Chapter 3

Running Standard Java Programs

In This Chapter

- Compiling and running a program
- Working with a workspace
- Editing your own Java code

If you're a programming newbie, running a program probably means, for you, clicking the mouse. You want to run Internet Explorer, so you double-click the Internet Explorer icon. That's all there is to it. As far as you're concerned, Internet Explorer is a black box. How the program does whatever it does is none of your concern.

But when you create your own program, the situation is a bit different. You start with no icon to click, and possibly no well-defined notion of what the program should (and should not) do.

So how do you create a brand-new Java program? Where do you click? How do you save your work? How do you get the program to run? What do you do if, at first, the program doesn't run correctly?

This chapter tells you what you need to know.

The example in this chapter is a *standard Oracle* Java program. A standard Oracle Java program runs only on a desktop or laptop computer. The example cannot run on an Android device. For an example that runs on Android devices, see Chapter 4.

Running a Canned Java Program

The best way to get to know Java is to "do Java," by writing, testing, and running your own Java programs. This section prepares you by describing how to run and test a program. Rather than write your own program, you run one that I've already written for you. The program calculates the monthly payments on a home mortgage loan, as shown in Figure 3-1.

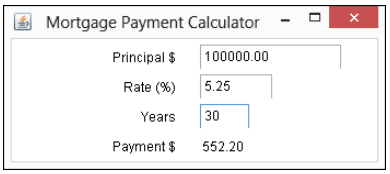


Figure 3-1: A run of the mortgage program in this chapter.

Here's how to run the mortgage program:

1. **First, follow the instructions in Chapter 2 for installing Java, installing and configuring Eclipse, and downloading this book's sample programs.**

   Thank goodness! You don't have to follow those instructions more than once.

2. **Launch Eclipse.**

   The Workspace Launcher dialog box in Eclipse appears, as shown in Figure 3-2.

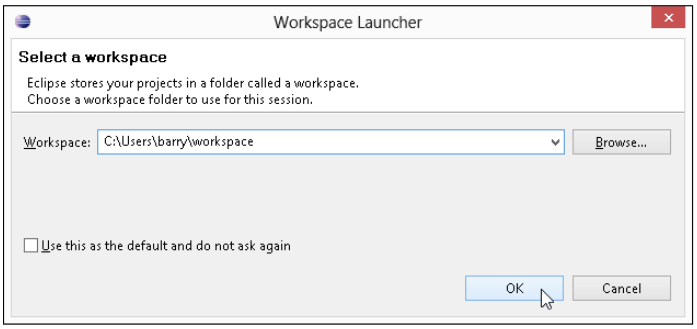


Figure 3-2: The Workspace Launcher in Eclipse.

For a complete how-to on launching Eclipse, see Chapter 2.

A *workspace* is a folder on the computer's hard drive. Eclipse stores Java programs in one or more workspace folders. Along with these Java programs, each workspace folder contains some Eclipse settings. These settings store information such as the version of Java that you're using, the colors you prefer for words in the editor, the size of the editor area when you drag the area's edges, and other preferences. You can have several workspaces with different programs and different settings in each workspace.

By default, the Workspace Launcher offers to open whatever workspace you opened the last time you ran Eclipse. In this example, you open the workspace that you use in Chapter 2, so don't modify anything in the Workspace field.

3. **In the Workspace Launcher dialog box, click OK.**

   The big Eclipse workbench stares at you from the computer screen, as shown in Figure 3-3.

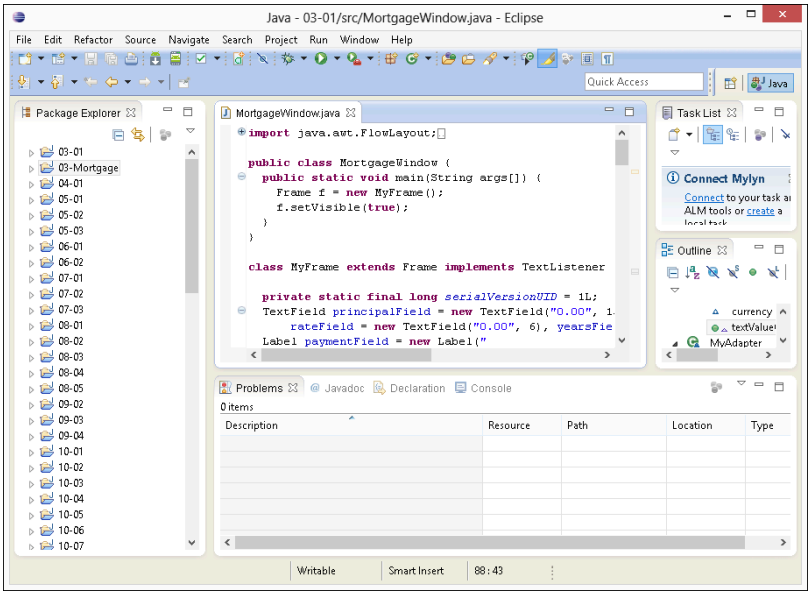


Figure 3-3: The Eclipse workbench.

In Figure 3-3, the leftmost part of the workbench is the Eclipse Package Explorer, which contains numbers such as 03-01, 04-01, and so on. Each number is the name of an Eclipse *project*, which is, formally, a collection of files and folders inside a workspace. Intuitively, a project is a basic work unit. For example, a self-contained collection of Java program files to manage a CD collection (along with the files containing the data) may constitute a single Eclipse project.

Looking again at the Package Explorer in Figure 3-3, you see projects named `03-01`, `04-01`, and so on. My project `03-01` holds the code in Listing 3-1. Project `04-01` contains the Android app whose code begins in Listing 4-1 (the first code listing in Chapter 4 of this book). Project 05-03 contains the code in Listing 5-3. The project named `03-Mortgage` is a slight anomaly because the code for this chapter's Mortgage example isn't in any of the listings.

Eclipse project names can include letters, digits, blank spaces, and other characters; for the names of this book's examples, I stick with digits and dashes.

To read more about topics such as the Eclipse Package Explorer, see the later section "What's All That Stuff in the Eclipse Window?"

When you launch Eclipse, you may see different elements than the ones shown in Figure 3-3. You may see the Eclipse Welcome screen with only a few icons in an otherwise barren window. You may also see a workbench like the one shown in Figure 3-3, but with no list of numbers (`03-01`, `04-01`, and so on) in the Package Explorer. If so, you may have missed some instructions in Chapter 2 for configuring Eclipse. Alternatively, you may have modified the workspace name in the Eclipse Workspace Launcher dialog box.

In any case, make sure that you see numbers like `03-01` and `04-01` in the Package Explorer. Seeing these numbers ensures that Eclipse is ready to run the sample programs from this book.

4. **In the Package Explorer, click the 03-Mortgage branch.**

   As a result, the 03-Mortgage project appears highlighted.

To see a sneak preview of the Java program you're running in Project 03-Mortgage, expand the `03-Mortgage` branch in the Package Explorer. Inside the `03-Mortgage` branch, you find the `src` branch, which in turn contains a `(default package)` branch. Inside the `(default package)` branch, you find the `MortgageWindow.java` branch. This `MortgageWindow.java` branch represents my Java program. Double-clicking the `MortgageWindow.java` branch makes my code appear in the Eclipse editor, as shown in Figure 3-4.

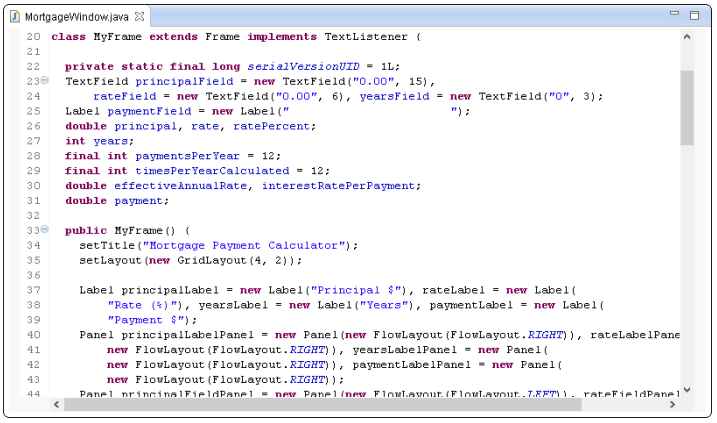


Figure 3-4: Java code in the Eclipse editor.

5. **Choose Run➪Run As➪Java Application from the main menu, as shown in Figure 3-5.**

   When you choose Run As➪Java Application, the computer runs the project's code. (In this example, the computer runs a Java program that I wrote.) The program displays the Mortgage Payment Calculator window on the screen, as shown in Figure 3-6.

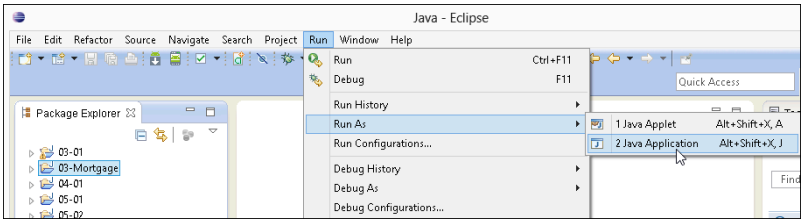


Figure 3-5: One way to run the code in Project 03-Mortgage.

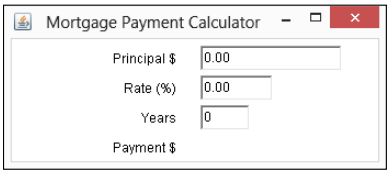


Figure 3-6: The Mortgage Payment Calculator begins its run.

6. **Type numbers into the fields in the Mortgage Payment Calculator window. (Refer to Figure 3-1.)**

When you type a principal amount in Step 6, don't include the country's currency symbol and don't group the digits. (U.S. residents: Omit dollar signs and commas.) For the percentage rate, omit the `%` symbol. For the number of years, don't use a decimal point. If you break any of these rules, the Java code can't read your number, and my Java program displays nothing in the *Payment* row.

Disclaimer: Your local mortgage company charges more (a lot more) than the amount that my Java program calculates.

If you follow this section's instructions and you don't see the results I describe, you can try these three strategies, listed in order from best to worst:

- Double-check all steps to make sure that you followed them correctly.
- Contact me at `Java4Android@allmycode.com` via e-mail, `@allmycode` on Twitter, or `/allmycode` on Facebook If you describe what happened, I can probably figure out what went wrong and tell you how to correct the problem.
- Panic.

Typing and Running Your Own Code

The earlier section "Running a Canned Java Program" is all about running someone else's Java code (code that you download from this book's website). But, eventually, you'll write code on your own. This section shows you how to create code by using the Eclipse IDE.

Separating your programs from mine

You can separate your code from this book's examples by creating a separate workspace. Here are two (distinct) ways to do it:

- **When you launch Eclipse, type a new folder name in the Workspace field of the Workspace Launcher dialog box in Eclipse.**

  If the folder doesn't already exist, Eclipse creates the folder. If the folder already exists, the Eclipse Package Explorer lists any projects that the folder contains.

- **In the main menu in the Eclipse workbench, choose File➪Switch Workspace, as shown in Figure 3-7.**

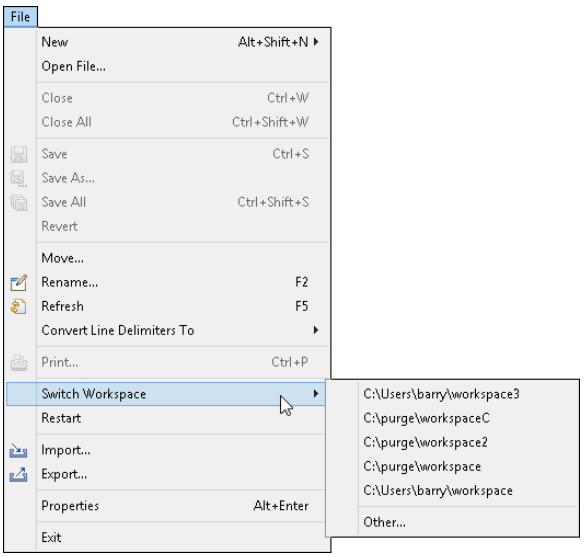


Figure 3-7: Switching to a different Eclipse workspace.

When you choose File➪Switch Workspace, Eclipse offers you a few of your previously opened workspace folders. If your choice of folder isn't in the list, select the Other option. In response, Eclipse reopens its Workspace Launcher dialog box.

Writing and running your program

Here's how to create a new Java project:

1. **Launch Eclipse.**
2. **From the Eclipse menu bar, choose File➪New➪Java Project.**

   The Create a Java Project dialog box appears.
3. **In the Create a Java Project dialog box, type a name for the project and then click Finish.**

   In Figure 3-8, I type the name `MyFirstProject`.

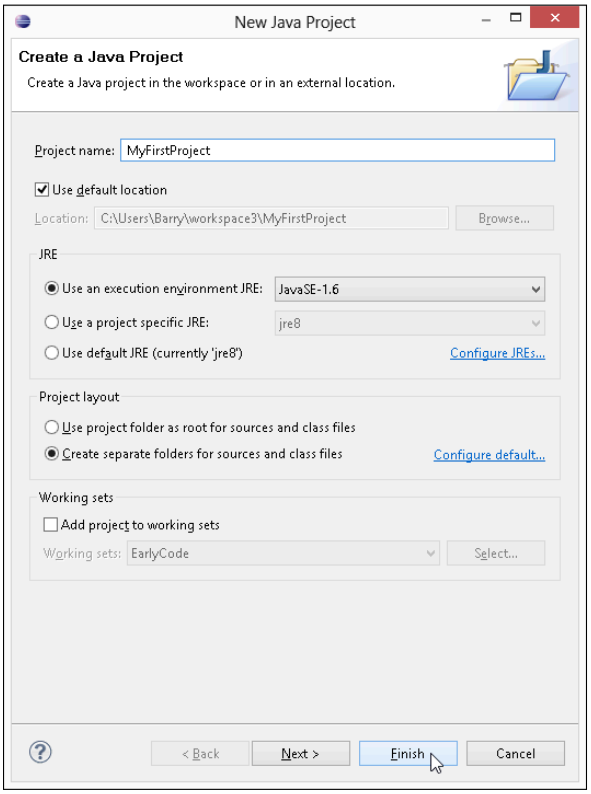


Figure 3-8: Getting Eclipse to create a new project.

If you click Next instead of Finish, you see other options that you don't need right now. To avoid confusion, just click Finish.

Clicking Finish returns you to the Eclipse workbench, with `MyFirst Project` in the Package Explorer, as shown in Figure 3-9.

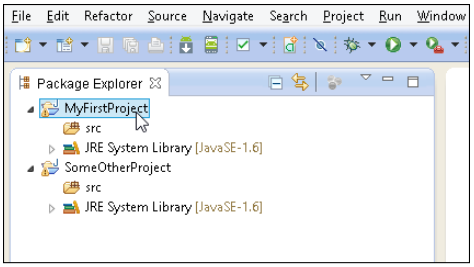


Figure 3-9: Your project appears in the Package Explorer in Eclipse.

The next step is to create a new Java source code file.

4. **Select the newly created project in the Package Explorer.**

   To create Figure 3-9, I selected `MyFirstProject` instead of `SomeOtherProject`.

5. **In the Eclipse main menu, choose File➪New➪Class.**

   The Eclipse Java Class dialog box appears, as shown in Figure 3-10.

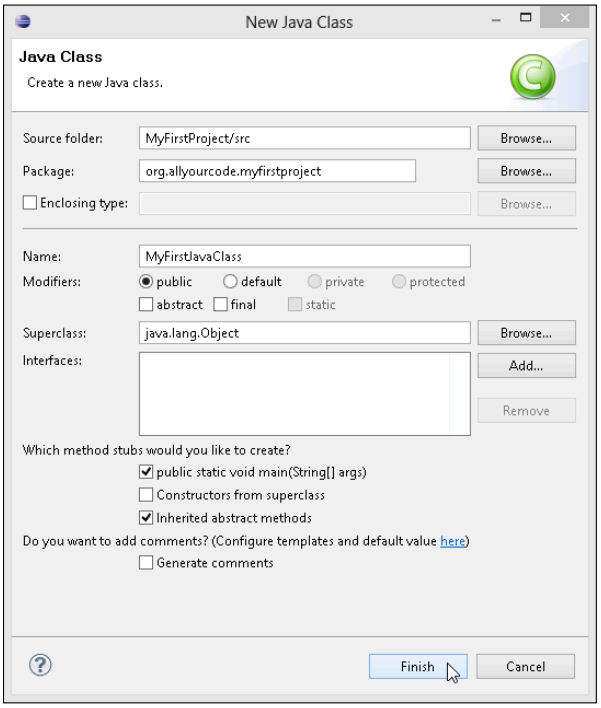


Figure 3-10: Getting Eclipse to create a new Java class.

Like every other windowed environment, Eclipse provides many ways to accomplish the same task. Rather than choose File➪New➪Class, you can right-click `MyFirstProject` in the Package Explorer in Windows (or control-click `MyFirstProject` in the Package Explorer on a Mac). In the resulting context menu, choose New➪Class. You can also start by pressing Alt-Shift+N in Windows (or Option-Command-N on a Mac). The choice of clicks and keystrokes is up to you.

6. **In the Name field in the Java Class dialog box, type the name of the new class.**

   In this example, I use the name `MyFirstJavaClass`, with no blank spaces between the words in the name. (Refer to Figure 3-10.)

The name in the Java Class dialog box cannot have blank spaces, and the only allowable punctuation symbol is the underscore character (_). You can name the class `MyFirstJavaClass` or `My_First_Java_Class`, but you can't name it `My First Java Class`, and you can't name it `JavaClass,MyFirst`. Finally, you can't start a class name with a digit. For example, you can name the class `Go4It` but not `2bOrNot2b`.

7. **In the Package field in the Java Class dialog box, type a package name. (Refer to Figure 3-10.)**

   In Java, you group code into bunches called *packages*. And in the Android world, each app comes in its own package.

   Don't worry much about making up package names. If you have your own domain name (`allyourcode.org`, for example), you should reverse the domain name (resulting in `org.allyourcode`) and then add a descriptive word. For example, `org.allyourcode.myfirst project` is a good package name. If you don't have a domain name, any words (separated from one another by dots) will work.

The package name contains one or more words. Each word can be any combination of letters, digits, and underscores (_) as long as the word doesn't start with a digit. A package name is a bunch of these words, separated from one another by dots. For example, `org.allyourcode.Go4It` is a valid package name, but `org.allyourcode. 2bOrNot2b` is not. (You can't start the third part of the package name with the digit `2`. For that matter, you can't start any of the three words in a name like `org.allyourcode.myfirstproject` with a digit.)

8. **Put a check mark in the `public static void main(String[] args)` check box.**

   The check mark tells Eclipse to create some boilerplate Java code.

9. **Accept the defaults for everything else in the Java Class dialog box. (In other words, click Finish.)**

Clicking Finish brings you back to the Eclipse workbench. Now `MyFirstProject` contains a file named `MyFirstJavaClass.java`. For your convenience, the `MyFirstJavaClass.java` file already has some code in it. The Eclipse editor displays the Java code, as shown in Figure 3-11.

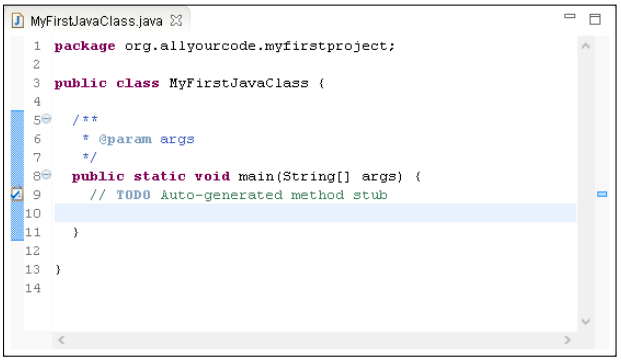


Figure 3-11: Eclipse writes some code in the editor.

10. Replace an existing line of code in the new Java program.

Type a line of code in the Eclipse editor. Replace the line

```
// TODO Auto-generated method stub
```

with these lines:

```
javax.swing.JOptionPane.showMessageDialog
                                    (null, "Hello");
```

Any program containing these lines of code runs only on a desktop (or laptop) computer. The code `javax.swing.JOptionPane.showMessageDialog` belongs to standard Oracle Java, but not to Android Java.

Copy the new lines of code exactly as you see them in Listing 3-1.

- Spell each word exactly the way I spell it in Listing 3-1.
- Capitalize each word exactly the way I do in Listing 3-1.
- Include all the punctuation symbols — the dots, the quotation marks, the semicolon — everything.

When you're done, the code in the Eclipse editor should look exactly like the code in Listing 3-1.

Do I see formatting in my Java program?

When you use the Eclipse editor to write a Java program, you see words in various colors. Certain words are always in blue. Other words are always in black. You even see some bold and italic phrases. You may think you see formatting, but you don't. Instead, what you see is *syntax coloring* or *syntax highlighting*.

No matter what you call it, the issue is this:

- In Microsoft Word, elements such as bold formatting are marked inside a document. When you save `MyPersonalDiary.doc`, the instructions to make the words *love* and *hate* bold are recorded inside the `MyPersonalDiary.doc` file.
- In a Java program editor, elements such as bold and coloring aren't marked inside the Java program file. Instead, the editor displays each word in a way that makes the Java program easy to read.

For example, in a Java program, certain words (such as `class`, `public`, and `void`) have their own, special meanings. So the Eclipse editor displays `class`, `public`, and `void` in bold, reddish letters. When I save my Java program file, the computer stores nothing about bold, colored letters in my Java program file. But the editor uses its discretion to highlight special words with reddish coloring.

Another editor may display the same words in a blue font. Another editor (such as Windows Notepad) displays all words in plain, old black.

Listing 3-1: A Program to Display a Greeting

```
public class MyFirstJavaClass {

  /**
   * @param args
   */
  public static void main(String[] args) {
    javax.swing.JOptionPane.showMessageDialog
                                    (null, "Hello");
  }

}
```

Java is *case-sensitive*, which means that `Showmessagedialog` isn't the same as `showMessageDialog`. If yOu tyPe `Showmessagedialog`, your progrAm won't worK. Be sUre to cAPItalize your codE eXactLy as it is shown in Listing 3-1.

Some people notice the difference between "curly" quotation marks and "straight" quotation marks. Is the distinction between the two types useful? (Do you see the difference?) Is it even appropriate to use the words *curly* and *straight* for the two kinds of quotation marks? In a Java program, a word like `"Hello"` (surrounded by straight quotation marks) stands for a string of characters. In fact, the code in Listing 3-1 makes the letters `Hello` appear on the user's screen. Here's the rule:

In Java, to denote a string of characters, always use straight quotation marks; never curly quotation marks.

In practice, if you copy code from a Kindle or from another electronic medium, you're probably copying curly quotation marks, and the code is incorrect. Fortunately, when you use the computer keyboard to type code in the Eclipse editor, you automatically type straight quotation marks. That's nice.

In a Java program, almost none of the spacing and indentation matters. In Listing 3-1, I don't need all the blank spaces before `(null, "Hello")`, but the blank spaces help me to remember that `(null, "Hello")` is a continuation of the `showMessageDialog` stuff. In other words, all the characters between the word `javax` and the word `"Hello"` are part of one big Java command. I separate the command into two lines because if I didn't, the command would run off the edge of the page.

If you type everything correctly, you see the information shown in Figure 3-12.

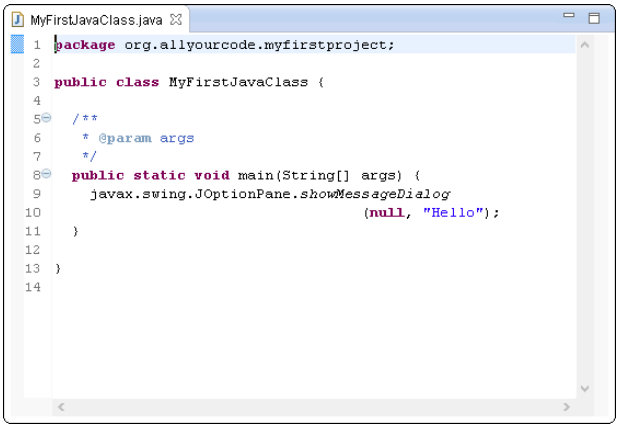


Figure 3-12: A Java program in the Eclipse editor.

If you don't type your part of the code exactly as it's shown in Listing 3-1, you may see jagged red underlines, tiny rectangles with X-like markings inside them, or other red marks in the Editor, as shown in Figure 3-13.

The red marks in the Eclipse editor refer to compile-time errors in the Java code. A *compile-time* error (also known as a *compiler* error) is an error that prevents the computer from translating the code. (See the talk about code translation in Chapter 1.)

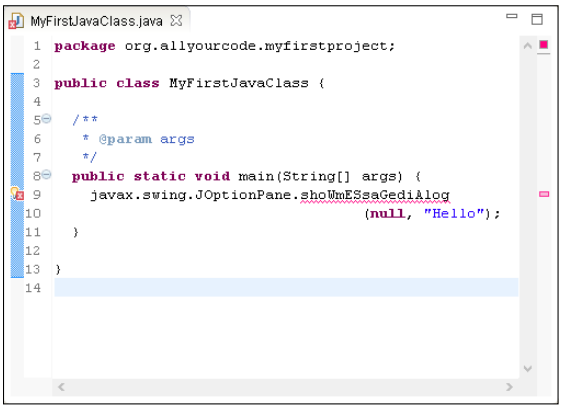


Figure 3-13: A Java program, typed incorrectly.

Here, the error markers in Figure 3-13 appear on line 9 of the Java program. Line numbers are designed to appear in the editor's left margin, but they do not appear by default. To make the Eclipse editor display line numbers, choose Window➪Preferences (in Windows) or Eclipse➪Preferences (on a Mac). Then choose General➪Editors➪Text Editors. Finally, add a check mark in the Show Line Numbers check box.

To fix compile-time errors, you must become a dedicated detective and join the elite squad known as *Law & Order: JPU (Java Programming Unit)*. You seldom find easy answers. Instead, comb the evidence slowly and carefully for clues. Compare everything you see in the editor, character by character, with my code in Listing 3-1. Don't miss a single detail, including spelling, punctuation, and uppercase versus lowercase.

Eclipse has a few nice features to help you find the source of a compile-time error. For example, you can hover over the jagged red underline. When you do, you see a brief explanation of the error along with suggestions for repairing the error — some *quick fixes,* in other words. See Figure 3-14.

In Figure 3-14, a pop-up message tells you that Java doesn't know what the word `shoWmESsaGediAlog` means — that is, `shoWmESsaGediAlog` is "undefined." Near the bottom of the figure, one quick-fix option is to repair the incorrect capitalization by changing `shoWmESsaGediAlog` to `showMessageDialog`.

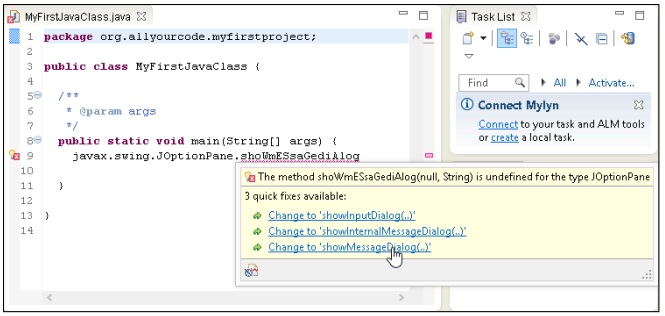


Figure 3-14: Eclipse offers helpful suggestions.

When you click the Change to `'showMessageDialog' (..)` option, the Eclipse editor replaces `shoWmESsaGediAlog` with `showMessage Dialog`. The editor's error markers disappear, and the incorrect code shown in Figure 3-13 changes to the correct code shown in Figure 3-12.

11. **Make any changes or corrections to the code in the Eclipse editor.**

    When at last you see no jagged underlines or blotches in the editor, you're ready to try running the program.

12. **Select `MyFirstJavaClass` either by clicking inside the editor or by clicking the `MyFirstProject` branch in the Package Explorer.**

13. **In the Eclipse main menu, choose Run➪Run As➪Java Application.**

    That does the trick. The new Java program runs, and you see the `Hello` message shown in Figure 3-15. It's like being in heaven!

Figure 3-15: Running the program shown in Listing 3-1.

What can possibly go wrong?

Ridding the editor of jagged underlines is cause for celebration. Eclipse likes the look of your code, so from that point on, it's smooth sailing. Right?

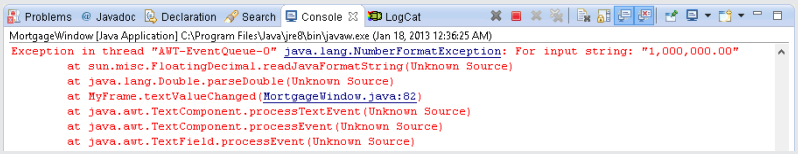
Well, it ain't necessarily so. In addition to some conspicuous compile-time errors, the code can have other, less obvious errors.

Imagine someone telling you to "go to the intersection, and then *rurn tight*." You notice immediately that the speaker has made a mistake, and you respond with a polite "Huh?" The nonsensical *rurn tight* phrase is like a compile-time error. Your "Huh?" is like the jagged underlines in the Eclipse editor. As a

human being who listens, you may be able to guess what *rurn tight* means, but the Eclipse editor never dares to fix the mistakes in your code.

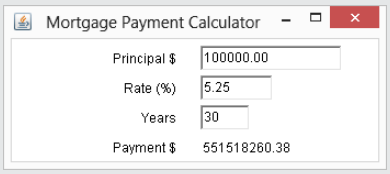
In addition to compile-time errors, other kinds of gremlins can hide inside a Java program:

- **Unchecked runtime exceptions:** You see no compile-time errors, but when you run the program, the run ends prematurely. Somewhere in the middle of the run, the instructions tell Java to do something that can't be done. For example, while you're running the Mortgage program in the earlier section "Running a Canned Java Program," you type `1,000,000.00` instead of `1000000.00`. Java doesn't like the commas in the number, so the program crashes and Eclipse displays a nasty-looking message, as shown in the first figure.



  This example shows an *unchecked runtime exception* — the equivalent of someone telling you to turn right at the intersection when the only thing to the right is a big, brick wall. The Eclipse editor doesn't warn you about an unchecked runtime exception because, until you run the program, the computer can't predict that the exception will occur.

- **Logic errors:** You see no error markers in the Eclipse editor, and when you run the code, the program runs to completion. But the answer isn't correct. Instead of $552.20 in the second figure, the payment amount is $551,518,260.38. The program incorrectly tells you to pay thousands of times what your house is worth and tells you to pay this amount each month! It's the equivalent of being told to turn right rather than turn left. You can drive in the wrong direction for quite a long time.



  Logic errors are the most challenging errors to find and to fix. And worst of all, logic errors often go unnoticed. In March 1985, I got a monthly home heating bill for $1,328,932.21. Clearly, a computer had printed the incorrect amount. When I called the gas company to complain, the telephone service representative said, "Don't be upset. Pay only half that amount."

- **Compile-time warnings:** A warning isn't as severe as an error message. So when Eclipse notices suspicious behavior in a program, the editor displays a jagged yellow underline, an exclamation point enclosed in a tiny yellow icon, and a few other not-so-intrusive clues.

  For example, in the third figure, you can see that, on Line 9, I added material related to `amount = 10` to the code from Listing 3-1. The problem is, I never make use of the `amount` or of the number `10` anywhere in my program. With its faint, yellow markings, Eclipse effectively tells me "Your `amount = 10` code isn't bad enough to be a showstopper. Eclipse can still manage to run the program. But are you sure you want `amount = 10` (this material that seems to serve no purpose) in your program?"

(continued)

(continued)

A subtle hint

Imagine being told, "Turn when you reach the intersection." The direction may be just fine. But if you're suspicious, you ask, "Which way should I turn? Left or right?"

When you're sure that you know what you're doing, you can ignore warnings and worry about them later. But a warning can be an indicator that the code has a more serious problem. My sweeping recommendation is this: Pay attention to warnings. But if you can't figure out why you're seeing a particular warning, don't let the warning prevent you from moving forward.

Icon yellow?

Your code is mellow.

Icon red?

Your code is dead!

What's All That Stuff in the Eclipse Window?

Believe it or not, an editor once rejected one of my book proposals. In the margin, the editor scribbled "This is not a word" next to text such as *can't, it's,* and *I've.* To this day, I still do not know what this editor did not like about contractions. My own opinion is that language always needs to expand. Where would we be without a few new words — words such as *dotcom*, *infomercial*, and *vaporware*?

Even the *Oxford English Dictionary* (the last word in any argument about words) grows by more than 4,000 entries each year. That's an increase of more than 1 percent per year — about 11 new words per day!

The fact is, human thought resembles a high-rise building: You can't build the 50th floor until you've built at least part of the 49th. You can't talk about *spam* until you have a word such as *e-mail*. In these fast-paced, changing times, you need verbal building blocks. That's why this section contains a bunch of new terms.

In this section, each newly defined term describes an aspect of the Eclipse IDE. Before you read all this Eclipse terminology, I provide these disclaimers:

- **This section is optional reading.** Refer to this section if you have trouble understanding some of this book's instructions. But if you have no trouble navigating the Eclipse IDE, don't complicate things by fussing over the terminology in this section.
- **This section provides explanations of terms, not formal definitions of terms.** Yes, my explanations are fairly precise; but no, they're not airtight. Almost every description in this section has hidden exceptions, omissions, exemptions, and exclusions. Take the paragraphs in this section as friendly reminders, not as legal contracts.
- **Eclipse is a useful tool.** But Eclipse isn't officially part of the Java ecosystem. Although I don't describe details in this book, you can write Java programs without ever using Eclipse.

Understanding the big picture

Your tour of Eclipse begins with the big Burd's-eye view:

- **Workbench:** The Eclipse desktop (refer to Figure 3-3). The workbench is the environment in which you develop code.
- **Area:** A section of the workbench. The workbench shown in Figure 3-3 contains five areas. To illustrate the point, I've drawn borders around each area, as shown in Figure 3-16.
- **Window:** A copy of the Eclipse workbench. In Eclipse, you can have several copies of the workbench open at a time. Each copy appears in its own window.

  To open a second window, go to the main Eclipse menu bar and choose Window➪New Window.

- **Action:** A choice that's offered to you, typically when you click something. For example, when you choose File➪New from the Eclipse main menu bar, you see a list of new elements you can create. The list usually includes Project, Folder, File, and Other, but it may also include items such as Package, Class, and Interface. Each of these things (each item on the menu) is an *action*.

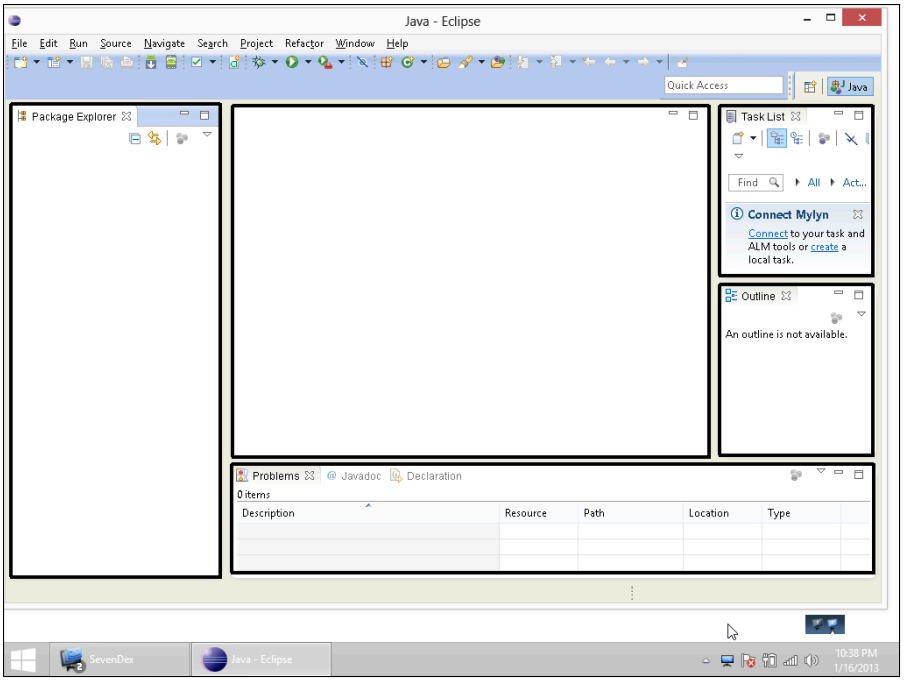


Figure 3-16: The workbench is divided into areas.

Views, editors, and other stuff

The next bunch of terms deals with things called views, editors, and tabs.

You may have difficulty understanding the difference between views and editors. (A *view* is like an *editor*, which is like a *view*, or something like that.) If views and editors seem the same to you, and you're not sure whether you can tell which is which, don't be upset. When you're an ordinary Eclipse user, the distinction between views and editors comes naturally as you gain experience using the workbench. You rarely have to decide whether the thing you're using is a view or an editor.

Anyway, if you ever have to distinguish between a view and an editor, here's what you need to know:

- **View:** A part of the Eclipse workbench that displays information for you to browse. In the simplest case, a view fills up an area in the workbench. For example, in Figure 3-3, earlier in this chapter, the Package Explorer view fills up the leftmost area.

  Many views display information as lists or trees. For example, in Figure 3-9, the Package Explorer view contains a tree.

You can use a view to make changes. For example, to delete `MyFirst Project` in Figure 3-9, right-click the `MyFirstProject` branch in the Package Explorer view. (On a Mac, control-click the `MyFirstProject` branch.) Then on the resulting context menu, choose Delete.

When you use a view to change something, the change takes place immediately. For example, when you choose Delete in the Package Explorer's context menu, whatever item you've selected is deleted immediately. In a way, this behavior is nothing new. The same kind of thing happens when you recycle a file using Windows Explorer or trash a file using the Macintosh Finder.

- **Editor:** A part of the Eclipse workbench that displays information for you to modify. A typical editor displays information in the form of text. This text can be the contents of a file. For example, an editor in Figure 3-11 displays the contents of the `MyFirstJavaClass.java` file.

When you use an editor to change something, the change doesn't take place immediately. For example, look at the editor shown in Figure 3-11. This editor displays the contents of the `MyFirstJavaClass.java` file. You can type all kinds of things in the editor. Nothing happens to `MyFirstJavaClass.java` until you choose File➪Save from the Eclipse menu bar. Of course, this behavior is nothing new. The same kind of thing happens when you work in Microsoft Word or in any other word processing program.

Like other authors, I occasionally become lazy and use the word *view* when I mean *view or editor* instead. I also write "the Eclipse editor" when I should write "an Eclipse editor" or "the Editor area of the Eclipse workbench." When you catch me blurring the terminology this way, just shake your head and move onward. When I'm being careful, I use the official Eclipse terminology. I refer to views and editors as *parts* of the Eclipse workbench. Unfortunately, this "parts" terminology doesn't stick in peoples' minds.

An area of the Eclipse workbench might contain several views or several editors. Most Eclipse users get along fine without giving this "several views" business a second thought (or even a first thought). But if you care about the terminology surrounding tabs and active views, here's the scoop:

- **Tab:** Something that's impossible to describe except by calling it a "tab." That which we call a tab by any other name would move us as well from one view to another or from one editor to another. The important thing is, views can be *stacked* on top of one another. Eclipse displays stacked views as though they're pages in a tabbed notebook. For example, Figure 3-17 displays one area of the Eclipse workbench. The area contains six views (Problems view, Javadoc view, Declaration view, Search view, Console view, and LogCat view). Each view has its own tab.

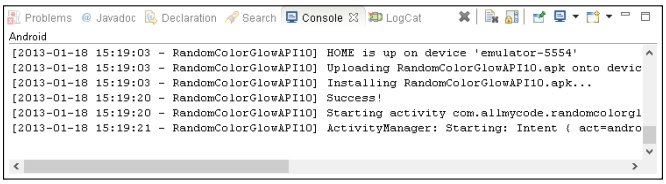


Figure 3-17: An area containing several views.

The Console view is shown in Figure 3-17, but it doesn't always appear as part of the Java perspective. Normally, the Console view appears automatically whenever the program crashes. If you want to force the Console view to appear, choose Window➪Show View➪Other. In the resulting Show View dialog box, expand the General branch. Finally, within that General branch, double-click the Console item.

A bunch of stacked views is a *tab group*. To bring a view in the stack to the forefront, you click that view's tab.

By the way, all this information about tabs and views holds true for tabs and editors. The only interesting thing is the way Eclipse uses the word *editor*. In Eclipse, each tabbed page of the Editor area is an individual editor. For example, the Editor area shown in Figure 3-18 contains three editors (not three tabs belonging to a single editor). The three editors display the contents of three files: `MyFirstJavaClass.java`, `MortgageWindow.java`, and `activity_main.xml`.

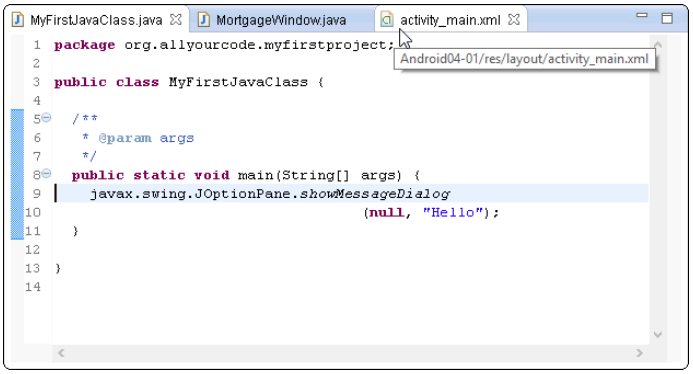


Figure 3-18: The Editor area contains three editors.

- **Active view or active editor:** In a tab group, the active view or editor refers to the view or editor that's in front.

  In Figure 3-18, the `MyFirstJavaClass.java` editor is the active editor. The `MortgageWindow.java` and `activity_main.xml` editors are inactive. (The `activity_main.xml` looks as though it's active, but that's because, in Figure 3-18, I'm hovering the mouse over that editor's tab.)

Looking inside a view or an editor

The terms in this section deal with individual views, individual editors, and individual areas:

- **Toolbar:** The bar of buttons (and other little items) at the top of a view, as shown in Figure 3-19.

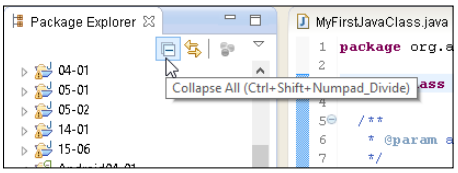


Figure 3-19: The toolbar in the Package Explorer view.

- **Menu button:** A downward-pointing arrow on the toolbar. When you click the menu button, a drop-down list of actions appears, as shown in Figure 3-20. Which actions you see in the list vary from one view to another.

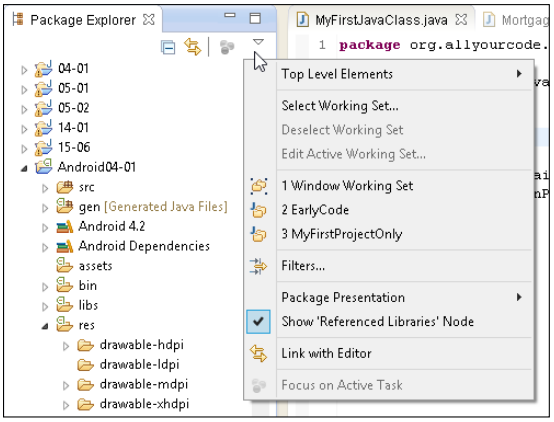


Figure 3-20: Clicking the menu button in the Package Explorer view.

- **Close button:** A button that eliminates a particular view or editor, as shown in Figure 3-21.

Figure 3-21: An editor's Close button.

- **Chevron:** A double arrow indicating that other tabs should appear in a particular area (but that the area is too narrow). The chevron shown in Figure 3-22 has a little number 2 beside it. The 2 tells you that, in addition to the two visible tabs, two tabs are invisible. Clicking the chevron opens a hover tip containing the labels of all the tabs. (See Figure 3-22.)

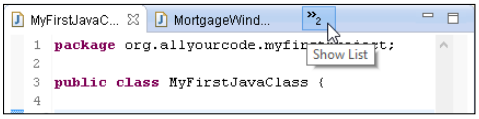


Figure 3-22: The chevron indicates that two editors are hidden.

- **Marker bar:** The vertical ruler on the left edge of the editor area. Eclipse displays tiny alert icons, called *markers*, inside the marker bar. (Refer to Figure 3-13.)

Returning to the big picture

The two terms in this section deal with the overall look and feel of Eclipse:

- **Layout:** An arrangement of certain views. The layout shown in Figure 3-3, for example, has seven views, four of which are active:
  - *Package Explorer view:* You see it on the far left side.
  - *Task List view and Outline views:* They're on the far right side.
  - *Problems, Javadoc, Declaration, and Console views:* They're near the bottom. In this area of the workspace, the Problems view is the active view.

  Along with all these views, the layout contains a single *editor area*. Any and all open editors appear inside this editor area.

- **Perspective:** A useful layout. If a particular layout is truly useful, someone gives that layout a name. And if a layout has a name, you can use the layout whenever you want. For example, the workbench shown in Figure 3-3 displays Eclipse's *Java perspective*. By default, the Java perspective contains six views in an arrangement much like the arrangement shown in Figure 3-3.

  Along with all these views, the Java perspective contains an editor area. (Sure, the editor area has several tabs, but the number of tabs has nothing to do with the Java perspective.)

  You can switch among perspectives by choosing Window➪Open Perspective on the Eclipse main menu bar. This book focuses almost exclusively on Eclipse's Java perspective. But if you like poking around, visit some of the other perspectives to get a glimpse of the power and versatility of Eclipse.