

How to use Eclipse with *Murach's* *Java Programming* (5th Edition)

The printed version of *Murach's Java Programming (5th Edition)* shows how to use the NetBeans IDE to develop Java applications. However, if you prefer using the Eclipse IDE, you can use this PDF file to learn how to use Eclipse with *Murach's Java Programming*.

Table of Contents

Chapter 1	An introduction to Eclipse	1
Chapter 2	How to debug and deploy with Eclipse	25
Chapter 3	Object-oriented programming with Eclipse	39
Chapter 4	GUI and database programming with Eclipse	61



MIKE MURACH & ASSOCIATES, INC.

1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963

murachbooks@murach.com • www.murach.com

Copyright © 2017 Mike Murach & Associates. All rights reserved.

1

An introduction to Eclipse

This chapter shows how to use Eclipse with chapter 1 of *Murach's Java Programming (5th Edition)*. In addition, the end of this chapter includes Eclipse versions of exercises 1-1 and 1-2. If you complete these exercises, you should have all of the fundamental skills for using Eclipse to develop simple Java applications.

How to work with an existing project	2
How to select a workspace.....	2
How to import a project into a workspace.....	4
How to open a file in the code editor.....	6
How to compile and run a project	8
How to enter input for a console application.....	8
How to work with two or more projects	10
How to remove a project from a workspace	10
How to work with a new project	12
How to create a new project	12
How to create a new class.....	12
How to work with Java source code and files.....	14
How to use the code completion feature	16
How to detect and correct syntax errors.....	18
How to suppress and fix warnings.....	20
Perspective	22

How to work with an existing project

In Eclipse, a *project* is a folder that contains all the folders and files for an application. The easiest way to get started with Eclipse is to import an existing project. For example, you can import the projects for any of the applications presented in the book after you download them from our website as described in the appendixes.

How to select a workspace

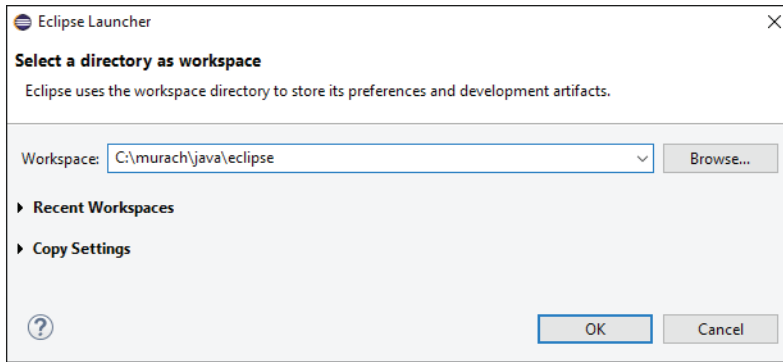
When you start Eclipse for the first time, it displays a dialog box like the one shown in figure 1-1 that allows you to select the workspace. **A workspace is a folder that stores information about how to configure Eclipse, including which projects to display.** In this figure, the dialog box selects a workspace that you can use with *Murach's Java Programming*. However, you can use any workspace you want, including the default workspace that's selected the first time you start Eclipse.

After you select a workspace, Eclipse may display a Welcome page. If so, you can click the Workbench icon to go to a *workbench* like the one shown in figure 1-3. This is where you work on code.

Once you select a workspace, you can view all of the projects available from that workspace as described in the next few figures. If you want to switch to a different workspace, you can select File→Switch Workspace to select the new workspace. This restarts Eclipse, selects the new workspace, and displays its projects.

Once you have selected a workspace, it appears in the Workspace menu that's available from the dialog box shown in this figure. This makes it easy to switch between workspaces.

The dialog box for selecting a workspace



Description

- In Eclipse, a *project* contains the folders and files for a Java application.
- When you start Eclipse, you typically select the workspace you want to use. A *workspace* stores information about how to configure Eclipse, including which projects to display.
- When Eclipse is running, you can switch to a different workspace by selecting File→Switch Workspace. This restarts Eclipse and selects the new workspace.
- Once you have selected a workspace, it appears in the Workspace menu. This makes it easy to switch between workspaces.
- After you select a workspace, Eclipse may display a Welcome page. If so, you can click the Workbench icon to go to the *workbench*, which is where you work on code.

Figure 1-1 How to select a workspace


How to import a project into a workspace

When you select a workspace for the first time, it doesn't display any projects. To change that, you can import an existing project into the workspace as shown in figure 1-2. All of the projects for *Murach's Java Programming* are located within this folder:

```
\murach\java\eclipse
```

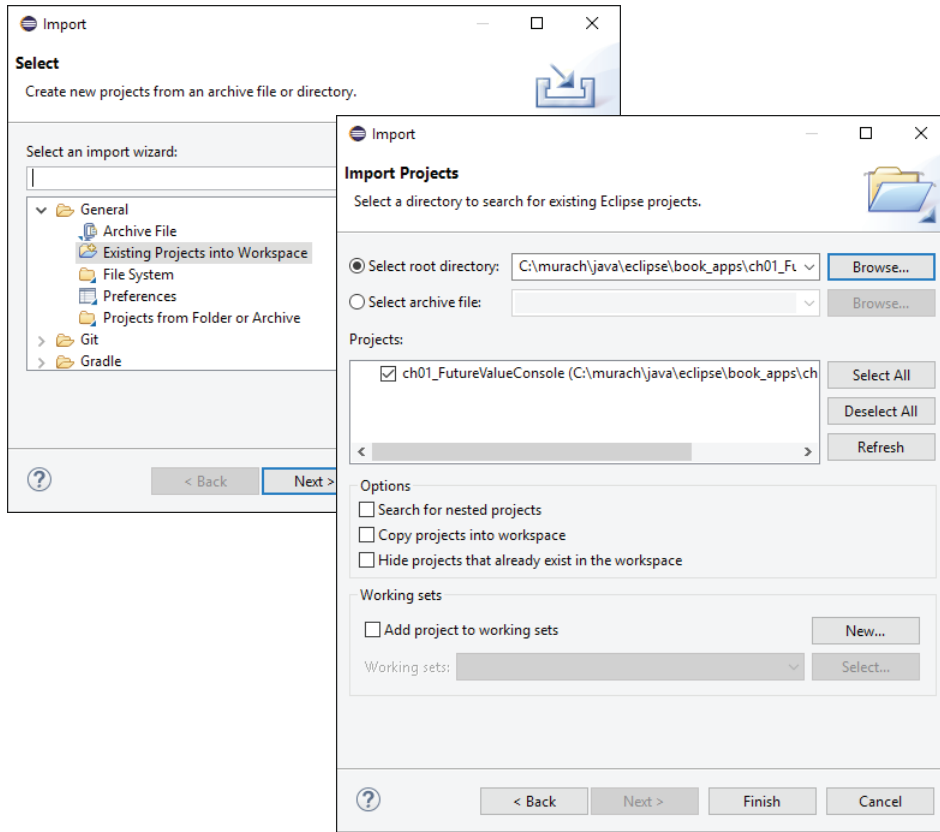
In this figure, I've selected the folder that contains the project named `ch01_FutureValueConsole`. As a result, that's the only project that's shown in the Projects section of the Import Projects dialog box.

If there are multiple projects in the folder you select, all of the projects are displayed in the Projects section of this dialog box. Then, you can import the ones you want by selecting them. For example, if you select the `book_apps` folder, the Projects section displays all projects in that folder. Then, you can select the ones you want to import.

When you import projects, it's possible to copy the project into to the existing workspace.  However, for the purposes of the book, you don't need to do that.

It's also possible to import other types of projects. For example, it's possible to import projects that are stored in archive files. However, for the purposes of the book, the only types of projects you need to import are existing projects as shown in this figure.

The dialog boxes for importing a project



How to import a project

1. Select File→Import from the menu system.
2. In the first dialog box, select General→Existing Projects into Workspace and click on the Next button.
3. In the second dialog box, click the Browse button and navigate to the folder that contains the project you want to import. This should display the project in the Projects pane and select it.
4. Click the Finish button to import the selected project or projects.

How to import all projects in the book_apps folder

- To import all projects in the book_apps folder, navigate to the book_apps folder in step 3. This should select all projects in the book_apps folder.

Figure 1-2 How to import a project into a workspace

How to open a file in the code editor

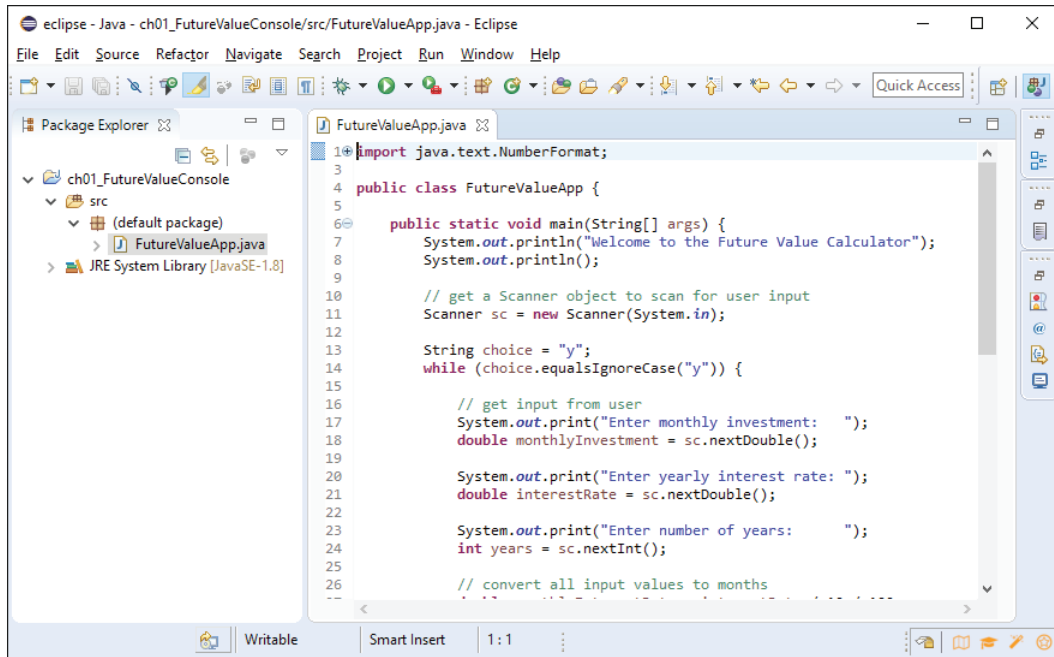
Figure 1-3 shows a workspace that contains a project for a Java application. In this example, the project is named `ch01_FutureValueConsole`.

The Package Explorer shows that the folder for the project contains a subfolder named `src`. This folder contains the source code for the application. The second icon that looks like a stack of books is called a *library*, and it contains the libraries of code that your application uses. In this case, the application uses just the default libraries for Java 8, as indicated by the JRE System Library [JavaSE-1.8] label. However, if your system is set up to use Java 9, the Package Explorer shows that. In addition, you can add more libraries later if necessary.

Within the `src` folder, the source files can be organized into *packages*. In this case, there's only one source file, and it's stored in the default package. For simple applications like this, that's acceptable. However, as you develop more complex applications, it's considered a best practice to store your source files in a package as described in chapter 10.

The application shown here consists of a single source code file named `FutureValueApp.java`. In this figure, this file is open in the *code editor*. Because this class contains the `main()` method for the application, it's called the *main class*. When you run an application, the `main()` method in the main class is executed by default. For now, all you need to know is that you can open a source code file in the code editor by double-clicking on it in the Package Explorer.

The Package Explorer and the code editor



How to navigate through the Package Explorer

- The Package Explorer displays the projects, folders, files, and libraries that make up an Eclipse workspace.
- To expand or collapse the nodes in the Package Explorer, click on the arrows to the left of its folders and files.
- The src (source) folder typically contains one or more *packages*, which are folders that store the .java files that contain the code for your application.
- If it isn't visible, you can display the Package Explorer by selecting Window→Show View→Package Explorer.

How to open a file in the code editor

- You can use the *code editor* to edit the code in a .java file.
- To open a .java file in the code editor, use the Package Explorer to expand the src folder, expand the package that contains the file, and double-click on the .java file.
- At a minimum, a project consists of a single class that contains the *main() method*. The main() method is the starting point for the application. The class that contains the main() method can be referred to as the *main class*.

Figure 1-3 How to open a file in the code editor

How to compile and run a project

Figure 1-4 shows how to compile and run a project. An easy way to run a project is to press Ctrl+F11. Eclipse automatically compiles your project, so there's no need to compile it before running it.

If you want to remove all of the compiled files and force Eclipse to rebuild the entire project, you can use the Clean command as shown in the figure. This sometimes helps get a project to work correctly after you have copied, moved, or renamed some of its files.

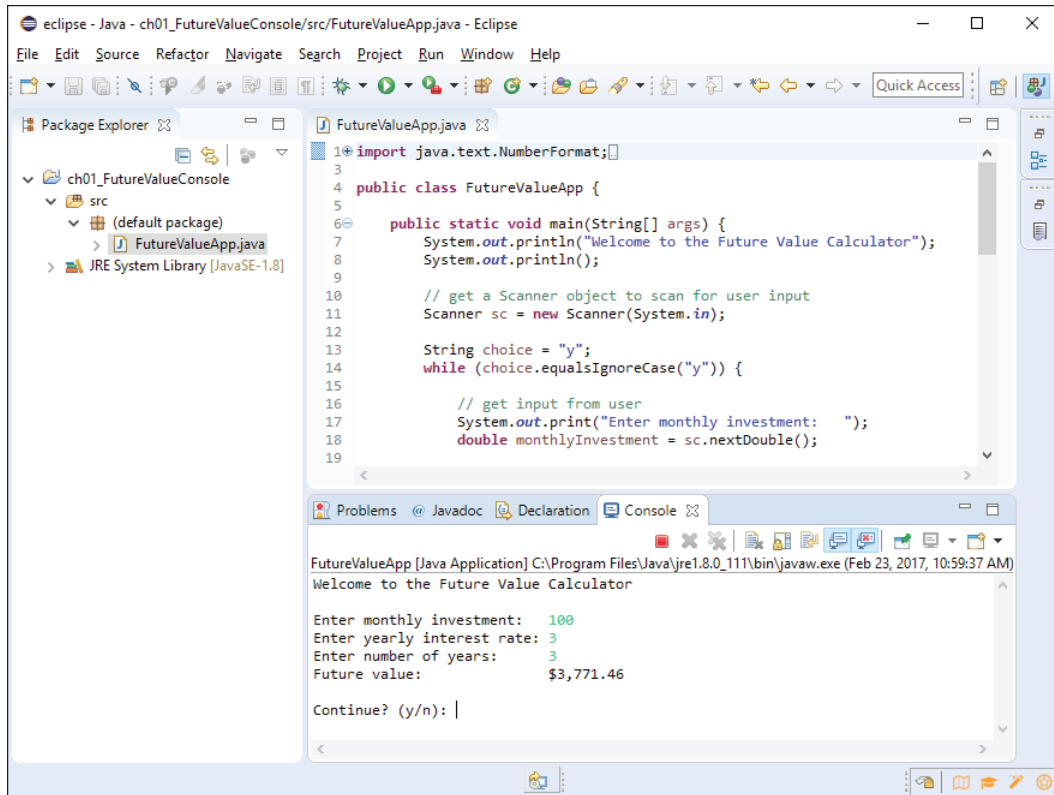
How to enter input for a console application

When you run a console application in Eclipse, any data that's written to the console is displayed in the Console window. In addition, the Console window can accept input.

In this figure, for example, the application started by displaying a welcome message. Then, it prompted the user to enter a monthly investment. At this prompt, the user typed "100" and pressed Enter. After that, the application prompted the user to enter a yearly interest rate and a number of years. At both of these prompts, the user typed "3" and pressed Enter. Then, the application performed the calculation, and displayed the result. After that, the application asked the user if he or she wanted to continue. At this point, the application is still running, and the user can enter "y" to perform another calculation or "n" to end the application.

When you're learning Java, it's common to create applications that use the console to display output and get input. Because of that, the first three sections of the book teach you Java using console applications. Then, section 4 of the book teaches you how to create applications that use a graphical user interface (GUI).

A project that uses the Console window for input and output



How to compile and run a project

- To run the current project, press Ctrl+F11 or click the Run button in the toolbar.
- Eclipse automatically compiles your projects. As a result, you usually don't need to compile a project before you run it.
- To delete all compiled files for a project and compile them again, select the project in the Package Explorer. Then, select Project→Clean.

How to work with the Console window

- When you run an application that prints data to the console, that data is displayed in the Console window.
- When you run an application that requests input from the console, the Console window pauses to accept the input. Then, you can click in the Console window, type the input, and press the Enter key.
- The Console window can also display messages and errors when you run an application.

Figure 1-4 How to compile and run a project

How to work with two or more projects

Up to this point, this chapter has shown a workspace that contains a single project. However, Eclipse lets you import multiple projects into a workspace.

Figure 1-5 presents the skills for working with a workspace that contains two or more projects. When you import multiple projects, all of the imported projects appear in the Package Explorer. Then, when you open any of the files for a project, they appear in separate tabs in the main window. After you open a file, you can run the project for that file by pressing Ctrl+F11 or clicking on the Run button in the toolbar. Or, if you want to run a different project, you can select the project in the Package Explorer and press Ctrl+F11 or click on the Run button.

How to remove a project from a workspace

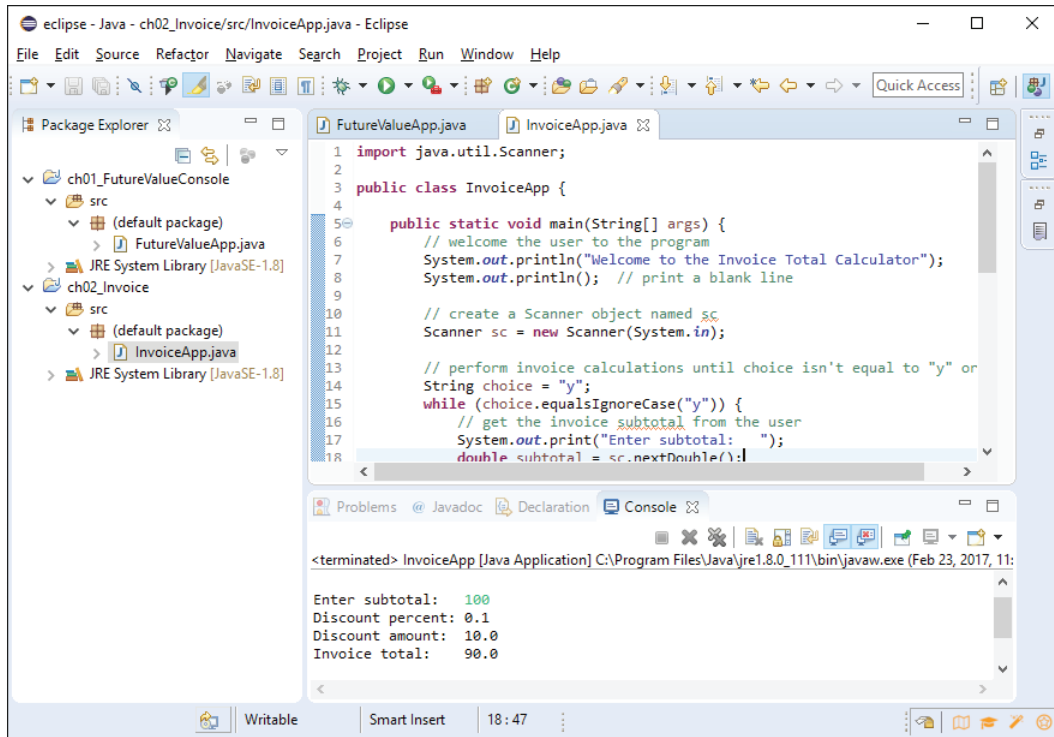
To remove a project from the current workspace, you can right-click on the project in the Project Explorer and select the Delete command from the context menu. This displays a confirmation dialog box that asks if you really want to delete the project from the workspace.

By default, the project is deleted from the current workspace, but the folder and files for the project remain on disk. This allows you to import the project again later if you want. However, if you want to delete the project from disk, you can select the “Delete project contents” checkbox from the confirmation dialog box. This removes the project from the workspace and deletes all of its folders and files.

The book often instructs you to right-click because that’s common in Windows. However, on Mac OS X, right-clicking is not enabled by default. If you want, you can enable right-clicking by editing the system preferences for your mouse. Or, if you prefer, you can hold down the Control key and click instead of right-clicking.

Similarly, the book presents keystrokes that work for Windows. However, with Mac OS X, you may need to modify some of these keystrokes by holding down the Command or Function (Fn) keys. In general, the Mac OS X keys are clearly marked in the menus. As a result, you can look them up if necessary.

Eclipse with two projects in the workspace



How to change the current project

- When you open a file for a project, Eclipse opens the file in a tab in the main window and makes the project associated with that file the current project.
- To change the current project, click on the project in the Package Explorer window.

How to remove a project from the workspace

- To remove a project from the workspace, right-click on the project in the Package Explorer, select Delete, make sure the “Delete project contents” check box is *not* selected, and click the OK button.
- To remove a project from the workspace and delete its folders and files, right-click on the project in the Package Explorer, select the Delete item, select the “Delete project contents” check box, and click the OK button.

Mac OS X notes

- To enable right-clicking, you can edit the system preferences for the mouse.
- To use the Windows keys shown in this book, you may need to hold down the Command or Function (Fn) keys to modify those keys.

Figure 1-5 How to work with two or more projects

How to work with a new project

Now that you understand how to code a class that prints data to the console, you're ready to learn how to use Eclipse to create a project that contains such a class. Then, you can use Eclipse to enter the code for this class.

How to create a new project

Figure 1-6 shows how to create a new project for a Java application. To do that, you can use the first dialog box shown in this figure to specify a name for your project. In this case, I named the project `ch01_Test`.

After entering the name for the project, you can click the Finish button to create the project. This creates the new project in the current workspace. If you've set your workspace to the workspace used by the applications in the book the current workspace should be:

```
\murach\java\eclipse
```

In addition, this creates a folder for the project and all of the support files for the project, including a `src` folder where you can store the source code for your project.

How to create a new class

After you create a new project, you need to add one or more classes to it. To create a new class, navigate to the new project in the Package Explorer and open the New Java Class dialog box as shown in this figure.

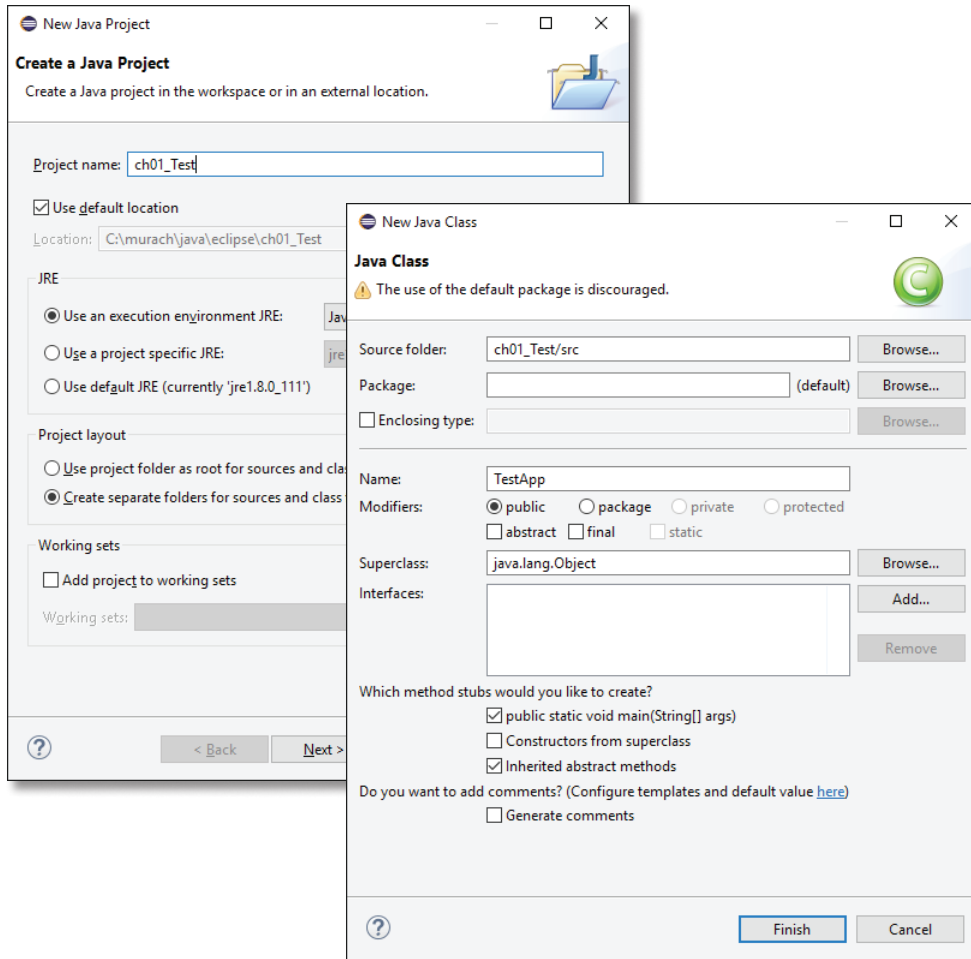
You can use this dialog box to enter a name for the class. This name should start with a capital letter. In this case, I named the class `TestApp`.

You can also use this dialog box to enter a name for the package. Here, I didn't specify a package. As a result, the class is added to the default package. Although it's typically a good idea to use packages to organize the classes in an application, it's acceptable to use the default package when you're first getting started. Then, once you learn how classes work, you can learn how to create and use packages.

If you want to generate a `main()` method for the class, you can select the appropriate check box. In this case, I selected the first check box to generate a `main()` method for the class.

When you click the Finish button, Eclipse creates the file for the class and generates some starting code for the class. If necessary, it also creates the folder for the package.

The dialog boxes for creating a new project and class



How to create a new project

- To create a new project, select File→New→Java Project. Then, enter a name for the project in the resulting dialog box and click the Finish button.

How to create a new class

- To create a new class, right-click on the project in the Package Explorer, and select New→Class. Then, respond to the resulting dialog box.
- To specify a name for the class, enter the name of the class in the Name text box.
- To specify a package, enter a name for the package in the Package text box.
- To add a main() method to the class, select the appropriate checkbox.

Figure 1-6 How to create a new project and class

How to work with Java source code and files

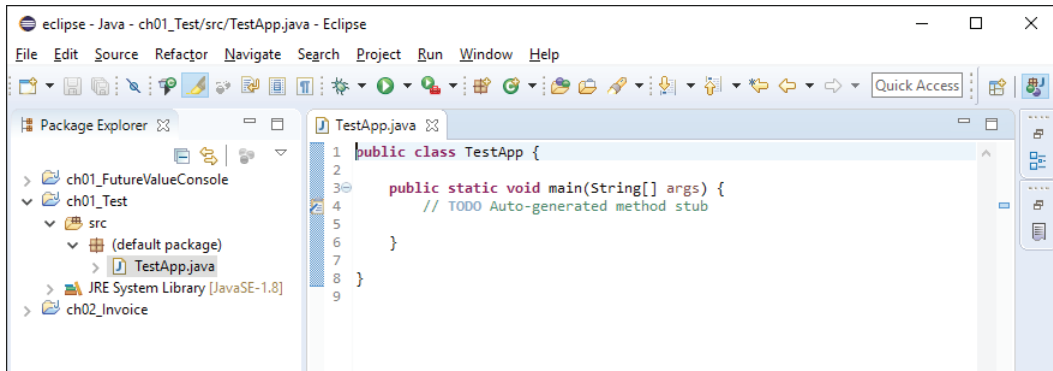
When you create a class, Eclipse typically opens the class in a new code editor window as shown in figure 1-7. To make it easier for you to recognize the Java syntax, the code editor uses different colors for different language elements. In addition, Eclipse provides standard File and Edit menus and keystroke shortcuts that let you save and edit the source code. For example, you can press Ctrl+S to save your source code, and you can use standard commands to cut, copy, and paste code.

When Eclipse creates a new class, it automatically generates some code for you. In this figure, for example, Eclipse generated the code that declares the class and its main() method. Although you can delete or modify the class and method declarations, you don't usually want to do that.

If the source code you want to work with isn't displayed in a code editor window, you can use the Package Explorer to navigate to the .java file. Then, you can double-click on it to open it in a code editor window.

You can also use the Package Explorer to rename or delete a .java file. To do that, just right-click on the file and select the appropriate command. If you rename a file, Eclipse automatically changes both the name of the .java file and the name of the class. Since the name of the .java file must match the name of the class, this is usually what you want.

The code editor with the starting source code for a project



Description

- To open a .java file in the code editor, double-click on it in the Package Explorer window. Then, you can use normal editing techniques to work with the source code.
- To collapse the code for a method or comment, click the minus sign (-) to its left. Then, a plus sign (+) appears to the left of the method or comment, and you can click the plus sign to display the code again.
- To save the source code for a file, select File→Save (Ctrl+S) or click the Save All Files button in the toolbar.
- To rename a file, right-click on it, select Refactor→Rename, and enter the new name in the resulting dialog box.
- To delete a file, you can right-click on it, select Delete, and confirm the deletion in the resulting dialog box.

Figure 1-7 How to work with Java source code and files

How to use the code completion feature

Figure 1-8 shows how to use the *code completion feature*. This feature prevents you from making typing mistakes, and it allows you to discover what fields and methods are available from various classes and objects. In this figure, for example, I started to enter a statement that prints text to the console.

First, I entered “sys” and pressed **Ctrl+Spacebar (both keys at the same time)**. This displayed a list with the System class as the only option. Then, I pressed the Enter key to automatically enter the rest of the class name.

Next, I typed a period. This displayed a list of fields and methods available from the System class. Then, I used the arrow keys to select the field named out and pressed the Enter key to automatically enter that field name.

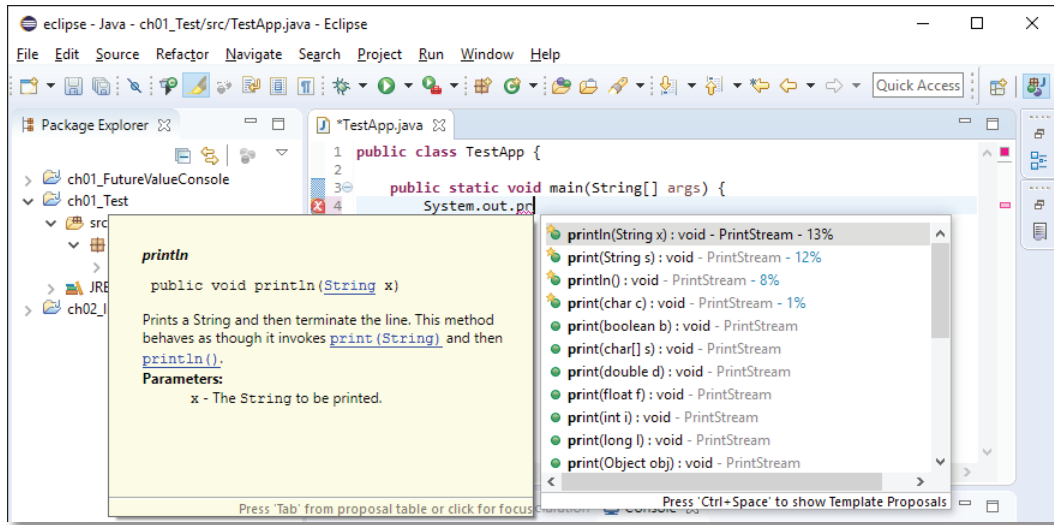
Finally, I typed another period. This displayed a long list of method names. Then, I typed “pr” to scroll down the list to the methods that start with “pr”, and I used the arrow keys to select one of the println() methods as shown in the figure. At this point, I could press Enter to have Eclipse enter the method into the editor for me.

When you use code completion, it automatically enters placeholders from the arguments where they are needed. In this figure, for example, if I pressed the Enter key, Eclipse would insert a placeholder for the argument and highlight the argument. Then, I could replace it with the argument I want to use.

The code completion feature can also make it easy for you to enter values for a string of text. If you type a quotation mark to identify a string, the code completion feature automatically enters both opening and closing quotation marks and places the cursor between the two. At this point, you can enter the text.

If you experiment with the code completion feature, you’ll gradually learn when it helps you enter code more quickly and when it makes sense to enter the code yourself. In addition, you’ll see that it helps you understand the kinds of fields and methods that are available to the various classes and objects that you’re working with. This will make more sense as you learn more about Java in the next few chapters, but it’s extremely helpful to most programmers, especially to those who are new to Java.

The code editor with a code completion list



Description

- You can use the *code completion feature* to help you enter the names of classes and objects and select from the methods and fields that are available for a class or object.
- The first time you use code completion, Eclipse will prompt you to enable it in the popup.
- To activate the code completion feature for entering a class or object name, press Ctrl+Spacebar after entering one or more letters of the class or object name. Then, a list of all the classes and objects that start with those letters is displayed. Or, you can simply wait a short period of time and the code completion window will open automatically.
- To activate the code completion feature for a method or field of a class or object, enter a period after a class or object name. Then, a list of all the methods and fields for that class or object is displayed.
- To insert an item from a code completion list, use the arrow keys to select the item and press the Enter key. If the item requires parentheses, they're added automatically. If the item requires one or more arguments, default values are added for those arguments and the first argument is highlighted so you can enter its value. Then, you can press the Tab key and enter the values for any remaining arguments.
- If you enter the opening quote for a string of text, the code completion feature automatically adds the closing quote and places the cursor between the two quotes so you can enter the text.

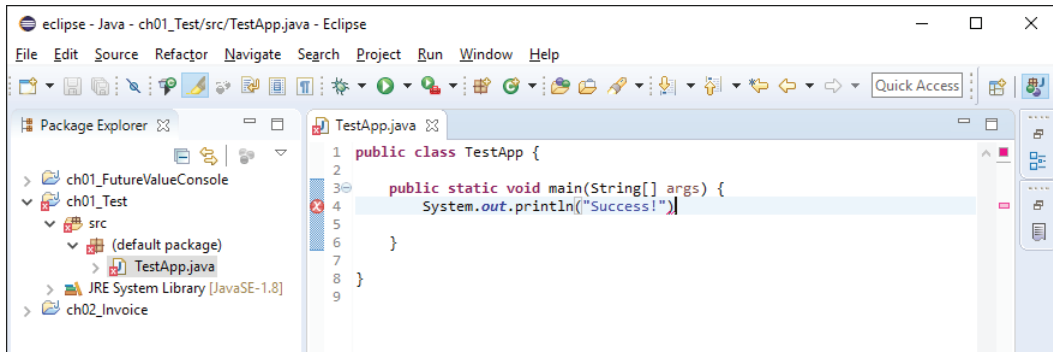
Figure 1-8 How to use the code completion feature

How to detect and correct syntax errors

A statement that won't compile causes a *syntax error*. As you enter text into the code editor, Eclipse identifies syntax errors whenever it detects them. To do that, it displays a red icon to the left of the statement in error as shown in figure 1-9.

If you position the mouse cursor over the red error icon, Eclipse displays a description of the error. In this figure, for example, if you positioned the mouse cursor over the error icon, Eclipse would display a description that indicates that it expects a semicolon at the end of the statement. Then, you could fix the error by typing the semicolon.

The code editor with an error displayed



Description

- Eclipse often detects *syntax errors* as you enter code into the code editor.
- When Eclipse detects a syntax error, it displays a red error icon to the left of the statement in error.
- To get more information about a syntax error, you can position the mouse pointer over the error icon.

Figure 1-9 How to detect and correct syntax errors

How to suppress and fix warnings

As you enter text into the code editor, Eclipse displays *warnings* for code that might cause problems. In figure 1-10, for example, Eclipse displays a warning. This warning is marked with a yellow icon to the left of the statement that caused the warning.

If you position the mouse cursor over the yellow error icon, Eclipse displays a description of the warning. In this figure, for example, if you positioned the mouse cursor over the error icon, Eclipse would display a message that indicates that you have a resource leak since the resource named `sc` is never closed.

At this point, you can click on the icon to display a menu that contains suggestions for suppressing or fixing the warning. From this menu, you can select one of these options by double-clicking on it. In this figure, the leak is not a serious problem since Java should close the resource named `sc` when the `main()` method ends. As a result, I have chosen to suppress the warning by adding a line of code known as an *annotation* immediately above the statement that caused the warning. This line of code looks like this:

```
@SuppressWarnings("resource")
```

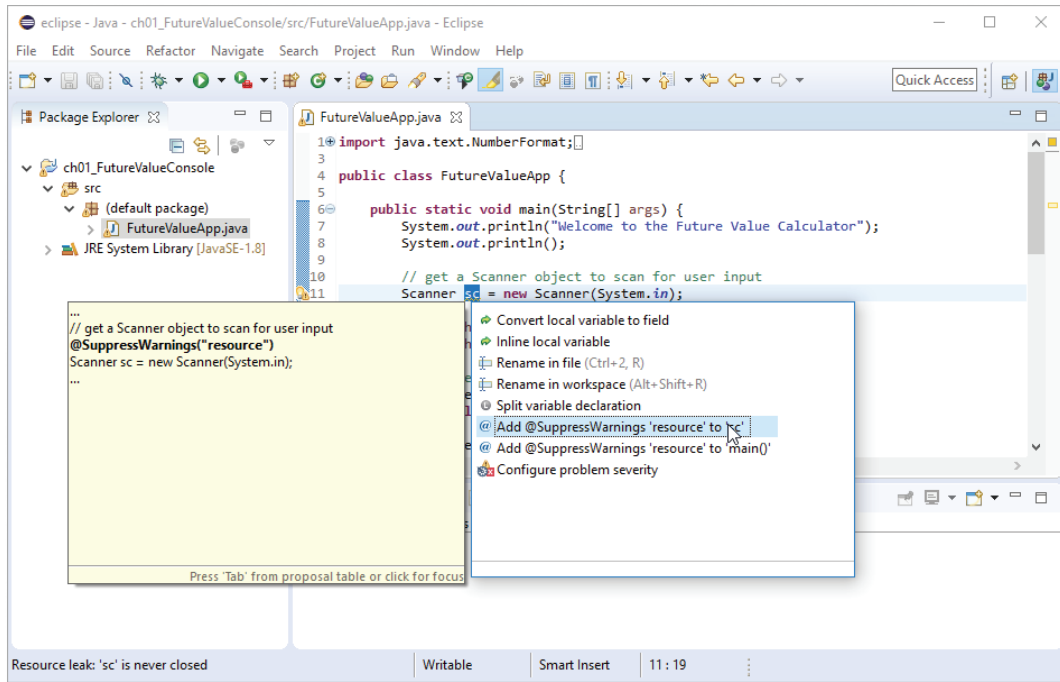
It doesn't change what the code does, but it tells Eclipse to not display a warning for this resource.

In the Eclipse projects that are part of the download for the book, I have added annotations to suppress warnings wherever I think the warning can be safely ignored. If you don't want to suppress warnings, you can delete these annotations, and Eclipse will display the warnings again. Then, if you want, you get rid of the warning by fixing the code. In this case, for example, you could use the menu shown in this figure to convert the local variable to a field. Or, you could add a statement to the end of the `main()` method that closes the resource named `sc`. To do that, you could add a statement like this:

```
sc.close();
```

At this point, you probably don't know the best way to fix a warning. But, you will learn how to fix these warnings as you progress through the book.

The code editor with a warning displayed



Description

- Eclipse often displays *warnings* as you enter code into the code editor.
- When Eclipse displays a warning, it displays a yellow icon to the left of the statement that caused the warning.
- To get more information about the warning, you can position the mouse pointer over the warning icon.
- To suppress a warning, you can often click on the warning icon and select one of the “Add @SuppressWarnings” items from the resulting menu. The downloadable code for this book suppresses warnings that don’t lead to serious problems.
- To fix a warning, you can often click on the warning icon and select one of the items from the resulting menu. However, fixing a warning often requires understanding how the Java code works, which is what you’ll learn as you progress through this book.

Figure 1-10 How to suppress and fix warnings

Perspective

In this chapter, you learned how to use Eclipse to create and run a Java application. With that as background, you're ready to learn how to write your own Java applications. But first, I recommend that you familiarize yourself with Eclipse by doing the exercises that follow.

Summary

- In Eclipse, a *project* contains the folders and files for a Java application.
- When you start Eclipse, you typically select the workspace you want to use. A *workspace* stores information about how to configure Eclipse, including which projects to display.
- In Eclipse, the *workbench* is where you work on code.
- The *src* (source) folder typically contains one or more *packages*, which are folders that store the .java files that contain the code for your application.
- You can use the *code editor* to edit the code in a .java file.
- Java code is stored in *classes*. The *main class* of an application is the class that contains the *main()* *method*, which is the starting point of the application.
- Eclipse often detects *syntax errors* and *warnings* as you enter code into the code editor.

Before you do the exercises for this chapter

Before you do any of the exercises for the book, you need to install the JDK and Eclipse. In addition, you need to install the source code for the book from our website (www.murach.com). For complete instructions, see the appendixes of the book.

Exercise 1-1 Use Eclipse to open and run two projects

This exercise shows you how to use Eclipse to open and run two console applications.

Open and run the Invoice application

1. Start Eclipse.
2. Import the project named `ch01_ex1_Invoice`. On a Windows system, the project should be stored in this directory:
`C:\murach\java\eclipse\ex_starts`
3. Open the `InvoiceApp.java` file in the code editor and review its code to get an idea of how this application works.

4. Press Ctrl+F11 to run the application. Enter a subtotal when you're prompted, and then enter "n" when you're asked if you want to continue.

Open and run the Test Score application

5. Import the project named ch01_ex2_TestScore. Then, open the TestScoreApp.java file in the code editor and review its code.
6. Click the Run button in the toolbar to run the application. Enter one or more scores when you're prompted, and enter 999 to end the application.

Run the applications again

7. Select the Invoice application in the Package Explorer. Then, press Ctrl+F11 to run this application.
8. Select the Test Score application in the Package Explorer and click the Run button to run this application.
9. Remove both projects from the workspace, but don't delete the files for the projects from the disk.

Exercise 1-2 Use Eclipse to develop an application

This exercise guides you through the process of using Eclipse to enter, save, compile, and run a simple application.

Enter the source code and run the application

1. Start Eclipse.
2. From the menu system, select the File→New→Java Project command. Then, use the resulting dialog boxes to create a Java project named ch01_Test, and store the project in this directory:

```
\murach\java\eclipse\ex_starts
```

3. Add a class named TestApp to the default package. Make sure to select the option that generates a main() method for this class.
4. Modify the generated code for the TestApp class so it looks like this (type carefully and use the same capitalization):

```
public class TestApp {  
    public static void main(String[] args) {  
        System.out.println("Success!");  
    }  
}
```

5. Press Ctrl+F11 to compile and run the application. This should display "Success!" in the Console window.

Use the code completion feature

6. Enter the statement that starts with System.out again, right after the first statement. This time, type "sys" and then press Ctrl+Spacebar. Then, use the code completion feature to select the System class, and complete the statement.

7. Enter this statement a third time, right after the second statement. This time, type `System`, enter a period, and select out from the list that's displayed. Then, enter another period, select the `println()` method, and complete the statement. You should now have the same statement three times in a row.
8. Run the application again. It should display the message three times in a row in the Console window.

Introduce and correct a syntax error

9. In the code editor window, delete the semicolon at the end of the first `println()` method. When you do, Eclipse should display an error icon to the left of the statement.
10. Correct the error. When you do, Eclipse should remove the error icon.
11. Use the `File→Save` command (`Ctrl+S`) to save the changes.

Introduce and suppress a warning

12. In the code editor window, delete the annotation that suppresses the resource leak warning. When you do, Eclipse should display a warning.
13. Position the mouse cursor over the yellow icon that's displayed on the left side of the code editor window to read the warning message.
14. Click on the yellow icon to display the context menu. Then, double-click on the item that says "Add `@SuppressWarnings` 'resource' to 'sc'". This should add the annotation that suppresses this warning.
15. Press `Ctrl+S` to save the changes. This should remove the warning.

2

How to debug and deploy with Eclipse

This chapter shows how to use Eclipse with chapter 6 of *Murach's Java Programming (5th Edition)*. In addition, the end of this chapter includes Eclipse versions of the exercises for chapter 6. If you complete these exercises, you should have all of the fundamental skills for using Eclipse to debug and deploy simple Java applications.

How to use Eclipse to debug an application	26
How to set and remove breakpoints	26
How to step through code.....	28
How to inspect variables.....	28
How to inspect the stack trace.....	30
How to use Eclipse to deploy an application	32
How to use Eclipse to create an executable JAR file	32
Perspective	34

How to use Eclipse to debug an application

To find and fix the errors that occur as you test an application, you use a technique known as *debugging*. Debugging is one of the most difficult and frustrating parts of programming. Fortunately, Eclipse includes a powerful tool called a *debugger* that can help you find and fix these errors.

How to set and remove breakpoints

The first step in debugging an application is to figure out what is causing the bug. To do that, it's often helpful to view the values of the variables at different points in the application's execution. This often helps you determine the cause of the bug, which is critical to debugging the application.

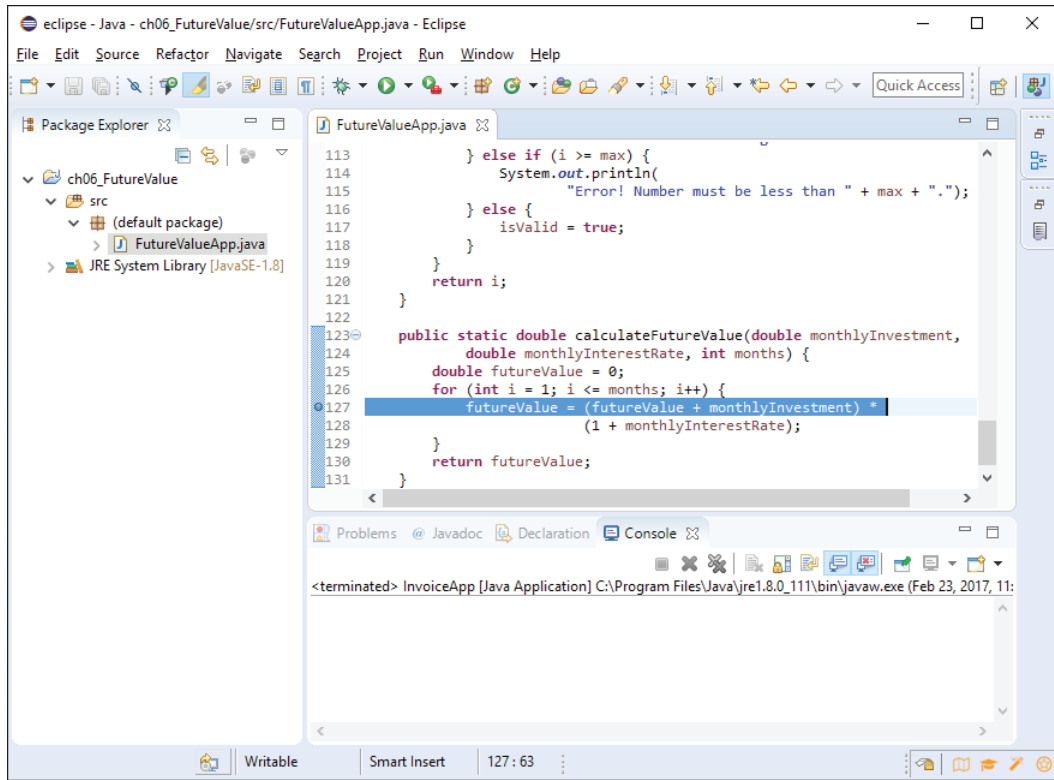
The easiest way to view the variable values as an application is executing is to set a *breakpoint* as shown in figure 2-1. To do that, you double-click on the line number to the left of the line of code. Then, the breakpoint is marked by a blue circle. Later, when you run the application with the debugger, execution will stop just prior to the statement at the breakpoint. Then, you will be able to view the variables that are in scope at that point in the application. You'll learn more about that in the next figure.

When debugging, it's important to set the breakpoint before the line in the application that's causing the bug. Often, you can figure out where to set a breakpoint by reading the runtime exception that's displayed when your application crashes. Sometimes, though, you have to experiment before finding a good location to set a breakpoint.

After you set the breakpoint, you need to run the current application with the debugger. To do that, you can use the Debug button that's available from the toolbar (just to the left of the Run button).

Once you set a breakpoint, it remains set until you remove it. That's true even if you exit from Eclipse. As a result, when you want to remove a breakpoint, you must do it yourself. One way to do that is to double-click the line number again.

A code editor window with a breakpoint



Description

- When debugging, you need to stop application execution before the line of code that caused the error. Then, you can examine variables and step through code as described in the next few figures.
- To stop application execution, you can set a *breakpoint*. Then, when you run the application, execution stops when it reaches the breakpoint.
- To set a breakpoint for a line, open the code editor for the class and double-click on the line number. The breakpoint is identified by a small blue circle that's placed to the left of the line number.
- To remove a breakpoint, double-click on the line number again.
- You can set and remove breakpoints either before you start debugging or while you're debugging. In most cases, you'll set at least one breakpoint before you start debugging.
- To start debugging for the current project, click the Debug button on the toolbar.
- You can also start debugging by right-clicking on a project or a class that contains a `main()` method and selecting Debug As → Java Application.

Figure 2-1 How to set and remove breakpoints

How to step through code

Until now, you've been using Eclipse in the Java perspective. However, when you click the Debug button to run an application with the debugger, Eclipse switches to the Debug perspective as shown in figure 2-2. But first, Eclipse may prompt you to confirm this perspective switch. After that, you can switch back to the Java perspective at any time by clicking the Java button that's on the right side of the toolbar.

When you run an application with the debugger and it encounters a breakpoint, execution stops just prior to the statement at the breakpoint. Once execution stops, a blue arrow marks the next statement to be executed. In addition, Eclipse displays the Variables window shown in the figure. This window shows the values of the variables that are in scope at the current point of execution.

Eclipse also displays the Debug toolbar while you're debugging. You can click the Step Over and Step Into buttons on this toolbar repeatedly to step through an application one statement at a time. Then, you can use the Variables window to observe how and when the variable values change as the application executes. That can help you determine the cause of a bug.

As you step through an application, you can click the Step Over button if you want to execute a method without stepping into it. Or, you can use the Step Return button to step out of any method that you don't want to step through. When you want to continue normal execution, you can click the Resume button. Then, the application will run until the next breakpoint is reached. Or, you can use the Terminate button to end the application's execution.

There's one additional button to the left of the Resume button you should be aware of: the Skip All Breakpoints button. If this button is active, Eclipse will not stop at any of your breakpoints, even in debugging mode.

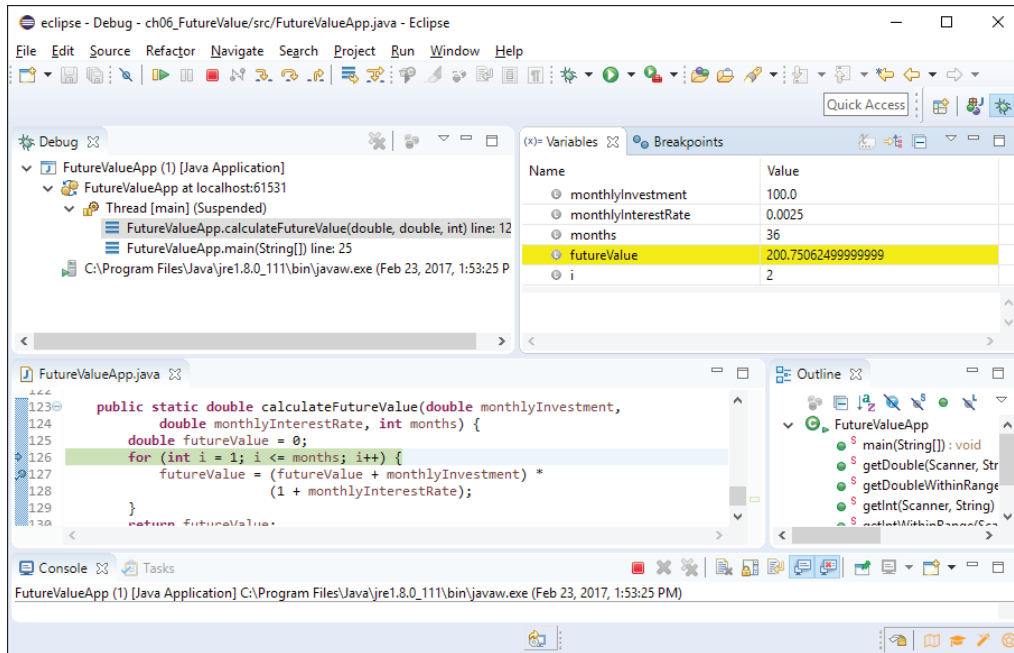
These are powerful debugging features that can help you find the cause of serious programming problems. Stepping through an application is also a good way to understand how the code in an existing application works. If, for example, you step through the Line Item application presented in chapter 5 of the book, you'll get a better idea of how that application works.

How to inspect variables

When you set breakpoints and step through code, the Variables window automatically displays the values of the variables that are in scope. In this figure, the execution point is in the calculateFutureValue() method of the FutureValueApp class. Here, the Variables window shows the values of the three parameters that are passed to the method (monthlyInvestment, monthlyInterestRate, and months) and two local variables that are declared within the method (futureValue and i).

For numeric variables and strings, the value of the variable is shown in the Variables window. However, when an object such as one that's created from the Scanner class is displayed in the Variables window, it doesn't display the values

A debugging session



Some of the buttons on the Debug toolbar

Button	Keyboard shortcut	Description
Step Over	F6	Steps through the code one statement at a time, skipping over called methods.
Step Into	F5	Steps through the code one statement at a time, including statements in called methods.
Step Return	F7	Finishes executing the code in the current method and returns to the calling method.
Resume	F8	Continues execution until the next breakpoint.
Terminate	Ctrl+F2	Ends the application's execution.

Description

- When a breakpoint is reached, execution is stopped before the line is executed.
- The arrow in the bar at the left side of the code editor shows the line that will be executed next. This arrow may overlap with the breakpoint symbol.
- The Variables window shows the values of the variables that are in scope for the current method. This window is displayed by default when you start a debugging session. If you close it, you can open it again by selecting Window→Show View→Variables.
- If a variable in the Variables window refers to an object, you can view the values for that object by clicking the arrow to the left of the object name to expand it.
- You can use the buttons on the Debug toolbar to control the execution of an application.

Figure 2-2 How to step through code and inspect variables

of its variables automatically. Instead, it displays a plus sign to the left of the object name. Then, you can view the values for the object by clicking on that plus sign to expand it.

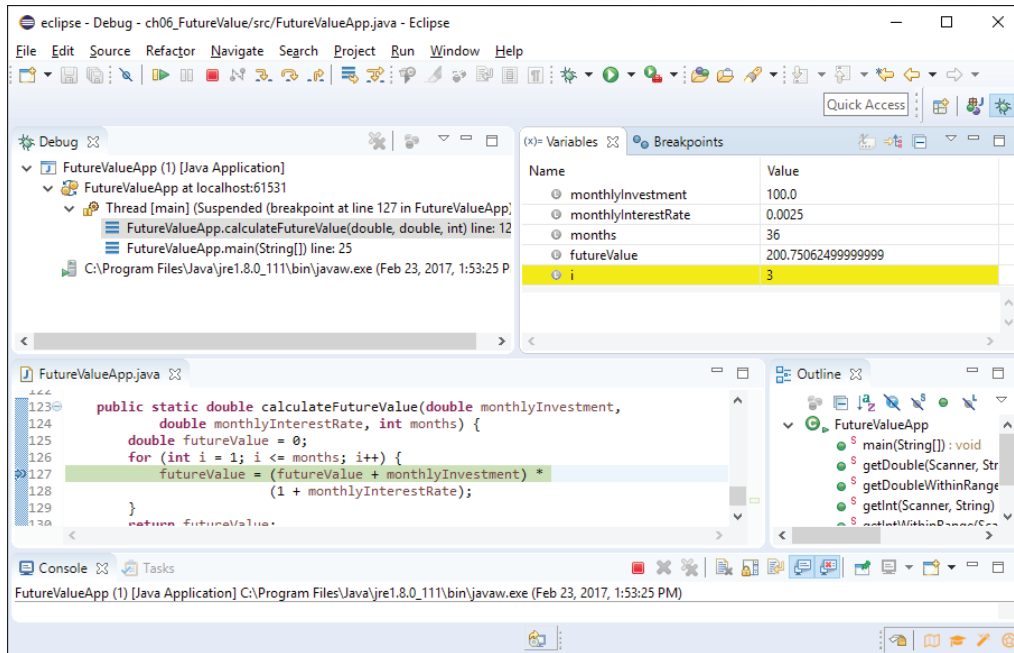
In chapter 7 of *Murach's Java Programming*, you'll learn how to create objects from classes that you define. Then, if the code within one of these objects is executing, you'll see a variable named *this* in the Variables window. This is a keyword that's used to refer to the current object, and you can expand it to view the values of the variables that are defined by the object. That will make more sense when you start to learn about object-oriented programming.

How to inspect the stack trace

When you're debugging, it's sometimes helpful to view the *stack trace*, which is a list of methods in the reverse order in which they were called. By default, **Eclipse displays a stack trace in the Debug window in its top left corner.**

The Debug window in figure 2-3 shows that code execution is on a line of the `calculateFutureValue()` method of the `FutureValueApp` class. This window also shows that this method was called by line 25 of the `main()` method of the `FutureValueApp` class. At this point, you may want to display line 25 of the `main()` method to view the code that called the `calculateFutureValue()` method. To do that, you can click on the `main()` method in the stack trace. This displays the source code for the `FutureValueApp` class in the code editor, opening it if necessary. If you experiment with this, you'll find that it can help you locate the origin of a bug.

A debugging session with the Call Stack window displayed



Description

- A *stack trace* is a list of the methods that have been called in the reverse order in which they were called.
- By default, Eclipse displays a stack trace in the Debug window that's located in the upper left corner of the IDE. If you close it, you can open it again by selecting Window→Show View→Debug.
- To jump to a line of code in the code editor that's displayed in the stack trace, click on that line in the stack trace.
- When you are finished with the debugger, you can return to the Java perspective by clicking the Java button that's displayed on the right side of the toolbar next to the Debug button.

Figure 2-3 How to inspect the stack trace

How to use Eclipse to deploy an application

One simple way to deploy an application is to create an executable JAR file that you can distribute to your users.

How to use Eclipse to create an executable JAR file

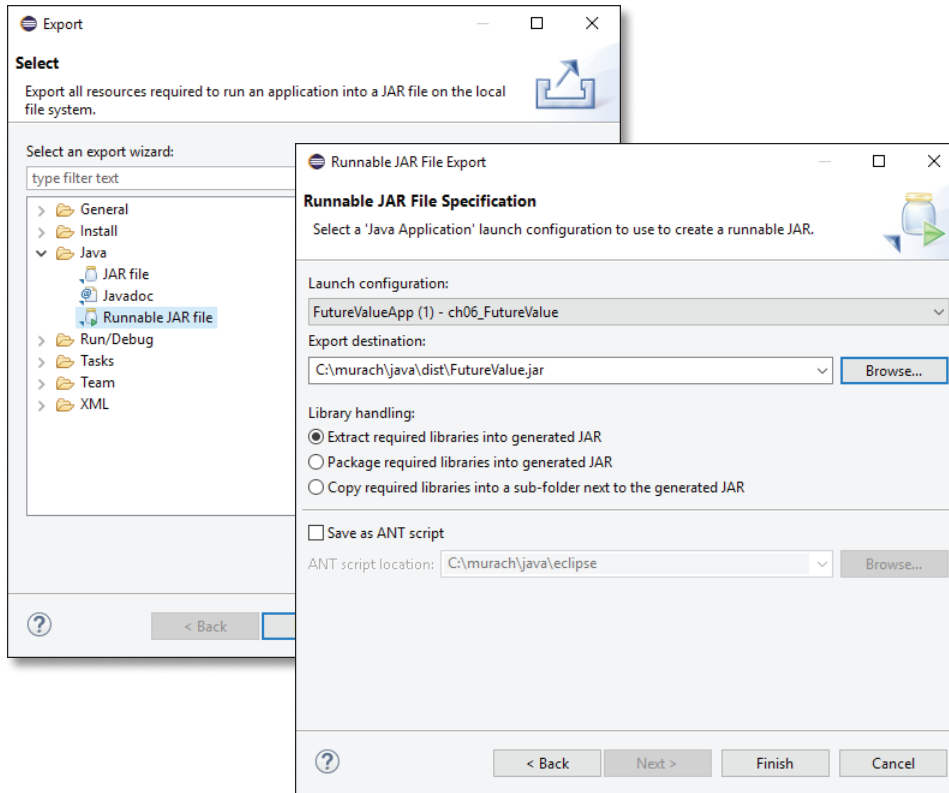
Figure 2-4 shows how to create an executable JAR file. Fortunately, if you're using Eclipse, all you need to do is to select the Export command of the File menu. Then, you can use the resulting dialog boxes to create an executable JAR file.

By default, the JAR file includes the .class files for the project, but does not include the .java files that contain the source code for the project. That makes sense because you typically don't want to make the source code available to users. The JAR file also contains any other files necessary to run the application, including any Java libraries that are needed by the application.

In addition to the class files and libraries, a JAR file always contains a *manifest file*. This file stores additional information about the files in the JAR file, including which .class file contains the main() method for the application.

Although it's possible to distribute the files for an application without storing them in a JAR file, it's almost always better to use a JAR file. This makes it easier to manage the files of the application. In addition, since a JAR file uses a compressed format, it can dramatically improve the download time for an application that's deployed to the web.

The dialog boxes for creating an executable JAR file



How to create an executable JAR file

1. In the Package Explorer, select the project.
2. From the menu system, select File→Export. This should display the first dialog box.
3. Expand the Java node, select the “Runnable JAR file” option, and click the Next button. This should display the second dialog box.
4. Select a launch configuration.
5. Use the “Export destination” option to specify a location for the JAR file.
6. Click the Finish button.

Description

- The executable JAR file contains all of the files necessary to run the application, including any Java libraries that are needed by the application.
- Because a JAR file uses a compressed format, it can dramatically improve the download time for an application that’s deployed to the web.
- Launch configurations are created when you run a project and consist of the name of the project and the class that contains the main() method.

Figure 2-4 How to create an executable JAR file

Perspective

The skills presented in this chapter should give you a solid foundation for using Eclipse to debug any application that you develop. However, Eclipse provides some additional features that you can use to debug your applications. After reading this chapter, you should have the background you need to learn more about other these features.

Once you're sure that an application is free of bugs, you can deploy it to the computers where it will be used. One way to do that is to create an executable JAR file for the application as described in this chapter.

Summary

- Eclipse includes a powerful tool known as a *debugger* that can help you find and fix errors.
- When debugging, Eclipse switches from the Java perspective to the Debug perspective. You can click on the Java and Debug buttons in the toolbar to switch between these perspectives.
- You can set a *breakpoint* on a line of code to stop code execution just before that line of code. Then, you can step through the code and view the values of the variables as the code executes.
- A *stack trace* is a list of methods in the reverse order in which they were called.

Exercise 2-1 Test and debug the Invoice application

This exercise guides you through the process of using Eclipse to test and debug an application.

Test the Invoice application with invalid data

1. Open the ch06_ex1_Invoice project, and test the Invoice application with an invalid subtotal like \$1000 (enter the dollar sign too). This should cause the application to crash with a runtime error and to display an error message in the Console window.
2. Study the error message, and note the line number of the statement in the InvoiceApp class that caused the crash.
3. Click on the link to that line of code. This should open the InvoiceApp.java file in the code editor and highlight the line of code that caused the crash. From this information, determine the cause of the problem and fix it.

Set a breakpoint and step through the application

4. Set a breakpoint on this line of code:
`double discountPercent = 0.0;`

5. Make sure that the `ch06_ex1_Invoice` project is selected in the Package Explorer. Then, click the Debug button in the toolbar. This runs the project with the debugger on. If Eclipse prompts you to switch to the Debug perspective, click the Yes button.
6. If necessary, click the Console tab to display the Console window. Then, enter a value of 100 for the subtotal when prompted by the application. When you do, the application runs to the breakpoint and stops.
7. If necessary, click the Variables tab to display the Variables window. Then, note that the `choice` and `customerType` variables have been assigned values.
8. Click the Step Into button in the toolbar repeatedly to step through the application one statement at a time. After each step, review the values in the Variables window to see how they have changed. Note how the application steps through the if/else statement based on the subtotal value.
9. Click the Resume button in the toolbar to continue the execution of the application.
10. In the Console window, enter “y” to continue and enter a value of 50 for the subtotal.
11. Display the Variables window again and inspect the values of the variables.
12. Click the Step Over button in the toolbar repeatedly to step through the application one statement at a time. After each step, review the values in the Variables window to see how they have changed.
13. When you’re done inspecting the variables, click the Terminate button to end the application. This should give you some idea of how useful the Eclipse debugging tools can be.
14. On the right side of the toolbar, click the Java button to switch back to the Java perspective.

Exercise 2-2 Test and debug the Future Value application

In this exercise, you’ll use Eclipse to find and fix syntax errors and a logic error in the Future Value application.

Use Eclipse to correct the syntax errors

1. Open the `ch06_ex2_FutureValue` project, and then display the `FutureValueApp.java` file. Note that the `getDouble()` method in this file contains syntax errors.
2. Use Eclipse to find and fix the errors.

Use `println` statements to trace code execution

3. Scroll down to the `calculateFutureValue()` method, and add a `println` statement within the loop that prints the value of the month and the future value each time the loop is executed.

4. Run the application to see how the `println` statement works. Review the values that are displayed in the Console window, and notice that the future value increases by too much each month.
5. Comment out the `println` statement so it no longer prints messages to the Console window.

Step through the application

6. In the `calculateFutureValue()` method, set a breakpoint on the statement that calculates the future value.
7. Run the application and enter values when prompted. The application should stop at the breakpoint.
8. Experiment with the Step Into, Step Over, and Step Return buttons as you step through the code of the application. At each step, notice the values that are displayed in the Variables window and use them to find the logic error.
9. When you're done experimenting, click on the Terminate button and remove the breakpoint.
10. Fix the logic error and then run the application again to be sure it produces correct results.
11. On the right side of the toolbar, click the Java button to switch back to the Java perspective.

Exercise 2-3 Deploy the Future Value application

In this exercise, you'll deploy the console version of the Future Value application and then run it.

1. Open the project named `ch06_ex3_FutureValue` that's in the `ex_starts` directory. Then, run the application to see how it works.
2. In the `ex_starts` directory, create a subdirectory named `fv`.
3. Export the project into an executable JAR file and store that JAR file in the `fv` directory.
4. Start a console and use the `java` command to run the JAR file.
5. Within the `fv` directory, create a script file to start the console and execute the JAR file. For Windows, create a `.bat` file. For Mac, create a `.sh` file.
6. Copy the `fv` directory to another computer and make sure you can run it on that computer. To get this to work, you may need to download the JRE from www.java.com, and you may need to modify the script file so it specifies the correct directory for the script file.

Exercise 2-4 Deploy a GUI version of the Future Value application

In this exercise, you'll deploy the GUI version of the Future Value application and then run it.

1. Open the project named `ch06_ex4_FutureValueGUI` that's in the `ex_starts` directory. Then, run the application and see how it works.
2. Export the project to an executable JAR file and store that JAR file in the `ex_starts` directory.
3. Double-click on the JAR file to run it.
4. Copy the JAR file to another computer and double-click on it to run it. To get this to work, you may need to download the JRE from www.java.com.

Object-oriented programming with Eclipse

Eclipse has many features that make it easier to develop object-oriented applications. For example, Eclipse makes it easy to generate get and set methods for a class, to begin coding a class that implements an interface, and to work with packages and modules. However, these skills don't make sense until you understand object-oriented programming. As a result, you may want to read chapters 7 through 10 of *Murach's Java Programming (5th Edition)* before you read the topics that follow.

How to work with classes and interfaces	40
How to create a new class.....	40
How to work with classes	42
How to work with interfaces.....	44
More object-oriented skills.....	46
How to work with packages.....	46
How to work with libraries	48
How to install the plug-in for Java 9 support.....	50
How to create modules	52
How to use modules.....	54
How to generate documentation	56
How to view documentation	58
Perspective	60

How to work with classes and interfaces

The first three figures in this chapter show how to use Eclipse to work with classes and interfaces as described in chapters 7 and 9 of *Murach's Java Programming (5th Edition)*. This includes Eclipse skills for creating a new class or interface as well as skills for generating methods for a class.

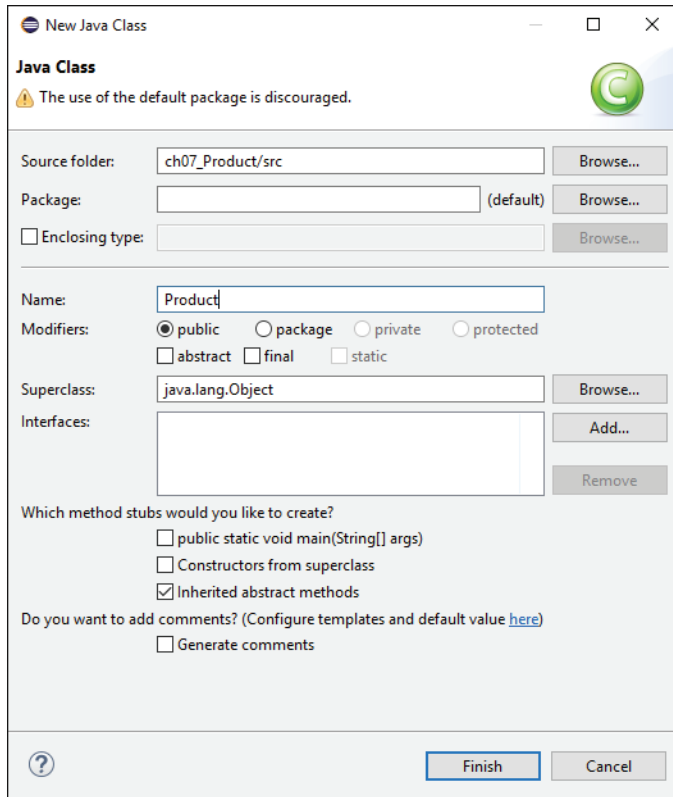
How to create a new class

When you develop object-oriented applications, you'll frequently need to add new classes to your projects. To do that with Eclipse, you can use the New Java Class dialog box shown in figure 3-1. In this figure, this dialog box is being used to create a class named `Product`.

Notice here that a package isn't specified for the class. Because of that, the class will be stored in the default package. Although it's typically a good idea to use packages to organize the classes in an application, it's acceptable to use the default package when you're first getting started. Then, once you learn how classes and interfaces work, you can learn how to create and use packages.

When you complete the New Java Class dialog box, Eclipse creates a file that stores the Java code for the class. For the `Product` class in this figure, that file will be named `Product.java`. Eclipse also generates the starting code for the class as shown in this figure. Note that the name of the class matches the name of the file, which is required. In addition, the public *access modifier* is used so the class can be accessed from other classes.

The dialog box for creating a new Java class



The code that's generated for the Product class

```
public class Product {
}
```

Description

- To create a new class, right-click on the package where you want to add the class, select New → Class, and respond to the resulting dialog box. At the least, you should enter a name for the class in the Name text box.
- Although this dialog box encourages you to select a package for the class, this isn't required. If you don't select a package for the class, Eclipse will store the class in the default package. To learn how to create and use packages, see figure 3-4.

Figure 3-1 How to create a new class

How to work with classes

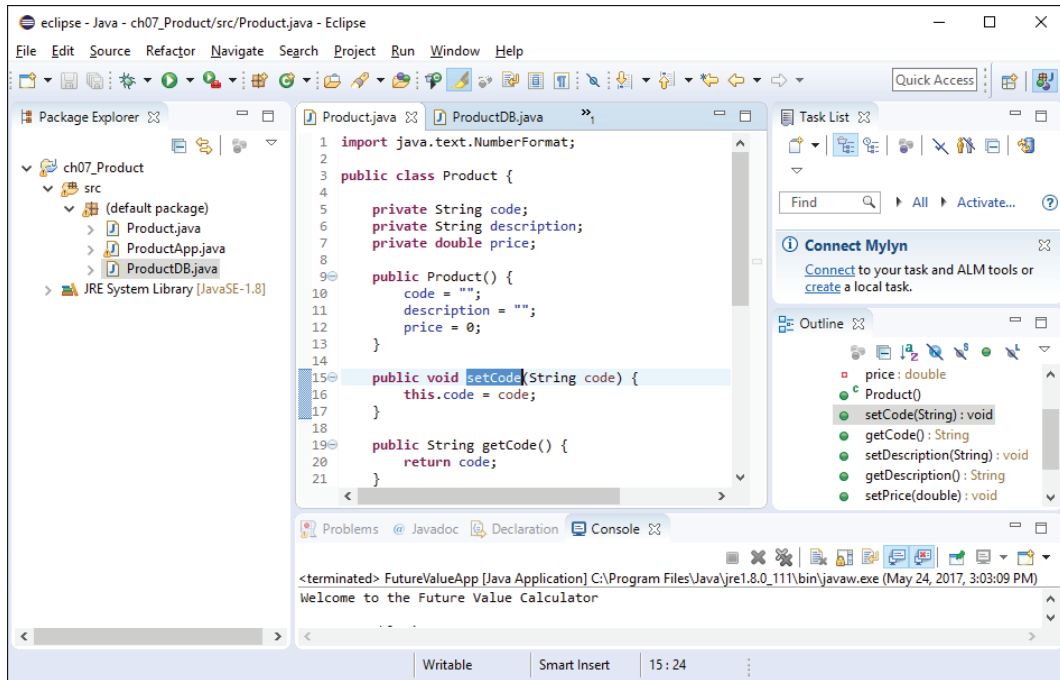
Figure 3-2 shows how Eclipse can make it easier to work with classes. To start, after you enter the private fields for a class, you can use Eclipse to generate the get and set methods for those fields. To do that, you click on the field in the code editor, select Refactor→Encapsulate Field from the menus, and use the Encapsulate Field dialog box shown in this figure to generate the get and set methods.

This dialog box lets you select several options that control how the methods are generated. When you use the Encapsulate Field dialog box, the get and set methods for a field are formatted like this:

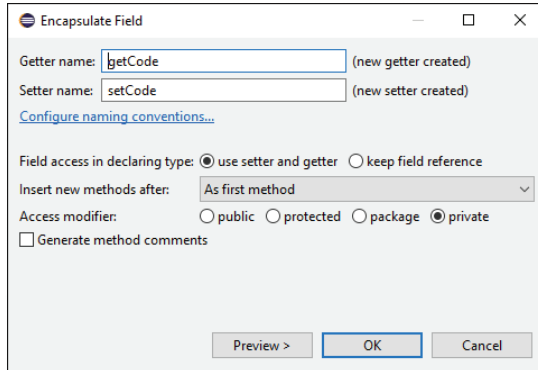
```
public String getCode() {  
    return code;  
}  
  
public void setCode(String code) {  
    this.code = code;  
}
```

Once you've created a class, you can use the Outline window shown in this figure to jump to any member of the class. To do that, just click on the member name. Although the usefulness of this feature isn't obvious for a simple class like the Product class, it can be helpful for classes with more members.

The Eclipse window for the Product application



The dialog box for generating get and set methods



Description

- To generate get and set methods for one or more fields, click on the field and select Refactor→Encapsulate Field and respond to the resulting dialog box.
- The Outline window lists all the members of the currently selected class. To display this window, select Window→Show View→Outline. To jump to a member, click on it in the Outline window.

Figure 3-2 How to work with classes

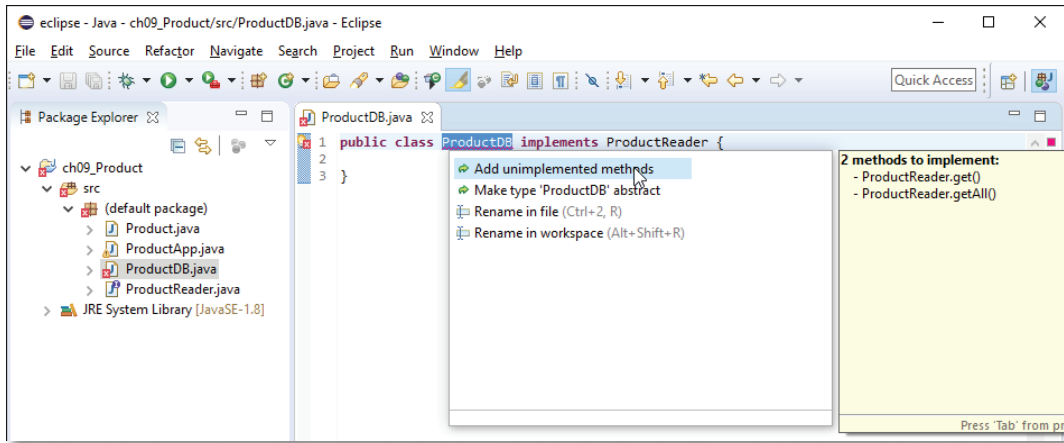
How to work with interfaces

Figure 3-3 shows how Eclipse can make it easier to work with interfaces. To start, you can create an interface using a technique similar to the one you use to create a class. Then, Eclipse generates the declaration for the interface, and you can enter the constants and methods for the interface.

When coding a class that implements an interface, you can automatically generate all the method declarations for the interface. To do that, you click the yellow light bulb icon that appears to the left of the class declaration when you enter the `implements` keyword followed by the name of an interface. This displays a menu that includes the “Add unimplemented methods” command as shown in this figure. Then, you can select this command to generate the method declarations for the interface. In this figure, for example, Eclipse generated the method declarations that implement the `ProductReader` interface. This saves you time and eliminates coding errors.

In the generated code, each method contains a single statement that returns a null value to indicate that the method is not supported yet. At this point, you can write the code that implements the method and returns a non-null value.

A class that implements the ProductReader interface



The code that's generated by Eclipse

```
public class ProductDB implements ProductReader {

    @Override
    public Product get(String code) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getAll() {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Description

- To add an interface to a project, right-click on the package you want to add the interface to, select **New**→**Interface**, and use the resulting dialog box to enter a name for the interface.
- When coding a class that implements an interface, you can automatically generate all the method declarations for the interface. To do that, create a new class as described in figure 3-1, use the **implements** keyword to identify the interface, click on the light bulb error icon in the left margin, and select the “Add unimplemented methods” command.

Figure 3-3 How to work with interfaces

More object-oriented skills

The rest of the figures in this chapter show how to use Eclipse to work with chapter 10 of *Murach's Java Programming (5th Edition)*. This includes the Eclipse skills for working with packages, libraries, modules, and documentation.

How to work with packages

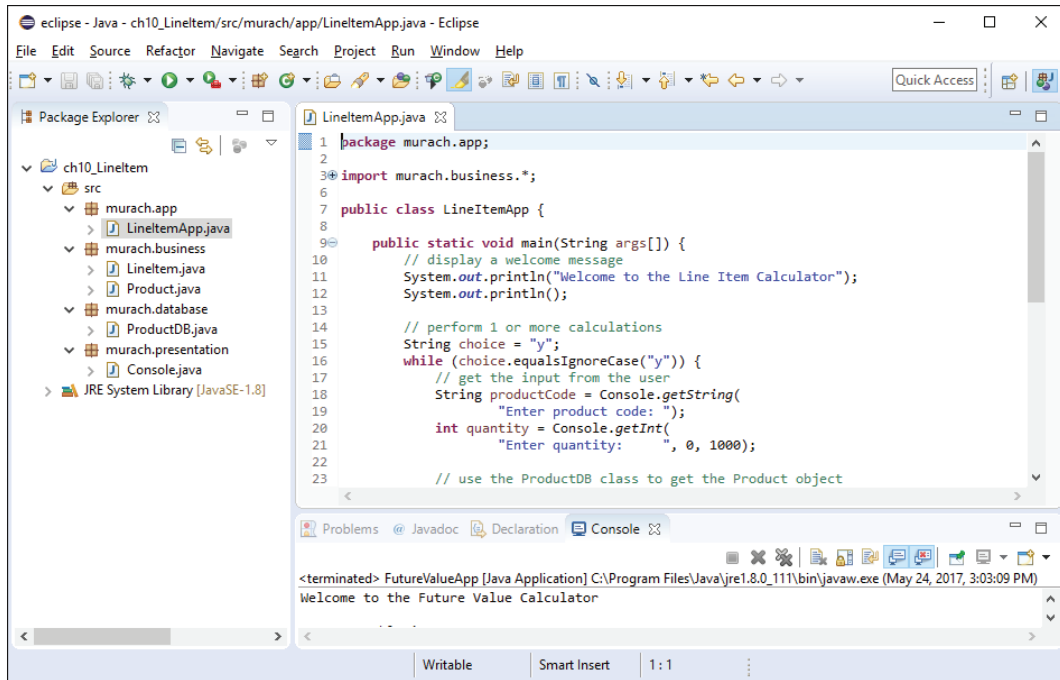
When you work with packages, you need to make sure that the name of the package corresponds with the name of the directory for the package. If you have to do this manually, it can quickly become a tedious task. Fortunately, Eclipse handles this for you automatically.

To navigate through the packages for the project, you can use the Package Explorer. To do that, you can click on the arrows to the left of the packages to expand or collapse them. In figure 3-4, for example, the Package Explorer displays the four packages for the Line Item application that's described in chapter 10 of the book.

To get started with packages, you can add a new package to a project as described in this figure. As you do that, remember that package names correspond to the directories and subdirectories that store the source code for the packages. If these directories and subdirectories don't already exist, they're created when you create the packages.

Once you've created packages for your application, your IDE typically takes care of the details for you automatically. For example, Eclipse automatically adds the necessary package statement to any new class that you add to a package. In addition, if you rename a package, Eclipse automatically renames the corresponding directories and modifies the package statements for all classes in the package. Similarly, if you move a class from one package to another, Eclipse can modify the package statement for that class.

An Eclipse project that contains multiple packages



Description

- To navigate through existing packages, use the Package Explorer window to expand or collapse the packages within a project.
- To add a new package to a project, right-click on the project name or the src folder in the Package Explorer window, select **New**→**Package**, and enter the name of the package in the resulting dialog box. This creates a directory, if necessary, as well as a subdirectory within that directory.
- If you specify a package when you add a new class and that package doesn't already exist, it's automatically created for you.
- To remove a package from a project, right-click on the package and select **Delete** from the resulting menu. This deletes the directory for the package and all subdirectories and files within that directory.
- If you add a new class to a package, Eclipse automatically adds the necessary package statement to the class.
- To rename a package, select **Refactor**→**Rename**. Then, Eclipse automatically modifies the package statement.
- To move a class from one package to another, drag it in the Package Explorer window. Then, click the **Refactor** button in the **Move Class** dialog box that's displayed so Eclipse modifies the package statement.

Figure 3-4 How to work with packages

How to work with libraries

If you want to make the packages of an application available to other applications, you can store them in a *library*. For example, you might want to use the classes in the `murach.business`, `murach.database`, and `murach.presentation` packages from the previous figure with other applications. To do that, you can typically use your IDE to create a library that stores these packages. For example, figure 3-5 shows how to use Eclipse to create this library.

To start, you create a project that contains just the packages and classes that you want to include in the library. One way to do that is to copy an existing project that contains the packages and classes you want and then delete any packages and classes from that project that you don't want to include in the library. To create the `ch10_MurachLib` project shown in this figure, for example, I copied the `ch10_LineItem` project shown in figure 3-4. Then, I renamed that project and deleted the `murach.app` package from it.

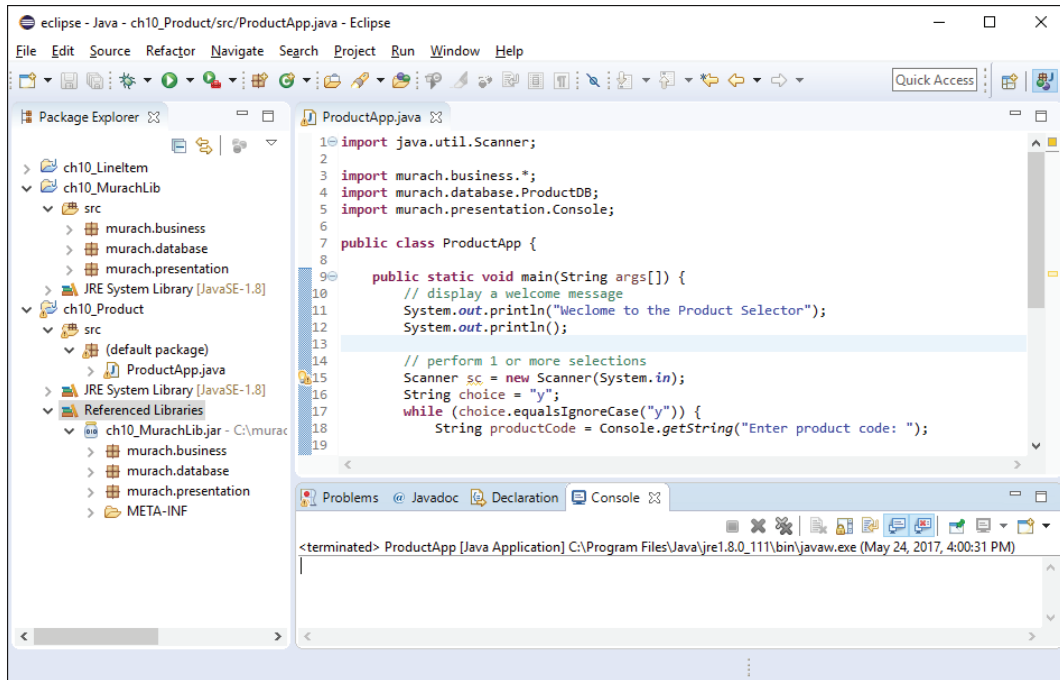
Another way to create a project for a library is to create a new Java Application project without a main class. Then, you can open another project that contains the packages and classes that you want to include in the library. Finally, you can copy those packages from the existing project and paste them into the new project.

After you create the project, you export it to create a *Java Archive (JAR) file* that contains the packages and classes for the library. When you export this file, you can specify its location. In some cases, you may want to copy the JAR file to the project's `src` subdirectory. That way, you can be sure that the file is always moved with the project. Note that the JAR file doesn't include the source code (.java files) by default. Instead, it only includes the .class files, which is usually what you want.

This figure also shows how to use a library after you create it. To do that, you start by creating or opening the project that is going to use the library. Then, you add the JAR file for the library to the project's `Libraries` folder. In this figure, for example, the project named `ch10_Product` uses the library that's stored in the `ch10_MurachLib.jar` file. Finally, you add import statements for these packages to the classes that use them. The `ProductApp` class shown in this figure, for example, imports all the classes from all three packages that are available from the library.

When you create a library that you want other applications to use, you typically store it in a central location. In addition, you typically give the JAR file a name that identifies its contents somewhat. For example, you might give the `ch10_MurachLib.jar` file a name like `murach.jar` to show that it comes from Mike Murach & Associates (www.murach.com).

An Eclipse project that uses a library



How to create a library

1. Create a project that contains just the packages and classes that you want to include in the library.
2. Right-click on the project and select Export→Java→Jar to create the JAR file and specify its name and directory.

How to use a library

1. Create or open the project that is going to use the library.
2. Right-click on the project and select Build Path→Add External Archives. Then, use the resulting dialog box to select the JAR file for the library.
3. Code the import statements for the packages and classes in the library that you want to use. Then, you can use the classes stored in those packages.

Description

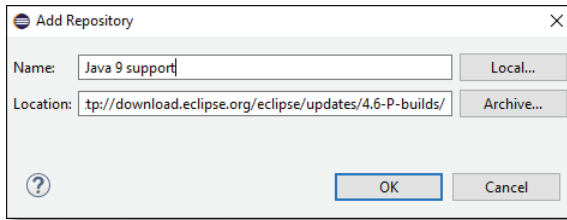
- A *library* can store one or more packages that each contains one or more classes.
- When you create a library, the library is stored in a *Java Archive (JAR) file*.
- After you create a library, you can make it available to other programmers by storing it in a central location.

Figure 3-5 How to work with libraries

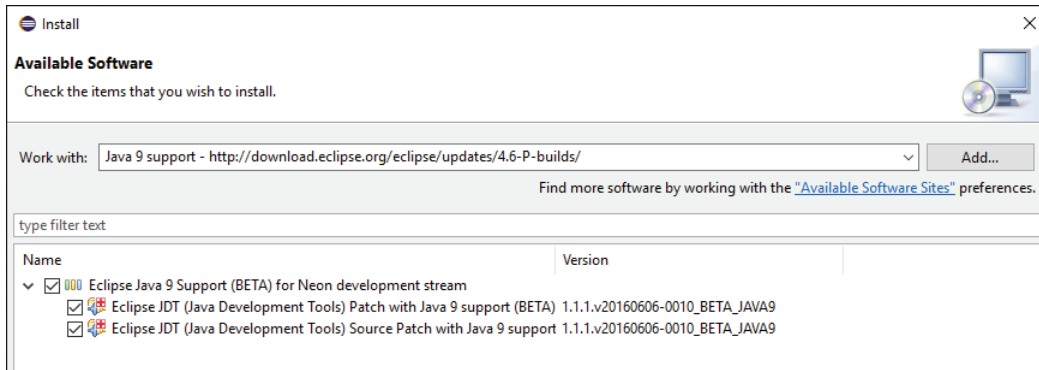
How to install the plug-in for Java 9 support

If you installed Java 9 and the latest version of Eclipse, Eclipse may already support Java 9 features such as modules. However, if you're using an older version of Eclipse, you may need to install a plug-in for Java 9 support before you can work with modules. To do that, you can follow the procedure shown in figure 3-6. Then, you should be able to work with modules as shown in the next figure, as well as other Java 9 features.

The dialog box for adding Java 9 support to Eclipse



The dialog box for installing Java 9 support



How to install Java 9 support

1. Start Eclipse.
2. From the menus, select Help→Install New Software.
3. From the Install dialog box, click the Add button and use the resulting dialog box to add a site for Java 9 support with this URL:
<http://download.eclipse.org/eclipse/updates/4.6-P-builds/>
4. From the Install dialog box, select the option for installing the plug-in.
5. Click the Next button to continue, and accept the defaults in the following dialog boxes.
6. When you're done installing the plug-in, restart Eclipse.

Description

- To get Eclipse to work with Java 9, you can install the plug-in for Java 9 support. This plug-in contains all of the tools needed to use Eclipse to work with the new Java 9 features.

Figure 3-6 How to install the plug-in for Java 9 support

How to create modules

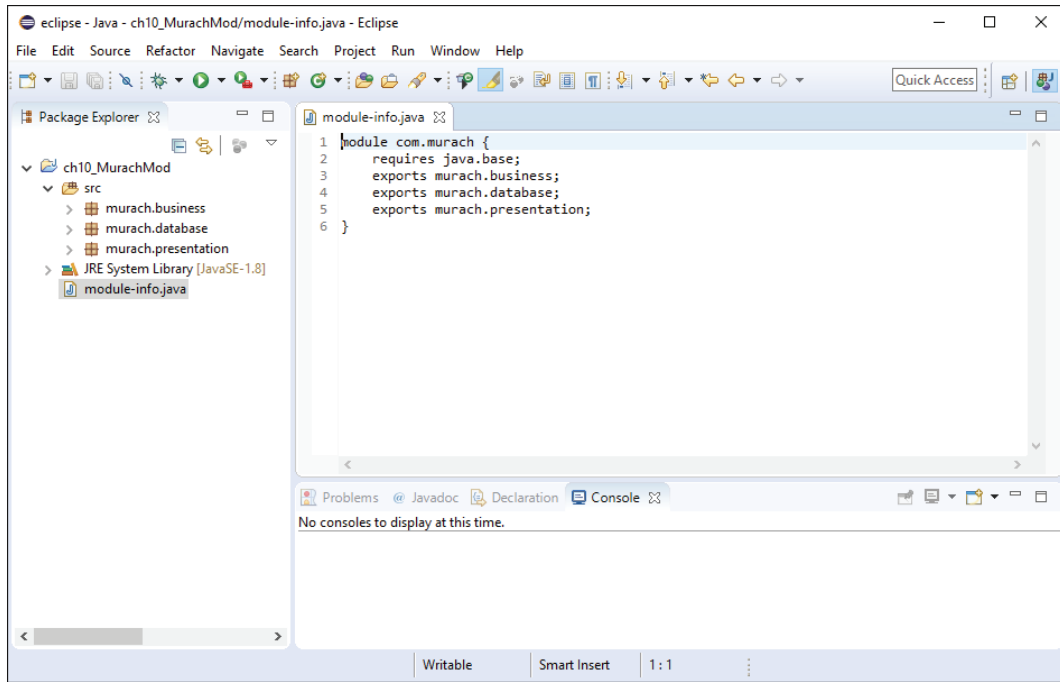
Figure 3-7 shows the library you saw earlier in this chapter after it has been converted to a module. Here, the name of the project has been changed to `ch10_MurachMod` to indicate that the project is for a module, not a library. In addition, a `module-info` file has been added to the root directory as described in this figure.

The `module-info` file shown in this figure accomplishes three tasks. First, it specifies a unique name of `com.murach` for the module. To make sure that it's unique, this name uses the standard convention of reversing the domain name for the company's website address. In this case, the code reverses `murach.com`. Of course, you could easily provide an even more unique name by appending more specifiers such as `com.murach.lineitem`.

Second, this `module-info` file specifies all dependencies for the module. To do that, it uses a `requires` statement to specify that the module requires the `java.base` module. Note that because the `java.base` module is automatically available to all other modules, it isn't necessary to code this statement. In the next figure, though, you'll see some examples of when this statement is required.

Third, the `module-info` file specifies all packages that the module exports. To do that, it uses three `exports` statements to identify three packages. If you wanted to split this module into three smaller modules, you could create one module for each package.

An Eclipse project that defines a module



A module-info file for a module named com.murach

```
module com.murach {  
    requires java.base;  
    exports murach.business;  
    exports murach.database;  
    exports murach.presentation;  
}
```

Description

- A module must contain a module-info.java file in its root (default) directory.
- With Eclipse, you can add a module-info file by right-clicking on the project name and selecting New→Other→General→File. Then, enter module-info.java for the name of the file.
- The module-info file should specify a unique name for the module immediately after the module keyword. This prevents naming conflicts with other modules.
- The module-info file must specify all other modules that the module requires. To do that, it can include one or more requires statements.
- The module-info file must specify all packages that the module exports. To do that, it can include one or more exports statements.

Figure 3-7 How to create modules

How to use modules

If you don't add a module-info file to a project, Java searches the classpath for the required libraries and loads them. This is how Java worked prior to JDK 9. However, if you add a module-info file to a project, the project uses the new module system to load the required modules. As mentioned earlier, this has many potential benefits, including improved security and performance.

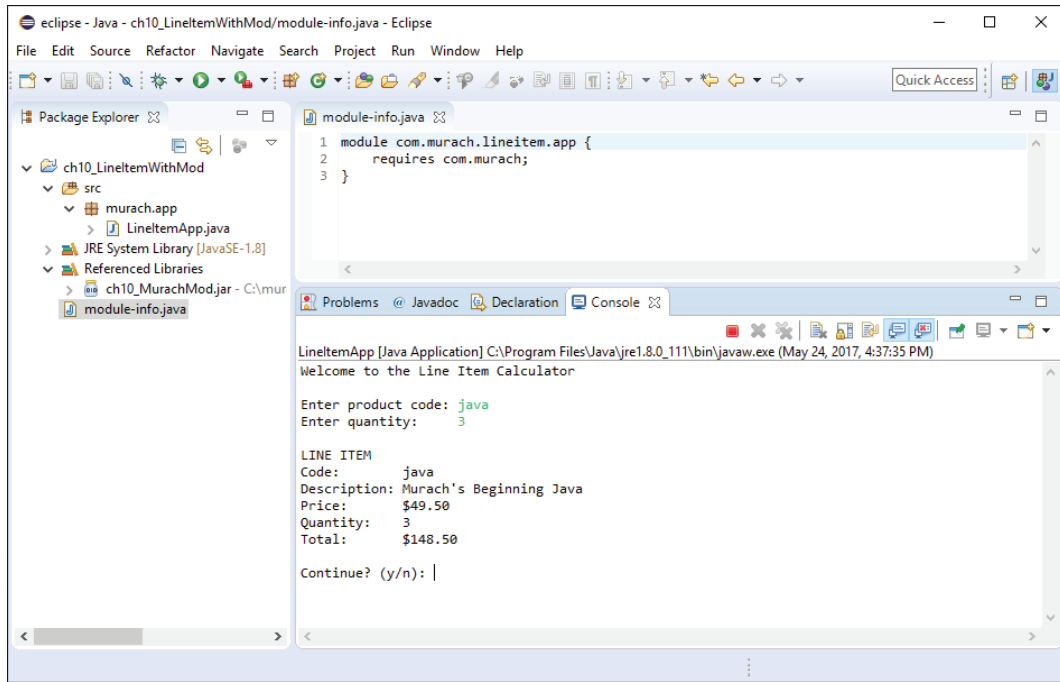
Figure 3-8 shows a project that uses the module created in the previous figure. To start, the JAR file for the module has been added to the project. To do that, you can use the same technique you use to add a JAR file for a library.

In addition, this project includes a module-info file. This file specifies a unique name for the project. More importantly, it specifies that this project requires the `com.murach` module that was created in the previous figure. It also requires the `java.base` module, which is automatically available.

Finally, the `LineItemApp` file is stored in the `murach.app` directory. This is necessary because the module system doesn't allow you to store any classes in the default package. As a result, once you add a module-info file to a project, you need to move any files that are in the default package to a different package.

The last example in this figure shows a module-info file that would be appropriate for an application that uses a SQLite database such the Product Manager application that you'll see in chapter 21 of the book. Here, the first statement specifies that the project requires the `java.sql` module. Then, the second statement specifies that the project requires the `sqlite.jdbc` module that's available from the library that's stored in the `sqlite-jdbc` JAR file. This library and the `java.sql` package are described in chapter 21 of the book.

An Eclipse project that uses a module



A module-info file that requires the com.murach module

```
module com.murach.lineitem.app {
    requires com.murach;
}
```

A module-info file that requires the java.sql and sqlite.jdbc modules

```
module com.murach.product.app {
    requires java.sql;
    requires sqlite.jdbc;
}
```

Description

- If you don't add a module-info file to a project, Java searches the classpath for the required libraries and loads them. This is how Java worked prior to JDK 9.
- If you add a module-info file to a project, the project uses the module system to load the required modules.
- The module-info file must specify the names of all modules that the project requires, except for the java.base module, which is automatically available to all applications.

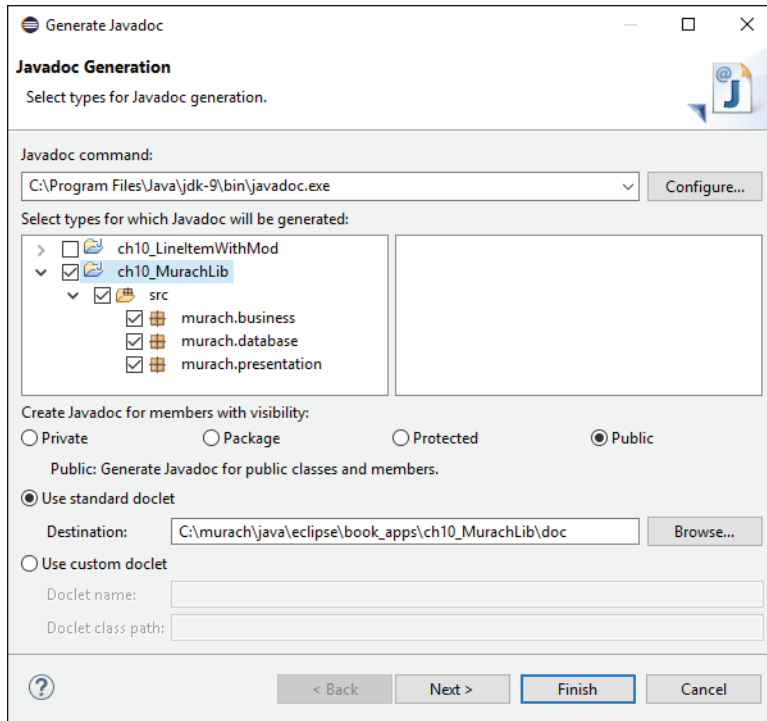
Figure 3-8 How to use modules

How to generate documentation

After you add javadoc comments to the classes of a project, you can use Eclipse to generate the documentation for those classes as shown in figure 3-9. The first time you use Eclipse to generate documentation, you must specify the location of the javadoc command on your system. In this figure, for example, the javadoc command for JDK 9 on a Windows system is specified, but you can use a similar technique to specify the javadoc command on other operating systems.

By default, Eclipse stores the documentation for a project in the doc subdirectory of the project's root directory. This documentation consists of a number of files and subdirectories. If you generate documentation for a project and the doc subdirectory already contains documentation, Eclipse overwrites the old files with the new ones, which is usually what you want.

The dialog box for generating documentation



Description

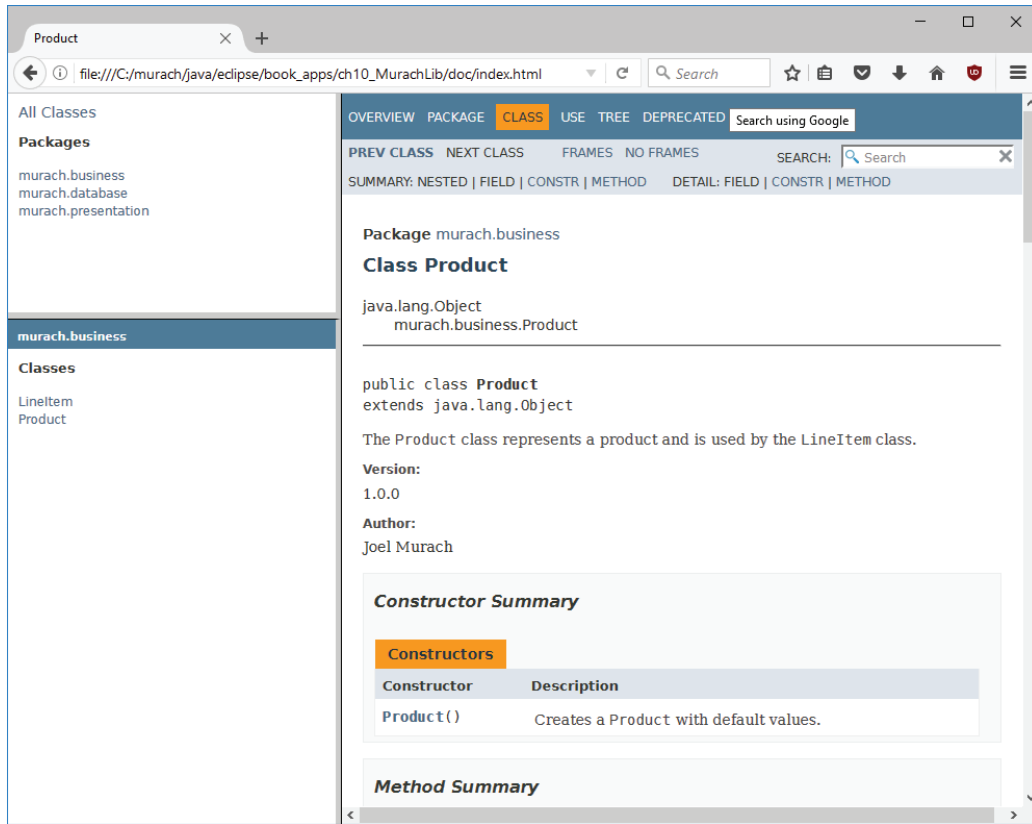
- With Eclipse, you can generate the documentation for a project by selecting Project→Generate Javadoc. Then, you can specify the javadoc command for your system as well as the packages you want to generate documentation for as shown above.
- By default, Eclipse stores the documentation for a project in a subdirectory named doc that's subordinate to the project's root directory.
- If the project already contains documentation, Eclipse overwrites existing files without any warning.

Figure 3-9 How to generate documentation

How to view documentation

After you generate the documentation for a project, you can view it as shown in figure 3-10. The easiest way to do that is to use the File Explorer (Windows) or Finder (Mac) to navigate to the documentation directory, which is usually `doc`. Then, you can double-click on the `index.html` file to display the documentation in your default browser.

The API documentation that's generated for the Product class



Description

- To view the generated documentation, use the File Explorer (Windows) or Finder (Mac) to navigate to the documentation directory, which is usually named `doc`. Then, double-click on the `index.html` file.

Figure 3-10 How to view documentation

Perspective

In this chapter, you learned how to use features of Eclipse for developing object-oriented applications. That includes creating and working with classes and interfaces, working with packages and libraries, creating and using modules, and generating and viewing documentation. These features can make it easier to develop object-oriented applications like the ones presented in chapters 7 through 10 of *Murach's Java Programming (5th Edition)*.

4

GUI and database programming with Eclipse

This chapter shows how to use Eclipse with **chapters 17 and 21 of *Murach's Java Programming (5th Edition)***. To start, if you want to work with JavaFX as described in chapter 17, you may need to install a plug-in and learn how to add the JavaFX JDK library to your project. Similarly, if you want to work with a database as described in chapter 21, you need to learn how to add a JAR file for the database driver to your project.

GUI programming with Eclipse	62
How to install the e(fx)clipse plug-in	62
How to work with JavaFX	64
Database programming with Eclipse	66
How to add a database driver to a project	66
Perspective	68

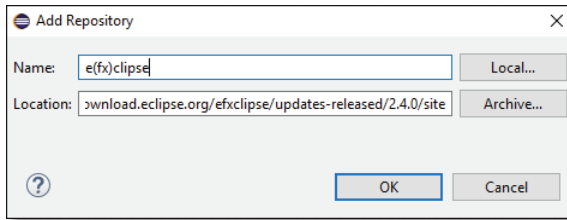
GUI programming with Eclipse

The first two figures in this chapter show how to use Eclipse to work with JavaFX as described in chapter 17 of *Murach's Java Programming (5th Edition)*. This includes installing the plug-in for JavaFX as well as some general skills for working with JavaFX.

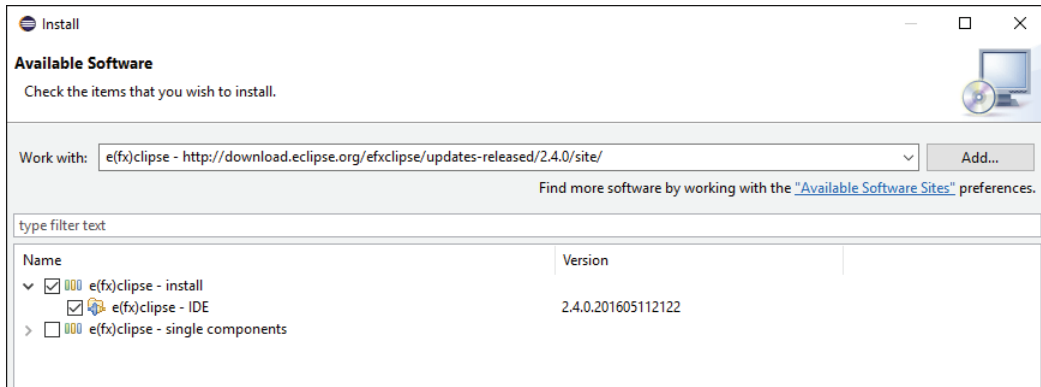
How to install the e(fx)clipse plug-in

Before you can use Eclipse to work with JavaFX, you may need to install a plug-in for JavaFX support. To do that, you can follow the procedure shown in figure 4-1. Then, you should be able to work with JavaFX as shown in the next figure. If you don't, Eclipse may display errors if you open a project that uses JavaFX such as the project shown in the next figure.

The dialog box for adding the e(fx)clipse site



The dialog box for installing the e(fx)clipse plug-in



How to install the e(fx)clipse plug-in

1. Start Eclipse.
2. From the menus, select Help→Install New Software.
3. From the Install dialog box, click the Add button and use the resulting dialog box to add a site for e(fx)clipse with this URL:
<http://download.eclipse.org/efxclipse/updates-released/2.4.0/site>
4. From the Install dialog box, select the option for installing e(fx)clipse.
5. Click the Next button to continue and accept the defaults in the following dialog boxes.
6. When you're done installing the plug-in, restart Eclipse.

Description

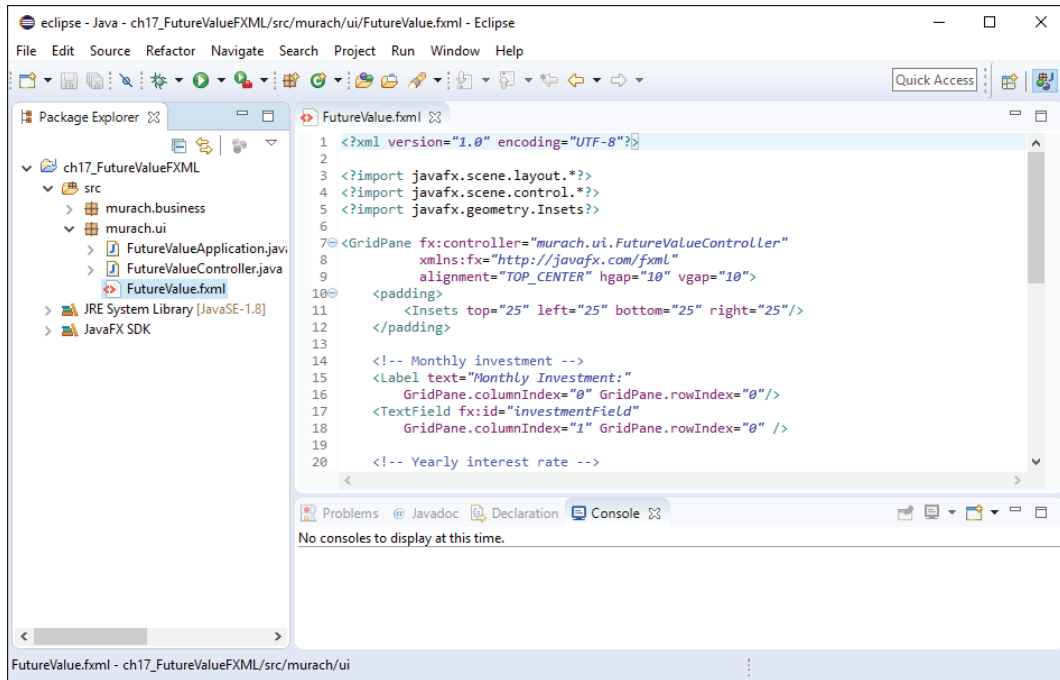
- To get JavaFX to work correctly with Eclipse, you can install the plug-in named e(fx)clipse. This plug-in contains all of the tools and runtime libraries needed to use Eclipse to develop GUIs with JavaFX.

Figure 4-1 How to install the e(fx)clipse plug-in

How to work with JavaFX

Figure 4-2 shows how to use Eclipse to work with JavaFX. To start, if you want to add JavaFX support to an existing project, you can add the JavaFX JDK library to the project as shown in this figure. Then, you can easily create a new JavaFX project. Or, you can add an FXML file to an existing project.

A project with the JavaFX SDK library



Description

- For an Eclipse project to work with JavaFX, you may need to add the JavaFX JDK library to the project. To do that, right-click on the project name and select Build Path→Add Libraries→JavaFX JDK.
- To create a JavaFX project, you can select New→Project→JavaFX→ JavaFX Project from the menu.
- To create an FXML file, you can right-click on a package and select New→Other→JavaFX→New FXML Document.

Figure 4-2 How to use Eclipse to work with JavaFX

Database programming with Eclipse

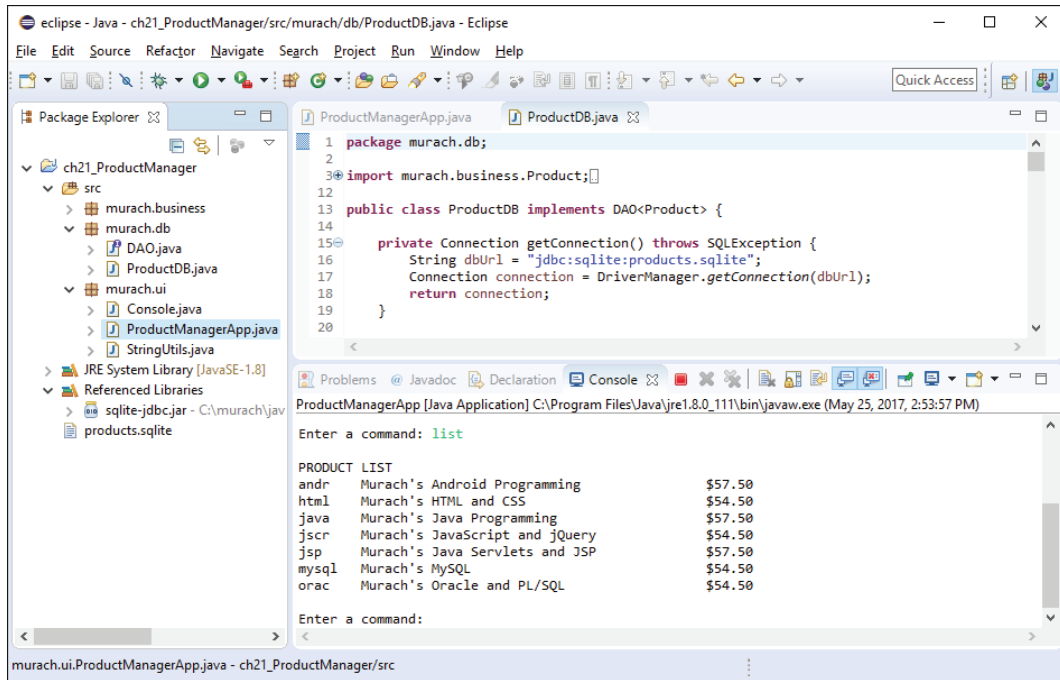
The last figure in this chapter shows how to use Eclipse to add a database driver to a project as described in chapter 21 of *Murach's Java Programming (5th Edition)*.

How to add a database driver to a project

To use Java to work with a database, you have to add the database driver to the project. In most cases, a database driver is stored in a JAR file. As a result, you can typically add a database driver to a project by adding the JAR file that contains the driver. Figure 4-3 shows how to do that with Eclipse.

Here, a JAR file named `sqlite-jdbc.jar` has been added to the project. This file contains all the classes necessary to work with a SQLite database. Once you add this JAR file to a project, any classes in the project can use the database driver to work with the database.

A project after a JDBC database driver for SQLite has been added



Description

- A database driver is a class that's typically stored in a JAR file.
- To add a database driver to an Eclipse project, right-click on the project name, select Build Path→Add External Archives, and use the resulting dialog box to select the JAR file that contains the driver.
- After you add the JAR file that contains a database driver to a project, the JRE can find and run the class for the driver.
- To remove a database driver from a project, right-click on the project name, select Properties, select the Java Build Path category, select the Libraries tab, select the JAR file that contains the driver, and select the Remove button.

Figure 4-3 How to add a database driver to a project

Perspective

In this chapter, you learned how to install the plug-in that's required to use Eclipse to work with JavaFX. You also learned some basic skills for using Eclipse to work with JavaFX. And, you learned how to add a database driver to an Eclipse project so the application can work with a database. You can use these skills as you read chapters 17 and 21 of *Murach's Java Programming (5th Edition)*.