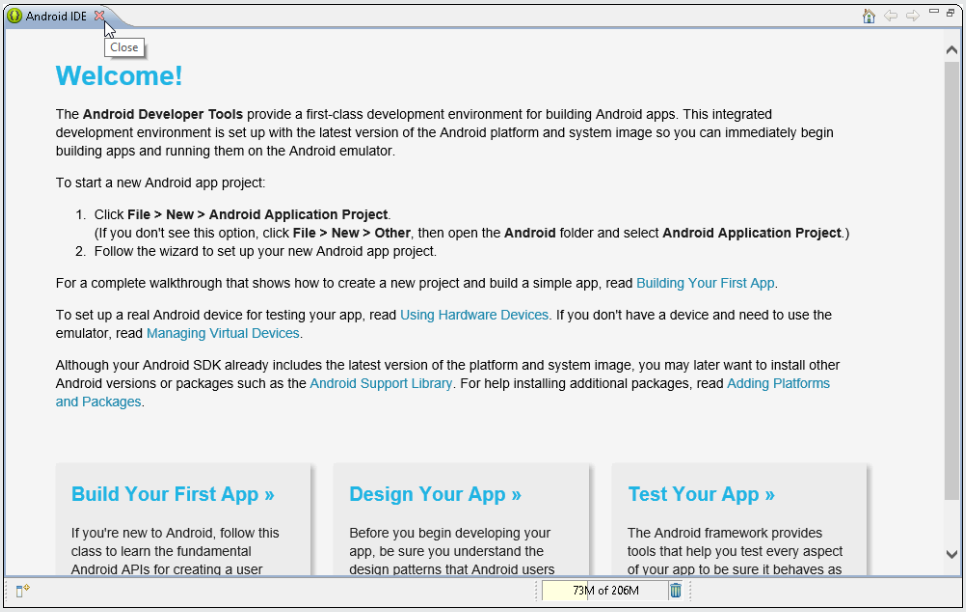
Part II

Writing Your Own Java Programs



Check out the article "Weird Computer Code" (and more) online at `www.dummies.com/extras/javaprogrammingforandroiddevelopers`.

In this part . . .

- Writing your first Java programs
- Assembling Java's building blocks
- Changing course as your program runs

Chapter 5

An Ode to Code

In This Chapter

- Reading the statements in a basic Java program
- Writing a Java console app
- Understanding the boilerplate Android activity

"Hello, hello, hello, . . . hello!"

—*The Three Stooges in* Dizzy Detectives *and other short films*

To most people, the words *Hello World* form a friendly (or even sugary) phrase. Is *Hello World* a song title? Is it the cheery slogan of a radio deejay? Maybe so. But to computer programmers, the phrase *Hello World* has a special meaning.

A *Hello World app* is the simplest program that can run in a particular programming language or on a particular platform. Authors create Hello World apps to show people how to start writing code for particular systems.

To help you get started with Java and Android, I devote this chapter to explaining a few Hello World programs. The programs don't do much. (In fact, you might argue that they don't do anything.) But they introduce some basic Java concepts.

To see Hello World apps for more than 450 different programming languages, visit `www.roesler-ac.de/wolfram/hello.htm`.

Examining a Standard Oracle Java Program

Listing 5-1 is a copy of the example in Chapter 3.

Listing 5-1: A Small Java Program

```
package org.allyourcode.myfirstproject;

public class MyFirstJavaClass {

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

}
```

When you run the program in Listing 5-1, the computer displays the word *Hello* in a dialog box, as shown in Figure 5-1. Now, I admit that writing and running a Java program just to make `Hello` appear on a computer screen is a lot of work, but every endeavor has to start somewhere.

Figure 5-1: Running the program in Listing 5-1.

Figure 5-2 describes the meaning of the code in Listing 5-1.

The next several sections present, explain, analyze, dissect, and otherwise demystify the Java program shown in Listing 5-1.

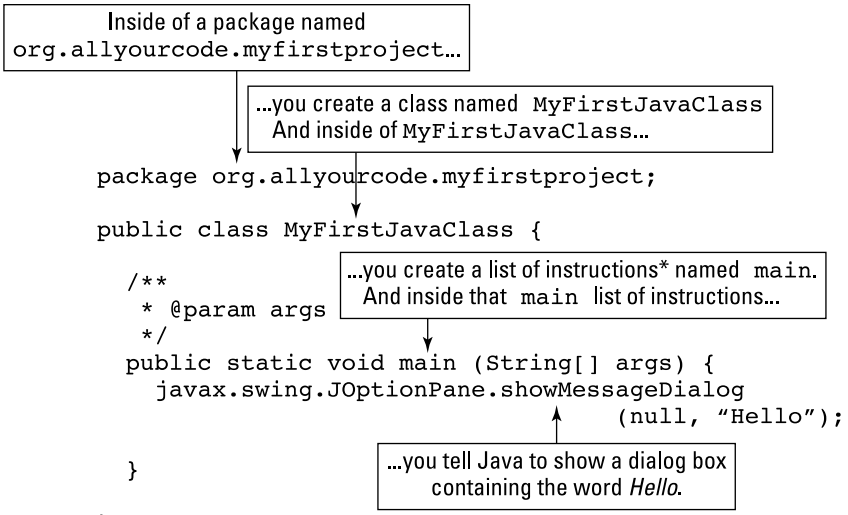


Figure 5-2: What you do in Listing 5-1.

* By the way, a list of instructions (such as the list named `main`) is called a *method*.

The Java class

Java is an object-oriented programming language. As a Java developer, your primary goal is to describe classes and objects. A *class* is a kind of category, like the category of all customers, the category of all accounts, the category of all geometric shapes, or, less concretely, the category of all `MyFirstJavaClass` elements, as shown in Listing 5-1. Just as the listing contains the words `class MyFirstJavaClass`, another piece of code to describe accounts might contain the words `class Account`. The `class Account` code would describe what it means to be (for example) one of several million bank accounts.

The previous paragraph contains a brief description of what it means to be a *class*. For a more detailed description, see Chapter 9.

You may know what js meant by the phrases "the category of all customers" and "the category of all geometric shapes," but you may wonder what "the category of all `MyFirstJavaClass` things" means or in what sense a computer program (such as the program in Listing 5-1) is a category. Here's my answer (which, I admit, is somewhat evasive): A Java program gets to be a "class" for esoteric, technical reasons and not because thinking of a Java program as a category always makes perfect sense. Sorry about that.

Except for the first line, the entire program In Listing 5-1 is a class. When I create a program like this one, I get to make up a name for my new class. In the listing, I choose the name `MyFirstJavaClass`. That's why the code starts with `class MyFirstJavaClass`, as shown in Figure 5-3.

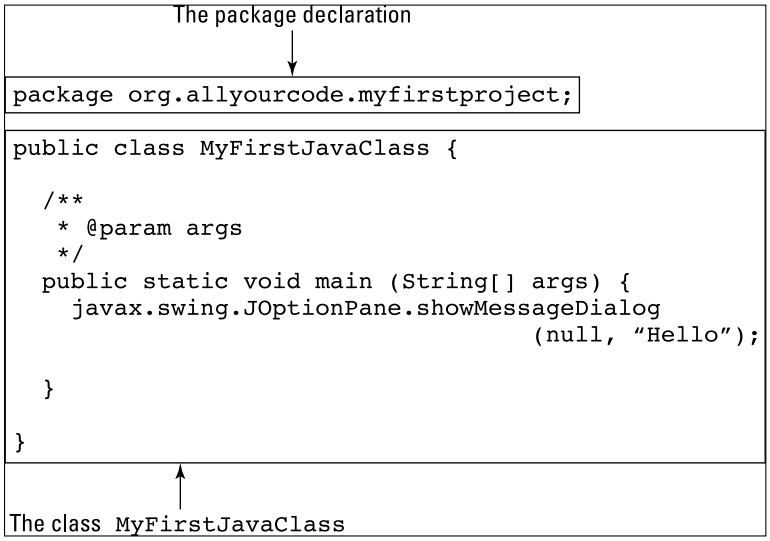


Figure 5-3: A simple Java program is a class.

The code inside the larger box in Figure 5-3 is, to be painfully correct, the *declaration* of a class. (This code is a *class declaration*.) I'm being slightly imprecise when I write in the figure that this code *is* a class. In reality, this code *describes* a class.

The declaration of a class has two parts: The first part is the *header,* and the rest — the part surrounded by curly braces, or {} —is the *class body,* as shown in Figure 5-4.

The word `class` is a Java *keyword*. No matter who writes a Java program, `class` is always used in the same way. On the other hand, `MyFirstJava Class` in Listing 5-1 is an *identifier* — a name for something (that is, a name that identifies something). The word `MyFirstJavaClass`, which I made up while I was writing Chapter 3, is the name of a particular class — the class that I'm creating by writing this program.

In Listing 5-1, the words `package`, `public`, `static`, and `void` are also Java keywords. No matter who writes a Java program, `package` and `class` and the other keywords always have the same meaning. For more jabber about keywords and identifiers, see the nearby sidebar, "Words, words, words."

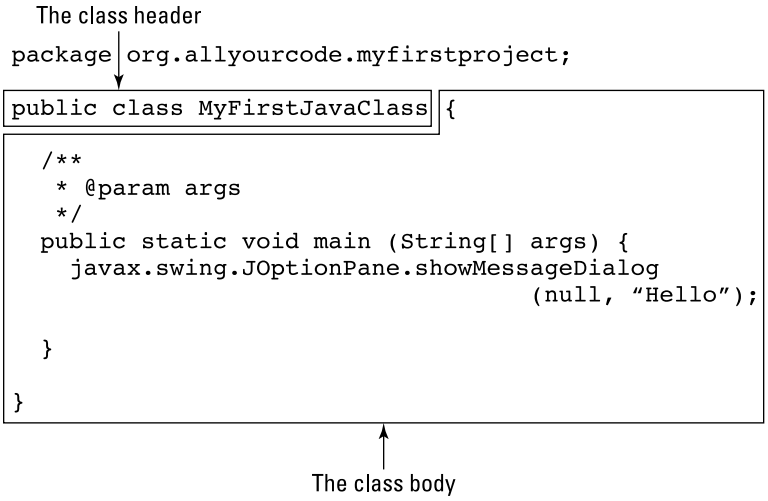


Figure 5-4: A class declaration's header and body.

To find out what the words `public`, `static`, and `void` mean, see Chapters 9 and 10.

tHE jAVA PROGRAMMING LANGUAGE IS cASe-sEnsITiVE. FOR EXAMPLE, iF YOU CHANGE A lowercase LETTER IN A WORD TO UPPERCASE OR CHANGE AN UPPERCASE WORD TO lowercase, YOU CHANGE THE WORD'S MEANING AND CAN EVEN MAKE THE WORD MEANINGLESS. iN THE FIRST LINE OF lIST-ING 5-1, FOR EXAMPLE, IF YOU TRIED TO REPLACE `class` WITH `Class`, THE WHOLE PROGRAM WOULD STOP WORKING.

The same holds true, to some extent, for the name of a file containing a particular class. For example, the name of the class in Listing 5-1 is `MyFirstJavaClass`, with 4 uppercase letters and 12 lowercase letters. So the code in the listing belongs in a file named `MyFirstJavaClass.java`, with exactly 4 uppercase letters and 12 lowercase letters in front of `.java`.

The names of classes

I'm known by several different names. My first name, used for informal conversation, is Barry. A longer name, used on this book's cover, is Barry Burd. The legal name that I use on tax forms is Barry A. Burd, and my passport (the most official document I own) sports the name Barry Abram Burd.

In the same way, elements in a Java program have several different names. For example, the class that's created in Listing 5-1 has the name `MyFirstJavaClass`. This is the class's *simple name* because, well, it's simple and it's a name.

Words, words, words

The Java language uses two kinds of words: keywords and identifiers. You can tell which words are keywords because Java has only 50 of them. Here's the complete list:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

As a rule, a *keyword* is a word whose meaning never changes (from one Java program to another). For example, in English, you can't change the meaning of the word *if*. It doesn't make sense to say, "I think that I shall never *if* I A poem lovely as a riff." The same concept holds true in a Java program: You can type `if (x > 5)` to mean "If x is greater than 5," but when you type `if (x > if)`, the computer complains that the code doesn't make sense.

In Listing 5-1, the words `package`, `public`, `class`, `static`, and `void` are keywords. Almost every other word in that listing is an *identifier,* which is generally a name for something. The identifiers in the listing include the package name `org.allyourcode.myfirstproject`, the class name `MyFirstJavaClass`, and a bunch of other words.

In programming lingo, words such as *Wednesday*, *Barry,* and *university* in the following sentence are identifiers, and the other words (*If*, *it's*, *is*, and *at*) are keywords:

If it's Wednesday, Barry is at the university.

(I'm undecided about the role of the word *the*. You can worry about it if you want.)

As in English and most other spoken languages, the names of items are reusable. For example, a recent web search turns up four people in the United States named Barry Burd (with the same uncommon spelling). You can even reuse well-known names. (A fellow student at Temple University had the name *John Wayne*, and in the 1980s two different textbooks were named *Pascalgorithms*.) The Android API has a prewritten class named `Activity`, but that doesn't stop you from defining another meaning for the name `Activity`.

Of course, having duplicate names can lead to trouble, so intentionally reusing a well-known name is generally a bad idea. (If you create your own thing named `Activity`, you'll find it difficult to refer to the prewritten `Activity` class in Android. As for my fellow Temple University student, everyone laughed when the teacher called roll.)

Listing 5-1 begins with the line `package org.allyourcode.myfirst project`. The first line is a *package declaration*. Because of this declaration, the newly created `MyFirstJavaClass` is inside a package named `org.allyourcode.myfirstproject`. So `org.allyourcode.myfirst project.MyFirstJavaClass` is the class's *fully qualified name*.

If you're sitting with me in my living room, you probably call me Barry. But if you've never met me and you're looking for me in a crowd of a thousand people, you probably call out the name Barry Burd. In the same way, the choice between a class's simple name and its fully qualified name depends on the context. For more information, see the later section "An import declaration."

Why Java methods are like meals at a restaurant

I'm a fly on the wall at Mom's Restaurant in a small town along Interstate 80. I see everything that goes on at Mom's: Mom toils year after year, fighting against the influx of high-volume, low-quality restaurant chains while the old-timers remain faithful to Mom's menu.

I see you walking into Mom's. Look — you're handing Mom a job application. You're probably a decent cook. If you get the job, you'll get carefully typed copies of every one of the restaurant's recipes. Here's one:

Scrambled eggs (serves 2)

5 large eggs, beaten

¼ cup 2% milk

1 cup shredded mozzarella

Salt and pepper to taste

A pinch of garlic powder

In a medium bowl, combine eggs and milk. Whisk until the mixture is smooth, and pour into preheated frying pan. Cook on medium heat, stirring the mixture frequently with a spatula. Cook for 2 to 3 minutes or until eggs are about halfway cooked. Add salt, pepper, and garlic powder. Add cheese a little at a time, and continue stirring. Cook for another 2 to 3 minutes. Serve.

Before your first day at work, Mom sends you home to study her recipes. But she sternly warns you not to practice cooking. "Save all your energy for your first day," she says.

On your first day, you don an apron. Mom rotates the sign on the front door so that the word *Open* faces the street. You sit quietly by the stove, tapping four fingers in round-robin fashion. Mom sits by the cash register, trying to look nonchalant. (After 25 years in business, she still worries that the morning regulars won't show up.)

At last! Here comes Joe the barber. Joe orders the breakfast special with two scrambled eggs.

What does Mom's Restaurant have to do with Java?

When you drill down inside the code of a Java class, you find these two important elements:

- **Method declaration:** The "recipe"

  "If anyone ever asks, here's how to make scrambled eggs."

- **Method call:** The "customer's order"

  Joe says, "I'll have the breakfast special with two scrambled eggs." It's time for you to follow the recipe.

Almost every computer programming language has elements akin to Java's methods. If you've worked with other languages, you may recall terms like *subprogram*, *procedure*, *function*, *subroutine*, *subprocedure*, or `PERFORM` *statement*. Whatever you call a *method* in your favorite programming language, it's a bunch of instructions, collected in one place and waiting to be executed.

Method declaration

A *method declaration* is a plan describing the steps that Java will take if and when the method is called into action. A *method call* is one of those calls to action. As a Java developer, you write both method declarations and method calls. Figure 5-5 shows you the method declaration and the method call from Listing 5-1.

If I'm being lazy, I refer to the code in the outer box in Figure 5-5 as a method. If I'm not being lazy, I refer to it as a method declaration.

A method declaration is a list of instructions: "Do this, then do that, and then do this other thing." The declaration in Listing 5-1 (and in Figure 5-5) contains a single instruction.

To top it all off, each method has a name. In Listing 5-1, the method declaration's name is `main`. The other words — such as `public`, `static`, and `void` — aren't parts of the method declaration's name.

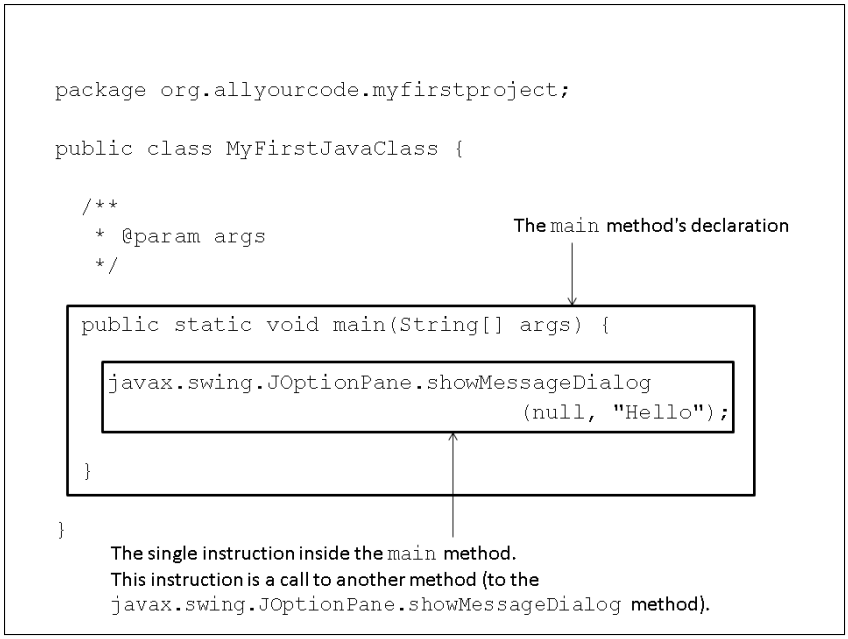


Figure 5-5: A method declaration and a method call.

The words `public`, `static`, and `void` are *modifiers* (similar to adjectives, in the English language). For more information about modifiers, see Chapters 9 and 10.

A method declaration has two parts: the *method header* (the first line) and the *method body* (the rest of it, which is the part surrounded by {} — curly braces), as shown in Figure 5-6.

Method call

A method *call* includes the name of the method being called, followed by some text in parentheses. So the code in Listing 5-1 contains a single method call:

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

In this code, `javax.swing.JOptionPane.showMessageDialog` is the name of a method, and `null, "Hello"` is the text in parentheses.

A Java instruction typically ends with a semicolon, so the following is a complete Java instruction:

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

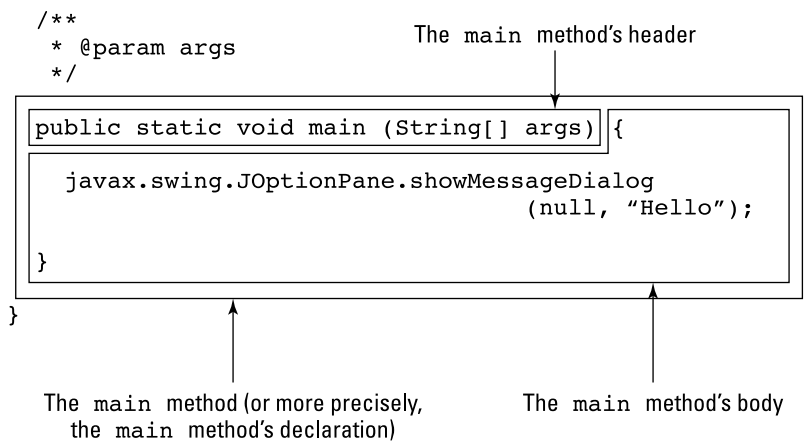


Figure 5-6: A method header and a method body.

This instruction tells the computer to execute whatever statements are inside the `javax.swing.JOptionPane.showMessageDialog` method declaration.

Another term for *Java instruction* is *Java statement*, or just *statement*.

The names of methods

Like many elements in Java, a method has several names, ranging from the shortest name to the longest name and with names in the middle. For example, the code in Listing 5-1 calls a method whose simple name is `showMessageDialog`.

In Java, each method lives inside a class, and `showMessageDialog` lives inside the API's `JOptionPane` class. So a longer name for the `showMessageDialog` method is `JOptionPane.showMessageDialog`.

A *package* in Java is a collection of classes. The `JOptionPane` class is part of an API package named `javax.swing`. So the `showMessageDialog` method's fully qualified name is `javax.swing.JOptionPane.showMessageDialog`. Which version of a method's name you use in the code depends on the context.

For more info on choosing between simple names and fully qualified names, see Chapter 9.

In Java, a package contains classes, and a class contains methods. (A class might contain other elements, too, but I tell you that story in Chapters 9 and 11.) A class's fully qualified name includes a package name, followed by the class's simple name. A method's fully qualified name includes a package name, followed by a class's simple name, followed by the method's simple name. To separate one part of a name from another, you use a period (or "dot").

Method parameters

In Listing 5-1, this call displays a dialog box:

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

The dialog box has the word *Message* in its title bar and an *i* icon on its face. (The letter *i* stands for *information*.) Why do you see the *Message* title and the *i* icon? For a clue, notice the method call's two parameters: `null` and `"Hello"`.

The effect of the values `null` and `"Hello"` depends entirely on the instructions inside the `showMessageDialog` method's declaration. You can read these instructions, if you want, because the entire Java API code is available for viewing — but you probably don't want to read the 2,600 lines of Java code in the `JOptionPane` class. (I'm sure you'd rather read the *CliffsNotes* version.)

Here's a brief description of the effect of the values `null` and `"Hello"` in the `showMessageDialog` call's parameter list:

- **In Java, the value `null` stands for "nothing."**

  In particular, the first parameter `null` in a call to `showMessageDialog` indicates that the dialog box doesn't initially appear inside any other window. That is, the dialog box can appear anywhere on the computer screen. (The dialog box appears inside of "nothing" in particular on the screen.)

- **In Java, double quotation marks denote a string of characters.**

  The second `"Hello"` parameter tells the `showMessageDialog` method to display the characters *Hello* on the face of the dialog box.

Even without my description of the `showMessageDialog` method's parameters, you can avoid reading the 2,600 lines of Java API code. Instead, you can examine the indispensable Java documentation pages. You can find these documentation pages by visiting

```
www.oracle.com/technetwork/java/javase/documentation
```

The main method in a standard Java program

Figure 5-7 shows a copy of the code from Listing 5-1 with arrows indicating what happens when the computer runs the code. The bulk of the code contains the declaration of a method named `main`.

Like any Java method, the `main` method is a recipe:

```
How to make scrambled eggs:
  Combine eggs and milk
  Whisk until smooth
  Pour into preheated frying pan
  Cook for 2 to 3 minutes while stirring the mixture
  Add salt, pepper, and garlic powder
  Add cheese a little at a time
  Cook for another 2 to 3 minutes
```

or

```
How to follow the main instructions for MyFirstJavaClass:
    Display "Hello" in a dialog box on the screen.
```

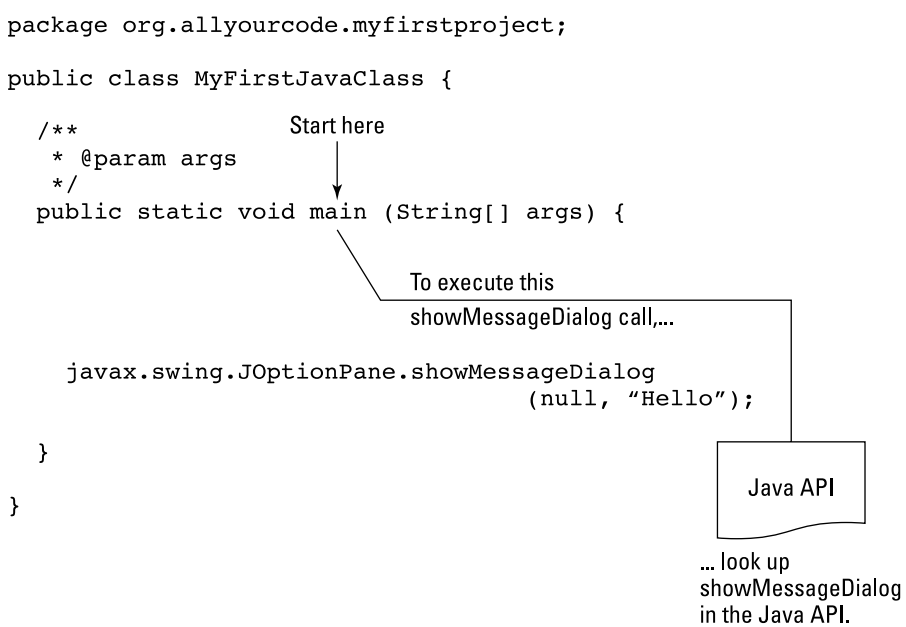


Figure 5-7: It all starts with the `main` method.

The word `main` plays a special role in Java. In particular, you never write code that explicitly calls a `main` method into action. The word `main` is the name of the method that's called into action when the program begins running.

When the `MyFirstJavaClass` program runs, the computer automatically finds the program's `main` method and executes any instructions inside the method's body. In the `MyFirstJavaClass` program, the `main` method's body has only one instruction. That instruction tells the computer to display *Hello* in a dialog box on the screen. So in Figure 5-1, `Hello` appears on the computer screen.

None of the instructions in a method is executed until the method is called into action. But if you give a method the name `main`, that method is called into action automatically.

Punctuating your code

In English, punctuation is vital. If you don't believe me, ask this book's copy editor, who suffered through my rampant abuse of commas and semicolons in the preparation of this manuscript. My apologies to her — I'll try harder in the next edition.

Anyway, punctuation is also important in a Java program. This list lays out a few of Java's punctuation rules:

- **Enclose a class body in a pair of curly braces.**

  In Listing 5-1, the `MyFirstJavaClass` body is enclosed in curly braces.

  The placement of a curly brace (at the end of a line, at the start of a line, or on a line of its own) is unimportant. The only important aspect of placement is consistency. The consistent placement of curly braces throughout the code makes the code easier for you to understand. And when you understand your own code, you *write* far better code. When you compose a program, Eclipse can automatically rearrange the code so that the placement of curly braces (and other program elements) is consistent. To make it happen, click the mouse anywhere inside the editor and choose Source➪Format.

- **Enclose a method body in a pair of curly braces.**

  In Listing 5-1, the `main` method's body is enclosed in curly braces.

- **A Java statement ends with a semicolon.**

For example, in Listing 5-1, the call to the `showMessageDialog` method ends with a semicolon.

- **A declaration ends with a semicolon.**

  Again in Listing 5-1, the first line of code (containing the package declaration) ends with a semicolon.

- **In spite of the previous two rules, don't place a semicolon immediately after a closing curly brace (}).**

  Listing 5-1 ends with two closing curly braces, and neither of these braces is followed by a semicolon.

- **Use parentheses to enclose a method's parameters, and use commas to separate the parameters.**

  In Listing 5-1 (where else?) the call to the `showMessageDialog` method has two parameters: `null` and `"Hello"`. The declaration of the `main` method has only one parameter: `args`.

  In the `main` method's parameter list, the `String[]` thing isn't a separate parameter. Instead, `String[]` is the `args` parameter's *type*. For more information about types, see Chapters 6, 9 and 12.

- **Use double quotation marks ("") to denote strings of characters.**

  In Listing 5-1, the `"Hello"` parameter tells the `showMessageDialog` method to display the characters *Hello* on the face of the dialog box.

- **Use dots to separate the parts of a qualified name.**

  In the Java API, the `javax.swing` package contains the `JOptionPane` class, which in turn contains the `showMessageDialog` method. So `javax.swing.JOptionPane.showMessageDialog` is the method's fully qualified name.

- **Use dots within a package name.**

  The dots in a package name are a bit misleading. A package name hints at uses for the code inside the package. But a package name doesn't classify packages into subpackages and sub-subpackages.

  For example, the Java API has the packages `javax.swing`, `javax.security.auth`, `javax.security.auth.login`, and many others. The word `javax` alone means nothing, and the `javax.security.auth.login` package isn't inside of the `javax.security.auth` package.

  The most blatant consequence of a package name's dots is to determine a file's location on the hard drive. For example, because of its package name, the code in Listing 5-1 must be in a folder named `myfirstproject`, which must be in a folder named `allyourcode`, which in turn must be in a folder named `org`, as shown in Figure 5-8.

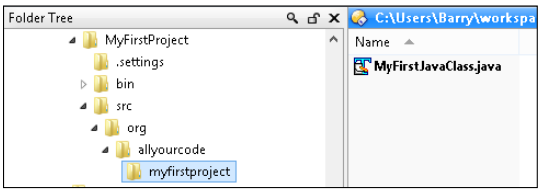


Figure 5-8: The folders containing a Java program.

Comments are your friends

Listing 5-2 has an enhanced version of the code in Listing 5-1. In addition to all the keywords, identifiers, and punctuation, Listing 5-2 has text that's meant for human beings (like you and me) to read.

Listing 5-2: Three Kinds of Comments

```
/*
 * Listing 5-2 in
 *   "Java For Android Developers For Dummies"
 *
 * Copyright 2013 Wiley Publishing, Inc.
 * All rights reserved.
] */

package org.allyourcode.myfirstproject;

/**
 * MyFirstJavaClass displays a dialog box
 * on the computer screen.
 *
 * @author  Barry Burd
 * @version 1.0 02/02/13
 * @see     java.swing.JOptionPane
 */
public class MyFirstJavaClass {

  /**
   * The starting point of execution.
   *
   * @param args
   *         (Not used.)
   */
  public static void main(String[] args) {
    javax.swing.JOptionPane.showMessageDialog
                                  (null, "Hello"); //null?
  }

}
```

A *comment* is a special section of text inside a program whose purpose is to help people understand the program. A comment is part of a good program's documentation.

The Java programming language has three kinds of comments:

- **Traditional comments:** The first seven lines in Listing 3-6 (over in Chapter 3) form one *traditional* comment. The comment begins with `/*` and ends with `*/`. Everything between the opening `/*` and the closing `*/` is for human eyes only. No information about `"Java For Android Developers For Dummies"` or `Wiley Publishing, Inc.` is translated by the compiler.

  To read about compilers, see Chapter 1.

  Lines 2–6 in Listing 5-2 have extra asterisks (`*`). I call them *extra* because these asterisks aren't required when you create a comment. They only make the comment look pretty. I include them in the listing because, for some reason that I don't entirely understand, most Java programmers insist on adding these extra asterisks.

- **End-of-line comments:** The text `//null?` in Listing 5-2 is an *end-of-line* comment — it starts with two slashes and goes to the end of a line of type. Once again, the compiler doesn't translate the text inside an end-of-line comment.

- **Javadoc comments:** A *javadoc* comment begins with a slash and two asterisks (`/**`). Listing 5-2 has two javadoc comments — one with the text `MyFirstJavaClass displays a dialog box . . .` and another with the text `The starting point. . . .`

  A *javadoc* comment is a special kind of traditional comment: It's meant to be read by people who never even look at the Java code.

  Wait — that doesn't make sense. How can you see the javadoc comments in Listing 5-2 if you never look at the listing?

  Well, with a few points and clicks, you can find all the javadoc comments in Listing 5-2 and turn them into a nice-looking web page, as shown in Figure 5-9.

To make documentation pages for your own code, follow these steps:

1. **Put Javadoc comments in your code.**
2. **From the main menu in Eclipse, choose Project⇨Generate Javadoc.**

   As a result, the Javadoc Generation dialog box appears.
3. **In the Javadoc Generation dialog box, select the Eclipse project whose code you want to document.**

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.allyourcode.myfirstproject

Class MyFirstJavaClass

```
java.lang.Object
  └ org.allyourcode.myfirstproject.MyFirstJavaClass
```

```
public class MyFirstJavaClass
extends java.lang.Object
```

MyFirstJavaClass displays a dialog box on the computer screen.

Version:
1.0 02/02/13
Author:
Barry Burd
See Also:
java.swing.JOptionPane

Constructor Summary

MyFirstJavaClass()

Method Summary

static void	main(java.lang.String[] args) The starting point of execution.

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MyFirstJavaClass

```
public MyFirstJavaClass()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

The starting point of execution.

Parameters:
args - (Not used.)

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Figure 5-9: Javadoc comments, generated from the code in Listing 5-2.

4. **Still in the Javadoc Generation dialog box, notice the name of the folder in the Destination field.**

   The computer puts the newly created documentation pages in that folder. If you prefer a different folder, you can change the folder name in this Destination field.

5. **Click Finish.**

   As a result, the computer creates the documentation pages.

If you visit the Destination folder and double-click the new `index.html` file's icon, you see your beautiful (and informative) documentation pages.

You can find the documentation pages for Java's built-in API classes by visiting `www.oracle.com/technetwork/java/javase/documentation`. Java's API contains thousands of classes, so don't memorize the names of the classes and their methods. Instead, you simply visit these online documentation pages.

What's Barry's excuse?

For years, I've been telling my students to put all kinds of comments in their code, and for years, I've been creating sample code (such as the code in Listing 5-1) containing few comments. Why?

Three little words: "Know your audience." When you write complicated, real-life code, your audience consists of other programmers, information technology managers, and people who need help deciphering what you've done. But when I write simple samples of code for this book, my audience is you — the novice Java programmer. Rather than read my comments, your best strategy is to stare at my Java statements — the statements that Java's compiler deciphers. That's why I put so few comments in this book's listings.

Besides, I'm a little lazy.

Another One-Line Method

Listing 5-3 contains another Hello World program. In fact, the code in Listing 5-3 is a bit simpler than the program in Listing 5-1.

Listing 5-3: A Console-Based Hello World Program

```
package com.allmycode.hello;

public class HelloText {

  public static void main(String[] args) {
    System.out.println("Hello");
  }

}
```

In Listing 5-3, the method call `System.out.println("Hello")` sends text to the Console view in Eclipse, as shown in Figure 5-10. Sending text to the Console is dull, dull, dull. But when you're writing code, a new program

often doesn't do what you think it should do. And adding a quick `System.out.println` call to the program helps you understand how the program behaves behind the scenes.

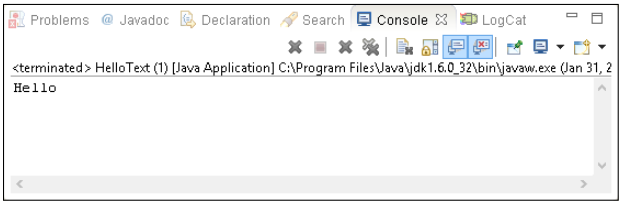


Figure 5-10: The Console view in Eclipse.

For concrete examples in which I use `System.out.println` to diagnose a program's behavior, see Chapter 13.

More Java Methods

To move beyond the rock-bottom simplicity of Listings 5-1 and 5-3, the code in Listing 5-4 mixes a few method declarations and a few method calls.

Listing 5-4: A Goodbye World Program

```
package com.allmycode.games;

import javax.swing.JOptionPane;

public class CountLives {

  public static void main(String[] args) {
    countdown();
  }

  static void countdown() {
    JOptionPane.showMessageDialog(null,
        "You have 2 more lives.", "The Game",
        JOptionPane.INFORMATION_MESSAGE);
    JOptionPane.showMessageDialog(null,
        „You have 1 more life.", „The Game",
        JOptionPane.WARNING_MESSAGE);
    JOptionPane.showMessageDialog(null,
        „You have no more lives.", „The Game",
        JOptionPane.ERROR_MESSAGE);
  }

}
```

Figures 5-11, 5-12, and 5-13 show a complete run of the code shown in Listing 5-4.]

Figure 5-11: The INFORMATION_MESSAGE from the first showMessageDialog call.

Figure 5-12: The WARNING_MESSAGE from the second showMessageDialog call.

Figure 5-13: The ERROR_MESSAGE from the third showMessageDialog call.

Figure 5-14 gives a more schematic overview of what happens when the computer runs the code shown in Listing 5-4. The `main` method calls the `countdown` method, which in turn calls Java's `showMessageDialog` method three times.

Using an import declaration

Compare the `showMessageDialog` calls in Listings 5-1 and 5-4. In Listing 5-1, you use the fully qualified name `javax.swing.JOptionPane.showMessageDialog`, but in Listing 5-4, you use the simpler name `JOptionPane.showMessageDialog`. What's this all about?

The answer is near the top of Listing 5-4. In that listing, you see the line

```
import javax.swing.JOptionPane;
```

This line, which announces that you intend to use the short name `JOptionPane` later in the listing's code, clarifies what you mean by `JOptionPane`. (You mean `javax.swing.JOptionPane`.) After having announced your intention in this *import declaration*, you can use the short name `JOptionPane` in the rest of the `CountLives` class code.

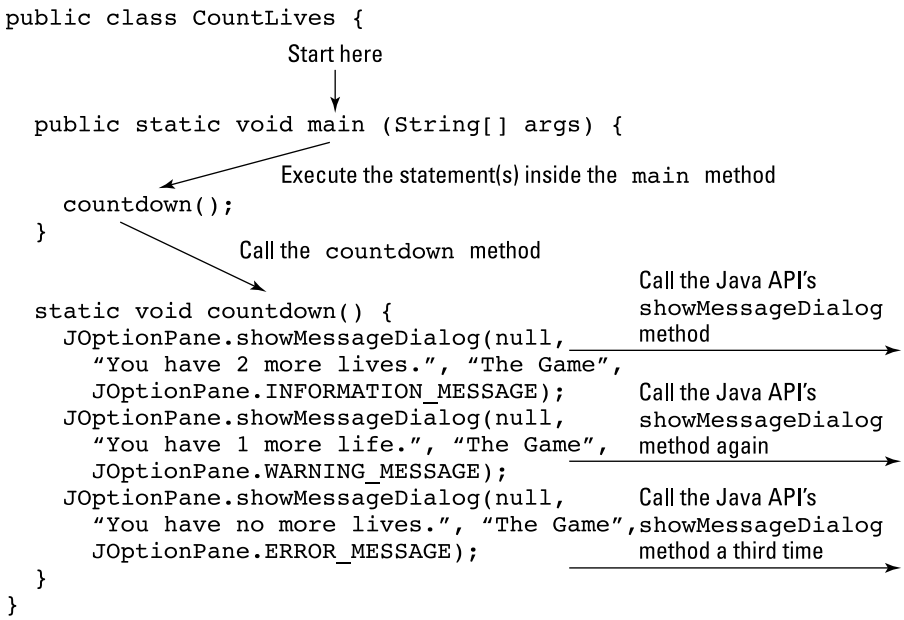


Figure 5-14: Going with the flow.

If you don't insert an import declaration at the top of the Java code file, you have to repeat the full `javax.swing.JOptionPane` name wherever you use the name `JOptionPane` in your code. (Refer to Listing 5-1.)

The details of this import business can be nasty, but (fortunately) many IDEs have features to help you write import declarations. For example, in Eclipse, you can avoid typing import declarations. You can quickly compose code using the shorter `JOptionPane.showMessageDialog` name. Then from the main menu in Eclipse, choose Source➪Organize Imports. When you do this, Eclipse adds the missing import declarations on your behalf.

More method parameters

Compare the `showMessageDialog` calls in Listings 5-1 and 5-4. The call in Listing 5-1 has two parameters, but each call in Listing 5-4 has four parameters. This is okay because the Java API contains at least two different `showMessageDialog` declarations — one with two parameters:

```
public static void showMessageDialog
         (Component parentComponent, Object message) {
// . . . etc.
```

And another with four parameters:

```
public static void showMessageDialog
         (Component parentComponent, Object message,
                        String title, int messageType) {
// . . . etc.
```

This example demonstrates *method overloading*. The Java API overloads the method name `showMessageDialog` by creating two (or more) ways to call `showMessageDialog`. A call with two parameters refers to one method declaration, and a call with four parameters refers to another declaration, as shown in Figure 5-15. The computer decides which method declaration to invoke by counting the parameters in the method call (and by checking other elements, as described in Chapter 7).

Here's what happens in the four-parameter version of `showMessageDialog`:

- **If the first parameter is `null`, the dialog box doesn't initially appear inside any other window.**

  This parameter serves the same purpose as the first parameter in the two-parameter `showMessageDialog` method.

- **The second parameter tells the `showMessageDialog` method which characters to display on the face of the dialog box.**

  This parameter serves the same purpose as the second parameter in the two-parameter `showMessageDialog` method.

Two parameters:

```
javax.swing.JOptionPane.showMessageDialog
                                (null, "Hello");
                                  ①     ②

public static void showMessageDialog
          (Component parentComponent, Object message) {
etc.                          ①                  ②
```

Four parameters:

```
                                  ①
JOptionPane.showMessageDialog (null,
  ②"You have 2 more lives.", "The Game",③
    JOptionPane.INFORMATION_MESSAGE);
                                  ④

public static void showMes①geDialog          ②
          (Component parentComponent, Object message,
                          String title, int messageType) {
etc.                                ③            ④
```

Figure 5-15: Parameters in the call match up with parameters in the declaration.

- **The third parameter tells the `showMessageDialog` method which characters to display on the title bar of the dialog box.**

  In Listing 5-4 (and back in Figures 5-11, 5-12, and 5-13), the title bar in every dialog box contains the words *The Game*.

- **The fourth parameter tells the `showMessageDialog` which icon to display on the face of the dialog box.**

  Figures 5-11, 5-12, and 5-13 show three of the five icons that may appear with a call to `showMessageDialog`. The remaining two possibilities are the question-mark icon (with the `JOptionPane.QUESTION_MESSAGE` parameter) and no icon (with the `JOptionPane.PLAIN_MESSAGE` parameter).

The `showMessageDialog` method calls in Listing 5-4 illustrate a point from the "R.java and the legend of the two vaudevillians" sidebar in Chapter 4, where the words `View.VISIBLE`, `View.INVISIBLE`, and `View.GONE` stand for the numbers 0, 4, and 8, respectively. Android uses these three numbers to represent different levels of screen visibility. In the same way, the names `JOptionPane.ERROR_MESSAGE`, `JOptionPane.INFORMATION_MESSAGE`, and `JOptionPane.WARNING_MESSAGE` stand for the numbers 0, 1, and 2. The statements inside the declaration of the `showOptionPane` message respond to each of these numbers by displaying a different icon.

Fewer method parameters

Another story about method parameters in Listing 5-4 begs to be told. In Listing 5-4 I call a method named `countdown`, and in the same class I declare my new `countdown` method.

When you call a method that's declared in the same class, you can use the method's simple name. It's the same way in real life. No one in my family calls me Barry Burd at home (unless they're really angry with me).

You may remember how the computer counts a method call's parameters and matches this with the number of parameters in the method's declaration. In Listing 5-4, the `countdown` call has no parameters (only an empty pair of parentheses) and the `countdown` method's declaration has the same number of parameters; namely, none. So the call and the declaration are compatible, and the computer executes the declaration's instructions.

To declare (or to call) a method with no parameters, use an empty pair of parentheses.

Hello, Android

An Android project's `src` directory contains your project's Java source code. Files in this directory have names such as `MainActivity.java`, `MyService.java`, `DatabaseHelper.java`, and `MoreStuff.java`.

You can cram hundreds of Java files into an Android project's `src` directory. But when you create a new project, Eclipse typically creates just one file for you. By default, Android creates a file named `MainActivity.java`. Listing 5-5 shows you the code in the `MainActivity.java` file.

Listing 5-5: Android Creates This Skeletal Activity Class

```
package com.allmycode.myfirstandroidapp;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
```

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the
    // action bar if it is present.
    getMenuInflater().inflate(R.menu.activity_main, menu);
    return true;
  }

}
```

Where's the main method?

To start the run of a standard Java program, the computer looks for a method named `main`. But the code in Listing 5-5 has no `main` method. Okay, I give up — how does a smartphone find the starting point of execution in an Android app?

The answer involves an app's XML code. You can build a standard Java program with Java code alone, but an Android app needs additional code. For one thing, every Android app needs its own `AndroidManifest.xml` file.

Chapter 4 describes an `AndroidManifest.xml` file.

Listing 5-6 contains a snippet of code from an `AndroidManifest.xml` file. (The code that I set in boldface is the most interesting code. The code that's not set in boldface isn't uninteresting. It's simply less interesting.)

Listing 5-6: The activity Element in an AndroidManifest.xml File

```
<activity
  android:name=
    "com.allmycode.myfirstandroidapp.MainActivity"
  android:label="@string/app_name" >
  <intent-filter>
    <action android:name=
      "android.intent.action.MAIN" />

    <category android:name=
      "android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

And here's what the code in Listing 5-6 "says" to your Android device:

- **The code's** `action` **element indicates that the program that's set forth in Listing 5-5 (the** `com.allmycode.myfirstandroidapp.MainActivity` **class) is** `MAIN`**.**

That is, the program in Listing 5-5 is the starting point of an app's execution. In response to this, your Android device reaches back inside the listing and executes the listing's onCreate method, `onCreate OptionsMenu` method, and several other methods that don't appear there.

- **The code's `category` element adds an icon to the device's Application Launcher screen.**

  On most Android devices, the user sees the Home screen. Then, by touching one element or another on the Home screen, the user gets to see the Launcher screen, which contains several apps' icons. By scrolling this screen, the user can find an appropriate app's icon. When the user taps the icon, the app starts running.

  In Listing 5-6, the category element's `LAUNCHER` value makes an icon for running `com.allmycode.myfirstandroidapp.MainActivity` (the Java program in Listing 5-5) available on the device's Launcher screen.

So there you have it. With the proper secret sauce (namely, the `action` and `category` elements in the `AndroidManifest.xml` file), an Android program's `onCreate` and `onCreateOptionsMenu` methods become the program's starting points of execution.

Extending a class

In Listing 5-5, the words `extends` and `@Override` tell an important story — a story that applies to all Java programs, not only to Android apps. The words `extends` and `@Override` tell the story of a class in the Android API. The API's `android.app.Activity` class forms the basis of all Android applications.

In Android developer lingo, an *activity* is one "screenful" of components. Each Android application can contain many activities. For example, an app's initial activity might list the films playing in your neighborhood. When you click a film's title, Android covers the entire list activity with another activity (perhaps an activity displaying a relevant film review).

When you *extend* the `android.app.Activity` class, you create a new kind of Android activity. In Listing 5-5, the words `extends Activity` tells the computer that a `MainActivity` is, in fact, an example of an Android `Activity`. That's good because the folks at Google have already written more than 5,000 lines of Java code to describe what an Android `Activity` can do. Being an example of an `Activity` in Android means that you can take advantage of all its prewritten code.

When you extend an existing Java class (such as the `Activity` class), you create a new class with the existing class's functionality. For details of this important concept, see Chapter 10.

Overriding methods

In Listing 5-5, a `MainActivity` is a kind of Android `Activity`. So a `MainActivity` is automatically a screenful of components with lots and lots of handy, prewritten code.

Of course, in some apps, you might not want all that prewritten code. After all, being a Republican or a Democrat doesn't mean believing everything in your party's platform. You can start by borrowing most of the platform's principles but then pick and choose among the remaining principles. In the same way, the code in Listing 5-5 declares itself to be an Android `Activity`, but then *overrides* two of the `Activity` class's existing methods.

In Listing 5-5, the word `@Override` indicates that the listing doesn't use the API's prewritten `onCreate` and `onCreateOptionsMenu` methods. Instead, the new `MainActivity` contains declarations for its own `onCreate` and `onCreateOptionsMenu` methods, as shown in Figure 5-16.

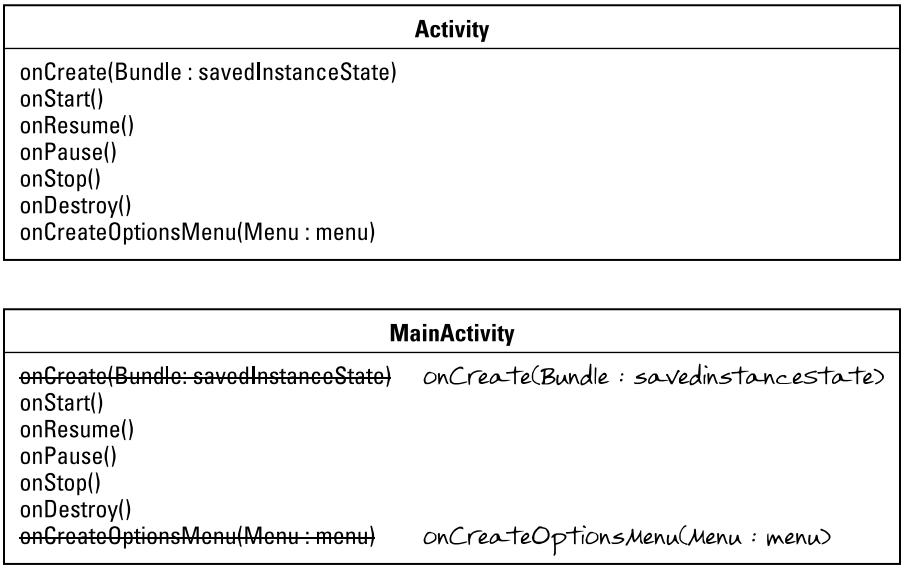


Figure 5-16: I don't like the prewritten `onCreate` and `OnCreate Options Menu` methods.

In particular, Listing 5-5's `onCreate` method calls `setContentView(R.layout.activity_main)`, which displays the material described in the `res/layout/activity_main.xml` file (the buttons and the text fields, for example) on the screen.

For an introduction to the `res/layout/activity_main.xml` file, see Chapter 4.

The other method in Listing 5-5 (the `onCreateOptionsMenu` method) does a similar trick with the `res/menu/activity_main.xml` file to display items on the app's Action bar.

An activity's workhorse methods

Every Android activity has a *lifecycle* — a set of stages that the activity undergoes from birth to death to rebirth, and so on. In particular, when your phone launches an activity, the phone calls the activity's `onCreate` method. The phone also calls the activity's `onStart` and `onResume` methods.

In Listing 5-5, I choose to declare my own `onCreate` method, but I don't bother declaring my own `onStart` and `onResume` methods. Rather than override the `onStart` and `onResume` methods, I silently use the `Activity` class's prewritten `onStart` and `onResume` methods.

To find out why you'd choose to override `onResume`, see Chapter 14.

When your phone ends an activity's run, the phone calls three additional methods: the activity's `onPause`, `onStop`, and `onDestroy` methods. So one complete sweep of your activity, from birth to death, involves the run of at least six methods — `onCreate`, then `onStart`, and then `onResume`, and later `onPause`, and then `onStop`, and, finally, `onDestroy`. As it is with all life forms, "ashes to ashes, dust to dust."

Don't despair. For an Android activity, reincarnation is a common phenomenon. For example, if you're running several apps at a time, the phone might run low on memory. In this case, Android can kill some running activities. As the phone's user, you have no idea that any activities have been destroyed. When you navigate back to a killed activity, Android re-creates the activity for you and you're none the wiser.

Here's another surprising fact. When you turn a phone from Portrait mode to Landscape mode, the phone destroys the *current* activity (the activity that's in Portrait mode) and re-creates that activity in Landscape mode. The phone calls all six of the activity's lifecycle methods (`onPause`, `onStop`, and so on) in order to turn the activity's display sideways. It's similar to starting on the transporter deck of the *Enterprise* and being a different person after being beamed down to the planet (except that you act like yourself and think like yourself, so no one knows that you're a completely different person).

Indeed, methods like `onCreate` and `onCreateOptionsMenu` in Listing 5-5 are the workhorses of Android development.