Chapter 12

Dealing with a Bunch of Things at a Time

In This Chapter

- Dealing with many objects at a time
- Creating versatile classes and methods
- Stepping through a list of items

All the world's a class,

And all the data, merely objects.

—Jimmy Shakespeare, 11-year-old computer geek

A *class* is a blueprint for things, and an *object* is a thing made from the blueprint. By *thing,* I mean a particular employee, a customer, an Android activity, or a more ethereal element, such as an `SQLiteOpenHelper`. Here's another quotation, this one from a more reliable source:

> *In fact, some Java classes are difficult to visualize. Android's* `SQLiteOpenHelper` *class assists developers in the creation of databases. An* `SQLiteOpenHelper` *doesn't look like anything in particular, and certainly not an online form or a bag of cheese.*
>
> —*Barry Burd*, Java Programming for Android Developers For Dummies, *Chapter 9*

This chapter covers a concept that you might not normally consider a class or an object — namely, a bunch of things. I use the word *bunch,* by the way, to avoid the formal terminology. (There's nothing wrong with the formal terminology, but I want to save it for this chapter's official grand opening, in the first section.)

Creating a Collection Class

A *collection class* is a class whose job is to store a bunch of objects at a time — a bunch of `String` objects, a bunch of `BagOfCheese` objects, a bunch of tweets, or whatever. You can create a collection class with the code in Listing 12-1.

Listing 12-1: Your First Collection Class

```
package com.allmycode.collections;

import java.util.ArrayList;

public class SimpleCollectionsDemo {

  public static void main(String[] args) {
    ArrayList arrayList = new ArrayList();
    arrayList.add("Hello");
    arrayList.add(", ");
    arrayList.add("readers");
    arrayList.add("!");

    for (int i = 0; i < 4; i++) {
      System.out.print(arrayList.get(i));
    }
  }

}
```

When you run the code in Listing 12-1, you see the output shown in Figure 12-1.

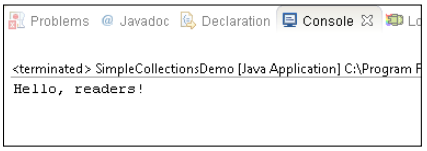

Figure 12-1: Running the code in Listing 12-1.

The code in Listing 12-1 constructs a new `ArrayList` instance and makes the `arrayList` variable refer to that new instance. The `ArrayList` class is one of many kinds of collection classes.

The statement `ArrayList arrayList = new ArrayList()` creates an empty list of things and makes the `arrayList` variable refer to that empty list. What does a list look like when it's empty? I don't know. I guess it looks like a blank sheet of paper. Anyway, the difference between having an empty list and

having *no* list is important. Before executing `ArrayList arrayList = new ArrayList()`, you have no list. After executing `ArrayList arrayList = new ArrayList()`, you have a list that happens to be empty.

The code in Listing 12-1 calls `arrayList.add` four times in order to put these four objects (all strings) into the list:

- `"Hello"`
- `", "`
- `"readers"`
- `"!"`

After calling `arrayList.add`, the list is no longer empty.

To display the objects in Eclipse's Console view, the code calls `System.out.print` four times, each time with a different object from the `arrayList` collection.

If you don't see Eclipse's Console view, click Window➪Show View➪Console.

There's a difference between `System.out.println` and `System.out.print` (without the `ln` ending): The `System.out.println` method goes to a new line after displaying its text; the `System.out.print` method does *not* go to a new line after displaying its text. In Listing 12-1, for example, with four calls to `System.out.print`, all four chunks of text appear on the same line in Eclipse's Console view.

The `for` statement in Listing 12-1 marches through the values in the `arrayList`. Every value in the list has an *index*, each ranging from 0 to 3.

In a Java collection, the initial index is always 0, not 1.

Java generics

If you look at Listing 12-1 in Eclipse's editor, you see lots of yellow warning markers, as shown in Figure 12-2. The warning text looks something like this: "ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized." What does that mean?

Starting with Java 5, the collection classes use generic types. You can recognize a generic type because of the angle brackets around its type name. For example, the following declaration uses `String` for a generic type:

```
ArrayList<String> arrayList = new ArrayList<String>();
```

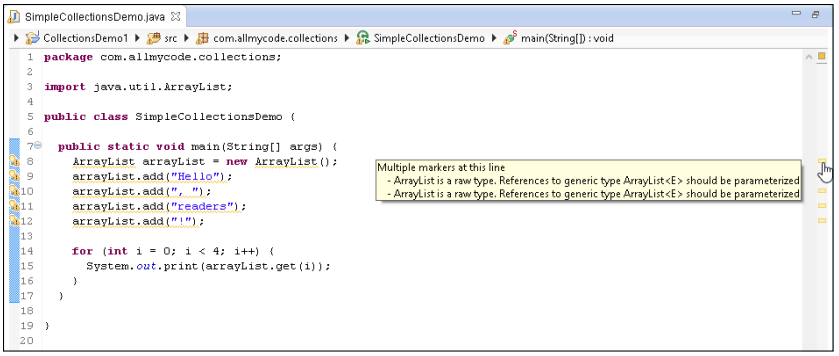

Figure 12-2: What are all these warnings about?

This improved declaration tells Java that the `arrayList` variable refers to a bunch of objects, each of which is an instance of `String`. When you substitute this new declaration in place of the nongeneric declaration from Listing 12-1, the yellow warning markers disappear, as you can see in Figure 12-3.

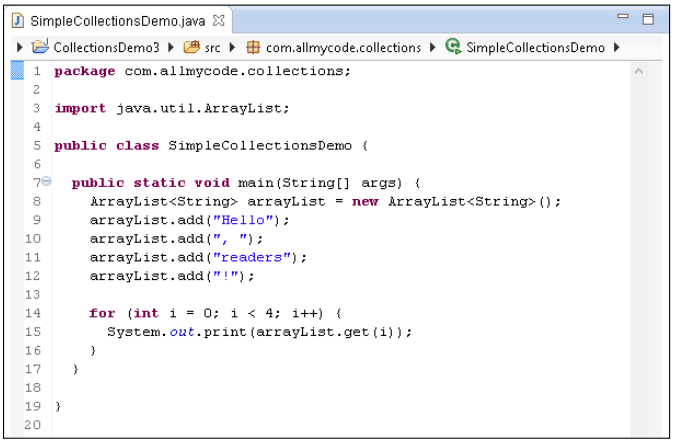

Figure 12-3: Using generics.

The yellow markers show warnings, not errors (refer to Figure 12-2), so you can get away with using the nongeneric declaration in Listing 12-1. But creating a nongeneric collection has some disadvantages. When you don't use generics (as in Listing 12-1), you create a collection that might contain objects of any kind. In that case, Java can't take advantage of any special properties of the items in the collection.

Here's an example. Chapter 9 starts with a description of the `BagOfCheese` class (which I've copied in Listing 12-2).

Listing 12-2: A Class in the Java Programming Language

```
package com.allmycode.andy;

public class BagOfCheese {
  String kind;
  double weight;
  int daysAged;
  boolean isDomestic;
}
```

You can put a few `BagOfCheese` objects into a nongeneric collection. But when you examine the objects in the collection, Java remembers only that the items in the collection are objects. Java doesn't remember that they're `BagOfCheese` objects, as shown in Figure 12-4.

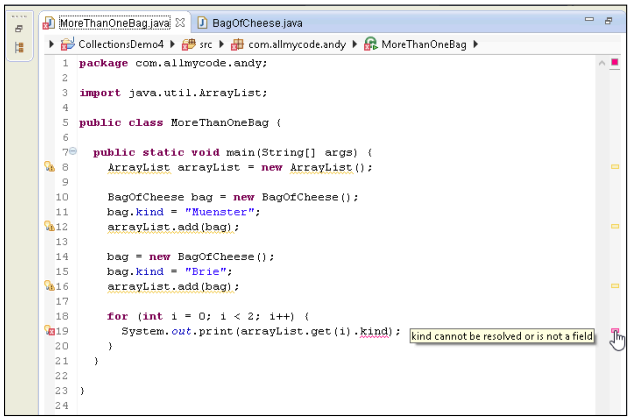

Figure 12-4: Your code without casting.

In Figure 12-4, Java doesn't remember that what you get from `arrayList` is always a `BagOfCheese` instance. So Java refuses to reference the object's `kind` field. (The last marker in Figure 12-4 is an error marker. Java can't run the code in that figure.)

Using casting, you can remind Java that the item you're getting from `arrayList` is a `BagOfCheese` instance.

```
System.out.print(((BagOfCheese) arrayList.get(i)).kind);
```

When you cast `arrayList.get(i)` to a `BagOfCheese` instance, you don't see the error message in Figure 12-4. You can run the code, warnings and all. Life is good, but the code is ugly! Look at all the parentheses you need in order to make the casting work correctly. It's a mess.

If you tweak the code to make `arrayList` generic, Java knows that what you get from `arrayList` is always a `BagOfCheese` instance, and every `BagOfCheese` instance has a `kind` field, as shown in Figure 12-5. No casting is required.

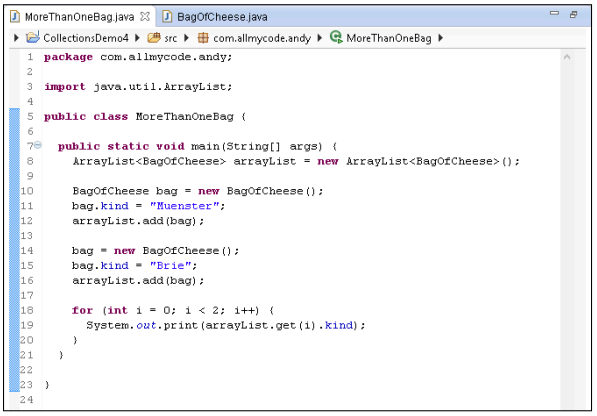

Figure 12-5: Java generics save the day.

You can use generics to create your own collection class. When you do, the generic type serves as a placeholder for an otherwise unknown type. Listing 12-3 contains a home-grown declaration of an `OrderedPair` class.

Listing 12-3: A Custom-Made Collection Class

```
package com.allmycode.collections;

public class OrderedPair<T> {
  private T x;
  private T y;

  public T getX() {
    return x;
  }
  public void setX(T x) {
    this.x = x;
  }
  public T getY() {
    return y;
  }
  public void setY(T y) {
    this.y = y;
  }
}
```

An `OrderedPair` object has two components: an `x` component and a `y` component. If you remember your high school math, you can probably plot ordered pairs of numbers on a two-dimensional grid. But who says that every ordered pair must contain numbers? The newly declared `OrderedPair` class stores objects of type `T`, and `T` can stand for any Java class. In Listing 12-4, I show you how to create an ordered pair of `BagOfCheese` objects.

Listing 12-4: Using the Custom-Made Collection Class

```
package com.allmycode.collections;

public class PairOfBags {

  public static void main(String[] args) {
    OrderedPair<BagOfCheese> pair =
        new OrderedPair<BagOfCheese>();

    BagOfCheese bag = new BagOfCheese();
    bag.kind = "Muenster";
    pair.setX(bag);

    bag = new BagOfCheese();
    bag.kind = "Brie";
    pair.setY(bag);

    System.out.println(pair.getX().kind);
    System.out.println(pair.getY().kind);
  }

}
```

Java's wrapper classes

Chapters 6 and 9 describe the difference between primitive types and reference types:

- **Each primitive type is baked into the language.**

  Java has eight primitive types.

- **Each reference type is a class or an interface.**

  You can define your own reference type. So the number of reference types in Java is potentially endless.

The difference between primitive types and reference types is one of Java's most controversial features, and developers often complain about the differences between primitive values and reference values.

Here's one of the primitive-versus-reference-type "gotchas:" You can't store a primitive value in an `ArrayList`. You can write

```
// THIS IS OKAY:
ArrayList<String> arrayList = new ArrayList<String>();
```

because String is a reference type. But you can't write

```
// DON'T DO THIS:
ArrayList<int> arrayList = new ArrayList<int>();
```

because `int` is a primitive type. Fortunately, each of Java's primitive types has a *wrapper* type, which is a reference type whose purpose is to contain another type's value. For example, an object of Java's `Integer` type contains a single `int` value. An object of Java's `Double` type contains a single `double` value. An object of Java's `Character` type contains a single `char` value. You can't create an `ArrayList` of `int` values, but you can create an `ArrayList` of `Integer` values.

```
// THIS IS OKAY:
ArrayList<Integer> arrayList = new ArrayList<Integer>();
```

Every primitive type's name begins with a lowercase letter. Every wrapper type's name begins with an uppercase letter.

In addition to containing primitive values, wrapper classes provide useful methods for working with primitive values. For example, the `Integer` wrapper class contains `parseInt` and other useful methods for working with `int` values:

```
String string = "17";
int number = Integer.parseInt(string);
```

On the downside, working with wrapper types can be clumsy. For example, you can't use arithmetic operators with Java's numeric wrapper types. Here's the way I usually create two `Integer` values and add them together:

```
Integer myInteger = new Integer(3);
Integer myOtherInteger = new Integer(15);

Integer sum =
  myInteger.intValue() + myOtherInteger.intValue();
```

Stepping through a collection

The program in Listing 12-1 uses a `for` loop with indexes to step through a collection. The code does what it's supposed to do, but it's a bit awkward. When you're piling objects into a collection, you shouldn't have to worry about which object is first in the collection, which is second, and which is third, for example.

Java has two features that make it easier to step through a collection of objects. One feature is the *iterator*. Listing 12-5 shows you how an iterator works.

Listing 12-5: Iterating through a Collection

```
package com.allmycode.collections;

import java.util.ArrayList;
import java.util.Iterator;

public class SimpleCollectionsDemo {

  public static void main(String[] args) {
    ArrayList<String> arrayList = new ArrayList<String>();
    arrayList.add("Hello");
    arrayList.add(", ");
    arrayList.add("readers");
    arrayList.add("!");

    Iterator<String> iterator = arrayList.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next());
    }
  }

}
```

The output from running Listing 12-5 is shown earlier, in Figure 12-1.

When you have a collection (such as an `ArrayList`), you can create an iterator to go along with that collection. In Listing 12-5, I show you how to create an iterator to go along with the `arrayList` collection, by calling

```
Iterator<String> iterator = arrayList.iterator();
```

After you've made this call, the variable iterator refers to something that can step through all values in the `arrayList` collection. Then, to step from one value to the next, you call `iterator.next()` repeatedly. And,

to find out whether another `iterator.next()` call will yield results, you call `iterator.hasNext()`. The call to `iterator.hasNext()` returns a `boolean` value: `true` when there are more values in the collection and `false` when you've already stepped through all the values in the collection.

An even nicer way to step through a collection is with Java's *enhanced* `for` *statement*. Listing 12-6 shows you how to use it.

Listing 12-6: Using the Enhanced for Statement

```
package com.allmycode.collections;

import java.util.ArrayList;

public class SimpleCollectionsDemo {

  public static void main(String[] args) {
    ArrayList<String> arrayList = new ArrayList<String>();
    arrayList.add("Hello");
    arrayList.add(", ");
    arrayList.add("readers");
    arrayList.add("!");

    for (String string : arrayList) {
      System.out.print(string);
    }
  }

}
```

An enhanced `for` statement doesn't have a counter. Instead, the statement has the format shown in Figure 12-6.

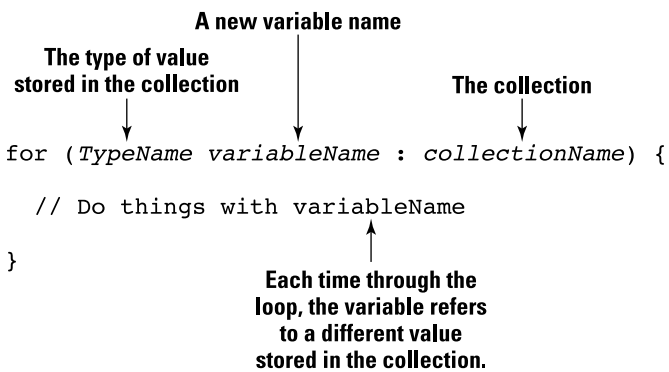

Figure 12-6: The anatomy of an enhanced `for` statement.

The enhanced `for` statement in Listing 12-6 achieves the same effect as the iterator in Listing 12-5 and the ordinary `for` statement in Listing 12-1. That is, the enhanced `for` statement steps through the values stored in the `arrayList` collection.

The enhanced `for` statement was introduced in Java 5.0. It's "enhanced" because, for stepping through a collection, it's easier to use than a pre-Java 5.0 `for` statement.

A cautionary tale

In an enhanced `for` statement, the variable that repeatedly stands for different values in the collection never refers directly to any of those values. Instead, this variable always contains a copy of the value in the collection. So, if you assign a value to that variable, you don't change any values inside the collection.

Here's a quiz. (Don't be scared. The quiz isn't graded.) What's the output of the following code?

```
package com.allmycode.collections;

import java.util.ArrayList;

public class SimpleCollectionsDemo {

  public static void main(String[] args) {
    ArrayList<String> arrayList = new ArrayList<String>();
    arrayList.add("Hello");
    arrayList.add(", ");
    arrayList.add("readers");
    arrayList.add("!");

    // THIS IS PRETTY BAD CODE
    for (String string : arrayList) {
      string = "Oops!";
      System.out.print(string);
    }

    System.out.println();

    for (String string : arrayList) {
      System.out.print(string);
    }
  }

}
```

The output is shown in Figure 12-7.

Figure 12-7: Running this section's bad code.

In the first `for` statement, the variable `string` is reassigned to refer to the word `"Oops!"` each time through the loop. The call to `System.out.print` displays that word `"Oops!"`. So far, so good.

But in the second `for` statement, the variable `string` isn't reassigned. Instead, the `string` variable retains whatever value it copies from the `arrayList` collection. So the second `for` statement displays the words `Hello, readers!`.

Java's many collection classes

The `ArrayList` class that I use in many of this chapter's examples is only the tip of the Java collections iceberg. The Java library contains many collections classes, each with its own advantages. Table 12-1 contains an abbreviated list.

Table 12-1 Some Collection Classes

Class Name	*Characteristic*
`ArrayList`	A resizable array.
`LinkedList`	A list of values, each having a field that points to the next one in the list.
`Stack`	A structure (which grows from bottom to top) that's optimized for access to the topmost value. You can easily add a value to the top or remove it from the top.
`Queue`	A structure (which grows at one end) that's optimized for adding values to one end (the rear) and removing values from the other end (the front).
`PriorityQueue`	A structure, like a queue, that lets certain (higher-priority) values move toward the front.
`HashSet`	A collection containing no duplicate values.
`HashMap`	A collection of key/value pairs.

Each collection class has its own set of methods (in addition to the methods that it inherits from `AbstractCollection`, the ancestor of all collection classes).

To find out which collection classes best meet your needs, visit the Android API documentation pages at `http://developer.android.com/reference`.

Arrays

In the "Stepping through a collection" section, earlier in this chapter, I cast aspersions on the use of an index in Listing 12-1. "You shouldn't have to worry about which object is first in the collection, which is second, and which is third," I write. Well, that's my story and I'm sticking to it, except in the case of an array. An array is a particular kind of collection that's optimized for indexing. That is, you can easily and efficiently find the 100th value stored in an array, the 1,000th value stored in an array, or the 1,000,000th value stored in an array.

The array is a venerable, tried-and-true feature of many programming languages, including newer languages such as Java and older languages such as FORTRAN. In fact, the array's history goes back so far that most languages (including Java) have special notation for dealing with arrays. Listing 12-7 illustrates the notation for arrays in a simple Java program.

Listing 12-7: Creating and Using an Array

```
package com.allmycode.collections;

public class SimpleCollectionsDemo {

  public static void main(String[] args) {
    String[] myArray = new String[4];
    myArray[0] = "Hello";
    myArray[1] = ", ";
    myArray[2] = "readers";
    myArray[3] = "!";

    for(int i = 0; i < 4; i++) {
      System.out.print(myArray[i]);
    }

    System.out.println();

    for (String string : myArray) {
      System.out.print(string);
    }
  }

}
```

Figure 12-8 shows the output of a run of the code in Listing 12-7. Both the ordinary `for` loop and the enhanced `for` loop display the same output.

Figure 12-8: Running the code in Listing 12-7.

In Listing 12-7, the ordinary `for` loop uses indexes, with each index marked by square brackets. As it is with all Java collections, the initial value's index is 0, not 1. Notice also the number `4` in the array's declaration — it indicates that "you can store `4` values in the array." The number 4 *doesn't* indicate that "you can assign a value to `myArray[4]`." In fact, if you add a statement such as `myArray[4] = "Oops!"` to the code in Listing 12-7, you get a nasty error message (`ArrayIndexOutOfBoundsException`) when you run the program.

The statement `String[] myArray = new String[4]` creates an empty array and makes the `myArray` variable refer to that empty array. The array can potentially store as many as four values. But, initially, that variable refers to an array that contains no values. It's not until Java executes the first assignment statement (`myArray[0] = "Hello"`) that the array contains any values.

You can easily and efficiently find the 100th value stored in an array or the 1,000,000th value stored in an array. Not bad for a day's work. So, what's the downside of using an array? The biggest disadvantage of an array is that each array has a fixed limit on the number of values it can hold. When you create the array in Listing 12-7, Java reserves space for as many as four `String` values. If, later in the program, you decide that you want to store a fifth element in the array, you need some clumsy, inefficient code to make yourself a larger array. You can also overestimate the size you need for an array, as shown in this example:

```
String[] myArray = new String[20000000];
```

When you overestimate, you probably waste a lot of memory space.

Another unpleasant feature of an array is the difficulty you can have in inserting new values. Imagine having a wooden box for each year in your collection of *Emperor Constantine Comics*. The series dates back to the year 307 A.D., when Constantine became head of the Roman Empire. You have only 1,700 boxes because you're missing about six years (mostly from the years 1150 to 1155). The boxes aren't numbered, but they're stacked one next to another in a line that's 200 meters long. (The line is as long as the 55th floor of a skyscraper is tall.)

At a garage sale in Istanbul, you find a rare edition of *Emperor Constantine Comics* from March 1152. After rejoicing over your first comic from the year 1152, you realize that you have to insert a new box into the pile between the years 1151 and 1153, which involves moving the year 2013 box about ten centimeters to the left, and then moving the 2012 box in place of the 2013 box, and then moving the 2011 box in place of the 2012 box. And so on. Life for the avid *Emperor Constantine Comics* collector is about to become tiresome! Inserting a value into the middle of a large array is equally annoying.

Java's varargs

In an app of some kind, you need a method that displays a bunch of words as a full sentence. How do you create such a method? You can pass a bunch of words to the sentence. In the method's body, you display each word, followed by a blank space, as shown here:

```
for (String word : words) {
  System.out.print(word);
  System.out.print(" ");
}
```

To pass words to the method, you create an array of `String` values:

```
String[] stringsE = { "Goodbye,", "kids." };
displayAsSentence(stringsE);
```

Notice the use of the curly braces in the initialization of `stringsE`. In Java, you can initialize any array by writing the array's values, separating the values from one another by commas, and surrounding the entire bunch of values with curly braces. When you do this, you create an *array initializer.*

Listing 12-8 contains an entire program to combine words into sentences.

Listing 12-8: A Program without Varargs

```
package com.allmycode.arrays;

public class UseArrays {

  public static void main(String[] args) {
    String[] stringsA = { "Hello,", "I", "must", "be",
                                              "going." };
    String[] stringsB = { "    ", "-Groucho" };
    String[] stringsC = { "Say", "Goodnight,",
                                             "Gracie." };
```

(continued)

Listing 12-8 *(continued)*

```
    String[] stringsD = { "    ", "-Nathan Birnbaum" };
    String[] stringsE = { "Goodbye,", "kids." };
    String[] stringsF = { "    ", "-Clarabell" };

    displayAsSentence(stringsA);
    displayAsSentence(stringsB);
    displayAsSentence(stringsC);
    displayAsSentence(stringsD);
    displayAsSentence(stringsE);
    displayAsSentence(stringsF);
  }

  static void displayAsSentence(String[] words) {
    for (String word : words) {
      System.out.print(word);
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

When you run the code in Listing 12-8, you see the output shown in Figure 12-9.

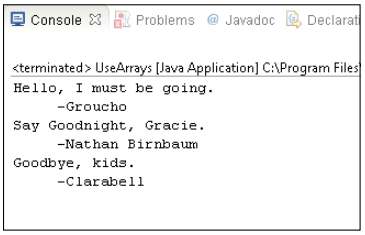

Figure 12-9: Running the code in Listing 12-8.

The code in Listing 12-8 is awkward because you have to declare six different arrays of `String` values. You can't combine the variable declarations and the method call. A statement such as

```
displayAsSentence("Say", "Goodnight,", "Gracie.");
```

is illegal because the call's parameter list has three values, and because the `displayAsSentence` method (in Listing 12-8) has only one parameter (one array). You can try fixing the problem by declaring `displayAsSentence` with three parameters:

```
static void displayAsSentence
    (String word0, String word1, String word2) {
```

But then you're in trouble when you want to pass five words to the method.

To escape from this mess, Java 5.0 introduces varargs. A parameter list with *varargs* has a type name followed by three dots. The dots represent any number of parameters, all of the same type. Listing 12-9 shows you how it works.

Listing 12-9: A Program with Varargs

```
package com.allmycode.varargs;

public class UseVarargs {

  public static void main(String[] args) {
    displayAsSentence("Hello,", "I", "must", "be",
                                          "going.");
    displayAsSentence("    ", "-Groucho");
    displayAsSentence("Say", "Goodnight,", "Gracie.");
    displayAsSentence("    ", "-Nathan Birnbaum");
    displayAsSentence("Goodbye,", "kids.");
    displayAsSentence("    ", "-Clarabell");
  }

  static void displayAsSentence(String... words) {
    for (String word : words) {
      System.out.print(word);
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

In Listing 12-9, the parameter list `(String... words)` stands for any number of `String` values — one `String` value, one hundred `String` values, or even no `String` values. So in Listing 12-9, I can call the `displayAsSentence` method with two parameters `("Goodbye,", "kids.")`, with three parameters `("Say", "Goodnight,", "Gracie.")`, and with five parameters `("Hello,", "I", "must", "be", "going.")`.

In the body of the `displayAsSentence` method, I treat the collection of parameters as an array. I can step through the parameters with an enhanced `for` statement, or I can refer to each parameter with an array index. For example, in Listing 12-9, during the first call to the `displayAsSentence` method, the expression `words[0]` stands for `"Hello"`. During the second call to the `displayAsSentence` method, the expression `words[2]` stands for `"Goodnight"`. And so on.

Using Collections in an Android App

I conclude this chapter with an Android app that's all about collections. This example will never become Google Play's featured app of the day, but the app demonstrates some of Java's collection features, and it shows you how to do a few interesting Android tricks.

The app begins by displaying five check boxes, as shown in Figure 12-10.

The user selects a few of the check boxes and then clicks the Show the List button. After the button is clicked, the app switches to a new activity (an Android `ListActivity`) that displays the numbers of the check boxes the user clicked, as shown in Figure 12-11.

In the app's code, I use an array to store the check boxes, I use an `ArrayList` for the items in the `ListActivity`, and I use an Android `ArrayAdapter` to determine which numbers the `ListActivity` displays.

Figure 12-10: The app's main activity.

Figure 12-11: The app's other activity.

The main activity's initial layout

In an Android app, you use *layouts* to describe the arrangement of widgets on the device's screen. The Android API has several kinds of layouts, including these:

- **LinearLayout:** Arranges widgets in a line across the screen, or in a column down the screen.
- **GridLayout:** Arranges widgets in a rectangular grid (that is, in the cells of a table whose borders are invisible).
- **RelativeLayout:** Arranges widgets by describing their positions relative to one another. For example, you can make the top of `button2` be 50 pixels below the bottom of `button1`.

In a `LinearLayout`, items appear one to the right of the other, or one beneath the other, depending on the layout's orientation. According to the code in Listing 12-10, the app's main activity has a `LinearLayout` with vertical orientation. So when you add items to the layout, new items appear beneath existing items on the activity's screen.

Listing 12-10: The Main Activity's Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:id="@+id/linearLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/main_activity" >
    </TextView>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onShowListClick"
        android:text="@string/show_list" >
    </Button>

</LinearLayout>
```

As always, you can download this chapter's Android app from the book's website (`www.allmycode.com/Java4Android`). But if you create the app from scratch, you can create most of Listing 12-10 by dragging and dropping items into Eclipse's graphical layout.

By default, Eclipse doesn't create an `android:id` attribute when you drag a `LinearLayout` or a `RelativeLayout` onto the Graphical Layout pane. But, as you can see in the next section, you need a way to refer back to the overall layout in this app's main activity. To create the code in Listing 12-10, you add your own `android:id` attribute to the code's `LinearLayout` element, either by right-clicking (in Windows) or Control-clicking (on the Mac) the layout on the Graphical Layout tab or typing the words `android:id="@+id/linearLayout"` on the `activity_main.xml` tab.

In this app example, the layout's `TextView` element is mere eye candy. The only interesting widget in Listing 12-10 is the button. When the user clicks the button, Android calls your main activity's `onShowListClick` method.

Both standard Oracle Java and Android Java have layouts. But the kinds of layouts that come with standard Java are different from the kinds that come with Android Java. For example, Android's `LinearLayout` is similar to (but not identical to) standard Java's `FlowLayout`. Android's `FrameLayout` is something like two of standard Java's layouts: `CardLayout` and `OverlayLayout`. Standard Java's `BorderLayout` has no direct counterpart in Android. But, in Android, you can achieve the same effect as a `BorderLayout` by combining some of Android's existing layouts or by creating your own custom layout.

The app's main activity

In Chapter 5, I introduce the lifecycle of an Android activity. Unlike a standard Java program, an Android activity has no `main` method. Instead, the Android operating system calls the activity's `onCreate`, `onStart`, and `onResume` methods.

Listing 12-11 contains the code for the app's main activity (refer to Figure 12-10). The `MainActivity` class in Listing 12-11 has all three lifecycle methods — `onCreate`, `onStart` and `onResume`, although you see only `onCreate` in Listing 12-11. The other two lifecycle methods (`onStart` and `onResume`) come quietly as a result of extending Android's `Activity` class.

Listing 12-11: The Main Activity's Java Code

```
package com.allmycode.lists;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.LinearLayout;

public class MainActivity extends Activity {
  static CheckBox[] checkBoxes = new CheckBox[5];

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    LinearLayout layout =
        (LinearLayout) findViewById(R.id.linearLayout);

    for (int i = 0; i < 5; i++) {
      checkBoxes[i] = new CheckBox(this);
      layout.addView(checkBoxes[i]);
    }
  }

  public void onShowListClick(View view) {
    Intent intent =
        new Intent(this, MyListActivity.class);
    startActivity(intent);
  }
}
```

The `onCreate` method in Listing 12-11 calls `findViewById` to locate the layout on the activity's screen. After assigning the result to the `layout` variable, the `onCreate` method's `for` loop adds five check boxes to `layout`. In the `CheckBox` constructor call . . .

> Hey, wait a minute! Don't I have to declare the five check boxes in the `activity_main.xml` file of Listing 12-10? Can I really add a widget to a layout using Java code? Yes, I can. To place a widget (a `TextView`, a `Button`, a `CheckBox`, or whatever) on the screen, I can either declare the widget in my `activity_main.xml` file or (as I do in Listing 12-11) call the layout's `addView` method in my Java code.
>
> Most Android developers agree that the `activity_main.xml` route is often the better way to go. After all, the widgets on a screen are just

"there" all at once. They don't usually appear one by one, as though someone commands them into existence. But creating check boxes with Java code is particularly good for this chapter's sample app, especially because I want to consider the check boxes as numbered from 0 to 4.

Anyway, in the `CheckBox` constructor call, the parameter `this` represents the app's main activity. When you create a widget in an Android app, you do so within a *context*, which is a pile of information about the environment in which a widget lives. A context includes facts such as an activity's package name, the values in the activity's `strings.xml` file, and a list of files associated with the activity.

Now here's the strange part: Android's `Activity` class is a subclass of Android's `Context` class. In other words. every activity *is* a context. In the `CheckBox` constructor in Listing 12-11, passing `this` to the constructor means that I'm passing a context to the constructor. For all the Android code that I've written, I've never gotten used to thinking of an activity as a kind of context. But the `android.app.Activity` class is a subclass of a subclass of a subclass of the `android.context.Context` class. So it's true. I can pass my main activity to the `CheckBox` constructor in Listing 12-11.

The `onShowListClick` method in Listing 12-11 responds to the click of the Show the List button in Figure 12-10. In that method's body, the call to `startActivity(intent)` makes a second activity replace the main activity. The main activity becomes *stopped* (that is, hidden) and another activity (an instance of the `MyListActivity` class) takes over the device's screen.

Android uses `Intent` objects to transition from one activity to another. The `Intent` object in Listing 12-11 is called an *explicit intent* because the name of the new activity's class (`MyListActivity`) is in the intent's constructor.

The alternative to an explicit intent is an *implicit intent*. With an implicit intent, you don't provide the new activity's class name. Instead, you provide information about the kinds of things you want the new activity to be able to do. When you call `startActivity` with an implicit intent, Android goes searching around the user's device for any activity (in your app or in other people's apps) that can do the kinds of things you want done. For example, with the intent `new Intent(Intent.ACTION_VIEW, "http://www.allmycode.com")`, Android searches for a web browser activity — an activity that can view this book's website. Anyway, don't get me started on Android's implicit intents. They can be extremely complicated. (Okay, if you insist, I describe implicit intents in great detail in my book *Android Application Development All-in-One For Dummies,* published by John Wiley & Sons, Inc.)

The app's List Activity

When the code in Listing 12-11 calls `startActivity(intent)`, an instance of the `MyListActivity` takes over the user's screen. The code for the `MyListActivity` class is in Listing 12-12, and the activity's screen is pictured earlier, in Figure 12-11.

Listing 12-12: The App's List Activity

```
package com.allmycode.lists;

import java.util.ArrayList;

import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;

public class MyListActivity extends ListActivity {

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ArrayList<Integer> listItems =
        new ArrayList<Integer>();
    for (int i = 0; i < 5; i++) {
      if (MainActivity.checkBoxes[i].isChecked()) {
        listItems.add(i);
      }
    }

    setListAdapter(new ArrayAdapter<Integer>(this,
        R.layout.my_list_layout, listItems));
  }
}
```

Android's `ListActivity` class is a subclass of the `Activity` class. So the class described in Listing 12-12 is a kind of activity. In particular, a `ListActivity` displays one thing after another, each in its own slot. The screen shown in Figure 12-11 displays three slots.

In a `ListActivity`, each slot can consist of one row or two rows or as many rows as the developer sees fit. The number of rows is determined by the layout of things in yet another XML file. On the screen shown in Figure 12-11, each slot has only one row.

When you declare a `ListActivity`, you call Android's `setListAdapter` method. In the call to `setListAdapter` in Listing 12-12, you have three parameters:

- **You provide a context.**

  In Listing 12-12, I provide the familiar `this` context.

- **You point to an XML file in your application's `res/layout` directory.**

  In Listing 12-12 I point to the `my_list_layout.xml` file.

- **You provide a collection of items, each to be displayed in its own slot.**

  In Listing 12-12, I provide `listItems`, which I declare to be an `ArrayList` of `Integer` values.

With respect to the layout file, Android treats a `ListActivity` a bit differently from the way it treats an ordinary `Activity`. To display a `ListActivity` (like the activity in Listing 12-12), Android reuses a layout XML file over and over again. Android reuses the layout file once for each of the items being displayed. In other words, the layout file for a `ListActivity` doesn't describe the entire screen. Instead, the layout file for a `ListActivity` describes only one of the many slots on the user's screen.

The `my_list_layout.xml` file for this chapter's app is shown in Listing 12-13. That XML file contains a single text view. So in Figure 12-11, each item in the list (each number beneath the "Using Collections" title) is in a single text view. Each slot in the list has a single text view. That's the way a `ListActivity` works.

Listing 12-13: The R.layout.my_list_layout.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android=
      "http://schemas.android.com/apk/res/android"
    android:id="@+id/identView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
</TextView>
```

The items displayed in Figure 12-11 correspond to the selected check boxes in Figure 12-10. (***Remember:*** When Java numbers the items in a collection, Java starts with 0.) To get the right numbers in `MyListActivity`, I fill the `listItems` collection in Listing 12-12. A `for` statement marches through the main activity's `checkBoxes` collection. The `for` statement adds the number `i` to `listItems` whenever the call to `checkBoxes[i].isChecked()` returns `true`.

For any check box that isn't selected, the call to `checkBoxes[i].isChecked()` returns `false`. So that `i` value doesn't get into the `listItems` collection. But for any check box that's selected, the call to `checkBoxes[i].isChecked()` returns `true`. That `i` value gets into the `listItems` collection and is displayed on the user's screen.

The app's AndroidManifest.xml file

When Eclipse creates a new Android project, Eclipse also offers to create a main activity. If you accept the offer to create a main activity, Eclipse puts an `activity` element in the project's `AndroidManifest.xml` file. This happens behind the scenes. So, when you add a second activity to an app (such as the activity in Listing 12-12) you can easily forget to manually add an `activity` element.

Listing 12-14 contains the `AndroidManifest.xml` file for this chapter's Android app:

Listing 12-14: The AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android"
  package="com.allmycode.lists"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="15" />

  <application
    android:icon="@drawable/icon"
    android:label="@string/app_name" >

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

(continued)

Listing 12-14 *(continued)*

```
    </activity>

    <activity android:name=".MyListActivity" />

  </application>

</manifest>
```

Notice that the code in Listing 12-14 contains two `activity` elements.

- The first `activity` element's `android:name` attribute has value `.MainActivity`.

  Eclipse creates this first element when you create the Android project, As the `android:name` attribute indicates, this element applies to the app's `MainActivity` class. Inside the `activity` element, the `intent-filter` element indicates that this activity's code can be the starting point of the app's execution, and this activity's icon can appear on the device's Apps screen.

- The second activity element's `android:name` attribute has value `.MyListActivity`.

  If you create this chapter's Android app on your own, you must edit the app's `AndroidManifest.xml` file and type this code by hand. As the `android:name` attribute indicates, this element applies to the app's `MyListActivity` class (refer to Listing 12-12).

  The `MyListActivity` class isn't the starting point of the app's execution, and the user shouldn't be able to launch this activity from the device's Apps screen. So the second `activity` element doesn't have the `MAIN` and `LAUNCHER` information that's in the listing's first `activity` element.

  In fact, I've rigged this app so that `MyListActivity` requires no `intent-filter` information, and no information at all between the `activity` element's start and end tags. So, in Listing 12-14, the second `activity` element has no start and end tags. Instead, this `activity` element has one empty element tag.

For more information about start tags, end tags, and empty element tags, see Chapter 4.

If you add an activity's Java code to an Android application, you must also add an `activity` element to the application's `AndroidManifest.xml` file.