**Section 25.1 Introduction**

• A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced “GOO-ee”) gives an app a distinctive “look-and-feel.”

• GUIs are built from GUI components—sometimes called controls or widgets.

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

• Java’s GUI, graphics and multimedia API of the future is JavaFX.

#### Section 25.2 JavaFX Scene Builder and the NetBeans IDE

• JavaFX Scene Builder is a standalone JavaFX GUI visual layout tool that can also be used with various IDEs.

• JavaFX Scene Builder enables you to create GUIs by dragging and dropping GUI components from Scene Builder’s library onto a design area, then modifying and styling the GUI—all without writing any code.

• JavaFX Scene Builder generates FXML (FX Markup Language)—an XML vocabulary for defining and arranging JavaFX GUI controls without writing any Java code.

• The FXML code is separate from the program logic that’s defined in Java source code—this separation of the interface (the GUI) from the implementation (the Java code) makes it easier to debug, modify and maintain JavaFX GUI apps.

#### Section 25.3 JavaFX App Window Structure

• The window in which a JavaFX app’s GUI is displayed is known as the stage and is an instance of class Stage (package javafx.stage).

• The stage contains one scene that defines the GUI as a scene graph—a tree structure of an app’s visual elements, such as GUI controls, shapes, images, video, text and more. The scene is an instance of class Scene (package javafx.scene).

• Each visual element in the scene graph is a node—an instance of a subclass of Node (packagejavafx.scene), which defines common attributes and behaviors for all nodes in the scene graph.

• The first node in the scene graph is known as the root node.

• Nodes that have children are typically layout containers that arrange their child nodes in the scene.

• The nodes arranged in a layout container are a combination of controls and possibly other layout containers.

• When the user interacts with a control, it generates an event. Programs can use event handling to specify what should happen when each user interaction occurs.

• An event handler is a method that responds to a user interaction. An FXML GUI’s event handlers are defined in a controller class.

#### Section 25.4 Welcome App—Displaying Text and an Image

• To create an app in NetBeans, you must first create a project—a group of related files, such as code files and images that make up an app.

• The NetBeans **Projects** window provides access to all of your projects. Within a project’s node, the contents are organized into folders and files.

• NetBeans creates three files for a JavaFX FXML Application project: an FXML markup file for the GUI, a file containing the app’s main class and a file containing the app’s controller class.

• To open a project’s FXML file in JavaFX Scene Builder, right click the FXML file in the **Projects**window, then select **Open**.

• Layout containers help you arrange GUI components. A VBox arranges its nodes vertically from top to bottom.

• To make a layout container the root node in a scene graph, select the layout container, then select Scene Builder’s **Edit > Trim Document to Selection** menu item.

• A VBox’s alignment determines the layout positioning of its children.

• The preferred size (width and height) of the scene graph’s root node is used by the scene to determine its window size when the app begins executing.

• You can set a Label’s text either by double clicking it and typing the text, or by selecting theLabel and setting its Text property in the **Inspector**’s **Properties** section.

• When adding controls to a VBox, each new control is placed below the preceding ones by default. You can change the order by dragging the children in **Scene Builder**’s **Hierarchy** window.

• To set the image to display, select the ImageView then set its **Image** property in the **Inspector**’s**Properties** section. An image’s aspect ratio is the ratio of the image’s width to its height.

• To specify an ImageView’s size, set its **Fit Width** and **Fit Height** properties in the **Inspector**’s **Layout**section.

#### Section 25.5.2 Technologies Overview

• A JavaFX app’s main class inherits from Application (packagejavafx.application.Application).

• The main class’s main method calls class Application’s static launch method to begin executing a JavaFX app. This method, in turn, causes the JavaFX runtime to create an object of the Application subclass and call its start method, which creates the GUI, attaches it to a Sceneand places it on the Stage that method start recevies as an argument.

• A GridPane (package javafx.scene.layout) arranges JavaFX nodes into columns and rows in a rectangular grid.

• Each cell in a GridPane can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls.

• Each component in a GridPane can span multiple columns or rows.

• A TextField (package javafx.scene.control) can accept text input or display text.

• A Slider (package javafx.scene.control) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the Slider’s thumb.

• A Button (package javafx.scene.control) allows the user to initiate an action.

• Class NumberFormat (package java.text) can format locale-specific currency and percentage strings.

• GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task.

• The code that performs a task in response to an event is called an event handler.

• For certain events you can link a control to its event-handling method by using the **Code** section of Scene Builder’s **Inspector** window. In this case, the class that implements the event-listener interface will be created for you and will call the method you specify.

• For events that occur when the value of a control’s property changes, you must create the event handler entirely in code.

• You implement the ChangeListener interface (package javafx.beans.value) to respond to the user moving the Slider’s thumb.

• JavaFX applications in which the GUI is implemented as FXML adhere to the Model-View-Controller (MVC) design pattern, which separates an app’s data (contained in the model) from the app’s GUI (the view) and the app’s processing logic (the controller). The controller implements logic for processing user inputs. The view presents the data stored in the model. When a user provides input, the controller modifies the model with the given input. When the model changes, the controller updates the view to present the changed data. In a simple app, the model and controller are often combined into a single class.

• In a JavaFX FXML app, you define the app’s event handlers in a controller class. The controller class defines instance variables for interacting with controls programmatically, as well as eventhandling methods.

• Class FXMLLoader’s static method load uses the FXML file that represents the app’s GUI to creates the GUI’s scene graph and returns a Parent (package javafx.scene) reference to the scene graph’s root node. It also initializes the controller’s instance variables, and creates and registers the event handlers for any events specified in the FXML.

#### Section 25.5.3 Building the App’s GUI

• If a control or layout will be manipulated programmatically in the controller class, you must provide a name for that control or layout. Each object’s name is specified via its **fx:id** property. You can set this property’s value by selecting a component in your scene, then expanding the **Inspector**window’s **Code** section—the **fx:id** property appears at the top.

• By default, the GridPane contains two columns and three rows. You can add a row above or below an existing row by right clicking a row and selecting **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**. You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.

• You can set a Button’s text by double clicking it, or by selecting the Button, then setting its **Text**property in the **Inspector** window’s **Properties** section.

• A GridPane column’s contents are left-aligned by default. To change the alignment, select the column by clicking the tab at the top or bottom of the column, then in the **Inspector**’s **Layout**section, set the **Halignment** property.

• Setting a node’s **Pref Width** property of a GridPane column to USE\_COMPUTED\_SIZE indicates that the width should be based on the widest child.

• To size a Button the same width as the other controls in a GridPane’s column, select the Button, then in the **Inspector**’s **Layout** section, set the **Max Width** property to MAX\_VALUE.

• As you design your GUI, you can preview it in Scene Builder by selecting **Preview > Show Preview in Window**.

• The space between a node’s contents and its top, right, bottom and left edges is known as the padding, which separates the contents from the node’s edges. To set the padding, select the node, then in the **Inspector**’s **Layout** section, set the **Padding** property’s values.

• You can specify the default amount of space between a GridPane’s columns and rows with its**Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively.

• You can type in a TextField only if it’s “in focus”—that is, it’s the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the Tab key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI.

#### Section 25.5.4 TipCalculator Class

• To display a GUI, you must attach it to a Scene, then attach the Scene to the Stage that’s passed into Application method start.

• By default, the Scene’s size is determined by the size of the scene graph’s root node. Overloaded versions of the Scene constructor allow you to specify the Scene’s size and fill (a color, gradient or image), which appears in the c’s background.

• Scene method setTitle specifies the text that appears in the Stage window’s title bar.

• Stage method setScene places a Scene onto a Stage.

• Stage method show displays the Stage window.

#### Section 25.5.5 TipCalculatorController Class

• The RoundingMode enum of package java.math is used to specify how BigDecimal values are rounded during calculations or when formatting floating-point numbers as Strings.

• Class NumberFormat of package java.text provides numeric formatting capabilities, such as locale-specific currency and percentage formats.

• A Button’s event handler receives an ActionEvent, which indicates that the Button was clicked. Many JavaFX controls support ActionEvents.

• Package javafx.scene.control contains many JavaFX control classes.

• The @FXML annotation preceding an instance variable indicates that the variable’s name can be used in the FXML file that describes the app’s GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI.

• When the FXMLLoader loads an FXML file to create a GUI, it also initializes each of the controller’s instance variables that are declared with @FXML to ensure that they refer to the corresponding GUI components in the FXML file.

• The @FXML annotation preceding a method indicates that the method can be used to specify a control’s event handler in the FXML file that describes the app’s GUI.

• When the FXMLLoader creates an object of a controller class, it determines whether the class contains an initialize method with no parameters and, if so, calls that method to initialize the controller. This method can be used to configure the controller before the GUI is displayed.

• An anonymous inner class is a class that’s declared without a name and typically appears inside a method declaration.

• Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared.

• An anonymous inner class can access its top-level class’s instance variables, static variables and methods, but has limited access to the local variables of the method in which it’s declared—it can access only the final local variables declared in the enclosing method’s body. (As of Java SE 8, an anonymous inner class may also access a class’s effectively final local variables.)

### Self-Review Exercises

[**25.1**](http://proquest.safaribooksonline.com/9780133813036/ch25lev1sec10_html#ch25ans1) Fill in the blanks in each of the following statements:

a) A(n) \_\_\_\_\_\_\_\_\_ can display text and accept text input from the user.

b) Use a(n) \_\_\_\_\_\_\_\_\_ to arrange GUI components into cells in a rectangular grid.

c) JavaFX Scene Builder’s \_\_\_\_\_\_\_\_\_ window shows the structure of the GUI and allows you to select and reorganize controls.

d) You implement interface \_\_\_\_\_\_\_\_\_ to respond to events when the user moves a Slider’s thumb.

e) A(n) \_\_\_\_\_\_\_\_\_ represents the app’s window.

f) The method \_\_\_\_\_\_\_\_\_ is called by the FXMLLoader before the GUI is displayed.

g) The contents of a scene are placed in its \_\_\_\_\_\_\_\_\_.

h) The elements in the scene graph are called \_\_\_\_\_\_\_\_\_.

i) \_\_\_\_\_\_\_\_\_ allows you to build JavaFX GUIs using drag-and-drop techniques.

j) A(n) \_\_\_\_\_\_\_\_\_ file contains the description of a JavaFX GUI.

[**25.2**](http://proquest.safaribooksonline.com/9780133813036/ch25lev1sec10_html#ch25ans2) State whether each of the following is true or false. If false, explain why.

a) You must create JavaFX GUIs by hand coding them in Java.

b) The layout VBox arranges components vertically in a scene.

c) To right align controls in a GridPane column, set its **Alignment** property to RIGHT.

d) The FXMLLoader initializes the controller’s @FXML instance variables.

e) You override class Application’s launch method to display a JavaFX app’s stage.

f) The control that the user is interacting with “has the focus.”

g) By default, a Slider allows you to select values from 0 to 255.

h) A node can span multiple columns in a GridPane.

i) Every Application subclass must override the start method.

### Answers to Self-Review Exercises

[**25.1**](http://proquest.safaribooksonline.com/9780133813036/ch25lev1sec9_html#ch25que1)

a) TextField.

b) GridPane.

c) **Hierarchy**.

d) ChangeListener<Number>.

e) Stage.

f) initialize.

g) scene graph.

h) nodes.

i) JavaFX Scene Builder.

j) FXML.

[**25.2**](http://proquest.safaribooksonline.com/9780133813036/ch25lev1sec9_html#ch25que2)

a) False. You can use JavaFX Scene Builder to create JavaFX GUIs without writing any code.

b) True.

c) False. The name of the property is **Halignment**.

d) True.

e) False. You override class Application’s start method to display a JavaFX app’s stage.

f) True.

g) False. By default a Slider allows you to select values from 0.0 to 100.0.

h) True.

i) True.

### Exercises

**25.3 (*Scrapbooking*** **App)** Find four images of famous landmarks using websites such as Flickr. Create an app similar to the **Welcome** app in which you arrange the images in a collage. Add text that identifies each landmark. You can use images that are part of your project or you can specify the URL of an image that’s online.

**25.4 (Enhanced** ***Tip Calculator*** **App)** Modify the **Tip Calculator** app to allow the user to enter the number of people in the party. Calculate and display the amount owed by each person if the bill were to be split evenly among the party members.

**25.5 (*Mortgage Calculator*** **App)** Create a mortgage calculator app that allows the user to enter a purchase price, down-payment amount and an interest rate. Based on these values, the app should calculate the loan amount (purchase price minus down payment) and display the monthly payment for 10-, 20- and 30-year loans. Allow the user to select a custom loan duration (in years) by using aSlider and display the monthly payment for that custom loan duration.

**25.6 (*College Loan Payoff Calculator*** **App)** A bank offers college loans that can be repaid in 5, 10, 15, 20, 25 or 30 years. Write an app that allows the user to enter the amount of the loan and the annual interest rate. Based on these values, the app should display the loan lengths in years and their corresponding monthly payments.

**25.7 (*Car Payment Calculator*** **App)** Typically, banks offer car loans for periods ranging from two to five years (24 to 60 months). Borrowers repay the loans in monthly installments. The amount of each monthly payment is based on the length of the loan, the amount borrowed and the interest rate. Create an app that allows the customer to enter the price of a car, the down-payment amount and the loan’s annual interest rate. The app should display the loan’s duration in months and the monthly payments for two-, three-, four- and five-year loans. The variety of options allows the user to easily compare repayment plans and choose the most appropriate.

**25.8 (*Miles-Per-Gallon Calculator*** **App)** Drivers often want to know the miles per gallon their cars get so they can estimate gasoline costs. Develop an app that allows the user to input the number of miles driven and the number of gallons used and calculates and displays the corresponding miles per gallon.

### Exercises

**25.3 (*Scrapbooking*** **App)** Find four images of famous landmarks using websites such as Flickr. Create an app similar to the **Welcome** app in which you arrange the images in a collage. Add text that identifies each landmark. You can use images that are part of your project or you can specify the URL of an image that’s online.

**25.4 (Enhanced** ***Tip Calculator*** **App)** Modify the **Tip Calculator** app to allow the user to enter the number of people in the party. Calculate and display the amount owed by each person if the bill were to be split evenly among the party members.

**25.5 (*Mortgage Calculator*** **App)** Create a mortgage calculator app that allows the user to enter a purchase price, down-payment amount and an interest rate. Based on these values, the app should calculate the loan amount (purchase price minus down payment) and display the monthly payment for 10-, 20- and 30-year loans. Allow the user to select a custom loan duration (in years) by using aSlider and display the monthly payment for that custom loan duration.

**25.6 (*College Loan Payoff Calculator*** **App)** A bank offers college loans that can be repaid in 5, 10, 15, 20, 25 or 30 years. Write an app that allows the user to enter the amount of the loan and the annual interest rate. Based on these values, the app should display the loan lengths in years and their corresponding monthly payments.

**25.7 (*Car Payment Calculator*** **App)** Typically, banks offer car loans for periods ranging from two to five years (24 to 60 months). Borrowers repay the loans in monthly installments. The amount of each monthly payment is based on the length of the loan, the amount borrowed and the interest rate. Create an app that allows the customer to enter the price of a car, the down-payment amount and the loan’s annual interest rate. The app should display the loan’s duration in months and the monthly payments for two-, three-, four- and five-year loans. The variety of options allows the user to easily compare repayment plans and choose the most appropriate.

**25.8 (*Miles-Per-Gallon Calculator*** **App)** Drivers often want to know the miles per gallon their cars get so they can estimate gasoline costs. Develop an app that allows the user to input the number of miles driven and the number of gallons used and calculates and displays the corresponding miles per gallon.

# Chapters on the Web

The following chapters are available at Java How to Program, 10/e’s Companion Website ([www.pearsonhighered.com/deitel](http://www.pearsonhighered.com/deitel)) as PDF documents:

• Chapter 26, JavaFX GUI: Part 2

• Chapter 27, JavaFX Graphics and Multimedia

• Chapter 28, Networking

• Chapter 29, Java Persistence API (JPA)

• Chapter 30, JavaServer™ Faces Web Apps: Part 1

• Chapter 31, JavaServer™ Faces Web Apps: Part 2

• Chapter 32, REST-Based Web Services

• Chapter 33, (Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML

• Chapter 34, (Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design

These files can be viewed in Adobe® Reader® ([get.adobe.com/reader](http://get.adobe.com/reader)).

## A. Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom ([Fig. A.1](http://proquest.safaribooksonline.com/9780133813036/app01_html#app01fig01)).

**Fig. A.1** | Operator precedence chart.

## B. ASCII Character Set

**Fig. B.1** | ASCII character set.

The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

Most users of this book are interested in the ASCII character set used to represent English characters on many computers. The ASCII character set is a subset of the Unicode character set used by Java to represent characters from most of the world’s languages. For more information on the Unicode character set, see the web bonus Appendix H.

## C. Keywords and Reserved Words

**Fig. C.1** | Java keywords.

Java also contains the reserved words true and false, which are boolean literals, and null, which is the literal that represents a reference to nothing. Like keywords, these reserved words cannot be used as identifiers.

## D. Primitive Types

**Fig. D.1** | Java primitive types.

You can use underscores to make numeric literal values more readable. For example, 1\_000\_000 is equivalent to 1000000.

For more information on IEEE 754 visit <http://grouper.ieee.org/groups/754/>. For more information on Unicode, see Appendix H.

## E. Using the Debugger

**And so shall I catch the fly.**

—William Shakespeare

**We are built to make mistakes, coded for error.**

—Lewis Thomas

**What we anticipate seldom occurs; what we least expect generally happens.**

—Benjamin Disraeli

**Objectives**

In this appendix you’ll learn:

 To set breakpoints to debug applications.

 To use the run command to run an application through the debugger.

 To use the stop command to set a breakpoint.

 To use the cont command to continue execution.

 To use the print command to evaluate expressions.

 To use the set command to change variable values during program execution.

 To use the step, step up and next commands to control execution.

 To use the watch command to see how a field is modified during program execution.

 To use the clear command to list breakpoints or remove a breakpoint.

**Outline**

[**E.1** Introduction](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec1_html#app05lev1sec1)

[**E.2** Breakpoints and the run, stop, cont and print Commands](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05lev1sec2)

[**E.3** The print and set Commands](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05lev1sec3)

[**E.4** Controlling Execution Using the step, step up and next Commands](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05lev1sec4)

[**E.5** The watch Command](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05lev1sec5)

[**E.6** The clear Command](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05lev1sec6)

[**E.7** Wrap-Up](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec7_html#app05lev1sec7)

### E.1 Introduction

In [Chapter 2](http://proquest.safaribooksonline.com/9780133813036/ch02_html#ch02), you learned that there are two types of errors—syntax errors and logic errors—and you learned how to eliminate syntax errors from your code. Logic errors do not prevent the application from compiling successfully, but they do cause an application to produce erroneous results when it runs. The JDK includes software called a **debugger** that allows you to monitor the execution of your applications so you can locate and remove logic errors. The debugger will be one of your most important application development tools. Many IDEs provide their own debuggers similar to the one included in the JDK or provide a graphical user interface to the JDK’s debugger.

This appendix demonstrates key features of the JDK’s debugger using command-line applications that receive no input from the user. The same debugger features discussed here can be used to debug applications that take user input, but debugging such applications requires a slightly more complex setup. To focus on the debugger features, we’ve opted to demonstrate the debugger with simple command-line applications involving no user input. For more information on the Java debugger visit<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html>.

### E.2 Breakpoints and the run, stop, cont and printCommands

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When application execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether logic errors exist. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Setting a breakpoint at a line of code that is not executable (such as a comment) causes the debugger to display an error message.

To illustrate the features of the debugger, we use application AccountTest ([Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01)), which creates and manipulates an object of class Account ([Fig. 3.8](http://proquest.safaribooksonline.com/9780133813036/ch03lev2sec9_html#ch03fig08)). Execution of AccountTest begins in main (lines 7–24). Line 9 creates an Account object with an initial balance of $50.00. Recall that Account’s constructor accepts one argument, which specifies the Account’s initial balance. Lines 12–13 output the initial account balance using Account method getBalance. Line 15 declares and initializes a local variable depositAmount. Lines 17–19 then print depositAmount and add it to the Account’s balanceusing its credit method. Finally, lines 22–23 display the new balance. [Note: The [Appendix E](http://proquest.safaribooksonline.com/9780133813036/app05_html#app05)examples directory contains a copy of Account.java identical to the one in [Fig. 3.8](http://proquest.safaribooksonline.com/9780133813036/ch03lev2sec9_html#ch03fig08).]

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig01a)

**1**   // Fig. E.1: AccountTest.java  
 **2**   // Create and manipulate an Account object.  
 **3**   
 **4**   public class AccountTest  
 **5**   {  
 **6**      // main method begins execution  
 **7**      public static void main(String[] args)  
 **8**      {  
 **9**         Account account = new Account("Jane Green", 50.00);  
**10**   
**11**         // display initial balance of Account object  
**12**         System.out.printf("initial account balance: $%.2f%n",  
**13**            account.getBalance());  
**14**   
**15**         double depositAmount = 25.0; // deposit amount  
**16**   
**17**         System.out.printf("%nadding %.2f to account balance%n%n",  
**18**            depositAmount);  
**19**         account.deposit(depositAmount); // add to account balance  
**20**   
**21**         // display new balance  
**22**         System.out.printf("new account balance: $%.2f%n",  
**23**            account.getBalance());  
**24**      }  
**25**   } // end class AccountTest

initial account balance: $50.00  
  
adding 25.00 to account balance  
  
new account balance: $75.00

**Fig. E.1** | Create and manipulate an Account object.

In the following steps, you’ll use breakpoints and various debugger commands to examine the value of the variable depositAmount declared in AccountTest ([Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01)).

**1. Opening the** **Command Prompt** **window and changing directories.** Open the **Command Prompt**window by selecting **Start > Programs > Accessories > Command Prompt**. Change to the directory containing the [Appendix E](http://proquest.safaribooksonline.com/9780133813036/app05_html#app05) examples by typing cd C:\examples\debugger [Note: If your examples are in a different directory, use that directory here.]

**2. Compiling the application for debugging**. The Java debugger works only with .class files that were compiled with the **-g** compiler option, which generates information that is used by the debugger to help you debug your applications. Compile the application with the -g command-line option by typing javac -g AccountTest.java Account.java. Recall from [Chapter 3](http://proquest.safaribooksonline.com/9780133813036/ch03_html#ch03) that this command compiles both AccountTest.java and Account.java. The command java -g \*.javacompiles all of the working directory’s .java files for debugging.

**3. Starting the debugger.** In the **Command Prompt**, type **jdb** ([Fig. E.2](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig02)). This command will start the Java debugger and enable you to use its features. [Note: We modified the colors of our**Command Prompt** window for readability.]

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig02a)

C:\examples\debugger>**javac -g AccountTest.java Account.java**  
  
C:\examples\debugger>**jdb**  
Initializing jdb ...  
>

**Fig. E.2** | Starting the Java debugger.

**4. Running an application in the debugger.** Run the AccountTest application through the debugger by typing **run** AccountTest ([Fig. E.3](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig03)). If you do not set any breakpoints before running your application in the debugger, the application will run just as it would using the java command.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig03a)

C:\examples\debugger>**jdb**  
Initializing jdb ... >  
**run AccountTest**  
run AccountTest  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
>  
VM Started: initial account balance: $50.00  
  
adding 25.00 to account balance  
  
new account balance: $75.00  
  
The application exited

**Fig. E.3** | Running the AccountTest application through the debugger.

**5. Restarting the debugger.** To make proper use of the debugger, you must set at least one breakpoint before running the application. Restart the debugger by typing jdb.

**6. Inserting breakpoints in Java.** You set a breakpoint at a specific line of code in your application. The line numbers used in these steps are from the source code in [Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01). Set a breakpoint at line 12 in the source code by typing stop at AccountTest:12 ([Fig. E.4](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig04)). The **stopcommand** inserts a breakpoint at the line number specified after the command. You can set as many breakpoints as necessary. Set another breakpoint at line 19 by typing stop at AccountTest:19 ([Fig. E.4](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig04)). When the application runs, it suspends execution at any line that contains a breakpoint. The application is said to be in **break mode** when the debugger pauses the application’s execution. Breakpoints can be set even after the debugging process has begun. The debugger command stop in, followed by a class name, a period and a method name (e.g., stop in Account.credit) instructs the debugger to set a breakpoint at the first executable statement in the specified method. The debugger pauses execution when program control enters the method.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig04a)

C:\examples\debugger>**jdb**  
Initializing jdb ...  
> **stop at AccountTest:12**  
Deferring breakpoint AccountTest:12.  
It will be set after the class is loaded.  
> **stop at AccountTest:19**  
Deferring breakpoint AccountTest:19.  
It will be set after the class is loaded.  
>

**Fig. E.4** | Setting breakpoints at lines 12 and 19.

**7. Running the application and beginning the debugging process.** Type run AccountTest to execute the application and begin the debugging process ([Fig. E.5](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig05)). The debugger prints text indicating that breakpoints were set at lines 12 and 19. It calls each breakpoint a “deferred breakpoint” because each was set before the application began running in the debugger. The application pauses when execution reaches the breakpoint on line 12. At this point, the debugger notifies you that a breakpoint has been reached and it displays the source code at that line (12). That line of code is the next statement that will execute.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig05a)

It will be set after the class is loaded.  
> **run AccountTest**  
run AccountTest  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
>  
VM Started: Set deferred breakpoint AccountTest:19  
Set deferred breakpoint AccountTest:12  
  
Breakpoint hit: "thread=main", AccountTest.main(), line=12 bci=13  
12          System.out.printf("initial account balance: $%.2f%n",  
  
main[1]

**Fig. E.5** | Restarting the AccountTest application.

**8. Using the** ***cont*** **command to resume execution.** Type cont. The **cont** **command** causes the application to continue running until the next breakpoint is reached (line 19), at which point the debugger notifies you ([Fig. E.6](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig06)). AccountTest’s normal output appears between messages from the debugger.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig06a)

main[1] **cont**  
> initial account balance: $50.00  
  
adding 25.00 to account balance  
  
Breakpoint hit: "thread=main", AccountTest.main(), line=19 bci=60  
19          account.deposit(depositAmount); // add to account balance  
  
main[1]

**Fig. E.6** | Execution reaches the second breakpoint.

**9. Examining a variable’s value.** Type print depositAmount to display the current value stored in the depositAmount variable ([Fig. E.7](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig07)). The **print** **command** allows you to peek inside the computer at the value of one of your variables. This command will help you find and eliminate logic errors in your code. The value displayed is 25.0—the value assigned to depositAmount in line 15 of [Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig07a)

main[1] **print depositAmount**  
 depositAmount = 25.0  
main[1]

**Fig. E.7** | Examining the value of variable depositAmount.

**10. Continuing application execution.** Type cont to continue the application’s execution. There are no more breakpoints, so the application is no longer in break mode. The application continues executing and eventually terminates ([Fig. E.8](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig08)). The debugger will stop when the application ends.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig08a)

main[1] **cont**  
> new account balance: $75.00  
  
The application exited

**Fig. E.8** | Continuing application execution and exiting the debugger.

### E.3 The print and set Commands

In the preceding section, you learned how to use the debugger’s print command to examine the value of a variable during program execution. In this section, you’ll learn how to use the print command to examine the value of more complex expressions. You’ll also learn the **set** **command**, which allows the programmer to assign new values to variables.

For this section, we assume that you’ve followed Step 1 and Step 2 in Section E.2 to open the**Command Prompt** window, change to the directory containing the [Appendix E](http://proquest.safaribooksonline.com/9780133813036/app05_html#app05) examples (e.g.,C:\examples\debugger) and compile the AccountTest application (and class Account) for debugging.

**1. Starting debugging.** In the **Command Prompt**, type jdb to start the Java debugger.

**2. Inserting a breakpoint.** Set a breakpoint at line 19 in the source code by typing stop at AccountTest:19.

**3.****Running the application and reaching a breakpoint**. Type run AccountTest to begin the debugging process ([Fig. E.9](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05fig09)). This will cause AccountTest’s main to execute until the breakpoint at line 19 is reached. This suspends application execution and switches the application into break mode. At this point, the statements in lines 9–13 created an Account object and printed the initial balance of the Account obtained by calling its getBalance method. The statement in line 15 ([Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01)) declared and initialized local variable depositAmount to 25.0. The statement in line 19 is the next statement that will execute.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig09a)

C:\examples\debugger>**jdb**  
Initializing jdb ...  
> stop at AccountTest:19  
Deferring breakpoint AccountTest:19.  
It will be set after the class is loaded.  
> run AccountTest  
run AccountTest  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
>  
VM Started: Set deferred breakpoint AccountTest:19  
initial account balance: $50.00  
  
adding 25.00 to account balance  
  
Breakpoint hit: "thread=main", AccountTest.main(), line=19 bci=60  
19          account.deposit(depositAmount); // add to account balance  
  
main[1]

**Fig. E.9** | Application execution suspended when debugger reaches the breakpoint at line 19.

**4. Evaluating arithmetic and boolean expressions.** Recall from Section E.2 that once the application has entered break mode, you can explore the values of the application’s variables using the debugger’s print command. You can also use the print command to evaluate arithmetic and boolean expressions. In the **Command Prompt** window, type print depositAmount - 2.0. The print command returns the value 23.0 ([Fig. E.10](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05fig10)). However, this command does not actually change the value of depositAmount. In the **Command Prompt** window, type print depositAmount == 23.0. Expressions containing the == symbol are treated as booleanexpressions. The value returned is false ([Fig. E.10](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05fig10)) because depositAmount does not currently contain the value 23.0—depositAmount is still 25.0.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig10a)

main[1] **print depositAmount - 2.0**  
 depositAmount - 2.0 = 23.0  
main[1] **print depositAmount == 23.0**  
 depositAmount == 23.0 = false  
main[1]

**Fig. E.10** | Examining the values of an arithmetic and boolean expression.

**5. Modifying values.** The debugger allows you to change the values of variables during the application’s execution. This can be valuable for experimenting with different values and for locating logic errors in applications. You can use the debugger’s set command to change the value of a variable. Type set depositAmount = 75.0. The debugger changes the value ofdepositAmount and displays its new value ([Fig. E.11](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05fig11)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig11a)

main[1] **set depositAmount = 75.0**  
 depositAmount = 75.0 = 75.0  
main[1]

**Fig. E.11** | Modifying values.

**6. Viewing the application result.** Type cont to continue application execution. Line 19 ofAccountTest ([Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01)) executes, passing depositAmount to Account method credit. Methodmain then displays the new balance. The result is $125.00 ([Fig. E.12](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec3_html#app05fig12)). This shows that the preceding step changed the value of depositAmount from its initial value (25.0) to 75.0.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig12a)

main[1] **cont**  
> new account balance: $125.00  
  
The application exited  
  
C:\examples\debugger>

**Fig. E.12** | Output showing new account balance based on altered value of depositAmount.

### E.4 Controlling Execution Using the step, step up and nextCommands

Sometimes you’ll need to execute an application line by line to find and fix errors. Walking through a portion of your application this way can help you verify that a method’s code executes correctly. In this section, you’ll learn how to use the debugger for this task. The commands you learn in this section allow you to execute a method line by line, execute all the statements of a method at once or execute only the remaining statements of a method (if you’ve already executed some statements within the method).

Once again, we assume you’re working in the directory containing the [Appendix E](http://proquest.safaribooksonline.com/9780133813036/app05_html#app05) examples and have compiled for debugging with the -g compiler option.

**1. Starting the debugger.** Start the debugger by typing jdb.

**2. Setting a breakpoint.** Type stop at AccountTest:19 to set a breakpoint at line 19.

**3. Running the application**. Run the application by typing run AccountTest. After the application displays its two output messages, the debugger indicates that the breakpoint has been reached and displays the code at line 19. The debugger and application then pause and wait for the next command to be entered.

**4. Using the** ***step*** **command**. The **step** **command** executes the next statement in the application. If the next statement to execute is a method call, control transfers to the called method. The stepcommand enables you to enter a method and study the individual statements of that method. For instance, you can use the print and set commands to view and modify the variables within the method. You’ll now use the step command to enter the credit method of class Account ([Fig. 3.8](http://proquest.safaribooksonline.com/9780133813036/ch03lev2sec9_html#ch03fig08)) by typing step ([Fig. E.13](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05fig13)). The debugger indicates that the step has been completed and displays the next executable statement—in this case, line 21 of class Account ([Fig. 3.8](http://proquest.safaribooksonline.com/9780133813036/ch03lev2sec9_html#ch03fig08)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig13a)

main[1] **step**  
>  
Step completed: "thread=main", Account.deposit(), line=24 bci=0  
24          if (depositAmount > 0.0) // if the depositAmount is valid  
  
main[1]

**Fig. E.13** | Stepping into the credit method.

**5. Using the** ***step up*** **command**. After you’ve stepped into the credit method, type **step up**. This command executes the remaining statements in the method and returns control to the place where the method was called. The credit method contains only one statement to add the method’s parameter amount to instance variable balance. The step up command executes this statement, then pauses before line 22 in AccountTest. Thus, the next action to occur will be to print the new account balance ([Fig. E.14](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05fig14)). In lengthy methods, you may want to look at a few key lines of code, then continue debugging the caller’s code. The step up command is useful for situations in which you do not want to continue stepping through the entire method line by line.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig14a)

main[1] step up  
>  
Step completed: "thread=main", AccountTest.main(), line=22 bci=65  
22          System.out.printf("new account balance: $%.2f%n",  
  
main[1]

**Fig. E.14** | Stepping out of a method.

**6. Using the** ***cont*** **command to continue execution.** Enter the cont command ([Fig. E.15](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05fig15)) to continue execution. The statement at lines 22–23 executes, displaying the new balance, then the application and the debugger terminate.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig15a)

main[1] **cont**  
> new account balance: $75.00  
  
The application exited  
  
C:\examples\debugger>

**Fig. E.15** | Continuing execution of the AccountTest application.

**7. Restarting the debugger.** Restart the debugger by typing jdb.

**8. Setting a breakpoint.** Breakpoints persist only until the end of the debugging session in which they’re set—once the debugger exits, all breakpoints are removed. (In Section E.6, you’ll learn how to manually clear a breakpoint before the end of the debugging session.) Thus, the breakpoint set for line 19 in Step 2 no longer exists upon restarting the debugger in Step 7. To reset the breakpoint at line 19, once again type stop at AccountTest:19.

**9. Running the application.** Type run AccountTest to run the application. As in Step 3,AccountTest runs until the breakpoint at line 19 is reached, then the debugger pauses and waits for the next command.

**10. Using the** ***next*** **command**. Type **next**. This command behaves like the step command, except when the next statement to execute contains a method call. In that case, the called method executes in its entirety and the application advances to the next executable line after the method call ([Fig. E.16](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05fig16)). Recall from Step 4 that the step command would enter the called method. In this example, the next command causes Account method credit to execute, then the debugger pauses at line 22 in AccountTest.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig16a)

main[1] **next**  
>  
Step completed: "thread=main", AccountTest.main(), line=22 bci=65  
22          System.out.printf("new account balance: $%.2f%n",  
  
main[1]

**Fig. E.16** | Stepping over a method call.

**11. Using the** ***exit*** **command**. Use the **exit** **command** to end the debugging session ([Fig. E.17](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec4_html#app05fig17)). This command causes the AccountTest application to immediately terminate rather than execute the remaining statements in main. When debugging some types of applications (e.g., GUI applications), the application continues to execute even after the debugging session ends.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig17a)

main[1] **exit**  
  
C:\examples\debugger>

**Fig. E.17** | Exiting the debugger.

### E.5 The watch Command

In this section, we present the **watch** **command**, which tells the debugger to watch a field. When that field is about to change, the debugger will notify you. In this section, you’ll learn how to use the watchcommand to see how the Account object’s field balance is modified during the execution of theAccountTest application.

As in the preceding two sections, we assume that you’ve followed Step 1 and Step 2 in Section E.2 to open the **Command Prompt**, change to the correct examples directory and compile classes AccountTestand Account for debugging (i.e., with the -g compiler option).

**1. Starting the debugger.** Start the debugger by typing jdb.

**2. Watching a class’s field**. Set a watch on Account’s balance field by typing watch Account.balance ([Fig. E.18](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05fig18)). You can set a watch on any field during execution of the debugger. Whenever the value in a field is about to change, the debugger enters break mode and notifies you that the value will change. Watches can be placed only on fields, not on local variables.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig18a)

C:\examples\debugger>**jdb**  
Initializing jdb ...  
> **watch Account.balance**  
Deferring watch modification of Account.balance.  
It will be set after the class is loaded.  
>

**Fig. E.18** | Setting a watch on Account’s balance field.

**3. Running the application**. Run the application with the command run AccountTest. The debugger will now notify you that field balance’s value will change ([Fig. E.19](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05fig19)). When the application begins, an instance of Account is created with an initial balance of $50.00 and a reference to the Account object is assigned to the local variable account (line 9, [Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01)). Recall from [Fig. 3.8](http://proquest.safaribooksonline.com/9780133813036/ch03lev2sec9_html#ch03fig08) that when the constructor for this object runs, if parameter initialBalance is greater than 0.0, instance variable balance is assigned the value of parameter initialBalance. The debugger notifies you that the value of balance will be set to 50.0.

**4. Adding money to the account.** Type cont to continue executing the application. The application executes normally before reaching the code on line 19 of [Fig. E.1](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec2_html#app05fig01) that calls Accountmethod credit to raise the Account object’s balance by a specified amount. The debugger notifies you that instance variable balance will change ([Fig. E.20](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05fig20)). Although line 19 of class AccountTestcalls method deposit, line 25 in Account’s method deposit actually changes the value ofbalance.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig19a)

> **run AccountTest**  
run AccountTest  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
>  
VM Started: Set deferred watch modification of Account.balance  
  
Field (Account.balance) is 0.0, will be 50.0: "thread=main", Acount.<init>(),  
line=18 bci=17  
18             this.balance = balance; // assign to instance variable balance  
  
main[1]

**Fig. E.19** | AccountTest application stops when account is created and its balance field will be modified.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig20a)

main[1] **cont**  
> initial account balance: $50.00  
  
adding 25.00 to account balance  
Field (Account.balance) is 50.0, will be 75.0: "thread=main",  
Account.deposit(), line=25 bci=13  
  
25             balance = balance + depositAmount; // add it to the balance  
  
main[1]

**Fig. E.20** | Changing the value of balance by calling Account method credit.

**5. Continuing execution.** Type cont—the application will finish executing because the application does not attempt any additional changes to balance ([Fig. E.21](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05fig21)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig21a)

main[1] **cont**  
> new account balance: $75.00  
  
The application exited  
  
C:\examples\debugger>

**Fig. E.21** | Continuing execution of AccountTest.

**6. Restarting the debugger and resetting the watch on the variable.** Type jdb to restart the debugger. Once again, set a watch on the Account instance variable balance by typing the watch Account.balance, then type run AccountTest to run the application.

**7. Removing the watch on the field.** Suppose you want to watch a field for only part of a program’s execution. You can remove the debugger’s watch on variable balance by typing**unwatch** Account.balance ([Fig. E.22](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec5_html#app05fig22)). Type cont—the application will finish executing without reentering break mode.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig22a)

main[1] **unwatch Account.balance**  
Removed: watch modification of Account.balance  
main[1] **cont**  
> initial account balance: $50.00  
  
adding 25.00 to account balance  
  
new account balance: $75.00  
  
The application exited  
  
C:\examples\debugger>

**Fig. E.22** | Removing the watch on variable balance.

### E.6 The clear Command

In the preceding section, you learned to use the unwatch command to remove a watch on a field. The debugger also provides the clear command to remove a breakpoint from an application. You’ll often need to debug applications containing repetitive actions, such as a loop. You may want to examine the values of variables during several, but possibly not all, of the loop’s iterations. If you set a breakpoint in the body of a loop, the debugger will pause before each execution of the line containing a breakpoint. After determining that the loop is working properly, you may want to remove the breakpoint and allow the remaining iterations to proceed normally. In this section, we use the compound interest application in [Fig. 5.6](http://proquest.safaribooksonline.com/9780133813036/ch05lev2sec10_html#ch05fig06) to demonstrate how the debugger behaves when you set a breakpoint in the body of a forstatement and how to remove a breakpoint in the middle of a debugging session.

**1. Opening the** **Command Prompt** **window, changing directories and compiling the application for debugging.** Open the **Command Prompt** window, then change to the directory containing the [Appendix E](http://proquest.safaribooksonline.com/9780133813036/app05_html#app05) examples. For your convenience, we’ve provided a copy of theInterest.java file in this directory. Compile the application for debugging by typing javac -g Interest.java.

**2. Starting the debugger and setting breakpoints.** Start the debugger by typing jdb. Set breakpoints at lines 13 and 22 of class Interest by typing stop at Interest:13, then stop at Interest:22.

**3. Running the application.** Run the application by typing run Interest. The application executes until reaching the breakpoint at line 13 ([Fig. E.23](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig23)).

**4. Continuing execution.** Type cont to continue—the application executes line 13, printing the column headings "Year" and "Amount on deposit". Line 13 appears before the for statement at lines 16–23 in Interest ([Fig. 5.6](http://proquest.safaribooksonline.com/9780133813036/ch05lev2sec10_html#ch05fig06)) and thus executes only once. Execution continues past line 13 until the breakpoint at line 22 is reached during the first iteration of the for statement ([Fig. E.24](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig24)).

**5. Examining variable values.** Type print year to examine the current value of variable year(i.e., the for’s control variable). Print the value of variable amount too ([Fig. E.25](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig25)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig23a)

It will be set after the class is loaded.  
> **run Interest**  
run Interest  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
>  
VM Started: Set deferred breakpoint Interest:22  
Set deferred breakpoint Interest:13  
  
Breakpoint hit: "thread=main", Interest.main(), line=13 bci=9  
13          System.out.printf("%s\_s%n", "Year", "Amount on deposit");  
  
main[1]

**Fig. E.23** | Reaching the breakpoint at line 13 in the Interest application.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig24a)

main[1] **cont**  
> Year   Amount on deposit  
Breakpoint hit: "thread=main", Interest.main(), line=22 bci=55  
22             System.out.printf("%4d%,20.2f%n", year, amount);  
  
main[1]

**Fig. E.24** | Reaching the breakpoint at line 22 in the Interest application.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig25a)

main[1] **print year**  
 year = 1  
main[1] **print amount**  
 amount = 1050.0  
main[1]

**Fig. E.25** | Printing year and amount during the first iteration of Interest’s for.

**6. Continuing execution.** Type cont to continue execution. Line 22 executes and prints the current values of year and amount. After the for enters its second iteration, the debugger notifies you that the breakpoint at line 22 has been reached a second time. The debugger pauses each time a line where a breakpoint has been set is about to execute—when the breakpoint appears in a loop, the debugger pauses during each iteration. Print the values of variables year and amountagain to see how the values have changed since the first iteration of the for ([Fig. E.26](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig26)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig26a)

main[1] **cont**  
>    1            1,050.00  
Breakpoint hit: "thread=main", Interest.main(), line=22 bci=55  
22             System.out.printf("%4d%,20.2f%n", year, amount);  
  
main[1] **print amount**  
 amount = 1102.5  
main[1] **print year**  
 year = 2  
main[1]

**Fig. E.26** | Printing year and amount during the second iteration of Interest’s for.

**7. Removing a breakpoint.** You can display a list of all of the breakpoints in the application by typing **clear** ([Fig. E.27](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig27)). Suppose you’re satisfied that the Interest application’s for statement is working properly, so you want to remove the breakpoint at line 22 and allow the remaining iterations of the loop to proceed normally. You can remove the breakpoint at line 22 by typingclear Interest:22. Now type clear to list the remaining breakpoints in the application. The debugger should indicate that only the breakpoint at line 13 remains ([Fig. E.27](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig27)). This breakpoint has already been reached and thus will no longer affect execution.

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig27a)

main[1] **clear**  
Breakpoints set:  
        breakpoint Interest:13  
        breakpoint Interest:22  
main[1] **clear Interest:22**  
Removed: breakpoint Interest:22  
main[1] **clear**  
Breakpoints set:  
        breakpoint Interest:13  
main[1]

**Fig. E.27** | Removing the breakpoint at line 22.

**8.****Continuing execution after removing a breakpoint.** Type cont to continue execution. Recall that execution last paused before the printf statement in line 22. If the breakpoint at line 22 was removed successfully, continuing the application will produce the correct output for the current and remaining iterations of the for statement without the application halting ([Fig. E.28](http://proquest.safaribooksonline.com/9780133813036/app05lev1sec6_html#app05fig28)).

[**Click here to view code image**](http://proquest.safaribooksonline.com/9780133813036/app06_html#pefig28a)

main[1] **cont**  
>    2            1,102.50  
   3            1,157.63  
   4            1,215.51  
   5            1,276.28  
   6            1,340.10  
   7            1,407.10  
   8            1,477.46  
   9            1,551.33  
  10            1,628.89  
  
The application exited  
  
C:\examples\debugger>

**Fig. E.28** | Application executes without a breakpoint set at line 22.

### E.7 Wrap-Up

In this appendix, you learned how to insert and remove breakpoints in the debugger. Breakpoints allow you to pause application execution so you can examine variable values with the debugger’s printcommand. This capability will help you locate and fix logic errors in your applications. You saw how to use the print command to examine the value of an expression and how to use the set command to change the value of a variable. You also learned debugger commands (including the step, step up andnext commands) that can be used to determine whether a method is executing correctly. You learned how to use the watch command to keep track of a field throughout the life of an application. Finally, you learned how to use the clear command to list all the breakpoints set for an application or remove individual breakpoints to continue execution without breakpoints.

# Appendices on the Web

The following appendices are available at Java How to Program, 9/e’s Companion Website ([www.pearsonhighered.com/deitel](http://www.pearsonhighered.com/deitel)) as PDF documents:

• Appendix F, Using the Java API Documentation

• Appendix G, Creating Documentation with javadoc

• Appendix H, Unicode®

• Appendix I, Formatted Output

• Appendix J, Number Systems

• Appendix K, Bit Manipulation

• Appendix L, Labeled break and continue Statements

• Appendix M, UML 2: Additional Diagram Types

• Appendix N, Design Patterns

These files can be viewed in Adobe® Reader® ([get.adobe.com/reader](http://get.adobe.com/reader)).