Part V
The Part of Tens

web extras

Enjoy an additional Part of Tens chapter from *Java Programming for Android Developers For Dummies* online at `www.dummies.com/extras/javaprogrammingforandroiddevelopers`.

In this part . . .

- Preventing mistakes
- Mining the web for more information

Chapter 15

Ten Ways to Avoid Mistakes

In This Chapter

- Checking your capitalization and value comparisons
- Watching out for fall-through
- Putting methods, listeners, and constructors where they belong
- Using static and nonstatic references
- Avoiding other heinous errors

"The only people who never make mistakes are the people who never do anything at all." One of my college professors said that. I don't remember the professor's name, so I can't give him proper credit. I guess that's my mistake.

Putting Capital Letters Where They Belong

Java is a case-sensitive language, so you really have to mind your *P*s and *Q*s — along with every other letter of the alphabet. Here are some concepts to keep in mind as you create Java programs:

- Java's keywords are all completely lowercase. For instance, in a Java `if` statement, the word *if* can't be *If* or *IF*.
- When you use names from the Java Application Programming Interface (API), the case of the names has to match what appears in the API.
- You also need to make sure that the names you make up yourself are capitalized the same way throughout the entire program. If you declare a `myAccount` variable, you can't refer to it as `MyAccount`, `myaccount`, or `Myaccount`. If you capitalize the variable name two different ways, Java thinks you're referring to two completely different variables.

For more info on Java's case-sensitivity, see Chapter 5.

Breaking Out of a switch Statement

If you don't break out of a `switch` statement, you get fall-through. For instance, if the value of `roll` is `7`, the following code prints all three words — `win`, `continue`, and `lose`:

```
switch (roll) {
case 7:
  System.out.println("win");
case 10:
  System.out.println("continue");
case 12:
  System.out.println("lose");
}
```

For the full story, see Chapter 8.

Comparing Values with a Double Equal Sign

When you compare two values with one another, you use a double equal sign. The line

```
if (inputNumber == randomNumber)
```

is correct, but the line

```
if (inputNumber = randomNumber)
```

is not correct. For a full report, see Chapter 6.

Adding Listeners to Handle Events

You want to know when the user clicks a widget, when an animation ends, or when something else happens, so you create listeners:

```
public class MainActivity extends Activity
    implements OnClickListener, AnimationListener {
  ...
  public void onClick(View view) {
    ...
  }
  public void onAnimationEnd(Animation animation) {
    ...
  }
```

When you create listeners, you must remember to set the listeners:

```
  ImageView widget = new ImageView(this);
  widget.setOnClickListener(this);
  ...
  AlphaAnimation animation =
      new AlphaAnimation(0.0F, 1.0F);
  animation.setAnimationListener(this);
  ...
```

If you forget the call to `setOnClickListener`, nothing happens when you click the widget. Clicking the widget harder a second time doesn't help.

For the rundown on listeners, see Chapter 11.

Defining the Required Constructors

When you define a constructor with parameters, as in

```
public Temperature(double number)
```

Java no longer creates a default parameterless constructor for you. In other words, you can no longer call

```
Temperature roomTemp = new Temperature();
```

unless you explicitly define your own parameterless `Temperature` constructor. For all the gory details on constructors, see Chapter 9.

Fixing Nonstatic References

If you try to compile the following code, you get an error message:

```
class WillNotWork {
  String greeting = "Hello";

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

You get an error message because `main` is static, but `greeting` isn't static. For the complete guide to finding and fixing this problem, see Chapter 9.

Staying within Bounds in an Array

When you declare an array with ten components, the components have indexes 0 through 9. In other words, if you declare

```
int guests[] = new int[10];
```

you can refer to the `guests` array's components by writing `guests[0]`, `guests[1]`, and so on, all the way up to `guests[9]`. You can't write `guests[10]`, because the `guests` array has no component with index 10.

For the latest gossip on arrays, see Chapter 12.

Anticipating Null Pointers

This book's examples aren't prone to throwing the `NullPointerException`, but in real-life Java programming, you see that exception all the time. A `NullPointerException` comes about when you call a method on an expression that doesn't have a "legitimate" value. Here's a cheap example:

```
public class ThrowNullPointerException {

  public static void main(String[] args) {
    String myString = null;
    display(myString);
```

```
  }

  static void display(String aString) {
    if (!aString.contains("confidential")) {
      System.out.println(aString);
    }
  }
}
```

The `display` method prints a string of characters only if that string doesn't contain the word *confidential*. The problem is that the `myString` variable (and thus the `aString` parameter) doesn't refer to a string of any kind — not even to the empty string (`""`).

When the computer reaches the call to `aString.contains`, the computer looks for a `contains` method belonging to `null`. But `null` is nothing. The `null` value has no methods. So you get a big `NullPointerException`, and the program comes crashing down around you.

To avoid this kind of calamity, think twice about any method call in your code. If the expression before the dot can possibly be `null`, add exception-handling code to your program:

```
try {
  if (!aString.contains("confidential")) {
    System.out.println(aString);
  }
} catch (NullPointerException e) {
  System.out.println("The string is null.");
}
```

For the story on handling exceptions, see Chapter 13.

Using Permissions

Some apps require explicit permissions. For example, the app in Chapter 13 talks to Twitter's servers over the Internet. This doesn't work unless you add a `<uses-permission>` element to the app's `AndroidManifest.xml` file:

```
<uses-permission android:name=
    "android.permission.INTERNET"/>
```

If you forget to add the <uses-permission> element to your AndroidManifest.xml file, the app can't communicate with Twitter's servers. The app fails without displaying a useful error message. Too bad!

The Activity Not Found

If you create a second activity for your app, you must add a new <activity> element in the app's AndroidManifest.xml file. For example, the Android app in Chapter 12 has two activities: MainActivity and MyListActivity. Eclipse automatically creates an <activity android:name=".MainActivity" element, but you have to type your own element for the MyListActivity:

```
<activity android:name=".MyListActivity">
    <intent-filter>
        <data android:scheme="checked" />
    </intent-filter>
</activity>
```

If you don't add this <activity> element, Android can't find the MyListActivity class, even though the MyListAcitivity.java file is in the app's Eclipse project directory. Your app crashes with an ActivityNotFoundException.

And that makes all the difference.