878                address.setId(rs.getInt("id"));
879                address.setStreet(rs.getString("street"));
880                address.setCity(rs.getString("city"));
881                address.setState(rs.getString("state"));
882                address.setZipcode(rs.getString("zipcode"));
883
884                return address;
885            }
886        }
887        catch (SQLException e)
888        {
889            e.printStackTrace();
890        }
891        finally
892        {
893            try
894            {
895                if (ps != null)
896                    ps.close();
897
898                if (conn != null)
899                    conn.close();
900            }
901            catch (SQLException e)
902            {
903                e.printStackTrace();
904            }
905        }
906    }
907
908    // find an address by street name and zip code
909    public Address getAddressByStreetAndZipcode(String street, String zipcode)
910    {
911        Connection conn = null;
912        PreparedStatement ps = null;
913
914        try
915        {
916            conn = dataSource.getConnection();
917
918            String sql = "SELECT * FROM address WHERE street = ? AND zipcode = ? ";
919
920            ps = conn.prepareStatement(sql);
921
922            ps.setString(1, street);
923            ps.setString(2, zipcode);
924
925            ResultSet rs = ps.executeQuery();
926
927            if (rs.next())
928            {
929                Address address = new Address();
930
931                address.setId(rs.getInt("id"));
932                address.setStreet(rs.getString("street"));
933                address.setCity(rs.getString("city"));
934                address.setState(rs.getString("state"));
935                address.setZipcode(rs.getString("zipcode"));
936
937                return address;
938            }
939        }
940        catch (SQLException e)
941        {
942            e.printStackTrace();
943        }
944        finally
945        {
946            try
947            {
948                if (ps != null)
949                    ps.close();
950
951                if (conn != null)
952                    conn.close();
953            }
954            catch (SQLException e)
955            {
956                e.printStackTrace();
957            }
958        }
959    }
960
961    // find an address by city name and state name
962    public Address getAddressByCityAndState(String city, String state)
963    {
964        Connection conn = null;
965        PreparedStatement ps = null;
966
967        try
968        {
969            conn = dataSource.getConnection();
970
971            String sql = "SELECT * FROM address WHERE city = ? AND state = ? ";
972
973            ps = conn.prepareStatement(sql);
974
975            ps.setString(1, city);
976            ps.setString(2, state);
977
978            ResultSet rs = ps.executeQuery();
979
980            if (rs.next())
981            {
982                Address address = new Address();
983
984                address.setId(rs.getInt("id"));
985                address.setStreet(rs.getString("street"));
986                address.setCity(rs.getString("city"));
987                address.setState(rs.getString("state"));
988                address.setZipcode(rs.getString("zipcode"));
989
990                return address;
991            }
992        }
993        catch (SQLException e)
994        {
995            e.printStackTrace();
996        }
997        finally
998        {
999            try
1000            {
1001                if (ps != null)
1002                    ps.close();
1003
1004                if (conn != null)
1005                    conn.close();
1006            }
1007            catch (SQLException e)
1008            {
1009                e.printStackTrace();
1010            }
1011        }
1012    }
1013
1014    // find an address by city name and zip code
1015    public Address getAddressByCityAndZipcode(String city, String zipcode)
1016    {
1017        Connection conn = null;
1018        PreparedStatement ps = null;
1019
1020        try
1021        {
1022            conn = dataSource.getConnection();
1023
1024            String sql = "SELECT * FROM address WHERE city = ? AND zipcode = ? ";
1025
1026            ps = conn.prepareStatement(sql);
1027
1028            ps.setString(1, city);
1029            ps.setString(2, zipcode);
1030
1031            ResultSet rs = ps.executeQuery();
1032
1033            if (rs.next())
1034            {
1035                Address address = new Address();
1036
1037                address.setId(rs.getInt("id"));
1038                address.setStreet(rs.getString("street"));
1039                address.setCity(rs.getString("city"));
1040                address.setState(rs.getString("state"));
1041                address.setZipcode(rs.getString("zipcode"));
1042
1043                return address;
1044            }
1045        }
1046        catch (SQLException e)
1047        {
1048            e.printStackTrace();
1049        }
1050        finally
1051        {
1052            try
1053            {
1054                if (ps != null)
1055                    ps.close();
1056
1057                if (conn != null)
1058                    conn.close();
1059            }
1060            catch (SQLException e)
1061            {
1062                e.printStackTrace();
1063            }
1064        }
1065    }
1066
1067    // find an address by city name and street name
1068    public Address getAddressByCityAndStreet(String city, String street)
1069    {
1070        Connection conn = null;
1071        PreparedStatement ps = null;
1072
1073        try
1074        {
1075            conn = dataSource.getConnection();
1076
1077            String sql = "SELECT * FROM address WHERE city = ? AND street = ? ";
1078
1079            ps = conn.prepareStatement(sql);
1080
1081            ps.setString(1, city);
1082            ps.setString(2, street);
1083
1084            ResultSet rs = ps.executeQuery();
1085
1086            if (rs.next())
1087            {
1088                Address address = new Address();
1089
1090                address.setId(rs.getInt("id"));
1091                address.setStreet(rs.getString("street"));
1092                address.setCity(rs.getString("city"));
1093                address.setState(rs.getString("state"));
1094                address.setZipcode(rs.getString("zipcode"));
1095
1096                return address;
1097            }
1098        }
1099        catch (SQLException e)
1100        {
1101            e.printStackTrace();
1102        }
1103        finally
1104        {
1105            try
1106            {
1107                if (ps != null)
1108                    ps.close();
1109
1110                if (conn != null)
1111                    conn.close();
1112            }
1113            catch (SQLException e)
1114            {
1115                e.printStackTrace();
1116            }
1117        }
1118    }
1119
1120    // find an address by state name and zip code
1121    public Address getAddressByStateAndZipcode(String state, String zipcode)
1122    {
1123        Connection conn = null;
1124        PreparedStatement ps = null;
1125
1126        try
1127        {
1128            conn = dataSource.getConnection();
1129
1130            String sql = "SELECT * FROM address WHERE state = ? AND zipcode = ? ";
1131
1132            ps = conn.prepareStatement(sql);
1133
1134            ps.setString(1, state);
1135            ps.setString(2, zipcode);
1136
1137            ResultSet rs = ps.executeQuery();
1138
1139            if (rs.next())
1140            {
1141                Address address = new Address();
1142
1143                address.setId(rs.getInt("id"));
1144                address.setStreet(rs.getString("street"));
1145                address.setCity(rs.getString("city"));
1146                address.setState(rs.getString("state"));
1147                address.setZipcode(rs.getString("zipcode"));
1148
1149                return address;
1150            }
1151        }
1152        catch (SQLException e)
1153        {
1154            e.printStackTrace();
1155        }
1156        finally
1157        {
1158            try
1159            {
1160                if (ps != null)
1161                    ps.close();
1162
1163                if (conn != null)
1164                    conn.close();
1165            }
1166            catch (SQLException e)
1167            {
1168                e.printStackTrace();
1169            }
1170        }
1171    }
1172
1173    // find an address by state name and street name
1174    public Address getAddressByStateAndStreet(String state, String street)
1175    {
1176        Connection conn = null;
1177        PreparedStatement ps = null;
1178
1179        try
1180        {
1181            conn = dataSource.getConnection();
1182
1183            String sql = "SELECT * FROM address WHERE state = ? AND street = ? ";
1184
1185            ps = conn.prepareStatement(sql);
1186
1187            ps.setString(1, state);
1188            ps.setString(2, street);
1189
1190            ResultSet rs = ps.executeQuery();
1191
1192            if (rs.next())
1193            {
1194                Address address = new Address();
1195
1196                address.setId(rs.getInt("id"));
1197                address.setStreet(rs.getString("street"));
1198                address.setCity(rs.getString("city"));
1199                address.setState(rs.getString("state"));
1200                address.setZipcode(rs.getString("zipcode"));
1201
1202                return address;
1203            }
1204        }
1205        catch (SQLException e)
1206        {
1207            e.printStackTrace();
1208        }
1209        finally
1210        {
1211            try
1212            {
1213                if (ps != null)
1214                    ps.close();
1215
1216                if (conn != null)
1217                    conn.close();
1218            }
1219            catch (SQLException e)
1220            {
1221                e.printStackTrace();
1222            }
1223        }
1224    }
1225
1226    // find an address by state name and city name
1227    public Address getAddressByStateAndCity(String state, String city)
1228    {
1229        Connection conn = null;
1230        PreparedStatement ps = null;
1231
1232        try
1233        {
1234            conn = dataSource.getConnection();
1235
1236            String sql = "SELECT * FROM address WHERE state = ? AND city = ? ";
1237
1238            ps = conn.prepareStatement(sql);
1239
1240            ps.setString(1, state);
1241            ps.setString(2, city);
1242
1243            ResultSet rs = ps.executeQuery();
1244
1245            if (rs.next())
1246            {
1247                Address address = new Address();
1248
1249                address.setId(rs.getInt("id"));
1250                address.setStreet(rs.getString("street"));
1251                address.setCity(rs.getString("city"));
1252                address.setState(rs.getString("state"));
1253                address.setZipcode(rs.getString("zipcode"));
1254
1255                return address;
1256            }
1257        }
1258        catch (SQLException e)
1259        {
1260            e.printStackTrace();
1261        }
1262        finally
1263        {
1264            try
1265            {
1266                if (ps != null)
1267                    ps.close();
1268
1269                if (conn != null)
1270                    conn.close();
1271            }
1272            catch (SQLException e)
1273            {
1274                e.printStackTrace();
1275            }
1276        }
1277    }
1278
1279    // find an address by state name and zip code
1280    public Address getAddressByStateAndZipcode(String state, String zipcode)
1281    {
1282        Connection conn = null;
1283        PreparedStatement ps = null;
1284
1285        try
1286        {
1287            conn = dataSource.getConnection();
1288
1289            String sql = "SELECT * FROM address WHERE state = ? AND zipcode = ? ";
1290
1291            ps = conn.prepareStatement(sql);
1292
1293            ps.setString(1, state);
1294            ps.setString(2, zipcode);
1295
1296            ResultSet rs = ps.executeQuery();
1297
1298            if (rs.next())
1299            {
1300                Address address = new Address();
1301
1302                address.setId(rs.getInt("id"));
1303                address.setStreet(rs.getString("street"));
1304                address.setCity(rs.getString("city"));
1305                address.setState(rs.getString("state"));
1306                address.setZipcode(rs.getString("zipcode"));
1307
1308                return address;
1309            }
1310        }
1311        catch (SQLException e)
1312        {
1313            e.printStackTrace();
1314        }
1315        finally
1316        {
1317            try
1318            {
1319                if (ps != null)
1320                    ps.close();
1321
1322                if (conn != null)
1323                    conn.close();
1324            }
1325            catch (SQLException e)
1326            {
1327                e.printStackTrace();
1328            }
1329        }
1330    }
1331
1332    // find an address by zip code and city name
1333    public Address getAddressByZipcodeAndCity(String zipcode, String city)
1334    {
1335        Connection conn = null;
1336        PreparedStatement ps = null;
1337
1338        try
1339        {
1340            conn = dataSource.getConnection();
1341
1342            String sql = "SELECT * FROM address WHERE zipcode = ? AND city = ? ";
1343
1344            ps = conn.prepareStatement(sql);
1345
1346            ps.setString(1, zipcode);
1347            ps.setString(2, city);
1348
1349            ResultSet rs = ps.executeQuery();
1350
1351            if (rs.next())
1352            {
1353                Address address = new Address();
1354
1355                address.setId(rs.getInt("id"));
1356                address.setStreet(rs.getString("street"));
1357                address.setCity(rs.getString("city"));
1358                address.setState(rs.getString("state"));
1359                address.setZipcode(rs.getString("zipcode"));
1360
1361                return address;
1362            }
1363        }
1364        catch (SQLException e)
1365        {
1366            e.printStackTrace();
1367        }
1368        finally
1369        {
1370            try
1371            {
1372                if (ps != null)
1373                    ps.close();
1374
1375                if (conn != null)
1376                    conn.close();
1377            }
1378            catch (SQLException e)
1379            {
1380                e.printStackTrace();
1381            }
1382        }
1383    }
1384
1385    // find an address by zip code and street name
1386    public Address getAddressByZipcodeAndStreet(String zipcode, String street)
1387    {
1388        Connection conn = null;
1389        PreparedStatement ps = null;
1390
1391        try
1392        {
1393            conn = dataSource.getConnection();
1394
1395            String sql = "SELECT * FROM address WHERE zipcode = ? AND street = ? ";
1396
1397            ps = conn.prepareStatement(sql);
1398
1399            ps.setString(1, zipcode);
1400            ps.setString(2, street);
1401
1402            ResultSet rs = ps.executeQuery();
1403
1404            if (rs.next())
1405            {
1406                Address address = new Address();
1407
1408                address.setId(rs.getInt("id"));
1409                address.setStreet(rs.getString("street"));
1410                address.setCity(rs.getString("city"));
1411                address.setState(rs.getString("state"));
1412                address.setZipcode(rs.getString("zipcode"));
1413
1414                return address;
1415            }
1416        }
1417        catch (SQLException e)
1418        {
1419            e.printStackTrace();
1420        }
1421        finally
1422        {
1423            try
1424            {
1425                if (ps != null)
1426                    ps.close();
1427
1428                if (conn != null)
1429                    conn.close();
1430            }
1431            catch (SQLException e)
1432            {
1433                e.printStackTrace();
1434            }
1435        }
1436    }
1437
1438    // find an address by zip code and state name
1439    public Address getAddressByZipcodeAndState(String zipcode, String state)
1440    {
1441        Connection conn = null;
1442        PreparedStatement ps = null;
1443
1444        try
1445        {
1446            conn = dataSource.getConnection();
1447
1448            String sql = "SELECT * FROM address WHERE zipcode = ? AND state = ? ";
1449
1450            ps = conn.prepareStatement(sql);
1451
1452            ps.setString(1, zipcode);
1453            ps.setString(2, state);
1454
1455            ResultSet rs = ps.executeQuery();
1456
1457            if (rs.next())
1458            {
1459                Address address = new Address();
1460
1461                address.setId(rs.getInt("id"));
1462                address.setStreet(rs.getString("street"));
1463                address.setCity(rs.getString("city"));
1464                address.setState(rs.getString("state"));
1465                address.setZipcode(rs.getString("zipcode"));
1466
1467                return address;
1468            }
1469        }
1470        catch (SQLException e)
1471        {
1472            e.printStackTrace();
1473        }
1474        finally
1475        {
1476            try
1477            {
1478                if (ps != null)
1479                    ps.close();
1480
1481                if (conn != null)
1482                    conn.close();
1483            }
1484            catch (SQLException e)
1485            {
1486                e.printStackTrace();
1487            }
1488        }
1489    }
1490
1491    // find an address by zip code and state name and city name
1492    public Address getAddressByZipcodeAndStateAndCity(String zipcode, String state, String city)
1493    {
1494        Connection conn = null;
1495        PreparedStatement ps = null;
1496
1497        try
1498        {
1499            conn = dataSource.getConnection();
1500
1501            String sql = "SELECT * FROM address WHERE zipcode = ? AND state = ? AND city = ? ";
1502
1503            ps = conn.prepareStatement(sql);
1504
1505            ps.setString(1, zipcode);
1506            ps.setString(2, state);
1507            ps.setString(3, city);
1508
1
```

31.8 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

122     // obtain a connection from the connection pool
123     Connection connection = dataSource.getConnection();
124
125     // check whether connection was successful
126     if (connection == null)
127     {
128         throw new SQLException("Unable to connect to DataSource");
129     }
130
131     try
132     {
133         // create a PreparedStatement to insert a new address book entry
134         PreparedStatement getAddresses = connection.prepareStatement(
135             "SELECT FIRSTNAME, LASTNAME, STREET, CITY, STATE, ZIP " +
136             "FROM ADDRESSES ORDER BY LASTNAME, FIRSTNAME");
137
138         CachedRowSet rowSet =
139             RowSetProvider.newFactory().createCachedRowSet();
140         rowSet.populate(getAddresses.executeQuery());
141         return rowSet;
142     }
143     finally
144     {
145         connection.close(); // return this connection to pool
146     }
147 }
148
149 // save a new address book entry
150 public String save() throws SQLException
151 {
152     // check whether dataSource was injected by the server
153     if (dataSource == null)
154     {
155         throw new SQLException("Unable to obtain DataSource");
156     }
157
158     // obtain a connection from the connection pool
159     Connection connection = dataSource.getConnection();
160
161     // check whether connection was successful
162     if (connection == null)
163     {
164         throw new SQLException("Unable to connect to DataSource");
165     }
166
167     try
168     {
169         // create a PreparedStatement to insert a new address book entry
170         PreparedStatement addEntry =
171             connection.prepareStatement("INSERT INTO ADDRESSES " +
172                 "(FIRSTNAME,LASTNAME,STREET,CITY,STATE,ZIP)" +
173                 " VALUES (?, ?, ?, ?, ?, ?)");

```

Fig. 31.2 | AddressBean interacts with a database to store and retrieve addresses. (Part 4 of 5.)

31.2 Accessing Databases in Web Apps 31_9

```

175      // specify the PreparedStatement's arguments
176      addEntry.setString(1, getFirstName());
177      addEntry.setString(2, getLastName());
178      addEntry.setString(3, getStreet());
179      addEntry.setString(4, getCity());
180      addEntry.setString(5, getState());
181      addEntry.setString(6, getZipcode());
182
183
184      addEntry.executeUpdate(); // insert the entry
185      return "index"; // go back to index.xhtml page
186    }
187    finally
188    {
189      connection.close(); // return this connection to pool
190    }
191  }
192 }
```

Fig. 31.2 | AddressBean interacts with a database to store and retrieve addresses. (Part 5 of 5.)

Defining a Data Source with the Annotation @DataSourceDefinition

To connect to the addressbook database from a web app, you must configure a **data source name** that will be used to locate the database. Lines 18–24

```

@DataSourceDefinition(
    name = "java:global/jdbc/addressbook",
    className = "org.apache.derby.jdbc.ClientDataSource",
    url = "jdbc:derby://localhost:1527/addressbook",
    databaseName = "addressbook",
    user = "APP",
    password = "APP")
```

use Java EE 7's **@DataSourceDefinition** annotation to create a data source name for the addressbook database. Here we specified the following attributes:

- **name**—The JNDI (Java Naming and Directory Interface) name we'll use to look up the data source. JNDI is a technology for locating application components (such as databases) in a distributed application (such as a multitier web application).
- **className**—The **DataSource** subclass. An object of this class will be used to interact with the database. A **DataSource** (package `javax.sql`) enables a web application to obtain a **Connection** to a database. **ClientDataSource** is one of several **DataSource** subclasses provided by Java DB. Apps that are expected to manage many connections at once would typically use **ClientConnectionPoolDataSource** or **ClientXADataSource**.
- **url**—The URL for connecting to the database. This is the database URL is specified in the NetBeans **Services** tab's **Databases > Java DB** node.
- **databaseName**—The database's name.
- **user**—The username for logging into the database.
- **password**—The password for logging into the database.¹



31_10 Chapter 31 JavaServer™ Faces Web Apps: Part 2

Though we do not do so here, the `@DataSourceDefinition` annotation also can create the database, by specifying the attribute

```
properties = {"createDatabase=create"}
```

The app could then create the database's table(s) programmatically. We manually created the database in advance so we could prepopulate it with sample address data.

Class AddressBean's Annotations—`@Named` and `@javax.faces.view.ViewScoped`

In Chapter 30, we introduced the `@ManagedBean` annotation (from the package `javax.faces.bean`) to indicate that the JSF framework should create and manage the JavaBean object(s) used in the application. `@ManagedBean` is deprecated in Java EE 7 and Contexts and Dependency Injection (CDI) should be used instead. Switching to CDI simply requires changing from JSF's `@ManagedBean` annotation to CDI's `@Named annotation` (line 26):

```
@Named("addressBean")
```

As with `@ManagedBean`, if you do not specify a name in parentheses, the JavaBean object's variable name will be the JavaBean class's name with a lowercase first letter.

We also added the annotation

```
@javax.faces.view.ViewScoped
```

to indicate that CDI should manage this JavaBean's lifetime, based on the JSF view that first referenced the JavaBean. A `ViewScoped` JavaBean's class must be `Serializable` (as indicated in line 28).

Injecting the `DataSource` into Class `AddressBean`

Lines 39–40 use the `@Resource` annotation to inject a `DataSource` object into the `AddressBean`. The annotation's `lookup` attribute specifies the JNDI name for the data source we created in lines 18–24. The `@Resource` annotation enables the server (GlassFish in our case) to hide all the complex details of setting up a `DataSource` object that can interact with the `addressbook` database. The server creates a `DataSource` for you—an object of the type you specified in the `@DataSourceDefinition`—and assigns the `DataSource` object to the annotated variable declared at line 40. You can now trivially obtain a `Connection` for interacting with the database.

AddressBean Method `getAddresses`

Method `getAddresses` (lines 115–148) is called when the `index.xhtml` page is requested. The method returns a list of addresses for display in the page (Section 31.2.3). First, we check whether variable `dataSource` is `null` (lines 118–121), which would indicate that the server was unable to create the `DataSource` object. If the `DataSource` was created successfully, we use it to obtain a `Connection` to the database (line 124). Next, we check whether variable `connection` is `null` (lines 127–130), which would indicate that we were unable to connect. If the connection was successful, lines 135–142 get the set of addresses from the database and return them.

1. Future versions of Java EE might include password aliases to hide the database's password for additional security.

31.2 Accessing Databases in Web Apps 31_11

The `PreparedStatement` at lines 135–137 obtains all the addresses. Because database connections are a limited resources, you should use and close them quickly in your web apps. For this reason, we create a `CachedRowSet` and populate it with the `ResultSet` returned by the `PreparedStatement`'s `executeQuery` method (lines 139–141). We then return the `CachedRowSet` (a disconnected RowSet) for use in the `index.xhtml` page (line 142) and close the connection object (line 146) in the `finally` block.

AddressBean Method save

Method `save` (lines 151–191) stores a new address in the database (Section 31.2.4). This occurs when the user submits the `addentry.xhtml` form—assuming the form's fields validate successfully. As in `getAddresses`, we ensure that the `DataSource` is not `null`, then obtain the `Connection` object and ensure that its not `null`. Lines 171–174 create a `PreparedStatement` for inserting a new record in the database. Lines 177–182 specify the values for each of the parameters in the `PreparedStatement`. Line 184 then executes the `PreparedStatement` to insert the new record. Line 185 returns the string "index", which as you'll see in Section 31.2.4 causes the app to display the `index.xhtml` page again.

31.2.3 `index.xhtml` Facelets Page

`index.xhtml` (Fig. 31.3) is the default web page for the `AddressBook` app. When this page is requested, it obtains the list of addresses from the `AddressBean` and displays them in tabular format using an `h:dataTable` element. The user can click the **Add Entry** button (line 17) to view the `addentry.xhtml` page. Recall that the default action for an `h:commandButton` is to submit a form. In this case, we specify the button's **action attribute** with the value "addentry". The JSF framework assumes this is a page in the app, appends `.xhtml` extension to the action attribute's value and returns the `addentry.xhtml` page to the client browser.

The h:dataTable Element

The `h:dataTable` element (lines 19–46) inserts tabular data into a page. We discuss only the attributes and nested elements that we use here. For more details on this element, its attributes and other JSF tag library elements, visit [bit.ly/JSF2TagLibraryReference](#).

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.html -->
4  <!-- Displays an h:dataTable of the addresses in the address book -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8   xmlns:h="http://java.sun.com/jsf/html"
9   xmlns:f="http://java.sun.com/jsf/core">
10 <h:head>
11   <title>Address Book</title>
12   <h:outputStylesheet name="style.css" library="css"/>
13 </h:head>
```

Fig. 31.3 | Displays an `h:dataTable` of the addresses in the address book. (Part 1 of 2.)

31_12 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

14   <h:body>
15     <h1>Address Book</h1>
16     <h:form>
17       <p><h:commandButton value="Add Entry" action="addentry"/></p>
18     </h:form>
19     <h:dataTable value="#{addressBean.addresses}" var="address"
20       rowClasses="oddRows,evenRows" headerClass="header"
21       styleClass="table" cellpadding="5" cellspacing="0">
22       <h:column>
23         <f:facet name="header">First Name</f:facet>
24         #{address.FIRSTNAME}
25       </h:column>
26       <h:column>
27         <f:facet name="header">Last Name</f:facet>
28         #{address.LASTNAME}
29       </h:column>
30       <h:column>
31         <f:facet name="header">Street</f:facet>
32         #{address.STREET}
33       </h:column>
34       <h:column>
35         <f:facet name="header">City</f:facet>
36         #{address.CITY}
37       </h:column>
38       <h:column>
39         <f:facet name="header">State</f:facet>
40         #{address.STATE}
41       </h:column>
42       <h:column>
43         <f:facet name="header">Zip code</f:facet>
44         #{address.ZIP}
45       </h:column>
46     </h:dataTable>
47   </h:body>
48 </html>

```

Fig. 31.3 | Displays an `h:dataTable` of the addresses in the address book. (Part 2 of 2.)

The `h:dataTable` element's **value** attribute (line 19) specifies the collection of data you wish to display. In this case, we use `AddressBean`'s `addresses` property, which calls the `getAddresses` method (Fig. 31.2). The collection returned by this method is a `CachedRowSet`, which is a type of `ResultSet`.

The `h:dataTable` iterates over its `value` collection and, one at a time, assigns each element to the variable specified by the **var** attribute. This variable is used in the `h:dataTable`'s nested elements to access each element of the collection—each element in this case represents one row (i.e., `address`) in the `CachedRowSet`.

The **rowClasses** attribute (line 20) is a space-separated list of CSS style class names that are used to style the rows in the tabular output. These style classes are defined in the app's `styles.css` file in the `css` library (which is inserted into the document at line 12). You can open this file to view the various style class definitions. We specified two style classes—all the odd numbered rows will have the first style (`oddRows`) and all the even numbered rows the second style (`evenRows`). You can specify as many styles as you like—

31.2 Accessing Databases in Web Apps 31_13

they'll be applied in the order you list them one row at a time until all the styles have been applied, then the **h:DataTable** will automatically cycle through the styles again for the next set of rows. The **columnClasses** attribute works similarly for columns in the table.

The **headerClass** attribute (line 20) specifies the column header CSS style. Headers are defined with **f:facet** elements nested in **h:column** elements (discussed momentarily). The **footerClass** attribute works similarly for column footers in the table.

The **styleClass** attribute (line 21) specifies the CSS styles for the entire table. The **cellpadding** and **cellspacing** attributes (line 21) specify the number of pixels around each table cell's contents and the number of pixels between table cells, respectively.

The h:column Elements

Lines 22–45 define the table's columns with six nested **h:column** elements. We focus here on the one at lines 22–25. When the **CachedRowSet** is populated in the **AddressBean** class, it automatically uses the database's column names as property names for each row object in the **CachedRowSet**. Line 28 inserts into the column the **FIRSTNAME** property of the **CachedRowSet**'s current row. To display a column header above the column, you define an **f:facet** element (line 23) and set its **name** attribute to "header". Similarly, to display a column footer, use an **f:facet** with its **name** attribute set to "footer". The header is formatted with the CSS style specified in the **h: dataTable**'s **headerClass** attribute (line 20). The remaining **h:column** elements perform similar tasks for the current row's **LASTNAME**, **STREET**, **CITY**, **STATE** and **ZIP** properties.

31.2.4 addentry.xhtml Facelets Page

When the user clicks **Add Entry** in the **index.xhtml** page, **addentry.xhtml** (Fig. 31.4) is displayed. Each **h:inputText** in this page has its **required** attribute set to "true" and includes a **maxlength** attribute that restricts the user's input to the maximum length of the corresponding database field. When the user clicks **Save** (lines 48–49), the input element's values are validated and (if successful) assigned to the properties of the **addressBean** managed object. In addition, the button specifies as its **action** the EL expression

```
#{addressBean.save}
```

which invokes the **addressBean** object's **save** method to store the new address in the database. When you call a method with the **action** attribute, if the method returns a value (in this case, it returns the string "index"), that value is used to request the corresponding page from the app. If the method does not return a value, the current page is re-requested.

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- addentry.html -->
4  <!-- Form for adding an entry to an address book -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8      xmlns:h="http://java.sun.com/jsf/html">
```

Fig. 31.4 | Form for adding an entry to an address book. (Part 1 of 2.)

31_14 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

9   <h:head>
10  <title>Address Book: Add Entry</title>
11  <h:outputStylesheet name="style.css" library="css"/>
12  </h:head>
13  <h:body>
14    <h1>Address Book: Add Entry</h1>
15    <h:form>
16      <h:panelGrid columns="3">
17        <h:outputText value="First name:"/>
18        <h:inputText id="firstNameInputText" required="true"
19          requiredMessage="Please enter first name"
20          value="#{addressBean.firstName}" maxLength="30"/>
21        <h:message for="firstNameInputText" styleClass="error"/>
22        <h:outputText value="Last name:"/>
23        <h:inputText id="lastNameInputText" required="true"
24          requiredMessage="Please enter last name"
25          value="#{addressBean.lastName}" maxLength="30"/>
26        <h:message for="lastNameInputText" styleClass="error"/>
27        <h:outputText value="Street:"/>
28        <h:inputText id="streetInputText" required="true"
29          requiredMessage="Please enter the street address"
30          value="#{addressBean.street}" maxLength="150"/>
31        <h:message for="streetInputText" styleClass="error"/>
32        <h:outputText value="City:"/>
33        <h:inputText id="cityInputText" required="true"
34          requiredMessage="Please enter the city"
35          value="#{addressBean.city}" maxLength="30"/>
36        <h:message for="cityInputText" styleClass="error"/>
37        <h:outputText value="State:"/>
38        <h:inputText id="stateInputText" required="true"
39          requiredMessage="Please enter state"
40          value="#{addressBean.state}" maxLength="2"/>
41        <h:message for="stateInputText" styleClass="error"/>
42        <h:outputText value="Zipcode:"/>
43        <h:inputText id="zipcodeInputText" required="true"
44          requiredMessage="Please enter zipcode"
45          value="#{addressBean.zipcode}" maxLength="5"/>
46        <h:message for="zipcodeInputText" styleClass="error"/>
47      </h:panelGrid>
48      <h:commandButton value="Save Address"
49        action="#{addressBean.save}"/>
50    </h:form>
51    <h:outputLink value="index.xhtml">Return to Addresses</h:outputLink>
52  </h:body>
53 </html>

```

Fig. 31.4 | Form for adding an entry to an address book. (Part 2 of 2.)

31.3 Ajax

The term **Ajax**—short for **Asynchronous JavaScript and XML**—was coined by Jesse James Garrett of Adaptive Path, Inc., in 2005 to describe a range of technologies for developing highly responsive, dynamic web applications. Ajax applications include Google Maps, Yahoo's Flickr and many more. Ajax separates the *user interaction* portion of an ap-

plication from its *server interaction*, enabling both to proceed *in parallel*. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications, reducing or even eliminating the performance advantage that desktop applications have traditionally had over web-based applications. This has huge ramifications for the desktop applications industry—the applications platform of choice is shifting from the desktop to the web. Many people believe that the web—especially in the context of abundant open-source software, inexpensive computers and exploding Internet bandwidth—will create the next major growth phase for Internet companies.

Ajax makes **asynchronous** calls to the server to exchange small amounts of data with each call. *Where normally the entire page would be submitted and reloaded with every user interaction on a web page, Ajax allows only the necessary portions of the page to reload, saving time and resources.*

Ajax applications typically make use of client-side scripting technologies such as JavaScript to interact with page elements. They use the browser's **XMLHttpRequest** object to perform the asynchronous exchanges with the web server that make Ajax applications so responsive. This object can be used by most scripting languages to pass XML data from the client to the server and to process XML data sent from the server back to the client.

Using Ajax technologies in web applications can dramatically improve performance, but programming Ajax directly is complex and error prone. It requires page designers to know both scripting and markup languages. As you'll soon see, JSF makes adding Ajax capabilities to your web apps fairly simple.

Traditional Web Applications

Figure 31.5 presents the typical interactions between the client and the server in a traditional web application, such as one that uses a user registration form.

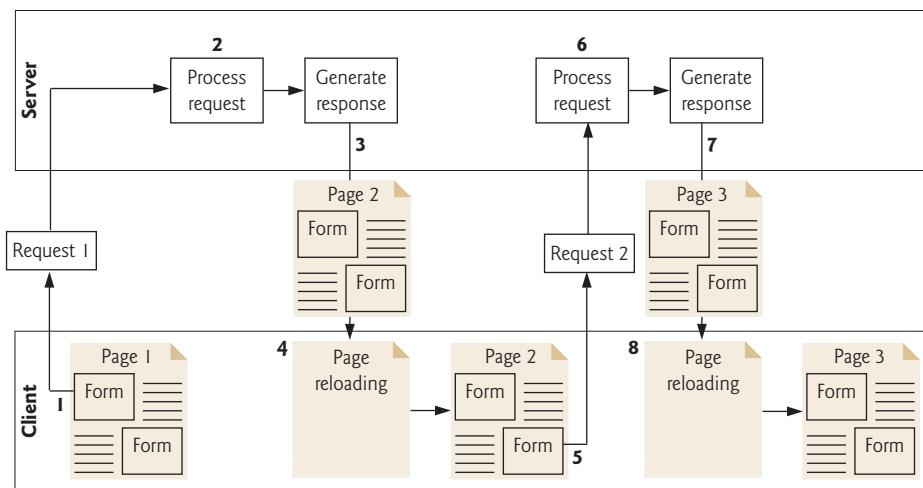


Fig. 31.5 | Classic web application reloading the page for every user interaction.

31_16 Chapter 31 JavaServer™ Faces Web Apps: Part 2

The user first fills in the form's fields, then *submits* the form (Fig. 31.5, *Step 1*). The browser generates a request to the server, which receives the request and processes it (*Step 2*). The server generates and sends a response containing the exact page that the browser will render (*Step 3*), which causes the browser to load the new page (*Step 4*) and temporarily makes the browser window blank. The client *waits* for the server to respond and *reloads the entire page* with the data from the response (*Step 4*). While such a **synchronous request** is being processed on the server, *the user cannot interact with the client web page*. If the user interacts with and submits another form, the process begins again (*Steps 5–8*).

This model was originally designed for a web of *hypertext documents*—what some people call the “brochure web.” As the web evolved into a full-scale applications platform, the model shown in Fig. 31.5 yielded “choppy” application performance. Every full-page refresh required users to reestablish their understanding of the full-page contents. Users began to demand a model that would yield the responsiveness of desktop applications.

Ajax Web Applications

Ajax applications add a layer between the client and the server to manage communication between the two (Fig. 31.6).

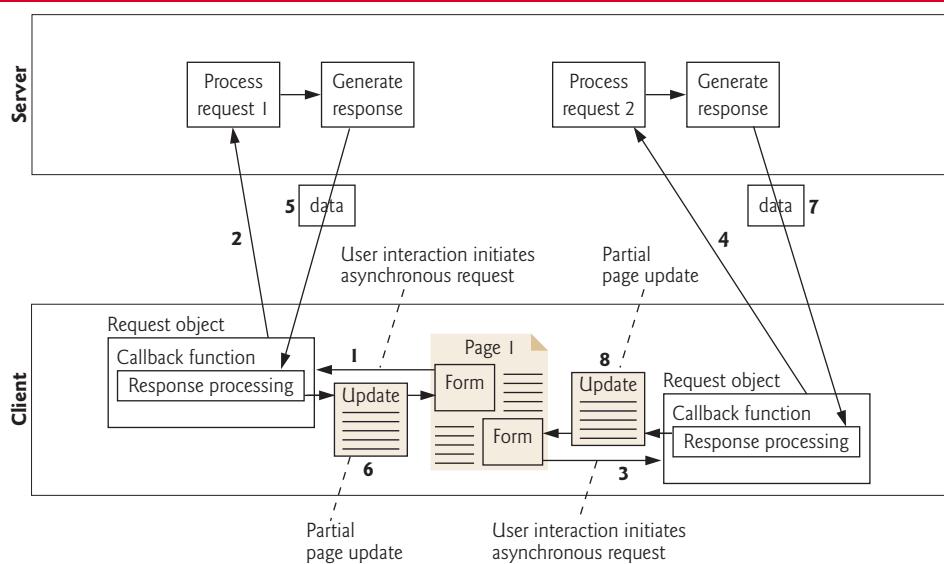


Fig. 31.6 | Ajax-enabled web application interacting with the server asynchronously.

When the user interacts with the page, the client creates an XMLHttpRequest object to manage a request (*Step 1*). This object sends the request to the server (*Step 2*) and awaits the response. The requests are asynchronous, so the user can continue interacting with the application on the client side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server (*Steps 3 and 4*). Once the server responds to the original request (*Step 5*), the XMLHttpRequest object that issued the request calls a client-side function to process the data returned by the server. This function—known as a **callback function**—uses **partial page updates** (*Step 6*) to dis-

31.4 Adding Ajax Functionality to the Validation App 31_17

play the data in the existing web page *without reloading the entire page*. At the same time, the server may be responding to the second request (*Step 7*) and the client side may be starting to do another partial page update (*Step 8*). The callback function updates only a designated part of the page. Such partial page updates help make web applications more responsive, making them feel more like desktop applications. The web application does not load a new page while the user interacts with it.

31.4 Adding Ajax Functionality to the Validation App

The example in this section adds Ajax capabilities to the Validation app that we presented in Section 30.7. Figure 31.7 shows the sample outputs from the ValidationAjax version of the app that we'll build momentarily. Part (a) shows the initial form that's displayed when this app first executes. Parts (b) and (c) show validation errors that are displayed when the user submits an empty form and invalid data, respectively. Part (d) shows the page after the form is submitted successfully.

a) Submitting the form before entering any information

The screenshot shows a web browser window with a title bar "Validating Form Data". The address bar shows "localhost:8080/ValidationAjax/". The main content area has a heading "Please fill out the following form:". Below it is a message "All fields are required and must contain valid information". There are three input fields: "Name:", "E-mail:", and "Phone:". Below the fields is a "Submit" button.

b) Error messages displayed after submitting the empty form

The screenshot shows the same web browser window as part (a). After submission, red error messages appear next to each input field: "Name:" with "Please enter your name", "E-mail:" with "Please enter a valid e-mail address", and "Phone:" with "Please enter a valid phone number". The "Submit" button is highlighted with a cursor.

Fig. 31.7 | JSP that demonstrates validation of user input. (Part 1 of 2.)

31_18 Chapter 31 JavaServer™ Faces Web Apps: Part 2

c) Error messages displayed after submitting invalid information

The screenshot shows a browser window titled "Validating Form Data" with the URL "localhost:8080/ValidationAjax/". The page contains the heading "Please fill out the following form:" and a note "All fields are required and must contain valid information". Below this, there are three input fields with error messages: "Name: Paul plus a bunch of other" with "Name must be fewer than 30 characters" in red; "E-mail: not a valid email" with "Invalid e-mail address format" in red; and "Phone: 55-1234" with "Invalid phone number format" in red. A blue "Submit" button is at the bottom.

d) Successfully submitted form

The screenshot shows a browser window titled "Validating Form Data" with the URL "localhost:8080/ValidationAjax/". The page contains the heading "Please fill out the following form:" and a note "All fields are required and must contain valid information". Below this, there are three input fields: "Name: Paul", "E-mail: paul@somedomain.com", and "Phone: (555) 555-5555". A blue "Submit" button is at the bottom. Below the form, a yellow box displays the submitted data: "Name: Paul", "E-Mail: paul@somedomain.com", and "Phone: (555) 555-5555".

Fig. 31.7 | JSP that demonstrates validation of user input. (Part 2 of 2.)

As you can see, the app has the same functionality as the version in Section 30.7; however, you'll notice a couple of changes in how the app works. First, the URL displayed in the web browser always reads `localhost:8080/ValidationAjax/`, whereas the URL in the Section 30.7 changes after the form is submitted the first time. Also, in the non-Ajax version of the app, the page refreshes each time you press the **Submit** button. In the Ajax version, only the parts of the page that need updating actually change.

index.xhtml

The changes required to add Ajax functionality to this app are minimal. All of the changes are in the `index.xhtml` file (Fig. 31.8) and are highlighted. The `ValidationBean` class is identical to the version in Section 30.7, so we don't show it here.

31.4 Adding Ajax Functionality to the Validation App

31_19

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- index.xhtml -->
4  <!-- Validating user input -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8   xmlns:h="http://java.sun.com/jsf/html"
9   xmlns:f="http://java.sun.com/jsf/core">
10 <h:head>
11   <title>Validating Form Data</title>
12   <h:outputStylesheet name="style.css" library="css"/>
13 </h:head>
14 <h:body>
15   <h:form>
16     <h1>Please fill out the following form:</h1>
17     <p>All fields are required and must contain valid information</p>
18     <h:panelGrid columns="3">
19       <h:outputText value="Name:"/>
20       <h:inputText id="nameInputText" required="true"
21         requiredMessage="Please enter your name"
22         value="#{validationBean.name}"
23         validatorMessage="Name must be fewer than 30 characters">
24           <f:validateLength maximum="30" />
25         </h:inputText>
26         <h:message id="nameMessage" for="nameInputText"
27           styleClass="error"/>
28       <h:outputText value="E-mail:"/>
29       <h:inputText id="emailInputText" required="true"
30         requiredMessage="Please enter a valid e-mail address"
31         value="#{validationBean.email}"
32         validatorMessage="Invalid e-mail address format">
33           <f:validateRegex pattern=
34             "\w+([-.\']\w+)*@\w+([.-]\w+)*\.\w+([.-]\w+)*" />
35         </h:inputText>
36         <h:message id="emailMessage" for="emailInputText"
37           styleClass="error"/>
38       <h:outputText value="Phone:"/>
39       <h:inputText id="phoneInputText" required="true"
40         requiredMessage="Please enter a valid phone number"
41         value="#{validationBean.phone}"
42         validatorMessage="Invalid phone number format">
43           <f:validateRegex pattern=
44             "((\(\d{3}\)\s)?|\s(\d{3}-))?\d{3}-\d{4}" />
45         </h:inputText>
46         <h:message id="phoneMessage" for="phoneInputText"
47           styleClass="error"/>
48     </h:panelGrid>
49     <h:commandButton value="Submit">
50       <f:ajax execute="nameInputText emailInputText phoneInputText"
51         render=
52           "nameMessage emailMessage phoneMessage resultOutputText"/>
53     </h:commandButton>

```

Fig. 31.8 | Ajax enabling the Validation app. (Part 1 of 2.)

31.20 Chapter 31 JavaServer™ Faces Web Apps: Part 2

```

54      <h:outputText id="resultOutputText" escape="false"
55          value="#{validationBean.response}"/>
56      </h:form>
57  </h:body>
58 </html>
```

Fig. 31.8 | Ajax enabling the Validation app. (Part 2 of 2.)

Adding *id* Attributes to Elements

The Facelets elements that will be submitted as part of an Ajax request and the Facelets elements that will participate in the partial page updates must have *id* attributes. The *h:inputText* elements in the original Validation example already had *id* attributes. These elements will be submitted to the server as part of an Ajax request. We'd like the *h:Message* elements that show validation errors and the *h:outputText* element that displays the result to be updated with partial page updates. For this reason, we've added *id* attributes to these elements.

f:ajax Element

The other key change to this page is at lines 49–53 where the *h:commandButton* now contains an *f:ajax* element, which intercepts the form submission when the user clicks the button and makes an Ajax request instead. The *f:ajax* element's **execute** attribute specifies a space-separated list of element *ids*—the values of these elements are submitted as part of the Ajax request. The *f:ajax* element's **render** attribute specifies a space-separated list of element *ids* for the elements that should be updated via partial page updates.

31.5 Wrap-Up

In this chapter, we built an AddressBook application that allowed a user to add and view contacts. You learned how to insert user input into a Java DB database and how to display the contents of a database on a web page using an *h:dataTable* JSF element. We also demonstrated how to add Ajax capabilities to JSF web apps by enhancing the Validation app from Section 30.7. In Chapter 32, you'll use NetBeans to create web services and consume them from desktop and web applications.

Summary

Section 30.2.2 Class *AddressBean*

- To connect to the addressbook database from a web app, you must configure a data source name that will be used to locate the database.
- Java EE 7's *@DataSourceDefinition* annotation creates a data source name, specifying its JNDI (Java Naming and Directory Interface) name that is used to look up the data source.
- JNDI is a technology for locating application components (such as databases) in a distributed application (such as a mult-tier web application).
- A *DataSource* (package *javax.sql*) enables a web application to obtain a *Connection* to a database.
- *ClientDataSource* is one of several *DataSource* subclasses provided by Java DB. Apps that are expected to manage many connections at once would typically use *ClientConnectionPoolDataSource* or *ClientXADataSource*.

- `@ManagedBean` is deprecated in Java EE 7 and Contexts and Dependency Injection (CDI) should be used instead. Switching to CDI simply requires changing from JSF's `@ManagedBean` annotation to CDI's `@Named` annotation (line 26):
- As with `@ManagedBean`, if you do not specify a name in parentheses, the JavaBean object's variable name will be the JavaBean class's name with a lowercase first letter.
- The annotation `@javax.faces.view.ViewScoped` indicates that CDI should manage a JavaBean's lifetime, based on the JSF view that first referenced the JavaBean. A `ViewScoped` JavaBean's class must be `Serializable`.
- The annotation `@Resource` (p. 10) can be used to inject a `DataSource` object into a managed bean. The annotation's `lookup` attribute specifies the JNDI name of a data source.
- The `@Resource` annotation enables the server to hide all the complex details of setting up a `DataSource` object that can interact with a database. The server creates a `DataSource` for you and assigns the `DataSource` object to the annotated variable. You can then trivially obtain a `Connection` for interacting with the database.
- Database connections are limited resources, so you should use and close them quickly in your web apps. You can use a `CachedRowSet` to store the results of a query for use later.

Section 30.2.3 index.xhtml Facelets Page

- You can use an `h:dataTable` element (p. 11) to display a collection of objects, such as the rows in a `CachedRowSet`, in tabular format.
- If you specify an `h:commandButton`'s `action` attribute (p. 11) with a value that is the name of a web page (without the filename extension), the JSF framework assumes this is a page in the app, appends `.xhtml` extension to the `action` attribute's value and returns the page to the client browser.
- The `h:dataTable` element's `value` attribute (p. 12) specifies the collection of data you wish to display. The `h:dataTable` iterates over its `value` collection and, one at a time, assigns each element to the variable specified by the `var` attribute (p. 12). This variable is used in the `h:dataTable`'s nested elements to access each element of the collection.
- The `h:dataTable` `rowClasses` attribute (p. 12) is a space-separated list of CSS style class names that are used to style the rows in the tabular output. You can specify as many styles as you like—they'll be applied in the order you list them one row at a time until all the styles have been applied, then the `h:DataTable` will automatically cycle through the styles again for the next set of rows. The `columnClasses` attribute works similarly for columns in the table.
- The `headerClass` attribute (p. 13) specifies the column header CSS style. The `footerClass` attribute (p. 13) works similarly for column footers in the table.
- The `styleClass` attribute (p. 13) specifies the CSS styles for the entire table. The `cellpadding` and `cellspacing` attributes (p. 13) specify the number of pixels around each table cell's contents and the number of pixels between table cells, respectively.
- An `h:column` element (p. 13) defines a column in an `h:dataTable`.
- To display a column header above a column, define an `f:facet` element (p. 13) and set its `name` attribute to "header". Similarly, to display a column footer, use an `f:facet` with its `name` attribute set to "footer".

Section 30.2.4 addentry.xhtml Facelets Page

- You can call a managed bean's methods in EL expressions.



31_22 Chapter 31 JavaServer™ Faces Web Apps: Part 2

- When you call a managed bean method with the `action` attribute, if the method returns a value, that value is used to request the corresponding page from the app. If the method does not return a value, the current page is re-requested.

Section 31.3 Ajax

- The term Ajax—short for Asynchronous JavaScript and XML—was coined by Jesse James Garrett of Adaptive Path, Inc., in February 2005 to describe a range of technologies for developing highly responsive, dynamic web applications.
- Ajax separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel. This enables Ajax web-based applications to perform at speeds approaching those of desktop applications.
- Ajax makes asynchronous calls to the server to exchange small amounts of data with each call. Where normally the entire page would be submitted and reloaded with every user interaction on a web page, Ajax reloads only the necessary portions of the page, saving time and resources.
- Ajax applications typically make use of client-side scripting technologies such as JavaScript to interact with page elements. They use the browser's `XMLHttpRequest` object to perform the asynchronous exchanges with the web server that make Ajax applications so responsive.
- In a traditional web application, the user fills in a form's fields, then submits the form. The browser generates a request to the server, which receives the request and processes it. The server generates and sends a response containing the exact page that the browser will render. The browser loads the new page, temporarily making the browser window blank. The client waits for the server to respond and reloads the entire page with the data from the response. While such a synchronous request is being processed on the server, the user cannot interact with the web page. This model yields “choppy” application performance.
- In an Ajax application, when the user interacts with the page, the client creates an `XMLHttpRequest` object to manage a request. This object sends the request to the server and awaits the response. The requests are asynchronous, so the user can interact with the application on the client side while the server processes the earlier request concurrently. Other user interactions could result in additional requests to the server. Once the server responds to the original request, the `XMLHttpRequest` object that issued the request calls a client-side function to process the data returned by the server. This callback function uses partial page updates to display the data in the existing web page without reloading the entire page. At the same time, the server may be responding to the second request and the client side may be starting to do another partial page update.
- Partial page updates help make web applications more responsive, making them feel more like desktop applications.

Section 31.4 Adding Ajax Functionality to the Validation App

- The Facelets elements that will be submitted as part of an Ajax request and the Facelets elements that will participate in the partial page updates must have `id` attributes.
- When you nest an `f:ajax` element (p. 20) in an `h:commandButton` element, the `f:ajax` element intercepts the form submission and makes an Ajax request instead.
- The `f:ajax` element's `execute` attribute (p. 20) specifies a space-separated list of element `ids`—the values of these elements are submitted as part of the Ajax request.
- The `f:ajax` element's `render` attribute (p. 20) specifies a space-separated list of element `ids` for the elements that should be updated via partial page updates.

Self-Review Exercise

31_23

Self-Review Exercise

31.1 Fill in the blanks in each of the following statements.

- a) Ajax is an acronym for _____.
- b) A(n) _____ allows the server to manage a limited number of database connections and share them among requests.
- c) is a technology for locating application components (such as databases) in a distributed application.
- d) A(n) _____ enables a web application to obtain a Connection to a database.
- e) The annotation _____ can be used to inject a DataSource object into a managed bean.
- f) A(n) _____ element displays a collection of objects in tabular format.
- g) An `h:commandButton`'s _____ attribute can specify the name of another page in the web app that should be returned to the client.
- h) To specify headers or footers for the columns in `h:dataTables`, use _____ elements nested with their `name` attributes set to _____ and _____, respectively.
- i) _____ separates the user interaction portion of an application from its server interaction, enabling both to proceed asynchronously in parallel.
- j) _____ help make web applications more responsive, making them feel more like desktop applications.
- k) The `f:ajax` element's _____ attribute specifies a space-separated list of element `ids`—the values of these elements are submitted as part of the Ajax request.
- l) The `f:ajax` element's _____ attribute specifies a space-separated list of element `ids` for the elements that should be updated via partial page updates.

Answers to Self-Review Exercise

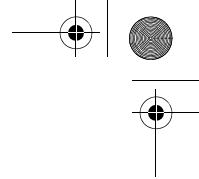
31.1 a) Asynchronous JavaScript and XML. b) connection pool. c) JNDI (Java Naming and Directory Interface). d) DataSource. e) @Resource. f) `h:dataTable`. g) `action`. h) `f:facet`, "header", "footer". i) Ajax. j) partial page updates. k) `execute`. l) `render`.

Exercises

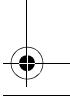
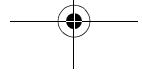
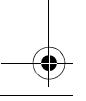
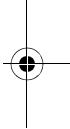
31.2 (*Guestbook Application*) Create a JSF web app that allows users to sign and view a guestbook. Use the Guestbook database to store guestbook entries. [Note: A SQL script to create the Guestbook database is provided in the examples directory for this chapter.] The Guestbook database has a single table, `Messages`, which has four columns: `Date`, `Name`, `Email` and `Message`. The database already contains a few sample entries. Using the AddressBook app in Section 31.2 as your guide, create two Facelets pages and a managed bean. The `index.xhtml` page should show the Guestbook entries in tabular format and should provide a button to add an entry to the Guestbook. When the user clicks this button, display an `addentry.xhtml` page. Provide `h:inputText` elements for the user's name and email address, an `h:inputTextarea` for the message and a `Sign Guestbook` button to submit the form. When the form is submitted, you should store in the Guestbook database a new entry containing the user's input and the date of the entry.

31.3 (*AddressBook Application Modification: Ajax*) Combine the two Facelets pages of the AddressBook application (Section 31.2) into a single page. Use Ajax capabilities to submit the new address book entry and to perform a partial page update that rerenders `h:dataTable` with the updated list of addresses.

31.4 (*AddressBook Application Modification*) Modify your solution to Exercise 31.3 to add a search capability that allows the user to search by last name. When the user presses the `Search` button, use Ajax to submit the search key and perform a partial page update that displays only the matching addresses in the `h:dataTable`.



31_24 Chapter 31 JavaServer™ Faces Web Apps: Part 2



REST Web Services

32

Objectives

In this chapter you will learn:

- What a web service is.
- How to publish and consume web services in NetBeans.
- How XML, JSON, XML-Based Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) Architecture enable Java web services.
- How to create client desktop and web applications that consume web services.
- How to use session tracking in web services to maintain client state information.
- How to connect to databases from web services.
- How to pass objects of user-defined types to and return them from a web service.



Outline

32_2 Chapter 32 REST Web Services

- 32.1** Introduction
- 32.2** Web Service Basics
- 32.3** Simple Object Access Protocol (SOAP)
- 32.4** Representational State Transfer (REST)
- 32.5** JavaScript Object Notation (JSON)
- 32.6** Publishing and Consuming SOAP-Based Web Services
 - 32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans
 - 32.6.2 Defining the `WelcomeSOAP` Web Service in NetBeans
 - 32.6.3 Publishing the `WelcomeSOAP` Web Service from NetBeans
 - 32.6.4 Testing the `WelcomeSOAP` Web Service with GlassFish Application Server's `Tester` Web Page
 - 32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)
 - 32.6.6 Creating a Client to Consume the `WelcomeSOAP` Web Service
 - 32.6.7 Consuming the `WelcomeSOAP` Web Service
- 32.7** Publishing and Consuming REST-Based XML Web Services
 - 32.7.1 Creating a REST-Based XML Web Service
 - 32.7.2 Consuming a REST-Based XML Web Service
- 32.8** Publishing and Consuming REST-Based JSON Web Services
 - 32.8.1 Creating a REST-Based JSON Web Service
 - 32.8.2 Consuming a REST-Based JSON Web Service
- 32.9** Session Tracking in a SOAP Web Service
 - 32.9.1 Creating a `Blackjack` Web Service
 - 32.9.2 Consuming the `Blackjack` Web Service
- 32.10** Consuming a Database-Driven SOAP Web Service
 - 32.10.1 Creating the `Reservation` Database
 - 32.10.2 Creating a Web Application to Interact with the `Reservation` Service
- 32.11** Equation Generator: Returning User-Defined Types
 - 32.11.1 Creating the `Equation-GeneratorXML` Web Service
 - 32.11.2 Consuming the `Equation-GeneratorXML` Web Service
 - 32.11.3 Creating the `Equation-GeneratorJSON` Web Service
 - 32.11.4 Consuming the `Equation-GeneratorJSON` Web Service
- 32.12** Wrap-Up

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#) | [Making a Difference](#)

32.1 Introduction

This chapter introduces web services, which promote software portability and reusability in applications that operate over the Internet. A **web service** is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network. Web services communicate using such technologies as XML, JSON and HTTP. In this chapter, we use two Java APIs that facilitate web services. The first, **JAX-WS**, is based on the **Simple Object Access Protocol (SOAP)**—an XML-based protocol that allows web services and clients to communicate, even if the client and the web service are written in different languages. The second, **JAX-RS**, uses **Representational State Transfer (REST)**—a network architecture that uses the web's traditional request/response mechanisms such as GET and POST requests. For more information on SOAP-based and REST-based web services, visit our Web Services Resource Centers:

www.deitel.com/WebServices/
www.deitel.com/RESTWebServices/

These Resource Centers include information about designing and implementing web services in many languages and about web services offered by companies such as Google, Amazon and eBay. You'll also find many additional tools for publishing and consuming web services. For more information about REST-based Java web services, check out the Jersey project:

jersey.java.net/

The XML used in this chapter is created and manipulated for you by the APIs, so you need not know the details of XML to use it here. To learn more about XML, read the following tutorials:

www.deitel.com/articles/xml_tutorials/20060401/XMLBasics/
www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData/

and visit our XML Resource Center:

www.deitel.com/XML/

Business-to-Business Transactions

Rather than relying on proprietary applications, businesses can conduct transactions via standardized, widely available web services. This has important implications for **business-to-business (B2B) transactions**. Web services are platform and language independent, enabling companies to collaborate without worrying about the compatibility of their hardware, software and communications technologies. Companies such as Amazon, Google, eBay, PayPal and many others make their server-side applications available to partners via web services.

By purchasing some web services and using other free ones that are relevant to their businesses, companies can spend less time developing applications and can create new ones that are more innovative. E-businesses for example, can provide their customers with enhanced shopping experiences. Consider an online music store. The store's website links to information about various artists, enabling users to purchase their music, to learn about the artists, to find more titles by those artists, to find other artists' music they may enjoy, and more. The store's website may also link to the site of a company that sells concert tickets and provides a web service that displays upcoming concert dates for various artists, allowing users to buy tickets. By consuming the concert-ticket web service on its site, the online music store can provide an additional service to its customers, increase its site traffic and perhaps earn a commission on concert-ticket sales. The company that sells concert tickets also benefits from the business relationship by selling more tickets and possibly by receiving revenue from the online music store for the use of the web service.

Any Java programmer with a knowledge of web services can write applications that "consume" web services. The resulting applications would invoke web services running on servers that could be thousands of miles away.

NetBeans

NetBeans is one of many tools that enable you to *publish* and/or *consume* web services. We demonstrate how to use NetBeans to implement web services using the JAX-WS and JAX-RS APIs and how to invoke them from client applications. For each example, we provide the web service's code, then present a client application that uses the web service. Our first examples build simple web services and client applications in NetBeans. Then we demonstrate web services that use more sophisticated features, such as manipulating databases

32.4 Chapter 32 REST Web Services

with JDBC and manipulating class objects. For information on downloading and installing the NetBeans and the GlassFish server, see Section 30.1.

32.2 Web Service Basics

The machine on which a web service resides is referred to as a **web service host**. The client application sends a request over a network to the web service host, which processes the request and returns a response over the network to the application. This kind of distributed computing benefits systems in various ways. For example, an application without direct access to data on another system might be able to retrieve the data via a web service. Similarly, an application lacking the processing power to perform specific computations could use a web service to take advantage of another system's superior resources.

In Java, a web service is implemented as a class that resides on a server—it's not part of the client application. Making a web service available to receive client requests is known as **publishing a web service**; using a web service from a client application is known as **consuming a web service**.

32.3 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) is a platform-independent protocol that uses XML to interact with web services, typically over HTTP. You can view the SOAP specification at www.w3.org/TR/soap/. Each request and response is packaged in a **SOAP message**—XML markup containing the information that a web service requires to process the message. SOAP messages are written in XML so that they're computer readable, human readable and platform independent. Most **firewalls**—security barriers that restrict communication among networks—allow HTTP traffic to pass through, so that clients can browse the web by sending requests to and receiving responses from web servers. Thus, SOAP-based services can send and receive SOAP messages over HTTP connections with few limitations.

SOAP supports an extensive set of types, including the primitive types (e.g., `int`), as well as `DateTime`, `XmlNode` and others. SOAP can also transmit arrays of these types. When a program invokes a method of a SOAP web service, the request and all relevant information are packaged in a SOAP message enclosed in a **SOAP envelope** and sent to the server on which the web service resides. When the web service receives this SOAP message, it parses the XML representing the message, then processes the message's contents. The message specifies the *method* that the client wishes to execute and the *arguments* the client passed to that method. Next, the web service calls the method with the specified arguments (if any) and sends the response back to the client in another SOAP message. The client parses the response to retrieve the method's result. In Section 32.6, you'll build and consume a basic SOAP web service.

32.4 Representational State Transfer (REST)

Representational State Transfer (REST) refers to an architectural style for implementing web services. Such web services are often called **RESTful web services**. Though REST itself is not a standard, RESTful web services are implemented using web standards. Each method in a RESTful web service is identified by a unique URL. Thus, when the server receives a request,

32.5 JavaScript Object Notation (JSON)

32_5

it immediately knows what operation to perform. Such web services can be used in a program or directly from a web browser. The results of a particular operation may be cached locally by the browser when the service is invoked with a GET request. This can make subsequent requests for the same operation faster by loading the result directly from the browser's cache. Amazon's web services (aws.amazon.com) are RESTful, as are many others.

RESTful web services are alternatives to those implemented with SOAP. Unlike SOAP-based web services, the request and response of REST services are not wrapped in envelopes. REST is also not limited to returning data in XML format. It can use a variety of formats, such as XML, JSON, HTML, plain text and media files. In Sections 32.7–32.8, you'll build and consume basic RESTful web services.

32.5 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is an alternative to XML for representing data. JSON is a text-based data-interchange format used to represent objects in JavaScript as collections of name/value pairs represented as **Strings**. It's commonly used in Ajax applications. JSON is a simple format that makes objects easy to read, create and parse and, because it's much less verbose than XML, allows programs to transmit data efficiently across the Internet. Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

```
{ propertyName1 : value1, propertyName2 : value2 }
```

Arrays are represented in JSON with square brackets in the following format:

```
[value1, value2, value3]
```

Each value in an array can be a string, a number, a JSON object, **true**, **false** or **null**. To appreciate the simplicity of JSON data, examine this representation of an array of address-book entries:

```
[{first: 'Cheryl', last: 'Black'},
 {first: 'James', last: 'Blue'},
 {first: 'Mike', last: 'Brown'},
 {first: 'Meg', last: 'Gold'}]
```

Many programming languages now support the JSON data format. An extensive list of JSON libraries sorted by language can be found at www.json.org.

32.6 Publishing and Consuming SOAP-Based Web Services

This section presents our first example of publishing (enabling for client access) and consuming (using) a web service. We begin with a SOAP-based web service.

32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans

When you create a web service in NetBeans, you focus on its logic and let the IDE and server handle its infrastructure. First you create a **Web Application project**. NetBeans uses this project type for web services that are invoked by other applications.

32_6 Chapter 32 REST Web Services

Creating a Web Application Project in NetBeans

To create a web application, perform the following steps:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Select **Java Web** from the dialog's **Categories** list, then select **Web Application** from the **Projects** list. Click **Next >**.
3. Specify **WelcomeSOAP** in the **Project Name** field and specify where you'd like to store the project in the **Project Location** field. You can click the **Browse** button to select the location. Click **Next >**.
4. Select **GlassFish Server 4.1** from the **Server** drop-down list and **Java EE 7 Web** from the **Java EE Version** drop-down list.
5. Click **Finish** to create the project.

This creates a web application that will run in a web browser, similar to the projects used in Chapters 30 and 31.

Adding a Web Service Class to a Web Application Project

Perform the following steps to add a web service class to the project:

1. In the **Projects** tab in NetBeans, right click the **WelcomeSOAP** project's node and select **New > Web Service...** to open the **New Web Service** dialog.
2. Specify **WelcomeSOAP** in the **Web Service Name** field.
3. Specify **com.deitel.welcomesoap** in the **Package** field.
4. Click **Finish** to create the web service class.

The IDE generates a sample web service class with the name from *Step 2* in the package from *Step 3*. You can find this class in your project's **Web Services** node. In this class, you'll define the methods that your web service makes available to client applications. When you eventually build your application, the IDE will generate other supporting files for your web service.

32.6.2 Defining the WelcomeSOAP Web Service in NetBeans

Figure 32.1 contains the completed **WelcomeSOAPService** code (reformatted to match the coding conventions we use in this book). First we discuss this code, then show how to use the NetBeans web service design view to add the **welcome** method to the class.

```

1 // Fig. 32.1: WelcomeSOAP.java
2 // Web service that returns a welcome message via SOAP.
3 package com.deitel.welcomesoap;
4
5 import javax.jws.WebService; // program uses the annotation @WebService
6 import javax.jws.WebMethod; // program uses the annotation @WebMethod
7 import javax.jws.WebParam; // program uses the annotation @WebParam
8

```

Fig. 32.1 | Web service that returns a welcome message via SOAP. (Part 1 of 2.)

32.6 Publishing and Consuming SOAP-Based Web Services

32-7

```

9  @WebService() // annotates the class as a web service
10 public class WelcomeSOAP
11 {
12     // WebMethod that returns welcome message
13     @WebMethod(operationName = "welcome")
14     public String welcome(@WebParam(name = "name") String name)
15     {
16         return "Welcome to JAX-WS web services with SOAP, " + name + "!";
17     }
18 }
```

Fig. 32.1 | Web service that returns a welcome message via SOAP. (Part 2 of 2.)

Annotation import Declarations

Lines 5–7 import the annotations used in this example. By default, each new web service class created with the JAX-WS APIs is a POJO (plain old Java object), so you do *not* need to extend a class or implement an interface to create a web service.

@WebService Annotation

Line 9 contains a **@WebService annotation** (imported at line 5) which indicates that class `WelcomeSOAP` implements a web service. The annotation is followed by parentheses that may contain optional annotation attributes. The optional **name attribute** specifies the name of the service endpoint interface class that will be generated for the client. A **service endpoint interface (SEI)** class (sometimes called a **proxy class**) is used to interact with the web service—a client application consumes the web service by invoking methods on the service endpoint interface object. The optional **serviceName attribute** specifies the service name, which is also the name of the class that the client uses to obtain a service endpoint interface object. If the **serviceName** attribute is not specified, the web service's name is assumed to be the Java class name followed by the word **Service**. NetBeans places the **@WebService** annotation at the beginning of each new web service class you create. You can then add the **name** and **serviceName** properties in the parentheses following the annotation.

When you deploy a web application containing a class that uses the **@WebService** annotation, the server (GlassFish in our case) recognizes that the class implements a web service and creates all the **server-side artifacts** that support the web service—that is, the framework that allows the web service to wait for client requests and respond to those requests once it's deployed on an application server. Some popular open-source application servers that support Java web services include GlassFish (glassfish.dev.java.net), Apache Tomcat (tomcat.apache.org) and JBoss Application Server (www.jboss.com/products/platforms/application).

WelcomeSOAP Service's welcome Method

The `WelcomeSOAP` service has only one method, `welcome` (lines 13–17), which takes the user's name as a `String` and returns a `String` containing a welcome message. This method is tagged with the **@WebMethod annotation** to indicate that it can be called remotely. Any methods that are not tagged with **@WebMethod** are *not* accessible to clients that consume the web service. Such methods are typically utility methods within the web service class. The **@WebMethod** annotation uses the **operationName** attribute to specify the method name

32_8 Chapter 32 REST Web Services

that is exposed to the web service's client. If the `operationName` is not specified, it's set to the actual Java method's name.



Common Programming Error 32.1

Failing to expose a method as a web method by declaring it with the `@WebMethod` annotation prevents clients of the web service from accessing the method. There's one exception—if none of the class's methods are declared with the `@WebMethod` annotation, then all the public methods of the class will be exposed as web methods.



Common Programming Error 32.2

Methods with the `@WebMethod` annotation cannot be static. An object of the web service class must exist for a client to access the service's web methods.

The `name` parameter to `welcome` is annotated with the `@WebParam` annotation (line 14). The optional `@WebParam` attribute `name` indicates the parameter name that is exposed to the web service's clients. If you don't specify the name, the actual parameter name is used.

Completing the Web Service's Code

[Note: If you enter the code in Fig. 32.1 manually, then you can skip the following steps.] NetBeans provides a web service design view in which you can define the method(s) and parameter(s) for your web services. To define the `WelcomeSOAP` class's `welcome` method, perform the following steps:

- With `WelcomeSOAP.java` open in the editor, click the **Design** button at the top of the editor to show the design view (Fig. 32.2).

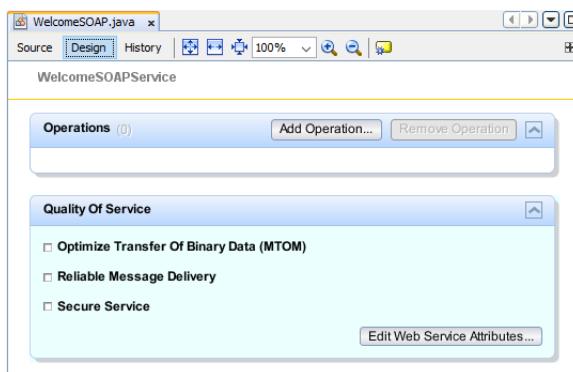


Fig. 32.2 | Web service design view.

- Click the **Add Operation...** button to display the **Add Operation...** dialog (Fig. 32.3).
- Specify the method name `welcome` in the **Name** field. The default **Return Type** (`String`) is correct for this example.

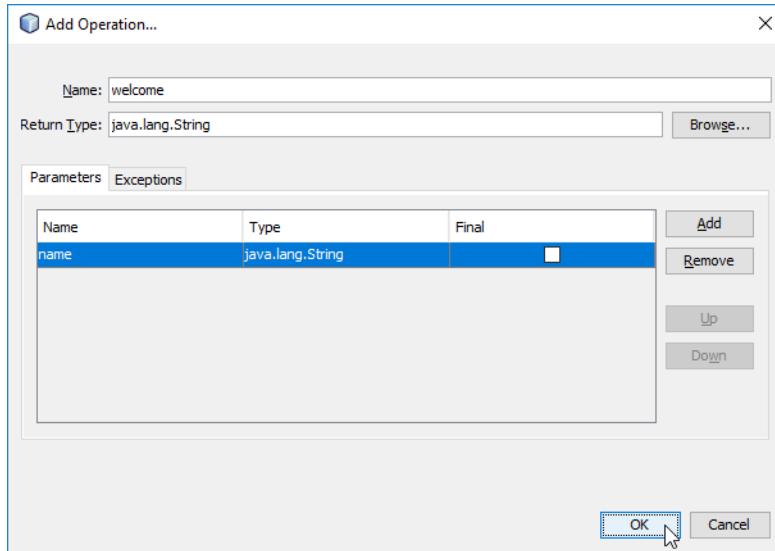


Fig. 32.3 | Adding an operation to a web service.

4. Add the method's name parameter by clicking the Add button to the right of the Parameters tab then entering name in the Name field. The parameter's default Type (String) is correct for this example.
5. Click OK to create the welcome method. The design view should now appear as shown in Fig. 32.3.

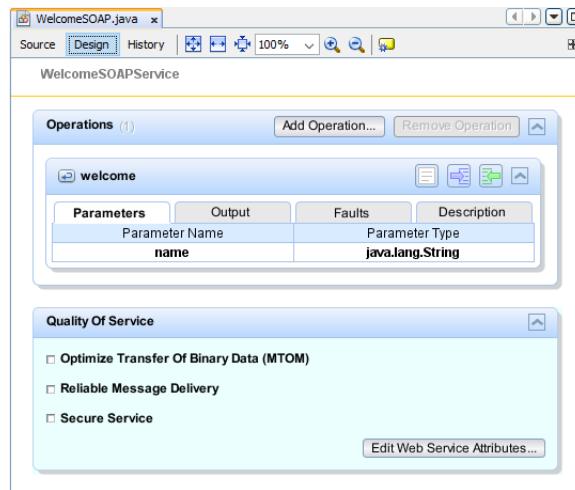


Fig. 32.4 | Web service design view after new operation is added.

32_10 Chapter 32 REST Web Services

- At the top of the design view, click the **Source** button to display the class's source code and add the code line 18 of Fig. 32.1 to the body of method `welcome`.

32.6.3 Publishing the WelcomeSOAP Web Service from NetBeans

Now that you've created the `WelcomeSOAP` web service class, you'll use NetBeans to build and *publish* (that is, deploy) the web service so that clients can consume its services. NetBeans handles all the details of building and deploying a web service for you. This includes creating the framework required to support the web service. Right click the project name `WelcomeSOAP` in the **Projects** tab and select **Deploy** to build and deploy the web application to the GlassFish server. If GlassFish is not already running, NetBeans will start it.

32.6.4 Testing the WelcomeSOAP Web Service with GlassFish Application Server's Tester Web Page

Next, you'll test the `WelcomeSOAP` web service. We previously selected the GlassFish application server to execute this web application. This server can dynamically create a web page that allows you to test a web service's methods from a web browser. To use this capability:

- Expand the project's **Web Services** in the NetBeans **Projects** tab.
- Right click the web service class name (`WelcomeSOAP`) and select **Test Web Service**.

The GlassFish application server builds the **Tester** web page and loads it into your web browser. Figure 32.5 shows the **Tester** web page for the `WelcomeSOAP` web service. The web service's name is automatically the class name followed by **Service**.



Fig. 32.5 | Tester web page created by GlassFish for the `WelcomeSOAP` web service.

Once you've deployed the web service, you can also type the URL

```
http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?Tester
```

in your web browser to view the **Tester** web page. `WelcomeSOAPService` is the name that clients use to access the web service—this is simply the class name followed by **Service**.

To test `WelcomeSOAP`'s `welcome` web method, type your name in the text field to the right of the `welcome` button then click the button to invoke the method. Figure 32.6 shows the results of invoking `WelcomeSOAP`'s `welcome` method with the value `Paul`.

32.6 Publishing and Consuming SOAP-Based Web Services

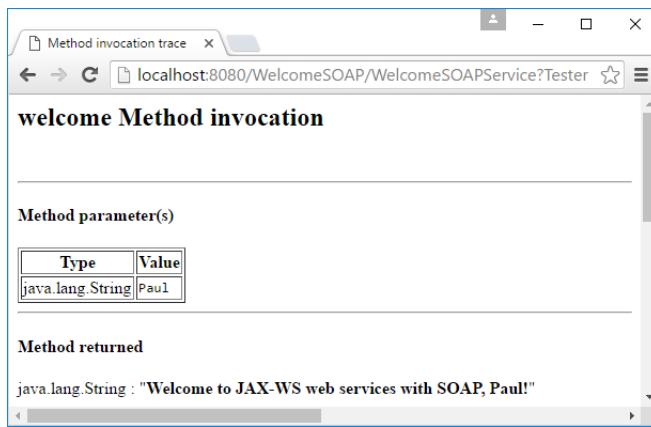


Fig. 32.6 | Results of testing WelcomeSOAP's welcome method.

Application Server Note

You can access the web service only when the application server is running. If NetBeans launches GlassFish for you, it will automatically shut it down when you close NetBeans. To keep it running, you can launch it independently of NetBeans before you deploy or run web applications. The GlassFish Quick Start Guide at

<https://glassfish.java.net/docs/4.0/quick-start-guide.pdf>

shows how to manually start and stop the server.

Testing the WelcomeSOAP Web Service from Another Computer

If your computer is connected to a network and allows HTTP requests, then you can test the web service from another computer on the network by typing the following URL (where *host* is the hostname or IP address of the computer on which the web service is deployed) into a browser on another computer:

<http://host:8080/WelcomeSOAP/WelcomeSOAPService?Tester>

32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)

To consume a web service, a client must determine its functionality and how to use it. For this purpose, web services normally contain a **service description**. This is an XML document that conforms to the **Web Service Description Language (WSDL)**—an XML vocabulary that defines the methods a web service makes available and how clients interact with them. The WSDL document also specifies lower-level information that clients might need, such as the required formats for requests and responses.

WSDL documents help applications determine how to interact with the web services described in the documents. You do not need to understand WSDL to take advantage of it—the GlassFish application server generates a web service's WSDL dynamically for you, and client tools can parse the WSDL to help create the client-side service endpoint interface class that a client uses to access the web service. Since GlassFish (and most other

32_12 Chapter 32 REST Web Services

servers) generate the WSDL dynamically, clients always receive a deployed web service's most up-to-date description. To access the `WelcomeSOAP` web service, the client code will need the following WSDL URL:

```
http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

Accessing the `WelcomeSOAP` Web Service's WSDL from Another Computer

Eventually, you'll want clients on other computers to use your web service. Such clients need the web service's WSDL, which they would access with the following URL:

```
http://host:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

where `host` is the hostname or IP address of the server that hosts the web service. As we discussed in Section 32.6.4, this works only if your computer allows HTTP connections from other computers—as is the case for publicly accessible web and application servers.

32.6.6 Creating a Client to Consume the `WelcomeSOAP` Web Service

Now you'll consume the web service from a client application. A web service client can be any type of application or even another web service. You enable a client application to consume a web service by **adding a web service reference** to the application.

Service Endpoint Interface (SEI)

An application that consumes a web service consists of an object of a service endpoint interface (SEI) class (sometimes called a *proxy class*) that's used to interact with the web service and a client application that consumes the web service by invoking methods on the service endpoint interface object. The client code invokes methods on the service endpoint interface object, which handles the details of passing method arguments to and receiving return values from the web service on the client's behalf. This communication can occur over a local network, over the Internet or even with a web service on the same computer. The web service performs the corresponding task and returns the results to the service endpoint interface object, which then returns the results to the client code. Figure 32.7 depicts the interactions among the client code, the SEI object and the web service. As you'll soon see, NetBeans creates these service endpoint interface classes for you.

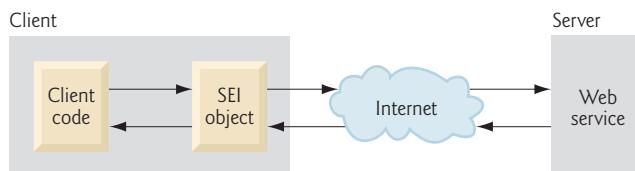


Fig. 32.7 | Interaction between a web service client and a web service.

Requests to and responses from web services created with JAX-WS (one of many different web service frameworks) are typically transmitted via SOAP. Any client capable of generating and processing SOAP messages can interact with a web service, regardless of the language in which the web service is written.

We now use NetBeans to create a client Java desktop GUI application. Then you'll add a web service reference to the project so the client can access the web service. When you add the reference, the IDE creates and compiles the **client-side artifacts**—the framework of Java code that supports the client-side service endpoint interface class. The client then calls methods on an object of the service endpoint interface class, which uses the rest of the artifacts to interact with the web service.

Creating a Desktop Application Project in NetBeans

Before performing the steps in this section, ensure that the `WelcomeSOAP` web service has been deployed and that the GlassFish application server is running (see Section 32.6.3). Perform the following steps to create a client Java desktop application in NetBeans:

1. Select **File > New Project...** to open the **New Project** dialog.
2. Select **Java** from the **Categories** list and **Java Application** from the **Projects** list, then click **Next >**.
3. Specify the name `WelcomeSOAPClient` in the **Project Name** field and uncheck the **Create Main Class** checkbox. Later, you'll add a subclass of `JFrame` that contains a `main` method.
4. Click **Finish** to create the project.

Step 2: Adding a Web Service Reference to an Application

Next, you'll add a web service reference to your application so that it can interact with the `WelcomeSOAP` web service. To add a web service reference, perform the following steps.

1. Right click the project name (`WelcomeSOAPClient`) in the NetBeans **Projects** tab and select **New > Web Service Client...** from the pop-up menu to display the **New Web Service Client** dialog.
2. In the **WSDL URL** field, specify the URL `http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL` (Fig. 32.8). This URL tells the IDE where to find the web service's WSDL description. [Note: If the GlassFish application server is located on a different computer, replace `localhost` with the hostname or IP address of that computer.] The IDE uses this WSDL description to generate the client-side artifacts that compose and support the service endpoint interface.
3. For the other options, leave the default settings, then click **Finish** to create the web service reference and dismiss the **New Web Service Client** dialog.

In the NetBeans **Projects** tab, the `WelcomeSOAPClient` project now contains a **Web Service References** folder with the `WelcomeSOAP` web service's service endpoint interface (Fig. 32.9). The service endpoint interface's name is listed as `WelcomeSOAPService`.

When you specify the web service you want to consume, NetBeans accesses and copies its WSDL information to a file in your project (named `WelcomeSOAPService.wsdl` in this example). You can view this file by double clicking the `WelcomeSOAPService` node in the project's **Web Service References** folder. If the web service changes, the client-side artifacts and the client's copy of the WSDL file can be regenerated by right clicking the `WelcomeSOAPService` node shown in Fig. 32.9 and selecting **Refresh....** Figure 32.9 also shows the IDE-generated client-side artifacts, which appear in the **Generated Sources (jax-ws)** folder.

32_14 Chapter 32 REST Web Services

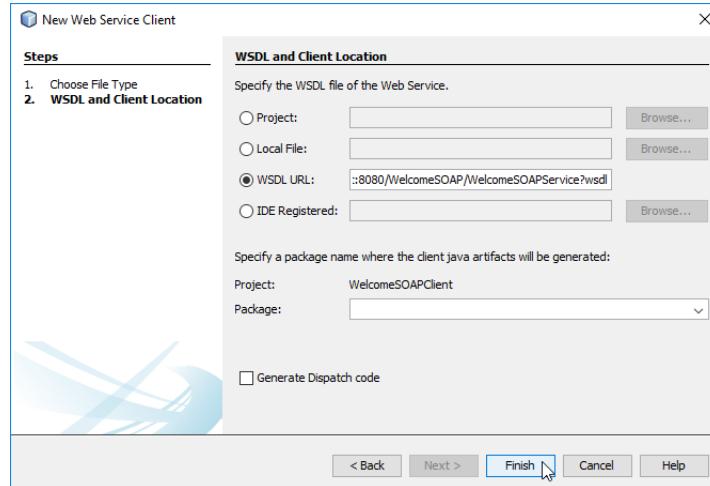


Fig. 32.8 | New Web Service Client dialog.

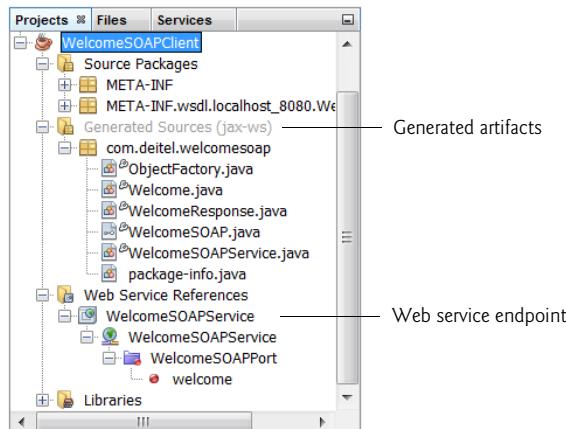


Fig. 32.9 | NetBeans Project tab after adding a web service reference to the project.

32.6.7 Consuming the WelcomeSOAP Web Service

For this example, we use a GUI application to interact with the WelcomeSOAP web service. To build the client application's GUI, add a subclass of `JFrame` to the project by performing the following steps:

1. Right click the project name (`WelcomeSOAPClient`) in the NetBeans Project tab and select `New > JFrame Form...` to display the `New JFrame Form` dialog.
2. Specify `WelcomeSOAPClientJFrame` in the `Class Name` field.
3. Specify `com.deitel.welcomesoapclient` in the `Package` field.
4. Click `Finish` to close the `New JFrame Form` dialog.

32.6 Publishing and Consuming SOAP-Based Web Services

32_15

Next, use the NetBeans GUI design tools to build the GUI shown in the sample screen captures at the end of Fig. 32.10. The GUI consists of a **Label**, a **Text Field** and a **Button**.

The application in Fig. 32.10 uses the `WelcomeSOAP` web service to display a welcome message to the user. To save space, we do not show the NetBeans autogenerated `initComponents` method, which contains the code that creates the GUI components, positions them and registers their event handlers. To view the complete source code, open the `WelcomeSOAPClientJFrame.java` file in this example's folder under `src\java\com\deitel\welcomesoapclient`. NetBeans places the GUI component instance-variable declarations at the end of the class (lines 114–116). Java allows instance variables to be declared anywhere in a class's body as long as they're placed outside the class's methods. We continue to declare our own instance variables at the top of the class.

```

1 // Fig. 32.10: WelcomeSOAPClientJFrame.java
2 // Client desktop application for the WelcomeSOAP web service.
3 package com.deitel.welcomesoapclient;
4
5 import com.deitel.welcomesoap>WelcomeSOAP;
6 import com.deitel.welcomesoap>WelcomeSOAPService;
7 import javax.swing.JOptionPane;
8
9 public class WelcomeSOAPClientJFrame extends javax.swing.JFrame
10 {
11     // references the service endpoint interface object (i.e., the proxy)
12     private WelcomeSOAP welcomeSOAPProxy;
13
14     // no-argument constructor
15     public WelcomeSOAPClientJFrame()
16     {
17         initComponents();
18
19         try
20         {
21             // create the objects for accessing the WelcomeSOAP web service
22             WelcomeSOAPService service = new WelcomeSOAPService();
23             welcomeSOAPProxy = service.getWelcomeSOAPPort();
24         }
25         catch (Exception exception)
26         {
27             exception.printStackTrace();
28             System.exit(1);
29         }
30     }
31
32     // The initComponents method is autogenerated by NetBeans and is called
33     // from the constructor to initialize the GUI. This method is not shown
34     // here to save space. Open WelcomeSOAPClientJFrame.java in this
35     // example's folder to view the complete generated code.
36

```

Fig. 32.10 | Client desktop application for the `WelcomeSOAP` web service. (Part I of 2.)

32_16 Chapter 32 REST Web Services

```

87     // call the web service with the supplied name and display the message
88     private void submitJButtonActionPerformed(
89         java.awt.event.ActionEvent evt)
90     {
91         String name = nameJTextField.getText(); // get name from JTextField
92
93         // retrieve the welcome string from the web service
94         String message = welcomeSOAPProxy.welcome(name);
95         JOptionPane.showMessageDialog(this, message,
96             "Welcome", JOptionPane.INFORMATION_MESSAGE);
97     }
98
99     // main method begins execution
100    public static void main(String args[])
101    {
102        java.awt.EventQueue.invokeLater(
103            new Runnable()
104            {
105                public void run()
106                {
107                    new WelcomeSOAPClientJFrame().setVisible(true);
108                }
109            }
110        );
111    }
112
113    // Variables declaration - do not modify
114    private javax.swing.JLabel nameJLabel;
115    private javax.swing.JTextField nameJTextField;
116    private javax.swing.JButton submitJButton;
117    // End of variables declaration
118 }

```



Fig. 32.10 | Client desktop application for the `WelcomeSOAP` web service. (Part 2 of 2.)

Lines 5–6 import the classes `WelcomeSOAP` and `WelcomeSOAPService` that enable the client application to interact with the web service. Notice that we do not have `import` declarations for most of the GUI components used in this example. When you create a GUI in NetBeans, it uses fully qualified class names (such as `javax.swing.JFrame` in line 9), so `import` declarations are unnecessary.

Line 12 declares a variable of type `WelcomeSOAP` that will refer to the service endpoint interface object. Line 22 in the constructor creates an object of type `WelcomeSOAPService`. Line 23 uses this object's `getWelcomeSOAPPort` method to obtain the `WelcomeSOAP` service endpoint interface object that the application uses to invoke the web service's methods.

The `Submit` button handler (lines 88–97) first retrieves the name the user entered from `nameJTextField`. It then calls the `welcome` method on the service endpoint interface

32.7 Publishing and Consuming REST-Based XML Web Services

32-17

object (line 94) to retrieve the welcome message from the web service. This object communicates with the web service on the client's behalf. Once the message has been retrieved, lines 95–96 display it in a message box by calling `JOptionPane`'s `showMessageDialog` method.

32.7 Publishing and Consuming REST-Based XML Web Services

The previous section used a service endpoint interface (proxy) object to pass data to and from a Java web service using the SOAP protocol. Now, we access a Java web service using the REST architecture. We recreate the `WelcomeSOAP` example to return data in plain XML format. You can create a **Web Application** project as you did in Section 32.6 to begin. Name the project `WelcomeRESTXML`.

32.7.1 Creating a REST-Based XML Web Service

NetBeans provides various templates for creating RESTful web services, including ones that can interact with databases on the client's behalf. In this chapter, we focus on simple RESTful web services. To create a RESTful web service:

1. Right-click the `WelcomeRESTXML` node in the **Projects** tab, and select **New > Other...** to display the **New File** dialog.
2. Select **Web Services** under **Categories**, then select **RESTful Web Services** from **Patterns** and click **Next >**.
3. Under **Select Pattern**, ensure **Simple Root Resource** is selected, and click **Next >**.
4. Set the **Resource Package** to `com.deitel.welcomerestxml`, the **Path** to `welcome` and the **Class Name** to `WelcomeRESTXMLResource`. Leave the **MIME Type** and **Representation Class** set to `application/xml` and `java.lang.String`, respectively. The correct configuration is shown in Fig. 32.11.
5. Click **Finish** to create the web service.

NetBeans generates the class and sets up the proper annotations. The class is placed in the project's **RESTful Web Services** folder. The code for the completed service is shown in Fig. 32.12. You'll notice that the completed code does not include some of the code generated by NetBeans. We removed the pieces that were unnecessary for this simple web service. The autogenerated `putXml` method is not necessary, because this example does not modify state on the server. The `UriInfo` instance variable is not needed, because we do not use HTTP query parameters. We also removed the autogenerated constructor, because we have no code to place in it.

Lines 6–9 contain the imports for the JAX-RS annotations that help define the RESTful web service. The `@Path annotation` on the `WelcomeRESTXMLResource` class (line 12) indicates the URI for accessing the web service. This URI is appended to the web application project's URL to invoke the service. Methods of the class can also use the `@Path annotation` (line 17). Parts of the path specified in curly braces indicate parameters—they're placeholders for values that are passed to the web service as part of the path.

32_18 Chapter 32 REST Web Services

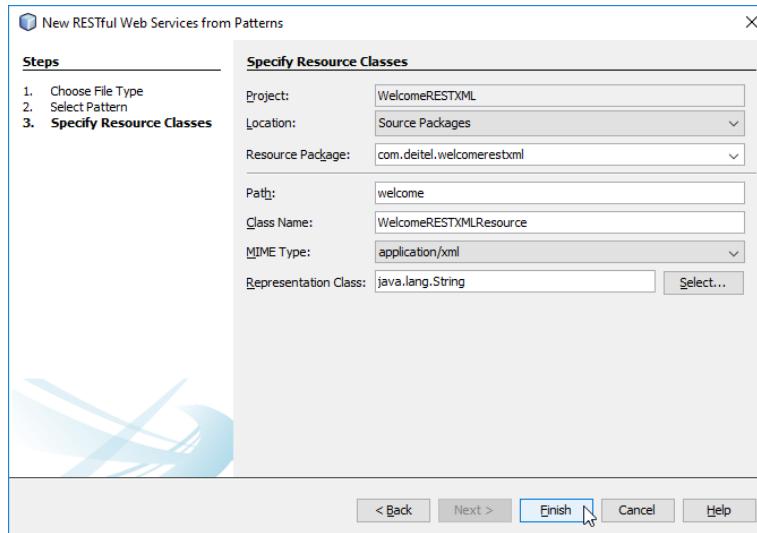


Fig. 32.11 | Creating the WelcomeRESTXML RESTful web service.

```

1 // Fig. 32.12: WelcomeRESTXMLResource.java
2 // REST web service that returns a welcome message as XML.
3 package com.deitel.welcomerestxml;
4
5 import java.io.StringWriter;
6 import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
7 import javax.ws.rs.Path; // annotation to specify path of resource
8 import javax.ws.rsPathParam; // annotation to get parameters from URI
9 import javax.ws.rs.Produces; // annotation to specify type of data
10 import javax.xml.bind.JAXB; // utility class for common JAXB operations
11
12 @Path("welcome") // URI used to access the resource
13 public class WelcomeRESTXMLResource
14 {
15     // retrieve welcome message
16     @GET // handles HTTP GET requests
17     @Path("{name}") // URI component containing parameter
18     @Produces("application/xml") // response formatted as XML
19     public String getXml(@PathParam("name") String name)
20     {
21         String message = "Welcome to JAX-RS web services with REST and " +
22             "XML, " + name + "!"; // our welcome message
23         StringWriter writer = new StringWriter();
24         JAXB.marshal(message, writer); // marshal String as XML
25         return writer.toString(); // return XML as String
26     }
27 }
```

Fig. 32.12 | REST web service that returns a welcome message as XML.

32.7 Publishing and Consuming REST-Based XML Web Services

32_19

The base path for the service is the project's `webresources` directory. For example, to get a welcome message for someone named John, the complete URL is

```
http://localhost:8080/WelcomeRESTXML/webresources/welcome/John
```

Arguments in a URL can be used as arguments to a web service method. To do so, you bind the parameters specified in the `@Path` specification to parameters of the web service method with the `@PathParam annotation`, as shown in line 19. When the request is received, the server passes the argument(s) in the URL to the appropriate parameter(s) in the web service method.

The `@GET annotation` denotes that this method is accessed via an HTTP GET request. The `putXml` method the IDE created for us had an `@PUT` annotation, which indicates that the method is accessed using the HTTP PUT method. Similar annotations exist for HTTP POST, DELETE and HEAD requests.

The `@Produces annotation` denotes the content type returned to the client. It's possible to have multiple methods with the same HTTP method and path but different `@Produces` annotations, and JAX-RS will call the method matching the content type requested by the client. Standard Java method overloading rules apply, so such methods must have different names. The `@Consumes annotation` for the autogenerated `putXml` method (which we deleted) restricts the content type that the web service will accept from a PUT operation.

Line 10 imports the `JAXB class` from package `javax.xml.bind`. **JAXB (Java Architecture for XML Binding)** is a set of classes for converting POJOs to and from XML. There are many related classes in the same package that implement the serializations we perform, but the `JAXB class` contains easy-to-use wrappers for common operations. After creating the welcome message (lines 21–22), we create a `StringWriter` (line 23) to which JAXB will output the XML. Line 24 calls the `JAXB class's static method marshal` to convert the `String` containing our message to XML format. Line 25 calls `StringWriter's toString` method to retrieve the XML text to return to the client.

Testing the RESTful Web Service

Section 32.6.4 demonstrated testing a SOAP service using GlassFish's `Tester` page. GlassFish does not provide a testing facility for RESTful services, but you can enter the web service's URL directly in your browser to test the web service. To do so:

1. First, deploy the web service's project. Right click the `WelcomeRESTXML` project in the NetBeans `Projects` tab and select `Deploy`. This will compile and deploy the web service, if you have not yet done so.
2. Open a web browser and enter the following URL in the browser's address bar:
`http://localhost:8080/WelcomeRESTXML/webresources/welcome/Paul`—you can replace `Paul` with your own name.

Once GlassFish deploys a REST web service, a client can access it on the server (in this case, `localhost` at port 8080) at the location

```
/ProjectName/webresources/methodName
```

If the method requires parameters, as in this example, each parameter follows the method name in the form

```
/argument
```

32_20 Chapter 32 REST Web Services

and the arguments are passed to the method's parameters in the same order as they're declared in the method's parameter list. So the URL

```
http://localhost:8080/WelcomeRESTXML/webresources/welcome/Paul
```

invokes the `WelcomeRESTXML` web service's `welcome` method and passes `Paul` to the method's `name` parameter. The web service then returns an XML response that's displayed directly in the web browser (Fig. 32.13).



Fig. 32.13 | Test page for the `WelcomeRESTXML` web service.

WADL

WADL (Web Application Description Language) has similar design goals to WSDL, but describes RESTful services instead of SOAP services. You can access this app's WADL at

```
http://localhost:8080/WelcomeRESTJSON/webresources/application.wadl
```

Client-code-generation tools can use this description to help implement a client that interacts with this web service.

32.7.2 Consuming a REST-Based XML Web Service

As we did with SOAP, we create a Java application that retrieves the welcome message from the web service and displays it to the user. First, create a Java application with the name `WelcomeRESTXMLClient`. RESTful web services do *not* require web service references, so you can begin building the GUI immediately by creating a `JFrame` form called `WelcomeRESTXMLClientJFrame` and placing it in the `com.deitel.welcomerestxmlclient` package. The GUI is identical to the one in Fig. 32.10, including the names of the GUI elements. To create the GUI quickly, you can simply copy and paste the GUI from the `Design` view of the `WelcomeSOAPClientJFrame` class and paste it into the `Design` view of the `WelcomeRESTXMLClientJFrame` class. Figure 32.14 contains the completed code.

```
1 // Fig. 32.14: WelcomeRESTXMLClientJFrame.java
2 // Client that consumes the WelcomeRESTXML service.
3 package com.deitel.welcomerestxmlclient;
4
5 import javax.swing.JOptionPane;
```

Fig. 32.14 | Client that consumes the `WelcomeRESTXML` service. (Part 1 of 3.)

32.7 Publishing and Consuming REST-Based XML Web Services

32_21

```

6  import javax.xml.bind.JAXB; // utility class for common JAXB operations
7
8  public class WelcomeRESTXMLClientJFrame extends javax.swing.JFrame
9  {
10     // no-argument constructor
11     public WelcomeRESTXMLClientJFrame()
12     {
13         initComponents();
14     }
15
16     // The initComponents method is autogenerated by NetBeans and is called
17     // from the constructor to initialize the GUI. This method is not shown
18     // here to save space. Open WelcomeRESTXMLClientJFrame.java in this
19     // example's folder to view the complete generated code.
20
21     // call the web service with the supplied name and display the message
22     private void submitJButtonActionPerformed(
23         java.awt.event.ActionEvent evt)
24     {
25         String name = nameJTextField.getText(); // get name from JTextField
26
27         // the URL for the REST service
28         String url = "http://localhost:8080/WelcomeRESTXML/" +
29             "webresources/welcome/" + name;
30
31         // read from URL and convert from XML to Java String
32         String message = JAXB.unmarshal(url, String.class);
33
34         // display the message to the user
35         JOptionPane.showMessageDialog(this, message,
36             "Welcome", JOptionPane.INFORMATION_MESSAGE);
37     }
38
39     // main method begins execution
40     public static void main(String args[])
41     {
42         java.awt.EventQueue.invokeLater(
43             new Runnable()
44             {
45                 public void run()
46                 {
47                     new WelcomeRESTXMLClientJFrame().setVisible(true);
48                 }
49             }
50         );
51     }
52
53     // Variables declaration - do not modify
54     private javax.swing.JLabel nameLabel;
55     private javax.swing.JTextField nameJTextField;
56     private javax.swing.JButton submitJButton;
57
58     // End of variables declaration
59 }

```

Fig. 32.14 | Client that consumes the WelcomeRESTXML service. (Part 2 of 3.)

32.22 Chapter 32 REST Web Services

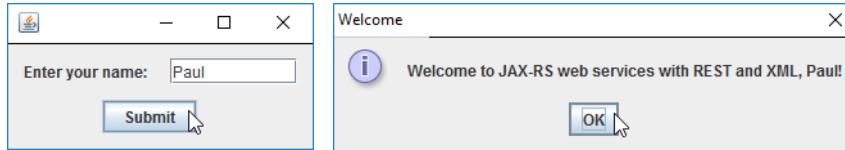


Fig. 32.14 | Client that consumes the WelcomeRESTXML service. (Part 3 of 3.)

You can access a RESTful web service with classes from Java API. As in the RESTful XML web service, we use the JAXB library. The JAXB class (imported on line 6) has a static `unmarshal` method that takes as arguments a filename or URL as a `String`, and a `Class<T>` object indicating the Java class to which the XML will be converted (line 83). In this example, the XML contains a `String` object, so we use the Java compiler shortcut `String.class` to create the `Class<String>` object we need as the second argument. The `String` returned from the call to the `unmarshal` method is then displayed to the user via `JOptionPane`'s `showMessageDialog` method (lines 86–87), as it was with the SOAP service. The URL used in this example to extract data from the web service matches the URL we used to test the web service directly in a web browser.

32.8 Publishing and Consuming REST-Based JSON Web Services

While XML was designed primarily as a document interchange format, JSON is designed as a *data* exchange format. Data structures in most programming languages do not map directly to XML constructs—for example, the distinction between elements and attributes is not present in programming-language data structures. JSON is a subset of the JavaScript programming language, and its components—objects, arrays, strings, numbers—can be easily mapped to constructs in Java and other programming languages.

The standard Java libraries do not currently provide capabilities for working with JSON, but there are many open-source JSON libraries for Java and other languages; you can find a list of them at json.org. We chose the Gson library from <https://github.com/google/gson>, which provides a simple way to convert POJOs to and from JSON. A JAR file containing the library can be downloaded from

```
http://search.maven.org/remotecontent?filepath=com/google/code/
gson/gson/2.7/gson-2.7.jar
```

32.8.1 Creating a REST-Based JSON Web Service

To begin, create a `WelcomeRESTJSON` web application, then create the web service by following the steps in Section 32.7.1. In *Step 4*, change the **Resource Package** to `com.deitel.welcomerestjson`, the **Class Name** to `WelcomeRESTJSONResource` and the **MIME Type** to `application/json`. Additionally, you must download the Gson library's JAR file, then add it to the project as a library. To add the JAR file to the project, right click your project's **Libraries** folder, select **Add JAR/Folder...** locate the downloaded Gson JAR file and click **Open**. The complete code for the service is shown in Fig. 32.15.

32.8 Publishing and Consuming REST-Based JSON Web Services

32_23

```

1 // Fig. 32.15: WelcomeRESTJSONResource.java
2 // REST web service that returns a welcome message as JSON.
3 package com.deitel.welcomerestjson;
4
5 import com.google.gson.Gson; // converts POJO to JSON and back again
6 import javax.ws.rs.GET; // annotation to indicate method uses HTTP GET
7 import javax.ws.rs.Path; // annotation to specify path of resource
8 import javax.ws.rsPathParam; // annotation to get parameters from URI
9 import javax.ws.rs.Produces; // annotation to specify type of data
10
11 @Path("welcome") // path used to access the resource
12 public class WelcomeRESTJSONResource
13 {
14     // retrieve welcome message
15     @GET // handles HTTP GET requests
16     @Path("{name}") // takes name as a path parameter
17     @Produces("application/json") // response formatted as JSON
18     public String getJson(@PathParam("name") String name)
19     {
20         // add welcome message to field of TextMessage object
21         TextMessage message = new TextMessage(); // create wrapper object
22         message.setMessage(String.format("%s, %s!",
23             "Welcome to JAX-RS web services with REST and JSON", name));
24
25         return new Gson().toJson(message); // return JSON-wrapped message
26     }
27 }
28
29 // private class that contains the message we wish to send
30 class TextMessage
31 {
32     private String message; // message we're sending
33
34     // returns the message
35     public String getMessage()
36     {
37         return message;
38     }
39
40     // sets the message
41     public void setMessage(String value)
42     {
43         message = value;
44     }
45 }
```

Fig. 32.15 | REST web service that returns a welcome message as JSON.

All the annotations and the basic structure of the `WelcomeRESTJSONResource` class are the same as REST XML example. The argument to the `@Produces` attribute (line 17) is "application/json". The `TextMessage` class (lines 30–45) addresses a difference between JSON and XML. JSON does not permit strings or numbers to stand on their

32.24 Chapter 32 REST Web Services

own—they must be encapsulated in a composite data type. So, we created class `TextMessage` to encapsulate the `String` representing the message.

When a client invokes this web service, line 21 creates the `TextMessage` object, then lines 22–23 set its contained message. Next, line 25 creates a `Gson` object (from package `com.google.gson.Gson`) and calls its `toJson` method to convert the `TextMessage` into its JSON `String` representation. We return this `String`, which is then sent back to the client in the web service's response. There are multiple overloads of the `toJson` method, such as one that sends its output to a `Writer` instead of returning a `String`.

RESTful services returning JSON can be tested in the same way as those returning XML. Follow the procedure outlined in Section 32.7.1, but use the URL

```
http://localhost:8080/WelcomeRESTJSON/webresources/welcome/Paul
```

to invoke the web service. In this case, the browser will display the JSON response

```
{"message": "Welcome to JAX-RS web services with REST and JSON, Paul!"}
```

32.8.2 Consuming a REST-Based JSON Web Service

We now create a Java application that retrieves the welcome message from the web service and displays it to the user. First, create a Java application with the name `WelcomeRESTJSONClient`. Then, create a `JFrame` form called `WelcomeRESTXMLClientJFrame` and place it in the `com.deitel.welcomerestjsonclient` package. The GUI is identical to the one in Fig. 32.10. To create the GUI quickly, copy it from the `Design` view of the `WelcomeSOAPClientJFrame` class and paste it into the `Design` view of the `WelcomeRESTJSONClientJFrame` class. Figure 32.16 contains the completed code.

```

1 // Fig. 32.16: WelcomeRESTJSONClientJFrame.java
2 // Client that consumes the WelcomeRESTJSON service.
3 package com.deitel.welcomerestjsonclient;
4
5 import com.google.gson.Gson; // converts POJO to JSON and back again
6 import java.io.InputStreamReader;
7 import java.net.URL;
8 import javax.swing.JOptionPane;
9
10 public class WelcomeRESTJSONClientJFrame extends javax.swing.JFrame
11 {
12     // no-argument constructor
13     public WelcomeRESTJSONClientJFrame()
14     {
15         initComponents();
16     }
17
18     // The initComponents method is autogenerated by NetBeans and is called
19     // from the constructor to initialize the GUI. This method is not shown
20     // here to save space. Open WelcomeRESTJSONClientJFrame.java in this
21     // example's folder to view the complete generated code.
22 }
```

Fig. 32.16 | Client that consumes the `WelcomeRESTJSON` service. (Part 1 of 3.)

32.8 Publishing and Consuming REST-Based JSON Web Services

32_25

```

73     // call the web service with the supplied name and display the message
74     private void submitJButtonActionPerformed(
75         java.awt.event.ActionEvent evt)
76     {
77         String name = nameJTextField.getText(); // get name from JTextField
78
79         // retrieve the welcome string from the web service
80         try
81         {
82             // the URL of the web service
83             String url = "http://localhost:8080/WelcomeRESTJSON/" +
84                 "webresources/welcome/" + name;
85
86             // open URL, using a Reader to convert bytes to chars
87             InputStreamReader reader =
88                 new InputStreamReader(new URL(url).openStream());
89
90             // parse the JSON back into a TextMessage
91             TextMessage message =
92                 new Gson().fromJson(reader, TextMessage.class);
93
94             // display message to the user
95             JOptionPane.showMessageDialog(this, message.getMessage(),
96                 "Welcome", JOptionPane.INFORMATION_MESSAGE);
97         }
98         catch (Exception exception)
99         {
100             exception.printStackTrace(); // show exception details
101         }
102     }
103
104    // main method begin execution
105    public static void main(String args[])
106    {
107        java.awt.EventQueue.invokeLater(
108            new Runnable()
109            {
110                public void run()
111                {
112                    new WelcomeRESTJSONClientJFrame().setVisible(true);
113                }
114            }
115        );
116    }
117
118    // Variables declaration - do not modify
119    private javax.swing.JLabel nameJLabel;
120    private javax.swing.JTextField nameJTextField;
121    private javax.swing.JButton submitJButton;
122    // End of variables declaration
123 }
124

```

Fig. 32.16 | Client that consumes the WelcomeRESTJSON service. (Part 2 of 3.)

32.26 Chapter 32 REST Web Services

```

125 // private class that contains the message we are receiving
126 class TextMessage
127 {
128     private String message; // message we're receiving
129
130     // returns the message
131     public String getMessage()
132     {
133         return message;
134     }
135
136     // sets the message
137     public void setMessage(String value)
138     {
139         message = value;
140     }
141 }
```

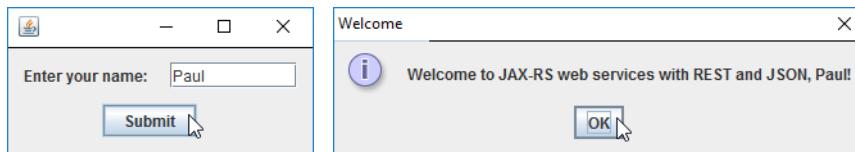


Fig. 32.16 | Client that consumes the WelcomeRESTJSON service. (Part 3 of 3.)

Lines 83–84 create the URL String that is used to invoke the web service. Lines 87–88 create a URL object using this String, then call the URL’s `openStream` method to invoke the web service and obtain an `InputStream` from which the client can read the response. The `InputStream` is wrapped in an `InputStreamReader` so it can be passed as the first argument to the `Gson` class’s `fromJson` method. This method is overloaded. The version we use takes as arguments a `Reader` from which to read a JSON String and a `Class<T>` object indicating the Java class to which the JSON String will be converted (line 92). In this example, the JSON String contains a `TextMessage` object, so we use the Java compiler shortcut `TextMessage.class` to create the `Class<TextMessage>` object we need as the second argument. Lines 95–96 display the message in the `TextMessage` object.

The `TextMessage` classes in the web service and client are unrelated. Technically, the client can be written in any programming language, so the manner in which a response is processed can vary greatly. Since our client is written in Java, we duplicated the `TextMessage` class in the client so we could easily convert the JSON object back to Java.

32.9 Session Tracking in a SOAP Web Service

Section 30.8 described the advantages of using session tracking to maintain client-state information so you can personalize the users’ browsing experiences. Now we’ll incorporate *session tracking* into a web service. Suppose a client application needs to call several methods from the same web service, possibly several times each. In such a case, it can be beneficial for the web service to maintain state information for the client, thus eliminating the need for client information to be passed between the client and the web service multiple

32.9 Session Tracking in a SOAP Web Service

32_27

times. For example, a web service that provides local restaurant reviews could store the client user's street address during the initial request, then use it to return personalized, localized results in subsequent requests. Storing session information also enables a web service to distinguish between clients.

32.9.1 Creating a Blackjack Web Service

Our next example is a web service that assists you in developing a blackjack card game. The `Blackjack` web service (Fig. 32.17) provides web methods to shuffle a deck of cards, deal a card from the deck and evaluate a hand of cards. After presenting the web service, we use it to serve as the dealer for a game of blackjack (Fig. 32.18). The `Blackjack` web service uses an `HttpSession` object to maintain a unique deck of cards for each client application. Several clients can use the service at the same time, but web method calls made by a specific client use only the deck of cards stored in that client's session. Our example uses the following blackjack rules:

Two cards each are dealt to the dealer and the player. The player's cards are dealt face up. Only the first of the dealer's cards is dealt face up. Each card has a value. A card numbered 2 through 10 is worth its face value. Jacks, queens and kings each count as 10. Aces can count as 1 or 11—whichever value is more beneficial to the player (as we'll soon see). If the sum of the player's two initial cards is 21 (i.e., the player was dealt a card valued at 10 and an ace, which counts as 11 in this situation), the player has "blackjack" and immediately wins the game—if the dealer does not also have blackjack (which would result in a "push"—i.e., a tie). Otherwise, the player can begin taking additional cards one at a time. These cards are dealt face up, and the player decides when to stop taking cards. If the player "busts" (i.e., the sum of the player's cards exceeds 21), the game is over and the player loses. When the player is satisfied with the current set of cards, the player "stands" (i.e., stops taking cards), and the dealer's hidden card is revealed. If the dealer's total is 16 or less, the dealer must take another card; otherwise, the dealer must stand. The dealer must continue taking cards until the sum of the dealer's cards is greater than or equal to 17. If the dealer exceeds 21, the player wins. Otherwise, the hand with the higher point total wins. If the dealer and the player have the same point total, the game is a "push," and no one wins. The value of an ace for a dealer depends on the dealer's other card(s) and the casino's house rules. A dealer typically must hit for totals of 16 or less and must stand for totals of 17 or more. However, for a "soft 17"—a hand with a total of 17 with one ace counted as 11—some casinos require the dealer to hit and some require the dealer to stand (we require the dealer to stand). Such a hand is known as a "soft 17" because taking another card cannot bust the hand.

The web service (Fig. 32.17) stores each card as a `String` consisting of a number, 1–13, representing the card's face (ace through king, respectively), followed by a space and a digit, 0–3, representing the card's suit (hearts, diamonds, clubs or spades, respectively). For example, the jack of clubs is represented as "11 2" and the two of hearts as "2 0". To create and deploy this web service, follow the steps that we presented in Sections 32.6.2–32.6.3 for the `WelcomeSOAP` service.

32_28 Chapter 32 REST Web Services

```

1 // Fig. 32.17: Blackjack.java
2 // Blackjack web service that deals cards and evaluates hands
3 package com.deitel.blackjack;
4
5 import com.sun.xml.ws.developer.servlet.HttpSessionScope;
6 import java.util.ArrayList;
7 import java.util.Random;
8 import javax.jws.WebMethod;
9 import javax.jws.WebParam;
10 import javax.jws.WebService;
11
12 @ HttpSessionScope // enable web service to maintain session state
13 @WebService()
14 public class Blackjack
15 {
16     private ArrayList<String> deck; // deck of cards for one user session
17     private static final Random randomObject = new Random();
18
19     // deal one card
20     @WebMethod(operationName = "dealCard")
21     public String dealCard()
22     {
23         String card = "";
24         card = deck.get(0); // get top card of deck
25         deck.remove(0); // remove top card of deck
26         return card;
27     }
28
29     // shuffle the deck
30     @WebMethod(operationName = "shuffle")
31     public void shuffle()
32     {
33         // create new deck when shuffle is called
34         deck = new ArrayList<String>();
35
36         // populate deck of cards
37         for (int face = 1; face <= 13; face++) // loop through faces
38             for (int suit = 0; suit <= 3; suit++) // loop through suits
39                 deck.add(face + " " + suit); // add each card to deck
40
41         String tempCard; // holds card temporarily during swapping
42         int index; // index of randomly selected card
43
44         for (int i = 0; i < deck.size() ; i++) // shuffle
45         {
46             index = randomObject.nextInt(deck.size() - 1);
47
48             // swap card at position i with randomly selected card
49             tempCard = deck.get(i);
50             deck.set(i, deck.get(index));
51             deck.set(index, tempCard);
52         }
53     }
}

```

Fig. 32.17 | Blackjack web service that deals cards and evaluates hands. (Part I of 2.)

32.9 Session Tracking in a SOAP Web Service

32_29

```

54
55     // determine a hand's value
56     @WebMethod(operationName = "getHandValue")
57     public int getHandValue(@WebParam(name = "hand") String hand)
58     {
59         // split hand into cards
60         String[] cards = hand.split("\t");
61         int total = 0; // total value of cards in hand
62         int face; // face of current card
63         int aceCount = 0; // number of aces in hand
64
65         for (int i = 0; i < cards.length; i++)
66         {
67             // parse string and get first int in String
68             face = Integer.parseInt(
69                 cards[i].substring(0, cards[i].indexOf(" ")));
70
71             switch (face)
72             {
73                 case 1: // if ace, increment aceCount
74                     ++aceCount;
75                     break;
76                 case 11: // jack
77                 case 12: // queen
78                 case 13: // king
79                     total += 10;
80                     break;
81                 default: // otherwise, add face
82                     total += face;
83                     break;
84             }
85         }
86
87         // calculate optimal use of aces
88         if (aceCount > 0)
89         {
90             // if possible, count one ace as 11
91             if (total + 11 + aceCount - 1 <= 21)
92                 total += 11 + aceCount - 1;
93             else // otherwise, count all aces as 1
94                 total += aceCount;
95         }
96
97         return total;
98     }
99 }
```

Fig. 32.17 | Blackjack web service that deals cards and evaluates hands. (Part 2 of 2.)*Session Tracking in Web Services: @HttpSessionScope Annotation*

In JAX-WS 2.2, it's easy to enable session tracking in a web service. You simply precede your web service class with the **@HttpSessionScope annotation**. This annotation is located in package `com.sun.xml.ws.developer.servlet`. To use this package you must add the JAX-WS 2.2 library to your project. To do so, right click the **Libraries** node in your Black-

32_30 Chapter 32 REST Web Services

jack web application project and select **Add Library**.... Then, in the dialog that appears, locate and select **JAX-WS 2.2**, then click **Add Library**. Once a web service is annotated with `@HttpSessionScope`, the server automatically maintains a separate instance of the class for each client session. Thus, the `deck` instance variable (line 16) will be maintained separately for each client.

Client Interactions with the Blackjack Web Service

A client first calls the `Blackjack` web service's `shuffle` web method (lines 30–53) to create a new deck of cards (line 34), populate it (lines 37–39) and shuffle it (lines 41–52). Lines 37–39 generate `Strings` in the form "`face suit`" to represent each possible card in the deck.

Lines 20–27 define the `dealCard` web method. Method `shuffle` *must* be called before method `dealCard` is called the first time for a client—otherwise, `deck` could be `null`. The method gets the top card from the deck (line 24), removes it from the deck (line 25) and returns the card's value as a `String` (line 26). Without using session tracking, the deck of cards would need to be passed back and forth with each method call. Session tracking makes the `dealCard` method easy to call (it requires no arguments) and eliminates the overhead of sending the deck over the network multiple times.

Method `getHandValue` (lines 56–98) determines the total value of the cards in a hand by trying to attain the highest score possible without going over 21. Recall that an ace can be counted as either 1 or 11, and all face cards count as 10. This method does not use the `session` object, because the deck of cards is not used in this method.

As you'll soon see, the client application maintains a hand of cards as a `String` in which each card is separated by a tab character. Line 60 splits the hand of cards (represented by `hand`) into individual cards by calling `String` method `split` and passing to it a `String` containing the delimiter characters (in this case, just a tab). Method `split` uses the delimiter characters to separate tokens in the `String`. Lines 65–85 count the value of each card. Lines 68–69 retrieve the first integer—the face—and use that value in the `switch` statement (lines 71–84). If the card is an ace, the method increments variable `aceCount`. We discuss how this variable is used shortly. If the card is an 11, 12 or 13 (jack, queen or king), the method adds 10 to the total value of the hand (line 79). If the card is anything else, the method increases the total by that value (line 82).

Because an ace can have either of two values, additional logic is required to process aces. Lines 88–95 process the aces after all the other cards. If a hand contains several aces, only one ace can be counted as 11. The condition in line 91 determines whether counting one ace as 11 and the rest as 1 will result in a total that does not exceed 21. If this is possible, line 92 adjusts the total accordingly. Otherwise, line 94 adjusts the total, counting each ace as 1.

Method `getHandValue` maximizes the value of the current cards without exceeding 21. Imagine, for example, that the dealer has a 7 and receives an ace. The new total could be either 8 or 18. However, `getHandValue` always maximizes the value of the cards without going over 21, so the new total is 18.

32.9.2 Consuming the Blackjack Web Service

The blackjack application in Fig. 32.18 keeps track of the player's and dealer's cards, and the web service tracks the cards that have been dealt. The constructor (lines 34–83) sets up the GUI (line 36), changes the window's background color (line 40) and creates the `Blackjack` web service's service endpoint interface object (lines 46–47). In the GUI, each

32.9 Session Tracking in a SOAP Web Service

32_31

player has 11 `JLabels`—the maximum number of cards that can be dealt without automatically exceeding 21 (i.e., four aces, four twos and three threes). These `JLabels` are placed in an `ArrayList` of `JLabels` (lines 59–82), so we can index the `ArrayList` during the game to determine the `JLabel` that will display a particular card image.

```

1 // Fig. 32.18: BlackjackGameJFrame.java
2 // Blackjack game that uses the Blackjack Web Service.
3 package com.deitel.blackjackclient;
4
5 import com.deitel.blackjack.Blackjack;
6 import com.deitel.blackjack.BlackjackService;
7 import java.awt.Color;
8 import java.util.ArrayList;
9 import javax.swing.ImageIcon;
10 import javax.swing.JLabel;
11 import javax.swing.JOptionPane;
12 import javax.xml.ws.BindingProvider;
13
14 public class BlackjackGameJFrame extends javax.swing.JFrame
15 {
16     private String playerCards;
17     private String dealerCards;
18     private ArrayList<JLabel> cardboxes; // list of card image JLabels
19     private int currentPlayerCard; // player's current card number
20     private int currentDealerCard; // blackjackProxy's current card number
21     private BlackjackService blackjackService; // used to obtain proxy
22     private Blackjack blackjackProxy; // used to access the web service
23
24     // enum of game states
25     private enum GameStatus
26     {
27         PUSH, // game ends in a tie
28         LOSE, // player loses
29         WIN, // player wins
30         BLACKJACK // player has blackjack
31     }
32
33     // no-argument constructor
34     public BlackjackGameJFrame()
35     {
36         initComponents();
37
38         // due to a bug in NetBeans, we must change the JFrame's background
39         // color here rather than in the designer
40         getContentPane().setBackground(new Color(0, 180, 0));
41
42         // initialize the blackjack proxy
43         try
44         {
45             // create the objects for accessing the Blackjack web service
46             blackjackService = new BlackjackService();
47             blackjackProxy = blackjackService.getBlackjackPort();

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part I of 10.)

32_32 Chapter 32 REST Web Services

```

48         // enable session tracking
49         ((BindingProvider) blackjackProxy).getRequestContext().put(
50             BindingProvider.SESSION_MAINTAIN_PROPERTY, true);
51     }
52   }
53   catch (Exception e)
54   {
55     e.printStackTrace();
56   }
57
58   // add JLabels to cardBoxes ArrayList for programmatic manipulation
59   cardboxes = new ArrayList<JLabel>();
60
61   cardboxes.add(dealerCard1JLabel);
62   cardboxes.add(dealerCard2JLabel);
63   cardboxes.add(dealerCard3JLabel);
64   cardboxes.add(dealerCard4JLabel);
65   cardboxes.add(dealerCard5JLabel);
66   cardboxes.add(dealerCard6JLabel);
67   cardboxes.add(dealerCard7JLabel);
68   cardboxes.add(dealerCard8JLabel);
69   cardboxes.add(dealerCard9JLabel);
70   cardboxes.add(dealerCard10JLabel);
71   cardboxes.add(dealerCard11JLabel);
72   cardboxes.add(playerCard1JLabel);
73   cardboxes.add(playerCard2JLabel);
74   cardboxes.add(playerCard3JLabel);
75   cardboxes.add(playerCard4JLabel);
76   cardboxes.add(playerCard5JLabel);
77   cardboxes.add(playerCard6JLabel);
78   cardboxes.add(playerCard7JLabel);
79   cardboxes.add(playerCard8JLabel);
80   cardboxes.add(playerCard9JLabel);
81   cardboxes.add(playerCard10JLabel);
82   cardboxes.add(playerCard11JLabel);
83 }
84
85 // play the dealer's hand
86 private void dealerPlay()
87 {
88   try
89   {
90     // while the value of the dealers's hand is below 17
91     // the dealer must continue to take cards
92     String[] cards = dealerCards.split("\t");
93
94     // display dealer's cards
95     for (int i = 0; i < cards.length; i++)
96     {
97       displayCard(i, cards[i]);
98     }
99

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 2 of 10.)

32.9 Session Tracking in a SOAP Web Service

32_33

```

100     while (blackjackProxy.getHandValue(dealerCards) < 17)
101     {
102         String newCard = blackjackProxy.dealCard(); // deal new card
103         dealerCards += "\t" + newCard; // deal new card
104         displayCard(currentDealerCard, newCard);
105         ++currentDealerCard;
106         JOptionPane.showMessageDialog(this, "Dealer takes a card",
107             "Dealer's turn", JOptionPane.PLAIN_MESSAGE);
108     }
109
110     int dealersTotal = blackjackProxy.getHandValue(dealerCards);
111     int playersTotal = blackjackProxy.getHandValue(playerCards);
112
113     // if dealer busted, player wins
114     if (dealersTotal > 21)
115     {
116         gameOver(GameStatus.WIN);
117         return;
118     }
119
120     // if dealer and player are below 21
121     // higher score wins, equal scores is a push
122     if (dealersTotal > playersTotal)
123     {
124         gameOver(GameStatus.LOSE);
125     }
126     else if (dealersTotal < playersTotal)
127     {
128         gameOver(GameStatus.WIN);
129     }
130     else
131     {
132         gameOver(GameStatus.PUSH);
133     }
134 }
135 catch (Exception e)
136 {
137     e.printStackTrace();
138 }
139 }
140
141 // displays the card represented by cardValue in specified JLabel
142 private void displayCard(int card, String cardValue)
143 {
144     try
145     {
146         // retrieve correct JLabel from cardBoxes
147         JLabel displayLabel = cardboxes.get(card);
148
149         // if string representing card is empty, display back of card
150         if (cardValue.equals(""))
151     {

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 3 of 10.)

32_34 Chapter 32 REST Web Services

```

152         displayLabel.setIcon(new ImageIcon(getClass().getResource(
153             "/com/deitel/java/blackjackclient/"
154             + "blackjack_images/cardback.png")));
155         return;
156     }
157
158     // retrieve the face value of the card
159     String face = cardValue.substring(0, cardValue.indexOf(" "));
160
161     // retrieve the suit of the card
162     String suit =
163         cardValue.substring(cardValue.indexOf(" ") + 1);
164
165     char suitLetter; // suit letter used to form image file
166
167     switch (Integer.parseInt(suit))
168     {
169         case 0: // hearts
170             suitLetter = 'h';
171             break;
172         case 1: // diamonds
173             suitLetter = 'd';
174             break;
175         case 2: // clubs
176             suitLetter = 'c';
177             break;
178         default: // spades
179             suitLetter = 's';
180             break;
181     }
182
183     // set image for displayLabel
184     displayLabel.setIcon(new ImageIcon(getClass().getResource(
185             "/com/deitel/java/blackjackclient/blackjack_images/" +
186             face + suitLetter + ".png")));
187 }
188 catch (Exception e)
189 {
190     e.printStackTrace();
191 }
192 }
193
194 // displays all player cards and shows appropriate message
195 private void gameOver(GameStatus winner)
196 {
197     String[] cards = dealerCards.split("\t");
198
199     // display blackjackProxy's cards
200     for (int i = 0; i < cards.length; i++)
201     {
202         displayCard(i, cards[i]);
203     }
204 }
```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 4 of 10.)

32.9 Session Tracking in a SOAP Web Service

32_35

```

205      // display appropriate status image
206      if (winner == GameStatus.WIN)
207      {
208          statusJLabel.setText("You win!");
209      }
210      else if (winner == GameStatus.LOSE)
211      {
212          statusJLabel.setText("You lose.");
213      }
214      else if (winner == GameStatus.PUSH)
215      {
216          statusJLabel.setText("It's a push.");
217      }
218      else // blackjack
219      {
220          statusJLabel.setText("Blackjack!");
221      }
222
223      // display final scores
224      int dealersTotal = blackjackProxy.getHandValue(dealerCards);
225      int playersTotal = blackjackProxy.getHandValue(playerCards);
226      dealerTotalJLabel.setText("Dealer: " + dealersTotal);
227      playerTotalJLabel.setText("Player: " + playersTotal);
228
229      // reset for new game
230      standJButton.setEnabled(false);
231      hitJButton.setEnabled(false);
232      dealJButton.setEnabled(true);
233 }
234
235 // The initComponents method is autogenerated by NetBeans and is called
236 // from the constructor to initialize the GUI. This method is not shown
237 // here to save space. Open BlackjackGameJFrame.java in this
238 // example's folder to view the complete generated code
239
542 // handles dealJButton click
543 private void dealJButtonActionPerformed(
544     java.awt.event.ActionEvent evt)
545 {
546     String card; // stores a card temporarily until it's added to a hand
547
548     // clear card images
549     for (int i = 0; i < cardboxes.size(); i++)
550     {
551         cardboxes.get(i).setIcon(null);
552     }
553
554     statusJLabel.setText("");
555     dealerTotalJLabel.setText("");
556     playerTotalJLabel.setText("");
557
558     // create a new, shuffled deck on remote machine
559     blackjackProxy.shuffle();

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 5 of 10.)

32_36 Chapter 32 REST Web Services

```

560
561     // deal two cards to player
562     playerCards = blackjackProxy.dealCard(); // add first card to hand
563     displayCard(11, playerCards); // display first card
564     card = blackjackProxy.dealCard(); // deal second card
565     displayCard(12, card); // display second card
566     playerCards += "\t" + card; // add second card to hand
567
568     // deal two cards to blackjackProxy, but only show first
569     dealerCards = blackjackProxy.dealCard(); // add first card to hand
570     displayCard(0, dealerCards); // display first card
571     card = blackjackProxy.dealCard(); // deal second card
572     displayCard(1, ""); // display back of card
573     dealerCards += "\t" + card; // add second card to hand
574
575     standJButton.setEnabled(true);
576     hitJButton.setEnabled(true);
577     dealJButton.setEnabled(false);
578
579     // determine the value of the two hands
580     int dealersTotal = blackjackProxy.getHandValue(dealerCards);
581     int playersTotal = blackjackProxy.getHandValue(playerCards);
582
583     // if hands both equal 21, it is a push
584     if (playersTotal == dealersTotal && playersTotal == 21)
585     {
586         gameOver(GameStatus.PUSH);
587     }
588     else if (dealersTotal == 21) // blackjackProxy has blackjack
589     {
590         gameOver(GameStatus.LOSE);
591     }
592     else if (playersTotal == 21) // blackjack
593     {
594         gameOver(GameStatus.BLACKJACK);
595     }
596
597     // next card for blackjackProxy has index 2
598     currentDealerCard = 2;
599
600     // next card for player has index 13
601     currentPlayerCard = 13;
602 }
603
604     // handles standJButton click
605     private void hitJButtonActionPerformed(
606         java.awt.event.ActionEvent evt)
607     {
608         // get player another card
609         String card = blackjackProxy.dealCard(); // deal new card
610         playerCards += "\t" + card; // add card to hand
611

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 6 of 10.)

32.9 Session Tracking in a SOAP Web Service

32_37

```

612      // update GUI to display new card
613      displayCard(currentPlayerCard, card);
614      ++currentPlayerCard;
615
616      // determine new value of player's hand
617      int total = blackjackProxy.getHandValue(playerCards);
618
619      if (total > 21) // player busts
620      {
621          gameOver(GameStatus.LOSE);
622      }
623      else if (total == 21) // player cannot take any more cards
624      {
625          hitJButton.setEnabled(false);
626          dealerPlay();
627      }
628  }
629
630  // handles standJButton click
631  private void standJButtonActionPerformed(
632      java.awt.event.ActionEvent evt)
633  {
634      standJButton.setEnabled(false);
635      hitJButton.setEnabled(false);
636      dealJButton.setEnabled(true);
637      dealerPlay();
638  }
639
640  // begins application execution
641  public static void main(String args[])
642  {
643      java.awt.EventQueue.invokeLater(
644          new Runnable()
645          {
646              public void run()
647              {
648                  new BlackjackGameJFrame().setVisible(true);
649              }
650          }
651      );
652  }
653
654  // Variables declaration - do not modify
655  private javax.swing.JButton dealJButton;
656  private javax.swing.JLabel dealerCard10JLabel;
657  private javax.swing.JLabel dealerCard11JLabel;
658  private javax.swing.JLabel dealerCard1JLabel;
659  private javax.swing.JLabel dealerCard2JLabel;
660  private javax.swing.JLabel dealerCard3JLabel;
661  private javax.swing.JLabel dealerCard4JLabel;
662  private javax.swing.JLabel dealerCard5JLabel;
663  private javax.swing.JLabel dealerCard6JLabel;
664  private javax.swing.JLabel dealerCard7JLabel;

```

Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 7 of 10.)

32_38 Chapter 32 REST Web Services

```

665  private javax.swing.JLabel dealerCard8JLabel;
666  private javax.swing.JLabel dealerCard9JLabel;
667  private javax.swing.JLabel dealerJLabel;
668  private javax.swing.JLabel dealerTotalJLabel;
669  private javax.swing.JButton hitJButton;
670  private javax.swing.JLabel playerCard10JLabel;
671  private javax.swing.JLabel playerCard11JLabel;
672  private javax.swing.JLabel playerCard1JLabel;
673  private javax.swing.JLabel playerCard2JLabel;
674  private javax.swing.JLabel playerCard3JLabel;
675  private javax.swing.JLabel playerCard4JLabel;
676  private javax.swing.JLabel playerCard5JLabel;
677  private javax.swing.JLabel playerCard6JLabel;
678  private javax.swing.JLabel playerCard7JLabel;
679  private javax.swing.JLabel playerCard8JLabel;
680  private javax.swing.JLabel playerCard9JLabel;
681  private javax.swing.JLabel playerJLabel;
682  private javax.swing.JLabel playerTotalJLabel;
683  private javax.swing.JButton standJButton;
684  private javax.swing.JLabel statusJLabel;
685 // End of variables declaration
686 }

```

- a) Dealer and player hands after the user clicks the **Deal** JButton.

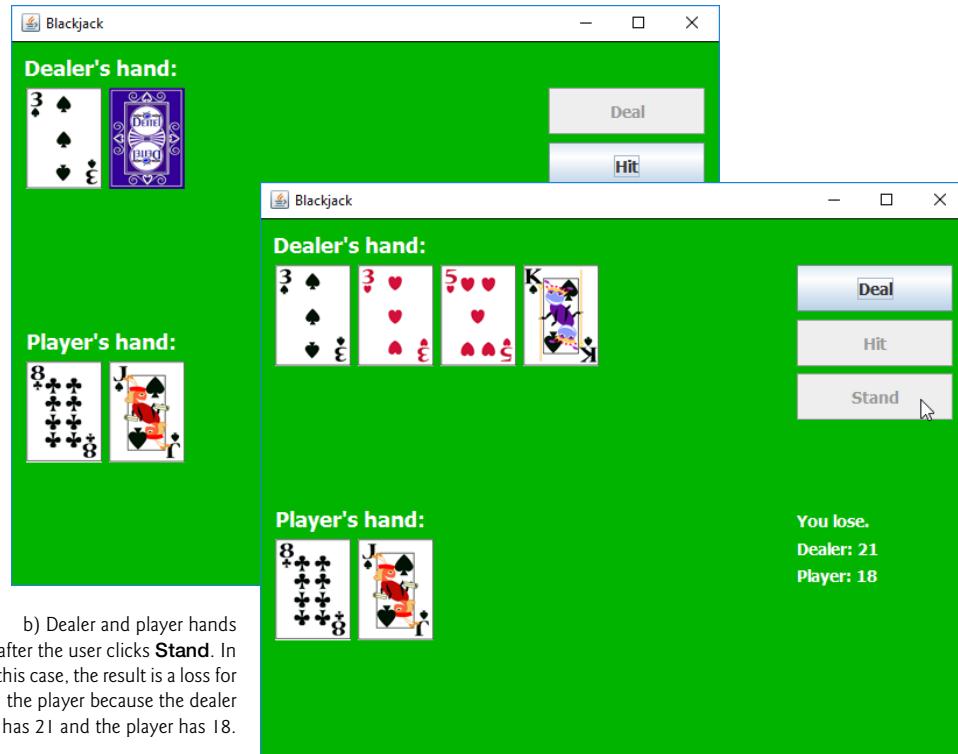
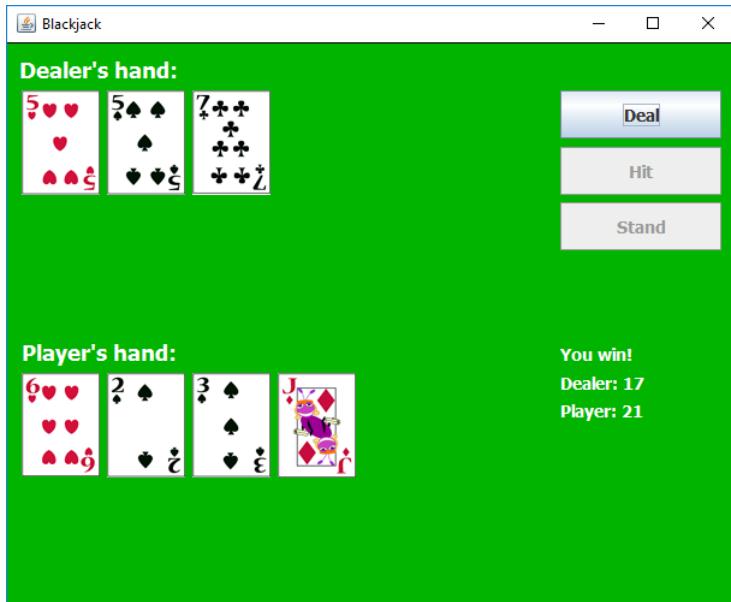


Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 8 of 10.)

32.9 Session Tracking in a SOAP Web Service

32_39

c) Hands after the user clicks **Hit** twice and draws 21. In this case, the player wins with the higher hand.



d) Hands after the player is dealt blackjack.



Fig. 32.18 | Blackjack game that uses the `Blackjack` web service. (Part 9 of 10.)

32_40 Chapter 32 REST Web Services

e) Hands after the dealer is dealt blackjack



Fig. 32.18 | Blackjack game that uses the Blackjack web service. (Part 10 of 10.)

Configuring the Client for Session Tracking

When interacting with a JAX-WS web service that performs session tracking, the client application must indicate whether it wants to allow the web service to maintain session information. Lines 50–51 in the constructor perform this task. We first cast the service endpoint interface object to interface type `BindingProvider`. A `BindingProvider` enables the client to manipulate the request information that will be sent to the server. This information is stored in an object that implements interface `RequestContext`. The `BindingProvider` and `RequestContext` are part of the framework that is created by the IDE when you add a web service client to the application. Next, we invoke the `BindingProvider`'s `get RequestContext` method to obtain the `RequestContext` object. Then we call the `RequestContext`'s `put` method to set the property

```
BindingProvider.SESSION_MAINTAIN_PROPERTY
```

to true. This enables the client side of the session-tracking mechanism, so that the web service knows which client is invoking the service's web methods.

Method `gameOver`

Method `gameOver` (lines 195–233) displays all the dealer's cards, shows the appropriate message in `statusJLabel` and displays the final point totals of both the dealer and the player. Method `gameOver` receives as an argument a member of the `GameStatus` enum (defined in lines 25–31). The enum constants represent whether the player tied, lost or won the game; its four members are `PUSH`, `LOSE`, `WIN` and `BLACKJACK`.

32.9 Session Tracking in a SOAP Web Service

32_41

Method dealJButtonActionPerformed

When the player clicks the **Deal JButton**, method `dealJButtonActionPerformed` (lines 543–602) clears all of the `JLabels` that display cards or game status information. Next, the deck is shuffled (line 559), and the player and dealer receive two cards each (lines 562–573). Lines 580–581 then total each hand. If the player and the dealer both obtain scores of 21, the program calls method `gameOver`, passing `GameStatus.PUSH` (line 586). If only the dealer has 21, the program passes `GameStatus.LOSE` to method `gameOver` (line 590). If only the player has 21 after the first two cards are dealt, the program passes `GameStatus.BLACKJACK` to method `gameOver` (line 594).

Method hitJButtonActionPerformed

If `dealJButtonActionPerformed` does not call `gameOver`, the player can take more cards by clicking the **Hit JButton**, which calls `hitJButtonActionPerformed` in lines 605–628. Each time a player clicks **Hit**, the program deals the player one more card (line 609) and displays it in the GUI (line 613). If the player exceeds 21, the game is over and the player loses (line 621). If the player has exactly 21, the player is not allowed to take any more cards (line 625), and method `dealerPlay` is called (line 626).

Method dealerPlay

Method `dealerPlay` (lines 86–139) displays the dealer's cards, then deals cards to the dealer until the dealer's hand has a value of 17 or more (lines 100–108). If the dealer exceeds 21, the player wins (line 116); otherwise, the values of the hands are compared, and `gameOver` is called with the appropriate argument (lines 122–133).

Method standJButtonActionPerformed

Clicking the **Stand JButton** indicates that a player does not want to be dealt another card. Method `standJButtonActionPerformed` (lines 631–638) disables the **Hit** and **Stand** buttons, enables the **Deal** button, then calls method `dealerPlay`.

Method displayCard

Method `displayCard` (lines 142–192) updates the GUI to display a newly dealt card. The method takes as arguments an integer index for the `JLabel` in the `ArrayList` that must have its image set and a `String` representing the card. An empty `String` indicates that we wish to display the card face down. If method `displayCard` receives a `String` that's not empty, the program extracts the face and suit from the `String` and uses this information to display the correct image. The `switch` statement (lines 167–181) converts the number representing the suit to an integer and assigns the appropriate character to variable `suitLetter` (`h` for hearts, `d` for diamonds, `c` for clubs and `s` for spades). The character in `suitLetter` is used to complete the image's filename (lines 184–186). *You must add the folder blackjack_images to your project so that lines 152–154 and 184–186 can access the images properly.* To do so, copy the folder `blackjack_images` from this chapter's examples folder and paste it into the project's `src\com\deitel\java\blackjackclient` folder.

32.42 Chapter 32 REST Web Services

32.10 Consuming a Database-Driven SOAP Web Service

Our prior examples accessed web services from desktop applications created in NetBeans. However, we can just as easily use them in web applications created with NetBeans. In fact, because web-based businesses are becoming increasingly popular, it's common for web applications to consume web services. In this section, we present an airline reservation web service that receives information regarding the type of seat a customer wishes to reserve and makes a reservation if such a seat is available. Later in the section, we present a web application that allows a customer to specify a reservation request, then uses the airline reservation web service to attempt to execute the request.

32.10.1 Creating the Reservation Database

Our web service uses a reservation database containing a single table named Seats to locate a seat matching a client's request. Review the steps presented in Section 31.2.1 for configuring a data source and the addressbook database. Then perform those steps for the reservation database used in this example. This chapter's examples directory contains the Seats.sql SQL script to create the seats table and populate it with sample data. The sample data is shown in Fig. 32.19.

number	location	class	taken
1	Aisle	Economy	0
2	Aisle	Economy	0
3	Aisle	First	0
4	Middle	Economy	0
5	Middle	Economy	0
6	Middle	First	0
7	Window	Economy	0
8	Window	Economy	0
9	Window	First	0
10	Window	First	0

Fig. 32.19 | Data from the seats table.

Creating the Reservation Web Service

You can now create a web service that uses the reservation database (Fig. 32.20). We used the @DataSourceDefinition annotation (lines 17–23) to create a data source named

```
java:global/jdbc/reservation
```

for accessing the database.

```

1 // Fig. 32.20: Reservation.java
2 // Airline reservation web service.
3 package com.deitel.reservation;
4
```

Fig. 32.20 | Airline reservation web service. (Part 1 of 3.)

32.10 Consuming a Database-Driven SOAP Web Service

32_43

```

5  import java.sql.Connection;
6  import java.sql.PreparedStatement;
7  import java.sql.ResultSet;
8  import java.sql.SQLException;
9  import javax.annotation.Resource;
10 import javax.annotation.sql.DataSourceDefinition;
11 import javax.jws.WebMethod;
12 import javax.jws.WebParam;
13 import javax.jws.WebService;
14 import javax.sql.DataSource;
15
16 // define the data source
17 @DataSourceDefinition(
18     name = "java:global/jdbc/reservation",
19     className = "org.apache.derby.jdbc.ClientDataSource",
20     url = "jdbc:derby://localhost:1527/reservation",
21     databaseName = "reservation",
22     user = "APP",
23     password = "APP")
24
25 @WebService()
26 public class Reservation
27 {
28     // allow the server to inject the DataSource
29     @Resource(literal="java:global/jdbc/reservation")
30     DataSource dataSource;
31
32     // a WebMethod that can reserve a seat
33     @WebMethod(operationName = "reserve")
34     public boolean reserve(@WebParam(name = "seatType") String seatType,
35                           @WebParam(name = "classType") String classType)
36     {
37         Connection connection = null;
38         PreparedStatement lookupSeat = null;
39         PreparedStatement reserveSeat = null;
40
41         try
42         {
43             connection = DriverManager.getConnection(
44                 DATABASE_URL, USERNAME, PASSWORD);
45             lookupSeat = connection.prepareStatement(
46                 "SELECT \"number\" FROM \"seats\" WHERE (\"taken\" = 0) " +
47                 "AND (\"location\" = ?) AND (\"class\" = ?)");
48             lookupSeat.setString(1, seatType);
49             lookupSeat.setString(2, classType);
50             ResultSet resultSet = lookupSeat.executeQuery();
51
52             // if requested seat is available, reserve it
53             if (resultSet.next())
54             {
55                 int seat = resultSet.getInt(1);
56                 reserveSeat = connection.prepareStatement(
57                     "UPDATE \"seats\" SET \"taken\"=1 WHERE \"number\"=?");

```

Fig. 32.20 | Airline reservation web service. (Part 2 of 3.)

32-44 Chapter 32 REST Web Services

```

58         reserveSeat.setInt(1, seat);
59         reserveSeat.executeUpdate();
60         return true;
61     }
62
63     return false;
64 }
65 catch (SQLException e)
66 {
67     e.printStackTrace();
68     return false;
69 }
70 catch (Exception e)
71 {
72     e.printStackTrace();
73     return false;
74 }
75 finally
76 {
77     try
78     {
79         lookupSeat.close();
80         reserveSeat.close();
81         connection.close();
82     }
83     catch (Exception e)
84     {
85         e.printStackTrace();
86         return false;
87     }
88 }
89 }
90 }
```

Fig. 32.20 | Airline reservation web service. (Part 3 of 3.)

The airline reservation web service has a single web method—`reserve` (lines 33–89)—which searches the `Seats` table to locate a seat matching a user's request. The method takes two arguments—a `String` representing the desired seat type (i.e., "Window", "Middle" or "Aisle") and a `String` representing the desired class type (i.e., "Economy" or "First"). If it finds an appropriate seat, method `reserve` updates the database to make the reservation and returns `true`; otherwise, no reservation is made, and the method returns `false`. The statements at lines 45–50 and lines 56–59 that query and update the database use objects of JDBC types `ResultSet` and `PreparedStatement`.



Software Engineering Observation 32.1

Using `PreparedStatement` to create SQL statements is highly recommended to secure against so-called SQL injection attacks in which executable code is inserted into SQL code. The site www.owasp.org/index.php/Preventing_SQL_Injection_in_Java provides a summary of SQL injection attacks and ways to mitigate against them.

32.10 Consuming a Database-Driven SOAP Web Service 32_45

Our database contains four columns—the seat number (i.e., 1–10), the seat type (i.e., Window, Middle or Aisle), the class type (i.e., Economy or First) and a column containing either 1 (true) or 0 (false) to indicate whether the seat is taken. Lines 45–50 retrieve the seat numbers of any available seats matching the requested seat and class type. This statement fills the `resultSet` with the results of the query

```
SELECT number
  FROM seats
 WHERE (taken = 0) AND (type = type) AND (class = class)
```

The parameters `type` and `class` in the query are replaced with values of method `reserve`'s `seatType` and `classType` parameters.

If `resultSet` is not empty (i.e., at least one seat is available that matches the selected criteria), the condition in line 53 is `true` and the web service reserves the first matching seat number. Recall that `ResultSet` method `next` returns `true` if a nonempty row exists, and positions the cursor on that row. We obtain the seat number (line 55) by accessing `resultSet`'s first column (i.e., `resultSet.getInt(1)`—the first column in the row). Then lines 56–59 configure a `PreparedStatement` and execute the SQL:

```
UPDATE seats
SET taken = 1
WHERE (number = number)
```

which marks the seat as taken in the database. The parameter `number` is replaced with the value of `seat`. Method `reserve` returns `true` (line 60) to indicate that the reservation was successful. If there are no matching seats, or if an exception occurred, method `reserve` returns `false` (lines 63, 68, 73 and 86) to indicate that no seats matched the user's request.

32.10.2 Creating a Web Application to Interact with the Reservation Service

This section presents a `ReservationClient` JSF web application that consumes the `Reservation` web service. The application allows users to select "Aisle", "Middle" or "Window" seats in "Economy" or "First" class, then submit their requests to the web service. If the database request is not successful, the application instructs the user to modify the request and try again. The application presented here was built using the techniques presented in Chapters 30–31. We assume that you've already read those chapters and thus know how to build a Facelets page and a corresponding JavaBean.

index.xhtml

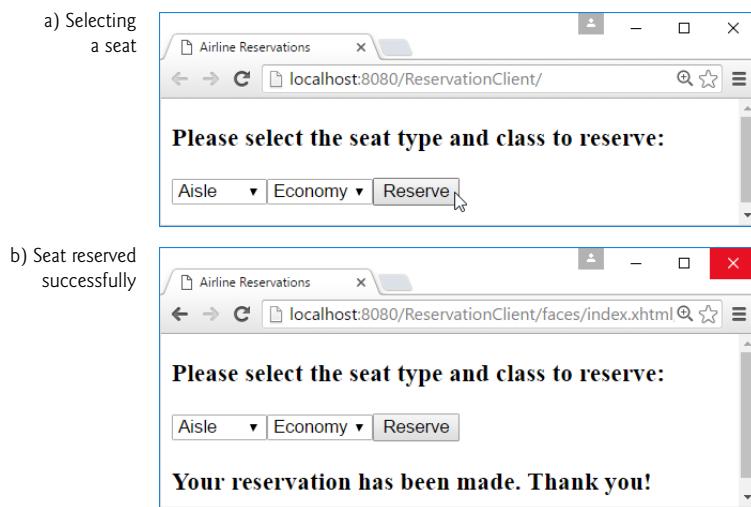
`index.xhtml` (Fig. 32.21) defines two `h:selectOneMenu` and an `h:commandButton`. The `h:selectOneMenu` at lines 16–20) displays all the seat types from which users can select. The one at lines 21–24) provides choices for the class type. The values of these are stored in the `seatType` and `classType` properties of the `reservationBean` (Fig. 32.22). Users click the **Reserve** button (lines 25–26) to submit requests after making selections from the `h:selectOneMenu`s. Clicking the button calls the `reservationBean`'s `reserveSeat` method. The page displays the result of each attempt to reserve a seat in line 28.

32_46 Chapter 32 REST Web Services

```

1  <?xml version='1.0' encoding='UTF-8' ?>
2
3  <!-- Fig. 31.21: index.xhtml -->
4  <!-- Facelets page that allows a user to select a seat -->
5  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
6   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
7  <html xmlns="http://www.w3.org/1999/xhtml"
8   xmlns:h="http://java.sun.com/jsf/html"
9   xmlns:f="http://java.sun.com/jsf/core">
10 <h:head>
11   <title>Airline Reservations</title>
12 </h:head>
13 <h:body>
14 <h:form>
15   <h3>Please select the seat type and class to reserve:</h3>
16   <h:selectOneMenu value="#{reservationBean.seatType}">
17     <f:selectItem itemValue="Aisle" itemLabel="Aisle" />
18     <f:selectItem itemValue="Middle" itemLabel="Middle" />
19     <f:selectItem itemValue="Window" itemLabel="Window" />
20   </h:selectOneMenu>
21   <h:selectOneMenu value="#{reservationBean.classType}">
22     <f:selectItem itemValue="Economy" itemLabel="Economy" />
23     <f:selectItem itemValue="First" itemLabel="First" />
24   </h:selectOneMenu>
25   <h:commandButton value="Reserve"
26     action="#{reservationBean.reserveSeat}" />
27 </h:form>
28   <h3>#{reservationBean.result}</h3>
29 </h:body>
30 </html>

```

**Fig. 32.21** | Facelets page that allows a user to select a seat. (Part I of 2.)

c) Attempting to reserve an aisle economy seat when no more are available—because no seats match the requested seat type and class, the user is asked to try again

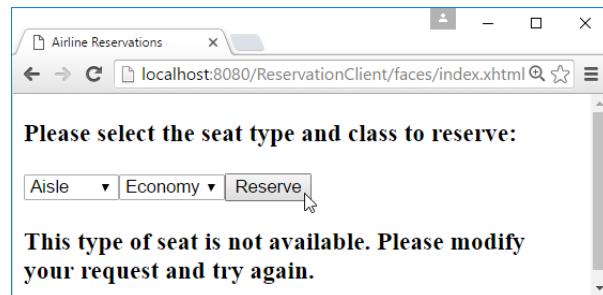


Fig. 32.21 | Facelets page that allows a user to select a seat. (Part 2 of 2.)

ReservationBean.java

Class **ReservationBean** (Fig. 32.22) defines the **seatType**, **classType** and **result** properties and the **reserveSeat** method that are used in the **index.xhtml** page. When the user clicks the **Reserve** button in **index.xhtml**, method **reserveSeat** (lines 59–76) executes. Lines 63–64 use the service endpoint interface object (created in lines 24–25) to invoke the web service’s **reserve** method, passing the selected seat type and class type as arguments. If **reserve** returns **true**, line 67 sets **result** to a message thanking the user for making a reservation; otherwise, lines 69–70 set **result** to a message notifying the user that the requested seat type is not available and instructing the user to try again.

```

1 // Fig. 31.22: ReservationBean.java
2 // Bean for seat reservation client.
3 package reservationclient;
4
5 import com.deitel.reservation.Reservation;
6 import com.deitel.reservation.ReservationService;
7 import java.io.Serializable;
8 import javax.faces.bean.ManagedBean;
9
10 @Named("reservationBean")
11 @javax.faces.view.ViewScoped
12 public class ReservationBean implements Serializable
13 {
14     // references the service endpoint interface object (i.e., the proxy)
15     private Reservation reservationServiceProxy; // reference to proxy
16     private String seatType; // type of seat to reserve
17     private String classType; // class of seat to reserve
18     private String result; // result of reservation attempt
19
20     // no-argument constructor
21     public ReservationBean()
22     {
23         // get service endpoint interface
24         ReservationService reservationService = new ReservationService();
25         reservationServiceProxy = reservationService.getReservationPort();
26     }
}

```

Fig. 32.22 | Page bean for seat reservation client. (Part 1 of 2.)

32_48 Chapter 32 REST Web Services

```

27      // return classType
28      public String getClassType()
29      {
30          return classType;
31      }
32
33      // set classType
34      public void setClassType(String classType)
35      {
36          this.classType = classType;
37      }
38
39      // return seatType
40      public String getSeatType()
41      {
42          return seatType;
43      }
44
45      // set seatType
46      public void setSeatType(String seatType)
47      {
48          this.seatType = seatType;
49      }
50
51      // return result
52      public String getResult()
53      {
54          return result;
55      }
56
57
58      // invoke the web service when the user clicks Reserve button
59      public void reserveSeat()
60      {
61          try
62          {
63              boolean reserved = reservationServiceProxy.reserve(
64                  getSeatType(), getClassType());
65
66              if (reserved)
67                  result = "Your reservation has been made. Thank you!";
68              else
69                  result = "This type of seat is not available. " +
70                  "Please modify your request and try again.";
71          }
72          catch (Exception e)
73          {
74              e.printStackTrace();
75          }
76      }
77  }

```

Fig. 32.22 | Page bean for seat reservation client. (Part 2 of 2.)

32.11 Equation Generator: Returning User-Defined Types

32_49

32.11 Equation Generator: Returning User-Defined Types

Most of the web services we've demonstrated received and returned primitive-type instances. It's also possible to process instances of class types in a web service. These types can be passed to or returned from web service methods.

This section presents a RESTful `EquationGenerator` web service that generates random arithmetic equations of type `Equation`. The client is a math-tutoring application that accepts information about the mathematical question that the user wishes to attempt (addition, subtraction or multiplication) and the skill level of the user (1 specifies equations using numbers from 1 through 9, 2 specifies equations involving numbers from 10 through 99, and 3 specifies equations containing numbers from 100 through 999). Each web service then generates an equation consisting of random numbers in the proper range. The client application receives the `Equation` and displays the sample question to the user. We present the web service and client twice—once using XML and once using JSON.

Defining Class Equation

We define class `Equation` in Fig. 32.23. All the programs in this section have a copy of this class in their corresponding package. Except for the package name, the class is identical in each project, so we show it only once. Like the `TextMessage` class used earlier, the server-side and client-side copies of class `Equation` are unrelated to each other. The only requirement for *serialization* and *deserialization* to work with the JAXB and Gson classes is that class `Equation` must have the same `public` properties on both the server and the client. Such properties can be `public` instance variables or `private` instance variables that have corresponding, appropriately named `set` and `get` methods.

```

1 // Fig. 32.23: Equation.java
2 // Equation class that contains information about an equation.
3 package com.deitel.equationgeneratorxml;
4
5 public class Equation
6 {
7     private int leftOperand;
8     private int rightOperand;
9     private int result;
10    private String operationType;
11
12    // required no-argument constructor
13    public Equation()
14    {
15        this(0, 0, "add");
16    }
17
18    // constructor that receives the operands and operation type
19    public Equation(int leftValue, int rightValue, String type)
20    {
21        leftOperand = leftValue;
22        rightOperand = rightValue;
23    }

```

Fig. 32.23 | Equation class that contains information about an equation. (Part 1 of 3.)

32_50 Chapter 32 REST Web Services

```

24      // determine result
25      if (type.equals("add")) // addition
26      {
27          result = leftOperand + rightOperand;
28          operationType = "+";
29      }
30      else if (type.equals("subtract")) // subtraction
31      {
32          result = leftOperand - rightOperand;
33          operationType = "-";
34      }
35      else // multiplication
36      {
37          result = leftOperand * rightOperand;
38          operationType = "*";
39      }
40  }
41
42  // gets the leftOperand
43  public int getLeftOperand()
44  {
45      return leftOperand;
46  }
47
48  // required setter
49  public void setLeftOperand(int value)
50  {
51      leftOperand = value;
52  }
53
54  // gets the rightOperand
55  public int getRightOperand()
56  {
57      return rightOperand;
58  }
59
60  // required setter
61  public void setRightOperand(int value)
62  {
63      rightOperand = value;
64  }
65
66  // gets the resultValue
67  public int getResult()
68  {
69      return result;
70  }
71
72  // required setter
73  public void setResult(int value)
74  {
75      result = value;
76  }

```

Fig. 32.23 | Equation class that contains information about an equation. (Part 2 of 3.)

```

77
78     // gets the operationType
79     public String getOperationType()
80     {
81         return operationType;
82     }
83
84     // required setter
85     public void setOperationType(String value)
86     {
87         operationType = value;
88     }
89
90     // returns the left hand side of the equation as a String
91     public String getLeftHandSide()
92     {
93         return leftOperand + " " + operationType + " " + rightOperand;
94     }
95
96     // returns the right hand side of the equation as a String
97     public String getRightHandSide()
98     {
99         return "" + result;
100    }
101
102    // returns a String representation of an Equation
103    public String toString()
104    {
105        return getLeftHandSide() + " = " + getRightHandSide();
106    }
107 }

```

Fig. 32.23 | Equation class that contains information about an equation. (Part 3 of 3.)

Lines 19–40 define a constructor that takes two `ints` representing the left and right operands, and a `String` representing the arithmetic operation. The constructor stores this information, then calculates the result. The parameterless constructor (lines 13–16) calls the three-argument constructor (lines 19–40) and passes default values.

Class `Equation` defines `get` and `set` methods for example variables `leftOperand` (lines 43–52), `rightOperand` (lines 55–64), `result` (line 67–76) and `operationType` (lines 79–88). It also provides `get` methods for the left-hand and right-hand sides of the equation and a `toString` method that returns the entire equation as a `String`. An instance variable can be serialized only if it has both a `get` and a `set` method. Because the different sides of the equation and the result of `toString` can be generated from the other instance variables, there's no need to send them across the wire. The client in this case study does not use the `getRightHandSide` method, but we included it in case future clients choose to use it.

32.11.1 Creating the EquationGeneratorXML Web Service

Figure 32.24 presents the `EquationGeneratorXML` web service's class for creating randomly generated Equations. Method `getXml` (lines 19–38) takes two parameters—a `String` representing the mathematical operation ("add", "subtract" or "multiply") and an `int`

32.52 Chapter 32 REST Web Services

representing the difficulty level. JAX-RS automatically converts the arguments to the correct type and will return a “not found” error to the client if the argument cannot be converted from a `String` to the destination type. Supported types for conversion include integer types, floating-point types, `boolean` and the corresponding type-wrapper classes.

```

1 // Fig. 32.24: EquationGeneratorXMLResource.java
2 // RESTful equation generator that returns XML.
3 package com.deitel.equationgeneratorxml;
4
5 import java.io.StringWriter;
6 import java.security.SecureRandom;
7 import javax.ws.rs.PathParam;
8 import javax.ws.rs.Path;
9 import javax.ws.rs.GET;
10 import javax.ws.rs.Produces;
11 import javax.xml.bind.JAXB; // utility class for common JAXB operations
12
13 @Path("equation")
14 public class EquationGeneratorXMLResource
15 {
16     private static SecureRandom randomObject = new SecureRandom();
17
18     // retrieve an equation formatted as XML
19     @GET
20     @Path("{operation}/{level}")
21     @Produces("application/xml")
22     public String getXml(@PathParam("operation") String operation,
23                          @PathParam("Level") int level)
24     {
25         // compute minimum and maximum values for the numbers
26         int minimum = (int) Math.pow(10, level - 1);
27         int maximum = (int) Math.pow(10, level);
28
29         // create the numbers on the left-hand side of the equation
30         int first = randomObject.nextInt(maximum - minimum) + minimum;
31         int second = randomObject.nextInt(maximum - minimum) + minimum;
32
33         // create Equation object and marshal it into XML
34         Equation equation = new Equation(first, second, operation);
35         StringWriter writer = new StringWriter(); // XML output here
36         JAXB.marshal(equation, writer); // write Equation to StringWriter
37         return writer.toString(); // return XML string
38     }
39 }
```

Fig. 32.24 | RESTful equation generator that returns XML.

The `getXml` method first determines the minimum (inclusive) and maximum (exclusive) values for the numbers in the equation it will return (lines 26–27). It then uses a `Random` object (created at line 16) to generate two random numbers in that range (lines 30–31). Line 34 creates an `Equation` object, passing these two numbers and the requested operation to the constructor. The `getXml` method then uses JAXB to convert the `Equation` object to XML (line 36), which is output to the `StringWriter` created on line 35. Finally,

it retrieves the data that was written to the `StringWriter` and returns it to the client. For example, if you invoke the web service with

```
http://localhost:8080/EquationGeneratorXML/webresources/equation/
add/1
```

the response will have the format

```
<equation>
<leftOperand>6</leftOperand>
<operationType>+</operationType>
<result>11</result>
<rightOperand>5</rightOperand>
</equation>
```

[Note: We'll reimplement this web service with JSON in Section 32.11.3.]

32.11.2 Consuming the EquationGeneratorXML Web Service

The `EquationGeneratorXMLClient` application (Fig. 32.25) retrieves an XML-formatted `Equation` object from the `EquationGeneratorXML` web service. The application then displays the `Equation`'s left-hand side and waits for user to submit an answer.

```

1 // Fig. 32.25: EquationGeneratorXMLClientJFrame.java
2 // Math-tutoring program using REST and XML to generate equations.
3 package com.deitel.equationgeneratorxmlclient;
4
5 import javax.swing.JOptionPane;
6 import javax.xml.bind.JAXB; // utility class for common JAXB operations
7
8 public class EquationGeneratorXMLClientJFrame extends javax.swing.JFrame
9 {
10    private String operation = "add"; // operation user is tested on
11    private int difficulty = 1; // 1, 2, or 3 digits in each number
12    private int answer; // correct answer to the question
13
14    // no-argument constructor
15    public EquationGeneratorXMLClientJFrame()
16    {
17        initComponents();
18    }
19
20    // The initComponents method is autogenerated by NetBeans and is called
21    // from the constructor to initialize the GUI. This method is not shown
22    // here to save space. Open EquationGeneratorXMLClientJFrame.java in
23    // this example's folder to view the complete generated code.
24
25    // determine if the user answered correctly
26    private void checkAnswerJButtonActionPerformed(
27        java.awt.event.ActionEvent evt)
28    {
29        if (answerJTextField.getText().equals(""))
30        {
```

// The initComponents method is autogenerated by NetBeans and is called
// from the constructor to initialize the GUI. This method is not shown
// here to save space. Open EquationGeneratorXMLClientJFrame.java in
// this example's folder to view the complete generated code.

Fig. 32.25 | Math-tutoring program using REST and XML to generate equations. (Part I of 3.)

32_54 Chapter 32 REST Web Services

```

31         JOptionPane.showMessageDialog(
32             this, "Please enter your answer.");
33     }
34
35     int userAnswer = Integer.parseInt(answerJTextField.getText());
36
37     if (userAnswer == answer)
38     {
39         equationJLabel.setText(""); // clear label
40         answerJTextField.setText(""); // clear text field
41         checkAnswerJButton.setEnabled(false);
42         JOptionPane.showMessageDialog(this, "Correct! Good Job!",
43             "Correct", JOptionPane.PLAIN_MESSAGE);
44     }
45     else
46     {
47         JOptionPane.showMessageDialog(this, "Incorrect. Try again.",
48             "Incorrect", JOptionPane.PLAIN_MESSAGE);
49     }
50 }
51
52 // retrieve equation from web service and display left side to user
53 private void generateJButtonActionPerformed(
54     java.awt.event.ActionEvent evt)
55 {
56     try
57     {
58         String url = String.format("http://localhost:8080/" +
59             "EquationGeneratorXML/webresources/equation/%s/%d",
60             operation, difficulty);
61
62         // convert XML back to an Equation object
63         Equation equation = JAXB.unmarshal(url, Equation.class);
64
65         answer = equation.getResult();
66         equationJLabel.setText(equation.getLeftHandSide() + " =");
67         checkAnswerJButton.setEnabled(true);
68     }
69     catch (Exception exception)
70     {
71         exception.printStackTrace();
72     }
73 }
74
75 // obtains the mathematical operation selected by the user
76 private void operationJComboBoxItemStateChanged(
77     java.awt.event.ItemEvent evt)
78 {
79     String item = (String) operationJComboBox.getSelectedItem();
80
81     if (item.equals("Addition"))
82         operation = "add"; // user selected addition

```

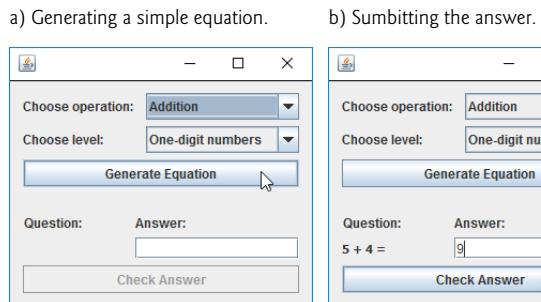
Fig. 32.25 | Math-tutoring program using REST and XML to generate equations. (Part 2 of 3.)

```

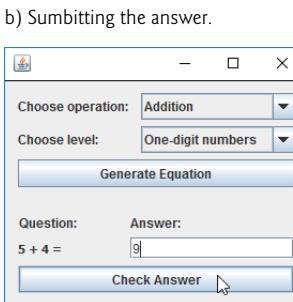
83         else if (item.equals("Subtraction"))
84             operation = "subtract"; // user selected subtraction
85         else
86             operation = "multiply"; // user selected multiplication
87     }
88
89     // obtains the difficulty level selected by the user
90     private void levelJComboBoxItemStateChanged(
91         java.awt.event.ItemEvent evt)
92     {
93         // indices start at 0, so add 1 to get the difficulty level
94         difficulty = levelJComboBox.getSelectedIndex() + 1;
95     }
96
97     // main method begins execution
98     public static void main(String args[])
99     {
100         java.awt.EventQueue.invokeLater(
101             new Runnable()
102             {
103                 public void run()
104                 {
105                     new EquationGeneratorXMLClientJFrame().setVisible(true);
106                 }
107             }
108         );
109     }
110
111     // Variables declaration - do not modify
112     private javax.swing.JLabel answerJLabel;
113     private javax.swing.JTextField answerJTextField;
114     private javax.swing.JButton checkAnswerJButton;
115     private javax.swing.JLabel equationJLabel;
116     private javax.swing.JButton generateJButton;
117     private javax.swing.JComboBox levelJComboBox;
118     private javax.swing.JLabel levelJLabel;
119     private javax.swing.JComboBox operationJComboBox;
120     private javax.swing.JLabel operationJLabel;
121     private javax.swing.JLabel questionJLabel;
122     // End of variables declaration
123 }

```

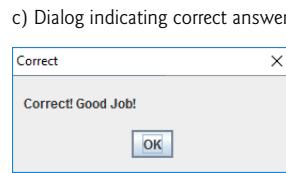
a) Generating a simple equation.



b) Submitting the answer.



c) Dialog indicating correct answer.

**Fig. 32.25** | Math-tutoring program using REST and XML to generate equations. (Part 3 of 3.)

32.56 Chapter 32 REST Web Services

The default setting for the difficulty level is 1, but the user can change this by choosing a level from the **Choose level** JComboBox. Changing the selected value invokes the `levelJComboBoxItemStateChanged` event handler (lines 212–217), which sets the `difficulty` instance variable to the level selected by the user. Although the default setting for the question type is **Addition**, the user also can change this by choosing from the **Choose operation** JComboBox. This invokes the `operationJComboBoxItemStateChanged` event handler in lines 198–209, which assigns to instance variable `operation` the String corresponding to the user's selection.

The event handler for `generateJButton` (lines 175–195) constructs the URL to invoke the web service, then passes this URL to the `unmarshal` method, along with an instance of `Class<Equation>`, so that JAXB can convert the XML into an `Equation` object (line 185). Once the XML has been converted back into an `Equation`, lines 183–184 retrieve the correct answer and display the left-hand side of the equation. The **Check Answer** button is then enabled (line 189), and the user must solve the problem and enter the answer.

When the user enters a value and clicks **Check Answer**, the `checkAnswerJButtonActionPerformed` event handler (lines 148–172) retrieves the user's answer from the dialog box (line 157) and compares it to the correct answer that was stored earlier (line 159). If they match, lines 161–165 reset the GUI elements so the user can generate another equation and tell the user that the answer was correct. If they do not match, a message box asking the user to try again is displayed (lines 169–170).

32.11.3 Creating the EquationGeneratorJSON Web Service

As you saw in Section 32.8, RESTful web services can return data formatted as JSON as well. Figure 32.26 is a reimplementation of the `EquationGeneratorXML` service that returns an `Equation` in JSON format. The logic implemented here is the same as the XML version except for the last line (line 34), which uses `Gson` to convert the `Equation` object into JSON instead of using JAXB to convert it into XML. The `@Produces` annotation (line 20) has also changed to reflect the JSON data format.

```

1 // Fig. 32.26: EquationGeneratorJSONResource.java
2 // RESTful equation generator that returns JSON.
3 package com.deitel.equationgeneratorjson;
4
5 import com.google.gson.Gson; // converts POJO to JSON and back again
6 import java.util.Random;
7 import javax.ws.rs.GET;
8 import javax.ws.rs.Path;
9 import javax.ws.rs.PathParam;
10 import javax.ws.rs.Produces;
11
12 @Path("equation")
13 public class EquationGeneratorJSONResource
14 {
15     static Random randomObject = new Random(); // random number generator
16 }
```

Fig. 32.26 | RESTful equation generator that returns JSON. (Part I of 2.)

```

17 // retrieve an equation formatted as JSON
18 @GET
19 @Path("{operation}/{level}")
20 @Produces("application/json")
21 public String getJson(@PathParam("operation") String operation,
22                      @PathParam("level") int level)
23 {
24     // compute minimum and maximum values for the numbers
25     int minimum = (int) Math.pow(10, level - 1);
26     int maximum = (int) Math.pow(10, level);
27
28     // create the numbers on the left-hand side of the equation
29     int first = randomObject.nextInt(maximum - minimum) + minimum;
30     int second = randomObject.nextInt(maximum - minimum) + minimum;
31
32     // create Equation object and return result
33     Equation equation = new Equation(first, second, operation);
34     return new Gson().toJson(equation); // convert to JSON and return
35 }
36 }
```

Fig. 32.26 | RESTful equation generator that returns JSON. (Part 2 of 2.)

32.11.4 Consuming the EquationGeneratorJSON Web Service

The program in Fig. 32.27 consumes the `EquationGeneratorJSON` service and performs the same function as `EquationGeneratorXMLClient`—the only difference is in how the `Equation` object is retrieved from the web service. Lines 181–183 construct the URL that is used to invoke the `EquationGeneratorJSON` service. As in the `WelcomeRESTJSONClient` example, we use the `URL` class and an `InputStreamReader` to invoke the web service and read the response (lines 186–187). The retrieved JSON is *deserialized* using `Gson` (line 191) and converted back into an `Equation` object. As before, we use the `getResult` method (line 194) of the deserialized object to obtain the answer and the `getLeftHandSide` method (line 195) to display the left side of the equation.

```

1 // Fig. 32.27: EquationGeneratorJSONClientJFrame.java
2 // Math-tutoring program using REST and JSON to generate equations.
3 package com.deitel.equationgeneratorjsonclient;
4
5 import com.google.gson.Gson; // converts POJO to JSON and back again
6 import java.io.InputStreamReader;
7 import java.net.URL;
8 import javax.swing.JOptionPane;
9
10 public class EquationGeneratorJSONClientJFrame extends javax.swing.JFrame
11 {
12     private String operation = "add"; // operation user is tested on
13     private int difficulty = 1; // 1, 2, or 3 digits in each number
14     private int answer; // correct answer to the question
15 }
```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 1 of 4.)

32_58 Chapter 32 REST Web Services

```

16    // no-argument constructor
17    public EquationGeneratorJSONClientJFrame()
18    {
19        initComponents();
20    }
21
22    // The initComponents method is autogenerated by NetBeans and is called
23    // from the constructor to initialize the GUI. This method is not shown
24    // here to save space. Open EquationGeneratorJSONClientJFrame.java in
25    // this example's folder to view the complete generated code.
26
27    // determine if the user answered correctly
28    private void checkAnswerJButtonActionPerformed(
29        java.awt.event.ActionEvent evt)
30    {
31        if (answerJTextField.getText().equals(""))
32        {
33            JOptionPane.showMessageDialog(
34                this, "Please enter your answer.");
35        }
36
37        int userAnswer = Integer.parseInt(answerJTextField.getText());
38
39        if (userAnswer == answer)
40        {
41            equationJLabel.setText(""); // clear label
42            answerJTextField.setText(""); // clear text field
43            checkAnswerJButton.setEnabled(false);
44            JOptionPane.showMessageDialog(this, "Correct! Good Job!",
45                "Correct", JOptionPane.PLAIN_MESSAGE);
46        }
47        else
48        {
49            JOptionPane.showMessageDialog(this, "Incorrect. Try again.",
50                "Incorrect", JOptionPane.PLAIN_MESSAGE);
51        }
52    }
53
54    // retrieve equation from web service and display left side to user
55    private void generateJButtonActionPerformed(
56        java.awt.event.ActionEvent evt)
57    {
58        try
59        {
60            // URL of the EquationGeneratorJSON service, with parameters
61            String url = String.format("http://localhost:8080/" +
62                "EquationGeneratorJSON/webresources/equation/%s/%d",
63                operation, difficulty);
64
65            // open URL and create a Reader to read the data
66            InputStreamReader reader =
67                new InputStreamReader(new URL(url).openStream());
68
69        }
70    }
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88

```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 2 of 4.)

```

189         // convert the JSON back into an Equation object
190         Equation equation =
191             new Gson().fromJson(reader, Equation.class);
192
193         // update the internal state and GUI to reflect the equation
194         answer = equation.getResult();
195         equationJLabel.setText(equation.getLeftHandSide() + " =");
196         checkAnswerJButton.setEnabled(true);
197     }
198     catch (Exception exception)
199     {
200         exception.printStackTrace();
201     }
202 }
203
204 // obtains the mathematical operation selected by the user
205 private void operationJComboBoxItemStateChanged(
206     java.awt.event.ItemEvent evt)
207 {
208     String item = (String) operationJComboBox.getSelectedItem();
209
210     if (item.equals("Addition"))
211         operation = "add"; // user selected addition
212     else if (item.equals("Subtraction"))
213         operation = "subtract"; // user selected subtraction
214     else
215         operation = "multiply"; // user selected multiplication
216 }
217
218 // obtains the difficulty level selected by the user
219 private void levelJComboBoxItemStateChanged(
220     java.awt.event.ItemEvent evt)
221 {
222     // indices start at 0, so add 1 to get the difficulty level
223     difficulty = levelJComboBox.getSelectedIndex() + 1;
224 }
225
226 // main method begins execution
227 public static void main(String args[])
228 {
229     java.awt.EventQueue.invokeLater(
230         new Runnable()
231         {
232             public void run()
233             {
234                 new EquationGeneratorJSONClientJFrame().setVisible(true);
235             }
236         }
237     );
238 }
239
240 // Variables declaration - do not modify
241 private javax.swing.JLabel answerJLabel;

```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 3 of 4.)

32_60 Chapter 32 REST Web Services

```

242     private javax.swing.JTextField answerJTextField;
243     private javax.swing.JButton checkAnswerJButton;
244     private javax.swing.JLabel equationJLabel;
245     private javax.swing.JButton generateJButton;
246     private javax.swing.JComboBox levelJComboBox;
247     private javax.swing.JLabel levelJLabel;
248     private javax.swing.JComboBox operationJComboBox;
249     private javax.swing.JLabel operationJLabel;
250     private javax.swing.JLabel questionJLabel;
251     // End of variables declaration
252 }
```

Fig. 32.27 | Math-tutoring program using REST and JSON to generate equations. (Part 4 of 4.)

32.12 Wrap-Up

This chapter introduced web services—a set of technologies for building distributed systems in which system components communicate with one another over networks. In particular, we presented JAX-WS SOAP-based web services and JAX-RS REST-based web services. You learned that a web service is a class that allows client software to call the web service’s methods remotely via common data formats and protocols, such as XML, JSON, HTTP, SOAP and REST. We also benefits of distributed computing with web services.

We explained how NetBeans and the JAX-WS and JAX-RS APIs facilitate publishing and consuming web services. You learned how to define web services and methods using both SOAP protocol and REST architecture, and how to return data in both XML and JSON formats. You consumed SOAP-based web services using proxy classes to call the web service’s methods. You also consumed REST-based web services by using class URL to invoke the services and open `InputStreams` from which the clients could read the services’ responses. You learned how to define web services and web methods, as well as how to consume them both from Java desktop applications and from web applications. After explaining the mechanics of web services through our `Welcome` examples, we demonstrated more sophisticated web services that use session tracking, database access and user-defined types. We also explained XML and JSON serialization and showed how to retrieve objects of user-defined types from web services.

Summary

Section 32.1 Introduction

- A web service (p. 2) is a software component stored on one computer that can be accessed by an application (or other software component) on another computer over a network.
- Web services communicate using such technologies as XML, JSON and HTTP.
- JAX-WS (p. 2) is based on the Simple Object Access Protocol (SOAP; p. 2)—an XML-based protocol that allows web services and clients to communicate.
- JAX-RS (p. 2) uses Representational State Transfer (REST; p. 2)—a network architecture that uses the web’s traditional request/response mechanisms such as `GET` and `POST` requests.
- Web services enable businesses to conduct transactions via standardized, widely available web services rather than relying on proprietary applications.

- Web services are platform and language independent, so companies can collaborate via web services without hardware, software and communications compatibility issues.
- NetBeans is one of the many tools that enable you to publish and/or consume web services.

Section 32.2 Web Service Basics

- The machine on which a web service resides is referred to as a web service host.
- A client application that accesses the web service sends a method call over a network to the web service host, which processes the call and returns a response over the network to the application.
- In Java, a web service is implemented as a class. The class that represents the web service resides on a server—it's not part of the client application.
- Making a web service available to receive client requests is known as publishing a web service (p. 4); using a web service from a client application is known as consuming a web service (p. 4).

Section 32.3 Simple Object Access Protocol (SOAP)

- SOAP is a platform-independent protocol that uses XML to make remote procedure calls, typically over HTTP. Each request and response is packaged in a SOAP message (p. 4)—an XML message containing the information that a web service requires to process the message.
- SOAP messages are written in XML so that they're computer readable, human readable and platform independent.
- SOAP supports an extensive set of types—the primitive types, as well as `DateTime`, `XmlNode` and others. SOAP can also transmit arrays of these types.
- When a program invokes a method of a SOAP web service, the request and all relevant information are packaged in a SOAP message, enclosed in a SOAP envelope (p. 4) and sent to the server on which the web service resides.
- When a web service receives a SOAP message, it parses the XML representing the message, then processes the message's contents. The message specifies the method that the client wishes to execute and the arguments the client passed to that method.
- After a web service parses a SOAP message, it calls the appropriate method with the specified arguments (if any) and sends the response back to the client in another SOAP message. The client parses the response to retrieve the method's result.

Section 32.4 Representational State Transfer (REST)

- Representational State Transfer (REST) refers to an architectural style for implementing web services. Such web services are often called RESTful web services (p. 4). Though REST itself is not a standard, RESTful web services are implemented using web standards.
- Each operation in a RESTful web service is identified by a unique URL.
- REST can return data in many formats, including XML and JSON.

Section 32.5 JavaScript Object Notation (JSON)

- JavaScript Object Notation (JSON; p. 5) is an alternative to XML for representing data.
- JSON is a text-based data-interchange format used to represent objects in JavaScript as collections of name/value pairs represented as `Strings`.
- JSON is a simple format that makes objects easy to read, create and parse and allows programs to transmit data efficiently across the Internet, because it's much less verbose than XML.
- Each value in a JSON array can be a string, a number, a JSON object, `true`, `false` or `null`.

32_62 Chapter 32 REST Web Services

Section 32.6.1 Creating a Web Application Project and Adding a Web Service Class in NetBeans

- When you create a web service in NetBeans, you focus on the web service’s logic and let the IDE handle the web service’s infrastructure.
- To create a web service in NetBeans, you first create a **Web Application** project (p. 5).

Section 31.6.2 Defining the *WelcomeSOAP* Web Service in NetBeans

- By default, each new web service class created with the JAX-WS APIs is a POJO (plain old Java object)—you do not need to extend a class or implement an interface to create a web service.
- When you deploy a web application containing a JAX-WS web service, the server creates the server-side artifacts that support the web service.
- The `@WebService` annotation (p. 7) indicates that a class represents a web service. The optional `name` attribute (p. 7) specifies the service endpoint interface (SEI; p. 7) class’s name. The optional `serviceName` attribute (p. 7) specifies the name of the class that the client uses to obtain an SEI object.
- Methods that are tagged with the `@WebMethod` annotation (p. 7) can be called remotely.
- The `@WebMethod` annotation’s optional `operationName` attribute (p. 7) specifies the method name that is exposed to the web service’s clients.
- Web method parameters are annotated with the `@WebParam` annotation (p. 8). The optional `name` attribute (p. 8) indicates the parameter name that is exposed to the web service’s clients.

Section 31.6.3 Publishing the *WelcomeSOAP* Web Service from NetBeans

- NetBeans handles all the details of building and deploying a web service for you. This includes creating the framework required to support the web service.

Section 31.6.4 Testing the *WelcomeSOAP* Web Service with GlassFish Application Server’s Tester Web Page

- GlassFish can dynamically create a web page for testing a web service’s methods from a web browser. To open the test page, expand the project’s **Web Services** node in the NetBeans **Projects** tab, then right click the web service class name and select **Test Web Service**.
- A client can access a web service only when the application server is running. If NetBeans launches the application server for you, the server will shut down when you close NetBeans. To keep the application server up and running, you can launch it independently of NetBeans.

Section 32.6.5 Describing a Web Service with the Web Service Description Language (WSDL)

- To consume a web service, a client must know where to find it and must be provided with the web service’s description.
- JAX-WS uses the Web Service Description Language (WSDL; p. 11)—a standard XML vocabulary for describing web services in a platform-independent manner.
- The server generates a web service’s WSDL dynamically for you, and client tools can parse the WSDL to help create the client-side proxy class that a client uses to access the web service.

Section 31.6.6 Creating a Client to Consume the *WelcomeSOAP* Web Service

- A web service reference (p. 12) defines the service endpoint interface class so that a client can access the a service.

- An application that consumes a SOAP-based web service invokes methods on a service endpoint interface (SEI) object that interact with the web service on the client's behalf.
- The service endpoint interface object handles the details of passing method arguments to and receiving return values from the web service. This communication can occur over a local network, over the Internet or even with a web service on the same computer.
- NetBeans creates these service endpoint interface classes for you.
- When you add the web service reference, the IDE creates and compiles the client-side artifacts—the framework of Java code that supports the client-side service endpoint interface class. The service endpoint interface class uses the rest of the artifacts to interact with the web service.
- A web service reference is added by giving NetBeans the URL of the web service's WSDL file.

Section 31.6.7 Consuming the WelcomeSOAP Web Service

- To consume a JAX-WS web service, you must obtain an SEI object. You then invoke the web service's methods through the SEI object.

Section 32.7.1 Creating a REST-Based XML Web Service

- The **RESTful Web Services** plug-in for NetBeans provides various templates for creating RESTful web services, including ones that can interact with databases on the client's behalf.
- The `@Path` annotation (p. 17) on a JAX-RS web service class indicates the URI for accessing the web service. This is appended to the web application project's URL to invoke the service. Methods of the class can also use the `@Path` annotation.
- Parts of the path specified in curly braces indicate parameters—they're placeholders for arguments that are passed to the web service as part of the path. The base path for the service is the project's `resources` directory.
- Arguments in a URL can be used as arguments to a web service method. To do so, you bind the parameters specified in the `@Path` specification to parameters of a web service method with the `@PathParam` annotation (p. 19). When the request is received, the server passes the argument(s) in the URL to the appropriate parameter(s) in the web service method.
- The `@GET` annotation (p. 19) denotes that a method is accessed via an HTTP GET request. Similar annotations exist for HTTP PUT, POST, DELETE and HEAD requests.
- The `@Produces` annotation (p. 19) denotes the content type returned to the client. It's possible to have multiple methods with the same HTTP method and path but different `@Produces` annotations, and JAX-RS will call the method matching the content type requested by the client.
- The `@Consumes` annotation (p. 19) restricts the content type that a web service accepts from a PUT request.
- JAXB (Java Architecture for XML Binding; p. 19) is a set of classes for converting POJOs to and from XML. Class `JAXB` (package `javax.xml.bind`) contains static methods for common operations.
- Class `JAXB`'s static method `marshal` (p. 19) converts a Java object to XML format.
- WADL (Web Application Description Language; p. 20) has similar design goals to WSDL, but describes RESTful services instead of SOAP services.

Section 32.7.2 Consuming a REST-Based XML Web Service

- Clients of RESTful web services do not require web service references.
- The `JAXB` class has a static `unmarshal` method that takes as arguments a filename or URL as a `String`, and a `Class<T>` object indicating the Java class to which the XML will be converted.

32_64 Chapter 32 REST Web Services

Section 32.8 Publishing and Consuming REST-Based JSON Web Services

- JSON components—objects, arrays, strings, numbers—can be easily mapped to constructs in Java and other programming languages.

Section 32.8.1 Creating a REST-Based JSON Web Service

- To add a JAR file as a library in NetBeans, right click your project's **Libraries** folder, select **Add JAR/Folder...**, locate the JAR file and click **Open**.
- For a web service method that returns JSON text, the argument to the `@Produces` attribute must be "application/json".
- In JSON, all data must be encapsulated in a composite data type.
- Create a `Gson` object (from package `com.google.gson`) and call its `toJson` method to convert an object into its JSON String representation.

Section 32.8.2 Consuming a REST-Based JSON Web Service

- To read JSON data from a URL, create a `URL` object and call its `openStream` method (p. 26). This invokes the web service and returns an `InputStream` from which the client can read the response. Wrap the `InputStream` in an `InputStreamReader` so it can be passed as the first argument to the `Gson` class's `fromJson` method (p. 26).

Section 32.9 Session Tracking in a SOAP Web Service

- It can be beneficial for a web service to maintain client state information, thus eliminating the need to pass client information between the client and the web service multiple times. Storing session information also enables a web service to distinguish between clients.

Section 31.9.1 Creating a Blackjack Web Service

- In JAX-WS 2.2, to enable session tracking in a web service, you simply precede your web service class with the `@HttpSessionScope` annotation (p. 29) from package `com.sun.xml.ws.developer.servlet`. To use this package you must add the JAX-WS 2.2 library to your project.
- Once a web service is annotated with `@HttpSessionScope`, the server automatically maintains a separate instance of the class for each client session.

Section 31.9.2 Consuming the Blackjack Web Service

- In the JAX-WS framework, the client must indicate whether it wants to allow the web service to maintain session information. To do this, first cast the proxy object to interface type `BindingProvider`. A `BindingProvider` enables the client to manipulate the request information that will be sent to the server. This information is stored in an object that implements interface `RequestContext`. The `BindingProvider` and `RequestContext` are part of the framework that is created by the IDE when you add a web service client to the application.
- Next, invoke the `BindingProvider`'s `getRequestContext` method to obtain the `RequestContext` object. Then call the `RequestContext`'s `put` method to set the property `BindingProvider.SESSION_MAINTAIN_PROPERTY` to `true`, which enables session tracking from the client side so that the web service knows which client is invoking the service's web methods.

Section 32.11 Equation Generator: Returning User-Defined Types

- It's also possible to process instances of class types in a web service. These types can be passed to or returned from web service methods.
- An instance variable can be serialized only if it's `public` or has both a `get` and a `set` method.

Self-Review Exercises 32-65

- Properties that can be generated from the values of other properties should not be serialized to prevent redundancy.
- JAX-RS automatically converts arguments from an `@Path` annotation to the correct type, and it will return a “not found” error to the client if the argument cannot be converted from the `String` passed as part of the URL to the destination type. Supported types for conversion include integer types, floating-point types, `boolean` and the corresponding type-wrapper classes.

Self-Review Exercises

32.1 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) All methods of a web service class can be invoked by clients of that web service.
- b) When consuming a web service in a client application created in NetBeans, you must create the proxy class that enables the client to communicate with the web service.
- c) A proxy class communicating with a web service normally uses SOAP to send and receive messages.
- d) Session tracking is automatically enabled in a client of a web service.
- e) Web methods cannot be declared `static`.
- f) A user-defined type used in a web service must define both `get` and `set` methods for any property that will be serialized.
- g) Operations in a REST web service are defined by their own unique URLs.
- h) A SOAP-based web service can return data in JSON format.

32.2 Fill in the blanks for each of the following statements:

- a) A key difference between SOAP and REST is that SOAP messages have data wrapped in a(n) _____.
- b) A web service in Java is a(n) _____—it does not need to implement any interfaces or extend any classes.
- c) Web service requests are typically transported over the Internet via the _____ protocol.
- d) To set the exposed name of a web method, use the _____ element of the `@WebMethod` annotation.
- e) _____ transforms an object into a format that can be sent between a web service and a client.
- f) To return data in JSON format from a method of a REST-based web service, the `@Produces` annotation is set to _____.
- g) To return data in XML format from a method of a REST-based web service, the `@Produces` annotation is set to _____.

Answers to Self-Review Exercises

32.1 a) False. Only methods declared with the `@WebMethod` annotation can be invoked by a web service’s clients. b) False. The proxy class is created by NetBeans when you add a web service client to the application. c) True. d) False. In the JAX-WS framework, the client must indicate whether it wants to allow the web service to maintain session information. First, you must cast the proxy object to interface type `BindingProvider`, then use the `BindingProvider`’s `getRequestContext` method to obtain the `RequestContext` object. Finally, you must use the `RequestContext`’s `put` method to set the property `BindingProvider.SESSION_MAINTAIN_PROPERTY` to `true`. e) True. f) True. g) True. h) False. A SOAP web service implicitly returns data in XML format.

32.2 a) SOAP message or SOAP envelope. b) POJO (plain old Java object) c) HTTP. d) `operationName`. e) serialization. f) “`application/json`”. g) “`application/xml`”.

32_66 Chapter 32 REST Web Services

Exercises

32.3 (Phone Book Web Service) Create a RESTful web service that stores phone book entries in the database PhoneBookDB and a web client application that consumes this service. The web service should output XML. Use the steps in Section 31.2.1 to create the PhoneBook database and a data source name for accessing it. The database contains one table—PhoneBook—with three columns—LastName, FirstName and PhoneNumber. The LastName and FirstName columns store up to 30 characters. The PhoneNumber column supports phone numbers of the form (800) 555-1212 that contain 14 characters. Use the PhoneBookDB.sql script provided in the examples folder to create the PhoneBook table.

Give the client user the capability to enter a new contact (web method addEntry) and to find contacts by last name (web method getEntries). Pass only Strings as arguments to the web service. The getEntries web method should return an array of Strings that contains the matching phone book entries. Each String in the array should consist of the last name, first name and phone number for one phone book entry. These values should be separated by commas.

The SELECT query that will find a PhoneBook entry by last name should be:

```
SELECT LastName, FirstName, PhoneNumber
FROM PhoneBook
WHERE (LastName = LastName)
```

The INSERT statement that inserts a new entry into the PhoneBook database should be:

```
INSERT INTO PhoneBook (LastName, FirstName, PhoneNumber)
VALUES (LastName, FirstName, PhoneNumber)
```

32.4 (Phone Book Web Service Modification) Modify Exercise 32.3 so that it uses a class named PhoneBookEntry to represent a row in the database. The web service should return objects of type PhoneBookEntry in XML format for the getEntries method, and the client application should use the JAXB method unmarshal to retrieve the PhoneBookEntry objects.

32.5 (Phone-Book Web Service with JSON) Modify Exercise 32.4 so that the PhoneBookEntry class is passed to and from the web service as a JSON object. Use serialization to convert the JSON object into an object of type PhoneBookEntry.

32.6 (Blackjack Web Service Modification) Modify the Blackjack web service example in Section 32.9 to include class Card. Modify web method dealCard so that it returns an object of type Card and modify web method getHandValue so that it receives an array of Card objects from the client. Also modify the client application to keep track of what cards have been dealt by using ArrayLists of Card objects. The proxy class created by NetBeans will treat a web method's array parameter as a List, so you can pass these ArrayLists of Card objects directly to the getHandValue method. Your Card class should include set and get methods for the face and suit of the card.

32.7 (Project: Airline Reservation Web-Service Modification) Modify the airline reservation web service in Section 32.10 so that it contains two separate methods—one that allows users to view all available seats, and another that allows users to reserve a particular seat that is currently available. Use an object of type Ticket to pass information to and from the web service. The web service must be able to handle cases in which two users view available seats, one reserves a seat and the second user tries to reserve the same seat, not knowing that it's now taken. The names of the methods that execute should be reserve and getAllAvailableSeats.

32.8 (Project: Morse Code Web Service) In Exercise 14.22, you learned about Morse Code and wrote applications that could translate English phrases into Morse Code and vice versa. Create a SOAP-based web service that provides two methods—one that translates an English phrase into

Morse Code and one that translates Morse Code into English. Next, build a Morse Code translator GUI application that invokes the web service to perform these translations.

Making a Difference

32.9 (*Project: Spam Scanner Web Service*) In Exercise 14.27, you created a spam scanner application that scanned an e-mail and gave it a point rating based on the occurrence of certain words and phrases that commonly appear in spam e-mails and how many times the words and phrases occurred in the e-mail. Create a SOAP-based Spam scanner web service. Next, modify the GUI application you created in Exercise 14.27 to use the web service to scan an e-mail. Then display the point rating returned by the web service.

32.10 (*Project: SMS Web Service*) In Exercise 14.28, you created an SMS message-translator application. Create a SOAP-based web service with three methods:

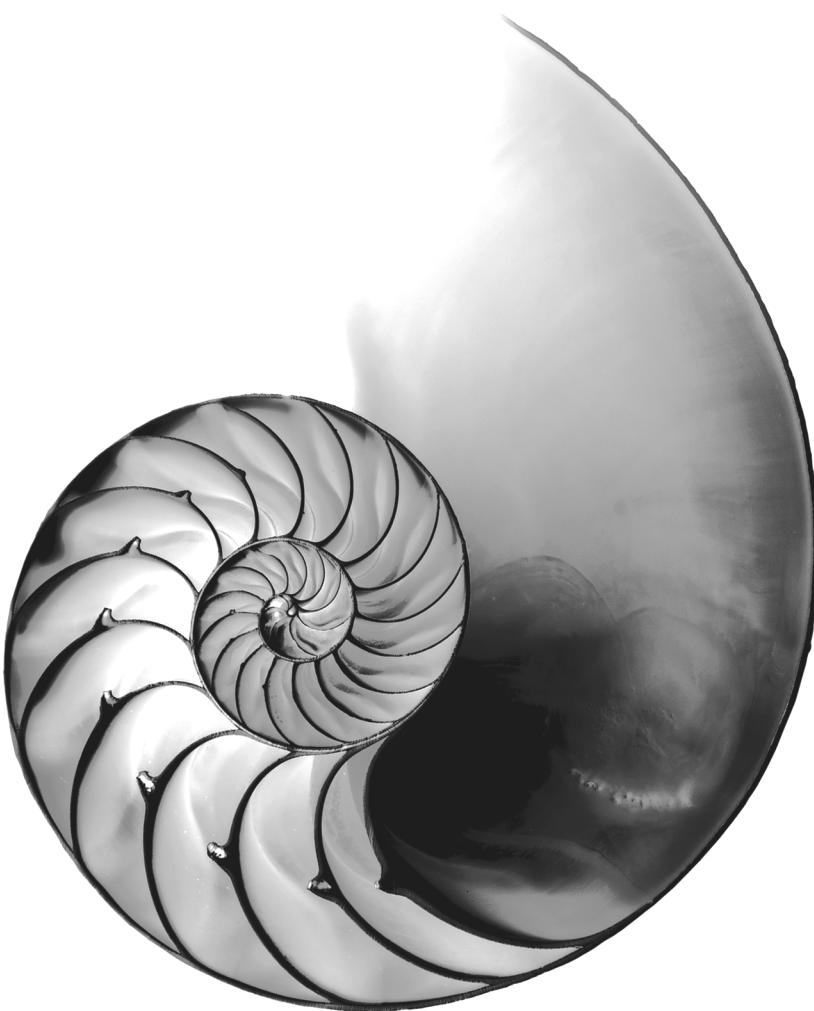
- a) one that receives an SMS abbreviation and returns the corresponding English word or phrase,
- b) one that receives an entire SMS message and returns the corresponding English text, and
- c) one that translates English text into an SMS message.

Use the web service from a GUI application that displays the web service's responses.

32.11 (*Project: Gender-Neutrality Web Service*) In Exercise 1.12, you researched eliminating sexism in all forms of communication. You then described the algorithm you'd use to read through a paragraph of text and replace gender-specific words with gender-neutral equivalents. Create a SOAP-based web service that receives a paragraph of text, then replaces gender-specific words with gender-neutral ones. Use the web service from a GUI application that displays the resulting gender-neutral text.

ATM Case Study, Part I: Object-Oriented Design with the UML

33



Objectives

In this chapter you'll learn:

- A simple object-oriented design methodology.
- What a requirements document is.
- To identify classes and class attributes from a requirements document.
- To identify objects' states, activities and operations from a requirements document.
- To determine the collaborations among objects in a system.
- To work with the UML's use case, class, state, activity, communication and sequence diagrams to graphically model an object-oriented system.



Outline

- | | |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 33.1 Case Study Introduction
33.2 Examining the Requirements Document
33.3 Identifying the Classes in a Requirements Document
33.4 Identifying Class Attributes | 33.5 Identifying Objects' States and Activities
33.6 Identifying Class Operations
33.7 Indicating Collaboration Among Objects
33.8 Wrap-Up |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Answers to Self-Review Exercises

33.1 Case Study Introduction

Now we begin the *optional* portion of our object-oriented design and implementation case study. In this chapter and Chapter 34, you'll design and implement an object-oriented automated teller machine (ATM) software system. The case study provides you with a concise, carefully paced, complete design and implementation experience. In Sections 33.2–33.7 and 34.2–34.3, you'll perform the steps of an object-oriented design (OOD) process using the UML while relating these steps to the object-oriented concepts discussed in Chapters 2–10. In this chapter, you'll work with six popular types of UML diagrams to graphically represent the design. In Chapter 34, you'll tune the design with inheritance, then fully implement the ATM as a Java application (Section 34.4). This is not an exercise; rather, it's an end-to-end learning experience that concludes with a detailed walkthrough of the complete Java code that implements our design.

These chapters can be studied as a continuous unit after you've completed the introduction to object-oriented programming in Chapters 8–11.

33.2 Examining the Requirements Document

We begin our design process by presenting a **requirements document** that specifies the purpose of the ATM system and *what* it must do. Throughout the case study, we refer often to this requirements document.

Requirements Document

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 33.1). Each user can have only one account at the bank. ATM users should be able to view their account balance, withdraw cash (i.e., take money out of an account) and deposit funds (i.e., place money into an account). The user interface of the automated teller machine contains:

- a screen that displays messages to the user
- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.

The cash dispenser begins each day loaded with 500 \$20 bills. [Note: Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user's account number from an ATM card, whereas this ATM asks the user to type the account number on the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

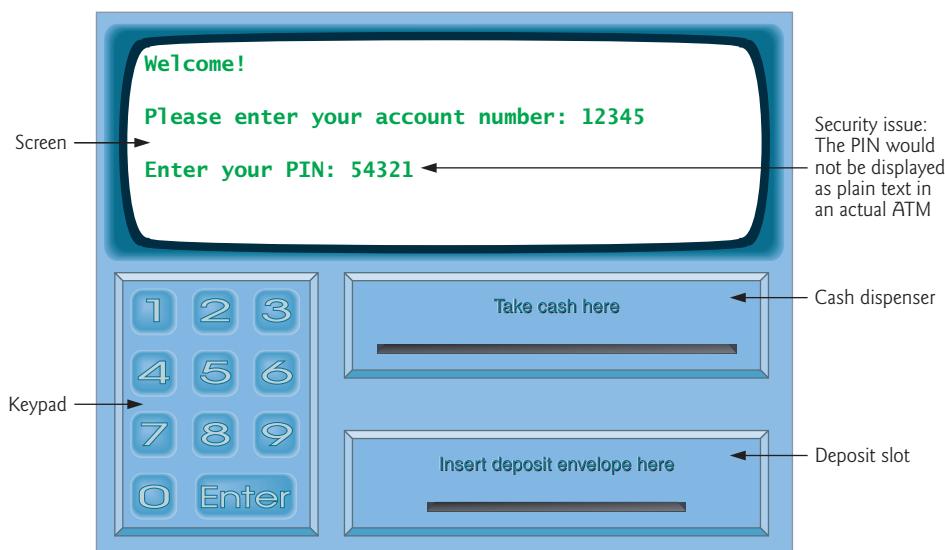


Fig. 33.1 | Automated teller machine user interface.

The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM's hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version to run on a personal computer. This version should use the computer's monitor to simulate the ATM's screen, and the computer's keyboard to simulate the ATM's keypad.

An ATM session consists of authenticating a user (i.e., proving the user's identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank's account information database (i.e., an organized collection of data stored on a computer; database access was presented in Chapter 24). For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [Note: We assume that the bank plans to build only one ATM, so we need not worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM. Also, any

business system like an ATM faces complex and challenging security issues that are beyond the scope of this case study. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM (assuming no one is currently using it), the user should experience the following sequence of events (shown in Fig. 33.1):

1. The screen displays `Welcome!` and prompts the user to enter an account number.
2. The user enters a five-digit account number using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN using the keypad.¹
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 33.2). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to *Step 1* to restart the authentication process.

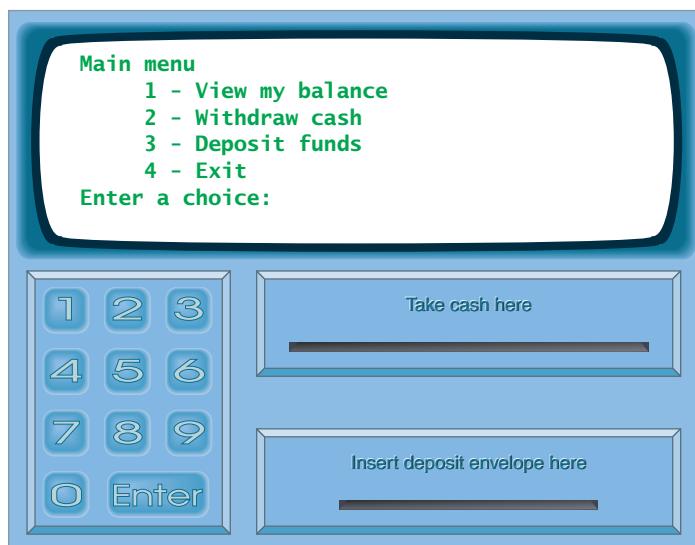


Fig. 33.2 | ATM main menu.

After the ATM authenticates the user, the main menu (Fig. 33.2) should contain a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). It also should contain an option to allow the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4).

1. In this simple, command-line, text-based ATM, as you type the PIN, it appears on the screen. This is an obvious security breach—you would not want someone looking over your shoulder at an ATM and seeing your PIN displayed on the screen.

If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database. The following steps describe what occurs when the user enters 2 to make a withdrawal:

1. The screen displays a menu (Fig. 33.3) containing standard withdrawal amounts: \$20 (option 1), \$40 (option 2), \$60 (option 3), \$100 (option 4) and \$200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).

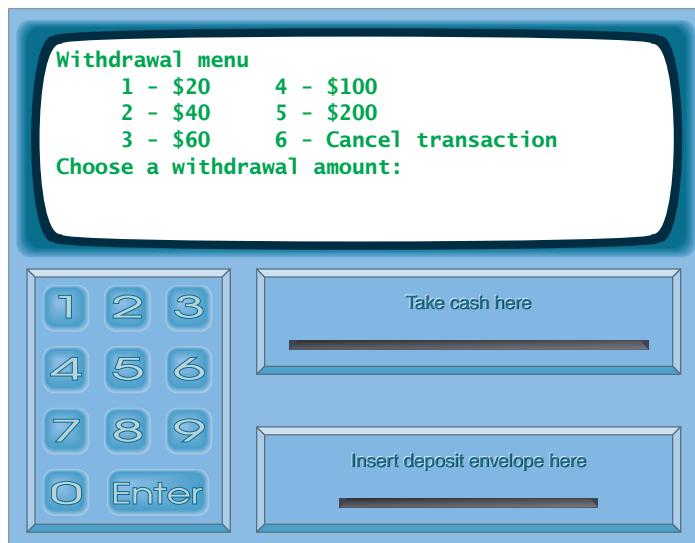


Fig. 33.3 | ATM withdrawal menu.

2. The user enters a menu selection using the keypad.
3. If the withdrawal amount chosen is greater than the user's account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to *Step 1*. If the withdrawal amount chosen is less than or equal to the user's account balance (i.e., an acceptable amount), the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu and waits for user input.
4. If the cash dispenser contains enough cash, the ATM proceeds to *Step 5*. Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then returns to *Step 1*.
5. The ATM debits the withdrawal amount from the user's account in the bank's database (i.e., subtracts the withdrawal amount from the user's account balance).
6. The cash dispenser dispenses the desired amount of money to the user.
7. The screen displays a message reminding the user to take the money.

The following steps describe the actions that occur when the user enters 3 (when viewing the main menu of Fig. 33.2) to make a deposit:

1. The screen prompts the user to enter a deposit amount or type 0 (zero) to cancel.
2. The user enters a deposit amount or 0 using the keypad. [Note: The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., \$27.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 2725). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., $2725 \div 100 = 27.25$).]
3. If the user specifies a deposit amount, the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu and waits for user input.
4. The screen displays a message telling the user to insert a deposit envelope.
5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits the deposit amount to the user's account in the bank's database (i.e., adds the deposit amount to the user's account balance). [Note: This money is *not* immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the envelope must clear (i.e., money must be transferred from the check writer's account to the check recipient's account). When either of these events occurs, the bank appropriately updates the user's balance stored in its database. This occurs independently of the ATM system.] If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, it should return to the main menu so that the user can perform additional transactions. If the user exits the system, the screen should display a thank you message, then display the welcome message for the next user.

Analyzing the ATM System

The preceding statement is a simplified example of a requirements document. Typically, such a document is the result of a detailed process of **requirements gathering**, which might include interviews with possible users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements document for banking software (e.g., the ATM system described here) might interview banking experts to gain a better understanding of what the software must do. The analyst would use the information gained to compile a list of **system requirements** to guide systems designers as they design the system.

The process of requirements gathering is a key task of the first stage of the software life cycle. The **software life cycle** specifies the stages through which software goes from the time it's first conceived to the time it's retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages. **Waterfall models** perform each stage once in succession, whereas **iterative models** may *repeat* one or more stages several times throughout a product's life cycle.

The analysis stage focuses on defining the problem to be solved. When designing any system, one must *solve the problem right*, but of equal importance, one must *solve the right*

problem. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements document describes the requirements of our ATM system in sufficient detail that you need not go through an extensive analysis stage—it's been done for you.

To capture what a proposed system should do, developers often employ a technique known as **use case modeling**. This process identifies the **use cases** of the system, each representing a different capability that the system provides to its clients. For example, ATMs typically have several use cases, such as “View Account Balance,” “Withdraw Cash,” “Deposit Funds,” “Transfer Funds Between Accounts” and “Buy Postage Stamps.” The simplified ATM system we build in this case study allows only the first three.

Each use case describes a typical scenario for which the user uses the system. You've already read descriptions of the ATM system's use cases in the requirements document; the lists of steps required to perform each transaction type (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—“View Account Balance,” “Withdraw Cash” and “Deposit Funds,” respectively.

Use Case Diagrams

We now introduce the first of several UML diagrams in the case study. We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and its use cases. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout this case study). Use case diagrams are often accompanied by informal text that gives more detail—like the text that appears in the requirements document. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 33.4 shows the use case diagram for our ATM system. The stick figure represents an **actor**, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. A use case diagram can include multiple actors. For example, the use case diagram for a real bank's ATM system might also include an actor named Administrator who refills the cash dispenser each day.

Our requirements document supplies the actors—“ATM users should be able to view their account balance, withdraw cash and deposit funds.” Therefore, the actor in each of the three use cases is the user who interacts with the ATM. An external entity—a real person—plays the part of the user to perform financial transactions. Figure 33.4 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

Software engineers (more precisely, systems designers) must analyze the requirements document or a set of use cases and design the system before programmers implement it in a particular programming language. During the analysis stage, systems designers focus on understanding the requirements document to produce a high-level specification that describes *what* the system is supposed to do. The output of the design stage—a **design specification**—should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several sections, we perform the steps of a simple object-oriented

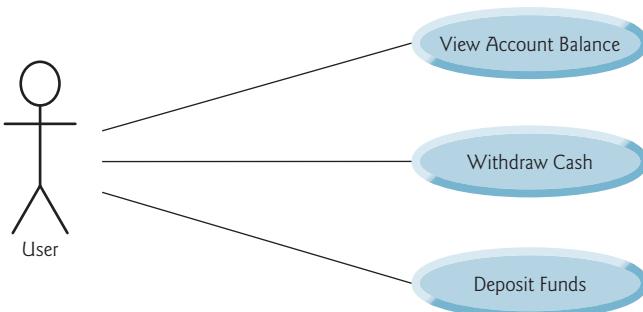


Fig. 33.4 | Use case diagram for the ATM system from the User's perspective.

design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text.

The UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation, now part of IBM. RUP is a rich process intended for designing “industrial strength” applications. For this case study, we present our own simplified design process.

Designing the ATM System

We now begin the design stage of our ATM system. A **system** is a set of components that interact to solve a problem. For example, to perform the ATM system’s designated tasks, our ATM system has a user interface (Fig. 33.1), and contains software that executes financial transactions and interacts with a database of bank account information. **System structure** describes the system’s objects and their interrelationships. **System behavior** describes how the system changes as its objects interact with one another.

Every system has both structure and behavior—designers must specify both. There are several types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 standard specifies 13 diagram types for documenting the system models. Each models a distinct characteristic of a system’s structure or behavior—six diagrams relate to system structure, the remaining seven to system behavior. We list here only the six diagram types used in our case study—one models system structure; the other five model system behavior.

1. **Use case diagrams**, such as the one in Fig. 33.4, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as “View Account Balance,” “Withdraw Cash” and “Deposit Funds”).
2. **Class diagrams**, which you’ll study in Section 33.3, model the classes, or “building blocks,” used in a system. Each noun or “thing” described in the requirements document is a candidate to be a class in the system (e.g., **Account**, **Keypad**). Class diagrams help us specify the *structural relationships* between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically *composed of* a screen, a keypad, a cash dispenser and a deposit slot.

3. **State machine diagrams**, which you'll study in Section 33.5, model the ways in which an object changes state. An object's **state** is indicated by the values of all its attributes at a given time. When an object changes state, it may behave differently in the system. For example, after validating a user's PIN, the ATM transitions from the "user not authenticated" state to the "user authenticated" state, at which point it allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).
4. **Activity diagrams**, which you'll also study in Section 33.5, model an object's **activity**—is workflow (sequence of events) during program execution. An activity diagram models the *actions* the object performs and specifies the *order* in which it performs them. For example, an activity diagram shows that the ATM must obtain the balance of the user's account (from the bank's account information database) *before* the screen can display the balance to the user.
5. **Communication diagrams** (called **collaboration diagrams** in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You'll learn in Section 33.7 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank's account information database to retrieve an account balance.
6. **Sequence diagrams** also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You'll learn in Section 33.7 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 33.3, we continue designing our ATM system by identifying the classes from the requirements document. We accomplish this by extracting key *nouns and noun phrases* from the requirements document. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

Web Resource

We've created an extensive UML Resource Center that contains many links to additional information, including introductions, tutorials, blogs, books, certification, conferences, developer tools, documentation, e-books, FAQs, forums, groups, UML in Java, podcasts, security, tools, downloads, training courses, videos and more. Browse our UML Resource Center at www.deitel.com/UML/.

Self-Review Exercises for Section 33.2

- 33.1** Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 33.4 to reflect this change.
- 33.2** _____ model the interactions among objects in a system with an emphasis on *when* these interactions occur.
- Class diagrams
 - Sequence diagrams
 - Communication diagrams
 - Activity diagrams

- 33.3** Which of the following choices lists stages of a typical software life cycle in sequential order?
- design, analysis, implementation, testing
 - design, analysis, testing, implementation
 - analysis, design, testing, implementation
 - analysis, design, implementation, testing

33.3 Identifying the Classes in a Requirements Document

Now we begin designing the ATM system. In this section, we identify the classes that are needed to build the system by analyzing the *nouns* and *noun phrases* that appear in the requirements document. We introduce UML class diagrams to model these classes. This is an important first step in defining the system's structure.

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We'll eventually describe these classes using UML class diagrams and implement these classes in Java. First, we review the requirements document of Section 33.2 and identify key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these are actually attributes of other classes in the system. We may also conclude that some of the nouns do not correspond to parts of the system and thus should not be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 33.5 lists the nouns and noun phrases found in the requirements document. We list them from left to right in the order in which we first encounter them. We list only the singular form of each.

Nouns and noun phrases in the ATM requirements document

bank	money / funds	account number	ATM
screen	PIN	user	keypad
bank database	customer	cash dispenser	balance inquiry
transaction	\$20 bill / cash	withdrawal	account
deposit slot	deposit	balance	deposit envelope

Fig. 33.5 | Nouns and noun phrases in the ATM requirements document.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We don't model "bank" as a class, because the bank is not a part of the ATM system—the bank simply wants us to build the ATM. "Customer" and "user" also represent outside entities—they're important because they *interact* with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 33.4.

We do not model "\$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they're not part of what is being automated. We can ade-

quately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to the ATM system's classes in Section 33.4.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements document does not say anything about what the system should do with deposit envelopes after it receives them. We can assume that simply acknowledging the receipt of an envelope—an operation performed by the class that models the deposit slot—is sufficient to represent the presence of an envelope in the system. We assign operations to the ATM system's classes in Section 33.6.

In our simplified ATM system, representing various amounts of “money,” including an account’s “balance,” as attributes of classes seems most appropriate. Likewise, the nouns “account number” and “PIN” represent significant pieces of information in the ATM system. They’re important attributes of a bank account. They do not, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements document frequently describes a “transaction” in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., “balance inquiry,” “withdrawal” and “deposit”) as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of the withdrawal. A balance inquiry, however, does not require any additional data other than the account number. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. In Section 34.3, we “factor out” common features of all transactions into a general “transaction” class using the object-oriented concept of inheritance.

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 33.5. Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry
- withdrawal
- deposit

The elements of this list are likely to be classes that we'll need to implement our system.

We can now model the classes in our system based on the list we've created. We capitalize class names in the design process—a UML convention—as we'll do when we write the actual Java code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., `MultipleWordName`). Using this convention, we create classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` and `Deposit`. We construct

our system using these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Modeling Classes

The UML enables us to model, via **class diagrams**, the classes in the ATM system and their interrelationships. Figure 33.6 represents class ATM. Each class is modeled as a rectangle with three compartments. The top one contains the name of the class centered horizontally in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Sections 33.4–33.5.) The bottom compartment contains the class's operations (discussed in Section 33.6). In Fig. 33.6, the middle and bottom compartments are empty because we've not yet determined this class's attributes and operations.



Fig. 33.6 | Representing a class in the UML using a class diagram.

Class diagrams also show the relationships between the classes of the system. Figure 33.7 shows how our classes ATM and Withdrawal relate to one another. For the moment, for simplicity, we choose to model only this subset of classes. We present a more complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner to create more readable diagrams, when appropriate. Such a diagram is said to be an **elided diagram**—one in which some information, such as the contents of the second and third compartments, is *not* modeled. We'll place information in these compartments in Sections 33.4–33.6.

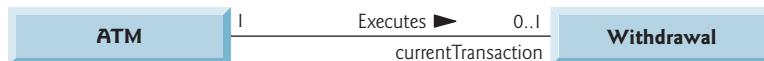


Fig. 33.7 | Class diagram showing an association among classes.

In Fig. 33.7, the solid line that connects the two classes represents an **association**—a relationship between classes. The numbers near each end of the line are **multiplicity** values, which indicate how many objects of each class participate in the association. In this case, following the line from left to right reveals that, at any given moment, one ATM object participates in an association with either zero or one Withdrawal objects—zero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 33.8 lists and explains the multiplicity types.

An association can be named. For example, the word **Executes** above the line connecting classes ATM and Withdrawal in Fig. 33.7 indicates the name of that association. This part of the diagram reads “one object of class ATM executes zero or one objects of class Withdrawal.” Association names are *directional*, as indicated by the filled arrowhead—so

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, n	m or n
$m..n$	At least m , but not more than n
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig. 33.8 | Multiplicity types.

it would be improper, for example, to read the preceding association from right to left as “zero or one objects of class `Withdrawal` execute one object of class `ATM`.”

The word `currentTransaction` at the `Withdrawal` end of the association line in Fig. 33.7 is a **role name**, identifying the role the `Withdrawal` object plays in its relationship with the `ATM`. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of “professor” when relating to students. The same person may take on the role of “colleague” when participating in an association with another professor, and “coach” when coaching student athletes. In Fig. 33.7, the role name `currentTransaction` indicates that the `Withdrawal` object participating in the `Executes` association with an object of class `ATM` represents the transaction currently being processed by the `ATM`. In other contexts, a `Withdrawal` object may take on other roles (e.g., the “previous transaction”). Notice that we do not specify a role name for the `ATM` end of the `Executes` association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being *composed* of objects of other classes. Consider a real-world automated teller machine. What “pieces” does a manufacturer put together to build a working `ATM`? Our requirements document tells us that the `ATM` is composed of a screen, a keypad, a cash dispenser and a deposit slot.

In Fig. 33.9, the **solid diamonds** attached to the `ATM` class’s association lines indicate that `ATM` has a **composition** relationship with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. Composition implies a *whole/part relationship*. The class that has the composition symbol (the solid diamond) on its end of the association line is the *whole* (in this case, `ATM`), and the classes on the other end of the association lines are the *parts*—in this case, `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. The compositions in Fig. 33.9 indicate that an object of class `ATM` is formed from one object of class `Screen`, one object of class `CashDispenser`, one object of class `Keypad` and one object of class `DepositSlot`. The `ATM` *has a* screen, a keypad, a cash dispenser and a deposit slot. (As we saw in Chapter 9, the *is-a* relationship defines inheritance. We’ll see in Section 34.3 that there’s a nice opportunity to use inheritance in the `ATM` system design.)

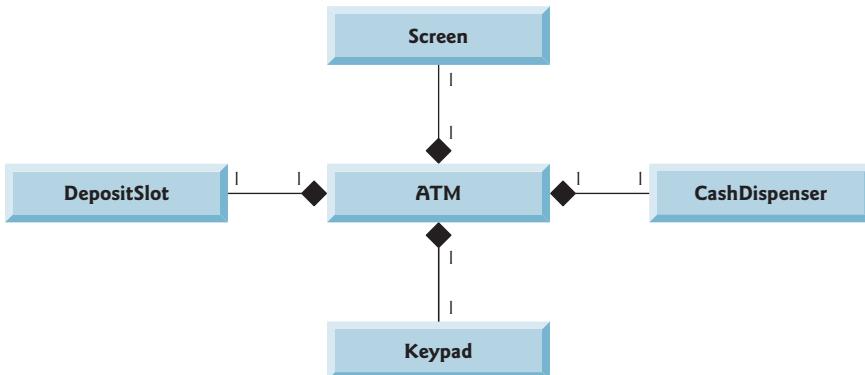


Fig. 33.9 | Class diagram showing composition relationships.

According to the UML specification (www.omg.org/technology/documents/formal/uml.htm), composition relationships have the following properties:

1. Only one class in the relationship can represent the *whole* (i.e., the diamond can be placed on only *one* end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.
2. The *parts* in the composition relationship exist only as long as the whole does, and the whole is responsible for the creation and destruction of its parts. For example, the act of constructing an ATM includes manufacturing its parts. Also, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.
3. A *part* may belong to only one *whole* at a time, although it may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these properties. If a *has-a* relationship does not satisfy one or more of these criteria, the UML specifies that **hollow diamonds** be attached to the ends of association lines to indicate **aggregation**—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer *has a* monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating composition's second and third properties.

Figure 33.10 shows a class diagram for the ATM system. This diagram models most of the classes that we've identified, as well as the associations between them that we can infer from the requirements document. Classes **BalanceInquiry** and **Deposit** participate in associations similar to those of class **Withdrawal**, so we've chosen to omit them from this diagram to keep it simple. In Section 34.3, we expand our class diagram to include all the classes in the ATM system.

Figure 33.10 presents a graphical model of ATM system's structure. It includes classes **BankDatabase** and **Account**, and several associations that were not present in either Fig. 33.7 or Fig. 33.9. It shows that class **ATM** has a **one-to-one relationship** with class **BankDatabase**—one **ATM** object *authenticates users against* one **BankDatabase** object. In

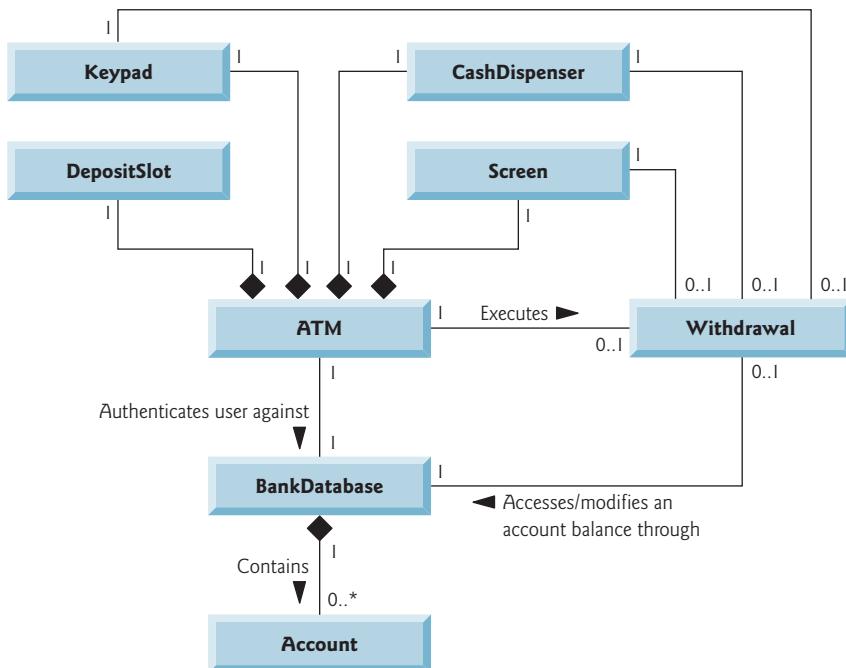


Fig. 33.10 | Class diagram for the ATM system model.

Fig. 33.10, we also model the fact that the bank's database contains information about many accounts—one **BankDatabase** object participates in a *composition* relationship with zero or more **Account** objects. The multiplicity value $0..*$ at the **Account** end of the association between class **BankDatabase** and class **Account** indicates that zero or more objects of class **Account** take part in the association. Class **BankDatabase** has a **one-to-many relationship** with class **Account**—the **BankDatabase** can contain many **Accounts**. Similarly, class **Account** has a **many-to-one relationship** with class **BankDatabase**—there can be many **Accounts** stored in the **BankDatabase**. Recall from Fig. 33.8 that the multiplicity value $*$ is identical to $0..*$. We include $0..*$ in our class diagrams for clarity.

Figure 33.10 also indicates that at any given time 0 or 1 **Withdrawal** objects can exist. If the user is performing a withdrawal, “one object of class **Withdrawal** accesses/modifies an account balance through one object of class **BankDatabase**.” We could have created an association directly between class **Withdrawal** and class **Account**. The requirements document, however, states that the “ATM must interact with the bank's account information database” to perform transactions. A bank account contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the **BankDatabase** can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 33.10 also models associations between class **Withdrawal** and classes **Screen**, **CashDispenser** and **Keypad**. A withdrawal transaction includes prompting the user to choose a withdrawal amount, and receiving numeric input. These

actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 33.10, take part in several associations with the other classes of the ATM system. Like class `Withdrawal`, each of these classes associates with classes `ATM` and `BankDatabase`. An object of class `BalanceInquiry` also associates with an object of class `Screen` to display the balance of an account to the user. Class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class `Deposit` accesses the deposit slot.

We've now identified the initial classes in our ATM system—we may discover others as we proceed with the design and implementation. In Section 33.4 we determine the attributes for each of these classes, and in Section 33.5 we use these attributes to examine how the system changes over time.

Self-Review Exercises for Section 33.3

33.4 Suppose we have a class `Car` that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 33.9) that models some of the composition relationships of class `Car`.

33.5 Suppose we have a class `File` that represents an electronic document in a standalone, non-networked computer represented by class `Computer`. What sort of association exists between class `Computer` and class `File`?

- Class `Computer` has a one-to-one relationship with class `File`.
- Class `Computer` has a many-to-one relationship with class `File`.
- Class `Computer` has a one-to-many relationship with class `File`.
- Class `Computer` has a many-to-many relationship with class `File`.

33.6 State whether the following statement is *true* or *false*, and if *false*, explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

33.7 Modify the class diagram of Fig. 33.10 to include class `Deposit` instead of class `Withdrawal`.

33.4 Identifying Class Attributes

[Note: This section may be read after Chapter 4.]

Classes have attributes (data) and operations (behaviors). Class attributes are implemented as fields, and class operations are implemented as methods. In this section, we determine many of the attributes needed in the ATM system. In Section 33.5 we examine how these attributes represent an object's state. In Section 33.6 we determine class operations.

Identifying Attributes

Consider the attributes of some real-world objects: A person's attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio's attributes include its station, volume and AM or FM settings. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it's in. A personal computer's attributes include its manufacturer (e.g., Dell, Sun, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements document. For each such word and phrase we find

that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 33.3. We also create attributes to represent any additional data that a class may need, as such needs become clear throughout the design process.

Figure 33.11 lists the words or phrases from the requirements document that describe each class. We formed this list by reading the requirements document and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements document describes the steps taken to obtain a “withdrawal amount,” so we list “amount” next to class `Withdrawal`.

Class	Descriptive words and phrases
ATM	user is authenticated
BalanceInquiry	account number
Withdrawal	account number amount
Deposit	account number amount
BankDatabase	<i>[no descriptive words or phrases]</i>
Account	account number PIN balance
Screen	<i>[no descriptive words or phrases]</i>
Keypad	<i>[no descriptive words or phrases]</i>
CashDispenser	begins each day loaded with 500 \$20 bills
DepositSlot	<i>[no descriptive words or phrases]</i>

Fig. 33.11 | Descriptive words and phrases from the ATM requirements document.

Figure 33.11 leads us to create one attribute of class `ATM`. Class `ATM` maintains information about the state of the ATM. The phrase “user is authenticated” describes a state of the ATM (we introduce states in Section 33.5), so we include `userAuthenticated` as a **Boolean attribute** (i.e., an attribute that has a value of either `true` or `false`) in class `ATM`. The Boolean attribute type in the UML is equivalent to the `boolean` type in Java. This attribute indicates whether the ATM has successfully authenticated the current user—`userAuthenticated` must be `true` for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share one attribute. Each transaction involves an “account number” that corresponds to the account of the user making the transaction. We assign an integer attribute `accountNumber` to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements document also suggest some differences in the attributes required by each transaction class. The requirements document indicates that to withdraw cash or deposit funds, users must input a specific “amount” of money to be withdrawn or deposited, respectively. Thus, we assign to classes `Withdrawal`

and `Deposit` an attribute `amount` to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for these transactions to take place. Class `BalanceInquiry`, however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class `Account` has several attributes. The requirements document states that each bank account has an “account number” and “PIN,” which the system uses for identifying accounts and authenticating users. We assign to class `Account` two integer attributes: `accountNumber` and `pin`. The requirements document also specifies that an account maintains a “balance” of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes: `availableBalance` and `totalBalance`. Attribute `availableBalance` tracks the amount of money that a user can withdraw from the account. Attribute `totalBalance` refers to the total amount of money that the user has “on deposit” (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits \$50.00 into an empty account. The `totalBalance` attribute would increase to \$50.00 to record the deposit, but the `availableBalance` would remain at \$0. [Note: We assume that the bank updates the `availableBalance` attribute of an `Account` some length of time after the ATM transaction occurs, in response to confirming that \$50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

Class `CashDispenser` has one attribute. The requirements document states that the cash dispenser “begins each day loaded with 500 \$20 bills.” The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class `CashDispenser` an integer attribute `count`, which is initially set to 500.

For real problems in industry, there’s no guarantee that requirements documents will be precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we will continue to add, modify and delete information about the classes in our system.

Modeling Attributes

The class diagram in Fig. 33.12 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 33.11 lead us to identify these attributes. For simplicity, Fig. 33.12 does not show the associations among classes—we showed these in Fig. 33.10. This is a common practice of systems designers when designs are being developed. Recall from Section 33.3 that in the UML, a class’s attributes are placed in the middle compartment of the class’s rectangle. We list each attribute’s name and type separated by a colon (:), followed in some cases by an equal sign (=) and an initial value.

Consider the `userAuthenticated` attribute of class `ATM`:

```
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The **attribute name** is `userAuthenticated`. The **attribute type** is `Boolean`. In Java, an attribute can be represented by a primitive type, such as `boolean`, `int` or `double`, or a reference type like a class. We've chosen to model only primitive-type attributes in Fig. 33.12—we discuss the reasoning behind this decision shortly. The attribute types in Fig. 33.12 are in UML notation. We'll associate the types `Boolean`, `Integer` and `Double` in the UML diagram with the primitive types `boolean`, `int` and `double` in Java, respectively.

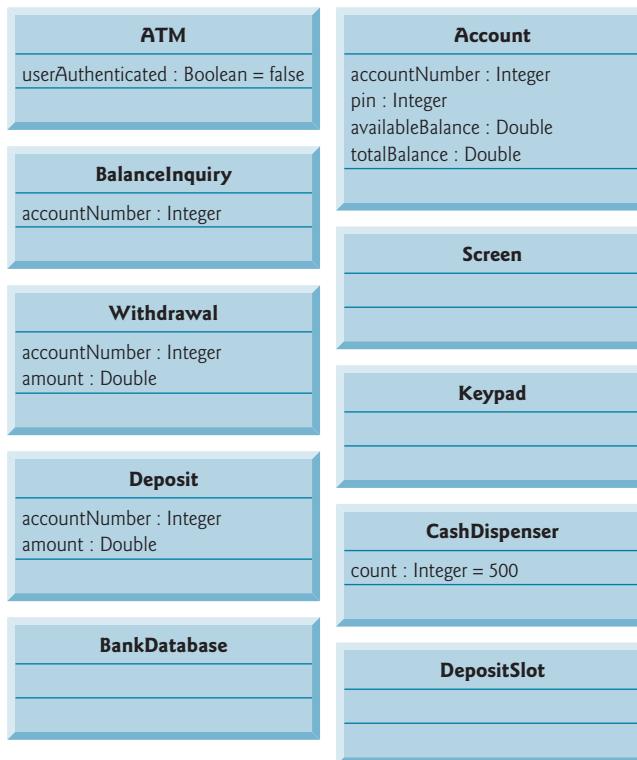


Fig. 33.12 | Classes with attributes.

We can also indicate an initial value for an attribute. The `userAuthenticated` attribute in class `ATM` has an initial value of `false`. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the `accountNumber` attribute of class `BalanceInquiry` is an integer. Here we show no initial value, because the value of this attribute is a number that we do not yet know. This number will be determined at execution time based on the account number entered by the current `ATM` user.

Figure 33.12 does not include attributes for classes `Screen`, `Keypad` and `DepositSlot`. These are important components of our system, for which our design process has not yet revealed any attributes. We may discover some, however, in the remaining phases of design or when we implement these classes in Java. This is perfectly normal.



Software Engineering Observation 33.1

At early stages in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.

Figure 33.12 also does not include attributes for class `BankDatabase`. Recall that attributes in Java can be represented by either primitive types or reference types. We've chosen to include only primitive-type attributes in the class diagram in Fig. 33.12 (and in similar class diagrams throughout the case study). A reference-type attribute is modeled more clearly as an association between the class holding the reference and the class of the object to which the reference points. For example, the class diagram in Fig. 33.10 indicates that class `BankDatabase` participates in a composition relationship with zero or more `Account` objects. From this composition, we can determine that when we implement the ATM system in Java, we'll be required to create an attribute of class `BankDatabase` to hold references to zero or more `Account` objects. Similarly, we can determine reference-type attributes of class `ATM` that correspond to its composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. These composition-based attributes would be redundant if modeled in Fig. 33.12, because the compositions modeled in Fig. 33.10 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 33.12 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 33.5 we identify the states and activities of the objects in the model, and in Section 33.6 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we'll continue to strengthen the structure of our model.

Self-Review Exercises for Section 33.4

33.8 We typically identify the attributes of the classes in our system by analyzing the _____ in the requirements document.

- a) nouns and noun phrases
- b) descriptive words and phrases
- c) verbs and verb phrases
- d) All of the above.

33.9 Which of the following is *not* an attribute of an airplane?

- a) length
- b) wingspan
- c) fly
- d) number of seats

33.10 Describe the meaning of the following attribute declaration of class `CashDispenser` in the class diagram in Fig. 33.12:

```
count : Integer = 500
```

33.5 Identifying Objects' States and Activities

In Section 33.4, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 33.12. We now show how these attributes represent an object's state. We identify some key states that our objects may occupy and discuss how objects *change state* in response to various events occurring in the system. We also discuss the workflow, or **activities**, that objects perform in the ATM system, and we present the activities of BalanceInquiry and Withdrawal transaction objects.

State Machine Diagrams

Each object in a system goes through a series of states. An object's state is indicated by the values of its attributes at a given time. **State machine diagrams** (commonly called **state diagrams**) model several states of an object and show under what circumstances the object changes state. Unlike the class diagrams presented in earlier case study sections, which focused primarily on the system's *structure*, state diagrams model some of the system's *behavior*.

Figure 33.13 is a simple state diagram that models some of the states of an object of class ATM. The UML represents each state in a state diagram as a **rounded rectangle** with the name of the state placed inside it. A **solid circle** with an attached stick (\Rightarrow) arrowhead designates the **initial state**. Recall that we modeled this state information as the Boolean attribute userAuthenticated in the class diagram of Fig. 33.12. This attribute is initialized to false, or the "User not authenticated" state, according to the state diagram.

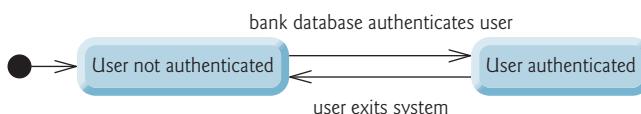


Fig. 33.13 | State diagram for the ATM object.

The arrows with stick (\Rightarrow) arrowhead indicate **transitions** between states. An object can transition from one state to another in response to various *events* that occur in the system. The name or description of the event that causes a transition is written near the line that corresponds to the transition. For example, the ATM object changes from the "User not authenticated" to the "User authenticated" state after the database authenticates the user. Recall from the requirements document that the database authenticates a user by comparing the account number and PIN entered by the user with those of an account in the database. If the user has entered a valid account number and the correct PIN, the ATM object transitions to the "User authenticated" state and changes its userAuthenticated attribute to a value of true. When the user exits the system by choosing the "exit" option from the main menu, the ATM object returns to the "User not authenticated" state.



Software Engineering Observation 33.2

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only key states and state transitions.

Activity Diagrams

Like a state diagram, an activity diagram models aspects of system behavior. Unlike a state diagram, an activity diagram models an object's **workflow** (sequence of events) during program execution. An activity diagram models the **actions** the object will perform and in what *order*. The activity diagram in Fig. 33.14 models the actions involved in executing a balance-inquiry transaction. We assume that a `BalanceInquiry` object has already been initialized and assigned a valid account number (that of the current user), so the object knows which balance to retrieve. The diagram includes the actions that occur after the user selects a balance inquiry from the main menu and before the ATM returns the user to the main menu—a `BalanceInquiry` object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the balance of the account from the database. Next, the `BalanceInquiry` displays the balance on the screen. This action completes the execution of the transaction. Recall that we've chosen to represent an account balance as both the `availableBalance` and `totalBalance` attributes of class `Account`, so the actions modeled in Fig. 33.14 refer to the retrieval and display of *both* balance attributes.

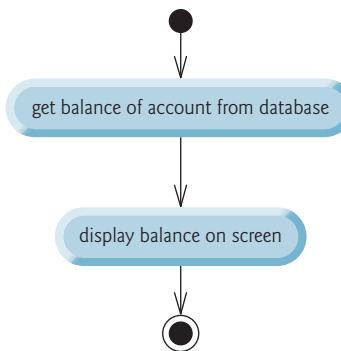


Fig. 33.14 | Activity diagram for a `BalanceInquiry` object.

The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an *action expression*—for example, “get balance of account from database”—that specifies an action to be performed. An arrow with a stick (\Rightarrow) arrowhead connects two action states, indicating the order in which the actions represented by the action states occur. The solid circle (at the top of Fig. 33.14) represents the activity's *initial state*—the beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the “get balance of account from database” action expression. The transaction then displays *both* balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 33.14) represents the *final state*—the end of the workflow after the object performs the modeled actions. We used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapters 3–4.

Figure 33.15 shows an activity diagram for a withdrawal transaction. We assume that a `Withdrawal` object has been assigned a valid account number. We do not model the user selecting a withdrawal from the main menu or the ATM returning the user to the main

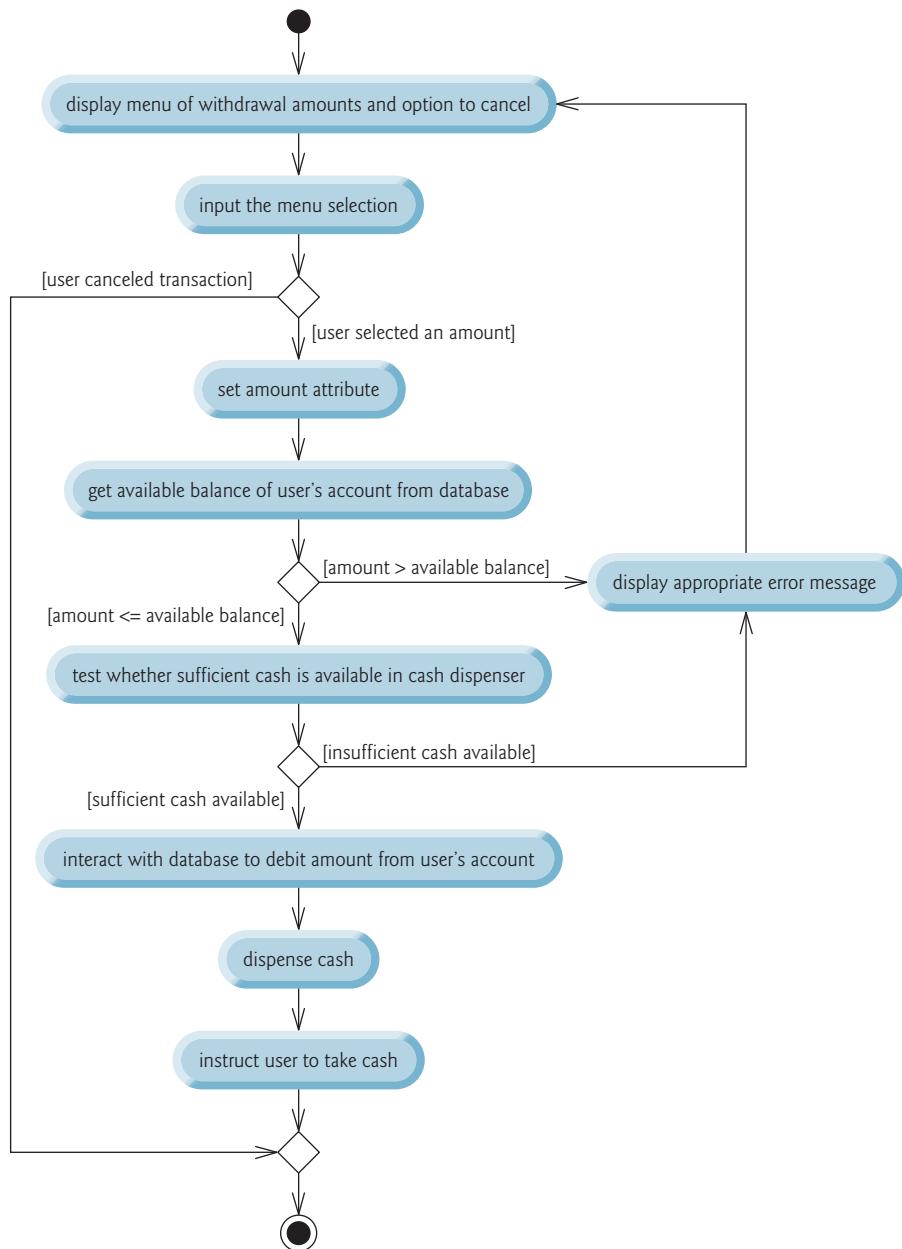


Fig. 33.15 | Activity diagram for a withdrawal transaction.

menu because these are not actions performed by a `Withdrawal` object. The transaction first displays a menu of standard withdrawal amounts (shown in Fig. 33.3) and an option to cancel the transaction. The transaction then receives a menu selection from the user. The activity flow now arrives at a decision (a fork indicated by the small diamond symbol).

This point determines the next action based on the associated guard condition (in square brackets next to the transition), which states that the transition occurs if this guard condition is met. If the user cancels the transaction by choosing the “cancel” option from the menu, the activity flow immediately skips to the final state. Note the merge (indicated by the small diamond symbol) where the cancellation flow of activity joins the main flow of activity before reaching the activity’s final state. If the user selects a withdrawal amount from the menu, *Withdrawal sets amount* (an attribute originally modeled in Fig. 33.12) to the value chosen by the user.

After setting the withdrawal amount, the transaction retrieves the available balance of the user’s account (i.e., the *availableBalance* attribute of the user’s *Account* object) from the database. The activity flow then arrives at another decision. If the requested withdrawal amount exceeds the user’s available balance, the system displays an appropriate error message informing the user of the problem, then returns to the beginning of the activity diagram and prompts the user to input a new amount. If the requested withdrawal amount is less than or equal to the user’s available balance, the transaction proceeds. The transaction next tests whether the cash dispenser has enough cash remaining to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message, then returns to the beginning of the activity diagram and prompts the user to choose a new amount. If sufficient cash is available, the transaction interacts with the database to debit the withdrawal amount from the user’s account (i.e., subtract the amount from *both* the *availableBalance* and *totalBalance* attributes of the user’s *Account* object). The transaction then dispenses the desired amount of cash and instructs the user to take it. Finally, the main flow of activity merges with the cancellation flow of activity before reaching the final state.

We’ve taken the first steps in modeling the ATM software system’s behavior and have shown how an object’s attributes participate in performing the object’s activities. In Section 33.6, we investigate the behaviors for all classes to give a more accurate interpretation of the system behavior by filling in the third compartments of the classes in our class diagram.

Self-Review Exercises for Section 33.5

33.11 State whether the following statement is *true* or *false*, and if *false*, explain why: State diagrams model structural aspects of a system.

33.12 An activity diagram models the _____ that an object performs and the order in which it performs them.

- a) actions
- b) attributes
- c) states
- d) state transitions

33.13 Based on the requirements document, create an activity diagram for a deposit transaction.

33.6 Identifying Class Operations

In this section, we determine some of the class operations (or behaviors) needed to implement the ATM system. An operation is a service that objects of a class provide to clients (users) of the class. Consider the operations of some real-world objects. A radio’s operations include setting its station and volume (typically invoked by a person’s adjusting the

radio's controls). A car's operations include accelerating (invoked by the driver's pressing the accelerator pedal), decelerating (invoked by the driver's pressing the brake pedal or releasing the gas pedal), turning and shifting gears. Software objects can offer operations as well—for example, a software graphics object might offer operations for drawing a circle, drawing a line, drawing a square and the like. A spreadsheet software object might offer operations like printing the spreadsheet, totaling the elements in a row or column and graphing information in the spreadsheet as a bar chart or pie chart.

We can derive many of the class operations by examining the key *verbs and verb phrases* in the requirements document. We then relate these verbs and verb phrases to classes in our system (Fig. 33.16). The verb phrases in Fig. 33.16 help us determine the operations of each class.

Class	Verbs and verb phrases
ATM	executes financial transactions
BalanceInquiry	<i>[none in the requirements document]</i>
Withdrawal	<i>[none in the requirements document]</i>
Deposit	<i>[none in the requirements document]</i>
BankDatabase	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Account	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Screen	displays a message to the user
Keypad	receives numeric input from the user
CashDispenser	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
DepositSlot	receives a deposit envelope

Fig. 33.16 | Verbs and verb phrases for each class in the ATM system.

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 33.16. The “executes financial transactions” phrase associated with class ATM implies that class ATM instructs transactions to execute. Therefore, classes BalanceInquiry, Withdrawal and Deposit each need an operation to provide this service to the ATM. We place this operation (which we've named `execute`) in the third compartment of the three transaction classes in the updated class diagram of Fig. 33.17. During an ATM session, the ATM object will invoke these transaction operations as necessary.

The UML represents operations (that is, methods) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

```
operationName(parameter1, parameter2, ..., parameterN) : return type
```

Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:

```
parameterName : parameterType
```

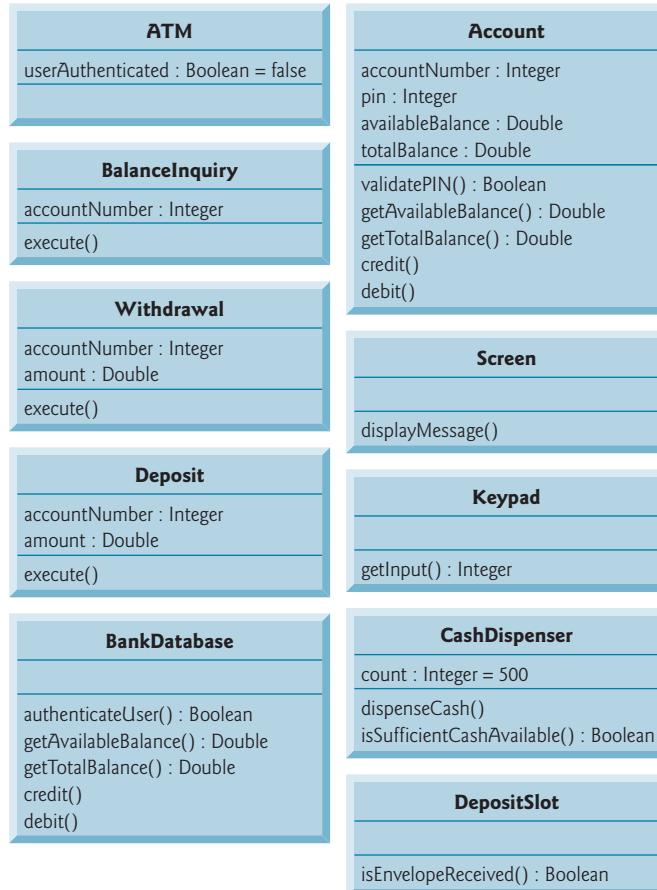


Fig. 33.17 | Classes in the ATM system with attributes and operations.

For the moment, we do not list the parameters of our operations—we'll identify and model some of them shortly. For some of the operations, we do not yet know the return types, so we also omit them from the diagram. These omissions are perfectly normal at this point. As our design and implementation proceed, we'll add the remaining return types.

Authenticating a User

Figure 33.16 lists the phrase “authenticates a user” next to class `BankDatabase`—the database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class `BankDatabase` needs an operation that provides an authentication service to the ATM. We place the operation `authenticateUser` in the third compartment of class `BankDatabase` (Fig. 33.17). However, an object of class `Account`, not class `BankDatabase`, stores the account number and PIN that must be accessed to authenticate a user, so class `Account` must provide a service to validate a PIN obtained through user input against a PIN stored in an `Account` object. Therefore, we add a `validatePIN` operation to

class Account. We specify a return type of Boolean for the authenticateUser and validatePIN operations. Each operation returns a value indicating either that the operation was successful in performing its task (i.e., a return value of true) or that it was not (i.e., a return value of false).

Other BankDatabase and Account Operations

Figure 33.16 lists several additional verb phrases for class BankDatabase: “retrieves an account balance,” “credits a deposit amount to an account” and “debits a withdrawal amount from an account.” Like “authenticates a user,” these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class Account actually perform the operations to which these phrases refer. Thus, we assign an operation to both class BankDatabase and class Account to correspond to each of these phrases. Recall from Section 33.3 that, because a bank account contains sensitive information, we do not allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we’ll see in Section 33.7, class ATM invokes the operations of class BankDatabase, each of which in turn invokes the operation with the same name in class Account.

Getting the Balances

The phrase “retrieves an account balance” suggests that classes BankDatabase and Account each need a getBalance operation. However, recall that we created *two* attributes in class Account to represent a balance—availableBalance and totalBalance. A balance inquiry requires access to *both* balance attributes so that it can display them to the user, but a withdrawal needs to check *only* the value of availableBalance. To allow objects in the system to obtain each balance attribute individually, we add operations getAvailableBalance and getTotalBalance to the third compartment of classes BankDatabase and Account (Fig. 33.17). We specify a return type of Double for these operations because the balance attributes they retrieve are of type Double.

Crediting and Debiting an Account

The phrases “credits a deposit amount to an account” and “debits a withdrawal amount from an account” indicate that classes BankDatabase and Account must perform operations to update an account during a deposit and withdrawal, respectively. We therefore assign credit and debit operations to classes BankDatabase and Account. You may recall that crediting an account (as in a deposit) adds an amount only to the totalBalance attribute. Debiting an account (as in a withdrawal), on the other hand, subtracts the amount from *both* balance attributes. We hide these implementation details inside class Account. This is a good example of encapsulation and information hiding.

Deposit Confirmations Performed by Another Banking System

If this were a real ATM system, classes BankDatabase and Account would also provide a set of operations to allow another banking system to update a user’s account balance after either confirming or rejecting all or part of a deposit. Operation confirmDepositAmount, for example, would add an amount to the availableBalance attribute, thus making deposited funds available for withdrawal. Operation rejectDepositAmount would subtract an amount from the totalBalance attribute to indicate that a specified amount, which

had recently been deposited through the ATM and added to the `totalBalance`, was not found in the deposit envelope. The bank would invoke this operation after determining either that the user failed to include the correct amount of cash or that any checks did not clear (i.e., they “bounced”). While adding these operations would make our system more complete, we do *not* include them in our class diagrams or our implementation because they’re beyond the scope of the case study.

Displaying Messages

Class `Screen` “displays a message to the user” at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class `Screen` an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (`displayMessage`) in the third compartment of class `Screen` in our class diagram (Fig. 33.17). We do not worry about the parameter of this operation at this time—we model it later in this section.

Keyboard Input

From the phrase “receives numeric input from the user” listed by class `Keypad` in Fig. 33.16, we conclude that class `Keypad` should perform a `getInput` operation. Because the ATM’s keypad, unlike a computer keyboard, contains only the numbers 0–9, we specify that this operation returns an integer value. Recall from the requirements document that in different situations the user may be required to enter a different type of number (e.g., an account number, a PIN, the number of a menu option, a deposit amount as a number of cents). Class `Keypad` simply obtains a numeric value for a client of the class—it does not determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user entered an appropriate number in a given situation, then respond accordingly (i.e., display an error message via class `Screen`). [Note: When we implement the system, we simulate the ATM’s keypad with a computer keyboard, and for simplicity we assume that the user does not enter nonnumeric input using keys on the computer keyboard that do not appear on the ATM’s keypad.]

Dispensing Cash

Figure 33.16 lists “dispenses cash” for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 33.17. Class `CashDispenser` also “indicates whether it contains enough cash to satisfy a withdrawal request.” Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`.

Figure 33.16 also lists “receives a deposit envelope” for class `DepositSlot`. The deposit slot must indicate whether it received an envelope, so we place an operation `isEnvelopeReceived`, which returns a `Boolean` value, in the third compartment of class `DepositSlot`. [Note: A real hardware deposit slot would most likely send the ATM a signal to indicate that an envelope was received. We simulate this behavior, however, with an

operation in class `DepositSlot` that class `ATM` can invoke to find out whether the deposit slot received an envelope.]

Class ATM

We do not list any operations for class `ATM` at this time. We're not yet aware of any services that class `ATM` provides to other classes in the system. When we implement the system with Java code, however, operations of this class, and additional operations of the other classes in the system, may emerge.

Identifying and Modeling Operation Parameters for Class `BankDatabase`

So far, we've not been concerned with the *parameters* of our operations—we've attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires to perform its assigned task.

Consider `BankDatabase`'s `authenticateUser` operation. To authenticate a user, this operation must know the account number and PIN supplied by the user. So we specify that `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to an `Account` object's account number and PIN in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and class `Account`'s attribute names. We list these parameters in the class diagram in Fig. 33.18 that models only class `BankDatabase`. [Note: It's perfectly normal to model only one class. In this case, we're examining the parameters of this one class, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit these parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

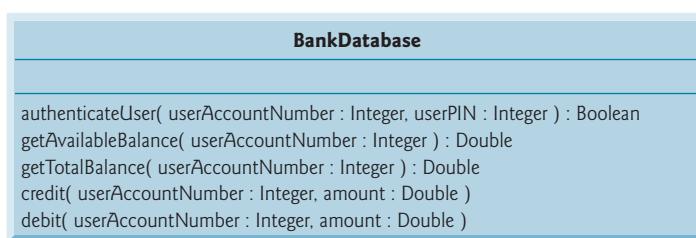


Fig. 33.18 | Class `BankDatabase` with operation parameters.

Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). Figure 33.18 thus specifies that operation `authenticateUser` takes two parameters—`userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in Java, we'll represent these parameters with `int` values.

Class `BankDatabase` operations `getAvailableBalance`, `getTotalBalance`, `credit` and `debit` also each require a `userAccountNumber` parameter to identify the account to which the database must apply the operations, so we include these parameters in the class diagram of Fig. 33.18. In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

Identifying and Modeling Operation Parameters for Class Account

Figure 33.19 models class Account's operation parameters. Operation validatePIN requires only a userPIN parameter, which contains the user-specified PIN to be compared with the account's PIN. Like their BankDatabase counterparts, operations credit and debit in class Account each require a Double parameter amount that indicates the amount of money involved in the operation. Operations getAvailableBalance and getTotalBalance in class Account require no additional data to perform their tasks. Class Account's operations do *not* require an account-number parameter to distinguish between Accounts, because these operations can be invoked only on a specific Account object.

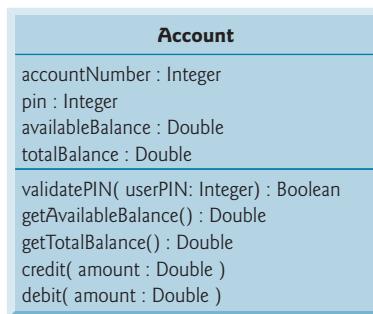


Fig. 33.19 | Class Account with operation parameters.

Identifying and Modeling Operation Parameters for Class Screen

Figure 33.20 models class Screen with a parameter specified for operation displayMessage. This operation requires only a String parameter message that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the String type listed in Fig. 33.20 refers to the UML type. When we implement the system in Java, we'll use the Java class `String` to represent this parameter.

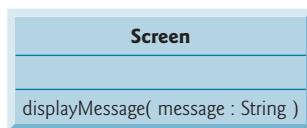


Fig. 33.20 | Class Screen with operation parameters.

Identifying and Modeling Operation Parameters for Class CashDispenser

Figure 33.21 specifies that operation dispenseCash of class CashDispenser takes a Double parameter amount to indicate the amount of cash (in dollars) to be dispensed. Operation isSufficientCashAvailable also takes a Double parameter amount to indicate the amount of cash in question.

Identifying and Modeling Operation Parameters for Other Classes

We do not discuss parameters for operation execute of classes BalanceInquiry, Withdrawal and Deposit, operation getInput of class Keypad and operation isEnvelope-

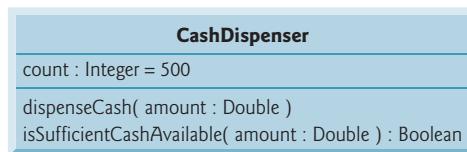


Fig. 33.21 | Class **CashDispenser** with operation parameters.

Received of class **DepositSlot**. At this point in our design process, we cannot determine whether these operations require additional data, so we leave their parameter lists empty. Later, we may decide to add parameters.

In this section, we've determined many of the operations performed by the classes in the ATM system. We've identified the parameters and return types of some of the operations. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary. We also might determine that some of our class operations need additional parameters and different return types, or that some parameters are unnecessary or require different types.

Self-Review Exercises for Section 33.6

33.14 Which of the following is *not* a behavior?

- a) reading data from a file
- b) printing output
- c) text output
- d) obtaining input from the user

33.15 If you were to add to the ATM system an operation that returns the `amount` attribute of class **Withdrawal**, how and where would you specify this operation in the class diagram of Fig. 33.17?

33.16 Describe the meaning of the following operation listing that might appear in a class diagram for an object-oriented design of a calculator:

`add(x : Integer, y : Integer) : Integer`

33.7 Indicating Collaboration Among Objects

In this section, we concentrate on the collaborations (interactions) among objects. When two objects communicate with each other to accomplish a task, they're said to **collaborate**—objects do this by invoking one another's operations. A **collaboration** consists of an object of one class sending a **message** to an object of another class. Messages are sent in Java via method calls.

In Section 33.6, we determined many of the operations of the system's classes. Now, we concentrate on the messages that invoke these operations. To identify the collaborations in the system, we return to the requirements document in Section 33.2. Recall that this document specifies the range of activities that occur during an ATM session (e.g., authenticating a user, performing transactions). The steps used to describe how the system must perform each of these tasks are our first indication of the collaborations in our system. As we proceed through this section and Chapter 34, we may discover additional collaborations.

Identifying the Collaborations in a System

We identify the collaborations in the system by carefully reading the sections of the requirements document that specify what the ATM should do to authenticate a user and to perform each transaction type. For each action or step described, we decide which objects in our system must interact to achieve the desired result. We identify one object as the sending object and another as the receiving object. We then select one of the receiving object's operations (identified in Section 33.6) that must be invoked by the sending object to produce the proper behavior. For example, the ATM displays a welcome message when idle. We know that an object of class `Screen` displays a message to the user via its `displayMessage` operation. Thus, we decide that the system can display a welcome message by employing a collaboration between the `ATM` and the `Screen` in which the `ATM` sends a `displayMessage` message to the `Screen` by invoking the `displayMessage` operation of class `Screen`. [Note: To avoid repeating the phrase “an object of class...,” we refer to an object by using its class name preceded by an article (e.g., “a,” “an” or “the”)—for example, “the `ATM`” refers to an object of class `ATM`.]

Figure 33.22 lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they first occur during an ATM session (i.e., the order in which they're discussed in the requirements document). We list each collaboration involving a unique sender, message and recipient only once, even though the collaborations may occur at several different times throughout an ATM session. For example, the first row in Fig. 33.22 indicates that the `ATM` collaborates with the `Screen` whenever the `ATM` needs to display a message to the user.

Let's consider the collaborations in Fig. 33.22. Before allowing a user to perform any transactions, the `ATM` must prompt the user to enter an account number, then to enter a PIN. It accomplishes these tasks by sending a `displayMessage` message to the `Screen`. Both actions refer to the same collaboration between the `ATM` and the `Screen`, which is already listed in Fig. 33.22. The `ATM` obtains input in response to a prompt by sending a `getInput` message to the `Keypad`. Next, the `ATM` must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an `authenticateUser` message to the `BankDatabase`. Recall that the `BankDatabase` cannot authenticate a user directly—only the user's `Account` (i.e., the `Account` that contains the account number specified by the user) can access the user's PIN on record to authenticate the user. Figure 33.22 therefore lists a collaboration in which the `BankDatabase` sends a `validatePIN` message to an `Account`.

After the user is authenticated, the `ATM` displays the main menu by sending a series of `displayMessage` messages to the `Screen` and obtains input containing a menu selection by sending a `getInput` message to the `Keypad`. We've already accounted for these collaborations, so we do not add anything to Fig. 33.22. After the user chooses a type of transaction to perform, the `ATM` executes the transaction by sending an `execute` message to an object of the appropriate transaction class (i.e., a `BalanceInquiry`, a `Withdrawal` or a `Deposit`). For example, if the user chooses to perform a balance inquiry, the `ATM` sends an `execute` message to a `BalanceInquiry`.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A `BalanceInquiry` retrieves the amount of money available in the user's account by sending a `getAvailableBalance` message to the `BankDatabase`, which responds by sending a `getAvailableBalance` message to the user's

An object of class...	sends the message...	to an object of class...
ATM	displayMessage getInput authenticateUser execute execute execute	Screen Keypad BankDatabase BalanceInquiry Withdrawal Deposit
BalanceInquiry	getAvailableBalance getTotalBalance displayMessage	BankDatabase BankDatabase Screen
Withdrawal	displayMessage getInput getAvailableBalance isSufficientCashAvailable debit dispenseCash	Screen Keypad BankDatabase CashDispenser BankDatabase CashDispenser
Deposit	displayMessage getInput isEnvelopeReceived credit	Screen Keypad DepositSlot BankDatabase
BankDatabase	validatePIN getAvailableBalance getTotalBalance debit credit	Account Account Account Account Account

Fig. 33.22 | Collaborations in the ATM system.

Account. Similarly, the BalanceInquiry retrieves the amount of money on deposit by sending a getTotalBalance message to the BankDatabase, which sends the same message to the user's Account. To display both parts of the user's account balance at the same time, the BalanceInquiry sends a displayMessage message to the Screen.

A Withdrawal responds to an execute message by sending displayMessage messages to the Screen to display a menu of standard withdrawal amounts (i.e., \$20, \$40, \$60, \$100, \$200). The Withdrawal sends a getInput message to the Keypad to obtain the user's selection. Next, the Withdrawal determines whether the requested amount is less than or equal to the user's account balance. The Withdrawal can obtain the amount of money available by sending a getAvailableBalance message to the BankDatabase. The Withdrawal then tests whether the cash dispenser contains enough cash by sending an isSufficientCashAvailable message to the CashDispenser. A Withdrawal sends a debit message to the BankDatabase to decrease the user's account balance. The BankDatabase in turn sends the same message to the appropriate Account, which decreases both the totalBalance and the availableBalance. To dispense the requested amount of cash, the Withdrawal sends a dispenseCash message to the CashDispenser. Finally, the Withdrawal sends a displayMessage message to the Screen, instructing the user to take the cash.

A Deposit responds to an execute message first by sending a displayMessage message to the Screen to prompt the user for a deposit amount. The Deposit sends a getInput message to the Keypad to obtain the user's input. The Deposit then sends a displayMessage message to the Screen to tell the user to insert a deposit envelope. To determine whether the deposit slot received an incoming deposit envelope, the Deposit sends an isEnvelopeReceived message to the DepositSlot. The Deposit updates the

user's account by sending a `credit` message to the `BankDatabase`, which subsequently sends a `credit` message to the user's `Account`. Recall that crediting funds to an `Account` increases the `totalBalance` but not the `availableBalance`.

Interaction Diagrams

Now that we've identified possible collaborations between our ATM system's objects, let's graphically model these interactions using the UML. The UML provides several types of **interaction diagrams** that model the behavior of a system by modeling how objects interact. The **communication diagram** emphasizes *which objects* participate in collaborations. Like the communication diagram, the **sequence diagram** shows collaborations among objects, but it emphasizes *when* messages are sent between objects *over time*.

Communication Diagrams

Figure 33.23 shows a communication diagram that models the ATM executing a `BalanceInquiry`. Objects are modeled in the UML as rectangles containing names in the form `objectName : ClassName`. In this example, which involves only one object of each type, we disregard the object name and list only a colon followed by the class name. [Note: Specifying each object's name in a communication diagram is recommended when modeling multiple objects of the same type.] Communicating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of an operation (i.e., a method in Java) belonging to the receiving object—think of the name as a “service” that the receiving object provides to sending objects (its clients).



Fig. 33.23 | Communication diagram of the ATM executing a balance inquiry.

The solid filled arrow represents a message—or **synchronous call**—in the UML and a method call in Java. This arrow indicates that the flow of control is from the sending object (the ATM) to the receiving object (a `BalanceInquiry`). Since this is a synchronous call, the sending object can't send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object. The sender just waits. In Fig. 33.23, the ATM calls `BalanceInquiry` method `execute` and can't send another message until `execute` has finished and returns control to the ATM. [Note: If this were an **asynchronous call**, represented by a stick (\Rightarrow) arrowhead, the sending object would not have to wait for the receiving object to return control—it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls are implemented in Java using a technique called multithreading, which is discussed in Chapter 23.]

Sequence of Messages in a Communication Diagram

Figure 33.24 shows a communication diagram that models the interactions among system objects when an object of class `BalanceInquiry` executes. We assume that the object's `accountNumber` attribute contains the account number of the current user. The collaborations in Fig. 33.24 begin after the ATM sends an `execute` message to a `BalanceInquiry`

(i.e., the interaction modeled in Fig. 33.23). The number to the left of a message name indicates the order in which the message is passed. The **sequence of messages** in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message 1 and ends with message 3. The **BalanceInquiry** first sends a `getAvailableBalance` message to the **BankDatabase** (message 1), then sends a `getTotalBalance` message to the **BankDatabase** (message 2). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a Java method call)—the **BalanceInquiry** passes attribute `accountNumber` with its messages to the **BankDatabase** to indicate which **Account**'s balance information to retrieve. Recall from Fig. 33.18 that operations `getAvailableBalance` and `getTotalBalance` of class **BankDatabase** each require a parameter to identify an account. The **BalanceInquiry** next displays the `availableBalance` and the `totalBalance` to the user by passing a `displayMessage` message to the **Screen** (message 3) that includes a parameter indicating the message to be displayed.

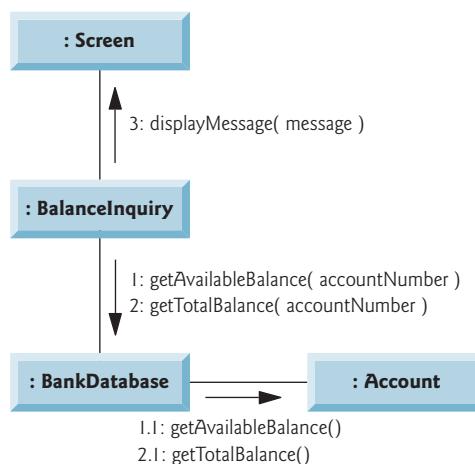


Fig. 33.24 | Communication diagram for executing a balance inquiry.

Figure 33.24 models two additional messages passing from the **BankDatabase** to an **Account** (message 1.1 and message 2.1). To provide the ATM with the *two* balances of the user's **Account** (as requested by messages 1 and 2), the **BankDatabase** must pass a `getAvailableBalance` and a `getTotalBalance` message to the user's **Account**. Such messages passed within the handling of another message are called **nested messages**. The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message 1.1 is the first message nested in message 1—the **BankDatabase** passes a `getAvailableBalance` message during **BankDatabase**'s processing of a message by the same name. [Note: If the **BankDatabase** needed to pass a second nested message while processing message 1, the second message would be numbered 1.2.] A message may be passed only when *all* the nested messages from the previous message have been passed. For example, the **BalanceInquiry** passes message 3 only after messages 2 and 2.1 have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in Fig. 33.24 using a flat numbering scheme (i.e., 1, 2, 3, 4, 5), someone looking at the diagram might not be able to determine that `BankDatabase` passes the `getAvailableBalance` message (message 1.1) to an `Account` *during* the `BankDatabase`'s processing of message 1, as opposed to *after* completing the processing of message 1. The nested decimal numbers make it clear that the second `getAvailableBalance` message (message 1.1) is passed to an `Account` within the handling of the first `getAvailableBalance` message (message 1) by the `BankDatabase`.

Sequence Diagrams

Communication diagrams emphasize the participants in collaborations, but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. Figure 33.25 shows a sequence diagram modeling the sequence of interactions that occur when a `Withdrawal` executes. The dotted line extending down from an object's rectangle is that object's **lifeline**, which represents the progression of time. Actions occur along an object's lifeline in chronological order from top to bottom—an action near the top happens before one near the bottom.

Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object's lifeline. An **activation**, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return message, represented as a dashed line with a stick (\Rightarrow) arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 33.25 begins when a `Withdrawal` prompts the user to choose a withdrawal amount by sending a `displayMessage` message to the `Screen`. The `Withdrawal` then sends a `getInput` message to the `Keypad`, which obtains input from the user. We've already modeled the control logic involved in a `Withdrawal` in the activity diagram of Fig. 33.15, so we do not show this logic in the sequence diagram of Fig. 33.25. Instead, we model the best-case scenario in which the balance of the user's account is greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a sufficient amount of cash to satisfy the request. You can model control logic in a sequence diagram with UML frames (which are not covered in this case study). For a quick overview of UML frames, visit www.agilemodeling.com/style/frame.htm.

After obtaining a withdrawal amount, the `Withdrawal` sends a `getAvailableBalance` message to the `BankDatabase`, which in turn sends a `getAvailableBalance` message to the user's `Account`. Assuming that the user's account has enough money available to permit the transaction, the `Withdrawal` next sends an `isSufficientCashAvailable` message to the `CashDispenser`. Assuming that there's enough cash available, the `Withdrawal` decreases the balance of the user's account (i.e., both the `totalBalance` and the `availableBalance`) by sending a `debit` message to the `BankDatabase`. The `BankDatabase` responds by sending a `debit` message to the user's `Account`. Finally, the `Withdrawal` sends

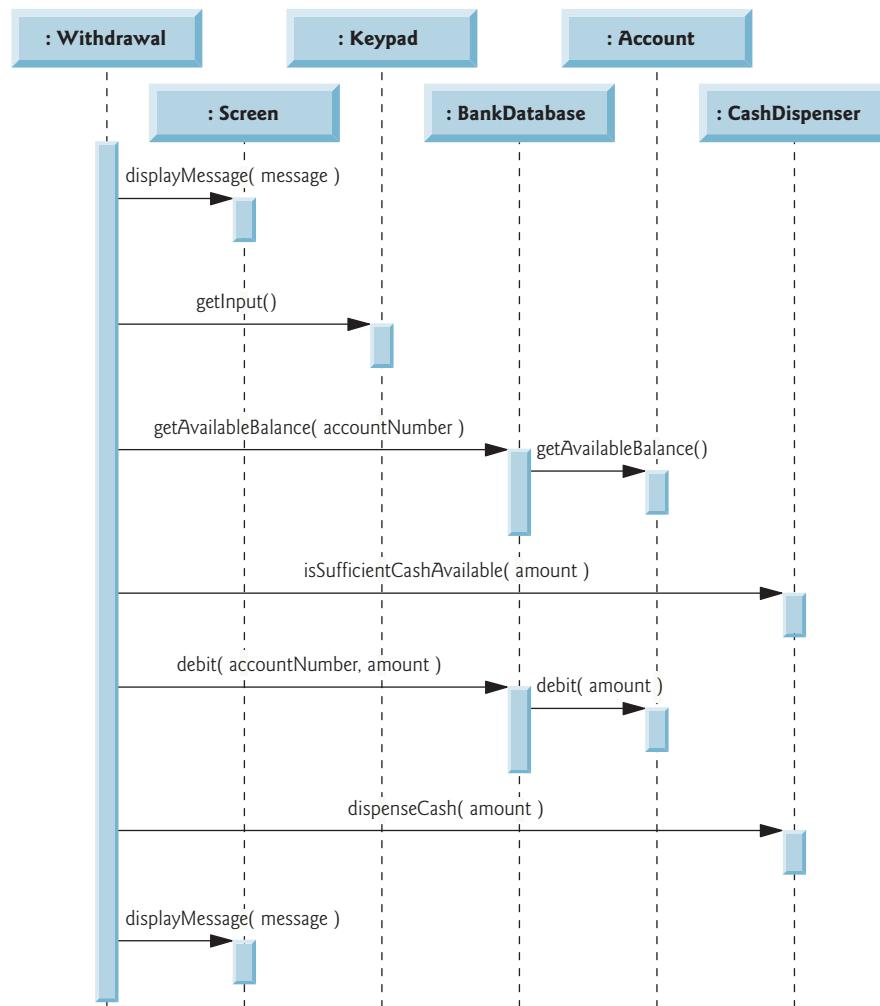


Fig. 33.25 | Sequence diagram that models a `Withdrawal` executing.

a `dispenseCash` message to the `CashDispenser` and a `displayMessage` message to the `Screen`, telling the user to remove the cash from the machine.

We've identified the collaborations among objects in the ATM system and modeled some of them using UML interaction diagrams—both communication diagrams and sequence diagrams. In Section 34.2, we enhance the structure of our model to complete a preliminary object-oriented design, then we begin implementing the ATM system in Java.

Self-Review Exercises for Section 33.7

- 33.17** A(n) _____ consists of an object of one class sending a message to an object of another class.
- association
 - aggregation
 - collaboration
 - composition

33.18 Which form of interaction diagram emphasizes *what* collaborations occur? Which form emphasizes *when* collaborations occur?

33.19 Create a sequence diagram that models the interactions among objects in the ATM system that occur when a Deposit executes successfully, and explain the sequence of messages modeled by the diagram.

33.8 Wrap-Up

In this chapter, you learned how to work from a detailed requirements document to develop an object-oriented design. You worked with six popular types of UML diagrams to graphically model an object-oriented automated teller machine software system. In Chapter 34, we tune the design using inheritance, then completely implement the design as a Java application.

Answers to Self-Review Exercises

33.1 Figure 33.26 contains a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

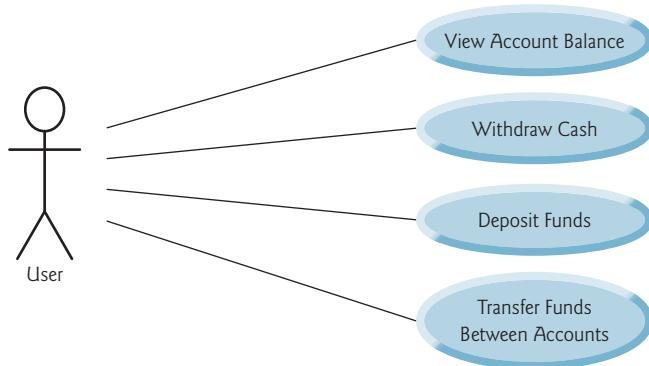


Fig. 33.26 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

33.2 b.

33.3 d.

33.4 [Note: Answers may vary.] Figure 33.27 presents a class diagram that shows some of the composition relationships of a class Car.

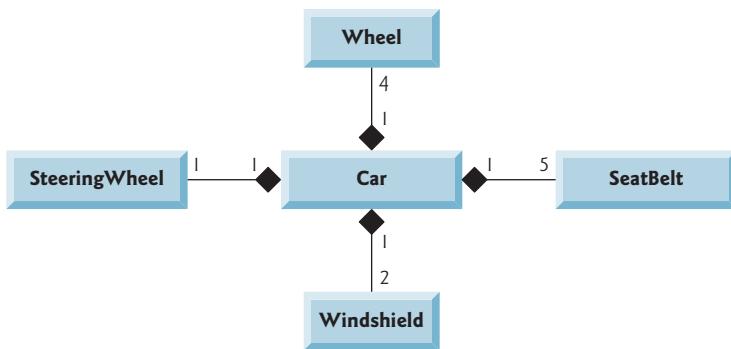


Fig. 33.27 | Class diagram showing composition relationships of a class Car.

33.5 c. [Note: In a computer network, this relationship could be many-to-many.]

33.6 True.

33.7 Figure 33.28 presents a class diagram for the ATM including class Deposit instead of class Withdrawal (as in Fig. 33.10). Deposit does not access CashDispenser, but does access DepositSlot.

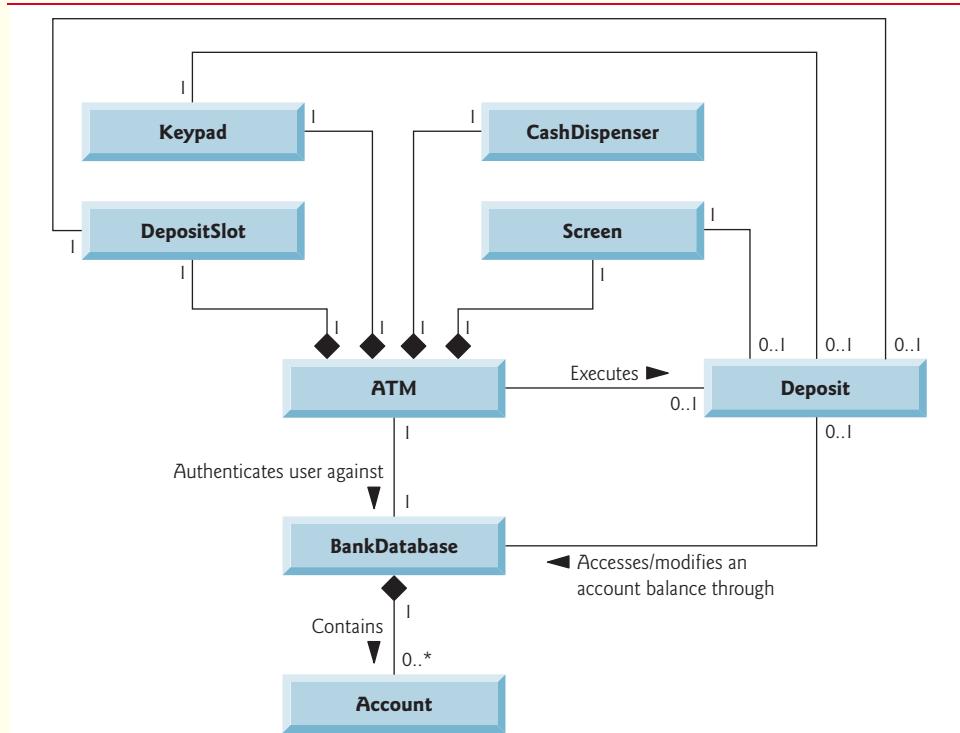


Fig. 33.28 | Class diagram for the ATM system model including class Deposit.

33.8 b.

33.9 c. Fly is an operation or behavior of an airplane, not an attribute.

33.10 This indicates that count is an Integer with an initial value of 500. This attribute keeps track of the number of bills available in the CashDispenser at any given time.

33.11 False. State diagrams model some of the behavior of a system.

33.12 a.

33.13 Figure 33.29 models the actions that occur after the user chooses the deposit option from the main menu and before the ATM returns the user to the main menu. Recall that part of receiving a deposit amount from the user involves converting an integer number of cents to a dollar amount. Also recall that crediting a deposit amount to an account increases only the totalBalance attribute of the user's Account object. The bank updates the availableBalance attribute of the user's Account object only after confirming the amount of cash in the deposit envelope and after the enclosed checks clear—this occurs independently of the ATM system.

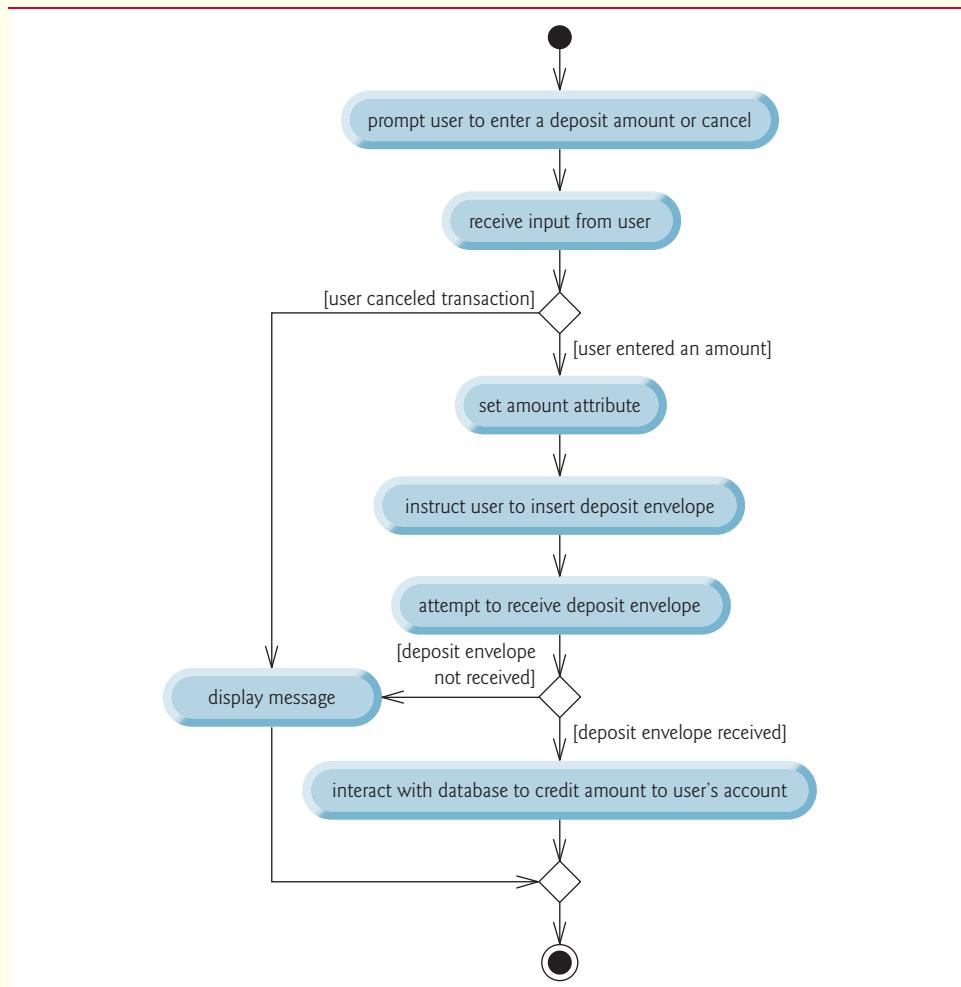


Fig. 33.29 | Activity diagram for a deposit transaction.

33.14 c.

33.15 To specify an operation that retrieves the *amount* attribute of class *Withdrawal*, the following operation listing would be placed in the operation (i.e., third) compartment of class *Withdrawal*:

```
getAmount() : Double
```

33.16 This operation listing indicates an operation named *add* that takes integers *x* and *y* as parameters and returns an integer value.

33.17 c.

33.18 Communication diagrams emphasize *what* collaborations occur. Sequence diagrams emphasize *when* collaborations occur.

33.19 Figure 33.30 presents a sequence diagram that models the interactions between objects in the ATM system that occur when a *Deposit* executes successfully. A *Deposit* first sends a *displayMessage* message to the *Screen* to ask the user to enter a deposit amount. Next the *Deposit* sends a *getInput* message to the *Keypad* to receive input from the user. The *Deposit* then instructs the user to enter a deposit envelope by sending a *displayMessage* message to the *Screen*. The *Deposit* next sends an *isEnvelopeReceived* message to the *DepositSlot* to confirm that the deposit envelope has been received by the ATM. Finally, the *Deposit* increases the *totalBalance* attribute (but not the *availableBalance* attribute) of the user's *Account* by sending a *credit* message to the *BankDatabase*. The *BankDatabase* responds by sending the same message to the user's *Account*.

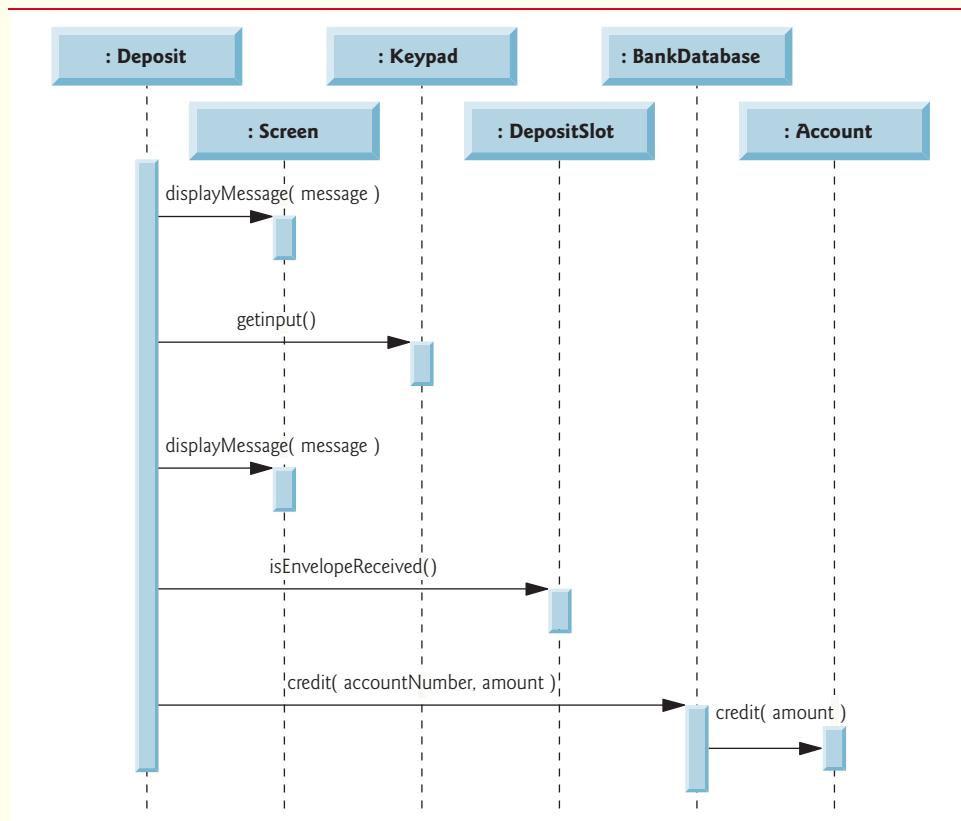


Fig. 33.30 | Sequence diagram that models a *Deposit* executing.

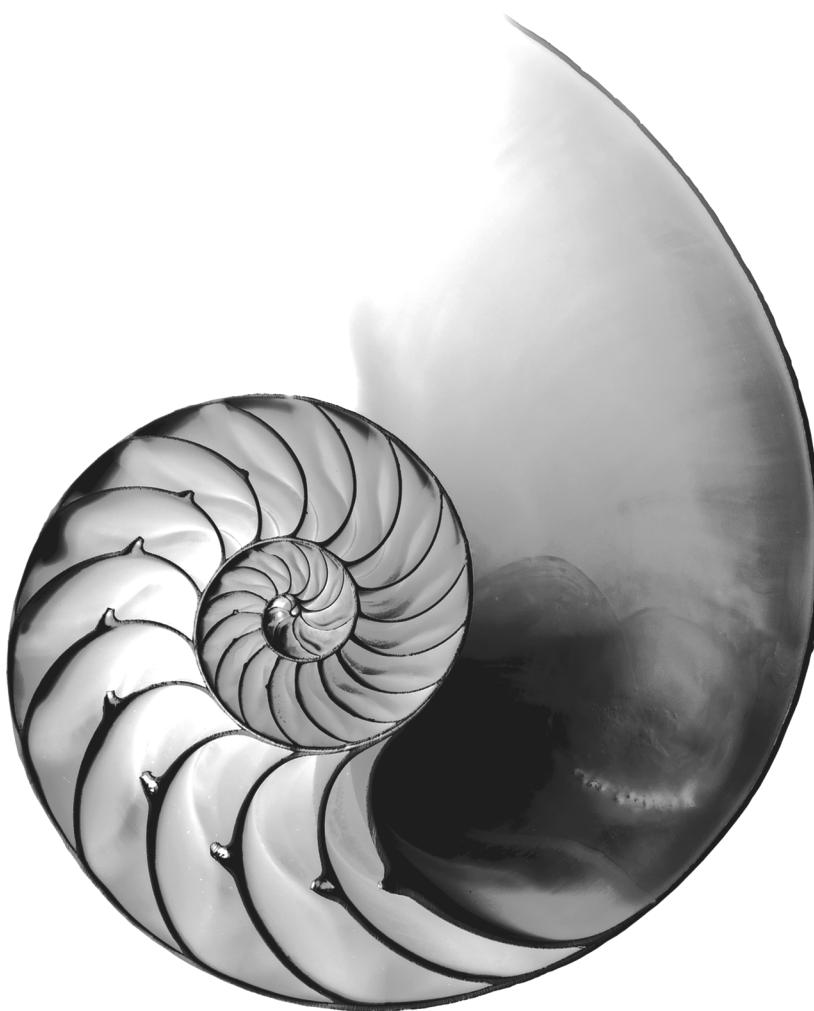
ATM Case Study Part 2: Implementing the Design

34

Objectives

In this chapter you'll:

- Incorporate inheritance into the design of the ATM.
- Incorporate polymorphism into the design of the ATM.
- Fully implement in Java the UML-based object-oriented design of the ATM software.
- Study a detailed code walkthrough of the ATM software system that explains the implementation issues.



Outline

34.1	Introduction	34.4.4 Class CashDispenser
34.2	Starting to Program the Classes of the ATM System	34.4.5 Class DepositSlot
34.3	Incorporating Inheritance and Polymorphism into the ATM System	34.4.6 Class Account
34.4	ATM Case Study Implementation	34.4.7 Class BankDatabase
34.4.1	Class ATM	34.4.8 Class Transaction
34.4.2	Class Screen	34.4.9 Class BalanceInquiry
34.4.3	Class Keypad	34.4.10 Class Withdrawal
		34.4.11 Class Deposit
		34.4.12 Class ATMCaseStudy
		34.5 Wrap-Up

Answers to Self-Review Exercises

34.1 Introduction

In Chapter 33, we developed an object-oriented design for our ATM system. We now implement our object-oriented design in Java. In Section 34.2, we show how to convert class diagrams to Java code. In Section 34.3, we tune the design with inheritance and polymorphism. Then we present a full Java code implementation of the ATM software in Section 34.4. The code is carefully commented and the discussions of the implementation are thorough and precise. Studying this application provides the opportunity for you to see a more substantial application of the kind you’re likely to encounter in industry.

34.2 Starting to Program the Classes of the ATM System

[*Note:* This section may be read after Chapter 8.]

Visibility

We now apply access modifiers to the members of our classes. We’ve introduced access modifiers `public` and `private`. Access modifiers determine the **visibility** or accessibility of an object’s attributes and methods to other objects. Before we can begin implementing our design, we must consider which attributes and methods of our classes should be `public` and which should be `private`.

We’ve observed that attributes normally should be `private` and that methods invoked by clients of a given class should be `public`. Methods that are called as “utility methods” only by other methods of the same class normally should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute, whereas a minus sign (-) indicates private visibility. Figure 34.1 shows our updated class diagram with visibility markers included. [*Note:* We do not include any operation parameters in Fig. 34.1—this is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 33.17–33.21.]

Navigability

Before we begin implementing our design in Java, we introduce an additional UML notation. The class diagram in Fig. 34.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. **Navigability arrows**



Fig. 34.1 | Class diagram with visibility markers.

(represented as arrows with stick (\Rightarrow) arrowheads in the class diagram) indicate the direction in which an association can be traversed. When implementing a system designed using the UML, you use navigability arrows to determine which objects need references to other objects. For example, the navigability arrow pointing from class **ATM** to class **BankDatabase** indicates that we can navigate from the former to the latter, thereby enabling the **ATM** to invoke the **BankDatabase**'s operations. However, since Fig. 34.2 does *not* contain a navigability arrow pointing from class **BankDatabase** to class **ATM**, the **BankDatabase** cannot access the **ATM**'s operations. Associations in a class diagram that have navigability arrows at both ends or have none at all indicate **bidirectional navigability**—navigation can proceed in either direction across the association.

Like the class diagram of Fig. 33.10, that of Fig. 34.2 omits classes **BalanceInquiry** and **Deposit** for simplicity. The navigability of the associations in which these classes participate closely parallels that of class **Withdrawal**. Recall from Section 33.3 that **BalanceInquiry** has an association with class **Screen**. We can navigate from class **BalanceInquiry** to class **Screen** along this association, but we cannot navigate from class **Screen** to class

`BalanceInquiry`. Thus, if we were to model class `BalanceInquiry` in Fig. 34.2, we would place a navigability arrow at class `Screen`'s end of this association. Also recall that class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. We can navigate from class `Deposit` to each of these classes, but *not* vice versa. We therefore would place navigability arrows at the `Screen`, `Keypad` and `DepositSlot` ends of these associations. [Note: We model these additional classes and associations in our final class diagram in Section 34.3, after we've simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

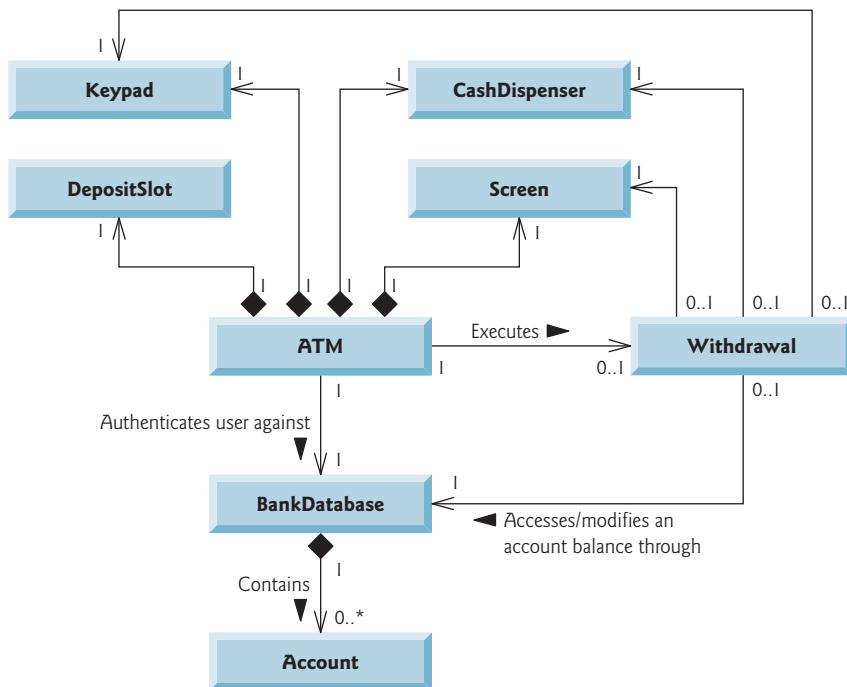


Fig. 34.2 | Class diagram with navigability arrows.

Implementing the ATM System from Its UML Design

We're now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 34.1 and Fig. 34.2 into Java code. The code will represent the “skeleton” of the system. In Section 34.3, we modify the code to incorporate inheritance. In Section 34.4, we present the complete working Java code for our model.

As an example, we develop the code from our design of class `Withdrawal` in Fig. 34.1. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 34.2 to determine the associations among classes. We follow the following four guidelines for each class:

1. Use the name located in the first compartment to declare the class as a `public` class with an empty no-argument constructor. We include this constructor simply as a placeholder to remind us that *most classes will indeed need custom constructors*.

ters. In Section 34.4, when we complete a working version of this class, we'll add arguments and code the body of the constructor as needed. For example, class `Withdrawal` yields the code in Fig. 34.3. If we find that the class's instance variables require only default initialization, then we'll remove the empty no-argument constructor because it's unnecessary.

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal {
3     // no-argument constructor
4     public Withdrawal() { }
5 }
```

Fig. 34.3 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

2. Use the attributes located in the second compartment to declare the instance variables. For example, the `private` attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 34.4. [Note: The constructor of the complete working version of this class will assign values to these attributes.]

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal {
3     // attributes
4     private int accountNumber; // account to withdraw funds from
5     private double amount; // amount to withdraw
6
7     // no-argument constructor
8     public Withdrawal() { }
9 }
```

Fig. 34.4 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

3. Use the associations described in the class diagram to declare the references to other objects. For example, according to Fig. 34.2, `Withdrawal` can access one object of class `Screen`, one object of class `Keypad`, one object of class `CashDispenser` and one object of class `BankDatabase`. This yields the code in Fig. 34.5. [Note: The constructor of the complete working version of this class will initialize these instance variables with references to actual objects.]

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal {
3     // attributes
4     private int accountNumber; // account to withdraw funds from
5     private double amount; // amount to withdraw
6
7     // references to associated objects
8     private Screen screen; // ATM's screen
9     private Keypad keypad; // ATM's keypad
```

Fig. 34.5 | Java code for class `Withdrawal` based on Figs. 34.1–34.2. (Part 1 of 2.)

```
10    private CashDispenser cashDispenser; // ATM's cash dispenser
11    private BankDatabase bankDatabase; // account info database
12
13    // no-argument constructor
14    public Withdrawal() { }
15 }
```

Fig. 34.5 | Java code for class `Withdrawal` based on Figs. 34.1–34.2. (Part 2 of 2.)

4. Use the operations located in the third compartment of Fig. 34.1 to declare the shells of the methods. If we have not yet specified a return type for an operation, we declare the method with return type `void`. Refer to the class diagrams of Figs. 33.17–33.21 to declare any necessary parameters. For example, adding the `public` operation `execute` in class `Withdrawal`, which has an empty parameter list, yields the code in Fig. 34.6. [Note: We code the bodies of methods when we implement the complete system in Section 34.4.]

This concludes our discussion of the basics of generating classes from UML diagrams.

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal {
3     // attributes
4     private int accountNumber; // account to withdraw funds from
5     private double amount; // amount to withdraw
6
7     // references to associated objects
8     private Screen screen; // ATM's screen
9     private Keypad keypad; // ATM's keypad
10    private CashDispenser cashDispenser; // ATM's cash dispenser
11    private BankDatabase bankDatabase; // account info database
12
13    // no-argument constructor
14    public Withdrawal() { }
15
16    // operations
17    public void execute() { }
18 }
```

Fig. 34.6 | Java code for class `Withdrawal` based on Figs. 34.1–34.2.

Self-Review Exercises for Section 34.2

34.1 State whether the following statement is *true* or *false*, and if *false*, explain why: If an attribute of a class is marked with a minus sign (-) in a class diagram, the attribute is not directly accessible outside the class.

34.2 In Fig. 34.2, the association between the `ATM` and the `Screen` indicates that:

- we can navigate from the `Screen` to the `ATM`
- we can navigate from the `ATM` to the `Screen`
- Both (a) and (b); the association is bidirectional
- None of the above

34.3 Write Java code to begin implementing the design for class `Keypad`.

34.3 Incorporating Inheritance and Polymorphism into the ATM System

[Note: This section may be read after Chapter 10.]

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for *commonality among classes* in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 33.3, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform. Figure 34.7 shows the attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. These classes have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the ATM invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of transactions*. Figure 34.7 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing classes `BalanceInquiry`, `Withdrawal` and `Deposit`. We place the common functionality in a superclass, `Transaction`, that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend.

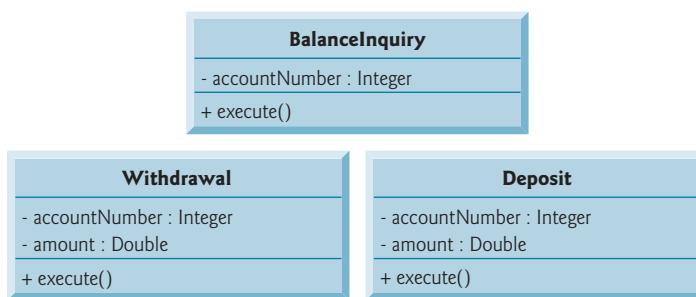


Fig. 34.7 | Attributes and operations of `BalanceInquiry`, `Withdrawal` and `Deposit`.

Generalization

The UML specifies a relationship called a **generalization** to model inheritance. Figure 34.8 is the class diagram that models the generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend class `Transaction`. Class `Transaction` is said to be a generalization of classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Class `BalanceInquiry`, `Withdrawal` and `Deposit` are said to be **specializations** of class `Transaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we *factor out* this *common attribute* and place it in superclass `Transaction`. We no longer list `accountNumber` in the second compartment of each subclass, because the

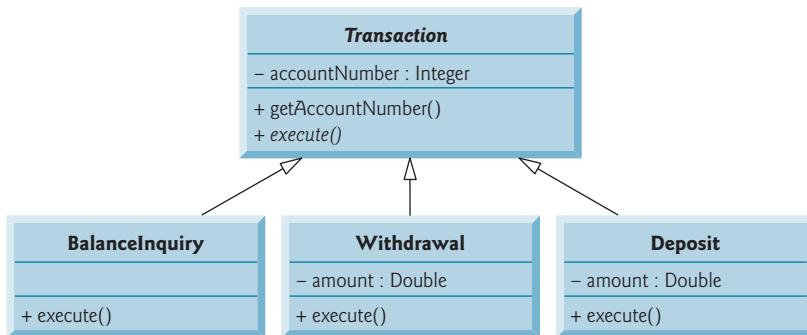


Fig. 34.8 | Class diagram modeling generalization of superclass **Transaction** and subclasses **BalanceInquiry**, **Withdrawal** and **Deposit**. Abstract class names (e.g., **Transaction**) and method names (e.g., **execute** in class **Transaction**) appear in italics.

three subclasses *inherit* this attribute from **Transaction**. Recall, however, that subclasses cannot directly access **private** attributes of a superclass. We therefore include **public** method `getAccountNumber` in class **Transaction**. Each subclass will inherit this method, enabling the subclass to access its `accountNumber` as needed to execute a transaction.

According to Fig. 34.7, classes **BalanceInquiry**, **Withdrawal** and **Deposit** also share operation `execute`, so we placed **public** method `execute` in superclass **Transaction**. However, it does *not* make sense to implement `execute` in class **Transaction**, because the functionality that this method provides *depends on the type of the actual transaction*. We therefore declare method `execute` as **abstract** in superclass **Transaction**. Any class that contains at least one abstract method must also be declared abstract. This forces any subclass of **Transaction** that must be a *concrete* class (i.e., **BalanceInquiry**, **Withdrawal** and **Deposit**) to implement method `execute`. The UML requires that we place abstract class names (and abstract methods) in italics, so **Transaction** and its method `execute` appear in italics in Fig. 34.8. Method `execute` is *not* italicized in subclasses **BalanceInquiry**, **Withdrawal** and **Deposit**. Each subclass overrides superclass **Transaction**'s `execute` method with a concrete implementation that performs the steps appropriate for completing that type of transaction. Figure 34.8 includes operation `execute` in the third compartment of classes **BalanceInquiry**, **Withdrawal** and **Deposit**, because each class has a different concrete implementation of the overridden method.

Processing Transactions Polymorphically

Polymorphism provides the ATM with an elegant way to execute all transactions “in the general.” For example, suppose a user chooses to perform a balance inquiry. The ATM sets a **Transaction** reference to a new **BalanceInquiry** object. When the ATM uses its **Transaction** reference to invoke method `execute`, **BalanceInquiry**'s version of `execute` is called.

This *polymorphic* approach also makes the system easily *extensible*. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional **Transaction** subclass that overrides the `execute` method with a version of the method appropriate for executing the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type from the main menu and for the ATM to instantiate and execute objects of the new subclass.

The ATM could execute transactions of the new type using the current code, because it executes all transactions *polymorphically* using a general `Transaction` reference.

Recall that an abstract class like `Transaction` is one for which you never intend to instantiate objects. An abstract class simply declares common attributes and behaviors of its subclasses in an inheritance hierarchy. Class `Transaction` defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include abstract method `execute` in class `Transaction` if it lacks a concrete implementation. Conceptually, we include it because it corresponds to the defining behavior of *all* transactions—executing. Technically, we must include method `execute` in superclass `Transaction` so that the ATM (or any other class) can polymorphically invoke each subclass's *overridden* version of this method through a `Transaction` reference. Also, from a software engineering perspective, including an abstract method in a superclass forces the implementor of the subclasses to override that method with concrete implementations in the subclasses, or else the subclasses, too, will be abstract, preventing objects of those subclasses from being instantiated.

Additional Attribute of Classes Withdrawal and Deposit

Subclasses `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `accountNumber` from superclass `Transaction`, but classes `Withdrawal` and `Deposit` contain the additional attribute `amount` that distinguishes them from class `BalanceInquiry`. Classes `Withdrawal` and `Deposit` require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class `BalanceInquiry` has no need for such an attribute and requires only an account number to execute. Even though two of the three `Transaction` subclasses share this attribute, we do *not* place it in superclass `Transaction`—we place only features *common* to all the subclasses in the superclass, otherwise subclasses could inherit attributes (and methods) that they do not need and should not have.

Class Diagram with Transaction Hierarchy Incorporated

Figure 34.9 presents an updated class diagram of our model that incorporates inheritance and introduces class `Transaction`. We model an association between class `ATM` and class `Transaction` to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `Transaction` exist in the system at a time). Because a `Withdrawal` is a type of `Transaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal`. Subclass `Withdrawal` inherits superclass `Transaction`'s association with class `ATM`. Subclasses `BalanceInquiry` and `Deposit` inherit this association, too, so the previously omitted associations between `ATM` and classes `BalanceInquiry` and `Deposit` no longer exist either.

We also add an association between class `Transaction` and the `BankDatabase` (Fig. 34.9). All `Transactions` require a reference to the `BankDatabase` so they can access and modify account information. Because each `Transaction` subclass inherits this reference, we no longer model the association between class `Withdrawal` and the `BankDatabase`. Similarly, the previously omitted associations between the `BankDatabase` and classes `BalanceInquiry` and `Deposit` no longer exist.

We show an association between class `Transaction` and the `Screen`. All `Transactions` display output to the user via the `Screen`. Thus, we no longer include the association previously modeled between `Withdrawal` and the `Screen`, although `Withdrawal` still participates in associations with the `CashDispenser` and the `Keypad`. Our class diagram incor-

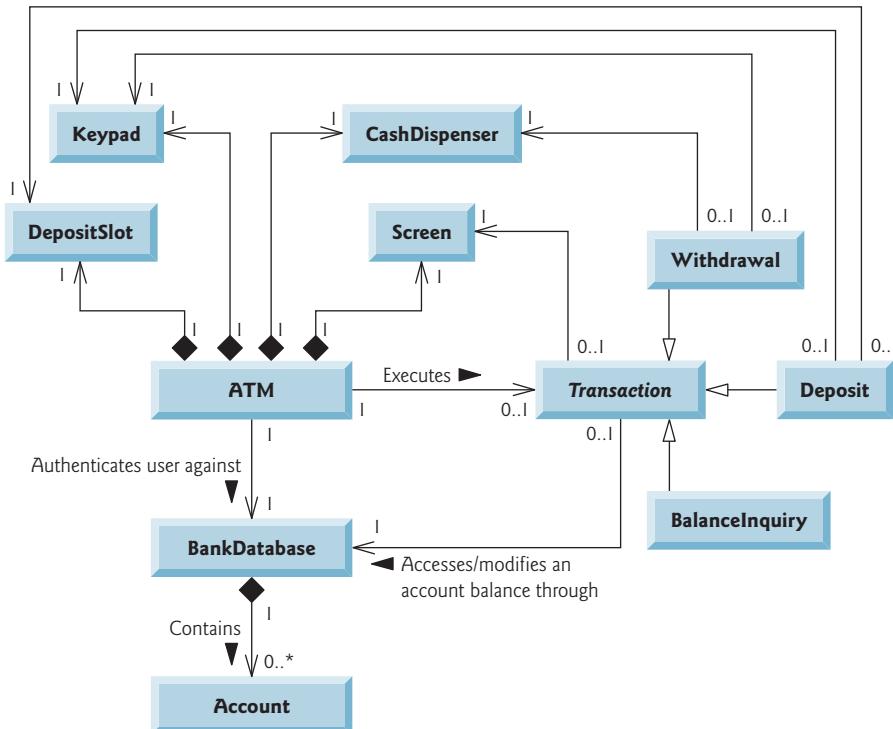


Fig. 34.9 | Class diagram of the ATM system (incorporating inheritance). The abstract class name **Transaction** appears in italics.

porating inheritance also models **Deposit** and **BalanceInquiry**. We show associations between **Deposit** and both the **DepositSlot** and the **Keypad**. Class **BalanceInquiry** takes part in no associations other than those inherited from class **Transaction**—a **BalanceInquiry** needs to interact only with the **BankDatabase** and with the **Screen**.

Figure 34.1 showed attributes and operations with visibility markers. Now in Fig. 34.10 we present a modified class diagram that incorporates inheritance. This abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we've employed inheritance in our system. To save space, as we did in Fig. 33.12, we do not include those attributes shown by associations in Fig. 34.9—we do, however, include them in the Java implementation in Section 34.4. We also omit all operation parameters, as we did in Fig. 34.1—incorporating inheritance does not affect the parameters already modeled in Figs. 33.17–33.21.



Software Engineering Observation 34.1

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, methods and associations is substantial (as in Figs. 34.9 and 34.10), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and methods.



Fig. 34.10 | Class diagram with attributes and operations (incorporating inheritance). The abstract class name **Transaction** and the abstract method name **execute** in class **Transaction** appear in italics.

Implementing the ATM System Design (Incorporating Inheritance)

In Section 34.2, we began implementing the ATM system design in Java code. We now incorporate inheritance, using class **Withdrawal** as an example.

1. If a class A is a generalization of class B, then class B extends class A in the class declaration. For example, abstract superclass **Transaction** is a generalization of class **Withdrawal**. Figure 34.11 shows the declaration of class **Withdrawal**.

```

1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal extends Transaction {
3 }
```

Fig. 34.11 | Java code for shell of class **Withdrawal**.

2. If class A is an abstract class and class B is a subclass of class A, then class B must implement the *abstract* methods of class A if class B is to be a *concrete* class. For example, class *Transaction* contains abstract method *execute*, so class *Withdrawal* must implement this method if we want to instantiate a *Withdrawal* object. Figure 34.12 is the Java code for class *Withdrawal* from Fig. 34.9 and Fig. 34.10. Class *Withdrawal* inherits field *accountNumber* from superclass *Transaction*, so *Withdrawal* does not need to declare this field. Class *Withdrawal* also inherits references to the *Screen* and the *BankDatabase* from its superclass *Transaction*, so we do not include these references in our code. Figure 34.10 specifies attribute *amount* and operation *execute* for class *Withdrawal*. Line 5 of Fig. 34.12 declares a field for attribute *amount*. Lines 13–14 declare the shell of a method for operation *execute*. Recall that subclass *Withdrawal* must provide a concrete implementation of the abstract method *execute* in superclass *Transaction*. The *keypad* and *cashDispenser* references (lines 6–7) are fields derived from *Withdrawal*'s associations in Fig. 34.9. The constructor in the complete working version of this class will initialize these references to actual objects.



Software Engineering Observation 34.2

Several UML modeling tools can convert UML-based designs into Java code, speeding the implementation process considerably. For more information on these tools, visit our UML Resource Center at www.deitel.com/UML/.

```

1 // Withdrawal.java
2 // Generated using the class diagrams in Fig. 34.9 and Fig. 34.10
3 public class Withdrawal extends Transaction {
4     // attributes
5     private double amount; // amount to withdraw
6     private Keypad keypad; // reference to keypad
7     private CashDispenser cashDispenser; // reference to cash dispenser
8
9     // no-argument constructor
10    public Withdrawal() { }
11
12    // method overriding execute
13    @Override
14    public void execute() { }
15 }
```

Fig. 34.12 | Java code for class *Withdrawal* based on Figs. 34.9 and 34.10.

Congratulations on completing the case study's design portion! We implement the ATM system in Java code in Section 34.4. We recommend that you carefully read the code and its description. The code is abundantly commented and precisely follows the design with which you're now familiar. The accompanying description is carefully written to guide your understanding of the implementation based on the UML design. Mastering this code is a wonderful culminating accomplishment after studying Sections 33.2–33.7 and 34.2–34.3.

Self-Review Exercises for Section 34.3

- 34.4** The UML uses an arrow with a _____ to indicate a generalization relationship.
- solid filled arrowhead
 - triangular hollow arrowhead
 - diamond-shaped hollow arrowhead
 - stick arrowhead
- 34.5** State whether the following statement is *true* or *false*, and if *false*, explain why: The UML requires that we underline abstract class names and method names.
- 34.6** Write Java code to begin implementing the design for class `Transaction` specified in Figs. 34.9 and 34.10. Be sure to include private reference-type attributes based on class `Transaction`'s associations. Also be sure to include public `get` methods that provide access to any of these private attributes that the subclasses require to perform their tasks.

34.4 ATM Case Study Implementation

This section contains the complete implementation of the ATM system. We consider the classes in the order in which we identified them in Section 33.3—`ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `Transaction`, `BalanceInquiry`, `Withdrawal` and `Deposit`.

We apply the guidelines from Sections 34.2–34.3 to code these classes based on their UML class diagrams of Figs. 34.9 and 34.10. To develop the bodies of methods, we refer to the activity diagrams in Section 33.5 and the communication and sequence diagrams presented in Section 33.7. Our ATM design does *not* specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation. This is a *normal* part of the object-oriented design process. As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements document in Section 33.2.

We conclude the discussion by presenting a Java application (`ATMCaseStudy`) that starts the ATM and puts the other classes in the system in use. Recall that we're developing a first version of the ATM system that runs on a personal computer and uses the computer's keyboard and monitor to approximate the ATM's keypad and screen. We also simulate only the actions of the ATM's cash dispenser and deposit slot. We attempt to implement the system, however, so that real hardware versions of these devices could be integrated without significant changes in the code.

34.4.1 Class ATM

Class `ATM` (Fig. 34.13) represents the ATM as a whole. Lines 5–11 implement the class's attributes. We determine all but one of these attributes from the UML class diagrams of Figs. 34.9 and 34.10. We implement the UML `Boolean` attribute `userAuthenticated` in Fig. 34.10 as a `boolean` in Java (line 5). Line 6 declares an attribute not found in our UML design—an `int` attribute `currentAccountNumber` that keeps track of the account number of the current authenticated user. We'll soon see how the class uses this attribute. Lines 7–11 declare reference-type attributes corresponding to the `ATM` class's associations modeled in the class diagram of Fig. 34.9. These attributes allow the ATM to access its parts (i.e., its `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`) and interact with the bank's account-information database (i.e., a `BankDatabase` object).

```
1 // ATM.java
2 // Represents an automated teller machine
3
4 public class ATM {
5     private boolean userAuthenticated; // whether user is authenticated
6     private int currentAccountNumber; // current user's account number
7     private Screen screen; // ATM's screen
8     private Keypad keypad; // ATM's keypad
9     private CashDispenser cashDispenser; // ATM's cash dispenser
10    private DepositSlot depositSlot; // ATM's deposit slot
11    private BankDatabase bankDatabase; // account information database
12
13    // constants corresponding to main menu options
14    private static final int BALANCE_INQUIRY = 1;
15    private static final int WITHDRAWAL = 2;
16    private static final int DEPOSIT = 3;
17    private static final int EXIT = 4;
18
19    // no-argument ATM constructor initializes instance variables
20    public ATM() {
21        userAuthenticated = false; // user is not authenticated to start
22        currentAccountNumber = 0; // no current account number to start
23        screen = new Screen(); // create screen
24        keypad = new Keypad(); // create keypad
25        cashDispenser = new CashDispenser(); // create cash dispenser
26        depositSlot = new DepositSlot(); // create deposit slot
27        bankDatabase = new BankDatabase(); // create acct info database
28    }
29
30    // start ATM
31    public void run() {
32        // welcome and authenticate user; perform transactions
33        while (true) {
34            // loop while user is not yet authenticated
35            while (!userAuthenticated) {
36                screen.displayMessageLine("\nWelcome!");
37                authenticateUser(); // authenticate user
38            }
39
40            performTransactions(); // user is now authenticated
41            userAuthenticated = false; // reset before next ATM session
42            currentAccountNumber = 0; // reset before next ATM session
43            screen.displayMessageLine("\nThank you! Goodbye!");
44        }
45    }
46
47    // attempts to authenticate user against database
48    private void authenticateUser() {
49        screen.displayMessage("\nPlease enter your account number: ");
50        int accountNumber = keypad.getInput(); // input account number
51        screen.displayMessage("\nEnter your PIN: "); // prompt for PIN
52        int pin = keypad.getInput(); // input PIN
53    }
```

Fig. 34.13 | Class ATM represents the ATM. (Part 1 of 3.)

```
54     // set userAuthenticated to boolean value returned by database
55     userAuthenticated =
56         bankDatabase.authenticateUser(accountNumber, pin);
57
58     // check whether authentication succeeded
59     if (userAuthenticated) {
60         currentAccountNumber = accountNumber; // save user's account #
61     }
62     else {
63         screen.displayMessageLine(
64             "Invalid account number or PIN. Please try again.");
65     }
66 }
67
68 // display the main menu and perform transactions
69 private void performTransactions() {
70     // local variable to store transaction currently being processed
71     Transaction currentTransaction = null;
72
73     boolean userExited = false; // user has not chosen to exit
74
75     // loop while user has not chosen option to exit system
76     while (!userExited) {
77         // show main menu and get user selection
78         int mainMenuSelection = displayMainMenu();
79
80         // decide how to proceed based on user's menu selection
81         switch (mainMenuSelection) {
82             // user chose to perform one of three transaction types
83             case BALANCE_INQUIRY:
84             case WITHDRAWAL:
85             case DEPOSIT:
86
87                 // initialize as new object of chosen type
88                 currentTransaction =
89                     createTransaction(mainMenuSelection);
90
91                 currentTransaction.execute(); // execute transaction
92                 break;
93             case EXIT: // user chose to terminate session
94                 screen.displayMessageLine("\nExiting the system...");
95                 userExited = true; // this ATM session should end
96                 break;
97             default: // user did not enter an integer from 1-4
98                 screen.displayMessageLine(
99                     "\nYou did not enter a valid selection. Try again.");
100                break;
101            }
102        }
103    }
104 }
```

Fig. 34.13 | Class ATM represents the ATM. (Part 2 of 3.)

```
105 // display the main menu and return an input selection
106 private int displayMainMenu() {
107     screen.displayMessageLine("\nMain menu:");
108     screen.displayMessageLine("1 - View my balance");
109     screen.displayMessageLine("2 - Withdraw cash");
110     screen.displayMessageLine("3 - Deposit funds");
111     screen.displayMessageLine("4 - Exit\n");
112     screen.displayMessage("Enter a choice: ");
113     return keypad.getInput(); // return user's selection
114 }
115
116 // return object of specified Transaction subclass
117 private Transaction createTransaction(int type) {
118     Transaction temp = null; // temporary Transaction variable
119
120     // determine which type of Transaction to create
121     switch (type) {
122         case BALANCE_INQUIRY: // create new BalanceInquiry transaction
123             temp = new BalanceInquiry(
124                 currentAccountNumber, screen, bankDatabase);
125             break;
126         case WITHDRAWAL: // create new Withdrawal transaction
127             temp = new Withdrawal(currentAccountNumber, screen,
128                 bankDatabase, keypad, cashDispenser);
129             break;
130         case DEPOSIT: // create new Deposit transaction
131             temp = new Deposit(currentAccountNumber, screen,
132                 bankDatabase, keypad, depositSlot);
133             break;
134     }
135
136     return temp; // return the newly created object
137 }
138 }
```

Fig. 34.13 | Class ATM represents the ATM. (Part 3 of 3.)

Lines 14–17 declare integer constants that correspond to the four options in the ATM’s main menu (i.e., balance inquiry, withdrawal, deposit and exit). Lines 20–28 declare the constructor, which initializes the class’s attributes. When an ATM object is first created, no user is authenticated, so line 21 initializes `userAuthenticated` to `false`. Likewise, line 22 initializes `currentAccountNumber` to 0 because there’s no current user yet. Lines 23–26 instantiate new objects to represent the ATM’s parts. Recall that class ATM has composition relationships with classes Screen, Keypad, CashDispenser and DepositSlot, so class ATM is responsible for their creation. Line 27 creates a new BankDatabase. [Note: If this were a real ATM system, the ATM class would receive a reference to an existing database object created by the bank. However, in this implementation we’re only simulating the bank’s database, so class ATM creates the `BankDatabase` object with which it interacts.]

ATM Method run

The class diagram of Fig. 34.10 does not list any operations for class ATM. We now implement one operation (i.e., `public` method) in class ATM that allows an external client of the

class (i.e., class `ATMCaseStudy`) to tell the ATM to run. ATM method `run` (lines 31–45) uses an infinite loop to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an authenticated user performs the desired transactions and chooses to exit, the ATM resets itself, displays a goodbye message to the user and restarts the process. We use an infinite loop here to simulate the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control). An ATM user has the option to exit the system but not the ability to turn off the ATM completely.

Authenticating a User

In method `run`'s infinite loop, lines 35–38 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., `!userAuthenticated` is `true`). Line 36 invokes method `displayMessageLine` of the ATM's `screen` to display a welcome message. Like `Screen` method `displayMessage` designed in the case study, method `displayMessageLine` (declared in lines 11–13 of Fig. 34.14) displays a message to the user, but this method also outputs a newline after the message. We've added this method during implementation to give class `Screen`'s clients more control over the placement of displayed messages. Line 37 invokes class `ATM`'s private utility method `authenticateUser` (declared in lines 48–66) to attempt to authenticate the user.

We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 49 of method `authenticateUser` invokes method `displayMessage` of the `screen` to prompt the user to enter an account number. Line 50 invokes method `getInput` of the `keypad` to obtain the user's input, then stores the integer value entered by the user in a local variable `accountNumber`. Method `authenticateUser` next prompts the user to enter a PIN (line 51), and stores the PIN input by the user in a local variable `pin` (line 52). Next, lines 55–56 attempt to authenticate the user by passing the `accountNumber` and `pin` entered by the user to the `bankDatabase`'s `authenticateUser` method. Class `ATM` sets its `userAuthenticated` attribute to the `boolean` value returned by this method—`userAuthenticated` becomes `true` if authentication succeeds (i.e., `accountNumber` and `pin` match those of an existing `Account` in `bankDatabase`) and remains `false` otherwise. If `userAuthenticated` is `true`, line 60 saves the account number entered by the user (i.e., `accountNumber`) in the ATM attribute `currentAccountNumber`. The other ATM methods use this variable whenever an ATM session requires access to the user's account number. If `userAuthenticated` is `false`, lines 63–64 use the `screen`'s `displayMessageLine` method to indicate that an invalid account number and/or PIN was entered and the user must try again. We set `currentAccountNumber` only after authenticating the user's account number and the associated PIN—if the database could not authenticate the user, `currentAccountNumber` remains 0.

After method `run` attempts to authenticate the user (line 37), if `userAuthenticated` is still `false`, the `while` loop in lines 35–38 executes again. If `userAuthenticated` is now `true`, the loop terminates and control continues with line 40, which calls class `ATM`'s utility method `performTransactions`.

Performing Transactions

Method `performTransactions` (lines 69–103) carries out an ATM session for an authenticated user. Line 71 declares a local `Transaction` variable to which we'll assign a `Balance-`

Inquiry, Withdrawal or Deposit object representing the ATM transaction the user selected. We use a Transaction variable here to allow us to take advantage of polymorphism. Also, we name this variable after the *role name* included in the class diagram of Fig. 33.7—currentTransaction. Line 73 declares another local variable—a boolean called userExited that keeps track of whether the user has chosen to exit. This variable controls a while loop (lines 76–102) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 78 displays the main menu and obtains the user’s menu selection by calling an ATM utility method displayMainMenu (declared in lines 106–114). This method displays the main menu by invoking methods of the ATM’s screen and returns a menu selection obtained from the user through the ATM’s keypad. Line 78 stores the user’s selection returned by displayMainMenu in local variable mainMenuSelection.

After obtaining a main menu selection, method performTransactions uses a switch statement (lines 81–101) to respond to the selection appropriately. If mainMenuSelection is equal to any of the three integer constants representing transaction types (i.e., if the user chose to perform a transaction), lines 88–89 call utility method createTransaction (declared in lines 117–137) to return a newly instantiated object of the type that corresponds to the selected transaction. Variable currentTransaction is assigned the reference returned by createTransaction, then line 91 invokes method execute of this transaction to execute it. We’ll discuss Transaction method execute and the three Transaction subclasses shortly. We assign the Transaction variable currentTransaction an object of one of the three Transaction subclasses so that we can execute transactions *polymorphically*. For example, if the user chooses to perform a balance inquiry, mainMenuSelection equals BALANCE_INQUIRY, leading createTransaction to return a BalanceInquiry object. Thus, currentTransaction refers to a BalanceInquiry, and invoking currentTransaction.execute() results in BalanceInquiry’s version of execute being called.

Creating a Transaction

Method createTransaction (lines 117–137) uses a switch statement to instantiate a new Transaction subclass object of the type indicated by the parameter type. Recall that method performTransactions passes mainMenuSelection to this method only when mainMenuSelection contains a value corresponding to one of the three transaction types. Therefore type is BALANCE_INQUIRY, WITHDRAWAL or DEPOSIT. Each case in the switch statement instantiates a new object by calling the appropriate Transaction subclass constructor. Each constructor has a unique parameter list, based on the specific data required to initialize the subclass object. A BalanceInquiry requires only the account number of the current user and references to the ATM’s screen and the bankDatabase. In addition to these parameters, a Withdrawal requires references to the ATM’s keypad and cashDispenser, and a Deposit requires references to the ATM’s keypad and depositSlot. We discuss the transaction classes in more detail in Sections 34.4.8–34.4.11.

Exiting the Main Menu and Processing Invalid Selections

After executing a transaction (line 91 in performTransactions), userExited remains false and lines 76–102 repeat, returning the user to the main menu. However, if a user does not perform a transaction and instead selects the main menu option to exit, line 95 sets userExited to true, causing the condition of the while loop (!userExited) to be

come false. This while is the final statement of method `performTransactions`, so control returns to the calling method `run`. If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 98–99 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

Awaiting the Next ATM User

When `performTransactions` returns control to method `run`, the user has chosen to exit the system, so lines 41–42 reset the ATM's attributes `userAuthenticated` and `currentAccountNumber` to prepare for the next ATM user. Line 43 displays a goodbye message before the ATM starts over and welcomes the next user.

34.4.2 Class Screen

Class `Screen` (Fig. 34.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class `Screen` approximates a real ATM's screen with a computer monitor and outputs text messages using standard console output methods `System.out.print`, `System.out.println` and `System.out.printf`. In this case study, we designed class `Screen` to have one operation—`displayMessage`. For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` methods—`displayMessage`, `displayMessageLine` and `displayDollarAmount`.

```
1 // Screen.java
2 // Represents the screen of the ATM
3
4 public class Screen {
5     // display a message without a carriage return
6     public void displayMessage(String message) {
7         System.out.print(message);
8     }
9
10    // display a message with a carriage return
11    public void displayMessageLine(String message) {
12        System.out.println(message);
13    }
14
15    // displays a dollar amount
16    public void displayDollarAmount(double amount) {
17        System.out.printf("$%,.2f", amount);
18    }
19 }
```

Fig. 34.14 | Class `Screen` represents the screen of the ATM.

Method `displayMessage` (lines 6–8) takes a `String` argument and prints it to the console. The cursor stays on the same line, making this method appropriate for displaying prompts to the user. Method `displayMessageLine` (lines 11–13) does the same using `System.out.println`, which outputs a newline to move the cursor to the next line. Finally, method `displayDollarAmount` (lines 16–18) outputs a properly formatted dollar amount (e.g., \$1,234.56). Line 17 uses `System.out.printf` to output a `double` value formatted with commas to increase readability and two decimal places.

34.4.3 Class Keypad

Class Keypad (Fig. 34.15) represents the keypad of the ATM and is responsible for receiving all user input. Recall that we’re simulating this hardware, so we use the computer’s keyboard to approximate the keypad. We use class Scanner to obtain console input from the user. A computer keyboard contains many keys not found on the ATM’s keypad. However, we assume that the user presses only the keys on the computer keyboard that also appear on the keypad—the keys numbered 0–9 and the *Enter* key.

```

1 // Keypad.java
2 // Represents the keypad of the ATM
3 import java.util.Scanner; // program uses Scanner to obtain user input
4
5 public class Keypad {
6     private Scanner input; // reads data from the command line
7
8     // no-argument constructor initializes the Scanner
9     public Keypad() {
10         input = new Scanner(System.in);
11     }
12
13     // return an integer value entered by user
14     public int getInput() {
15         return input.nextInt(); // we assume that user enters an integer
16     }
17 }
```

Fig. 34.15 | Class Keypad represents the ATM’s keypad.

Line 6 declares Scanner variable `input` as an instance variable. Line 10 in the constructor creates a new Scanner object that reads input from the standard input stream (`System.in`) and assigns the object’s reference to variable `input`. Method `getInput` (lines 14–16) invokes Scanner method `nextInt` (line 15) to return the next integer input by the user. [Note: Method `nextInt` can throw an `InputMismatchException` if the user enters non-integer input. Because the real ATM’s keypad permits only integer input, we assume that no exception will occur and do not attempt to fix this problem. See Chapter 11, Exception Handling: A Deeper Look, for information on catching exceptions.] Recall that `nextInt` obtains all the input used by the ATM. `Keypad`’s `getInput` method simply returns the integer input by the user. If a client of class `Keypad` requires input that satisfies some criteria (i.e., a number corresponding to a valid menu option), the client must perform the error checking.

34.4.4 Class CashDispenser

Class `CashDispenser` (Fig. 34.16) represents the cash dispenser of the ATM. Line 6 declares constant `INITIAL_COUNT`, which indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., 500). Line 7 implements attribute `count` (modeled in Fig. 34.10), which keeps track of the number of bills remaining in the `CashDispenser` at any time. The constructor (lines 10–12) sets `count` to the initial count. `CashDispenser` has two `public` methods—`dispenseCash` (lines 15–18) and `isSufficientCashAvail-`

able (lines 21–30). The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only after establishing that sufficient cash is available by calling `isSufficientCashAvailable`. Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available.

```
1 // CashDispenser.java
2 // Represents the cash dispenser of the ATM
3
4 public class CashDispenser {
5     // the default initial number of bills in the cash dispenser
6     private final static int INITIAL_COUNT = 500;
7     private int count; // number of $20 bills remaining
8
9     // no-argument CashDispenser constructor initializes count to default
10    public CashDispenser() {
11        count = INITIAL_COUNT; // set count attribute to default
12    }
13
14    // simulates dispensing of specified amount of cash
15    public void dispenseCash(int amount) {
16        int billsRequired = amount / 20; // number of $20 bills required
17        count -= billsRequired; // update the count of bills
18    }
19
20    // indicates whether cash dispenser can dispense desired amount
21    public boolean isSufficientCashAvailable(int amount) {
22        int billsRequired = amount / 20; // number of $20 bills required
23
24        if (count >= billsRequired) {
25            return true; // enough bills available
26        }
27        else {
28            return false; // not enough bills available
29        }
30    }
31 }
```

Fig. 34.16 | Class `CashDispenser` represents the ATM's cash dispenser.

Method `isSufficientCashAvailable` has a parameter `amount` that specifies the amount of cash in question. Line 22 calculates the number of \$20 bills required to dispense the specified amount. The ATM allows the user to choose only withdrawal amounts that are multiples of \$20, so we divide `amount` by 20 to obtain the number of `billsRequired`. Lines 24–29 return `true` if the `CashDispenser`'s `count` is greater than or equal to `billsRequired` (i.e., enough bills are available) and `false` otherwise (i.e., not enough bills). For example, if a user wishes to withdraw \$80 (i.e., `billsRequired` is 4), but only three bills remain (i.e., `count` is 3), the method returns `false`.

Method `dispenseCash` (lines 15–18) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this method would interact with the device to physically dispense cash. Our version of the method simply decreases the `count` of bills remaining by the number required to dispense the specified `amount`. It's the responsibility

of the client of the class (i.e., `Withdrawal`) to inform the user that cash has been dispensed—`CashDispenser` cannot interact directly with `Screen`.

34.4.5 Class DepositSlot

Class `DepositSlot` (Fig. 34.17) represents the ATM's deposit slot. Like class `CashDispenser`, class `DepositSlot` merely simulates the functionality of a real hardware deposit slot. `DepositSlot` has no attributes and only one method—`isEnvelopeReceived`—which indicates whether a deposit envelope was received.

Recall from the requirements document that the ATM allows the user up to two minutes to insert an envelope. The current version of method `isEnvelopeReceived` simply returns `true` immediately (line 8), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, method `isEnvelopeReceived` might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If `isEnvelopeReceived` were to receive such a signal within two minutes, the method would return `true`. If two minutes elapsed and the method still had not received a signal, then the method would return `false`.

```

1 // DepositSlot.java
2 // Represents the deposit slot of the ATM
3
4 public class DepositSlot {
5     // indicates whether envelope was received (always returns true,
6     // because this is only a software simulation of a real deposit slot)
7     public boolean isEnvelopeReceived() {
8         return true; // deposit envelope was received
9     }
10 }
```

Fig. 34.17 | Class `DepositSlot` represents the ATM's deposit slot.

34.4.6 Class Account

Class `Account` (Fig. 34.18) represents a bank account. Each `Account` has four attributes (modeled in Fig. 34.10)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 5–8 implement these attributes as private fields. Variable `availableBalance` represents the amount of funds available for withdrawal. Variable `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

```

1 // Account.java
2 // Represents a bank account
3
4 public class Account {
5     private int accountNumber; // account number
```

Fig. 34.18 | Class `Account` represents a bank account. (Part I of 2.)

```
6  private int pin; // PIN for authentication
7  private double availableBalance; // funds available for withdrawal
8  private double totalBalance; // funds available + pending deposits
9
10 // Account constructor initializes attributes
11 public Account(int theAccountNumber, int thePIN,
12                 double theAvailableBalance, double theTotalBalance) {
13     accountNumber = theAccountNumber;
14     pin = thePIN;
15     availableBalance = theAvailableBalance;
16     totalBalance = theTotalBalance;
17 }
18
19 // determines whether a user-specified PIN matches PIN in Account
20 public boolean validatePIN(int userPIN) {
21     if (userPIN == pin) {
22         return true;
23     }
24     else {
25         return false;
26     }
27 }
28
29 // returns available balance
30 public double getAvailableBalance() {
31     return availableBalance;
32 }
33
34 // returns the total balance
35 public double getTotalBalance() {
36     return totalBalance;
37 }
38
39 // credits an amount to the account
40 public void credit(double amount) {
41     totalBalance += amount; // add to total balance
42 }
43
44 // debits an amount from the account
45 public void debit(double amount) {
46     availableBalance -= amount; // subtract from available balance
47     totalBalance -= amount; // subtract from total balance
48 }
49
50 // returns account number
51 public int getAccountNumber() {
52     return accountNumber;
53 }
54 }
```

Fig. 34.18 | Class Account represents a bank account. (Part 2 of 2.)

The Account class has a constructor (lines 11–17) that takes an account number, the PIN established for the account, the account's initial available balance and the account's

initial total balance as arguments. Lines 13–16 assign these values to the class's attributes (i.e., fields).

Method `validatePIN` (lines 20–27) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., attribute `pin`). Recall that we modeled this method's parameter `userPIN` in Fig. 33.19. If the two PINs match, the method returns `true`; otherwise, it returns `false`.

Methods `getAvailableBalance` (lines 30–32) and `getTotalBalance` (lines 35–37) return the values of `double` attributes `availableBalance` and `totalBalance`, respectively.

Method `credit` (lines 40–42) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. This method adds the amount only to attribute `totalBalance`. The money credited to an account during a deposit does *not* become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class `Account` includes only methods required for carrying out ATM transactions. Therefore, we omit the methods that some other bank system would invoke to add to attribute `availableBalance` (to confirm a deposit) or subtract from attribute `totalBalance` (to reject a deposit).

Method `debit` (lines 45–48) subtracts an amount of money (i.e., parameter `amount`) from an `Account` as part of a withdrawal transaction. This method subtracts the amount from *both* attribute `availableBalance` and attribute `totalBalance`, because a withdrawal affects *both* measures of an account balance.

Method `getAccountNumber` (lines 51–53) provides access to an `Account`'s `accountNumber`. We include this method in our implementation so that a client of the class (i.e., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can invoke this method on each of its `Account` objects to locate the one with a specific account number.

34.4.7 Class BankDatabase

Class `BankDatabase` (Fig. 34.19) models the bank's database with which the ATM interacts to access and modify a user's account information. We study database access in Chapter 24. For now we model the database as an array. An exercise in Chapter 24 asks you to reimplement this portion of the ATM using an actual database.

```

1 // BankDatabase.java
2 // Represents the bank account information database
3
4 public class BankDatabase {
5     private Account[] accounts; // array of Accounts
6
7     // no-argument BankDatabase constructor initializes accounts
8     public BankDatabase() {
9         accounts = new Account[2]; // just 2 accounts for testing
10        accounts[0] = new Account(12345, 54321, 1000.0, 1200.0);
11        accounts[1] = new Account(98765, 56789, 200.0, 200.0);
12    }
13

```

Fig. 34.19 | Class `BankDatabase` represents the bank's account information database. (Part I of 2.)

```
14 // retrieve Account object containing specified account number
15 private Account getAccount(int accountNumber) {
16     // loop through accounts searching for matching account number
17     for (Account currentAccount : accounts) {
18         // return current account if match found
19         if (currentAccount.getAccountNumber() == accountNumber) {
20             return currentAccount;
21         }
22     }
23
24     return null; // if no matching account was found, return null
25 }
26
27 // determine whether user-specified account number and PIN match
28 // those of an account in the database
29 public boolean authenticateUser(int userAccountNumber, int userPIN) {
30     // attempt to retrieve the account with the account number
31     Account userAccount = getAccount(userAccountNumber);
32
33     // if account exists, return result of Account method validatePIN
34     if (userAccount != null) {
35         return userAccount.validatePIN(userPIN);
36     }
37     else {
38         return false; // account number not found, so return false
39     }
40 }
41
42 // return available balance of Account with specified account number
43 public double getAvailableBalance(int userAccountNumber) {
44     return getAccount(userAccountNumber).getAvailableBalance();
45 }
46
47 // return total balance of Account with specified account number
48 public double getTotalBalance(int userAccountNumber) {
49     return getAccount(userAccountNumber).getTotalBalance();
50 }
51
52 // credit an amount to Account with specified account number
53 public void credit(int userAccountNumber, double amount) {
54     getAccount(userAccountNumber).credit(amount);
55 }
56
57 // debit an amount from Account with specified account number
58 public void debit(int userAccountNumber, double amount) {
59     getAccount(userAccountNumber).debit(amount);
60 }
61 }
```

Fig. 34.19 | Class BankDatabase represents the bank's account information database. (Part 2 of 2.)

We determine one reference-type attribute for class `BankDatabase` based on its composition relationship with class `Account`. Recall from Fig. 34.9 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 5 implements attribute `accounts`—an

array of Account objects—to implement this composition relationship. Class BankDatabase's no-argument constructor (lines 8–12) initializes accounts with new Account objects. For the sake of testing the system, we declare accounts to hold just two array elements, which we instantiate as new Account objects with test data. The Account constructor has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance. Recall that class BankDatabase serves as an intermediary between class ATM and the actual Account objects that contain a user's account information. Thus, the methods of class BankDatabase do nothing more than invoke the corresponding methods of the Account object belonging to the current ATM user.

We include private utility method getAccount (lines 15–25) to allow the BankDatabase to obtain a reference to a particular Account within array accounts. To locate the user's Account, the BankDatabase compares the value returned by method getAccountNumber for each element of accounts to a specified account number until it finds a match. Lines 17–22 traverse the accounts array. If the account number of currentAccount equals the value of parameter accountNumber, the method immediately returns the currentAccount. If no account has the given account number, then line 24 returns null.

Method authenticateUser (lines 29–40) proves or disproves the identity of an ATM user. This method takes a user-specified account number and PIN as arguments and indicates whether they match the account number and PIN of an Account in the database. Line 31 calls method getAccount, which returns either an Account with userAccountNumber as its account number or null to indicate that userAccountNumber is invalid. If getAccount returns an Account object, line 35 returns the boolean value returned by that object's validatePIN method. BankDatabase's authenticateUser method does not perform the PIN comparison itself—rather, it forwards userPIN to the Account object's validatePIN method to do so. The value returned by Account method validatePIN indicates whether the user-specified PIN matches the PIN of the user's Account, so method authenticateUser simply returns this value to the class's client (i.e., ATM).

BankDatabase trusts the ATM to invoke method authenticateUser and receive a return value of true before allowing the user to perform transactions. BankDatabase also trusts that each Transaction object created by the ATM contains the valid account number of the current authenticated user and that this is the account number passed to the remaining BankDatabase methods as argument userAccountNumber. Methods getAvailableBalance (lines 43–45), getTotalBalance (lines 48–50), credit (lines 53–55) and debit (lines 58–60) therefore simply retrieve the user's Account object with utility method getAccount, then invoke the appropriate Account method on that object. We know that the calls to getAccount from these methods will never return null, because userAccountNumber must refer to an existing Account. Methods getAvailableBalance and getTotalBalance return the values returned by the corresponding Account methods. Also, credit and debit simply redirect parameter amount to the Account methods they invoke.

34.4.8 Class Transaction

Class Transaction (Fig. 34.20) is an abstract superclass that represents the notion of an ATM transaction. It contains the common features of subclasses BalanceInquiry, Withdrawal and Deposit. This class expands upon the “skeleton” code first developed in Section 34.3. Line 4 declares this class to be abstract. Lines 5–7 declare the class's private attributes. Recall from the class diagram of Fig. 34.10 that class Transaction con-

tains an attribute `accountNumber` (line 5) that indicates the account involved in the `Transaction`. We derive attributes `screen` (line 6) and `bankDatabase` (line 7) from class `Transaction`'s associations modeled in Fig. 34.9—all transactions require access to the ATM's screen and the bank's database.

```
1 // Transaction.java
2 // Abstract superclass Transaction represents an ATM transaction
3
4 public abstract class Transaction {
5     private int accountNumber; // indicates account involved
6     private Screen screen; // ATM's screen
7     private BankDatabase bankDatabase; // account info database
8
9     // Transaction constructor invoked by subclasses using super()
10    public Transaction(int userAccountNumber, Screen atmScreen,
11                      BankDatabase atmBankDatabase) {
12
13        accountNumber = userAccountNumber;
14        screen = atmScreen;
15        bankDatabase = atmBankDatabase;
16    }
17
18    // return account number
19    public int getAccountNumber() {
20        return accountNumber;
21    }
22
23    // return reference to screen
24    public Screen getScreen() {
25        return screen;
26    }
27
28    // return reference to bank database
29    public BankDatabase getBankDatabase() {
30        return bankDatabase;
31    }
32
33    // perform the transaction (overridden by each subclass)
34    abstract public void execute();
35 }
```

Fig. 34.20 | Abstract superclass `Transaction` represents an ATM transaction.

Class `Transaction`'s constructor (lines 10–16) takes as arguments the current user's account number and references to the ATM's screen and the bank's database. Because `Transaction` is an *abstract* class, this constructor will be called only by the constructors of the `Transaction` subclasses.

The class has three `public get` methods—`getAccountNumber` (lines 19–21), `getScreen` (lines 24–26) and `getBankDatabase` (lines 29–31). These are inherited by `Transaction` subclasses and used to gain access to class `Transaction`'s `private` attributes.

Class `Transaction` also declares `abstract` method `execute` (line 34). It does not make sense to provide this method's implementation, because a generic transaction cannot

be executed. So, we declare this method `abstract` and force each `Transaction` subclass to provide a concrete implementation that executes that particular type of transaction.

34.4.9 Class BalanceInquiry

`Class BalanceInquiry` (Fig. 34.21) extends `Transaction` and represents a balance-inquiry ATM transaction. `BalanceInquiry` does not have any attributes of its own, but it inherits `Transaction` attributes `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`'s public `get` methods. The `BalanceInquiry` constructor takes arguments corresponding to these attributes and simply forwards them to `Transaction`'s constructor using `super` (line 9).

```

1 // BalanceInquiry.java
2 // Represents a balance inquiry ATM transaction
3
4 public class BalanceInquiry extends Transaction {
5     // BalanceInquiry constructor
6     public BalanceInquiry(int userAccountNumber, Screen atmScreen,
7             BankDatabase atmBankDatabase) {
8
9         super(userAccountNumber, atmScreen, atmBankDatabase);
10    }
11
12    // performs the transaction
13    @Override
14    public void execute() {
15        // get references to bank database and screen
16        BankDatabase bankDatabase = getBankDatabase();
17        Screen screen = getScreen();
18
19        // get the available balance for the account involved
20        double availableBalance =
21            bankDatabase.getAvailableBalance(getAccountNumber());
22
23        // get the total balance for the account involved
24        double totalBalance =
25            bankDatabase.getTotalBalance(getAccountNumber());
26
27        // display the balance information on the screen
28        screen.displayMessageLine("\nBalance Information:");
29        screen.displayMessage(" - Available balance: ");
30        screen.displayDollarAmount(availableBalance);
31        screen.displayMessage("\n - Total balance:      ");
32        screen.displayDollarAmount(totalBalance);
33        screen.displayMessageLine("");
34    }
35}

```

Fig. 34.21 | `Class BalanceInquiry represents a balance-inquiry ATM transaction.`

`Class BalanceInquiry` overrides `Transaction`'s abstract method `execute` to provide a concrete implementation (lines 13–34) that performs the steps involved in a balance

inquiry. Lines 16–17 get references to the bank database and the ATM’s screen by invoking methods inherited from superclass `Transaction`. Lines 20–21 retrieve the available balance of the account involved by invoking method `getAvailableBalance` of `BankDatabase`. Line 21 uses inherited method `getAccountNumber` to get the account number of the current user, which it then passes to `getAvailableBalance`. Lines 24–25 retrieve the total balance of the current user’s account. Lines 28–33 display the balance information on the ATM’s screen. Recall that `displayDollarAmount` takes a `double` argument and outputs it to the screen formatted as a dollar amount. For example, if a user’s `availableBalance` is `1000.5`, line 30 outputs `$1,000.50`. Line 33 inserts a blank line of output to separate the balance information from subsequent output (i.e., the main menu repeated by class `ATM` after executing the `BalanceInquiry`).

34.4.10 Class Withdrawal

Class `Withdrawal` (Fig. 34.22) extends `Transaction` and represents a withdrawal ATM transaction. This class expands upon the “skeleton” code for this class developed in Fig. 34.12. Recall from the class diagram of Fig. 34.10 that class `Withdrawal` has one attribute, `amount`, which line 5 implements as an `int` field. Figure 34.9 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 6–7 implement reference-type attributes `keypad` and `cashDispenser`, respectively. Line 10 declares a constant corresponding to the cancel menu option. We’ll soon discuss how the class uses this constant.

```
1 // Withdrawal.java
2 // Represents a withdrawal ATM transaction
3
4 public class Withdrawal extends Transaction {
5     private int amount; // amount to withdraw
6     private Keypad keypad; // reference to keypad
7     private CashDispenser cashDispenser; // reference to cash dispenser
8
9     // constant corresponding to menu option to cancel
10    private final static int CANCELED = 6;
11
12    // Withdrawal constructor
13    public Withdrawal(int userAccountNumber, Screen atmScreen,
14                      BankDatabase atmBankDatabase, Keypad atmKeypad,
15                      CashDispenser atmCashDispenser) {
16
17        // initialize superclass variables
18        super(userAccountNumber, atmScreen, atmBankDatabase);
19
20        // initialize references to keypad and cash dispenser
21        keypad = atmKeypad;
22        cashDispenser = atmCashDispenser;
23    }
24}
```

Fig. 34.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part I of 3.)

```
25    // perform transaction
26    @Override
27    public void execute() {
28        boolean cashDispensed = false; // cash was not dispensed yet
29        double availableBalance; // amount available for withdrawal
30
31        // get references to bank database and screen
32        BankDatabase bankDatabase = getBankDatabase();
33        Screen screen = getScreen();
34
35        // loop until cash is dispensed or the user cancels
36        do {
37            // obtain a chosen withdrawal amount from the user
38            amount = displayMenuOfAmounts();
39
40            // check whether user chose a withdrawal amount or canceled
41            if (amount != CANCELED) {
42                // get available balance of account involved
43                availableBalance =
44                    bankDatabase.getAvailableBalance(getAccountNumber());
45
46                // check whether the user has enough money in the account
47                if (amount <= availableBalance) {
48                    // check whether the cash dispenser has enough money
49                    if (cashDispenser.isSufficientCashAvailable(amount)) {
50                        // update the account involved to reflect the withdrawal
51                        bankDatabase.debit(getAccountNumber(), amount);
52
53                        cashDispenser.dispenseCash(amount); // dispense cash
54                        cashDispensed = true; // cash was dispensed
55
56                        // instruct user to take cash
57                        screen.displayMessageLine("\nYour cash has been" +
58                            " dispensed. Please take your cash now.");
59                    }
60                    else { // cash dispenser does not have enough cash
61                        screen.displayMessageLine(
62                            "\nInsufficient cash available in the ATM." +
63                            "\n\nPlease choose a smaller amount.");
64                    }
65                }
66                else { // not enough money available in user's account
67                    screen.displayMessageLine(
68                        "\nInsufficient funds in your account." +
69                        "\n\nPlease choose a smaller amount.");
70                }
71            }
72            else { // user chose cancel menu option
73                screen.displayMessageLine("\nCanceling transaction...");
74                return; // return to main menu because user canceled
75            }
76        } while (!cashDispensed);
77    }
```

Fig. 34.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 2 of 3.)

```
78      // display a menu of withdrawal amounts and the option to cancel;
79      // return the chosen amount or 0 if the user chooses to cancel
80      private int displayMenuOfAmounts() {
81          int userChoice = 0; // local variable to store return value
82
83          Screen screen = getScreen(); // get screen reference
84
85          // array of amounts to correspond to menu numbers
86          int[] amounts = {0, 20, 40, 60, 100, 200};
87
88          // loop while no valid choice has been made
89          while (userChoice == 0) {
90              // display the withdrawal menu
91              screen.displayMessageLine("\nWithdrawal Menu:");
92              screen.displayMessageLine("1 - $20");
93              screen.displayMessageLine("2 - $40");
94              screen.displayMessageLine("3 - $60");
95              screen.displayMessageLine("4 - $100");
96              screen.displayMessageLine("5 - $200");
97              screen.displayMessageLine("6 - Cancel transaction");
98              screen.displayMessageLine("Choose a withdrawal amount: ");
99
100             int input = keypad.getInput(); // get user input through keypad
101
102             // determine how to proceed based on the input value
103             switch (input) {
104                 case 1: // if the user chose a withdrawal amount
105                 case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
106                     // corresponding amount from amounts array
107                 case 4:
108                 case 5:
109                     userChoice = amounts[input]; // save user's choice
110                     break;
111                 case CANCELED: // the user chose to cancel
112                     userChoice = CANCELED; // save user's choice
113                     break;
114                 default: // the user did not enter a value from 1-6
115                     screen.displayMessageLine(
116                         "\nInvalid selection. Try again.");
117             }
118         }
119     }
120
121     return userChoice; // return withdrawal amount or CANCELED
122 }
123 }
```

Fig. 34.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 3 of 3.)

Class Withdrawal's constructor (lines 13–23) has five parameters. It uses super to pass parameters userAccountNumber, atmScreen and atmBankDatabase to superclass Transaction's constructor to set the attributes that Withdrawal inherits from Transaction. The constructor also takes references atmKeypad and atmCashDispenser as parameters and assigns them to reference-type attributes keypad and cashDispenser.

Class `Withdrawal` overrides `Transaction` method `execute` with a concrete implementation (lines 26–77) that performs the steps of a withdrawal. Line 28 declares and initializes a local `boolean` variable `cashDispensed`, which indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 29 declares local `double` variable `availableBalance`, which will store the user's available balance during a withdrawal transaction. Lines 32–33 get references to the bank database and the ATM's screen by invoking methods inherited from superclass `Transaction`.

Lines 36–76 execute until cash is dispensed (i.e., until `cashDispensed` becomes `true`) or until the user chooses to cancel (in which case, the loop terminates). We use this loop to continuously return the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 38 displays a menu of withdrawal amounts and obtains a user selection by calling private utility method `displayMenuOfAmounts` (declared in lines 81–122). This method displays the menu of amounts and returns either an `int` withdrawal amount or an `int` constant `CANCELED` to indicate that the user has chosen to cancel the transaction.

Method `displayMenuOfAmounts` (lines 81–122) first declares local variable `userChoice` (initially 0) to store the value that the method will return (line 82). Line 84 gets a reference to the screen by calling method `getScreen` inherited from superclass `Transaction`. Line 87 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index 0) because the menu has no option 0. Lines 90–119 repeat until `userChoice` takes on a value other than 0. We'll see shortly that this occurs when the user makes a valid selection from the menu. Lines 92–99 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 101 obtains integer `input` through the keypad. The `switch` statement at lines 104–118 determines how to proceed based on the user's input. If the user selects a number between 1 and 5, line 110 sets `userChoice` to the value of the element in `amounts` at index `input`. For example, if the user enters 3 to withdraw \$60, line 110 sets `userChoice` to the value of `amounts[3]` (i.e., 60). Variable `userChoice` no longer equals 0, so the loop terminates and line 121 returns `userChoice`. If the user selects the cancel menu option, lines 113–114 execute, setting `userChoice` to `CANCELED` and causing the method to return this value. If the user does not enter a valid menu selection, lines 116–117 display an error message and the user is returned to the withdrawal menu.

Line 41 in method `execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 73–74 execute and display an appropriate message to the user before returning control to the calling method (i.e., ATM method `performTransactions`). If the user has chosen a withdrawal amount, lines 43–44 retrieve the available balance of the current user's `Account` and store it in variable `availableBalance`. Next, line 47 determines whether the selected amount is less than or equal to the user's available balance. If it's not, lines 67–69 display an appropriate error message. Control then continues to the end of the `do...while`, and the loop repeats because `cashDispensed` is still `false`. If the user's balance is high enough, the `if` statement at line 49 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the `cashDispenser`'s `isSufficientCashAvailable` method. If this method returns `false`, lines 61–63 display an appropriate error message and the `do...while` repeats. If sufficient cash is available, then the requirements for the withdrawal are satis-

fied, and line 51 debits amount from the user's account in the database. Lines 53–54 then instruct the cash dispenser to dispense the cash to the user and set cashDispensed to true. Finally, lines 57–58 display a message to the user that cash has been dispensed. Because cashDispensed is now true, control continues after the do...while. No additional statements appear below the loop, so the method returns.

34.4.11 Class Deposit

Class Deposit (Fig. 34.23) extends Transaction and represents a deposit transaction. Recall from Fig. 34.10 that class Deposit has one attribute amount, which line 5 implements as an int field. Lines 6–7 create reference attributes keypad and depositSlot that implement the associations between class Deposit and classes Keypad and DepositSlot modeled in Fig. 34.9. Line 8 declares a constant CANCELED that corresponds to the value a user enters to cancel. We'll soon discuss how the class uses this constant.

```
1 // Deposit.java
2 // Represents a deposit ATM transaction
3
4 public class Deposit extends Transaction {
5     private double amount; // amount to deposit
6     private Keypad keypad; // reference to keypad
7     private DepositSlot depositSlot; // reference to deposit slot
8     private final static int CANCELED = 0; // constant for cancel option
9
10    // Deposit constructor
11    public Deposit(int userAccountNumber, Screen atmScreen,
12                   BankDatabase atmBankDatabase, Keypad atmKeypad,
13                   DepositSlot atmDepositSlot) {
14
15        // initialize superclass variables
16        super(userAccountNumber, atmScreen, atmBankDatabase);
17
18        // initialize references to keypad and deposit slot
19        keypad = atmKeypad;
20        depositSlot = atmDepositSlot;
21    }
22
23    // perform transaction
24    @Override
25    public void execute() {
26        BankDatabase bankDatabase = getBankDatabase(); // get reference
27        Screen screen = getScreen(); // get reference
28
29        amount = promptForDepositAmount(); // get deposit amount from user
30
31        // check whether user entered a deposit amount or canceled
32        if (amount != CANCELED) {
33            // request deposit envelope containing specified amount
34            screen.displayMessage(
35                "\nPlease insert a deposit envelope containing ");
```

Fig. 34.23 | Class Deposit represents a deposit ATM transaction. (Part I of 2.)

```
36         screen.displayDollarAmount(amount);
37         screen.displayMessageLine(".");
38
39         // receive deposit envelope
40         boolean envelopeReceived = depositSlot.isEnvelopeReceived();
41
42         // check whether deposit envelope was received
43         if (envelopeReceived) {
44             screen.displayMessageLine("\nYour envelope has been " +
45                 "received.\nNOTE: The money just deposited will not " +
46                 "be available until we verify the amount of any " +
47                 "enclosed cash and your checks clear.");
48
49             // credit account to reflect the deposit
50             bankDatabase.credit(getAccountNumber(), amount);
51         }
52         else { // deposit envelope not received
53             screen.displayMessageLine("\nYou did not insert an " +
54                 "envelope, so the ATM has canceled your transaction.");
55         }
56     }
57     else { // user canceled instead of entering amount
58         screen.displayMessageLine("\nCanceling transaction...");
59     }
60 }
61
62 // prompt user to enter a deposit amount in cents
63 private double promptForDepositAmount() {
64     Screen screen = getScreen(); // get reference to screen
65
66     // display the prompt
67     screen.displayMessage("\nPlease enter a deposit amount in " +
68         "CENTS (or 0 to cancel): ");
69     int input = keypad.getInput(); // receive input of deposit amount
70
71     // check whether the user canceled or entered a valid amount
72     if (input == CANCELED) {
73         return CANCELED;
74     }
75     else {
76         return (double) input / 100; // return dollar amount
77     }
78 }
79 }
```

Fig. 34.23 | Class Deposit represents a deposit ATM transaction. (Part 2 of 2.)

Like Withdrawal, class Deposit's constructor (lines 11–21) passes three parameters to superclass Transaction's constructor. The constructor also has parameters atmKeypad and atmDepositSlot, which it assigns to corresponding attributes.

Method execute (lines 24–60) overrides the abstract version in superclass Transaction with a concrete implementation that performs the steps required in a deposit transaction. Lines 26–27 get references to the database and the screen. Line 29 prompts the user

to enter a deposit amount by invoking `private` utility method `promptForDepositAmount` (declared in lines 63–78) and sets attribute `amount` to the value returned. Method `promptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM’s keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the `double` value representing the dollar amount to be deposited.

Line 64 in method `promptForDepositAmount` gets a reference to the ATM’s screen. Lines 67–68 display a message asking the user to input a deposit amount as a number of cents or “0” to cancel the transaction. Line 69 receives the user’s input from the keypad. Lines 72–77 determine whether the user has entered a real deposit amount or chosen to cancel. If the latter, line 73 returns the constant `CANCELED`. Otherwise, line 76 returns the deposit amount after converting from the number of cents to a dollar amount by casting `input` to a `double`, then dividing by 100. For example, if the user enters 125 as the number of cents, line 76 returns 125.0 divided by 100, or 1.25—125 cents is \$1.25.

Line 32 in method `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If so, line 58 displays an appropriate message, and the method returns. If the user enters a deposit amount, lines 34–37 instruct the user to insert a deposit envelope with the correct amount. Recall that `Screen` method `displayDollarAmount` outputs a `double` formatted as a dollar amount.

Line 40 sets a local `boolean` variable to the value returned by `depositSlot`’s `isEnvelopeReceived` method, indicating whether a deposit envelope has been received. Recall that we coded method `isEnvelopeReceived` (Fig. 34.17) to always return `true`, because we’re simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code method `execute` of class `Deposit` to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for *all* possible return values. Thus, class `Deposit` is prepared for future versions of `isEnvelopeReceived` that could return `false`. Lines 44–50 execute if the deposit slot receives an envelope. Lines 44–47 display an appropriate message to the user. Line 50 then credits the deposit amount to the user’s account in the database. Lines 53–54 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The method then returns without modifying the user’s account.

34.4.12 Class ATMCaseStudy

Class `ATMCaseStudy` (Fig. 34.24) is a simple class that allows us to start, or “turn on,” the ATM and test the implementation of our ATM system model. Class `ATMCaseStudy`’s `main` method instantiates a new `ATM` object named `theATM` and invokes its `run` method to start the ATM.

```
1 // ATMCaseStudy.java
2 // Driver program for the ATM case study
3
4 public class ATMCaseStudy {
5     // main method creates and runs the ATM
6     public static void main(String[] args) {
```

Fig. 34.24 | `ATMCaseStudy.java` starts the ATM.

```

7     ATM theATM = new ATM();
8     theATM.run();
9 }
10 }
```

Fig. 34.24 | ATMCaseStudy.java starts the ATM.

34.5 Wrap-Up

In this chapter, you used inheritance to tune the design of the ATM software system, and you fully implemented the ATM in Java. Congratulations on completing the entire ATM case study! We hope you found this experience to be valuable and that it reinforced many of the object-oriented programming concepts that you've learned. In the next chapter, we present the Java Platform Module System—Java 9's most important new software-engineering technology.

Answers to Self-Review Exercises

34.1 True. The minus sign (-) indicates private visibility.

34.2 b.

34.3 The design for class Keypad yields the code in Fig. 34.25. Recall that class Keypad has no attributes for the moment, but attributes may become apparent as we continue the implementation. Also, if we were designing a real ATM, method `getInput` would need to interact with the ATM's keypad hardware. We'll actually read input from the keyboard of a personal computer when we write the complete Java code in Section 34.4.

```

1 // Class Keypad represents an ATM's keypad
2 public class Keypad {
3     // no attributes have been specified yet
4
5     // no-argument constructor
6     public Keypad() { }
7
8     // operations
9     public int getInput() { }
10 }
```

Fig. 34.25 | Java code for class Keypad based on Figs. 34.1–34.2.

34.4 b.

34.5 False. The UML requires that we italicize abstract class names and method names.

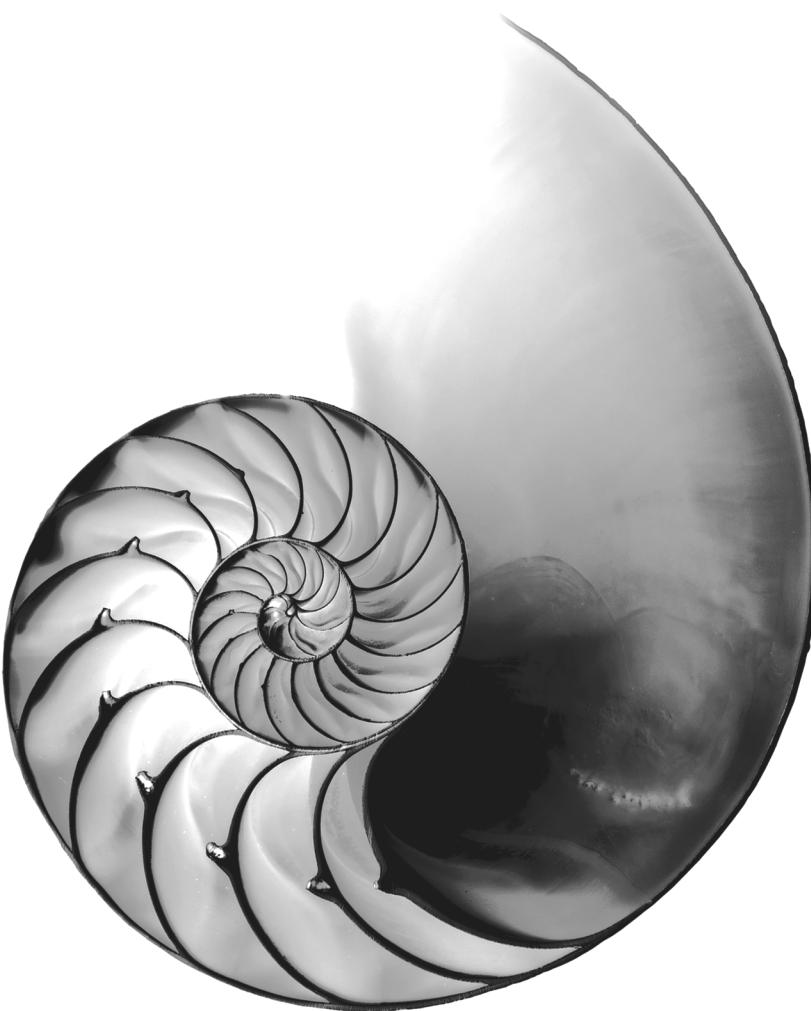
34.6 The design for class Transaction yields the code in Fig. 34.26. The bodies of the class constructor and methods are completed in Section 34.4. When fully implemented, methods `getScreen` and `getBankDatabase` will return superclass `Transaction`'s private reference attributes `screen` and `bankDatabase`, respectively. These methods allow the `Transaction` subclasses to access the ATM's screen and interact with the bank's database.

```
1 // Abstract class Transaction represents an ATM transaction
2 public abstract class Transaction {
3     // attributes
4     private int accountNumber; // indicates account involved
5     private Screen screen; // ATM's screen
6     private BankDatabase bankDatabase; // account info database
7
8     // no-argument constructor invoked by subclasses using super()
9     public Transaction() { }
10
11    // return account number
12    public int getAccountNumber() { }
13
14    // return reference to screen
15    public Screen getScreen() { }
16
17    // return reference to bank database
18    public BankDatabase getBankDatabase() { }
19
20    // abstract method overridden by subclasses
21    public abstract void execute();
22 }
```

Fig. 34.26 | Java code for class `Transaction` based on Figs. 34.9 and 34.10.

Swing GUI Components: Part 2

35



Objectives

In this chapter you'll:

- Create and manipulate sliders, menus, pop-up menus and windows.
- Programmatically change the look-and-feel of a GUI, using Swing's pluggable look-and-feel.
- Create a multiple-document interface with `JDesktopPane` and `JInternalFrame`.
- Use additional layout managers `BoxLayout` and `GridBagLayout`.



35.1	Introduction	22.7	JDesktopPane and JInternalFrame
22.2	JSlider	22.8	JTabbedPane
35.3	Understanding Windows in Java	22.9	BoxLayout Layout Manager
35.4	Using Menus with Frames	22.10	GridBagLayout Layout Manager
22.5	JPopupMenu	35.11	Wrap-Up
35.6	Pluggable Look-and-Feel		

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

35.1 Introduction

[Note: JavaFX (Chapters 12, 13 and 22) is Java's GUI, graphics and multimedia API of the future. This chapter is provided *as is* for those still interested in Swing GUIs.]

In this chapter, we continue our study of Swing GUIs. We discuss additional components and layout managers and lay the groundwork for building more complex GUIs. We begin with sliders for selecting from a range of integer values, then discuss additional details of windows. Next, you'll use menus to organize an application's commands.

The look-and-feel of a Swing GUI can be uniform across all platforms on which a Java program executes, or the GUI can be customized by using Swing's **pluggable look-and-feel (PLAF)**. We provide an example that illustrates how to change between Swing's default metal look-and-feel (which looks and behaves the same across platforms), the Nimbus look-and-feel (introduced in Chapter 26), a look-and-feel that simulates **Motif** (a UNIX look-and-feel) and one that simulates the Microsoft Windows look-and-feel.

Many of today's applications use a multiple-document interface (MDI)—a main window (often called the *parent window*) containing other windows (often called *child windows*) to manage several open documents in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so that you can compose or read multiple e-mail messages. We demonstrate Swing's classes for creating multiple-document interfaces. Finally, you'll learn about additional layout managers for organizing graphical user interfaces. We use several more Swing GUI components in later chapters as they're needed.

Swing is now considered a legacy technology. For GUIs, graphics and multimedia in new Java apps, you should use the features presented in this book's JavaFX chapters.

8 Java SE 8: Implementing Event Listeners with Lambdas

Throughout this chapter, we use anonymous inner classes and nested classes to implement event handlers so that the examples can compile and execute with both Java SE 7 and Java SE 8. In many of the examples, you could implement the functional event-listener interfaces with Java SE 8 lambdas (as demonstrated in Section 17.16).

35.2 JSlider

JSiders enable a user to select from a range of integer values. Class **JSlider** inherits from **JComponent**. Figure 35.1 shows a horizontal **JSlider** with **tick marks** and the **thumb** that allows a user to select a value. **JSiders** can be customized to display **major tick marks**,

minor tick marks and labels for the tick marks. They also support **snap-to ticks**, which cause the *thumb*, when positioned between two tick marks, to snap to the closest one.



Fig. 35.1 | JSlider component with horizontal orientation.

Most Swing GUI components support mouse and keyboard interactions—e.g., if a JSlider has the focus (i.e., it's the currently selected GUI component in the user interface), pressing the left arrow key or right arrow key causes the JSlider's thumb to decrease or increase by 1, respectively. The down arrow key and up arrow key also cause the thumb to decrease or increase by 1 tick, respectively. The *PgDn* (page down) key and *PgUp* (page up) key cause the thumb to decrease or increase by **block increments** of one-tenth of the range of values, respectively. The *Home* key moves the thumb to the minimum value of the JSlider, and the *End* key moves the thumb to the maximum value of the JSlider.

JSliders have either a horizontal or a vertical orientation. For a horizontal JSlider, the minimum value is at the left end and the maximum is at the right end. For a vertical JSlider, the minimum value is at the bottom and the maximum is at the top. The minimum and maximum value positions on a JSlider can be reversed by invoking JSlider method **setInverted** with boolean argument **true**. The relative position of the thumb indicates the current value of the JSlider.

The program in Figs. 35.2–35.4 allows the user to size a circle drawn on a subclass of JPanel called OvalPanel (Fig. 35.2). The user specifies the circle's diameter with a horizontal JSlider. Class OvalPanel knows how to draw a circle on itself, using its own instance variable **diameter** to determine the diameter of the circle—the **diameter** is used as the width and height of the bounding box in which the circle is displayed. The **diameter** value is set when the user interacts with the JSlider. The event handler calls method **setDiameter** in class OvalPanel to set the **diameter** and calls **repaint** to draw the new circle. The **repaint** call results in a call to OvalPanel's **paintComponent** method.

```

1 // Fig. 22.2: OvalPanel.java
2 // A customized JPanel class.
3 import java.awt.Graphics;
4 import java.awt.Dimension;
5 import javax.swing.JPanel;
6
7 public class OvalPanel extends JPanel
8 {
9     private int diameter = 10; // default diameter
10
11    // draw an oval of the specified diameter
12    @Override
13    public void paintComponent(Graphics g)
14    {
15        super.paintComponent(g);

```

Fig. 35.2 | JPanel subclass for drawing circles of a specified diameter. (Part I of 2.)

```
16     g.fillOval(10, 10, diameter, diameter);
17 }
18
19 // validate and set diameter, then repaint
20 public void setDiameter(int newDiameter)
21 {
22     // if diameter invalid, default to 10
23     diameter = (newDiameter >= 0 ? newDiameter : 10);
24     repaint(); // repaint panel
25 }
26
27 // used by layout manager to determine preferred size
28 public Dimension getPreferredSize()
29 {
30     return new Dimension(200, 200);
31 }
32
33 // used by layout manager to determine minimum size
34 public Dimension getMinimumSize()
35 {
36     return getPreferredSize();
37 }
38 } // end class OvalPanel
```

Fig. 35.2 | JPanel subclass for drawing circles of a specified diameter. (Part 2 of 2.)

```
1 // Fig. 22.3: SliderFrame.java
2 // Using JSliders to size an oval.
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JSlider;
7 import javax.swing.SwingConstants;
8 import javax.swing.event.ChangeListener;
9 import javax.swing.event.ChangeEvent;
10
11 public class SliderFrame extends JFrame
12 {
13     private final JSlider diameterJSlider; // slider to select diameter
14     private final OvalPanel myPanel; // panel to draw circle
15
16     // no-argument constructor
17     public SliderFrame()
18     {
19         super("Slider Demo");
20
21         myPanel = new OvalPanel(); // create panel to draw circle
22         myPanel.setBackground(Color.YELLOW);
23
24         // set up JSlider to control diameter value
25         diameterJSlider =
26             new JSlider(SwingConstants.HORIZONTAL, 0, 200, 10);
```

Fig. 35.3 | JSlider value used to determine the diameter of a circle. (Part 1 of 2.)

```

27     diameterJSlider.setMajorTickSpacing(10); // create tick every 10
28     diameterJSlider.setPaintTicks(true); // paint ticks on slider
29
30     // register JSlider event listener
31     diameterJSlider.addChangeListener(
32         new ChangeListener() // anonymous inner class
33     {
34         // handle change in slider value
35         @Override
36         public void stateChanged(ChangeEvent e)
37         {
38             myPanel.setDiameter(diameterJSlider.getValue());
39         }
40     });
41
42
43     add(diameterJSlider, BorderLayout.SOUTH);
44     add(myPanel, BorderLayout.CENTER);
45 }
46 } // end class SliderFrame

```

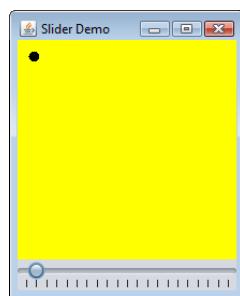
Fig. 35.3 | JSlider value used to determine the diameter of a circle. (Part 2 of 2.)

```

1 // Fig. 22.4: SliderDemo.java
2 // Testing SliderFrame.
3 import javax.swing.JFrame;
4
5 public class SliderDemo
6 {
7     public static void main(String[] args)
8     {
9         SliderFrame sliderFrame = new SliderFrame();
10        sliderFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        sliderFrame.setSize(220, 270);
12        sliderFrame.setVisible(true);
13    }
14 } // end class SliderDemo

```

a) Initial GUI with default circle



b) GUI after the user moves the JSlider's thumb to the right

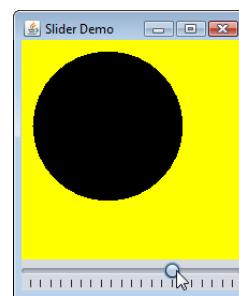


Fig. 35.4 | Test class for SliderFrame.

Class OvalPanel (Fig. 35.2) contains a `paintComponent` method (lines 12–17) that draws a filled oval (a circle in this example), a `setDiameter` method (lines 20–25) that

changes the circle's diameter and repaints the `OvalPanel`, a `getPreferredSize` method (lines 28–31) that returns the preferred width and height of an `OvalPanel` and a `getMinimumSize` method (lines 34–37) that returns an `OvalPanel`'s minimum width and height. Methods `getPreferredSize` and `getMinimumSize` are used by some layout managers to determine the size of a component.

Class `SliderFrame` (Fig. 35.3) creates the `JSlider` that controls the diameter of the circle. Class `SliderFrame`'s constructor (lines 17–45) creates `OvalPanel` object `myPanel` (line 21) and sets its background color (line 22). Lines 25–26 create `JSlider` object `diameterJSlider` to control the diameter of the circle drawn on the `OvalPanel`. The `JSlider` constructor takes four arguments. The first specifies the orientation of `diameterJSlider`, which is `HORIZONTAL` (a constant in interface `SwingConstants`). The second and third arguments indicate the minimum and maximum integer values in the range of values for this `JSlider`. The last argument indicates that the initial value of the `JSlider` (i.e., where the thumb is displayed) should be 10.

Lines 27–28 customize the appearance of the `JSlider`. Method `setMajorTickSpacing` indicates that each major tick mark represents 10 values in the range of values supported by the `JSlider`. Method `setPaintTicks` with a `true` argument indicates that the tick marks should be displayed (they aren't displayed by default). For other methods that are used to customize a `JSlider`'s appearance, see the `JSlider` online documentation (docs.oracle.com/javase/8/docs/api/javax/swing/JSlider.html).

`JSliders` generate `ChangeEvent`s (package `javax.swing.event`) in response to user interactions. An object of a class that implements interface `ChangeListener` (package `javax.swing.event`) and declares method `stateChanged` can respond to `ChangeEvent`s. Lines 31–41 register a `ChangeListener` to handle `diameterJSlider`'s events. When method `stateChanged` (lines 35–39) is called in response to a user interaction, line 38 calls `myPanel`'s `setDiameter` method and passes the current value of the `JSlider` as an argument. `JSlider` method `getValue` returns the current thumb position.

35.3 Understanding Windows in Java

A `JFrame` is a **window** with a **title bar** and a **border**. Class `JFrame` is a subclass of `Frame` (package `java.awt`), which is a subclass of `Window` (package `java.awt`). As such, `JFrame` is one of the *heavyweight* Swing GUI components. When you display a window from a Java program, the window is provided by the local platform's windowing toolkit, and therefore the window will look like every other window displayed on that platform. When a Java application executes on a Macintosh and displays a window, the window's title bar and borders will look like those of other Macintosh applications. When a Java application executes on a Microsoft Windows system and displays a window, the window's title bar and borders will look like those of other Microsoft Windows applications. And when a Java application executes on a UNIX platform and displays a window, the window's title bar and borders will look like those of other UNIX applications on that platform.

Returning Window Resources to the System

By default, when the user closes a `JFrame` window, it's hidden (i.e., removed from the screen), but you can control this with `JFrame` method `setDefaultCloseOperation`. Interface `WindowConstants` (package `javax.swing`), which class `JFrame` implements, declares three constants—`DISPOSE_ON_CLOSE`, `DO NOTHING_ON_CLOSE` and `HIDE_ON_CLOSE` (the de-

fault)—for use with this method. Some platforms allow only a limited number of windows to be displayed on the screen. Thus, a window is a valuable resource that should be given back to the system when it's no longer needed. Class `Window` (an indirect superclass of `JFrame`) declares method `dispose` for this purpose. When a `Window` is no longer needed in an application, you should explicitly dispose of it. This can be done by calling the `Window`'s `dispose` method or by calling method `setDefaultCloseOperation` with the argument `WindowConstants.DISPOSE_ON_CLOSE`. Terminating an application also returns window resources to the system. Using `DO NOTHING ON CLOSE` indicates that the program will determine what to do when the user attempts to close the window. For example, the program might want to ask whether to save a file's changes before closing a window.

Displaying and Positioning Windows

By default, a window is not displayed on the screen until the program invokes the window's `setVisible` method (inherited from class `java.awt.Component`) with a `true` argument. A window's size should be set with a call to method `setSize` (inherited from class `java.awt.Component`). The position of a window when it appears on the screen is specified with method `setLocation` (inherited from class `java.awt.Component`).

Window Events

When the user manipulates the window, this action generates **window events**. Event listeners are registered for window events with `Window` method `addWindowListener`. The `WindowListener` interface provides seven window-event-handling methods—`windowActivated` (called when the user makes a window the active window), `windowClosed` (called after the window is closed), `windowClosing` (called when the user initiates closing of the window), `windowDeactivated` (called when the user makes another window the active window), `windowDeiconified` (called when the user restores a minimized window), `windowIconified` (called when the user minimizes a window) and `windowOpened` (called when a program first displays a window on the screen).

35.4 Using Menus with Frames

Menus are an integral part of GUIs. They allow the user to perform actions without unnecessarily cluttering a GUI with extra components. In Swing GUIs, menus can be attached only to objects of the classes that provide method `setJMenuBar`. Two such classes are `JFrame` and `JApplet`. The classes used to declare menus are `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` and class `JRadioButtonMenuItem`.



Look-and-Feel Observation 35.1

Menus simplify GUIs because components can be hidden within them. These components will be visible only when the user looks for them by selecting the menu.

Overview of Several Menu-Related Components

Class `JMenuBar` (a subclass of `JComponent`) contains the methods necessary to manage a **menu bar**, which is a container for menus. Class `JMenu` (a subclass of `javax.swing.JMenuItem`) contains the methods necessary for managing menus. Menus contain menu items and are added to menu bars or to other menus as submenus. When a menu is clicked, it expands to show its list of menu items.

Class **JMenuItem** (a subclass of `javax.swing.AbstractButton`) contains the methods necessary to manage **menu items**. A menu item is a GUI component inside a menu that, when selected, causes an action event. A menu item can be used to initiate an action, or it can be a **submenu** that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class **JCheckBoxMenuItem** (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off. When a `JCheckBoxMenuItem` is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check is removed.

Class **JRadioButtonMenuItem** (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off like `JCheckBoxMenuItem`s. When multiple `JRadioButtonMenuItem`s are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time. When a `JRadioButtonMenuItem` is selected, a filled circle appears to the left of the menu item. When another `JRadioButtonMenuItem` is selected, the filled circle of the previously selected menu item is removed.

Using Menus in an Application

Figures 35.5–35.6 demonstrate various menu items and how to specify special characters called **mnemonics** that can provide quick access to a menu or menu item from the keyboard. Mnemonics can be used with all subclasses of `javax.swing.AbstractButton`. Class `MenuFrame` (Fig. 35.5) creates the GUI and handles the menu-item events. Most of the code in this application appears in the class's constructor (lines 34–151).

```

1 // Fig. 22.5: MenuFrame.java
2 // Demonstrating menus.
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.BorderLayout;
6 import java.awt.event.ActionListener;
7 import java.awt.event.ActionEvent;
8 import java.awt.event.ItemListener;
9 import java.awt.event.ItemEvent;
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MenuFrame extends JFrame
22 {
23     private final Color[] colorValues =
24         {Color.BLACK, Color.BLUE, Color.RED, Color.GREEN};

```

Fig. 35.5 | JMenus and mnemonics. (Part 1 of 5.)

```
25  private final JRadioButtonMenuItem[] colorItems; // color menu items
26  private final JRadioButtonMenuItem[] fonts; // font menu items
27  private final JCheckBoxMenuItem[] styleItems; // font style menu items
28  private final JLabel displayJLabel; // displays sample text
29  private final ButtonGroup fontButtonGroup; // manages font menu items
30  private final ButtonGroup colorButtonGroup; // manages color menu items
31  private int style; // used to create style for font
32
33  // no-argument constructor set up GUI
34  public MenuFrame()
35  {
36      super("Using JMenus");
37
38      JMenu fileMenu = new JMenu("File"); // create file menu
39      fileMenu.setMnemonic('F'); // set mnemonic to F
40
41      // create About... menu item
42      JMenuItem aboutItem = new JMenuItem("About...");
43      aboutItem.setMnemonic('A'); // set mnemonic to A
44      fileMenu.add(aboutItem); // add about item to file menu
45      aboutItem.addActionListener(
46          new ActionListener() // anonymous inner class
47          {
48              // display message dialog when user selects About...
49              @Override
50              public void actionPerformed(ActionEvent event)
51              {
52                  JOptionPane.showMessageDialog(MenuFrame.this,
53                      "This is an example\nof using menus",
54                      "About", JOptionPane.PLAIN_MESSAGE);
55              }
56          }
57      );
58
59      JMenuItem exitItem = new JMenuItem("Exit"); // create exit item
60      exitItem.setMnemonic('x'); // set mnemonic to x
61      fileMenu.add(exitItem); // add exit item to file menu
62      exitItem.addActionListener(
63          new ActionListener() // anonymous inner class
64          {
65              // terminate application when user clicks exitItem
66              @Override
67              public void actionPerformed(ActionEvent event)
68              {
69                  System.exit(0); // exit application
70              }
71          }
72      );
73
74      JMenuBar bar = new JMenuBar(); // create menu bar
75      setJMenuBar(bar); // add menu bar to application
76      bar.add(fileMenu); // add file menu to menu bar
77
```

Fig. 35.5 | JMenus and mnemonics. (Part 2 of 5.)

```

78     JMenu formatMenu = new JMenu("Format"); // create format menu
79     formatMenu.setMnemonic('r'); // set mnemonic to r
80
81     // array listing string colors
82     String[] colors = { "Black", "Blue", "Red", "Green" };
83
84     JMenu colorMenu = new JMenu("Color"); // create color menu
85     colorMenu.setMnemonic('C'); // set mnemonic to C
86
87     // create radio button menu items for colors
88     colorItems = new JRadioButtonMenuItem[colors.length];
89     colorButtonGroup = new ButtonGroup(); // manages colors
90     ItemHandler itemHandler = new ItemHandler(); // handler for colors
91
92     // create color radio button menu items
93     for (int count = 0; count < colors.length; count++)
94     {
95         colorItems[count] =
96             new JRadioButtonMenuItem(colors[count]); // create item
97         colorMenu.add(colorItems[count]); // add item to color menu
98         colorButtonGroup.add(colorItems[count]); // add to group
99         colorItems[count].addActionListener(itemHandler);
100    }
101
102    colorItems[0].setSelected(true); // select first Color item
103
104    formatMenu.add(colorMenu); // add color menu to format menu
105    formatMenu.addSeparator(); // add separator in menu
106
107    // array listing font names
108    String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
109    JMenu fontMenu = new JMenu("Font"); // create font menu
110    fontMenu.setMnemonic('n'); // set mnemonic to n
111
112    // create radio button menu items for font names
113    fonts = new JRadioButtonMenuItem[fontNames.length];
114    fontButtonGroup = new ButtonGroup(); // manages font names
115
116    // create Font radio button menu items
117    for (int count = 0; count < fonts.length; count++)
118    {
119        fonts[count] = new JRadioButtonMenuItem(fontNames[count]);
120        fontMenu.add(fonts[count]); // add font to font menu
121        fontButtonGroup.add(fonts[count]); // add to button group
122        fonts[count].addActionListener(itemHandler); // add handler
123    }
124
125    fonts[0].setSelected(true); // select first Font menu item
126    fontMenu.addSeparator(); // add separator bar to font menu
127
128    String[] styleNames = { "Bold", "Italic" }; // names of styles
129    styleItems = new JCheckBoxMenuItem[styleNames.length];
130    StyleHandler styleHandler = new StyleHandler(); // style handler

```

Fig. 35.5 | JMenus and mnemonics. (Part 3 of 5.)

```
I31     // create style checkbox menu items
I32     for (int count = 0; count < styleNames.length; count++)
I33     {
I34         styleItems[count] =
I35             new JCheckBoxMenuItem(styleNames[count]); // for style
I36             fontMenu.add(styleItems[count]); // add to font menu
I37             styleItems[count].addItemListener(styleHandler); // handler
I38         }
I39     }
I40
I41     formatMenu.add(fontMenu); // add Font menu to Format menu
I42     bar.add(formatMenu); // add Format menu to menu bar
I43
I44     // set up label to display text
I45     displayJLabel = new JLabel("Sample Text", SwingConstants.CENTER);
I46     displayJLabel.setForeground(colorValues[0]);
I47     displayJLabel.setFont(new Font("Serif", Font.PLAIN, 72));
I48
I49     getContentPane().setBackground(Color.CYAN); // set background
I50     add(displayJLabel, BorderLayout.CENTER); // add displayJLabel
I51 } // end MenuFrame constructor
I52
I53 // inner class to handle action events from menu items
I54 private class ItemHandler implements ActionListener
I55 {
I56     // process color and font selections
I57     @Override
I58     public void actionPerformed(ActionEvent event)
I59     {
I60         // process color selection
I61         for (int count = 0; count < colorItems.length; count++)
I62         {
I63             if (colorItems[count].isSelected())
I64             {
I65                 displayJLabel.setForeground(colorValues[count]);
I66                 break;
I67             }
I68         }
I69
I70         // process font selection
I71         for (int count = 0; count < fonts.length; count++)
I72         {
I73             if (event.getSource() == fonts[count])
I74             {
I75                 displayJLabel.setFont(
I76                     new Font(fonts[count].getText(), style, 72));
I77             }
I78         }
I79
I80         repaint(); // redraw application
I81     }
I82 } // end class ItemHandler
I83
```

Fig. 35.5 | JMenus and mnemonics. (Part 4 of 5.)

```

184     // inner class to handle item events from checkbox menu items
185     private class StyleHandler implements ItemListener
186     {
187         // process font style selections
188         @Override
189         public void itemStateChanged(ItemEvent e)
190         {
191             String name = displayJLabel.getFont().getName(); // current Font
192             Font font; // new font based on user selections
193
194             // determine which items are checked and create Font
195             if (styleItems[0].isSelected() &&
196                 styleItems[1].isSelected())
197                 font = new Font(name, Font.BOLD + Font.ITALIC, 72);
198             else if (styleItems[0].isSelected())
199                 font = new Font(name, Font.BOLD, 72);
200             else if (styleItems[1].isSelected())
201                 font = new Font(name, Font.ITALIC, 72);
202             else
203                 font = new Font(name, Font.PLAIN, 72);
204
205             displayJLabel.setFont(font);
206             repaint(); // redraw application
207         }
208     }
209 } // end class MenuFrame

```

Fig. 35.5 | JMenus and mnemonics. (Part 5 of 5.)

```

1 // Fig. 22.6: MenuTest.java
2 // Testing MenuFrame.
3 import javax.swing.JFrame;
4
5 public class MenuTest
6 {
7     public static void main(String[] args)
8     {
9         MenuFrame menuFrame = new MenuFrame();
10        menuFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        menuFrame.setSize(500, 200);
12        menuFrame.setVisible(true);
13    }
14 } // end class MenuTest

```

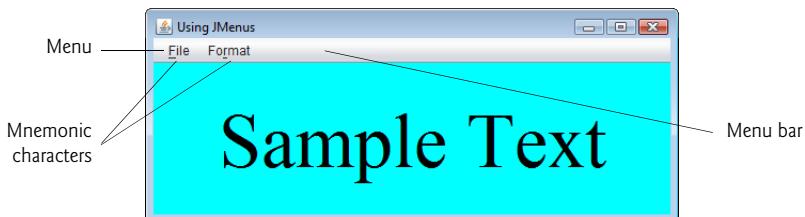


Fig. 35.6 | Test class for MenuFrame. (Part 1 of 2.)

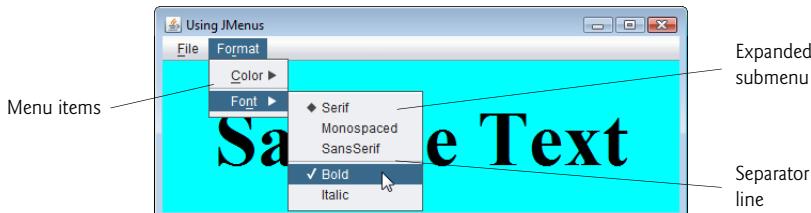


Fig. 35.6 | Test class for MenuFrame. (Part 2 of 2.)

Setting Up the File Menu

Lines 38–76 set up the `File` menu and attach it to the menu bar. The `File` menu contains an `About...` menu item that displays a message dialog when the menu item is selected and an `Exit` menu item that can be selected to terminate the application. Line 38 creates a `JMenu` and passes to the constructor the string "`File`" as the name of the menu. Line 39 uses `JMenu` method `setMnemonic` (inherited from class `AbstractButton`) to indicate that `F` is the mnemonic for this menu. Pressing the `Alt` key and the letter `F` opens the menu, just as clicking the menu name with the mouse would. In the GUI, the mnemonic character in the menu's name is displayed with an underline. (See the screen captures in Fig. 35.6.)



Look-and-Feel Observation 35.2

Mnemonics provide quick access to menu commands and button commands through the keyboard.



Look-and-Feel Observation 35.3

Different mnemonics should be used for each button or menu item. Normally, the first letter in the label on the menu item or button is used as the mnemonic. If several buttons or menu items start with the same letter, choose the next most prominent letter in the name (e.g., `x` is commonly chosen for an `Exit` button or menu item). Mnemonics are case insensitive.

Lines 42–43 create `JMenuItem` `aboutItem` with the text “`About...`” and set its mnemonic to the letter `A`. This menu item is added to `fileMenu` at line 44 with `JMenu` method `add`. To access the `About...` menu item through the keyboard, press the `Alt` key and letter `F` to open the `File` menu, then press `A` to select the `About...` menu item. Lines 46–56 create an `ActionListener` to process `aboutItem`'s action event. Lines 52–54 display a message dialog box. In most prior uses of `showMessageDialog`, the first argument was `null`. The purpose of the first argument is to specify the `parent window` that helps determine where the dialog box will be displayed. If the parent window is specified as `null`, the dialog box appears in the center of the screen. Otherwise, it appears centered over the specified parent window. In this example, the program specifies the parent window with `MenuFrame.this`—the `this` reference of the `MenuFrame` object. When using the `this` reference in an inner class, specifying `this` by itself refers to the inner-class object. To reference the outer-class object's `this` reference, qualify `this` with the outer-class name and a dot `(.)`.

Recall that dialog boxes are typically modal. A modal dialog box does not allow any other window in the application to be accessed until the dialog box is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class `JDialog` can be used to create your own modal or nonmodal dialogs.

Lines 59–72 create menu item `exitItem`, set its mnemonic to x, add it to `fileMenu` and register an `ActionListener` that terminates the program when the user selects `exitItem`. Lines 74–76 create the `JMenuBar`, attach it to the window with `JFrame` method `setJMenuBar` and use `JMenuBar` method `add` to attach the `fileMenu` to the `JMenuBar`.



Look-and-Feel Observation 35.4

Menus appear left to right in the order they're added to a JMenuBar.

Setting Up the Format Menu

Lines 78–79 create the `formatMenu` and set its mnemonic to r. F is not used because that's the `File` menu's mnemonic. Lines 84–85 create `colorMenu` (this will be a submenu in `Format`) and set its mnemonic to C. Line 88 creates `JRadioButtonMenuItem` array `colorItems`, which refers to the menu items in `colorMenu`. Line 89 creates `ButtonGroup` `colorButtonGroup`, which ensures that only one of the `Color` submenu items is selected at a time. Line 90 creates an instance of inner class `ItemHandler` (declared at lines 154–181) that responds to selections from the `Color` and `Font` submenus (discussed shortly). The loop at lines 93–100 creates each `JRadioButtonMenuItem` in array `colorItems`, adds each menu item to `colorMenu` and to `colorButtonGroup` and registers the `ActionListener` for each menu item.

Line 102 invokes `AbstractButton` method `setSelected` to select the first element in array `colorItems`. Line 104 adds `colorMenu` as a submenu of `formatMenu`. Line 105 invokes `JMenu` method `addSeparator` to add a horizontal `separator` line to the menu.



Look-and-Feel Observation 35.5

A submenu is created by adding a menu as a menu item in another menu.



Look-and-Feel Observation 35.6

Separators can be added to a menu to group menu items logically.



Look-and-Feel Observation 35.7

Any JComponent can be added to a JMenu or to a JMenuBar.

Lines 108–126 create the `Font` submenu and several `JRadioButtonMenuItem`s and select the first element of `JRadioButtonMenuItem` array `fonts`. Line 129 creates a `JCheckBoxMenuItem` array to represent the menu items for specifying bold and italic styles for the fonts. Line 130 creates an instance of inner class `StyleHandler` (declared at lines 185–208) to respond to the `JCheckBoxMenuItem` events. The for statement at lines 133–139 creates each `JCheckBoxMenuItem`, adds it to `fontMenu` and registers its `ItemListener`. Line 141 adds `fontMenu` as a submenu of `formatMenu`. Line 142 adds the `formatMenu` to `bar` (the menu bar).

Creating the Rest of the GUI and Defining the Event Handlers

Lines 145–147 create a `JLabel` for which the `Format` menu items control the font, font color and font style. The initial foreground color is set to the first element of array `colorValues` (`Color.BLACK`) by invoking `JComponent` method `setForeground`. The initial font is set to `Serif` with `PLAIN` style and 72-point size. Line 149 sets the background color of

the window's content pane to cyan, and line 150 attaches the `JLabel` to the CENTER of the content pane's `BorderLayout`.

`ItemHandler` method `actionPerformed` (lines 157–181) uses two `for` statements to determine which font or color menu item generated the event and sets the font or color of the `JLabel` `displayLabel`, respectively. The `if` condition at line 163 uses `AbstractButton` method `isSelected` to determine the selected `JRadioButtonMenuItem`. The `if` condition at line 173 invokes the event object's `getSource` method to get a reference to the `JRadioButtonMenuItem` that generated the event. Line 176 invokes `AbstractButton` method `getText` to obtain the name of the font from the menu item.

`StyleHandler` method `itemStateChanged` (lines 188–207) is called if the user selects a `JCheckBoxMenuItem` in the `fontMenu`. Lines 195–203 determine which `JCheckBoxMenuItem`s are selected and use their combined state to determine the new font style.

35.5 JPopupMenu

Applications often provide **context-sensitive pop-up menus** for several reasons—they can be convenient, there might not be a menu bar and the options they display can be specific to individual on-screen components. In Swing, such menus are created with class **JPopupMenu** (a subclass of `JComponent`). These menus provide options that are specific to the component for which the **popup trigger event** occurred—on most systems, when the user presses and releases the right mouse button.



Look-and-Feel Observation 35.8

The pop-up trigger event is platform specific. On most platforms that use a mouse with multiple buttons, the pop-up trigger event occurs when the user clicks the right mouse button on a component that supports a pop-up menu.

The application in Figs. 35.7–35.8 creates a `JPopupMenu` that allows the user to select one of three colors and change the background color of the window. When the user clicks the right mouse button on the `PopupFrame` window's background, a `JPopupMenu` containing colors appears. If the user clicks a `JRadioButtonMenuItem` for a color, `ItemHandler` method `actionPerformed` changes the background color of the window's content pane.

Line 25 of the `PopupFrame` constructor (Fig. 35.7, lines 21–70) creates an instance of class `ItemHandler` (declared in lines 73–89) that will process the item events from the menu items in the pop-up menu. Line 29 creates the `JPopupMenu`. The `for` statement (lines 33–39) creates a `JRadioButtonMenuItem` object (line 35), adds it to `popupMenu` (line 36), adds it to `ButtonGroup` `colorGroup` (line 37) to maintain one selected `JRadioButtonMenuItem` at a time and registers its `ActionListener` (line 38). Line 41 sets the initial background to white by invoking method `setBackground`.

```

1 // Fig. 22.7: PopupFrame.java
2 // Demonstrating JPopupMenu.
3 import java.awt.Color;
4 import java.awt.event.MouseAdapter;
5 import java.awt.event.MouseEvent;
6 import java.awt.event.ActionListener;
```

Fig. 35.7 | `JPopupMenu` for selecting colors. (Part 1 of 3.)

```
7 import java.awt.event.ActionEvent;
8 import javax.swing.JFrame;
9 import javax.swing.JRadioButtonMenuItem;
10 import javax.swing.JPopupMenu;
11 import javax.swing.ButtonGroup;
12
13 public class PopupFrame extends JFrame
14 {
15     private final JRadioButtonMenuItem[] items; // holds items for colors
16     private final Color[] colorValues =
17         { Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
18     private final JPopupMenu popupMenu; // allows user to select color
19
20     // no-argument constructor sets up GUI
21     public PopupFrame()
22     {
23         super("Using JPopups");
24
25         ItemHandler handler = new ItemHandler(); // handler for menu items
26         String[] colors = { "Blue", "Yellow", "Red" };
27
28         ButtonGroup colorGroup = new ButtonGroup(); // manages color items
29         popupMenu = new JPopupMenu(); // create pop-up menu
30         items = new JRadioButtonMenuItem[colors.length];
31
32         // construct menu item, add to pop-up menu, enable event handling
33         for (int count = 0; count < items.length; count++)
34         {
35             items[count] = new JRadioButtonMenuItem(colors[count]);
36             popupMenu.add(items[count]); // add item to pop-up menu
37             colorGroup.add(items[count]); // add item to button group
38             items[count].addActionListener(handler); // add handler
39         }
40
41         setBackground(Color.WHITE);
42
43         // declare a MouseListener for the window to display pop-up menu
44         addMouseListener(
45             new MouseAdapter() // anonymous inner class
46             {
47                 // handle mouse press event
48                 @Override
49                 public void mousePressed(MouseEvent event)
50                 {
51                     checkForTriggerEvent(event);
52                 }
53
54                 // handle mouse release event
55                 @Override
56                 public void mouseReleased(MouseEvent event)
57                 {
58                     checkForTriggerEvent(event);
59                 }
59 }
```

Fig. 35.7 | JPopupMenu for selecting colors. (Part 2 of 3.)

```

60          // determine whether event should trigger pop-up menu
61      private void checkForTriggerEvent(MouseEvent event)
62      {
63          if (event.isPopupTrigger())
64              popupMenu.show(
65                  event.getComponent(), event.getX(), event.getY());
66          }
67      }
68  );
69  );
70 } // end PopupFrame constructor
71
72 // private inner class to handle menu item events
73 private class ItemHandler implements ActionListener
74 {
75     // process menu item selections
76     @Override
77     public void actionPerformed(ActionEvent event)
78     {
79         // determine which menu item was selected
80         for (int i = 0; i < items.length; i++)
81         {
82             if (event.getSource() == items[i])
83             {
84                 getContentPane().setBackground(colorValues[i]);
85                 return;
86             }
87         }
88     }
89 } // end private inner class ItemHandler
90 } // end class PopupFrame

```

Fig. 35.7 | JPopupMenu for selecting colors. (Part 3 of 3.)

```

1 // Fig. 22.8: PopupTest.java
2 // Testing PopupFrame.
3 import javax.swing.JFrame;
4
5 public class PopupTest
6 {
7     public static void main(String[] args)
8     {
9         PopupFrame popupFrame = new PopupFrame();
10        popupFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        popupFrame.setSize(300, 200);
12        popupFrame.setVisible(true);
13    }
14 } // end class PopupTest

```

Fig. 35.8 | Test class for PopupFrame. (Part 1 of 2.)



Fig. 35.8 | Test class for PopupFrame. (Part 2 of 2.)

Lines 44–69 register a `MouseListener` to handle the mouse events of the application window. Methods `mousePressed` (lines 48–52) and `mouseReleased` (lines 55–59) check for the pop-up trigger event. Each method calls private utility method `checkForTriggerEvent` (lines 62–67) to determine whether the pop-up trigger event occurred. If it did, `MouseEvent` method `isPopupTrigger` returns `true`, and `JPopupMenu` method `show` displays the `JPopupMenu`. The first argument to method `show` specifies the **origin component**, whose position helps determine where the `JPopupMenu` will appear on the screen. The last two arguments are the *x*-*y* coordinates (measured from the origin component's upper-left corner) at which the `JPopupMenu` is to appear.



Look-and-Feel Observation 35.9

Displaying a `JPopupMenu` for the pop-up trigger event of multiple GUI components requires registering mouse-event handlers for each of those GUI components.

When the user selects a menu item from the pop-up menu, class `ItemHandler`'s method `actionPerformed` (lines 76–88) determines which `JRadioButtonMenuItem` the user selected and sets the background color of the window's content pane.

35.6 Pluggable Look-and-Feel

A program that uses Java's AWT GUI components (package `java.awt`) takes on the look-and-feel of the platform on which the program executes. A Java application running on a macOS looks like other macOS applications, one running on Microsoft Windows looks like other Windows applications, and one running on a Linux platform looks like other applications on that Linux platform. This is sometimes desirable, because it allows users of the application on each platform to use GUI components with which they're already familiar. However, it also introduces interesting portability issues.



Portability Tip 35.1

GUI components often look different on different platforms (fonts, font sizes, component borders, etc.) and might require different amounts of space to display. This could change their layout and alignments.



Portability Tip 35.2

GUI components on different platforms have might different default functionality—e.g., not all platforms allow a button with the focus to be “pressed” with the space bar.

Swing's lightweight GUI components eliminate many of these issues by providing uniform functionality across platforms and by defining a uniform cross-platform look-and-feel. Section 26.2 introduced the *Nimbus* look-and-feel. Earlier versions of Java used the **metal look-and-feel**, which is still the default. Swing also provides the flexibility to customize the look-and-feel to appear as a Microsoft Windows-style look-and-feel (only on Windows systems), a Motif-style (UNIX) look-and-feel (across all platforms) or a Macintosh look-and-feel (only on Mac systems).

Figures 35.9–35.10 demonstrate a way to change the look-and-feel of a Swing GUI. It creates several GUI components, so you can see the change in their look-and-feel at the same time. The output windows show the Metal, Nimbus, CDE/Motif, Windows and Windows Classic look-and-feels that are available on Windows systems. The installed look-and-feels will vary by platform.

We've covered the GUI components and event-handling concepts in this example previously, so we focus here on the mechanism for changing the look-and-feel. Class **UIManager** (package `javax.swing`) contains nested class **LookAndFeelInfo** (a `public static` class) that maintains information about a look-and-feel. Line 20 (Fig. 35.9) declares an array of type `UIManager.LookAndFeelInfo` (note the syntax used to identify the `static` inner class `LookAndFeelInfo`). Line 34 uses `UIManager` `static` method `getInstalledLookAndFeels` to get the array of `UIManager.LookAndFeelInfo` objects that describe each look-and-feel available on your system.



Performance Tip 35.1

Each look-and-feel is represented by a Java class. `UIManager` method `getInstalledLookAndFeels` does not load each class. Rather, it provides the names of the available look-and-feel classes so that a choice can be made (presumably once at program start-up). This reduces the overhead of having to load all the look-and-feel classes even if the program will not use some of them.

```
1 // Fig. 22.9: LookAndFeelFrame.java
2 // Changing the look-and-feel.
3 import java.awt.GridLayout;
4 import java.awt.BorderLayout;
5 import java.awt.event.ItemListener;
6 import java.awt.event.ItemEvent;
7 import javax.swing.JFrame;
8 import javax.swing.UIManager;
9 import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JButton;
12 import javax.swing.JLabel;
13 import javax.swing.JComboBox;
14 import javax.swing.JPanel;
15 import javax.swing.SwingConstants;
16 import javax.swing.SwingUtilities;
17
18 public class LookAndFeelFrame extends JFrame
19 {
```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 1 of 3.)

```
20    private final UIManager.LookAndFeelInfo[] looks;
21    private final String[] lookNames; // Look-and-feel names
22    private final JRadioButton[] radio; // for selecting look-and-feel
23    private final ButtonGroup group; // group for radio buttons
24    private final JButton button; // displays look of button
25    private final JLabel label; // displays look of label
26    private final JComboBox<String> comboBox; // displays look of combo box
27
28    // set up GUI
29    public LookAndFeelFrame()
30    {
31        super("Look and Feel Demo");
32
33        // get installed look-and-feel information
34        looks = UIManager.getInstalledLookAndFeels();
35        lookNames = new String[looks.length];
36
37        // get names of installed look-and-feels
38        for (int i = 0; i < looks.length; i++)
39            lookNames[i] = looks[i].getName();
40
41        JPanel northPanel = new JPanel();
42        northPanel.setLayout(new GridLayout(3, 1, 0, 5));
43
44        label = new JLabel("This is a " + lookNames[0] + " look-and-feel",
45                           SwingConstants.CENTER);
46        northPanel.add(label);
47
48        button = new JButton("JButton");
49        northPanel.add(button);
50
51        comboBox = new JComboBox<String>(lookNames);
52        northPanel.add(comboBox);
53
54        // create array for radio buttons
55        radio = new JRadioButton[looks.length];
56
57        JPanel southPanel = new JPanel();
58
59        // use a GridLayout with 3 buttons in each row
60        int rows = (int) Math.ceil(radio.length / 3.0);
61        southPanel.setLayout(new GridLayout(rows, 3));
62
63        group = new ButtonGroup(); // button group for look-and-feels
64        ItemHandler handler = new ItemHandler(); // look-and-feel handler
65
66        for (int count = 0; count < radio.length; count++)
67        {
68            radio[count] = new JRadioButton(lookNames[count]);
69            radio[count].addItemListener(handler); // add handler
70            group.add(radio[count]); // add radio button to group
71            southPanel.add(radio[count]); // add radio button to panel
72        }
73    }
```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 2 of 3.)

```

73
74     add(northPanel, BorderLayout.NORTH); // add north panel
75     add(southPanel, BorderLayout.SOUTH); // add south panel
76
77     radio[0].setSelected(true); // set default selection
78 } // end LookAndFeelFrame constructor
79
80 // use UIManager to change look-and-feel of GUI
81 private void changeTheLookAndFeel(int value)
82 {
83     try // change look-and-feel
84     {
85         // set look-and-feel for this application
86         UIManager.setLookAndFeel(looks[value].getClassName());
87
88         // update components in this application
89         SwingUtilities.updateComponentTreeUI(this);
90     }
91     catch (Exception exception)
92     {
93         exception.printStackTrace();
94     }
95 }
96
97 // private inner class to handle radio button events
98 private class ItemHandler implements ItemListener
99 {
100     // process user's look-and-feel selection
101     @Override
102     public void itemStateChanged(ItemEvent event)
103     {
104         for (int count = 0; count < radio.length; count++)
105         {
106             if (radio[count].isSelected())
107             {
108                 label.setText(String.format(
109                     "This is a %s look-and-feel", lookNames[count]));
110                 comboBox.setSelectedIndex(count); // set combobox index
111                 changeTheLookAndFeel(count); // change look-and-feel
112             }
113         }
114     }
115 } // end private inner class ItemHandler
116 } // end class LookAndFeelFrame

```

Fig. 35.9 | Look-and-feel of a Swing-based GUI. (Part 3 of 3.)

```

1 // Fig. 22.10: LookAndFeelDemo.java
2 // Changing the look-and-feel.
3 import javax.swing.JFrame;
4

```

Fig. 35.10 | Test class for LookAndFeelFrame. (Part 1 of 2.)

```

5  public class LookAndFeelDemo
6  {
7      public static void main(String[] args)
8      {
9          LookAndFeelFrame lookAndFeelFrame = new LookAndFeelFrame();
10         lookAndFeelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11         lookAndFeelFrame.setSize(400, 220);
12         lookAndFeelFrame.setVisible(true);
13     }
14 } // end class LookAndFeelDemo

```

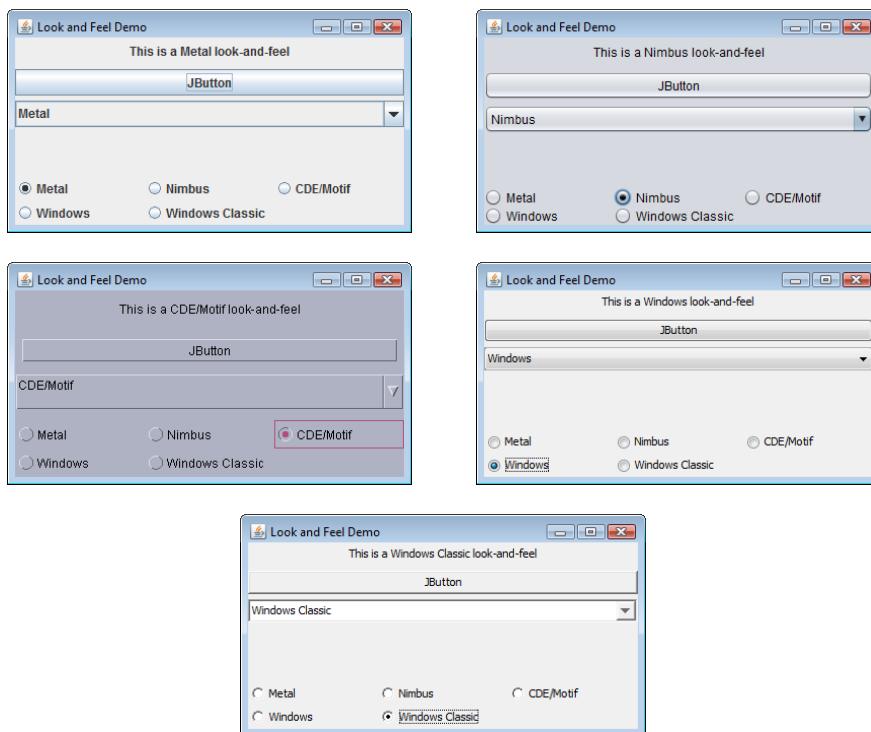


Fig. 35.10 | Test class for `LookAndFeelFrame`. (Part 2 of 2.)

Our utility method `changeTheLookAndFeel` (lines 81–95) is called by the event handler for the JRadioButtons at the bottom of the user interface. The event handler (declared in private inner class `ItemHandler` at lines 98–115) passes an integer representing the element in array `looks` that should be used to change the look-and-feel. Line 86 invokes static method `setLookAndFeel` of `UIManager` to change the look-and-feel. The `getClassName` method of class `UIManager.LookAndFeelInfo` determines the name of the look-and-feel class that corresponds to the `UIManager.LookAndFeelInfo` object. If the look-and-feel is not already loaded, it will be loaded as part of the call to `setLookAndFeel`. Line 89 invokes the static method `updateComponentTreeUI` of class `SwingUtilities` (package `javax.swing`) to change the look-and-feel of every GUI component attached to its argument (this instance of our application class `LookAndFeelFrame`) to the new look-and-feel.

35.7 JDesktopPane and JInternalFrame

A **multiple-document interface (MDI)** is a main window (called the **parent window**) containing other windows (called **child windows**) and is often used to manage several open documents. For example, many e-mail programs allow you to have several windows open at the same time, so you can compose or read multiple e-mail messages simultaneously. Similarly, many word processors allow the user to open multiple documents in separate windows within a main window, making it possible to switch between them without having to close one to open another. The application in Figs. 35.11–35.12 demonstrates Swing's **JDesktopPane** and **JInternalFrame** classes for implementing multiple-document interfaces.

```
1 // Fig. 22.11: DesktopFrame.java
2 // Demonstrating JDesktopPane.
3 import java.awt.BorderLayout;
4 import java.awt.Dimension;
5 import java.awt.Graphics;
6 import java.awt.event.ActionListener;
7 import java.awt.event.ActionEvent;
8 import java.util.Random;
9 import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class DesktopFrame extends JFrame
19 {
20     private final JDesktopPane theDesktop;
21
22     // set up GUI
23     public DesktopFrame()
24     {
25         super("Using a JDesktopPane");
26
27         JMenuBar bar = new JMenuBar();
28         JMenu addMenu = new JMenu("Add");
29         JMenuItem newFrame = new JMenuItem("Internal Frame");
30
31         addMenu.add(newFrame); // add new frame item to Add menu
32         bar.add(addMenu); // add Add menu to menu bar
33         setJMenuBar(bar); // set menu bar for this application
34
35         theDesktop = new JDesktopPane();
36         add(theDesktop); // add desktop pane to frame
37
38         // set up listener for newFrame menu item
39         newFrame.addActionListener(
```

Fig. 35.11 | Multiple-document interface. (Part I of 2.)

```
40         new ActionListener() // anonymous inner class
41     {
42         // display new internal window
43         @Override
44         public void actionPerformed(ActionEvent event)
45     {
46             // create internal frame
47             JInternalFrame frame = new JInternalFrame(
48                 "Internal Frame", true, true, true, true);
49
50             MyJPanel panel = new MyJPanel();
51             frame.add(panel, BorderLayout.CENTER);
52             frame.pack(); // set internal frame to size of contents
53
54             theDesktop.add(frame); // attach internal frame
55             frame.setVisible(true); // show internal frame
56         }
57     }
58 };
59 } // end DesktopFrame constructor
60 } // end class DesktopFrame
61
62 // class to display an ImageIcon on a panel
63 class MyJPanel extends JPanel
64 {
65     private static final SecureRandom generator = new SecureRandom();
66     private final ImageIcon picture; // image to be displayed
67     private final static String[] images = { "yellowflowers.png",
68         "purpleflowers.png", "redflowers.png", "redflowers2.png",
69         "lavenderflowers.png" };
70
71     // load image
72     public MyJPanel()
73     {
74         int randomNumber = generator.nextInt(images.length);
75         picture = new ImageIcon(images[randomNumber]); // set icon
76     }
77
78     // display ImageIcon on panel
79     @Override
80     public void paintComponent(Graphics g)
81     {
82         super.paintComponent(g);
83         picture.paintIcon(this, g, 0, 0); // display icon
84     }
85
86     // return image dimensions
87     public Dimension getPreferredSize()
88     {
89         return new Dimension(picture.getIconWidth(),
90             picture.getIconHeight());
91     }
92 } // end class MyJPanel
```

Fig. 35.11 | Multiple-document interface. (Part 2 of 2.)

Lines 27–33 create a `JMenuBar`, a `JMenu` and a `JMenuItem`, add the `JMenuItem` to the `JMenu`, add the `JMenu` to the `JMenuBar` and set the `JMenuBar` for the application window. When the user selects the `JMenuItem newFrame`, the application creates and displays a new `JInternalFrame` object containing an image.

Line 35 assigns `JDesktopPane` (package `javax.swing`) variable `theDesktop` a new `JDesktopPane` object that will be used to manage the `JInternalFrame` child windows. Line 36 adds the `JDesktopPane` to the `JFrame`. By default, the `JDesktopPane` is added to the center of the content pane's `BorderLayout`, so the `JDesktopPane` expands to fill the entire application window.

```
1 // Fig. 22.12: DesktopTest.java
2 // Demonstrating JDesktopPane.
3 import javax.swing.JFrame;
4
5 public class DesktopTest
6 {
7     public static void main(String[] args)
8     {
9         DesktopFrame desktopFrame = new DesktopFrame();
10        desktopFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        desktopFrame.setSize(600, 480);
12        desktopFrame.setVisible(true);
13    }
14 } // end class DesktopTest
```

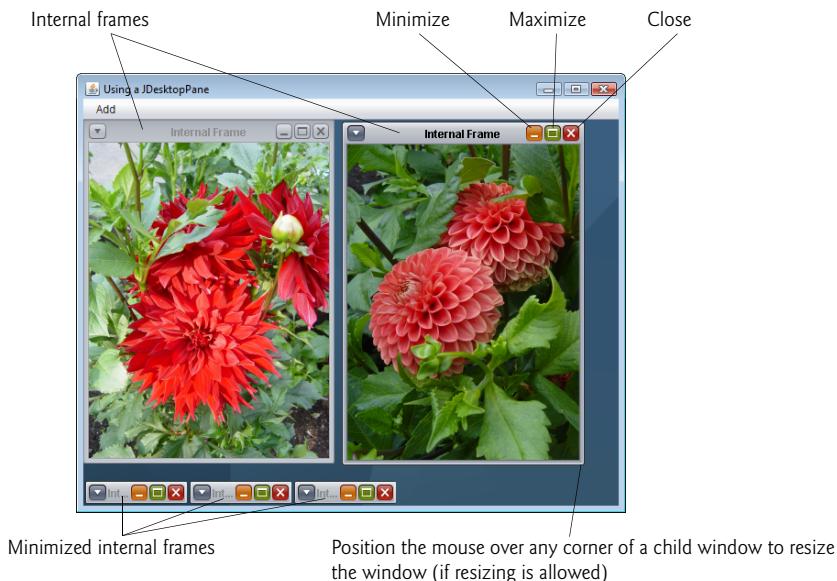


Fig. 35.12 | Test class for `DeskTopFrame`. (Part 1 of 2.)

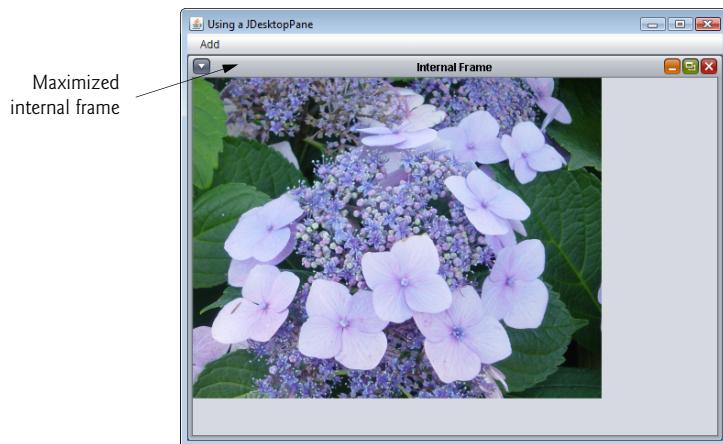


Fig. 35.12 | Test class for DeskTopFrame. (Part 2 of 2.)

Lines 39–58 register an `ActionListener` to handle the event when the user selects the `newFrame` menu item. When the event occurs, method `actionPerformed` (lines 43–56) creates a `JInternalFrame` object in lines 47–48. The `JInternalFrame` constructor used here takes five arguments—a `String` for the title bar of the internal window, a `boolean` indicating whether the internal frame can be resized by the user, a `boolean` indicating whether the internal frame can be closed by the user, a `boolean` indicating whether the internal frame can be maximized by the user and a `boolean` indicating whether the internal frame can be minimized by the user. For each of the `boolean` arguments, a `true` value indicates that the operation should be allowed (as is the case here).

As with `JFrames` and `JApplets`, a `JInternalFrame` has a content pane to which GUI components can be attached. Line 50 creates an instance of our class `MyJPanel1` (declared at lines 63–91) that is added to the `JInternalFrame` at line 51.

Line 52 uses `JInternalFrame` method `pack` to set the size of the child window. Method `pack` uses the preferred sizes of the components to determine the window's size. Class `MyJPanel1` declares method `getPreferredSize` (lines 87–91) to specify the panel's preferred size for use by the `pack` method. Line 54 adds the `JInternalFrame` to the `JDesktopPane`, and line 55 displays the `JInternalFrame`.

Classes `JInternalFrame` and `JDesktopPane` provide many methods for managing child windows. See the `JInternalFrame` and `JDesktopPane` online API documentation for complete lists of these methods:

docs.oracle.com/javase/8/docs/api/javax/swing/JInternalFrame.html
docs.oracle.com/javase/8/docs/api/javax/swing/JDesktopPane.html

35.8 JTabbedPane

A `JTabbedPane` arranges GUI components into layers, of which only one is visible at a time. Users access each layer via a tab—similar to folders in a file cabinet. When the user clicks a tab, the appropriate layer is displayed. The tabs appear at the top by default but also can be positioned at the left, right or bottom of the `JTabbedPane`. Any component

can be placed on a tab. If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab. Class `JTabbedPane` is a subclass of `JComponent`. The application in Figs. 35.13–35.14 creates one tabbed pane with three tabs. Each tab displays one of the `JPanels`—`panel1`, `panel2` or `panel3`.

```

1 // Fig. 22.13: JTabbedPaneFrame.java
2 // Demonstrating JTabbedPane.
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import javax.swing.JFrame;
6 import javax.swing.JTabbedPane;
7 import javax.swing.JLabel;
8 import javax.swing.JPanel;
9 import javax.swing.JButton;
10 import javax.swing.SwingConstants;
11
12 public class JTabbedPaneFrame extends JFrame
13 {
14     // set up GUI
15     public JTabbedPaneFrame()
16     {
17         super("JTabbedPane Demo ");
18
19         JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane
20
21         // set up panel1 and add it to JTabbedPane
22         JLabel label1 = new JLabel("panel one", SwingConstants.CENTER);
23         JPanel panel1 = new JPanel();
24         panel1.add(label1);
25         tabbedPane.addTab("Tab One", null, panel1, "First Panel");
26
27         // set up panel2 and add it to JTabbedPane
28         JLabel label2 = new JLabel("panel two", SwingConstants.CENTER);
29         JPanel panel2 = new JPanel();
30         panel2.setBackground(Color.YELLOW);
31         panel2.add(label2);
32         tabbedPane.addTab("Tab Two", null, panel2, "Second Panel");
33
34         // set up panel3 and add it to JTabbedPane
35         JLabel label3 = new JLabel("panel three");
36         JPanel panel3 = new JPanel();
37         panel3.setLayout(new BorderLayout());
38         panel3.add(new JButton("North"), BorderLayout.NORTH);
39         panel3.add(new JButton("West"), BorderLayout.WEST);
40         panel3.add(new JButton("East"), BorderLayout.EAST);
41         panel3.add(new JButton("South"), BorderLayout.SOUTH);
42         panel3.add(label3, BorderLayout.CENTER);
43         tabbedPane.addTab("Tab Three", null, panel3, "Third Panel");
44
45         add(tabbedPane); // add JTabbedPane to frame
46     }
47 } // end class JTabbedPaneFrame

```

Fig. 35.13 | `JTabbedPane` used to organize GUI components.

```

1 // Fig. 22.14: JTabbedPaneDemo.java
2 // Demonstrating JTabbedPane.
3 import javax.swing.JFrame;
4
5 public class JTabbedPaneDemo
6 {
7     public static void main(String[] args)
8     {
9         JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
10        tabbedPaneFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        tabbedPaneFrame.setSize(250, 200);
12        tabbedPaneFrame.setVisible(true);
13    }
14 } // end class JTabbedPaneDemo

```

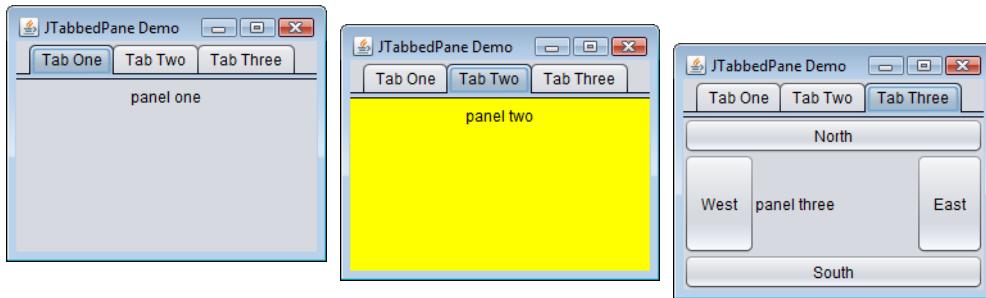


Fig. 35.14 | Test class for JTabbedPaneFrame.

The constructor (lines 15–46) builds the GUI. Line 19 creates an empty JTabbedPane with default settings—that is, tabs across the top. If the tabs do not fit on one line, they'll wrap to form additional lines of tabs. Next the constructor creates the JPanels panel11, panel12 and panel13 and their GUI components. As we set up each panel, we add it to tabbedPane, using JTabbedPane method **addTab** with four arguments. The first argument is a String that specifies the title of the tab. The second argument is an Icon reference that specifies an icon to display on the tab. If the Icon is a null reference, no image is displayed. The third argument is a Component reference that represents the GUI component to display when the user clicks the tab. The last argument is a String that specifies the tool tip for the tab. For example, line 25 adds JPanel panel11 to tabbedPane with title "Tab One" and the tool tip "First Panel". JPanel panel12 and panel13 are added to tabbedPane at lines 32 and 43. To view a tab, click it with the mouse or use the arrow keys to cycle through the tabs.

35.9 BoxLayout Layout Manager

In Chapter 26, we introduced three layout managers—FlowLayout, BorderLayout and GridLayout. This section and Section 35.10 present two additional layout managers (summarized in Fig. 35.15). We discuss them in the examples that follow.

Layout manager	Description
BoxLayout	Allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class Box declares a container that uses BoxLayout and provides static methods to create a Box with a horizontal or vertical BoxLayout.
GridBagLayout	Similar to GridLayout, but the components can vary in size and can be added in any order.

Fig. 35.15 | Additional layout managers.

The BoxLayout layout manager (in package javax.swing) arranges GUI components horizontally along a container's *x*-axis or vertically along its *y*-axis. The application in Figs. 35.16–35.17 demonstrate BoxLayout and the container class Box that uses BoxLayout as its default layout manager.

```
1 // Fig. 22.16: BoxLayoutFrame.java
2 // Demonstrating BoxLayout.
3 import java.awt.Dimension;
4 import javax.swing.JFrame;
5 import javax.swing.Box;
6 import javax.swing.JButton;
7 import javax.swing.BoxLayout;
8 import javax.swing.JPanel;
9 import javax.swing.JTabbedPane;
10
11 public class BoxLayoutFrame extends JFrame
12 {
13     // set up GUI
14     public BoxLayoutFrame()
15     {
16         super("Demonstrating BoxLayout");
17
18         // create Box containers with BoxLayout
19         Box horizontal1 = Box.createHorizontalBox();
20         Box vertical1 = Box.createVerticalBox();
21         Box horizontal2 = Box.createHorizontalBox();
22         Box vertical2 = Box.createVerticalBox();
23
24         final int SIZE = 3; // number of buttons on each Box
25
26         // add buttons to Box horizontal1
27         for (int count = 0; count < SIZE; count++)
28             horizontal1.add(new JButton("Button " + count));
29
30         // create strut and add buttons to Box vertical1
31         for (int count = 0; count < SIZE; count++)
32         {
33             vertical1.add(Box.createVerticalStrut(25));
34             vertical1.add(new JButton("Button " + count));
35         }
```

Fig. 35.16 | BoxLayout layout manager. (Part 1 of 2.)

```
36
37     // create horizontal glue and add buttons to Box horizontal12
38     for (int count = 0; count < SIZE; count++)
39     {
40         horizontal12.add(Box.createHorizontalGlue());
41         horizontal12.add(new JButton("Button " + count));
42     }
43
44     // create rigid area and add buttons to Box vertical12
45     for (int count = 0; count < SIZE; count++)
46     {
47         vertical12.add(Box.createRigidArea(new Dimension(12, 8)));
48         vertical12.add(new JButton("Button " + count));
49     }
50
51     // create vertical glue and add buttons to panel
52     JPanel panel = new JPanel();
53     panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
54
55     for (int count = 0; count < SIZE; count++)
56     {
57         panel.add(Box.createGlue());
58         panel.add(new JButton("Button " + count));
59     }
60
61     // create a JTabbedPane
62     JTabbedPane tabs = new JTabbedPane(
63         JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT);
64
65     // place each container on tabbed pane
66     tabs.addTab("Horizontal Box", horizontal11);
67     tabs.addTab("Vertical Box with Struts", vertical11);
68     tabs.addTab("Horizontal Box with Glue", horizontal12);
69     tabs.addTab("Vertical Box with Rigid Areas", vertical12);
70     tabs.addTab("Vertical Box with Glue", panel);
71
72     add(tabs); // place tabbed pane on frame
73 } // end BoxLayoutFrame constructor
74 } // end class BoxLayoutFrame
```

Fig. 35.16 | BoxLayout layout manager. (Part 2 of 2.)

```
1 // Fig. 22.17: BoxLayoutDemo.java
2 // Demonstrating BoxLayout.
3 import javax.swing.JFrame;
4
5 public class BoxLayoutDemo
6 {
7     public static void main(String[] args)
8     {
9         BoxLayoutFrame boxLayoutFrame = new BoxLayoutFrame();
```

Fig. 35.17 | Test class for BoxLayoutFrame. (Part 1 of 2.)

```

10     boxLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11     boxLayoutFrame.setSize(400, 220);
12     boxLayoutFrame.setVisible(true);
13 }
14 } // end class BoxLayoutDemo

```



Fig. 35.17 | Test class for `BoxLayoutFrame`. (Part 2 of 2.)

Creating Box Containers

Lines 19–22 create Box containers. References `horizontal11` and `horizontal12` are initialized with static Box method `createHorizontalBox`, which returns a Box container with a horizontal BoxLayout in which GUI components are arranged left-to-right. Variables `vertical11` and `vertical12` are initialized with static Box method `createVerticalBox`, which returns references to Box containers with a vertical BoxLayout in which GUI components are arranged top-to-bottom.

Struts

The loop at lines 27–28 adds three `JButtons` to `horizontal11`. The for statement at lines 31–35 adds three `JButtons` to `vertical11`. Before adding each button, line 33 adds a **vertical strut** to the container with static Box method `createVerticalStrut`. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. The int argument to method `createVerticalStrut` determines the height of the strut in pixels. When the container is resized, the distance between GUI components separated by struts does not change. Class Box also declares method `createHorizontalStrut` for horizontal BoxLayouts.

Glue

The for statement at lines 38–42 adds three JButtons to horizontal12. Before adding each button, line 40 adds **horizontal glue** to the container with static Box method **createHorizontalGlue**. Horizontal glue is an invisible GUI component that can be used between fixed-size GUI components to occupy additional space. Normally, extra space appears to the right of the last horizontal GUI component or below the last vertical one in a BoxLayout. Glue allows the extra space to be placed between GUI components. When the container is resized, components separated by glue components remain the same size, but the glue stretches or contracts to occupy the space between them. Class Box also declares method **createVerticalGlue** for vertical BoxLayouts.

Rigid Areas

The for statement at lines 45–49 adds three JButtons to vertical12. Before each button is added, line 47 adds a **rigid area** to the container with static Box method **createRigidArea**. A rigid area is an invisible GUI component that always has a fixed pixel width and height. The argument to method **createRigidArea** is a Dimension object that specifies the area's width and height.

Setting a BoxLayout for a Container

Lines 52–53 create a JPanel object and set its layout to a BoxLayout in the conventional manner, using Container method **setLayout**. The BoxLayout constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (**BoxLayout.X_AXIS**) or vertical (**BoxLayout.Y_AXIS**).

Adding Glue and JButtons

The for statement at lines 55–59 adds three JButtons to panel1. Before adding each button, line 57 adds a glue component to the container with static Box method **createGlue**. This component expands or contracts based on the size of the Box.

Creating the JTabbedPane

Lines 62–63 create a JTabbedPane to display the five containers in this program. The argument **JTabbedPane.TOP** sent to the constructor indicates that the tabs should appear at the top of the JTabbedPane. The argument **JTabbedPane.SCROLL_TAB_LAYOUT** specifies that the tabs should wrap to a new line if there are too many to fit on one line.

Attaching the Box Containers and JPanel to the JTabbedPane

The Box containers and the JPanel are attached to the JTabbedPane at lines 66–70. Try executing the application. When the window appears, resize the window to see how the glue components, strut components and rigid area affect the layout on each tab.

35.10 GridBagLayout Layout Manager

One of the most powerful predefined layout managers is **GridBagLayout** (in package `java.awt`). This layout is similar to GridLayout in that it arranges components in a grid, but it's more flexible. The components can vary in size (i.e., they can occupy multiple rows and columns) and can be added in any order.

The first step in using GridBagLayout is determining the appearance of the GUI. For this step you need only a piece of paper. Draw the GUI, then draw a grid over it, dividing

the components into rows and columns. The initial row and column numbers should be 0, so that the `GridBagLayout` layout manager can use the row and column numbers to properly place the components in the grid. Figure 35.18 demonstrates drawing the lines for the rows and columns over a GUI.

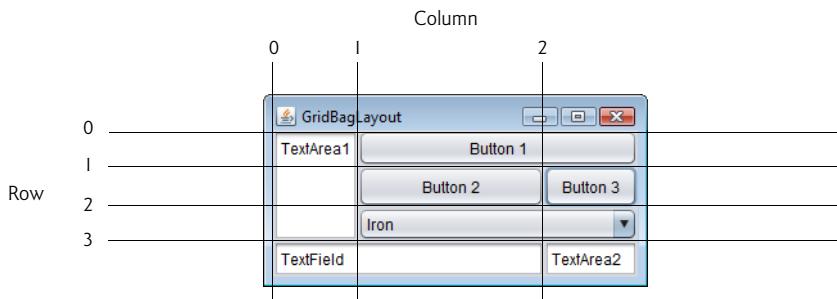


Fig. 35.18 | Designing a GUI that will use `GridBagLayout`.

GridBagConstraints

A `GridBagConstraints` object describes how a component is placed in a `GridBagLayout`. Several `GridBagConstraints` fields are summarized in Fig. 35.19.

Field	Description
<code>anchor</code>	Specifies the relative position (NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST, CENTER) of the component in an area that it does not fill.
<code>fill</code>	Resizes the component in the specified direction (NONE, HORIZONTAL, VERTICAL, BOTH) when the display area is larger than the component.
<code>gridx</code>	The column in which the component will be placed.
<code>gridy</code>	The row in which the component will be placed.
<code>gridwidth</code>	The number of columns the component occupies.
<code>gridheight</code>	The number of rows the component occupies.
<code>weightx</code>	The amount of extra space to allocate horizontally. The grid slot can become wider when extra space is available.
<code>weighty</code>	The amount of extra space to allocate vertically. The grid slot can become taller when extra space is available.

Fig. 35.19 | `GridBagConstraints` fields.

GridBagConstraints Field anchor

`GridBagConstraints` field `anchor` specifies the relative position of the component in an area that it does not fill. The variable `anchor` is assigned one of the following `GridBagConstraints` constants: **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST** or **CENTER**. The default value is **CENTER**.

GridBagConstraints Field fill

GridBagConstraints field `fill` defines how the component grows if the area in which it can be displayed is larger than the component. The variable `fill` is assigned one of the following GridBagConstraints constants: **NONE**, **VERTICAL**, **HORIZONTAL** or **BOTH**. The default value is **NONE**, which indicates that the component will not grow in either direction. **VERTICAL** indicates that it will grow vertically. **HORIZONTAL** indicates that it will grow horizontally. **BOTH** indicates that it will grow in both directions.

GridBagConstraints Fields gridx and gridy

Variables `gridx` and `gridy` specify where the upper-left corner of the component is placed in the grid. Variable `gridx` corresponds to the column, and variable `gridy` corresponds to the row. In Fig. 35.18, the JComboBox (displaying “Iron”) has a `gridx` value of 1 and a `gridy` value of 2.

GridBagConstraints Field gridwidth

Variable `gridwidth` specifies the number of columns a component occupies. The JComboBox occupies two columns. Variable `gridheight` specifies the number of rows a component occupies. The JTextArea on the left side of Fig. 35.18 occupies three rows.

GridBagConstraints Field weightx

Variable `weightx` specifies how to distribute extra horizontal space to grid slots in a GridLayout when the container is resized. A zero value indicates that the grid slot does not grow horizontally on its own. However, if the component spans a column containing a component with nonzero `weightx` value, the component with zero `weightx` value will grow horizontally in the same proportion as the other component(s) in that column. This is because each component must be maintained in the same row and column in which it was originally placed.

GridBagConstraints Field weighty

Variable `weighty` specifies how to distribute extra vertical space to grid slots in a GridLayout when the container is resized. A zero value indicates that the grid slot does not grow vertically on its own. However, if the component spans a row containing a component with nonzero `weighty` value, the component with zero `weighty` value grows vertically in the same proportion as the other component(s) in the same row.

Effects of weightx and weighty

In Fig. 35.18, the effects of `weighty` and `weightx` cannot easily be seen until the container is resized and additional space becomes available. Components with larger weight values occupy more of the additional space than those with smaller weight values.

Components should be given nonzero positive weight values—otherwise they’ll “huddle” together in the middle of the container. Figure 35.20 shows the GUI of Fig. 35.18 with all weights set to zero.

Demonstrating GridBagLayout

The application in Figs. 35.21–35.22 uses the `GridBagLayout` layout manager to arrange the components of the GUI in Fig. 35.18. The application does nothing except demonstrate how to use `GridBagLayout`.



Fig. 35.20 | GridBagLayout with the weights set to zero.

```
1 // Fig. 22.21: GridBagFrame.java
2 // Demonstrating GridBagLayout.
3 import java.awt.GridBagLayout;
4 import java.awt.GridBagConstraints;
5 import java.awt.Component;
6 import javax.swing.JFrame;
7 import javax.swing.JTextArea;
8 import javax.swing.JTextField;
9 import javax.swing.JButton;
10 import javax.swing.JComboBox;
11
12 public class GridBagFrame extends JFrame
13 {
14     private final GridBagLayout layout; // layout of this frame
15     private final GridBagConstraints constraints; // layout's constraints
16
17     // set up GUI
18     public GridBagFrame()
19     {
20         super("GridBagLayout");
21         layout = new GridBagLayout();
22         setLayout(layout); // set frame layout
23         constraints = new GridBagConstraints(); // instantiate constraints
24
25         // create GUI components
26         JTextArea textArea1 = new JTextArea("TextArea1", 5, 10);
27         JTextArea textArea2 = new JTextArea("TextArea2", 2, 2);
28
29         String[] names = { "Iron", "Steel", "Brass" };
30         JComboBox<String> comboBox = new JComboBox<String>(names);
31
32         JTextField textField = new JTextField("TextField");
33         JButton button1 = new JButton("Button 1");
34         JButton button2 = new JButton("Button 2");
35         JButton button3 = new JButton("Button 3");
36
37         // weightx and weighty for textArea1 are both 0: the default
38         // anchor for all components is CENTER: the default
39         constraints.fill = GridBagConstraints.BOTH;
40         addComponent(textArea1, 0, 0, 1, 3);
```

Fig. 35.21 | GridBagLayout layout manager. (Part I of 2.)

```
41 // weightx and weighty for button1 are both 0: the default
42 constraints.fill = GridBagConstraints.HORIZONTAL;
43 addComponent(button1, 0, 1, 2, 1);
44
45 // weightx and weighty for comboBox are both 0: the default
46 // fill is HORIZONTAL
47 addComponent(comboBox, 2, 1, 2, 1);
48
49 // button2
50 constraints.weightx = 1000; // can grow wider
51 constraints.weighty = 1; // can grow taller
52 constraints.fill = GridBagConstraints.BOTH;
53 addComponent(button2, 1, 1, 1, 1);
54
55 // fill is BOTH for button3
56 constraints.weightx = 0;
57 constraints.weighty = 0;
58 addComponent(button3, 1, 2, 1, 1);
59
60 // weightx and weighty for textField are both 0, fill is BOTH
61 addComponent(textField, 3, 0, 2, 1);
62
63 // weightx and weighty for textArea2 are both 0, fill is BOTH
64 addComponent(textArea2, 3, 2, 1, 1);
65
66 } // end GridBagFrame constructor
67
68 // method to set constraints on
69 private void addComponent(Component component,
70 int row, int column, int width, int height)
71 {
72     constraints.gridx = column;
73     constraints.gridy = row;
74     constraints.gridwidth = width;
75     constraints.gridheight = height;
76     layout.setConstraints(component, constraints); // set constraints
77     add(component); // add component
78 }
79 } // end class GridBagFrame
```

Fig. 35.21 | GridBagLayout layout manager. (Part 2 of 2.)

```
1 // Fig. 22.22: GridBagDemo.java
2 // Demonstrating GridBagLayout.
3 import javax.swing.JFrame;
4
5 public class GridBagDemo
6 {
7     public static void main(String[] args)
8     {
9         GridBagFrame gridBagFrame = new GridBagFrame();
```

Fig. 35.22 | Test class for GridBagFrame. (Part 1 of 2.)

```
10     gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11     gridBagFrame.setSize(300, 150);
12     gridBagFrame.setVisible(true);
13 }
14 } // end class GridBagDemo
```

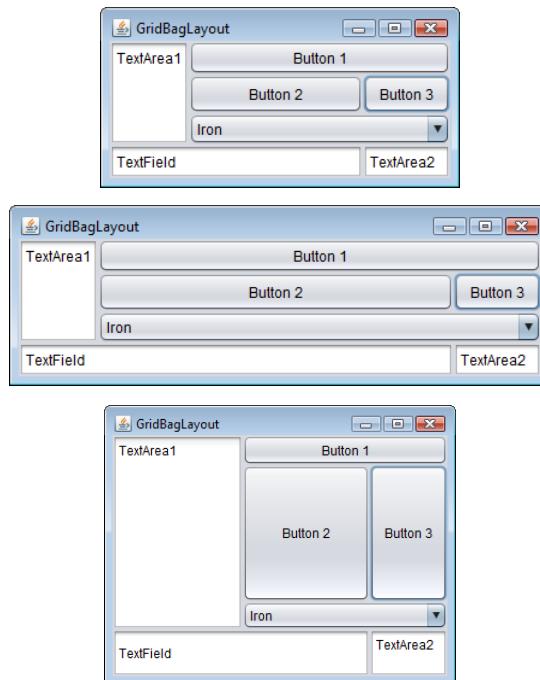


Fig. 35.22 | Test class for `GridBagFrame`. (Part 2 of 2.)

GUI Overview

The GUI contains three `JButtons`, two `JTextAreas`, a `JComboBox` and a `JTextField`. The layout manager is `GridBagLayout`. Lines 21–22 create the `GridBagLayout` object and set the layout manager for the `JFrame` to `layout`. Line 23 creates the `GridBagConstraints` object used to determine the location and size of each component in the grid. Lines 26–35 create each GUI component that will be added to the content pane.

JTextArea `textArea1`

Lines 39–40 configure `JTextArea textArea1` and add it to the content pane. The values for `weightx` and `weighty` values are not specified in `constraints`, so each has the value zero by default. Thus, the `JTextArea` will not resize itself even if space is available. However, it spans multiple rows, so the vertical size is subject to the `weighty` values of `JButtons button2` and `button3`. When either button is resized vertically based on its `weighty` value, the `JTextArea` is also resized.

Line 39 sets variable `fill` in `constraints` to `GridBagConstraints.BOTH`, causing the `JTextArea` to always fill its entire allocated area in the grid. An `anchor` value is not specified in `constraints`, so the default `CENTER` is used. We do not use variable `anchor` in this

application, so all the components will use the default. Line 40 calls our utility method `addComponent` (declared at lines 69–78). The `JTextArea` object, the row, the column, the number of columns to span and the number of rows to span are passed as arguments.

JButton button1

`JButton button1` is the next component added (lines 43–44). By default, the `weightx` and `weighty` values are still zero. The `fill` variable is set to `HORIZONTAL`—the component will always fill its area in the horizontal direction. The vertical direction is not filled. The `weighty` value is zero, so the button will become taller only if another component in the same row has a nonzero `weighty` value. `JButton button1` is located at row 0, column 1. One row and two columns are occupied.

JComboBox comboBox

`JComboBox comboBox` is the next component added (line 48). By default, the `weightx` and `weighty` values are zero, and the `fill` variable is set to `HORIZONTAL`. The `JComboBox` button will grow only in the horizontal direction. The `weightx`, `weighty` and `fill` variables retain the values set in `constraints` until they're changed. The `JComboBox` button is placed at row 2, column 1. One row and two columns are occupied.

JButton button2

`JButton button2` is the next component added (lines 51–54). It's given a `weightx` value of 1000 and a `weighty` value of 1. The area occupied by the button is capable of growing in the vertical and horizontal directions. The `fill` variable is set to `BOTH`, which specifies that the button will always fill the entire area. When the window is resized, `button2` will grow. The button is placed at row 1, column 1. One row and one column are occupied.

JButton button3

`JButton button3` is added next (lines 57–59). Both the `weightx` value and `weighty` value are set to zero, and the value of `fill` is `BOTH`. `JButton button3` will grow if the window is resized—it's affected by the weight values of `button2`. The `weightx` value for `button2` is much larger than that for `button3`. When resizing occurs, `button2` will occupy a larger percentage of the new space. The button is placed at row 1, column 2. One row and one column are occupied.

JTextField textField and JTextArea textArea2

Both the `JTextField textField` (line 62) and `JTextArea textArea2` (line 65) have a `weightx` value of 0 and a `weighty` value of 0. The value of `fill` is `BOTH`. The `JTextField` is placed at row 3, column 0, and the `JTextArea` at row 3, column 2. The `JTextField` occupies one row and two columns, the `JTextArea` one row and one column.

Method addComponent

Method `addComponent`'s parameters are a `Component` reference `component` and integers `row`, `column`, `width` and `height`. Lines 72–73 set the `GridBagConstraints` variables `gridx` and `gridy`. The `gridx` variable is assigned the column in which the `Component` will be placed, and the `gridy` value is assigned the row in which the `Component` will be placed. Lines 74–75 set the `GridBagConstraints` variables `gridwidth` and `gridheight`. The `gridwidth` variable specifies the number of columns the `Component` will span in the grid,

and the `gridheight` variable specifies the number of rows the Component will span in the grid. Line 76 sets the `GridBagConstraints` for a component in the `GridBagLayout`. Method `setConstraints` of class `GridBagLayout` takes a `Component` argument and a `GridBagConstraints` argument. Line 77 adds the component to the `JFrame`.

When you execute this application, try resizing the window to see how the constraints for each GUI component affect its position and size in the window.

GridBagConstraints Constants RELATIVE and REMAINDER

Instead of `gridx` and `gridy`, a variation of `GridBagLayout` uses `GridBagConstraints` constants `RELATIVE` and `REMAINDER`. `RELATIVE` specifies that the next-to-last component in a particular row should be placed to the right of the previous component in the row. `REMAINDER` specifies that a component is the last component in a row. Any component that is not the second-to-last or last component on a row must specify values for `GridBagConstraints` variables `gridwidth` and `gridheight`. The application in Figs. 35.23–35.24 arranges components in `GridBagLayout`, using these constants.

```

1 // Fig. 22.23: GridBagFrame2.java
2 // Demonstrating GridBagLayout constants.
3 import java.awt.GridBagLayout;
4 import java.awt.GridBagConstraints;
5 import java.awt.Component;
6 import javax.swing.JFrame;
7 import javax.swing.JComboBox;
8 import javax.swing.JTextField;
9 import javax.swing.JList;
10 import javax.swing.JButton;
11
12 public class GridBagFrame2 extends JFrame
13 {
14     private final GridBagLayout layout; // layout of this frame
15     private final GridBagConstraints constraints; // layout's constraints
16
17     // set up GUI
18     public GridBagFrame2()
19     {
20         super("GridBagLayout");
21         layout = new GridBagLayout();
22         setLayout(layout); // set frame layout
23         constraints = new GridBagConstraints(); // instantiate constraints
24
25         // create GUI components
26         String[] metals = { "Copper", "Aluminum", "Silver" };
27         JComboBox comboBox = new JComboBox(meals);
28
29         JTextField textField = new JTextField("TextField");
30
31         String[] fonts = { "Serif", "Monospaced" };
32         JList list = new JList(fonts);
33

```

Fig. 35.23 | `GridBagConstraints` constants `RELATIVE` and `REMAINDER`. (Part 1 of 2.)

```
34     String[] names = { "zero", "one", "two", "three", "four" };
35     JButton[] buttons = new JButton[names.length];
36
37     for (int count = 0; count < buttons.length; count++)
38         buttons[count] = new JButton(names[count]);
39
40     // define GUI component constraints for textField
41     constraints.weightx = 1;
42     constraints.weighty = 1;
43     constraints.fill = GridBagConstraints.BOTH;
44     constraints.gridwidth = GridBagConstraints.REMAINDER;
45     addComponent(textField);
46
47     // buttons[0] -- weightx and weighty are 1: fill is BOTH
48     constraints.gridwidth = 1;
49     addComponent(buttons[0]);
50
51     // buttons[1] -- weightx and weighty are 1: fill is BOTH
52     constraints.gridwidth = GridBagConstraints.RELATIVE;
53     addComponent(buttons[1]);
54
55     // buttons[2] -- weightx and weighty are 1: fill is BOTH
56     constraints.gridwidth = GridBagConstraints.REMAINDER;
57     addComponent(buttons[2]);
58
59     // comboBox -- weightx is 1: fill is BOTH
60     constraints.weighty = 0;
61     constraints.gridwidth = GridBagConstraints.REMAINDER;
62     addComponent(comboBox);
63
64     // buttons[3] -- weightx is 1: fill is BOTH
65     constraints.weighty = 1;
66     constraints.gridwidth = GridBagConstraints.REMAINDER;
67     addComponent(buttons[3]);
68
69     // buttons[4] -- weightx and weighty are 1: fill is BOTH
70     constraints.gridwidth = GridBagConstraints.RELATIVE;
71     addComponent(buttons[4]);
72
73     // list -- weightx and weighty are 1: fill is BOTH
74     constraints.gridwidth = GridBagConstraints.REMAINDER;
75     addComponent(list);
76 } // end GridBagFrame2 constructor
77
78 // add a component to the container
79 private void addComponent(Component component)
80 {
81     layout.setConstraints(component, constraints);
82     add(component); // add component
83 }
84 } // end class GridBagFrame2
```

Fig. 35.23 | GridBagConstraints constants RELATIVE and REMAINDER. (Part 2 of 2.)

```
1 // Fig. 22.24: GridBagDemo2.java
2 // Demonstrating GridBagLayout constants.
3 import javax.swing.JFrame;
4
5 public class GridBagDemo2
6 {
7     public static void main(String[] args)
8     {
9         GridBagConstraints gridBagFrame = new GridBagFrame2();
10        gridBagFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        gridBagFrame.setSize(300, 200);
12        gridBagFrame.setVisible(true);
13    }
14 } // end class GridBagDemo2
```

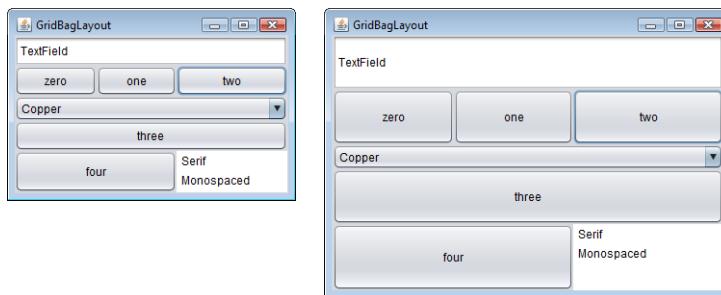


Fig. 35.24 | Test class for GridBagDemo2.

Setting the JFrame's Layout to a GridBagLayout

Lines 21–22 create a `GridBagLayout` and use it to set the `JFrame`'s layout manager. The components that are placed in `GridBagLayout` are created in lines 27–38—they are a `JComboBox`, a `JTextField`, a `JList` and five `JButtons`.

Configuring the JTextField

The `JTextField` is added first (lines 41–45). The `weightx` and `weighty` values are set to 1. The `fill` variable is set to `BOTH`. Line 44 specifies that the `JTextField` is the last component on the line. The `JTextField` is added to the content pane with a call to our utility method `addComponent` (declared at lines 79–83). Method `addComponent` takes a `Component` argument and uses `GridBagLayout` method `setConstraints` to set the constraints for the `Component`. Method `add` attaches the component to the content pane.

Configuring JButton buttons[0]

`JButton buttons[0]` (lines 48–49) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. Because `buttons[0]` is not one of the last two components on the row, it's given a `gridwidth` of 1 and so will occupy one column. The `JButton` is added to the content pane with a call to utility method `addComponent`.

Configuring JButton buttons[1]

`JButton buttons[1]` (lines 52–53) has `weightx` and `weighty` values of 1. The `fill` variable is `BOTH`. Line 52 specifies that the `JButton` is to be placed relative to the previous component. The `Button` is added to the `JFrame` with a call to `addComponent`.

Configuring JButton buttons[2]

JButton buttons[2] (lines 56–57) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the last component on the line, so REMAINDER is used. The JButton is added to the content pane with a call to addComponent.

Configuring JComboBox

The JComboBox (lines 60–62) has a weightx of 1 and a weighty of 0. The JComboBox will not grow vertically. The JComboBox is the only component on the line, so REMAINDER is used. The JComboBox is added to the content pane with a call to addComponent.

Configuring JButton buttons[3]

JButton buttons[3] (lines 65–67) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the only component on the line, so REMAINDER is used. The JButton is added to the content pane with a call to addComponent.

Configuring JButton buttons[4]

JButton buttons[4] (lines 70–71) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the next-to-last component on the line, so RELATIVE is used. The JButton is added to the content pane with a call to addComponent.

Configuring JList

The JList (lines 74–75) has weightx and weighty values of 1. The fill variable is BOTH. The JList is added to the content pane with a call to addComponent.

35.11 Wrap-Up

This chapter completes our introduction to GUIs. In this chapter, we discussed additional GUI topics, such as menus, sliders, pop-up menus, multiple-document interfaces, tabbed panes and Java’s pluggable look-and-feel. All these components can be added to existing applications to make them easier to use and understand. We also presented additional layout managers for organizing and sizing GUI components.

Summary

Section 22.2 JSlider

- JSliders (p. 2) enable you to select from a range of integer values. They can display major and minor tick marks, and labels for the tick marks (p. 2). They also support snap-to ticks (p. 3)—positioning the thumb (p. 2) between two tick marks snaps it to the closest tick mark.

- `JSliders` have either horizontal or vertical orientation. For a horizontal `JSlider`, the minimum value is at the extreme left and the maximum value at the extreme right. For a vertical `JSlider`, the minimum value is at the extreme bottom and the maximum value at the extreme top. The position of the thumb indicates the current value of the `JSlider`. Method `getValue` (p. 6) of class `JSlider` returns the current thumb position.
- `JSlider` method `setMajorTickSpacing` () sets the spacing for tick marks on a `JSlider`. Method `setPaintTicks` (p. 6) with a `true` argument indicates that the tick marks should be displayed.
- `JSliders` generate `ChangeEvent`s when the user interacts with a `JSlider`. A `ChangeListener` (p. 6) declares method `stateChanged` (p. 6) that can respond to `ChangeEvent`s.

Section 35.3 Understanding Windows in Java

- A window's (p. 6) events can be handled by a `WindowListener` (p. 7), which provides seven window-event-handling methods—`windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified` and `windowOpened`.

Section 35.4 Using Menus with Frames

- Menus neatly organize commands in a GUI. In Swing GUIs, menus can be attached only to objects of classes with method `setJMenuBar` (p. 7).
- A `JMenuBar` (p. 7) is a container for menus. A `JMenuItem` appears in a menu. A `JMenu` (p. 7) contains menu items and can be added to a `JMenuBar` or to other `JMenus` as submenus.
- When a menu is clicked, it expands to show its list of menu items.
- When a `JCheckBoxMenuItem` (p. 8) is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check is removed.
- In a `ButtonGroup`, only one `JRadioButtonMenuItem` (p. 8) can be selected at a time.
- `AbstractButton` method `setMnemonic` (p. 13) specifies the mnemonic (p. 8) for a button. Mnemonic characters are normally displayed with an underline.
- A modal dialog box (p. 13) does not allow access to any other window in the application until the dialog is dismissed. The dialogs displayed with class `JOptionPane` are modal dialogs. Class `JDialog` (p. 13) can be used to create your own modal or nonmodal dialogs.

Section 35.5 JPopupMenu

- Context-sensitive pop-up menus (p. 15) are created with class `JPopupMenu`. The pop-up trigger event occurs normally when the user presses and releases the right mouse button. `MouseEvent` method `isPopupTrigger` (p. 18) returns `true` if the pop-up trigger event occurred.
- `JPopupMenu` method `show` (p. 18) displays a `JPopupMenu`. The first argument specifies the origin component, which helps determine where the `JPopupMenu` will appear. The last two arguments are the coordinates from the origin component's upper-left corner, at which the `JPopupMenu` appears.

Section 35.6 Pluggable Look-and-Feel

- Class `UIManager.LookAndFeelInfo` (p. 19) maintains information about a look-and-feel.
- `UIManager` (p. 19) static method `getInstalledLookFeels` (p. 19) returns an array of `UIManager.LookAndFeelInfo` objects that describe the available look-and-feels.
- `UIManager` static method `setLookAndFeel` (p. 22) changes the look-and-feel. `SwingUtilities` (p. 22) static method `updateComponentTreeUI` (p. 22) changes the look-and-feel of every component attached to its `Component` argument to the new look-and-feel.

Section 22.7 JDesktopPane and JInternalFrame

- Many of today's applications use a multiple-document interface (MDI; p. 23) to manage several open documents that are being processed in parallel. Swing's `JDesktopPane` (p. 23) and `JInternalFrame` (p. 23) classes provide support for creating multiple-document interfaces.

Section 22.8 JTabbedPane

- A `JTabbedPane` (p. 26) arranges GUI components into layers, of which only one is visible at a time. Users access each layer by clicking its tab.

Section 22.9 BoxLayout Layout Manager

- `BoxLayout` (p. 939) arranges GUI components left-to-right or top-to-bottom in a container.
- Class `Box` represents a container with `BoxLayout` as its default layout manager and provides static methods to create a `Box` with a horizontal or vertical `BoxLayout`.

Section 22.10 GridBagLayout Layout Manager

- `GridBagLayout` (p. 32) is similar to `GridLayout`, but each component size can vary.
- A `GridBagConstraints` object (p. 33) specifies how a component is placed in a `GridBagLayout`.

Self-Review Exercises

35.1 Fill in the blanks in each of the following statements:

- The _____ class is used to create a menu object.
- The _____ method of class `JMenu` places a separator bar in a menu.
- `JSlider` events are handled by the _____ method of interface _____.
- The `GridBagConstraints` instance variable _____ is set to CENTER by default.

35.2 State whether each of the following is *true* or *false*. If *false*, explain why.

- When the programmer creates a `JFrame`, a minimum of one menu must be created and added to the `JFrame`.
- The variable `fill` belongs to the `GridBagLayout` class.
- Drawing on a GUI component is performed with respect to the (0, 0) upper-left corner coordinate of the component.
- The default layout for a `Box` is `BoxLayout`.

35.3 Find the error(s) in each of the following and explain how to correct the error(s).

```
a) JMenubar b;
b) mySlider = JSlider(1000, 222, 100, 450);
c) gbc.fill = GridBagConstraints.NORTHWEST; // set fill
d) // override to paint on a customized Swing component
   public void paintcomponent(Graphics g)
{
    g.drawString("HELLO", 50, 50);
}
e) // create a JFrame and display it
JFrame f = new JFrame("A Window");
f.setVisible(true);
```

Answers to Self-Review Exercises

35.1 a) `JMenu`. b) `addSeparator`. c) `stateChanged`, `ChangeListener`. d) `anchor`.

35.2 a) False. A `JFrame` does not require any menus.

- b) False. The variable `fill` belongs to the `GridBagConstraints` class.
c) True.
d) True.
- 35.3**
- a) `JMenuBar` should be `JMenuBar`.
 - b) The first argument to the constructor should be `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`, and the keyword `new` must be used after the `=` operator. Also, the minimum value should be less than the maximum and the initial value should be in range.
 - c) The constant should be either `BOTH`, `HORIZONTAL`, `VERTICAL` or `NONE`.
 - d) `paintcomponent` should be `paintComponent`, and the method should call `super.paintComponent(g)` as its first statement.
 - e) The `JFrame`'s `setSize` method must also be called to establish the size of the window.

Exercises

35.4 (*Fill-in-the-Blanks*) Fill in the blanks in each of the following statements:

- a) A `JMenuItem` that is a `JMenu` is called a(n) _____.
- b) Method _____ attaches a `JMenuBar` to a `JFrame`.
- c) Container class _____ has a default `BoxLayout`.
- d) A(n) _____ manages a set of child windows declared with class `JInternalFrame`.

35.5 (*True or False*) State whether each of the following is *true* or *false*. If *false*, explain why.

- a) Menus require a `JMenuBar` object so they can be attached to a `JFrame`.
- b) `BoxLayout` is the default layout manager for a `JFrame`.
- c) `JApplets` can contain menus.

35.6 (*Find the Code Errors*) Find the error(s) in each of the following. Explain how to correct the error(s).

- a) `x.add(new JMenuItem("Submenu Color")); // create submenu`
- b) `container.setLayout(new GridbagLayout());`

35.7 (*Display a Circle and Its Attributes*) Write a program that displays a circle of random size and calculates and displays the area, radius, diameter and circumference. Use the following equations: $diameter = 2 \times radius$, $area = \pi \times radius^2$, $circumference = 2 \times \pi \times radius$. Use the constant `Math.PI` for pi (π). All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

35.8 (*Using a JSlider*) Enhance the program in Exercise 35.7 by allowing the user to alter the radius with a `JSlider`. The program should work for all radii in the range from 100 to 200. As the radius changes, the diameter, area and circumference should be updated and displayed. The initial radius should be 150. Use the equations from Exercise 35.7. All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

35.9 (*Varying weightx and weighty*) Explore the effects of varying the `weightx` and `weighty` values of the program in Fig. 35.21. What happens when a slot has a nonzero weight but is not allowed to fill the whole area (i.e., the `fill` value is not `BOTH`)?

35.10 (*Synchronizing a Slider and a JTextField*) Write a program that uses the `paintComponent` method to draw the current value of a `JSlider` on a subclass of `JPanel`. In addition, provide a `JTextField` where a specific value can be entered. The `JTextField` should display the current value of the `JSlider` at all times. Changing the value in the `JTextField` should also update the `JSlider`. A `JLabel` should be used to identify the `JTextField`. The `JSlider` methods `setValue` and `getValue` should be used. [Note: The `setValue` method is a `public` method that does not return a value and takes one integer argument, the `JSlider` value, which determines the position of the thumb.]

35.11 (Creating a Color Chooser) Declare a subclass of JPanel called MyColorChooser that provides three JSlider objects and three JTextField objects. Each JSlider represents the values from 0 to 255 for the red, green and blue parts of a color. Use these values as the arguments to the Color constructor to create a new Color object. Display the current value of each JSlider in the corresponding JTextField. When the user changes the value of the JSlider, the JTextField should be changed accordingly. Use your new GUI component as part of an application that displays the current Color value by drawing a filled rectangle.

35.12 (Creating a Color Chooser: Modification) Modify the MyColorChooser class of Exercise 35.11 to allow the user to enter an integer value into a JTextField to set the red, green or blue value. When the user presses *Enter* in the JTextField, the corresponding JSlider should be set to the appropriate value.

35.13 (Creating a Color Chooser: Modification) Modify the application in Exercise 35.12 to draw the current color as a rectangle on an instance of a subclass of JPanel which provides its own paintComponent method to draw the rectangle and provides set methods to set the red, green and blue values for the current color. When any set method is invoked, the drawing panel should automatically repaint itself.

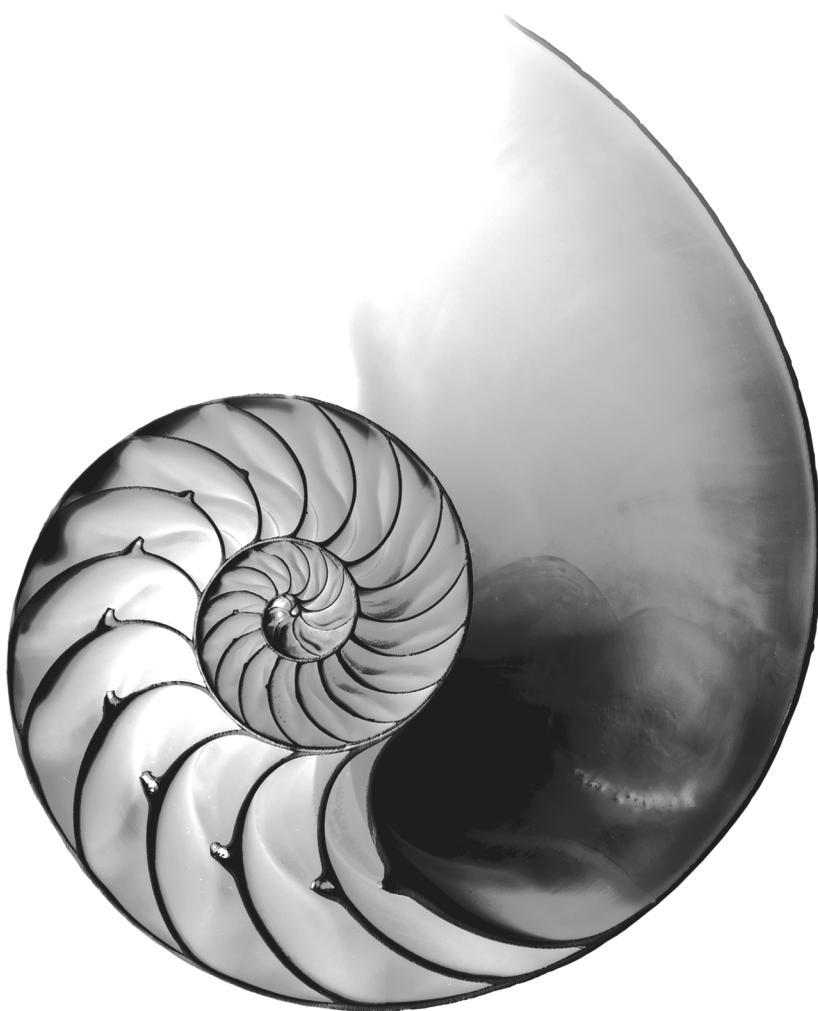
35.14 (Drawing Application) Modify the application in Exercise 35.13 to allow the user to drag the mouse across the drawing panel (a subclass of JPanel) to draw a shape in the current color. Enable the user to choose what shape to draw.

35.15 (Drawing Application Modification) Modify the application in Exercise 35.14 to allow the user to terminate the application by clicking the close box on the window that is displayed and by selecting Exit from a File menu. Use the techniques shown in Fig. 35.5.

35.16 (Complete Drawing Application) Using the techniques developed in this chapter and Chapter 26, create a complete drawing application. The program should use the GUI components from Chapter 26 and this chapter to enable the user to select the shape, color and fill characteristics. Each shape should be stored in an array of MyShape objects, where MyShape is the superclass in your hierarchy of shape classes. Use a JDesktopPane and JInternalFrames to allow the user to create multiple separate drawings in separate child windows. Create the user interface as a separate child window containing all the GUI components that allow the user to determine the characteristics of the shape to be drawn. The user can then click in any JInternalFrame to draw the shape.

36

Java Platform Module System



Objectives

In this chapter you'll:

- Understand the motivation for modularity.
- Review the Java Platform Module System JEPs and JSRs.
- Create `module` declarations that specify module dependencies with `requires` and specify which packages a module makes available to other modules with `exports`.
- Allow runtime reflection of types with `open` and `opens`.
- Use services to loosely couple system components to make large-scale systems easier to develop and maintain.
- Indicate that a module uses a service or provides a service implementation with the `uses` and `provides...with` directives, respectively.
- Use the `jdeps` command to determine a module's dependencies.
- Migrate non-modularized code to Java 9 with unnamed and automatic modules.
- Use the NetBeans IDE to create module graphs.
- See how the runtime determines dependencies with the module resolver.
- Use `jlink` to create smaller runtimes appropriate for resource-constrained devices.

Outline

36.1	Introduction	36.6	Migrating Code to Java 9
36.2	Module Declarations	36.6.1	Unnamed Module
36.2.1	<code>requires</code>	36.6.2	Automatic Modules
36.2.2	<code>requires transitive</code> —Implied	36.6.3	<code>jdeps</code> : Java Dependency Analysis
	Readability		
36.2.3	<code>exports</code> and <code>exports...to</code>	36.7	Resources in Modules; Using an
36.2.4	<code>uses</code>		Automatic Module
36.2.5	<code>provides...with</code>	36.7.1	Automatic Modules
36.2.6	<code>open</code> , <code>opens</code> and <code>opens...to</code>	36.7.2	Requiring Multiple Modules
36.2.7	Restricted Keywords	36.7.3	Opening a Module for Reflection
36.3	Modularized <code>Welcome</code> App	36.7.4	Module-Dependency Graph
36.3.1	<code>Welcome</code> App's Structure	36.7.5	Compiling the Module
36.3.2	Class <code>Welcome</code>	36.7.6	Running a Modularized App
36.3.3	<code>module-info.java</code>	36.8	Creating Custom Runtimes with
36.3.4	Module-Dependency Graph		<code>jlink</code>
36.3.5	Compiling a Module	36.8.1	Listing the JRE's Modules
36.3.6	Running an App from a Module's	36.8.2	Custom Runtime Containing Only
	Exploded Folders		<code>java.base</code>
36.3.7	Packaging a Module into a Modular	36.8.3	Creating a Custom Runtime for the
	JAR File		<code>Welcome</code> App
36.3.8	Running the <code>Welcome</code> App from a	36.8.4	Executing the <code>Welcome</code> App Using a
	Modular JAR File		Custom Runtime
36.3.9	Aside: Classpath vs. Module Path	36.8.5	Using the Module Resolver on a
36.4	Creating and Using a Custom		Custom Runtime
	Module	36.9	Services and <code>ServiceLoader</code>
36.4.1	Exporting a Package for Use in Other	36.9.1	Service-Provider Interface
	Modules	36.9.2	Loading and Consuming Service
36.4.2	Using a Class from a Package in		Providers
	Another Module	36.9.3	<code>uses</code> Module Directive and Service
36.4.3	Compiling and Running the Example		Consumers
36.4.4	Packaging the App into Modular JAR	36.9.4	Running the App with No Service
	Files		Providers
36.4.5	Strong Encapsulation and	36.9.5	Implementing a Service Provider
	Accessibility	36.9.6	<code>provides...with</code> Module Directive
36.5	Module-Dependency Graphs: A		and Declaring a Service Provider
	Deeper Look	36.9.7	Running the App with One Service
36.5.1	<code>java.sql</code>		Provider
36.5.2	<code>java.se</code>	36.9.8	Implementing a Second Service
36.5.3	Browsing the JDK Module Graph		Provider
36.5.4	Error: Module Graph with a Cycle	36.9.9	Running the App with Two Service
			Providers
		36.10	Wrap-Up

36.1 Introduction¹

In this chapter, we introduce the **Java Platform Module System (JPMS)**—Java 9’s most important new technology. Modularity—the result of **Project Jigsaw**²—helps developers at all levels be more productive as they build, maintain and evolve software systems, especially large systems. College students in upper-level programming courses will want to master modularity for career preparation.

-
1. We’d like to thank Brian Goetz, Alex Buckley, Alan Bateman, Lance Anderson, Mandy Chung and Paul Bakker for answering our questions and sharing insights.
 2. “Project Jigsaw.” <http://openjdk.java.net/projects/jigsaw/>.

Software Required

Before reading this chapter, install JDK 9 and the chapter's source-code examples as described in the Before You Begin section that follows the Preface. We'll present several module-dependency graphs that were created with an early access version of the NetBeans IDE that includes JDK 9 support:

<http://wiki.netbeans.org/JDK9Support>

Other IDE vendors will likely provide similar tools.

What is a Module?

Modularity adds a higher level of aggregation above packages. The key new language element is the **module**—a uniquely named, reusable group of related packages, as well as resources (like images and XML files) and a **module descriptor** specifying:

- the module's *name*,
- the module's *dependencies* (that is, other modules this module depends on),
- the packages it *explicitly* makes available to other modules (all other packages in the module are *implicitly* unavailable to other modules),
- the *services it offers*,
- the *services it consumes*, and
- to what other modules it allows *reflection*.

History

The Java SE Platform has been around since 1995. There are now approximately 10 million developers using it to build everything from small apps for resource-constrained devices—like those in the Internet of Things (IoT) and other embedded devices—to large-scale business-critical and mission-critical systems. There are massive amounts of legacy code out there, but until now, the Java platform has primarily been a monolithic one-size-fits-all solution. Over the years there have been various efforts geared to modularizing Java, but none is widely used.

Modularizing the Java SE Platform has been challenging to implement and the effort has taken many years. *JSR 277: Java Module System* was originally proposed in 2005³ for Java 7. This JSR was later superseded by *JSR 376: Java Platform Module System* and targeted for Java 8. The Java SE Platform is now modularized in Java 9, but only after Java 9 was delayed until July 2017.

Goals

According to JSR 376, the key goals of modularizing the Java SE Platform are:⁴

- Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.

3. “JSR 277: Java Module System.” <https://jcp.org/en/jsr/detail?id=277>.

4. “JSR 376: Java Platform Module System.” <https://jcp.org/en/jsr/detail?id=376>.

- Strong encapsulation—The packages in a module are accessible to other modules only if the module explicitly “exports” them. Even then, another module cannot use those packages unless it explicitly states that it “requires” the other module’s capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.
- Scalable Java Platform—Previously the Java Platform was a monolith consisting of a massive numbers of packages, making it challenging to develop, maintain and evolve. It couldn’t be easily subsetted. The platform is now modularized into 95 modules (this number will change as Java evolves). You can create custom runtimes consisting of only modules you need for your apps or the devices you’re targeting. For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime’s size.
- Greater platform integrity—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app’s classes. With strong encapsulation, these internal APIs are truly encapsulated and hidden from apps using the platform. One downside of this is that it can make migrating your legacy code to Java 9 problematic.
- Improved performance—The JVM uses various optimization techniques to improve application performance. JSR 376⁵ indicates that these techniques are more effective when it’s known in advance that required types are located only in specific modules.

Listing the JDK’s Modules

A crucial aspect of Java 9 is dividing the JDK into modules to support various configurations (JEP 200⁶). Using the `java` command from the JDK’s `bin` folder with the `--list-modules` option, as in:

```
java --list-modules
```

lists the JDK’s set of modules (Fig. 36.1), which includes the **standard modules** that implement the Java SE Specification (names starting with `java`), JavaFX modules (names starting with `javafx`), JDK-specific modules (names starting with `jdk`) and Oracle-specific modules (names starting with `oracle`). Each module name is followed by a *version string*. In this case, we used a JDK 9 early access version, so each module is followed by the version string “@9-ea”, indicating that it’s a Java 9 early access (“ea”) module. The “-ea” will be removed when Java 9 is released.

5. Reinhold, Mark. “JSR 376: Java Platform Module System.” <https://jcp.org/en/jsr/detail?id=376>.

6. Reinhold, Mark. “JEP 200: The Modular JDK.” <http://openjdk.java.net/jeps/200>.

java.activation@9-ea	jdk.httpserver@9-ea
java.base@9-ea	jdk.incubator.httpclient@9-ea
java.compiler@9-ea	jdk.internal.ed@9-ea
java.corba@9-ea	jdk.internal.jvmstat@9-ea
java.datatransfer@9-ea	jdk.internal.le@9-ea
java.desktop@9-ea	jdk.internal.opt@9-ea
java.instrument@9-ea	jdk.internal.vm.ci@9-ea
java.jnlp@9-ea	jdk.jartool@9-ea
java.logging@9-ea	jdk.javadoc@9-ea
java.management@9-ea	jdk.javaws@9-ea
java.management.rmi@9-ea	jdk.jcmd@9-ea
java.naming@9-ea	jdk.jconsole@9-ea
java.prefs@9-ea	jdk.jdeps@9-ea
java.rmi@9-ea	jdk.jdi@9-ea
java.scripting@9-ea	jdk.jdwp.agent@9-ea
java.se@9-ea	jdk.jfr@9-ea
java.se.ee@9-ea	jdk.jlink@9-ea
java.security.jgss@9-ea	jdk.jshell@9-ea
java.security.sasl@9-ea	jdk.jsobject@9-ea
java.smartcardio@9-ea	jdk.jstadv@9-ea
java.sql@9-ea	jdk.localedata@9-ea
java.sql.rowset@9-ea	jdk.management@9-ea
java.transaction@9-ea	jdk.management.agent@9-ea
java.xml@9-ea	jdk.naming.dns@9-ea
java.xml.bind@9-ea	jdk.naming.rmi@9-ea
java.xml.crypto@9-ea	jdk.net@9-ea
java.xml.ws@9-ea	jdk.pack@9-ea
java.xml.ws.annotation@9-ea	jdk.packager@9-ea
javafx.base@9-ea	jdk.packager.services@9-ea
javafx.controls@9-ea	jdk.plugin@9-ea
javafx.deploy@9-ea	jdk.plugin.dom@9-ea
javafx.fxml@9-ea	jdk.plugin.server@9-ea
javafx.graphics@9-ea	jdk.policytool@9-ea
javafx.media@9-ea	jdk.rmic@9-ea
javafx.swing@9-ea	jdk.scripting.nashorn@9-ea
javafx.web@9-ea	jdk.scripting.nashorn.shell@9-ea
jdk.accessibility@9-ea	jdk.sctp@9-ea
jdk.attach@9-ea	jdk.security.auth@9-ea
jdk.charsets@9-ea	jdk.security.jgss@9-ea
jdk.compiler@9-ea	jdk.snmp@9-ea
jdk.crypto.cryptoki@9-ea	jdk.unsupported@9-ea
jdk.crypto.ec@9-ea	jdk.xml.bind@9-ea
jdk.crypto.msapi@9-ea	jdk.xml.dom@9-ea
jdk.deploy@9-ea	jdk.xml.ws@9-ea
jdk.deploy.controlpanel@9-ea	jdk.zipfs@9-ea
jdk.dynalink@9-ea	oracle.desktop@9-ea
jdk.editpad@9-ea	oracle.net@9-ea
jdk.hotspot.agent@9-ea	

Fig. 36.1 | Output of `java --list-modules` showing the JDK's 95 modules.

JEPs and JSRs of Java Modularity

We discussed what JEPs and JSRs are in the Preface. The Java modularity JEPs and JSRs are shown in Fig. 36.2. We cite these throughout the chapter.

Java Modularity JEPs and JSRs

- JEP 200: The Modular JDK (<http://openjdk.java.net/jeps/200>)
- JEP 201: Modular Source Code (<http://openjdk.java.net/jeps/201>)
- JEP 220: Modular Run-Time Images (<http://openjdk.java.net/jeps/220>)
- JEP 260: Encapsulate Most Internal APIs (<http://openjdk.java.net/jeps/260>)
- JEP 261: Module System (<http://openjdk.java.net/jeps/261>)
- JEP 275: Modular Java Application Packaging (<http://openjdk.java.net/jeps/275>)
- JEP 282: jlink: The Java Linker (<http://openjdk.java.net/jeps/282>)
- JSR 376: Java Platform Module System (<https://www.jcp.org/en/jsr/detail?id=376>)
- JSR 379: Java SE 9 (<https://www.jcp.org/en/jsr/detail?id=379>)

Fig. 36.2 | Java Modularity JEPs and JSRs.

Quick Tour of the Chapter

This chapter introduces key modularity concepts you’re likely to use when building large-scale systems. Some of the key topics you’ll see throughout this chapter include:

- Module declarations—You’ll create module declarations that specify a module’s dependencies (with the `requires` directive), which packages a module makes available to other modules (with the `exports` directive), services it offers (with the `provides...with` directive), services it consumes (with the `uses` directive) and to what other modules it allows reflection (with the `open` modifier and the `opens` and `opens...to` directives).
- Module-dependency graphs—We’ll use the NetBeans IDE’s JDK 9 support to create module graphs that help you visualize the dependencies among modules.
- Module resolver—We’ll show you the steps the runtime’s module resolver performs to ensure that a module’s dependencies are fulfilled.
- `jlink` (the Java linker)—You’ll use this new JDK 9 tool to create smaller custom runtimes, then use them to execute apps. In fact, many of this book’s command-line apps can be executed on a custom runtime consisting only of the most fundamental JDK module—**java.base**—which includes core Java API packages, such as `java.lang`, `java.io` and `java.util`. As you’ll see, all modules *implicitly* depend on `java.base`.
- Reflection—*Reflection* enables a Java program to dynamically load types then create objects of those types and use them.⁷ These capabilities can still be used, despite Java 9’s strong encapsulation, but only with modules that *explicitly* allow it. We’ll show how to specify that a module allows reflection with an `open` modifier and the `opens` and `opens...to` directive in a module declaration.
- Migration—The Java platform has been in use for over 20 years, so enormous amounts of non-modularized legacy code will need to be migrated to the modular

7. *The Java™ Tutorials*, “Trail: The Reflection API,” <https://docs.oracle.com/javase/tutorial/reflect/>.

world of Java 9. Though there are traps and pitfalls due to Java 9's stronger encapsulation, we'll show how the unnamed module and automatic modules can help make migration straightforward. We'll use the `jdeps` tool to determine code dependencies among modules and on pre-Java-9 internal APIs (which are for the most part strongly encapsulated in Java 9). Much pre-Java-9 code will run without modification, but there are some issues that we explain in Section 36.6.

- Services and Service Providers—When you create substantial software systems that fulfill important needs, they can live on for decades. During that time, change is the rule rather than the exception. In Section 10.13, we discussed *tight coupling* and *loose coupling*. It's been proven that *tight coupling* makes it difficult to modify systems. We'll show how to create loosely coupled system components with service-provider interfaces and implementations and the `ServiceLoader` class. We'll also demonstrate the `uses` and `provides...with` directives in module declarations to indicate that a module uses a service or provides a service implementation, respectively.

We'll present the preceding concepts using several larger live-code examples with meaningful outputs, some code snippets, module graphs produced with the NetBeans IDE's **Graph** view of a module declaration and examples of various new commands (like `jlink`) and new options for existing commands (like `javac`, `java` and `jar`). Some additional example-rich sources are:

- Project Jigsaw: Module System Quick-Start Guide—<http://openjdk.java.net/projects/jigsaw/quick-start>
- Mak, Sander, and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. Sebastopol, CA: O'Reilly Media, 2017.

A Terminology Note

The Java Runtime Environment (JRE) includes the Java Virtual Machine (JVM) and other software for executing Java programs. As of Java 9, the JRE is now a proper subset of the Java Development Kit (JDK), which contains all the Java APIs and tools required to create and run Java programs. This chapter uses the terms Java Platform and Java SE Platform synonymously with the JDK.

36.2 Module Declarations

As we mentioned, a module must provide a module descriptor—metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more. A module descriptor is the compiled version of a **module declaration** that's defined in a file named **module-info.java**. Each module declaration begins with the keyword **module**, followed by a unique **module name** and a **module body** enclosed in braces, as in

```
module modulename {  
}
```

The module declaration's body can be empty or may contain various **module directives**, including `requires`, `exports`, `provides...with`, `uses` and `opens` (each of which we discuss). As you'll see in Section 36.3.5, compiling the module declaration creates the module descriptor, which is stored in a file named **module-info.class** in the module's root

folder. Here we briefly introduce each module directive. You'll see actual module declarations beginning in Section 36.3.3.

36.2.1 requires

A **requires module directive** specifies that this module depends on another module—this relationship is called a **module dependency**. Each module *must* explicitly state its dependencies. When module A **requires** module B, module A is said to **read** module B and module B is **read by** module A. To specify a dependency on another module, use **requires**, as in:

```
requires modulename;
```

Section 36.3.3 demonstrates a **requires** directive.⁸

36.2.2 requires transitive—Implied Readability

To specify a dependency on another module and to ensure that other modules reading your module also read that dependency—known as **implied readability**—use **requires transitive** as in:

```
requires transitive modulename;
```

Consider the following directive from the `java.desktop` module declaration:

```
requires transitive java.xml;
```

In this case, any module that reads `java.desktop` also *implicitly* reads `java.xml`. For example, if a method from the `java.desktop` module returns a type from the `java.xml` module, code in modules that read `java.desktop`, becomes dependent on `java.xml`. Without the **requires transitive** directive in `java.desktop`'s module declaration, such dependent modules will not compile unless they *explicitly* read `java.xml`.

According to JSR 379,⁹ Java SE's standard modules *must* grant implied readability in all cases like the one described here. Also, though a Java SE standard module may depend on non-standard modules, it *must not* grant implied readability to them.



Portability Tip 36.1

*Because Java SE standard modules **must not** grant implied readability to non-standard modules, code depending only on Java SE standard modules is portable across Java SE implementations.*

36.2.3 exports and exports...to

An **exports module directive** specifies one of the module's packages whose **public** types (and their nested **public** and **protected** types) should be accessible to code in all other modules. An **exports...to directive** enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—this is known as a **qualified export**. Section 36.4 demonstrates the **exports** directive.

8. There is also a **requires static** directive to indicate that a module is required at *compile time*, but *optional* at runtime. This is known as an *optional dependency* and is beyond this chapter's scope.
9. Clark, Iris, and Mark Reinhold. "Java SE 9 (JSR 379)." March 6, 2017. <http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html#s7>.

36.2.4 uses

A **uses module directive** specifies a service used by this module—making the module a **service consumer**. A service is an object of a class that implements the interface or extends the abstract class specified in the uses directive. Section 36.9.3 demonstrates the uses directive.

36.2.5 provides...with

A **provides...with module directive** specifies that a module provides a service implementation—making the module a **service provider**. The provides part of the directive specifies an interface or abstract class listed in a module’s uses directive and the with part of the directive specifies the name of the class that implements the interface or extends the abstract class. Section 36.9.6 demonstrates the provides...with directive.

36.2.6 open, opens and opens...to^{10,11}

Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its private members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is *strong encapsulation*. By default, a type in a module is not accessible to other modules unless it’s a **public** type *and* you export its package. You expose only the packages you want to expose. With Java 9, this also applies to reflection.

Allowing Runtime-Only Access to a Package

An **opens** module directive of the form

```
opens package
```

indicates that a specific *package*’s **public** types (and their nested **public** and **protected** types) are accessible to code in other modules at runtime only. Also, all of the types in the specified package (and all of the types’ members) are accessible via reflection.

Allowing Runtime-Only Access to a Package By Specific Modules

An **opens...to** module directive of the form

```
opens package to comma-separated-list-of-modules
```

indicates that a specific *package*’s **public** types (and their nested **public** and **protected** types) types are accessible to code in the listed module(s) at runtime only. Also, all of the types in the specified package (and all of the types’ members) are accessible via reflection to code in the specified modules.

-
10. Buckley, Alex. “JPMS: Modules in the Java Language and JVM.” February 23, 2017. <http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html>.
 11. Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Dan Smith. “The Java® Language Specification Java SE 9 Edition.” Section 7.7.2. February 22, 2017. <http://cr.openjdk.java.net/~mr/jigsaw/spec/java-se-9-jls-pr-diffs.pdf>.

Allowing Runtime-Only Access to All Packages in a Module

If all the packages in a given module should be accessible at runtime and via reflection to all other modules, you may **open** the entire module, as in

```
open module modulename {
    // module directives
}
```

Reflection Defaults

By default, a module with runtime reflective access to a package can see the package's **public** types (and their nested **public** and **protected** types). However, the code in other modules *can* access *all* types in the exposed package and *all* members within those types, including **private** members. For more information on using reflection to access all of a type's members, visit

<https://docs.oracle.com/javase/tutorial/reflect/>

Dependency Injection

Reflection is commonly used with *dependency injection*. One example of this, is an FXML-based JavaFX app, like those you've seen in Chapters 12, 13, 22 and miscellaneous other examples. When an FXML app loads, the controller object and the GUI components on which it *depends* are dynamically created as follows:

- First, because the app *depends* on a controller object that handles the GUI interactions, the **FXMLLoader** *injects* a controller object into the running app—that is, the **FXMLLoader** uses reflection to locate and load the controller class into memory, and to create an object of that class.
- Next, because the controller *depends* on the GUI components declared in FXML, the **FXMLLoader** creates the GUI components objects declared in the FXML and *injects* them into the controller object by assigning each to the controller object's corresponding @FXML instance variable.

Once this process is complete, the controller can interact with the GUI and respond to its events. We'll use the **opens...**to directive in Section 36.7.2 to allow the **FXMLLoader** to use reflection on a JavaFX app in a custom module.

36.2.7 Restricted Keywords

The keywords **exports**, **module**, **open**, **opens**, **provides**, **requires**, **to**, **transitive**, **uses** and **with** are *restricted keywords*. They're keywords only in module declarations and may be used as identifiers anywhere else in your code.

We mentioned in footnote 8 that there is also a **requires static** module directive. Of course, **static** is a regular keyword.

36.3 Modularized Welcome App

In this section, we create a simple **Welcome** app to demonstrate module fundamentals. We'll:

- create a class that resides in a module,
- provide a module declaration,

- compile the module declaration and `Welcome` class into a module, and
- run the class containing `main` in that module.

After covering these basics, we'll also demonstrate:

- packaging the `Welcome` app in a modular JAR file and
- running the app from that JAR file.

36.3.1 Welcome App's Structure

The app we present in this section consists of two .java files—`Welcome.java` contains the `Welcome` app class and `module-info.java` contains the module declaration. By convention, a modularized app has the following folder structure:



For our `Welcome` app, which will be defined in the package `com.deitel.welcome`, the folder structure is shown in Fig. 36.3.

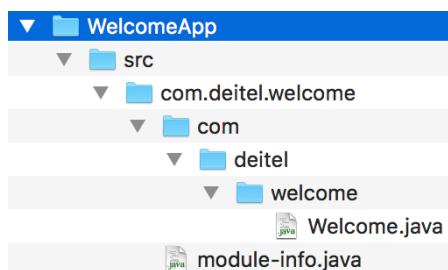


Fig. 36.3 | Folder structure for the `Welcome` app.

The `src` folder stores all of the app's source code. It contains the module's **root folder**, which has the module's name—`com.deitel.welcome` (we'll discuss module naming momentarily). The module's root folder contains nested folders representing the package's directory structure—`com/deitel/welcome`—which corresponds to the package `com.deitel.welcome`. This folder contains `Welcome.java`. The module's root folder contains the required module declaration `module-info.java`.

Module Naming Conventions

Like package names, module names must be *unique*. To ensure *unique* package names, you typically begin the name with your organization's Internet domain name in reverse order. Our domain name is `deitel.com`, so we begin our package names with `com.deitel`. By convention, module names also use the reverse-domain-name convention.

At compile time, if multiple modules have the same name, a compilation error occurs. At runtime, if multiple modules have the same name an exception occurs.

This example uses the same name for the module and its contained package, because there is only one package in the module. This is not required, but is a common convention. In a modular app, Java maintains the module names separately from package names and any type names in those packages, so duplicate module and package names *are* allowed.

Modules normally group related packages. As such, the packages will often have commonality among portions of their names. For example, if a module contains the packages

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

you'd typically name the module with the common portion of the package names—`com.deitel.sample`. If there's no common portion, then you'd choose a name representing the module's purpose. For example, the `java.base` module contains core packages that are considered fundamental to Java apps (such as `java.lang`, `java.io`, `java.time` and `java.util`), and the `java.sql` module contains the packages required for interacting with databases via JDBC (such as `java.sql` and `javax.sql`). These are just two of the many standard modules that you saw in Fig. 36.1. The online documentation for each provides a complete list of its exported packages—for the `java.base` module, visit:

```
http://download.java.net/java/jdk9/docs/api/java.base-summary.html
```

Listing the `java.base` Module's Contents

You can use the `java` command's `--list-modules` option to display information from the `java.base` module's descriptor, including its list of exported packages, as in:

```
java --list-modules java.base
```

Figure 36.4 shows the *portion* of the preceding command's output which lists the `java.base` module's packages that *any* module can access. You've used several of these packages in the book, including `java.io`, `java.lang`, `java.math`, `java.nio`, `java.time` and `java.util`.

<pre>exports java.io exports java.lang exports java.lang.annotation exports java.lang.invoke exports java.lang.module exports java.lang.ref exports java.lang.reflect exports java.math exports java.net exports java.net.spi exports java.nio exports java.nio.channels exports java.nio.channels.spi exports java.nio.charset exports java.nio.charset.spi exports java.nio.file exports java.nio.file.attribute exports java.nio.file.spi exports java.security</pre>	<pre>exports java.security.acl exports java.security.cert exports java.security.interfaces exports java.security.spec exports java.text exports java.text.spi exports java.time exports java.time.chrono exports java.time.format exports java.time.temporal exports java.time.zone exports java.util exports java.util.concurrent exports java.util.concurrent.atomic exports java.util.concurrent.locks exports java.util.function exports java.util.jar exports java.util.regex exports java.util.spi</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 36.4 | Partial output of the command `java --list-modules java.base`. (Part 1 of 2.)

<pre>exports java.util.stream exports java.util.zip exports javax.crypto exports javax.crypto.interfaces exports javax.crypto.spec exports javax.net exports javax.net.ssl</pre>	<pre>exports javax.security.auth exports javax.security.auth.callback exports javax.security.auth.login exports javax.security.auth.spi exports javax.security.auth.x500 exports javax.security.cert</pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 36.4 | Partial output of the command `java --list-modules java.base`. (Part 2 of 2.)

The complete output of the preceding command lists lots of additional information about the `java.base` module. Figure 36.5 shows some of the remaining output with sample lines from each category of information.

```
...
uses java.util.spi.CurrencyNameProvider
uses java.util.spi.ResourceBundleControlProvider
uses java.util.spi.LocaleNameProvider
...
provides java.nio.file.spi.FileSystemProvider
with jdk.internal.jrtfs.JrtFileSystemProvider
...
exports sun.net.sdp to oracle.net
exports jdk.internal.jimage to jdk.jlink
exports sun.net.www.protocol.http.ntlm to jdk.deploy
...
contains com.sun.crypto.provider
contains com.sun.java.util.jar.pack
contains com.sun.net.ssl
...
```

Fig. 36.5 | Partial output of the command `java --list-modules java.base` showing other categories of information that it displays.

The `uses` lines, like

```
uses java.util.spi.CurrencyNameProvider
```

indicate that there are types in the `java.base` module's packages which use objects that implement various service-provider interfaces. The `provides...with`

```
provides java.nio.file.spi.FileSystemProvider
with jdk.internal.jrtfs.JrtFileSystemProvider
```

indicates that this module's `jdk.internal.jrtfs` package contains a service-provider implementation class named `JrtFileSystemProvider` that implements the service-provider interface named `FileSystemProvider` from package `java.nio.file.spi`. Section 36.9 shows a substantial example demonstrating that service-provider interfaces and implementations can be used to create *loosely coupled* system components for systems that are easier to develop, maintain and evolve than tightly coupled systems.

The `exports...to` lines like

```
exports sun.net.sdp to oracle.net
```

indicate that the `java.base` module exports a given package (`sun.net.sdp`) *only to a specified module* (`oracle.net`). The `java.base` module has many of these qualified exports. Packages listed in such exports may be read only by the one or more designated modules in the comma-separated list after the keyword `to`. In the JDK, such qualified exports are used for packages (like `sun.net.sdp`) containing JDK internal implementations of types that should not be used by developers.

The `contains` lines, like

```
contains com.sun.crypto.provider
```

specify that the module contains packages that are not exported for use in other modules. Note that `contains` is not a directive like `requires` or `exports` that you can use in your modules. Rather, it's information inserted by the compiler to indicate that a module contains the specified package—the package is not exported for use by other modules. The JVM uses this information to improve performance when it loads classes from those packages at runtime.¹²

36.3.2 Class Welcome

Figure 36.6 presents a `Welcome` app that simply displays a `String` at the command line. When defining types that will be placed in modules, every type *must* be placed into a package (line 3).

```
1 // Fig. 36.6: Welcome.java
2 // Welcome class that will be placed in a module
3 package com.deitel.welcome; // all classes in modules must be packaged
4
5 public class Welcome {
6     public static void main(String[] args) {
7         // class System is in package java.lang from the java.base module
8         System.out.println("Welcome to the Java Platform Module System!");
9     }
10 }
```

Fig. 36.6 | Welcome class that will be placed in a module.

36.3.3 module-info.java

Figure 36.7 contains the module declaration for the `com.deitel.welcome` module. We call modules we create for our own use **application modules**.

```
1 // Fig. 36.7: module-info.java
2 // Module declaration for the com.deitel.welcome module
3 module com.deitel.welcome {
4     requires java.base; // implicit in all modules, so can be omitted
5 }
```

Fig. 36.7 | Module declaration for the `com.deitel.welcome` module.

12. Brian Goetz, e-mail message to authors, March 16, 2017.

Again, the module declaration begins with the keyword `module` followed by the module's name and braces that enclose the declaration's body. This module declaration contains a `requires` module directive, indicating that the app depends on types defined in module `java.base`. Actually, all modules depend on `java.base`, so the `requires` module directive in line 4 is *implicit* in all `module` declarations and may be omitted, as in:

```
module com.deitel.welcome {  
}
```



Software Engineering Observation 36.1

Every module implicitly depends on java.base. Writing requires java.base; in a module declaration is redundant.

36.3.4 Module-Dependency Graph

Figure 36.8 shows the **module-dependency graph** for `com.deitel.welcome`, indicating that the module reads only the standard module `java.base`. This dependency is indicated in the diagram with the arrow from `com.deitel.welcome` to `java.base`. This graph will be identical regardless of whether you include line 4 in the module declaration.

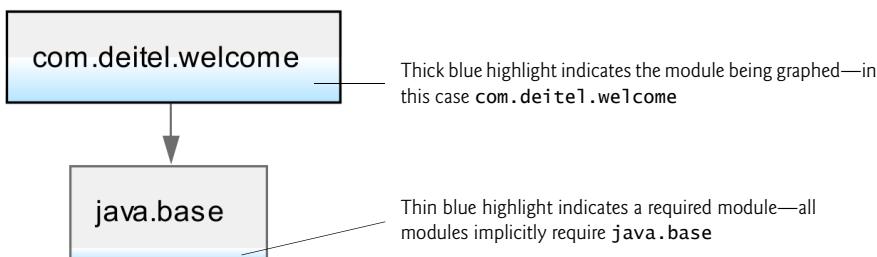


Fig. 36.8 | Module-dependency graph for the `com.deitel.welcome` module.

A module-dependency graph shows dependencies among **observable modules**¹³—that is, the built-in standard modules and any additional modules required by a given app or library module. The graph's *nodes* are modules and their *dependencies* are represented by directed edges (arrows) that connect the nodes. Some edges represent **explicit dependencies** on modules explicitly specified in a module declaration's `requires` clauses (as you'll see in Fig. 36.14). Some edges represent **implicit dependencies** in which one of the required modules in turn depends on other modules (as you'll see in Fig. 36.22). In Fig. 36.8, `java.base` is shown as an explicit dependency, because all modules depend on it.

This graph was produced with an early access version of NetBeans that has JDK 9 support—again, you can learn about this version of the IDE and download its installer from:

```
http://wiki.netbeans.org/JDK9Support
```

In a NetBeans project, when you open a module's `module-info.java` file, you can choose between the **Source** code and **Graph** views. In **Graph** view, NetBeans creates a module-de-

13. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." <http://openjdk.java.net/jeps/261>.

pendency- graph, based on the module declaration. When NetBeans graphs a module, it also graphs that module's dependencies, including the implicit dependency on `java.base`. Figure 36.8 shows that `java.base` itself does not have any dependencies.

To create this graph in NetBeans, we performed the following steps:

1. First, we created a `WelcomeApp` project containing the `com.deitel.welcome` package.
2. Next, we added the `com.deitel.welcome` module's `module-info.java` file by right-clicking the project, selecting **New > Other...**, selecting **Java Module Info** from the **Java** category of the dialog, then clicking **Next >** and **Finish**. The file is added to the project's default package automatically.
3. Finally, we opened `module-info.java` file, changed the module name from the default provided by NetBeans (the project name) to `com.deitel.welcome` and switched to **Graph** view.

You can arrange the nodes in NetBeans by dragging them or by right clicking the graph and selecting from various **Layout** options—we chose **Hierarchical**, in which the given module appears at the top and arrows point down to the module's dependencies. You may use **Zoom To Fit** to make the graph fill the available space in the window and **Export As Image** to save an image containing the graph.

36.3.5 Compiling a Module

To compile the `Welcome` app's module, open a command window, use the `cd` command to change to this chapter's `WelcomeApp` folder, then type:

```
javac -d mods/com.deitel.welcome ^
      src/com.deitel.welcome/module-info.java ^
      src/com.deitel.welcome/com/deitel/welcome/Welcome.java
```

The `-d` option indicates that `javac` should place the compiled code in the specified folder—in this case a `mods` folder that will contain a subfolder named `com.deitel.welcome` representing the compiled module. The name `mods` is used by convention for a folder that contains modules.

Note Regarding Lengthy Commands in This Chapter

For clarity, we split the preceding command into multiple lines, using line-continuation characters. Many of the commands we use in this chapter's examples are lengthy. This chapter shows the commands in Windows format, with the caret (^) line-continuation character. Linux and macOS users should replace the carets in the commands with the backslash (\) line-continuation character. You can also enter such lengthy commands as a single command without the line continuations.

Welcome App's Folder Structure After Compilation

If the code compiles correctly, the `WelcomeApp` folder's `mods` subfolder structure contains the compiled code (Fig. 36.9). This is known as the **exploded-module folder**, because the folders and `.class` files are not in a JAR (Java archive) file—collection of directories and files compressed into a single file, known as an archive. The exploded module's structure parallels that of the app's `src` folder described previously. We'll package the app as a JAR

shortly. Exploded module folders and modular JAR files (Section 36.3.7) together are **module artifacts**. These can be placed on the **module path**—a list of module artifact locations—when compiling and executing modularized code.^{14,15}

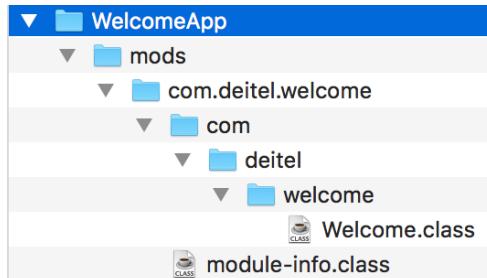


Fig. 36.9 | Welcome app's mods folder structure.

Listing the com.deitel.welcome Module's Contents

You can use the `java` command's `--list-modules` option to display information from the `com.deitel.welcome` module descriptor, as in:

```
java --module-path mods --list-modules com.deitel.welcome
```

The resulting output:

```
module com.deitel.welcome (file:///C:/examples/ch36/WelcomeApp/
  mods/com.deitel.welcome/)
  requires java.base (@9-ea)
  contains com
  contains com.deitel
  contains com.deitel.welcome
```

shows that the module **requires** the standard module `java.base` and **contains** the packages `com`, `com.deitel` and `com.deitel.welcome` (each folder is viewed as a package). Though the module **contains** these packages, they are *not* exported. Therefore, its contents *cannot* be used by other modules. The module declaration for this example *explicitly* required `java.base` and the preceding listing included

```
requires java.base
```

If the module declaration had *implicitly* required `java.base`, then the listing instead would have included

```
requires mandated java.base
```

There is no `requires mandated` module directive—it is simply included in the `--list-modules` output to indicate the implicit dependence on `java.base`.

-
14. Reinhold, Mark. “The State of the Module System.” March 8, 2016. <http://openjdk.java.net/projects/jigsaw/spec/sotms/#module-artifacts>.
 15. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. “JEP 261: Module System.” <http://openjdk.java.net/jeps/261>.

36.3.6 Running an App from a Module's Exploded Folders

To run the `Welcome` app from the module's exploded folders, use the following command (again, from the `WelcomeApp` folder):

```
java --module-path mods ^
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

The `--module-path` option specifies the module path—in this case, the `mods` folder. The `--module` option specifies the module name and the fully qualified class name of the app's entry point—that is, a class containing `main`. The program executes and displays:

```
Welcome to the Java Platform Module System!
```

In the preceding command, `--module-path` can be abbreviated as `-p` and `--module` as `-m`.

36.3.7 Packaging a Module into a Modular JAR File

You can use the `jar` command to package an exploded module folder as a **modular JAR file**¹⁶ that contains all of the module's files, including its `module-info.class` file, which is placed in the JAR's root folder. When running the app, you specify the JAR file on the module path. The folder in which you wish to output the JAR file must exist before running the `jar` command.

If a module contains an app's entry point, you can specify that class with the `jar` command's `--main-class` option, as in:

```
jar --create -f jars/com.deitel.welcome.jar ^
--main-class com.deitel.welcome.Welcome ^
-C mods/com.deitel.welcome .
```

The options are as follows:

- `--create` specifies that the command should create a new JAR file.
- `-f` specifies the name of the JAR file and is followed by the name—in this case, the file `com.deitel.welcome.jar` will be created in the folder named `jars`.
- `--main-class` specifies the fully qualified name of the app's entry point—a class that contains a `main` method.
- `-C` specifies which folder contains the files that should be included in the JAR file and is followed by the files to include—the dot (.) indicates that all files in the folder should be included.

You can simplify the `-create`, `-f` and `--main-class` options in the preceding command with the shorthand notation `-cfe`, followed by the JAR file name and main class, as in:

```
jar -cfe jars/com.deitel.welcome.jar ^
com.deitel.welcome.Welcome ^
-C mods/com.deitel.welcome .
```

16. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. "JEP 261: Module System." <http://openjdk.java.net/jeps/261>.

36.3.8 Running the Welcome App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the entry point, you can execute the app as follows:

```
java --module-path jars -m com.deitel.welcome
```

or

```
java -p jars -m com.deitel.welcome
```

The program executes and displays:

```
Welcome to the Java Platform Module System!
```

If you did not specify the entry point when creating the JAR, you may still run the app by specifying the module name and fully qualified class name, as in:

```
java --module-path jars ^  
-m com.deitel.welcome/com.deitel.welcome.Welcome
```

or

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

36.3.9 Aside: Classpath vs. Module Path

Before Java 9, the compiler and runtime located types via the *classpath*—a list of folders and library archive files containing compiled Java classes. In earlier Java versions, the classpath was defined by a combination of a CLASSPATH environment variable, extensions placed in a special folder of the JRE, and options provided to the javac and java commands.

Because types could be loaded from several different locations, the order in which those locations were searched resulted in brittle apps. For example, many years ago, one of the authors installed a Java app from a third-party vendor on his system. The app's installer placed an old version of a Java library into the JRE's extensions folder. Several Java apps on his system depended on a newer version of that library with additional types and enhanced versions of the library's older types. Because classes in the JRE's extensions folder were loaded *before* other classes on the classpath,¹⁷ the apps that depended on the newer library version stopped working, failing at runtime with NoClassDefFoundErrors and NoSuchMethodErrors—sometimes long after the apps began executing.

The reliable configuration provided by modules and module descriptors helps eliminate many such runtime classpath problems. Every module explicitly states its dependencies and these are resolved *as an app launches*. In Section 36.8.5, we'll show the steps that the JRE's *module resolver* performs at launch time.



Common Programming Error 36.1

The module path may contain only one of each module and every package may be defined in only one module. If two or more modules have the same name or export the same packages, the runtime immediately terminates before running the program.

17. "Understanding Extension Class Loading." <https://docs.oracle.com/javase/tutorial/ext/basics/load.html>.

36.4 Creating and Using a Custom Module

To demonstrate a module that depends on another custom module in addition to standard modules, let's reorganize one of the book's earlier, non-modularized examples. We'll declare Section 8.2's `Time1` and `Time1Test` classes in separate modules, then use class `Time1` from the module containing `Time1Test`. As you'll see, we'll *export* class `Time1`'s package from one module and *require* `Time1`'s enclosing module from a module containing the `Time1Test` class. Figure 36.10 shows the `src` folder structure for the app's two modules.

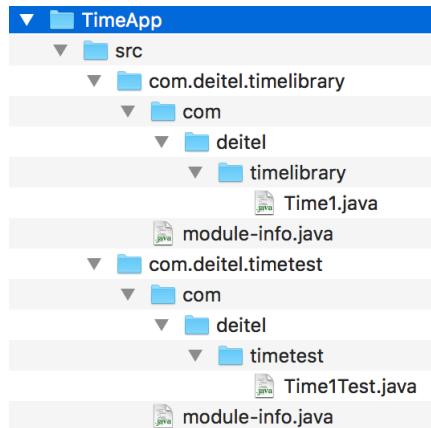


Fig. 36.10 | TimeApp example's `src` folder structure.

36.4.1 Exporting a Package for Use in Other Modules

As you learned previously, every class that you wish to place in a module *must* be declared in a package. For this reason, we added the package statement in line 3 (Fig. 36.11) to class `Time1` (which was originally declared in Fig. 8.1).

```

1 // Fig. 36.11: Time1.java
2 // Class Time1 that will be placed in a module.
3 package com.deitel.timelibrary;
4
5 public class Time1 {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime(int hour, int minute, int second) {
13        // validate hour, minute and second
14        if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
15            second < 0 || second >= 60) {
  
```

Fig. 36.11 | Class `Time1` that will be placed in a module. (Part 1 of 2.)

```

16         throw new IllegalArgumentException(
17             "hour, minute and/or second was out of range");
18     }
19
20     this.hour = hour;
21     this.minute = minute;
22     this.second = second;
23 }
24
25 // convert to String in universal-time format (HH:MM:SS)
26 public String toUniversalString() {
27     return String.format("%02d:%02d:%02d", hour, minute, second);
28 }
29
30 // convert to String in standard-time format (H:MM:SS AM or PM)
31 public String toString() {
32     return String.format("%d:%02d:%02d %s",
33         ((hour == 0 || hour == 12) ? 12 : hour % 12),
34         minute, second, (hour < 12 ? "AM" : "PM"));
35 }
36 }
```

Fig. 36.11 | Class Time1 that will be placed in a module. (Part 2 of 2.)

com.deitel.timelibrary Module Declaration

After placing Time1 in a package, we must declare the **module** via a module declaration (Fig. 36.12). Line 4 indicates that the module *com.deitel.timelibrary* exports the package *com.deitel.timelibrary*. Now the package's **public** classes (in this case, just class Time1) can be used by *any* module that reads the *com.deitel.timelibrary* module, provided that the module can be found on the module path, as you'll see in Section 36.4.3.

```

1 // Fig. 36.12: module-info.java
2 // Module declaration for the com.deitel.timelibrary module
3 module com.deitel.timelibrary {
4     exports com.deitel.timelibrary; // package available to other modules
5 }
```

Fig. 36.12 | Module declaration for the *com.deitel.timelibrary* module.

36.4.2 Using a Class from a Package in Another Module

The app's entry point—class Time1Test (which was originally declared in Fig. 8.2)—also must be packaged for placement in a module (line 3 of Fig. 36.13). In addition, class Time1Test manipulates an object of class Time1, which is declared in a package of another module. For this reason, we import Time1 in line 5.

```

1 // Fig. 36.13: Time1Test.java
2 // Time1 object used in an app.
3 package com.deitel.timetest;
```

Fig. 36.13 | Time1 object used in an app. (Part 1 of 2.)

```
4 import com.deitel.timelibrary.Time1;
5
6 public class Time1Test {
7     public static void main(String[] args) {
8         // create and initialize a Time1 object
9         Time1 time = new Time1(); // invokes Time1 constructor
10
11         // output string representations of the time
12         displayTime("After time object is created", time);
13         System.out.println();
14
15         // change time and output updated time
16         time.setTime(13, 27, 6);
17         displayTime("After calling setTime", time);
18         System.out.println();
19
20         // attempt to set time with invalid values
21         try {
22             time.setTime(99, 99, 99); // all values out of range
23         }
24         catch (IllegalArgumentException e) {
25             System.out.printf("Exception: %s%n%n", e.getMessage());
26         }
27
28         // display time after attempt to set invalid values
29         displayTime("After calling setTime with invalid values", time);
30     }
31
32
33     // displays a Time1 object in 24-hour and 12-hour formats
34     private static void displayTime(String header, Time1 t) {
35         System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
36             header, t.toUniversalString(), t.toString());
37     }
38 }
```

Fig. 36.13 | Time1 object used in an app. (Part 2 of 2.)

com.deitel.timetest Module Declaration

Because class Time1 is located in a package of the com.deitel.timelibrary module, the module containing class Time1Test (com.deitel.timetest) must declare its dependency on that other module. The module declaration (Fig. 36.14) indicates this dependency with the requires directive (line 4). Without this *and* the exports directive in Fig. 36.12, class Time1Test would not be able to import and use class Time1.

```
1 // Fig. 36.14: module-info.java
2 // Module declaration for the com.deitel.timetest module
3 module com.deitel.timetest {
4     requires com.deitel.timelibrary;
5 }
```

Fig. 36.14 | Module declaration for the com.deitel.timetest module.

com.deitel.timetest Module-dependency Graph

Figure 36.15 shows the Time1Test app's module-dependency graph indicating that:

- the module named `com.deitel.timetest` reads `com.deitel.timelibrary` and the standard module `java.base`, and
- the module named `com.deitel.timelibrary` reads the module `java.base`.

To create this graph in NetBeans, we performed the following steps:

1. Created a `TimeLibrary` project containing the `com.deitel.timelibrary` package and `com.deitel.timelibrary`'s `module-info.java` file.
2. Created a `TimeApp` project containing the `com.deitel.timetest` package and `com.deitel.timetest`'s `module-info.java` file.
3. Right clicked the `TimeApp` project's `Libraries` node and selected `Add Project...`, then selected the `TimeLibrary` project and clicked `Add Project JAR Files`—this adds the `TimeLibrary` project's modular JAR file to the `TimeApp` project.
4. Finally, we opened the `TimeApp` project's `module-info.java` file in `Graph` view.

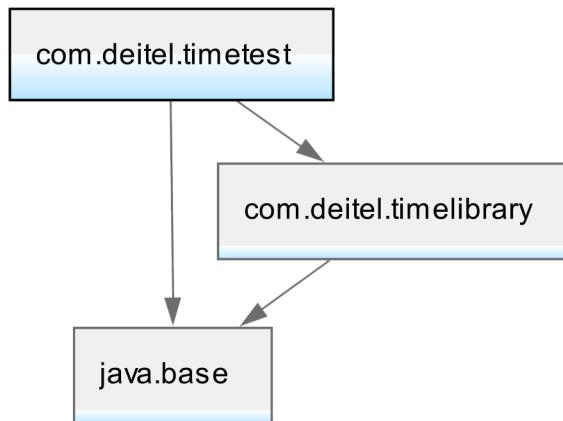


Fig. 36.15 | Module-dependency graph for the `com.deitel.timetest` module.

36.4.3 Compiling and Running the Example

You must compile both modules before running this app. The `com.deitel.timelibrary` module must be compiled first, because `com.deitel.timetest` depends on it. IDEs and other build tools (like Ant, Gradle and Maven) typically can deal with order-of-compilation issues like this for you.

Compiling Module `com.deitel.timelibrary`

To compile the `com.deitel.timelibrary` module, open a command window, use the `cd` command to change to this chapter's `TimeApp` folder on your system, then type:

```
javac -d mods/com.deitel.timelibrary ^
src/com.deitel.timelibrary/module-info.java ^
src/com.deitel.timelibrary/com/deitel/timelibrary/Time1.java
```

Compiling Module com.deitel.timetest

Next, to compile the `com.deitel.timetest` module, type:

```
javac --module-path mods -d mods/com.deitel.timetest ^
    src/com.deitel.timetest/module-info.java ^
    src/com.deitel.timetest/com/deitel/timetest/Time1Test.java
```

Here we added the option `--module-path` to indicate that the `mods` folder contains modules on which the `com.deitel.timetest` module depends—in this case, we previously compiled the `com.deitel.timelibrary` module into the `mods` folder.

Running the Example

Finally, to run this example, type:

```
java --module-path mods ^
    -m com.deitel.timetest/com.deitel.timetest.Time1Test
```

In this command:

- the option `--module-path` indicates where the app's modules are located, and
- the option `-m` specifies which class should be used as the app's entry point—that is, a class containing the `main` method that the JVM calls to launch the app.

For the `main` class, note that you must specify its module name followed by a slash and its fully qualified class name, because the class is now in a package contained in a module. The program's output is shown below:

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

36.4.4 Packaging the App into Modular JAR Files

In this section, we'll package each app into a modular JAR file then run the app. To package `com.deitel.timelibrary` into a modular JAR file, type:

```
jar --create -f jars/com.deitel.timelibrary.jar ^
    -C mods/com.deitel.timelibrary .
```

To package `com.deitel.timetest` into a modular JAR file, type:

```
jar --create -f jars/com.deitel.timetest.jar ^
    --main-class com.deitel.timetest.Time1Test ^
    -C mods/com.deitel.timetest .
```

Running the App from a Modular JAR File

Once you place an app in a modular JAR file for which you've specified the `main` class, you can execute the app as follows:

```
java --module-path jars -m com.deitel.timetest
```

The program executes and displays the same output shown in Section 36.4.3.

36.4.5 Strong Encapsulation and Accessibility

Before Java 9, you could use any `public` class that you imported into your code. Whether you could access the class's members was determined by how they were declared—`public`, `protected`, package access or `private`. Due to Java 9's **strong encapsulation** in modules, `public` types in a module are no longer *accessible* to your code by default—so `public` no longer means available to all:

- If a module exports a package, the `public` types in that package are accessible by *any* module that reads the package's module.
- If a module exports a package to a specific module (via `exports...to`), the `public` types in that package are accessible *only* to the specific module and only if that module *reads* the package's module.
- If a module does not export a package, the `public` types in that package are accessible *only* within their enclosing module.

Once you have access to a type in another module, then the normal rules of `public`, `protected`, package access and `private` apply.

Compilation Error When Attempting to Use an Inaccessible Type

The project `TimeAppMissingExports` in this chapter's `ExamplesShowingErrors` folder demonstrates that *explicitly named modules* have **strong encapsulation** and do not export packages unless you *explicitly* list them in `exports` directives. In this project, we removed the `exports` directive from the `com.deitel.timelibrary`'s module declaration, then recompiled the module. Next, we tried to recompile the `com.deitel.timetest` module. The compiler produced the following error message, which indicates that the package `com.deitel.timelibrary` is not exported and thus is inaccessible:

```
src\com.deitel.timetest\com\deitel\timetest\Time1Test.java:5:  
error: package com.deitel.timelibrary is not visible  
import com.deitel.timelibrary.Time1;  
          ^  
  (package com.deitel.timelibrary is declared in module  
  com.deitel.timelibrary, which is not in the module graph)  
1 error
```



Common Programming Error 36.2

When a requires dependency is not fulfilled by an `exports` clause in another module a compilation error occurs.

36.5 Module-Dependency Graphs: A Deeper Look

Previously, we've shown two module-dependency graphs. Here we continue our discussion of module graphs and show the errors that occur if a module directly or indirectly requires itself—known as a cycle.

36.5.1 java.sql

Figure 36.16 shows the module-dependency graph for a module named `modulegraphtest` that depends on the `java.sql` module, per the following module declaration:

```
module modulegraphtest {
    requires java.sql;
}
```

NetBeans highlights the module declared by the module declaration (`modulegraphtest`) with a thick blue line. It also highlights `java.sql`, because it's *explicitly* listed in a `requires` directive and `java.base`, because it's implicitly required by all modules. The other modules shown (`java.xml` and `java.logging`) are included in the graph, because `java.sql` depends on them.

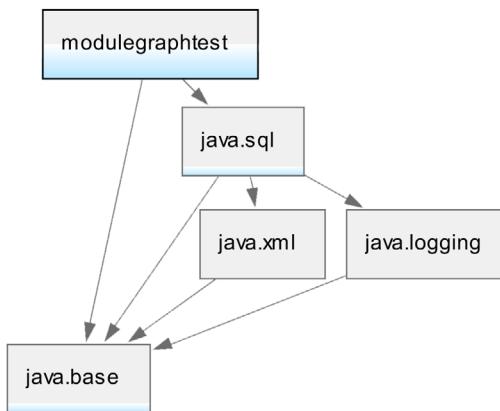


Fig. 36.16 | Dependency graph for a module that depends on `java.sql`.

36.5.2 java.se

Figure 36.17 shows the significantly more complex `java.se` module's dependency graph—this is an **aggregator module** that specifies via `requires transitive` all the modules necessary to support Java SE 9 apps. To produce this graph, we first downloaded the JDK 9 source code, as described at

```
http://hg.openjdk.java.net/jdk9/jdk9/raw-file/tip/common/doc/building.html
```

We then opened the `java.se` module's declaration (located in the source-code folder's `jdk/src/java.se/share/classes` folder) in NetBeans **Graph** view. We rotated the graph 90° for readability. There is also a `java.se.ee` aggregator module, which includes everything in the `java.se` module and additional Java SE modules with packages that overlap with the Java Enterprise Edition (EE) Platform.

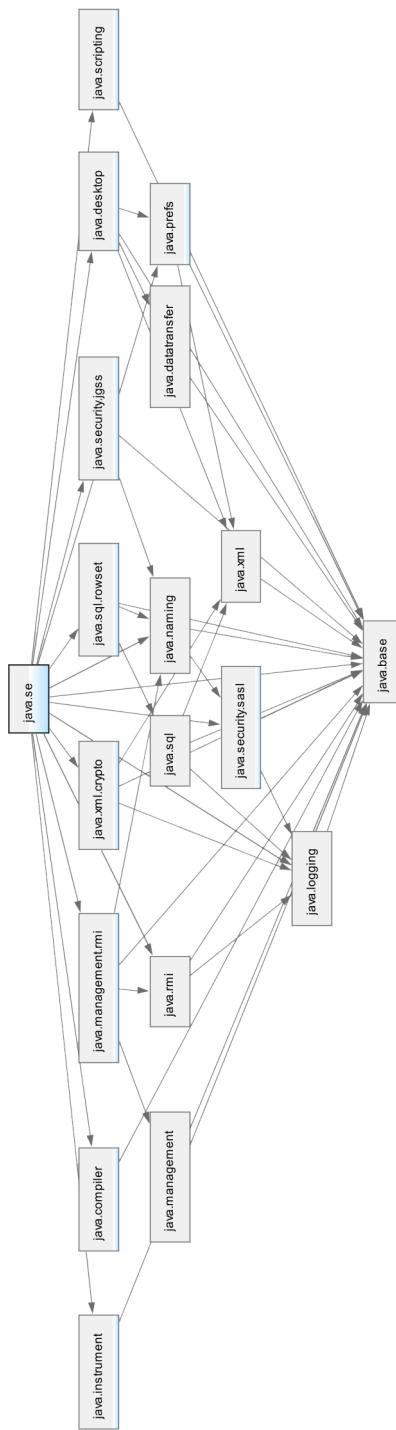


Fig. 36.17 | java.se module-dependency graph.

36.5.3 Browsing the JDK Module Graph

It's interesting to look at the JDK's full module-dependency graph. This is the largest of the module graphs we show. You can view the graph on our website at:

<http://deitel.com/bookresources/jhttp11/ModularJDKGraph.png>

When you open it with your web browser, it will initially display the complete image in the browser's window. Click the image to zoom in, then scroll horizontally and vertically to view the graph's details. We produced this image using the Graphviz tool available from

<http://www.graphviz.org/>

36.5.4 Error: Module Graph with a Cycle

A module is not allowed to directly or indirectly reference itself. Doing so would result in a *cycle* when computing the module's dependency graph.



Common Programming Error 36.3

A compilation error occurs if a module graph contains a cycle.

A Module That (Incorrectly) Requires Itself

Consider the following module declaration in which the module requires itself:

```
module mymodule {
    requires mymodule;
}
```

When you compile this declaration, the following error occurs, indicating a cycle in the module's dependencies:

```
module-info.java:2: error: cyclic dependence involving mymodule
    requires mymodule;
                           ^
1 error
```

Two Modules That (Incorrectly) Require One Another

Similarly, consider a project named CircularDependency containing two modules—`module1` and `module2`—with the structure shown in Fig. 36.18.

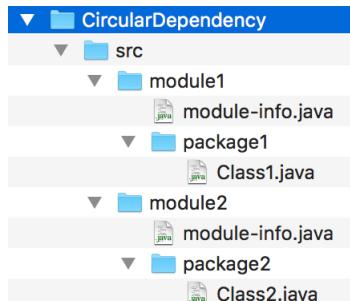


Fig. 36.18 | CircularDependency example's `src` folder structure.

If the module declarations for these two modules indicate that each module requires the other, as in

```
module module1 {  
    exports package1;  
    requires module2;  
}
```

and

```
module module2 {  
    exports package2;  
    requires module1;  
}
```

then, when you compile these modules

```
javac --module-source-path src ^  
    --module-path mods -d mods ^  
    src/module1/module-info.java ^  
    src/module1/package1/Class1.java ^  
    src/module2/module-info.java ^  
    src/module2/package2/Class2.java
```

the compiler again issues an error indicating a cycle in the module dependencies:

```
src\module1\module-info.java:9: error: cyclic dependence involving  
module2  
    requires module2;  
          ^  
1 error
```

Modules in a Cycle Are Really “One Thing”

Ultimately all the modules in a cycle are really one module—not separate modules.¹⁸ While we were writing this chapter, a friend of ours who works for a large organization told us that his group is preparing for Java 9 modularity. He indicated that they have multiple large pre-Java-9 JAR files. Initially they thought they’d make each JAR a separate module, but their JARs turned out to be so interdependent that they’ve decided to combine them into a single module. This kind of interdependency is what leads to cycles in your design. Ideally, when you modularize a previously monolithic system, you want to break that system into separate modules that are easier to maintain and secure. This can pose significant refactoring challenges in large code bases.

36.6 Migrating Code to Java 9

Many pre-Java-9 apps will run unaltered on Java 9. In fact, as we prepared this book, we tested every app using JDK 9 and they all compiled and ran without issue. In Java 9, all programs are compiled and executed using the module system. Java 9 strongly encapsulates types that are not exported by modules, so it’s possible that some apps will fail to compile because types that were accessible to them prior to Java 9 no longer are. For example, there are many pre-Java-9 **internal APIs** that were not meant for use outside the JDK, but were in fact used outside the JDK—many of these are not exported in Java 9 and thus are

18. Alex Buckley, e-mail message to authors, March 24, 2017.

inaccessible.¹⁹ If your code uses such internal APIs directly or indirectly, it will fail to compile.

Some internal APIs considered critically important are still available in Java 9. Various JEPs referenced by JSR 379²⁰ define new public APIs that replace these internal APIs. These internal APIs will eventually be removed.



Software Engineering Observation 36.2

Modularity enables strong encapsulation. Code that is not exported cannot be accessed by other modules.



Error-Prevention Tip 36.1

You can use the `jdeps` tool (Section 36.6.3) released with Java 8 to locate a type's dependencies or the dependencies for all types in a JAR file. In Java 9, the tool also supports modules. The `--jdk-internals` option specifically identifies uses of JDK internal APIs in code. Some pre-Java-9 internal APIs have been placed into packages that are exported in Java 9 and some are now strongly encapsulated. For each internal API that `jdeps` locates, you can review JEP 260 and update your code accordingly.



Common Programming Error 36.4

JDK 9 hides most pre-Java-9 internal APIs, so pre-Java-9 code that uses them will not compile and run on Java 9.

Java is more than two decades old so there's vast amounts of legacy Java code to migrate to Java 9. The module system provides mechanisms that can automatically place your code in modules to help you with migration.

36.6.1 Unnamed Module

In Java 9, all code is required to be placed in modules. When you execute code that's not in a module, the code is loaded from the *classpath* and placed in the **unnamed module**. This is why we can run some non-modularized code in the modularized JDK, but unfortunately without the benefits of modularization.

The unnamed module:

- *implicitly exports* all of its packages, and
- *implicitly reads* all other modules.

However, because the module is *unnamed*, there's no way to refer to it in a `requires` directive from a named module, so a named module cannot depend on the unnamed module.

36.6.2 Automatic Modules

There are enormous numbers of preexisting libraries that you can use in your apps. Many of these are not yet modularized. However, to facilitate migration, you can add *any* library's JAR file to an app's module path, then use the packages in that JAR. When you do, the JAR file implicitly becomes an **automatic module** and can be specified in a `module`

19. Reinhold, Mark. "JEP 260: Encapsulate Most Internal APIs." <http://openjdk.java.net/jeps/260>.

20. Clark, Iris, and Mark Reinhold. "Java SE 9 (JSR 379)." March 6, 2017. <http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>.

declaration's `requires` directives. The JAR's file name—minus the `.jar` extension—becomes its module name, which *must be a valid Java identifier* for use in a `requires` directive. Also, an automatic module:

- *implicitly exports* all of its packages—so, any module that reads the automatic module (including the unnamed module) has access to the `public` types in the automatic module's packages.
- *implicitly reads (requires)* all other modules, including other automatic modules and the unnamed module—so, an automatic module has access to all the `public` types exposed by the system's other modules.

We demonstrate an automatic module in Section 36.7.

36.6.3 jdeps: Java Dependency Analysis

Another tool to help you migrate your code to Java 9 is the **jdeps command**, which was introduced in Java 8 to help you determine a type's *class* and *package* dependencies. A key use of `jdeps` is to locate dependencies on pre-Java-9 internal APIs that are now strongly encapsulated in Java 9. To determine whether a class has any such dependencies, use the following command on your compiled pre-Java-9 code:

```
jdeps --jdk-internals YourClassName.class
```

or if you have many classes in a JAR file, use:

```
jdeps --jdk-internals YourJARName.jar
```

If this command produces no output, then your class or set of classes does not have any dependence on JDK internal APIs that are no longer accessible.



Error-Prevention Tip 36.2

Check every pre-Java-9 compiled class/JAR file with the `jdeps` command to ensure that your code does not depend on JDK internal APIs.

Determining the Modules You Need

Java 9 adds the ability to discover *module* dependencies in Java 9 code. When you're preparing to create *custom* runtimes, you also can use `jdeps` to determine your app's dependencies, so you know which modules to include. For example, this chapter's `Welcome` app depends only on `java.base`. We can confirm that by executing the following command from the `WelcomeApp` folder, which checks the `com.deitel.welcome` module's dependencies:

```
jdeps --module-path jars -m com.deitel.welcome
```

This produces the following output, showing the packages and modules the app uses:

```
com.deitel.welcome
[file:///C:/examples/ch36/WelcomeApp/jars/com.deitel.welcome.jar]
    requires java.base (@9-ea)
com.deitel.welcome -> java.base
    com.deitel.welcome      -> java.io      java.base
    com.deitel.welcome      -> java.lang    java.base
```

The output shows that our module `com.deitel.welcome` depends on the `java.base` module, and that our module specifically uses types from the `java.base` module's `java.io` and `java.lang` packages.

The preceding command may also be written as

```
jdeps jars/com.deitel.welcome.jar
```

In addition, you can use `jdeps` on a specific `.class` file, as in:

```
jdeps mods/com.deitel.welcome/com/deitel/welcome/Welcome.class
```

which produces

```
Welcome.class -> java.base
  com.deitel.welcome      -> java.io          java.base
  com.deitel.welcome      -> java.lang        java.base
```

Verbose jdeps Output

If you'd like more details, you can specify the `-v` (verbose) option as in:

```
jdeps -v jars/com.deitel.welcome.jar
```

which produces:

```
com.deitel.welcome
  [file:///C:/examples/ch36/WelcomeApp/jars/com.deitel.welcome.jar]
    requires java.base (@9-ea)
  com.deitel.welcome -> java.base
    com.deitel.welcome.Welcome      -> java.io.PrintStream      java.base
    com.deitel.welcome.Welcome      -> java.lang.Object        java.base
    com.deitel.welcome.Welcome      -> java.lang.String       java.base
    com.deitel.welcome.Welcome      -> java.lang.System       java.base
```

showing precisely which packages, types and modules the app uses. Knowing that the app requires only `java.base`, we can then use `jlink` to create a custom runtime containing only that module, which we'll do in Section 36.8.

Using jdeps to Produce DOT Files for Graphing Tools

You can use graphing tools—such as Graphviz (www.graphviz.org) and its web-based version (www.webgraphviz.com)—to produce module-dependency graphs using the DOT graph description language,²¹ which specifies a graph's nodes and edges. The `jdeps` tool can create DOT (.dot) files with the `--dot-output` option as in:

```
jdeps --dot-output . jars/com.deitel.welcome.jar
```

which produces two .dot files in the current folder (.):

- `summary.dot`—the description of module `com.deitel.welcome`'s dependencies.
- `com.deitel.welcome.dot`—the description of module `com.deitel.welcome`'s specific package dependencies.

Figure 36.19 shows the graph we produced by opening `summary.dot` in a text editor, then copying and pasting its contents

21. [https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language)).

```
digraph "summary" {
    "com.deitel.welcome" -> "java.base (java.base)";
}
```

into the textbox at webgraphviz.com and clicking **Generate Graph**.²²

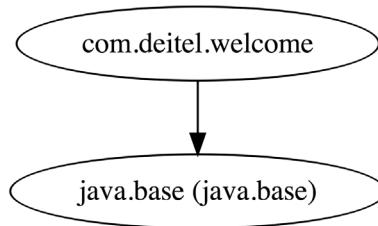


Fig. 36.19 | Webgraphviz.com graph based on summary.dot

Additional jdeps Options

For a complete list of `jdeps` options, visit

```
http://download.java.net/java/jdk9/docs/technotes/tools/windows/jdeps.html
```

for Windows or visit

```
http://download.java.net/java/jdk9/docs/technotes/tools/unix/jdeps.html
```

for macOS and Linux.

36.7 Resources in Modules; Using an Automatic Module

When the types in a module require resources—such images, videos, XML documents and more—those resources should be packaged with the module to ensure that they’re available when the module’s types are used at execution time. This is known as **resource encapsulation**.²³ In this section, we’ll migrate our non-modularized JavaFX VideoPlayer example from Section 22.6 into a module that also encapsulates the app’s resources—the FXML file that describes the GUI and its video file that will be loaded and played at execution time. By convention, resources typically are placed in a folder named `res`.

Recall that the original `VideoPlayer` example consisted of the following files all in Chapter 22’s `VideoPlayer` folder:

- `VideoPlayer.fxml`—The FXML file that describes the app’s GUI.
- `VideoPlayer.java`—The `Application` subclass that begins the app’s execution.
- `VideoPlayerController.java`—The controller class that responds to the GUI’s events and loads the video.

22. The `.dot` extension is also used by Microsoft Word document templates. On systems with Microsoft Word installed, open the `jdeps`-produced `.dot` files directly from a text editor.

23. “Java Platform Module System Requirements.” <http://openjdk.java.net/projects/jigsaw/spec/reqs/#resource-encapsulation>

- `sts117.mp4`—The NASA video²⁴ that the app loads and plays.
- `controlsfx-8.40.12`—The ControlsFX library containing the dialog class `ExceptionDialog`. We display an `ExceptionDialog` if the `MediaPlayer` encounters any errors.

Reorganizing for Modularization

For the purpose of this example, we reorganized the files into the folder structure shown in Fig. 36.20 to support modules. Notice the following about the structure:

- The files `VideoPlayer.fxml` and `sts117.mp4`, which are not Java source code files, are located in the module directory's `res` folder. These files will be read from the module's `res` folder when the app executes.
- As required for modularization, we placed the classes `VideoPlayer` and `VideoPlayerController` in a package—the folder structure `com/deitel/videoplayer` corresponds to the package `com.deitel.videoplayer`.
- As required, we created a `module-info.java` file in the module's root folder.

In addition, we renamed `controlsfx-8.40.12.jar` to `controlsfx.jar` and placed it directly in the `VideoPlayer` folder's `mods` subfolder.

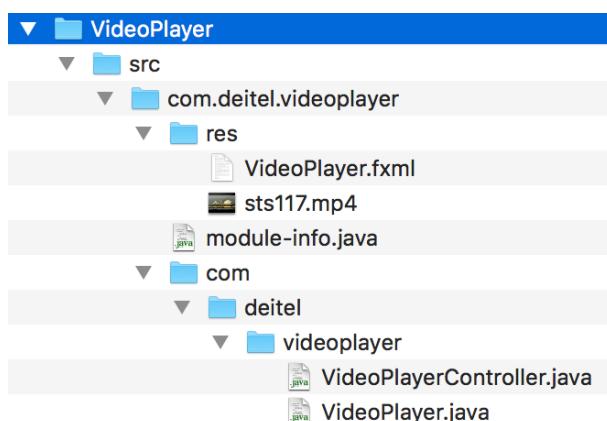


Fig. 36.20 | Modularized `VideoPlayer` `src` folder structure.

36.7.1 Automatic Modules

The ControlsFX library we used when developing the `VideoPlayer` in Section 22.6 was not designed to be a Java module. However, you can add *any* library's JAR file to an app's module path, then use the packages in that JAR. When you do, the JAR file implicitly becomes an **automatic module** and can be specified in a module declaration's `requires` directives. The JAR's file name—minus the `.jar` extension—becomes its module name, which *must be a valid Java identifier* for use in a `requires` directive. This is why we renamed the JAR by removing `-8.40.12` from the original filename. Also, an automatic module:

24. For NASA's terms of use, visit <http://www.nasa.gov/multimedia/guidelines/>.

- *implicitly exports* all of its packages—so, any module that reads the automatic module has access to the `public` types in the automatic module’s packages.
- *implicitly reads* all other modules in the app, including other automatic modules—so, an automatic module has access to all the `public` types exposed by the system’s other modules.

Code Changes for Modularization

We made the following code changes:

- `VideoPlayer.fxml`—We modified the controller class’s name to use its fully qualified name `com.deitel.videoplayer.VideoPlayerController` so that the `FXMLLoader` can find the controller class.
- `VideoPlayer.java`—We changed the name of the FXML file to load from "`VideoPlayer.fxml`" to "`/res/VideoPlayer.fxml`", which indicates that the FXML file is located in the module’s `res` folder. We also added the `package` statement

```
package com.deitel.videoplayer;
```

- `VideoPlayerController.java`—We modified the name of the video file from "`sts117.mp4`" to "`/res/sts117.mp4`", which indicates that the video file is located in the module’s `res` folder. We also added the `package` statement

```
package com.deitel.videoplayer;
```

The rest of the code is identical to what we presented in Section 22.6.

36.7.2 Requiring Multiple Modules

The `com.deitel.videoplayer` module declaration (Fig. 36.21) indicates that the module requires `javafx.controls`, `javafx.fxml`, `javafx.media` and `controlsfx` (the automatic module discussed in Section 36.7.1). The module exports the `com.deitel.videoplayer` package (line 9), because class `VideoPlayerController` is used by class `FXMLLoader` (module `javafx.fxml`) when it creates the controller object and the app’s GUI.

```
1 // Fig. 36.21: module-info.java
2 // Module declaration for the com.deitel.videoplayer module
3 module com.deitel.videoplayer {
4     requires javafx.controls;
5     requires javafx.fxml;
6     requires javafx.media;
7     requires controlsfx; // automatic module for ControlsFX
8
9     exports com.deitel.videoplayer;
10    opens com.deitel.videoplayer to javafx.fxml;
11 }
```

Fig. 36.21 | Module declaration for the `com.deitel.videoplayer` module.

36.7.3 Opening a Module for Reflection

In Fig. 36.21, the `opens...to` directive (line 10) indicates that the accessible types in the package `com.deitel.videoplayer` should be available via *reflection* at runtime to types in

the `javafx.fxml` module. As we discussed in Section 36.2.6, this enables the `FXMLLoader` to locate and load class `VideoPlayerController`. The `FXMLLoader` creates a `VideoPlayerController` object and *injects* into it references to the GUIs components that the `FXMLLoader` creates from the app's FXML file. For one module to *open* a package to another module, that package must first be exported (possibly as a qualified export using `exports...`to).

36.7.4 Module-Dependency Graph

Figure 36.22 shows the `com.deitel.videoplayer` module-dependency graph. Again, the ones with light blue highlights are explicitly specified in requires directives—except for `java.base`, which is implicitly required by all modules. The other modules shown are dependencies of the modules specified in the `requires` directives.



Fig. 36.22 | `com.deitel.videoplayer` module-dependency graph.

36.7.5 Compiling the Module

To compile the `com.deitel.videoplayer` module, type:

```

javac --module-path mods -d mods/com.deitel.videoplayer ^
src/com.deitel.videoplayer/module-info.java ^
src/com.deitel.videoplayer/com/deitel/videoplayer/*.java
  
```

Note that we included the `--module-path` option, because the `mods` folder contains `controlsfx.jar`—the automatic module that is required to compile this app.

Copying the Resource Files into the Module

Though some IDEs and build tools will automatically put the module's resources into the compiled module, the preceding `javac` command does not. Once you've compiled the module, copy the `res` folder from this project's `src/com/deitel/videoplayer` folder into the `mods/com/deitel/videoplayer` folder.

36.7.6 Running a Modularized App

To execute class `VideoPlayer` from the `com.deitel.videoplayer` module, type:

```
java --module-path mods ^
-m com.deitel.videoplayer/com.deitel.videoplayer.VideoPlayer
```

Figure 36.23 shows the app executing on Windows.

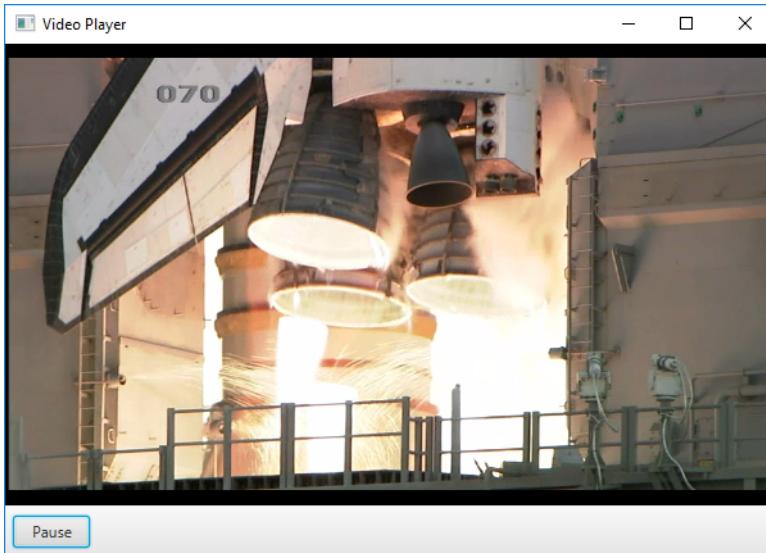


Fig. 36.23 | Modularized `VideoPlayer` app executing.

36.8 Creating Custom Runtimes with `jlink`

A new tool in JDK 9 is the **`jlink` command**—Java's linker for creating custom runtime images.²⁵ In a custom runtime, you can include just what's necessary for a given app or set of apps to execute. For example, if you're creating a runtime for a device that does not support GUIs, you can create a runtime without the corresponding modules that support Swing and JavaFX. In fact, many of this book's text-only, command-line examples can execute on a runtime that contains only the `java.base` module.

25. Denise, Jean-Francois. "JEP 282: jlink: The Java Linker." <http://openjdk.java.net/jeps/282>.

36.8.1 Listing the JRE's Modules

With modularization the JRE is a proper subset of the JDK.²⁶ If you run the command:

```
java --list-modules
```

from the JRE's `bin` folder, the result contains only the JRE's 73 modules (Fig. 36.24), rather than the full listing of the JDK's 95 modules. This number will change as Java evolves. In Section 36.8.3, we do this on a custom runtime produced with the `jlink` command—in that case, only the single module bundled with that runtime will be displayed.



Software Engineering Observation 36.3

You can use the modularized Java platform to conveniently form custom runtimes for smaller capacity devices.

java.activation@9-ea	jdk.charsets@9-ea
java.base@9-ea	jdk.crypto.cryptoki@9-ea
java.compiler@9-ea	jdk.crypto.ec@9-ea
java.corba@9-ea	jdk.crypto.mscapi@9-ea
java.datatransfer@9-ea	jdk.deploy@9-ea
java.desktop@9-ea	jdk.deploy.controlpanel@9-ea
java.instrument@9-ea	jdk.dynalink@9-ea
java.jnlp@9-ea	jdk.httpserver@9-ea
java.logging@9-ea	jdk.incubator.httpclient@9-ea
java.management@9-ea	jdk.internal.le@9-ea
java.management.rmi@9-ea	jdk.internal.vm.ci@9-ea
java.naming@9-ea	jdk.javaws@9-ea
java.prefs@9-ea	jdk.jdwp.agent@9-ea
java.rmi@9-ea	jdk.jfr@9-ea
java.scripting@9-ea	jdk.jsobject@9-ea
java.se@9-ea	jdk.localedata@9-ea
java.se.ee@9-ea	jdk.management@9-ea
java.security.jgss@9-ea	jdk.management.agent@9-ea
java.security.sasl@9-ea	jdk.naming.dns@9-ea
java.smartcardio@9-ea	jdk.naming.rmi@9-ea
java.sql@9-ea	jdk.net@9-ea
java.sql.rowset@9-ea	jdk.pack@9-ea
java.transaction@9-ea	jdk.plugin@9-ea
java.xml@9-ea	jdk.plugin.dom@9-ea
java.xml.bind@9-ea	jdk.plugin.server@9-ea
java.xml.crypto@9-ea	jdk.scripting.nashorn@9-ea
java.xml.ws@9-ea	jdk.scripting.nashorn.shell@9-ea
java.xml.ws.annotation@9-ea	jdk.sctp@9-ea
javafx.base@9-ea	jdk.security.auth@9-ea
javafx.controls@9-ea	jdk.security.jgss@9-ea
javafx.deploy@9-ea	jdk.snmp@9-ea
javafx.fxml@9-ea	jdk.unsupported@9-ea
javafx.graphics@9-ea	jdk.xml.dom@9-ea
javafx.media@9-ea	jdk.zipfs@9-ea
javafx.swing@9-ea	oracle.desktop@9-ea
javafx.web@9-ea	oracle.net@9-ea
jdk.accessibility@9-ea	

Fig. 36.24 | Output of `java --list-modules` showing the modules that compose the JRE.

26. Brian Goetz, e-mail message to authors, March 20, 2017.

36.8.2 Custom Runtime Containing Only `java.base`

For the purpose of this example, change to the `WelcomeApp` folder—after creating the custom runtime, you’ll execute the `Welcome` app using it. The following command creates a runtime containing only the module `java.base`:

```
jlink --module-path "%JAVA_HOME%"/jmods --add-modules java.base ^
--output javabaseruntime
```

The commands options are as follows:

- `--module-path` specifies one or more folders in which to locate the modules that will be included in the runtime—in this case, the JDK’s `jmods` folder, which contains the modular JAR files for all of the JDK’s modules.
- `--add-modules` specifies which modules to include in the runtime—in this case, just `java.base`.
- `--output` specifies the folder in which the runtime’s contents are placed—in this case, the folder `javabaseruntime`. This folder will be placed in the folder from which you execute the preceding command (unless you specify additional path information). If the folder already exists, an error occurs.

This runtime can execute an app that depends only on types from the packages in module `java.base`, including many of this book’s command-line apps.

Note Regarding the `JAVA_HOME` Variable

The `JAVA_HOME` environment variable must refer to JDK 9’s installation folder on your system—see the Before You Begin section before the preface for information on configuring this environment variable. On Windows, you specify `%JAVA_HOME%` to use `JAVA_HOME`’s value in a command. Linux and macOS users should replace `%JAVA_HOME%` with `$JAVA_HOME`. So, for example, the preceding command on Linux and macOS would be:

```
jlink --module-path "$JAVA_HOME"/jmods --add-modules java.base \
--output javabaseruntime
```

In either case, if the path contains spaces, place the environment variable in quotes ("").

Executing the `Welcome` App Using This Custom Runtime

To run the app with the custom runtime, on Windows use:

```
javabaseruntime\bin\java --module-path mods ^
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

or on macOS/Linux use:

```
javabaseruntime/bin/java --module-path mods \
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

The program executes and displays:

```
Welcome to the Java Platform Module System!
```

Listing the Modules in a Custom Runtime

Previously we used the command

```
java --list-modules
```

to list all the modules in the JDK. Once you have a custom runtime, you can use the `java` command from the custom runtime's `bin` folder to confirm the modules it includes, as in:

```
javabaseruntime\bin\java --list-modules
```

When executing the custom runtime's `java` command, use \ to separate folder names on Windows and / to separate the folder names on macOS and Linux. The preceding command produces the following output:

```
java.base@9-ea
```

Similarly the following command creates a custom runtime containing only the module `java.desktop` and any other modules on which it depends:

```
jlink --module-path "%JAVA_HOME%"/jmods ^
--add-modules java.desktop --output javadesktopruntime
```

For this custom runtime, running

```
javadesktopruntime\bin\java --list-modules
```

(again, use forward slashes on macOS and Linux) produces the following output

```
java.base@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.prefs@9-ea
java.xml@9-ea
```

36.8.3 Creating a Custom Runtime for the Welcome App

To create a custom runtime containing only the modules `com.deitel.welcome` and its dependencies (in this case, `java.base`), use:

```
jlink --module-path jars;"%JAVA_HOME%"/jmods ^
--add-modules com.deitel.welcome --output welcomeruntime
```

This creates a custom runtime in the folder `welcomeruntime`. The preceding command specifies multiple folders—`jars` and `%JAVA_HOME%`. On Windows, the path-separator character for lists of folders is a semicolon (;). Linux and macOS users should replace the semicolons in the commands with the colon (:) path-separator character, as in

```
jlink --module-path jars:"$JAVA_HOME"/jmods \
--add-modules com.deitel.welcome --output welcomeruntime
```

To see the list of modules included in the custom runtime, on Windows use:

```
welcomeruntime\bin\java --list-modules
```

(again, use forward slashes on macOS and Linux) which produces the following list of modules:

```
com.deitel.welcome
java.base@9-ea
```

36.8.4 Executing the Welcome App Using a Custom Runtime

To run the app with the custom runtime, on Windows use:

```
welcomeruntime\bin\java -m com.deitel.welcome
```

(Again, use forward slashes on macOS and Linux.) The program executes and displays:

```
Welcome to the Java Platform Module System!
```

36.8.5 Using the Module Resolver on a Custom Runtime

When you run a modularized app, the JVM uses a **module resolver** to determine which modules are required at execution time and ensure that their dependencies are satisfied—this is known as the **transitive closure** of those modules. To locate modules, the module resolver looks at the **observable modules**—that is, those built into the runtime (like `java.base`) and those located on the module path. For a required module that cannot be found, the runtime throws a `java.lang.module.FindException`.

For a given app and runtime, you can view the steps the module resolver follows to determine module dependencies and ensure that the required modules are available to the program. To do so, include `-Xdiag:resolver option27` in the `java` command, as in:

```
welcomeruntime\bin\java -Xdiag:resolver -m com.deitel.welcome
```

(Again, use forward slashes on macOS and Linux.) This uses the custom `welcomeruntime`'s `java` command to display the resolver's steps for locating modules, followed by the program's output:

```
[Resolver] Root module com.deitel.welcome located
[Resolver]   (jrt:/com.deitel.welcome)
[Resolver] Module java.base located, required by com.deitel.welcome
[Resolver]   (jrt:/java.base)
[Resolver] Result:
[Resolver]   com.deitel.welcome
[Resolver]   java.base
Welcome to the Java Platform Module System!
```

The module-resolution process for the `Welcome` app proceeds as follows:

1. First, the resolver locates the app's **initial module**—`com.deitel.welcome`—containing the app's entry point. The resolver refers to this as the *root module*. This is the root node in the module-dependency graph.
2. Next, the resolver locates `java.base`, because the `com.deitel.welcome` module descriptor specifies that `com.deitel.welcome` requires `java.base`.
3. Since `java.base` does not depend on other modules, the dependency graph is now complete and the resolver displays the resulting list of modules required to execute the program.

Next, the program executes and displays its output. If a required module were not found during this process, a `java.lang.module.FindException` would be displayed in this output and the program would not execute.

36.9 Services and ServiceLoader

In Section 10.13, we discussed “programming to an interface, not an implementation” as a mechanism for creating loosely coupled objects. We'll use these concepts in this section

27. Bateman, Alan, Alex Buckley, Jonathan Gibbons and Mark Reinhold. “JEP 261: Module System.” <http://openjdk.java.net/jeps/261>.

as we introduce services and class `ServiceLoader`, which help you create loosely coupled system components. This can make large-scale systems easier to develop and maintain.

MathTutor App

We'll develop a `MathTutor` app (consisting of three modules) that supports various types of randomly generated math problems. Rather than hard-coding these into the app, we'll load math problems through a *service-provider interface* that describes how to obtain math problems. We'll then define two *service providers*—classes that implement this interface. One service provider will create addition problems and the other multiplication problems. At runtime, we'll load objects of these service-provider implementation classes and use them. The completed app structure consisting of three modules is shown in Fig. 36.25.

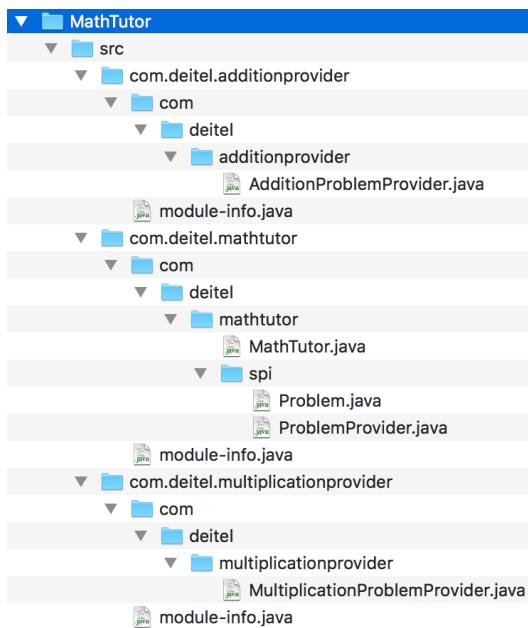


Fig. 36.25 | Folder structure for the `MathTutor` app's modules.

MathTutor App's Modules

Module `com.deitel.mathtutor` aggregates two related packages:

- `com.deitel.mathtutor`: This package contains class `MathTutor`—a command-line app that displays random math problems to the user, inputs the user's responses and displays whether each response is correct or incorrect.
- `com.deitel.mathtutor.spi`: This package contains the `ProblemProvider` service-provider interface and the supporting abstract class `Problem`, which represents a math problem. Class `MathTutor` uses `ProblemProviders` to obtain `Problem` objects.

Module `com.deitel.additionprovider` contains a package of the same name in which we declare class `AdditionProblemProvider`. This implementation of the service-provider interface `ProblemProvider` generates random addition `Problems`.

Module `com.deitel.multiplicationprovider` contains a package of the same name in which we declare class `MultiplicationProblemProvider`. This implementation of the service-provider interface `ProblemProvider` generates random multiplication Problems.

How We'll Demonstrate the App

We'll initially run the `MathTutor` app without placing the service-provider implementation modules on the module path to demonstrate what happens when *no* service providers are found at runtime. Next, we'll "plug in" the module `com.deitel.additionprovider` on the module path, then re-run the app to demonstrate that we're able to obtain Problems from an `AdditionProblemProvider`. Finally, we'll "plug in" both the `com.deitel.additionprovider` and `com.deitel.multiplicationprovider` modules on the module path, then re-run the app to demonstrate that we're able to obtain Problems generated by both an `AdditionProblemProvider` and a `MultiplicationProblemProvider`.

Plug-in Architecture

This "plug-in" architecture using a service-provider interface and multiple service-provider implementations makes the `MathTutor` app easy to extend. Simply create a module containing a `ProblemProvider` implementation, then add it to the module path when you run the app. It also makes the app more configurable, because you can choose which modules to include on the module path when you execute the app.

Reliable Configuration

The mechanisms for creating loosely coupled systems like the `MathTutor` app have been used extensively in Java since its early versions. A key new concept in Java 9—which also applies to modules in general—is *reliable configuration*. For the `MathTutor` app to be able to display Problems to the user, it must be able to locate and load `ProblemProvider` implementations. As you'll see, module declarations enable you to specify which service-provider interfaces a module uses and whether a module contains types that implement those interfaces.

36.9.1 Service-Provider Interface

The package `com.deitel.mathtutor.spi` contains the `com.deitel.mathtutor` module's service-provider interface `ProblemProvider` and the supporting abstract class `Problem`. The final component of this package's name—`spi`—is commonly used in packages that declare one or more service-provider interfaces. Interface `ProblemProvider` (Fig. 36.26) declares method `getProblem` (line 6) that returns a `Problem`.

```
1 // Fig. 36.26: ProblemProvider.java
2 // Service-provider interface for obtaining a Problem
3 package com.deitel.mathtutor.spi;
4
5 public interface ProblemProvider {
6     public Problem getProblem();
7 }
```

Fig. 36.26 | Service-provider interface for obtaining a Problem.

Abstract class `Problem` (Fig. 36.27) provides the common features of math problems in this example. Each has two `int` operands and an `int` result as well as a `String` representing the operation—the `MathTutor` displays this `String` with each math problem it presents to the user. Class `Problem`'s abstract method `getResult` is overridden in each service-provider implementation's concrete subclass of `Problem`.

```

1 // Fig. 36.27: Problem.java
2 // Problem superclass that contains information about a math problem.
3 package com.deitel.mathtutor.spi;
4
5 public abstract class Problem {
6     private int leftOperand;
7     private int rightOperand;
8     private int result;
9     private String operation;
10
11    // constructor
12    public Problem(int leftOperand, int rightOperand, String operation) {
13        this.leftOperand = leftOperand;
14        this.rightOperand = rightOperand;
15        this.operation = operation;
16    }
17
18    // gets the leftOperand
19    public int getLeftOperand() {return leftOperand;}
20
21    // gets the rightOperand
22    public int getRightOperand() {return rightOperand;}
23
24    // gets the operation
25    public String getOperation() {return operation;}
26
27    // gets the result
28    public abstract int getResult();
29 }
```

Fig. 36.27 | `Problem` superclass that contains information about a math problem.

36.9.2 Loading and Consuming Service Providers

Class `MathTutor` (Fig. 36.28) is the app's entry point. It provides the logic for locating and loading `ProblemProvider` implementations, then using them to present math problems to the user.

```

1 // Fig. 36.28: MathTutor.java
2 // Math tutoring app using ProblemProviders to display math problems.
3 package com.deitel.mathtutor;
4
5 import java.util.List;
6 import java.util.Random;
```

Fig. 36.28 | Math tutoring app using `ProblemProviders` to display math problems. (Part I of 3.)

```
7 import java.util.Scanner;
8 import java.util.ServiceLoader;
9 import java.util.ServiceLoader.Provider;
10 import java.util.stream.Collectors;
11 import com.deitel.mathtutor.spi.Problem;
12 import com.deitel.mathtutor.spi.ProblemProvider;
13
14 public class MathTutor {
15     private static Scanner input = new Scanner(System.in);
16
17     public static void main(String[] args) {
18         // get a service loader for ProblemProviders
19         ServiceLoader<ProblemProvider> serviceLoader =
20             ServiceLoader.load(ProblemProvider.class);
21
22         // get the list of service providers
23         List<Provider<ProblemProvider>> providersList =
24             serviceLoader.stream().collect(Collectors.toList());
25
26         // check whether there are any providers
27         if (providersList.isEmpty()) {
28             System.out.println(
29                 "Terminating MathTutor: No problem providers found.");
30             return;
31         }
32
33         boolean shouldContinue = true;
34         Random random = new Random();
35
36         do {
37             // choose a ProblemProvider at random
38             ProblemProvider provider =
39                 providersList.get(random.nextInt(providersList.size())).get();
40
41             // get the Problem
42             Problem problem = provider.getProblem();
43
44             // display the problem to the user
45             showProblem(problem);
46         } while (playAgain());
47     }
48
49     // show the math problem to the user
50     private static void showProblem(Problem problem) {
51         String problemStatement = String.format("What is %d %s %d? ",
52             problem.getLeftOperand(), problem.getOperation(),
53             problem.getRightOperand());
54
55         // display problem and get answer from user
56         System.out.printf(problemStatement);
57         int answer = input.nextInt();
58     }
```

Fig. 36.28 | Math tutoring app using ProblemProviders to display math problems. (Part 2 of 3.)

```
59     while (answer != problem.getResult()) {
60         System.out.println("Incorrect. Please try again: ");
61         System.out.printf(problemStatement);
62         answer = input.nextInt();
63     }
64
65     System.out.println("Correct!");
66 }
67
68 // play again?
69 private static boolean playAgain() {
70     System.out.printf("Try another? y to continue, n to terminate: ");
71     String response = input.next();
72
73     return response.toLowerCase().startsWith("y");
74 }
75 }
```

Fig. 36.28 | Math tutoring app using ProblemProviders to display math problems. (Part 3 of 3.)

Using ServiceLoader to Locate Service Providers

Lines 19–20

```
ServiceLoader<ProblemProvider> serviceLoader =
    ServiceLoader.load(ProblemProvider.class);
```

create a **ServiceLoader** (package `java.util`) that loads `ProblemProvider` implementations. `ServiceLoader`'s static `load` method receives as its argument the `Class` object representing the service-provider interface's type—`ProblemProvider.class` is a **class literal** that's equivalent to creating a `Class<ProblemProvider>` object, as in:

```
new Class<ProblemProvider>()
```

Method `load` returns a `ServiceLoader<ProblemProvider>` that knows only how to load `ProblemProvider` implementations.

There are several ways to get service-provider implementations from a `ServiceLoader`. In lines 23–24

```
List<Provider<ProblemProvider>> providersList =
    serviceLoader.stream().collect(Collectors.toList());
```

we obtain a `List` of the available service-provider implementations using `ServiceLoader`'s `stream` method. This returns a `Stream<Provider<ProblemProvider>>` representing all the available `ProblemProvider` implementations, if any. Interface `Provider` (imported at line 9) is a nested type of class `ServiceLoader`. For each available `ProblemProvider` implementation, the stream contains one `Provider<ProblemProvider>` object. Line 24 uses `Stream` method `collect` and the predefined `Collector` defined by `Collectors.toList` to get the `List` containing all the available implementations. If that `List` is empty (line 27) the program displays an appropriate message and terminates.

Using a Service-Provider Interface

If the `List` contains any service-provider implementations, lines 36–46 use them to display one math problem at a time to the user. Lines 38–39

```
ProblemProvider provider =
    providersList.get(random.nextInt(providersList.size())).get();
```

randomly select one Provider<ProblemProvider> object from the providersList, then invoke that object's get method to obtain its ProblemProvider. Line 42

```
Problem problem = provider.getProblem();
```

then gets a Problem from whichever ProblemProvider was selected.

Note the *loose coupling* of the MathTutor app and its ProblemProviders. The app does not refer in any way to AdditionProblemProviders or MultiplicationProblemProviders that generate math problems.

36.9.3 uses Module Directive and Service Consumers

Figure 36.29 shows the com.deitel.mathtutor module declaration. Note that this module exports the package com.deitel.mathtutor.spi containing the service-provider interface ProblemProvider and its supporting Problem class. This enables modules that implement interface ProblemProvider to access those types. The new feature in this declaration is the **uses module directive** (line 6). This directive indicates that there is a type in the com.deitel.mathtutor module that *uses* objects which implement the ProblemProvider interface. Such a module is called a **service consumer**.

```
1 // Fig. 36.29: module-info.java
2 // Module declaration for the com.deitel.mathtutor module
3 module com.deitel.mathtutor {
4     exports com.deitel.mathtutor.spi; // package for provider interface
5
6     uses com.deitel.mathtutor.spi.ProblemProvider;
7 }
```

Fig. 36.29 | Module declaration for the com.deitel.mathtutor module.

To be able to consume ProblemProviders, the ServiceLoader must be able to locate and load their implementations dynamically using Java's *reflection* capabilities. When you run this app, the module resolver will see in the descriptor that this module *uses* ProblemProvider implementations and thus is dependent on such providers. It will then search the modules on the module path looking for any modules that provide implementations of this interface. If it finds any such modules, it will add them to the module-dependency graph.

36.9.4 Running the App with No Service Providers

To compile the com.deitel.mathtutor module, type:

```
javac -d mods/com.deitel.mathtutor ^
    src/com/deitel/mathtutor/module-info.java ^
    src/com/deitel/mathtutor/com/deitel/mathtutor/MathTutor.java ^
    src/com/deitel/mathtutor/com/deitel/mathtutor/spi/*.java
```

Next, run the app with no `ProblemProvider` implementations on the module path by using the following java command, which places only the `com.deitel.mathtutor` module on the module path:

```
java --module-path mods/com.deitel.mathtutor ^
-m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The result is

```
Terminating MathTutor: No problem providers found.
```

36.9.5 Implementing a Service Provider

Next, let's create class `AdditionProblemProvider` (Fig. 36.30), which implements the service-provider interface `ProblemProvider` (line 10). This class's `com.deitel.additionprovider` package will be placed in the `com.deitel.additionprovider` module (Section 36.9.6). We import interface `ProblemProvider` and class `Problem` from the `com.deitel.mathtutor` module's exported package `com.deitel.mathtutor.spi` (lines 7–8). When the `MathTutor` calls an `AdditionProblemProvider`'s `getProblem` method (lines 14–23), the method creates an anonymous subclass of `Problem` (lines 16–22), passing to `Problem`'s constructor two random int values as the operands and the String "+" as the operation. Lines 18–21 override superclass `Problem`'s `getResult` method to return the sum of the left and right operands.

```

1 // Fig. 36.30: AdditionProblemProvider.java
2 // AdditionProblemProvider implementation of interface
3 // ProblemProvider for the MathTutor app.
4 package com.deitel.additionprovider;
5
6 import java.util.Random;
7 import com.deitel.mathtutor.spi.Problem;
8 import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class AdditionProblemProvider implements ProblemProvider {
11     private static Random random = new Random();
12
13     // returns a new addition problem
14     @Override
15     public Problem getProblem() {
16         return new Problem(random.nextInt(10), random.nextInt(10), "+");
17         // override getResult to add the operands
18     }
19     @Override
20     public int getResult() {
21         return getLeftOperand() + getRightOperand();
22     }
23 }
24 }
```

Fig. 36.30 | `AdditionProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

36.9.6 provides...with Module Directive and Declaring a Service Provider

Figure 36.31 shows the `com.deitel.additionprovider` module declaration. Note that this module requires the module `com.deitel.mathtutor`. Recall from Fig. 36.29 that this module exports the package `com.deitel.mathtutor.spi` containing the types used in class `AdditionProblemProvider`. The new feature in this module declaration is the **provides...with module directive**. Lines 6–7 specify that this module

- provides an implementation of interface `ProblemProvider`—declared in the `com.deitel.mathtutor` module's `com.deitel.mathtutor.spi` package
- with class `AdditionProblemProvider`—declared in this module's `com.deitel.additionprovider` package.

Such a module is called a **service provider**. The directive's **provides** part is followed by the name of an interface or abstract class that's specified in a module's **uses** directive. The directive's **with** part is followed by the name of a class that **implements** the interface or **extends** the abstract class.

```
1 // Fig. 36.31: module-info.java
2 // Module declaration for the com.deitel.additionprovider module
3 module com.deitel.additionprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider
7         with com.deitel.additionprovider.AdditionProblemProvider;
8 }
```

Fig. 36.31 | Module declaration for the `com.deitel.additionprovider` module.

36.9.7 Running the App with One Service Provider

Next, we'll run the app with the `AdditionProblemProvider` included in the module path. First, compile the `com.deitel.additionprovider` module, as follows:

```
javac --module-path mods -d mods/com.deitel.additionprovider ^
    src/com.deitel.additionprovider/module-info.java ^
    src/com.deitel.additionprovider/com/deitel/additionprovider/ ^
        AdditionProblemProvider.java
```

Then run the app with the following java command:

```
java --module-path mods ^
    -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following sample output shows addition problems:

```
What is 9 + 6? 15
Correct!
Try another? y to continue, n to terminate: y
What is 2 + 6? 7
Incorrect. Please try again:
What is 2 + 6? 8
Correct!
Try another? y to continue, n to terminate: n
```

36.9.8 Implementing a Second Service Provider

Class `MultiplicationProblemProvider` (Fig. 36.32) also implements the service-provider interface `ProblemProvider` (line 10). This class's `com.deitel.multiplicationprovider` package will be placed in the `com.deitel.multiplicationprovider` module (Fig. 36.33). Class `MultiplicationProblemProvider` is nearly identical to class `AdditionProblemProvider`, except that line 16 passes the String "*" for the Problem's operation and the overridden `Problem` method `getResult` returns the product of the left and right operands.

```

1 // Fig. 36.32: MultiplicationProblemProvider.java
2 // MultiplicationProblemProvider implementation of interface
3 // ProblemProvider for the MathTutor app.
4 package com.deitel.multiplicationprovider;
5
6 import java.util.Random;
7 import com.deitel.mathtutor.spi.Problem;
8 import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class MultiplicationProblemProvider implements ProblemProvider {
11     private static Random random = new Random();
12
13     // returns a new addition problem
14     @Override
15     public Problem getProblem() {
16         return new Problem(random.nextInt(10), random.nextInt(10), "*");
17         // override getResult to add the operands
18         @Override
19         public int getResult() {
20             return getLeftOperand() * getRightOperand();
21         }
22     };
23 }
24 }
```

Fig. 36.32 | `MultiplicationProblemProvider` implementation of interface `ProblemProvider` for the `MathTutor` app.

Figure 36.33 shows the `com.deitel.multiplicationprovider` module declaration. Again, this module requires the module `com.deitel.mathtutor`. Lines 6–7 specify that this module provides an implementation of the `ProblemProvider` interface with class `MultiplicationProblemProvider`.

```

1 // Fig. 36.33: module-info.java
2 // Module declaration for the com.deitel.multiplicationprovider module
3 module com.deitel.multiplicationprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider with
7         com.deitel.multiplicationprovider.MultiplicationProblemProvider;
8 }
```

Fig. 36.33 | Module declaration for the `com.deitel.multiplicationprovider` module.

36.9.9 Running the App with Two Service Providers

Next, we'll run the app with both the `AdditionProblemProvider` and the `MultiplicationProblemProvider` included in the module path. First, compile the `com.deitel.multiplicationprovider` module, as follows:

```
javac --module-path mods ^
    -d mods/com.deitel.multiplicationprovider ^
    src/com/deitel/multiplicationprovider/module-info.java ^
    src/com/deitel/multiplicationprovider/com/deitel/
        multiplicationprovider/MultiplicationProblemProvider.java
```

Then run the app with the following java command:

```
java --module-path mods ^
    -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

The following is a sample output showing both addition and multiplication problems:

```
What is 4 * 8? 20
Incorrect. Please try again:
What is 4 * 8? 32
Correct!
Try another? y to continue, n to terminate: y
What is 3 * 6? 18
Correct!
Try another? y to continue, n to terminate: y
What is 3 + 7? 10
Correct!
Try another? y to continue, n to terminate: y
What is 9 + 3? 12
Correct!
Try another? y to continue, n to terminate: n
```

36.10 Wrap-Up

In this chapter, we introduced Java 9's new Java Platform Module system. We introduced key modularity concepts you're likely to use when building large-scale systems.

You saw that all modules implicitly depend on `java.base`. You created module declarations that specify a module's dependencies (with the `requires` directive), which packages a module makes available to other modules (with the `exports` directive), services it offers (with the `provides...with` directive), services it consumes (with the `uses` directive) and to what other modules it allows reflection (with the `open` modifier and the `opens` and `opens...to` directives).

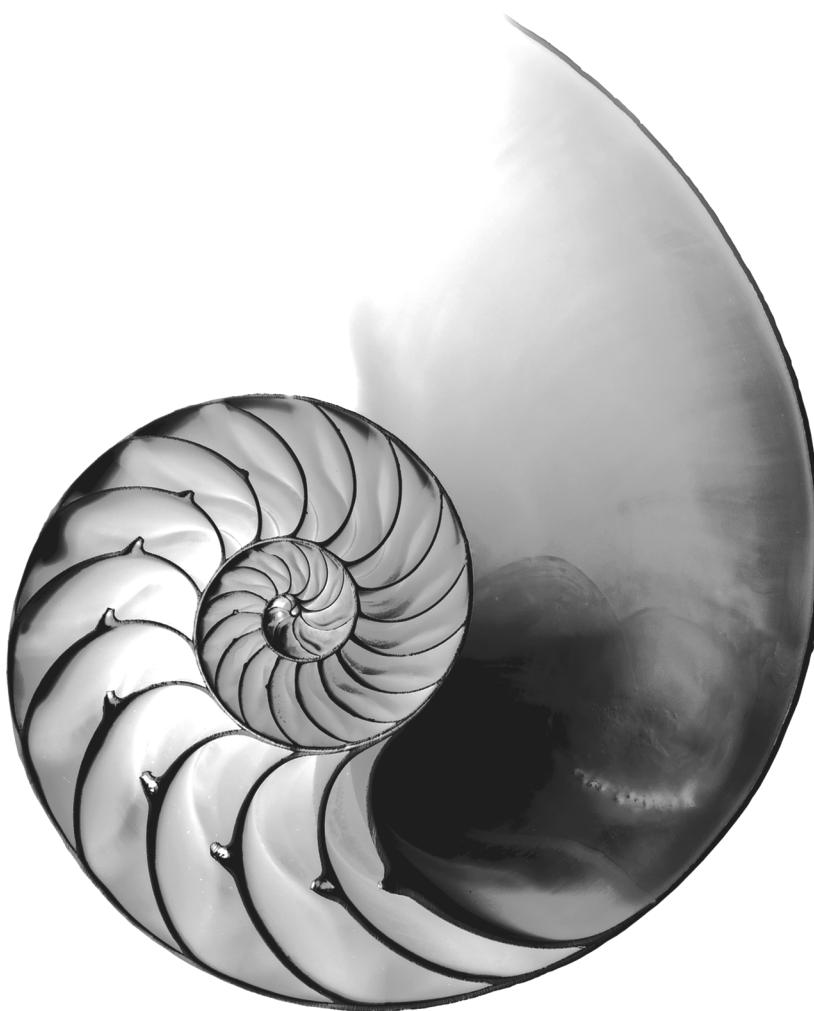
To help you visualize the dependencies among modules, we showed several module-dependency graphs that we created using the NetBeans IDE's JDK 9 support. We discussed the steps that the runtime's module resolver performs to ensure that a module's dependencies are fulfilled.

You used JDK 9's new `jlink` tool (the Java linker) to create smaller custom runtimes, then used them to execute apps. We discussed the module system's strong encapsulation and showed the steps required to explicitly allow runtime reflection via the `open` modifier or the `opens` and `opens...to` directives in a module declaration.

We discussed the enormous amount of non-modularized legacy code that will need to be migrated to modular Java 9, then showed how the unnamed module and automatic modules can help make migration straightforward. We used the `jdeps` tool to determine code dependencies among modules and showed how the tool can be used to check for uses of pre-Java-9 internal APIs (which are for the most part strongly encapsulated in Java 9).

Finally, we discussed services and service providers for building loosely coupled systems by using service-provider interfaces and implementations and the `ServiceLoader` class. We also demonstrated the `uses` and `provides...`with directives in module declarations to indicate that a module uses a service or provides a service implementation, respectively. In the next chapter, we discuss various additional Java 9 topics.

Additional Java 9 Topics



Objectives

In this chapter you'll:

- Briefly recap the Java 9 features we've already covered.
- Understand Java's new version numbering scheme.
- Use the new regular-expression `Matcher` methods `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` and `results`.
- Use the new `Stream` methods `takewhile` and `dropwhile` and the new `iterate` overload.
- Learn about the Java 9 JavaFX and other GUI and graphics enhancements.
- Use modules in JShell.
- Overview the Java 9 security-related changes and other Java 9 features.
- Become aware of the capabilities no longer available in JDK 9 and Java 9.
- Become aware of packages, classes and methods proposed for removal from future Java versions.



37.1 Introduction	37.9.3 Datagram Transport Layer Security (DTLS)
37.2 Recap: Java 9 Features Covered in Earlier Chapters	37.9.4 OCSP Stapling for TLS
37.3 New Version String Format	37.9.5 TLS Application-Layer Protocol Negotiation Extension
37.4 Regular Expressions: New Matcher Class Methods	37.10 Other Java 9 Topics
37.4.1 Methods <code>appendReplacement</code> and <code>appendTail</code>	37.10.1 Indify String Concatenation
37.4.2 Methods <code>replaceFirst</code> and <code>replaceAll</code>	37.10.2 Platform Logging API and Service
37.4.3 Method <code>results</code>	37.10.3 Process API Updates
37.5 New Stream Interface Methods	37.10.4 Spin-Wait Hints
37.5.1 Stream Methods <code>takeWhile</code> and <code>dropWhile</code>	37.10.5 UTF-8 Property Resource Bundles
37.5.2 Stream Method <code>iterate</code>	37.10.6 Use CLDR Locale Data by Default
37.5.3 Stream Method <code>ofNullable</code>	37.10.7 Elide Deprecation Warnings on Import Statements
37.6 Modules in JShell	37.10.8 Multi-Release JAR Files
37.7 JavaFX 9 Skin APIs	37.10.9 Unicode 8
37.8 Other GUI and Graphics Enhancements	37.10.10 Concurrency Enhancements
37.8.1 Multi-Resolution Images	37.11 Items Removed from the JDK and Java 9
37.8.2 TIFF Image I/O	37.12 Items Proposed for Removal from Future Java Versions
37.8.3 Platform-Specific Desktop Features	37.12.1 Enhanced Deprecation
37.9 Security Related Java 9 Topics	37.12.2 Items Likely to Be Removed in Future Java Versions
37.9.1 Filter Incoming Serialization Data	37.12.3 Finding Deprecated Features
37.9.2 Create PKCS12 Keystores by Default	37.12.4 Java Applets
	37.13 Wrap-Up

37.1 Introduction

Just before we published this book, Java Specification Request (JSR) 379: Java SE 9 was released as a draft at:

<http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>

The JSR details the

- features included in Java 9,
- features that have been removed from Java 9, and
- features that are proposed for removal from future Java versions.

Once this JSR is approved as final it will be posted at:

<https://www.jcp.org/en/jsretail?id=379>

This JSR is a must read for any Java 9 developer. It gives a high-level overview of the breadth and depth of Java 9 and provides links to all the key JEPs and JSRs.

In any new version of a language there are items of immediate benefit to most programmers, items of interest to some programmers and narrow-purpose, specialty topics that limited numbers of developers will use. We divided this chapter into several groups:

- A recap of the Java 9 features we covered in earlier chapters.
- Live-code examples and discussions of additional functionality that will be useful to a wider audience.
- A brief overview of specialty features with references to where you can learn more.
- A list of features removed from JDK 9 and Java 9.
- A list of features proposed for removal from future Java versions.

Developers should, of course, avoid features in the last two groups in new development, and replace uses of those features in old code as it's migrated to Java 9.

37.2 Recap: Java 9 Features Covered in Earlier Chapters

Here we list the Java 9 features already covered in the book and where you can find each:

- Underscore (_) is no longer a valid identifier (Section 2.2). This is one of several features of JEP 213: Milling Project Coin (<http://openjdk.java.net/jeps/213>).
- Mentioned enhancements to `SecureRandom` (Section 5.9) per JEP 273 (<http://openjdk.java.net/jeps/273>).
- As of Java 9, the compiler now issues a warning if you attempt to access a `static` class member through an instance of the class (Section 8.11).
- Introduced `private` interface methods (Section 10.11), another feature of JEP 213: Milling Project Coin.
- Mentioned the new Stack-Walking API (Section 11.7) from JEP 259 (<http://openjdk.java.net/jeps/259>).
- Mentioned that effectively `final AutoCloseable` variables can now be used in `try-with-resources` statements (Section 11.12), another feature of JEP 213: Milling Project Coin.
- Overviewed new JavaFX 9 features and other GUI and graphics enhancements (Section 13.8).
- Mentioned Java 9's more compact `String` representation (Section 14.3), per JEP 254 (<http://openjdk.java.net/jeps/254>).
- Presented the new convenience factory methods for creating read-only collections (Section 16.14), per JEP 269 (<http://openjdk.java.net/jeps/269>).
- Chapter 25, Introduction to JShell: Java 9's REPL for Interactive Java, presented detailed, example-driven coverage of the JDK's new `jshell` tool.
- Chapter 36, Java Platform Module System, presented detailed example-driven coverage of Java 9's new module system.

37.3 New Version String Format

Prior to Java 9, JDK versions were numbered `1.X.0_updateNumber` where X was the major Java version. For example,

- Java 8's current JDK version number is `jdk1.8.0_121` and
- Java 7's final JDK version number was `jdk1.7.0_80`.

This numbering scheme has changed. JDK 9 initially will be known as jdk-9. Future minor version updates will add new features, and security updates will fix security holes. These updates will be reflected in the JDK version numbers. For example, in 9.1.3:

- 9—is the major Java version number
- 1—is the minor version update number and
- 3—is the security update number.

So 9.2.5 would indicate the version of Java 9 for which there have been two minor version updates and five total security updates (across major and minor versions). For additional details, see JEP 223:

<http://openjdk.java.net/jeps/223>

37.4 Regular Expressions: New Matcher Class Methods

Java SE 9 adds several new `Matcher` method overloads—`appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` and `results` (Fig. 37.1).

```
1 // Fig. 37.1: MatcherMethods.java
2 // Java 9's new Matcher methods.
3 import java.util.regex.Matcher;
4 import java.util.regex.Pattern;
5
6 public class MatcherMethods {
7     public static void main(String[] args) {
8         String sentence = "a man a plan a canal panama";
9
10        System.out.printf("sentence: %s%n", sentence);
11
12        // using Matcher methods appendReplacement and appendTail
13        Pattern pattern = Pattern.compile("an"); // regex to match
14
15        // match regular expression to String and replace
16        // each match with uppercase letters
17        Matcher matcher = pattern.matcher(sentence);
18
19        // used to rebuild String
20        StringBuilder builder = new StringBuilder();
21
22        // append text to builder; convert each match to uppercase
23        while (matcher.find()) {
24            matcher.appendReplacement(
25                builder, matcher.group().toUpperCase());
26        }
27
28        // append the remainder of the original String to builder
29        matcher.appendTail(builder);
30        System.out.printf(
31            "%nAfter appendReplacement/appendTail: %s%n", builder);
32    }
```

Fig. 37.1 | Java 9's new `Matcher` methods. (Part I of 2.)

```

33    // using Matcher method replaceFirst
34    matcher.reset(); // reset matcher to its initial state
35    System.out.printf("%nBefore replaceFirst: %s%n", sentence);
36    String result = matcher.replaceFirst(m -> m.group().toUpperCase());
37    System.out.printf("After replaceFirst: %s%n", result);
38
39    // using Matcher method replaceAll
40    matcher.reset(); // reset matcher to its initial state
41    System.out.printf("%nBefore replaceAll: %s%n", sentence);
42    result = matcher.replaceAll(m -> m.group().toUpperCase());
43    System.out.printf("After replaceAll: %s%n", result);
44
45    // using method results to get a Stream<MatchResult>
46    System.out.printf("%nUsing Matcher method results:%n");
47    pattern = Pattern.compile("\\w+"); // regular expression to match
48    matcher = pattern.matcher(sentence);
49    System.out.printf("The number of words is: %d%n",
50                      matcher.results().count());
51
52    matcher.reset(); // reset matcher to its initial state
53    System.out.printf("Average characters per word is: %f%n",
54                      matcher.results()
55                          .mapToInt(m -> m.group().length())
56                          .average().orElse(0));
57}
58}

```

sentence: a man a plan a canal panama

After appendReplacement/appendTail: a mAN a pLAN a cANaL pANama

Before replaceFirst: a man a plan a canal panama
After replaceFirst: a mAN a plan a canal panama

Before replaceAll: a man a plan a canal panama
After replaceAll: a mAN a pLAN a cANaL pANama

Using Matcher method results:
The number of words is: 7
Average characters per word is: 3.000000

Fig. 37.1 | Java 9's new Matcher methods. (Part 2 of 2.)

37.4.1 Methods `appendReplacement` and `appendTail`

The new Matcher method overloads `appendReplacement` (lines 24–25) and `appendTail` (line 29) are used with Matcher method `find` (line 23) and a `StringBuilder` in a loop to iterate through a `String` and replace every-regular expression match with a specified `String`. At the end of the process, the `StringBuilder` contains the original `String`'s contents updated with the replacements. Lines 13–26 proceed as follows:

- Line 13 creates a `Pattern` to match—in this case, the literal characters "an".
- Line 17 creates a `Matcher` object for the `String` `sentence` (declared in line 8). This will be used to locate the `Pattern` "an" in `sentence`.

- Line 20 creates the `StringBuilder` in which the results will be placed.
- Line 23 uses `Matcher` method `find`, to locate an occurrence of "an" in the original `String`.
- If a match is found, method `find` returns `true`, and line 24 calls `Matcher` method `appendReplacement` to replace "an" with "AN". The method's second argument calls `Matcher` method `group` to get a `String` representing the set of characters that matched the regular expression (in this case, "an"). We then convert the matching characters to uppercase. Method `appendReplacement` then appends to the `StringBuilder` in the first argument all of characters up to the match in the original `String`, followed by the replacement specified in the second argument. Then, the loop-continuation condition attempts to `find` another match in the original `String`, starting from the first character *after* the preceding match.
- When method `find` returns `false`, the loop terminates and line 29 uses `Matcher` method `appendTail` to append the remaining characters of the original `String` sentence to the `StringBuilder`.

At the end of this process for the original `String` "a man a plan a canal panama", the `StringBuilder` contains "a mAN a pLAN a cANal pANama".

37.4.2 Methods `replaceFirst` and `replaceAll`

`Matcher` method overloads `replaceFirst` (line 36) and `replaceAll` (line 42) replace the first match or all matches in a `String`, respectively, using a `Function` that receives a `MatchResult` and returns a replacement `String`. Lines 36 and 42 implement interface `Function` with lambdas that group the matching characters and convert them to uppercase `Strings`. Lines 34 and 40 call `Matcher` method `reset` so that the subsequent calls to `replaceFirst` and `replaceAll` begin searching for matches from the first character in `sentence`.

37.4.3 Method `results`

The new `Matcher` method `results` (lines 50 and 54) returns a stream of `MatchResults`. In lines 47–50, we use the regular expression `\w+` to match sequences of word characters then simply count the matches to determine the number of words in `sentence`. After resetting the `Matcher` (line 52), lines 54–56 use a stream to map each word to its `int` number of characters (via `mapToInt`), then calculate the average length of each word using `IntStream` method `average`.

37.5 New Stream Interface Methods

Java 9 adds several new `Stream` methods—`takeWhile`, `dropWhile`, `iterate` and `ofNullable` (Fig. 37.2). All but `ofNullable` are also available in the numeric streams like `IntStream`.

```
1 // Fig. 37.2: StreamMethods.java
2 // Java 9's new stream methods takeWhile, dropWhile, iterate
3 // and ofNullable.
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`.
(Part 1 of 3.)

```
4 import java.util.stream.Collectors;
5 import java.util.stream.IntStream;
6 import java.util.stream.Stream;
7
8 public class StreamMethods {
9     public static void main(String[] args) {
10         int[] values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12         System.out.printf("Array values contains: %s%n",
13             IntStream.of(values)
14                 .mapToObj(String::valueOf)
15                 .collect(Collectors.joining(" ")));
16
17         // take the largest stream prefix of elements less than 6
18         System.out.println("Demonstrating takeWhile and dropWhile:");
19         System.out.printf("Elements less than 6: %s%n",
20             IntStream.of(values)
21                 .takeWhile(e -> e < 6)
22                 .mapToObj(String::valueOf)
23                 .collect(Collectors.joining(" ")));
24
25         // drop the largest stream prefix of elements less than 6
26         System.out.printf("Elements 6 or greater: %s%n",
27             IntStream.of(values)
28                 .dropWhile(e -> e < 6)
29                 .mapToObj(String::valueOf)
30                 .collect(Collectors.joining(" ")));
31
32         // use iterate to generate stream of powers of 3 less than 10000
33         System.out.printf("%nDemonstrating iterate:%n");
34         System.out.printf("Powers of 3 less than 10,000: %s%n",
35             IntStream.iterate(3, n -> n < 10_000, n -> n * 3)
36                 .mapToObj(String::valueOf)
37                 .collect(Collectors.joining(" ")));
38
39         // demonstrating ofNullable
40         System.out.printf("%nDemonstrating ofNullable:%n");
41         System.out.printf("Number of stream elements: %d%n",
42             Stream.ofNullable(null).count());
43         System.out.printf("Number of stream elements: %d%n",
44             Stream.ofNullable("red").count());
45     }
46 }
```

```
Array values contains: 1 2 3 4 5 6 7 8 9 10
Demonstrating takeWhile and dropWhile:
Elements less than 6: 1 2 3 4 5
Elements 6 or greater: 6 7 8 9 10

Demonstrating iterate:
Powers of 3 less than 10,000: 3 9 27 81 243 729 2187 6561
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`.
(Part 2 of 3.)

```
Demonstrating ofNullable:  
Number of stream elements: 0  
Number of stream elements: 1
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`.
(Part 3 of 3.)

37.5.1 Stream Methods `takeWhile` and `dropWhile`

Lines 19–30 demonstrate methods `takeWhile` and `dropWhile`, which based on a `Predicate` include or omit stream elements, respectively. These methods are meant for use on ordered streams. Unlike `filter`, which processes all of the stream's elements, each of these new methods process elements only until its `Predicate` argument becomes `false`.

The stream pipeline in lines 19–23 takes `ints` from the beginning of the stream while each `int` is less than 6. The predicate returns `true` only for the first five stream elements—as soon as the `Predicate` returns `false`, the remaining elements of the original stream are ignored. For the five elements that remain in the stream, we map each to a `String` and returns a `String` containing the elements separated by spaces.

The stream pipeline in lines 26–30 drops `ints` from the beginning of the stream while each `int` is less than 6. The resulting stream contains the elements beginning with the first one that was 6 or greater. For the elements that remain in the stream, we map each element to a `String` and collect the results into a `String` containing the elements separated by spaces.



Error-Prevention Tip 37.1

Invoke `takeWhile` and `dropWhile` only on ordered streams. If these methods are called on an unordered stream, the stream may return any subset of the matching elements, including none at all, thus giving you potentially unexpected results.



Performance Tip 37.1

According to the `Stream` interface documentation, you may encounter performance issues for the `takeWhile` and `dropWhile` methods on ordered parallel pipelines. For more information, see <http://download.java.net/java/jdk9/docs/api/java/util/stream/Stream.html>.

37.5.2 Stream Method `iterate`

In Section 4.8, we showed a `while` loop that calculated the powers of 3 less than 100. Lines 34–37 show how to use the new overload of `Stream` method `iterate` to generate a stream of `ints` containing the powers of 3 less than 10,000. The new overload takes as its arguments

- a seed value which becomes the stream's first element,
- a `Predicate` that determines when to stop producing elements, and
- a `UnaryOperator` that's invoked initially on the seed value, then on each prior value that `iterate` produces until the `Predicate` becomes `false`.

In this case, the seed value is 3, the `Predicate` indicates that `iterate` should continue producing elements while the last element produced is less than 10,000, and the `UnaryOperator` multiplies the prior element's value by 3 to produce the next element. Then we map

each element to a `String` and collect the results into a `String` containing the elements separated by spaces.

37.5.3 Stream Method ofNullable

The new `Stream` static method `ofNullable` receives a reference to an object and, if the reference is not `null`, returns a one-element stream containing the object; otherwise, it returns an empty stream. Lines 42 and 44 show mechanical examples demonstrating an empty stream and a one-element stream, respectively.

Method `ofNullable` typically would be used to ensure that a reference is not `null`, before performing operations in a stream pipeline. Consider a company employee database. A program could query the database to locate all the `Employees` in a given department and store them as a collection in a `Department` object referenced by the variable `department`. If the query were performed for a nonexistent department, the reference would be `null`. Rather than first checking whether `department` is `null`, then performing a task as in

```
if (department != null) {
    // do something
}
```

you can instead use code like the following:

```
Stream.ofNullable(department)
    .flatMap(Department::streamEmployees)
    ... // do something with each Employee
```

Here we assume that class `Department` contains a public method `streamEmployees` that returns a stream of `Employees`. If `department` is not `null`, the pipeline would `flatMap` the `Department` object into a stream of `Employees` for further processing. If `department` were `null`, `ofNullable` would return an empty stream, so the pipeline would simply terminate.

37.6 Modules in JShell

In Section 25.10, we demonstrated how to add your custom classes to the JShell classpath, so that you can then interact with them in JShell. Here we show how to do that with the `com.deitel.timelibrary` module from Section 36.4. For the purpose of this section, open a command window and change to the `TimeApp` folder in the `ch36` examples folder, then start `jshell`.

Adding a Module to the JShell Session

The `/env` command can specify the module path and the specific modules that JShell should load from that path. To add the `com.deitel.timelibrary` module, execute the following command:

```
jshell> /env -module-path jars -add-modules com.deitel.timelibrary
| Setting new options and restoring state.

jshell>
```

The `-module-path` option indicates where the modules you wish to load are located (in this case the `jars` folder in the folder from which you executed JShell). The `-add-modules` option indicates the specific modules to load (in this case, `com.deitel.timelibrary`).

Importing a Class from a Module's Exported Package(s)

Once the module is loaded, you may import types from any of the module's exported packages. The following command imports the module's `Time1` class:

```
jshell> import com.deitel.timelibrary.Time1
jshell>
```

Using the Imported Class

At this point, you can use class `Time1`, just as you used other classes in Chapter 25. Create a `Time1` object,

```
jshell> Time1 time = new Time1()
time ==> 12:00:00 AM

jshell>
```

Next, inspect its members with auto-completion by typing "time." and pressing *Tab*:

```
jshell> time.
equals()           getClass()          hashCode()
notify()          notifyAll()         toUniversalString()
setTime()         toString()          wait()

jshell> time.
```

View just the members that begin with "to" by typing "to" then pressing *Tab*:

```
jshell> time.to
toString()        toUniversalString()

jshell> time.to
```

Finally, type "U" then press *Tab* to auto-complete `toUniversalString()`, then press *Enter* to invoke the method and assign the 24-hour-clock-format `String` to an implicitly declared variable:

```
jshell> time.toUniversalString()
$3 ==> "00:00:00"

jshell>
```

37.7 JavaFX 9 Skin APIs

In Chapter 22, JavaFX Graphics, Animation and Video, we demonstrated how to format JavaFX objects using *Cascading Style Sheets (CSS)* technology which was originally developed for styling the elements in web pages. CSS allows you to specify *presentation* (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's *structure* and *content* (layout containers, shapes, text, GUI components, etc.). If a JavaFX GUI's presentation is determined entirely by a style sheet (which specifies the rules for styling the GUI), you can simply swap in a new style sheet—sometimes called a *theme* or a *skin*—to change the GUI's appearance.

Each JavaFX control also has a *skin class* that determines its default appearance and how the user can interact with the control. In JavaFX 8, these skin classes were defined as *internal APIs*, but many developers extended these classes to create custom skins.



Portability Tip 37.1

Due to strong encapsulation, the JavaFX 8 internal skin APIs are no longer accessible in Java 9. If you created custom skins based on these pre-Java-9 APIs, your code will no longer compile in Java 9, and any existing compiled code will not run in the Java 9 JRE.

As part of Java 9 modularization, JavaFX 9 makes the skin classes `public` APIs in the `javafx.scene.control.skin` package, as described by JEP 253:

<http://openjdk.java.net/jeps/253>

The new skin classes are direct or indirect subclasses of class `SkinBase` (package `javafx.scene.control`). You can extend the appropriate skin class to customize the look-and-feel for a given type of control. You can then specify the fully qualified name of your skin class for a given control via the JavaFX CSS property `-fx-skin`.

Generally CSS is the easiest way to control the look of your JavaFX GUIs. For precise control over every aspect of a control, including the control's size, position, mouse and keyboard interactions and more, extend `SkinBase` or one of its many new control-specific subclasses in package `javafx.scene.control.skin`.

37.8 Other GUI and Graphics Enhancements

In addition to the changes mentioned in Section 13.8 and 37.7, JSR 379 includes enhanced image support and additional desktop integration features.

37.8.1 Multi-Resolution Images

Apps often display different versions of an image, based on a device's screen size and resolution. Java 9 adds support for multi-resolution images in which a single image actually represents a set of images and class `Graphics` (package `java.awt`) can choose the appropriate resolution to use, based on the device. For more information, visit:

<http://openjdk.java.net/jeps/251>

37.8.2 TIFF Image I/O

The Image I/O framework provides APIs for loading and saving images. The framework supports plug-ins for different image formats, with PNG and JPEG required to be supported on all Java implementations. As of Java 9, all implementations are also required to support the TIFF (also called TIF) format—macOS uses TIFF as one of its standard image formats and various other platforms also support it. For more information on the Image I/O framework, visit:

<https://docs.oracle.com/javase/8/docs/technotes/guides/imageio/>

For more information on the new TIFF support, visit:

<http://openjdk.java.net/jeps/262>

37.8.3 Platform-Specific Desktop Features

In Java 9, various internal APIs that were used for operating-system-specific desktop integration—such as interacting with the dock in macOS—are no longer accessible due to the module system’s strong encapsulation. JEP 272 adds new public APIs to expose this capability for macOS and to provide similar capabilities for other operating systems (such as Windows and Linux). Other features that will be provided include

- login/logout and screen lock/unlock event listeners so a Java app can respond to those events
- getting the user’s attention via the dock or task bar with blinking or bouncing app icons and
- displaying progress bars in a dock or task bar.

For more information, visit:

<http://openjdk.java.net/jeps/272>

37.9 Security Related Java 9 Topics

It’s important for developers to be aware of Java security enhancements. In this section, we provide brief mentions of a few Java 9 security-related features and where you can learn more about each.

37.9.1 Filter Incoming Serialization Data

Java’s **object serialization** mechanism enables programs to create **serialized objects**—sequences of bytes that include each object’s data, as well as information about the object’s type and the types of the object’s data. After a serialized object has been output, it can be read into a program and **deserialized**—that is, the type information and bytes that represent the object are used to recreate the object in memory.

Deserialization has the potential for security problems. For example, if the bytes being deserialized are read from a network connection, an attacker could intercept the bytes and inject invalid data. If you do not validate the data after deserialization, it’s possible that the object would be in an invalid state that could affect the program’s execution. In addition, the deserialization mechanism enables any serialized object to be deserialized, provided that its type definition is available to the runtime. If the object being serialized contains an array, an attacker potentially could inject an arbitrarily large number of elements, potentially using all of the app’s available memory.

JEP 290, Filter Incoming Serialization Data:

<http://openjdk.java.net/jeps/290>

is a security enhancement to object serialization that enables programs to add filters that can restrict which types can be serialized, validate array lengths and more.

37.9.2 Create PKCS12 Keystores by Default

A keystore maintains security certificates that are used in encryption. Java has used a custom keystore since Java 1.2 (1998). By default, Java 9 now uses the popular and extensible

PKCS12 keystore, which is more secure and will enable Java systems to interoperate with other systems that support the same standard. For more information, visit:

<http://openjdk.java.net/jeps/229>

37.9.3 Datagram Transport Layer Security (DTLS)

Datagrams provide a connectionless mechanism to communicate information over a network. Java 9 adds support for the Datagram Transport Layer Security (DTLS) protocol which provides secure communication via datagrams. For more information, visit:

<http://openjdk.java.net/jeps/219>

37.9.4 OCSP Stapling for TLS

X.509 security certificates are used in public-key cryptography. JEP 249 is a security and performance enhancement for checking whether an X.509 security certificate is still valid. For details, visit:

<http://openjdk.java.net/jeps/249>

37.9.5 TLS Application-Layer Protocol Negotiation Extension

This is a security enhancement to the `javax.net.ssl` package to enable applications to choose from a list of protocols for communicating with one another over a secure connection. For more details, visit

<http://openjdk.java.net/jeps/244>

37.10 Other Java 9 Topics

In this section, we provide brief mentions of various other features of JSR 379. At the time of this writing during Java 9's early access stage, only limited documentation was available to us. So we concentrated on the information from the JSRs and JEPs. In a few cases, we did not comment on certain new Java 9 features. These include:

- JEP 193: Variable Handles (<http://openjdk.java.net/jeps/193>),
- JEP 268: XML Catalogs (<http://openjdk.java.net/jeps/268>) and
- JEP 274: Enhanced Method Handles (<http://openjdk.java.net/jeps/274>).

37.10.1 Indify String Concatenation

JEP 280, Indify String Concatenation, is a behind-the-scenes enhancement to `javac` that's geared to improving `String` concatenation performance in the future. The goal is to enable such performance enhancements to be developed and added to future Java implementations *without* having to modify the bytecodes `javac` produces. For more information, visit:

<http://openjdk.java.net/jeps/280>

37.10.2 Platform Logging API and Service

Developers commonly use logging frameworks for tracking information that helps them with debugging, maintenance and evolution of their systems, analytics, detecting security

breaches and more. JEP 264, Platform Logging API and Service, adds a logging API for use by platform classes in the `java.base` module. Developers can then implement a service provider that routes logging messages to their preferred logging framework. For more information, visit:

<http://openjdk.java.net/jeps/264>

37.10.3 Process API Updates

Java 9 includes enhancements to the APIs that enable Java programs to interact with operating-system-specific processes without having to use platform-specific native code written in C or C++. Some enhancements include access to a process's ID, arguments, start time, total CPU time and name, and terminating and monitoring processes from Java apps. For more information, visit:

<http://openjdk.java.net/jeps/102>

37.10.4 Spin-Wait Hints

Section 23.7 introduced a multithreading technique in which a thread that's waiting to acquire a lock on an object uses a loop to determine whether the lock is available and, if not, waits. Each time the thread is notified to check again, the loop repeats this process until the lock is acquired. This technique is known as a spin-wait loop. Java 9 adds a new API that enables such a loop to notify the JVM that it is a spin-wait loop. On some hardware platforms, the JVM can use this information to improve performance and reduce power consumption (especially crucial for battery-powered mobile devices). For more information, visit:

<http://openjdk.java.net/jeps/285>

37.10.5 UTF-8 Property Resource Bundles

`Class ResourceBundle` (package `java.util`) enables programs to load locale-specific information, such as `Strings` in different spoken languages. This technique is commonly used to localize apps for users in different regions. Java 9 upgrades class `ResourceBundle` to support resources that are encoded in UTF-8 format (<https://en.wikipedia.org/wiki/UTF-8>). For more information, visit:

<http://openjdk.java.net/jeps/226>

37.10.6 Use CLDR Locale Data by Default

CLDR—the Unicode Common Locale Data Repository (<http://cldr.unicode.org>)—is an extensive repository of locale-specific information that developers can use when internationalizing their apps. Data in the repository includes information on

- date, time, number and currency formatting
- translations for the names of spoken languages, countries, regions, months, days, etc.
- language-specific information like capitalization, gender rules, sorting rules, etc.
- country information, and more.

CLDR support was included with Java 8, but is now the default in Java 9. For more information, visit:

<http://openjdk.java.net/jeps/252>

37.10.7 Elide Deprecation Warnings on Import Statements

Many company coding guidelines require code to compile without warnings. In JDK 8, if you imported a deprecated type or statically imported a deprecated member of a type, the compiler would issue warnings, even if those types or members were never used in your code. Java allows you to prevent deprecation warnings in your code via the `@SuppressWarnings` annotation, but this cannot be applied to `import` declarations. For this reason, it was not possible to prevent certain compile-time warnings. JDK 9 no longer produces such warnings on `import` declarations. For more information, visit:

<http://openjdk.java.net/jeps/211>

37.10.8 Multi-Release JAR Files

Even with Java 9's release, many people and organizations will continue using older versions of Java—some for many years. In one session at the 2016 JavaOne conference, attendees were asked which Java versions they were using. Several developers indicated their companies were still using versions as old as Java 1.4, which was released more than 15 years ago.

Library vendors often support multiple Java versions. Prior to Java 9, this required providing separate JAR files specific to each Java version. JDK 9 provides support for multi-release JAR files—a single JAR may contain multiple versions of the same class that are geared to different Java versions. In addition, these multi-release JAR files may contain module descriptors for use with the Java Platform Module System (Chapter 36). For more information, visit:

<http://openjdk.java.net/jeps/238>

37.10.9 Unicode 8

Java 9 supports the latest version of the Unicode Standard (unicode.org)—Unicode 8. Appropriate changes have been made to classes `String` and `Character`, as well as several other classes dependent on Unicode. For more details, visit:

<http://openjdk.java.net/jeps/267>

37.10.10 Concurrency Enhancements

JEP 266, More Concurrency Updates, adds features in three categories:

- Support for reactive streams—a technique for asynchronous stream processing—via class `Flow` and its nested interfaces. For a reactive streams overview and links to various other resources, visit:

https://en.wikipedia.org/wiki/Reactive_Streams

- Various improvements that the Java team accumulated since Java 8.
- Additional methods in class `CompletableFuture` (listed below).

New Methods of Class *CompletableFuture*

Section 23.14 introduced class `CompletableFuture`, which enables you to *asynchronously* execute `Runnables` that perform tasks or `Suppliers` that return values. Java 9 enhances `CompletableFuture` with the following methods:

- `newIncompleteFuture`
- `defaultExecutor`
- `copy`
- `minimalCompletionStage`
- `completeAsync`
- `orTimeout`
- `completeOnTimeout`
- `delayedExecutor`
- `completedStage`
- `failedFuture`
- `failedStage`

For more information on the concurrency enhancements, visit:

<http://openjdk.java.net/jeps/266>

and see the online Java 9 documentation for `java.util.concurrent` (which includes class `CompletableFuture`'s methods) and related packages in the `java.base` module:

<http://download.java.net/java/jdk9/docs/api/overview-summary.html>

37.11 Items Removed from the JDK and Java 9

To help prepare the Java Platform for modularization, Java 9 removed several items from both the platform and its APIs. These are listed in JSR 379, Sections 8 and 9:

<http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>

Removed Platform Features

JSR 379 Section 8 lists the platform changes. These include removal of the Java extensions mechanism. Prior to Java 9, the extensions mechanism allowed you to place a library's JAR file in a special JRE folder to make the library available to all Java apps on that computer. Classes in that folder were guaranteed to load before app-specific classes, so this was sometimes used to upgrade libraries with newer versions. In Java 9, the extensions mechanism is replaced with *upgradeable modules*:

<http://openjdk.java.net/projects/jigsaw/goals-reqs/03#upgradeable-modules>

Upgradable modules are used primarily for standard technologies that evolve independently of the Java SE platform, but are bundled with the platform, such as JAXB—the Java Architecture for XML Binding. When a new JAXB version is released, its module can

be placed in the `java` command's **--upgrade-module-path**. The runtime will then use the new version, rather than the earlier version that was bundled with the platform.

Removed Methods

JSR 379 Section 9 lists methods that have been removed from various Java classes to help modularize the platform. According to the JSR, these methods were infrequently used, but keeping them would have required placing the packages of the `java.desktop` module into the `java.base` module, resulting in a much larger minimal runtime size. This would not make sense, because many apps do not require the `java.desktop` module's GUI and desktop integration capabilities.

37.12 Items Proposed for Removal from Future Java Versions

The Java Platform has been in use for more than 20 years. Over that time, some APIs have been deprecated in favor of newer ones—often to fix bugs, to improve security or simply because an improved API was added that rendered the prior ones obsolete. Yet, many deprecated APIs—some from as far back as Java 1.2, which was released in December 1998—have remained available in every new version of Java, mostly for backward compatibility.

37.12.1 Enhanced Deprecation

JEP 277

<http://openjdk.java.net/jeps/277>

adds new features to the `@Deprecated` annotation that enable developers to provide more information about deprecated APIs, including whether or not the API is scheduled to be removed in a future release. These enhanced annotations are now used throughout the Java 9 APIs and pointed out in the online API documentation to highlight features you should no longer use and that you should expect to be removed from future versions. For example, everything in the `java.applet` package is now deprecated (Section 37.12.4), so when you view the package's documentation at

<http://download.java.net/java/jdk9/docs/api/java/applet/package-summary.html>

you'll see deprecation notes in the package's description and for each type in the package. In addition, if you use the types in your code, you'll get warnings at compile time.

37.12.2 Items Likely to Be Removed in Future Java Versions

JSR 379, Section 10 lists the various packages, classes, fields and methods that are likely to be removed from future Java versions. The JSR indicates that these packages evolve separately from the Java SE Platform or are part of the Java EE Platform specification. According to the JSR, the classes, fields and methods proposed for removal typically do not work, are not useful or have been rendered obsolete by newer APIs.

37.12.3 Finding Deprecated Features

Each page in the online Java API documentation

<http://download.java.net/java/jdk9/docs/api/overview-summary.html>

now includes a **DEPRECATED** link so you can view the **Deprecated API** list containing the deprecated APIs:

<http://download.java.net/java/jdk9/docs/api/deprecated-list.html>

When you click a given item, its documentation generally mentions why it was deprecated and what you should use instead.



Error-Prevention Tip 37.2

Avoid using deprecated features in new code. Also, if you maintain or evolve legacy Java code, you should carefully study the Deprecated API list and consider replacing the listed items with the alternatives specified in the online Java documentation. This will help ensure that your code continues to compile and execute correctly in future Java versions.

37.12.4 Java Applets

As of Java 9 the Java Applet API is deprecated, per JEP 289 (<http://openjdk.java.net/jeps/289>). Previously this enabled Java to run in web browsers via a plug-in. Though this API has not been proposed for removal yet, it could be in a future Java version. Most popular web browsers removed Java plug-in support due to security issues.

37.13 Wrap-Up

In this chapter, we briefly recapped the Java 9 features covered in earlier chapters, then discussed various additional Java 9 topics. We presented the fundamentals of Java's new version numbering scheme. We demonstrated the new regular-expression `Matcher` methods `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` and `results`. We also demonstrated the new `Stream` methods `takeWhile` and `dropWhile` and the new `iterate` overload. We discussed the Java 9 JavaFX changes, including the new `public` skin APIs and other GUI and graphics enhancements. You saw how to use modules in JShell.

We overviewed the Java 9 security-related changes and various other Java 9 features. We discussed the capabilities that are no longer available in JDK 9 and Java 9. Finally, we discussed the packages, classes and methods proposed for removal from future Java versions.