Learn Java for Android Development: Inner Classes

by Shane Conder & Lauren Darcey 26 Oct 2010

Difficulty: Beginner Length: Medium Languages: English

Android SDK | Mobile Development

This post is part of a series called Learn Java for Android Development.

Learn Java for Android Challenge: Iteration

Learn Java for Android Development: More On Inner Classes

In this tutorial, you'll become familiar with the concept of inner classes in Java—those classes whose scope and definition are encompassed within another class. You'll also learn about anonymous inner classes, which are used quite frequently when developing with the Android SDK.

Android applications are written in the Java, an object-oriented programming language. In this tutorial, you'll learn about inner classes, when and why to use them and how they work. You'll also learn how to create new objects dynamically using anonymous inner classes.

What You'll Need

Technically, you don't need any tools to complete this tutorial but you will certainly need them to develop Android applications.

To develop Android applications (or any Java applications, for that matter), you need a development environment to write and build applications. Eclipse is a very popular development environment (IDE) for Java and the preferred IDE for Android development. It's freely available for Windows, Mac, and Linux operating systems.

For complete instructions on how to install Eclipse (including which versions are supported) and the Android SDK, see the Android developer website.

What is an Inner Class?

Most classes in Java are top-level classes. These classes, and the objects they define, are stand-alone. You can also create nested classes in order to clearly encapsulate and define subordinate objects that only matter in the context of the outer class. Nested classes are called inner classes.

Inner classes can have all the features of a regular class, but their scope is limited. Inner classes have another benefit: they have full access to the class in which they are nested—this feature makes inner classes perfect for implementing adapter functionality like iterators.

Here's an example of a top-level class with two inner classes:

```
public class User {

        //  User fields, including variables of type LoginInfo ar
        // Misc user methods

        class LoginInfo
        {
                // Login info fields
                // Login/Logout methods
```

```
                // Can access User fields/methods
        }

        class Preferences
        {
                // User preference fields
                // Get/Set preference methods
                // Reset preferences method
                // Can access User fields/methods
        }
}
```

In this example, the User class has two inner classes: LoginInfo and Preferences. While all user-related data and functionality could be defined in the User class, using the inner classes to compartmentalize functionality can make code easier to read and maintain. The inner classes LoginInfo and Preferences also have access to the protected/private fields and methods available within the User class, which they might not otherwise have due to security, if they were defined as stand-alone classes themselves.

It's important to remember, though, that inner classes really only exist to help the developer organize code; the compiler treats inner classes just like any other class, except that the inner classes have a limited scope, and are therefore tethered to the class they are defined with. Said another way, you would not be able to use or instantiate the LoginInfo or Preferences classes except with an instance of the User class, but the inner classes could access any fields or methods available in the outer class User, as needed.

Using Static Nested Classes

One particularly use for nested classes is static nested classes. A static inner class defines behavior that is not tied to a specific object instance, but applies across all instances. For example, we could add a third nested class, this time static, to the User class to control server-related functionality:

```
public class User {
```

```
        //  User fields, including variables of type LoginInfo a
        // Misc user methods

        class LoginInfo {}

        public static class ServerInfo {}
        {
                // Server info applies to all instances of User
        }
}
```

Because it's public, this static nested class can be instantiated using the following new statement:

```
User.ServerInfo sInfo = new User.ServerInfo();
```

The outer class does not have to be instantiated to perform this instantiation, thus the use of the class name. As such, a static nested class, unlike a non-static nested class (aka inner class) does not have access to members of the outer class--they may not even be instantiated.

The Power of Anonymous Inner Classes

Android uses anonymous inner classes to great effect. Anonymous inner classes are basically developer shorthand, allowing the developer to create, define, and use a custom object all in one "line." You may have seen examples of the use of anonymous inner class in sample code and not even realized it.
To create an anonymous inner class, you only provide the right-hand side of the definition. Begin with the new keyword, followed by the class or interface you wish to extend or implement, followed by the class definition. This will create the class and return it as a value which you can then use to call a method.
When we use an anonymous inner class, the object created does not get assigned a name (thus the term anonymous). The side effect, of course, is that the object is used used just once. For example, it's common to use an anonymous inner class to construct a custom version of an object as a return value. For example, here we extend the Truck class (assuming its defined elsewhere and has a field called mpg and two methods, start() and stop():

```
Truck getTruck()
{
        return new Truck()
        {
                int mpg = 3;
                void start() { /* start implementation */ }
                void stop() { /* stop implementation */ }
        };
}
```

Now let's look at practical examples of anonymous inner classes used in Android.

Using an Anonymous Inner Class to Define a Listener

Android developers often use anonymous inner classes to define specialized listeners, which register callbacks for specific behavior when an event occurs. For example, to listen for clicks on a View control, the developer must call the setOnClickListener() method, which takes a single parameter: a View.OnClickListener object.
Developers routinely use the anonymous inner class technique to create, define and use their custom View.OnClickListener, as follows:

```
Button aButton = (Button) findViewById(R.id.MyButton);
aButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                // User clicked my button, do something here!
            }
});
```

Using an Anonymous Inner Class to Start a Thread

Let's look at another example. It's quite common to define a new Thread class, provide the implementation of its run() method, and start that thread, all in one go:

```
new Thread() {
        public void run()
        {
                doWorkHere();
        }
}.start();
```

Using a Named Inner Class

Using anonymous inner classes for listeners in Android is so common that it is practically second nature to do so. Why, then, would you not want to use them? Let's answer this through a hypothetical example.
Let's say you have a screen that has 100 buttons on it (we did say hypothetical, right?). Now, let's say each button, when pressed, does the *exact same thing*. In this case, we'll just listen for clicks and Toast the text from the View object passed in (the text shown on the Button that was clicked):
Here's pseudo code to do that:

```
Button[] buttons = getAllOneHundredButtonsAsArray();
for (Button button : buttons) {
        button.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                        showToast(v.getText());
                }
        });
}
```

Short and elegant, so what's wrong with it? At each iteration, a new OnClickListener object is instantiated. Since each one is exactly the same, there is no good reason to create 100 of them. Instead, you can create a single, named, inner class, instantiate it once, then pass that to the setOnClickListener() method. For instance:

```
class MyActivity extends Activity {

    public void myMethod() {
       MyClickHandler handler = new MyClickHandler();
```

```
        Button[] buttons = getAllOneHundredButtonsAsArray();
        for (Button button : buttons) {
            button.setOnClickListener(handler);
        }
    }

    class MyClickHandler implements View.OnClickListener {
        public void onClick(View v) {
            showToast(((Button) v).getText());
        }
    }
}
```

If you prefer anonymity, you can still assign an anonymous inner class to a variable and use that, like so:

```
class MyActivity extends Activity {

    public void myMethod() {
        View.OnClickListener handler = new View.OnClickListener(
                public void onClick(View v) {
                    showToast(((Button) v).getText());
                }
            };

        Button[] buttons = getAllOneHundredButtonsAsArray();
        for (Button button : buttons) {
            button.setOnClickListener(handler);
        }
    }
}
```

The method is up to you, but keep in mind the potential memory and performance issues that instantiating a bunch of objects may have.

A Note on Nuances

This tutorial is meant to be an introductory guide to inner classes in Java. There are style considerations and nuances when using inner classes in different and creative ways. Even beyond that, you can explore further to learn more about the internal effects and marginal performance differences that can show up when you use nested classes in different ways. All of this, however, is well beyond the scope of this tutorial.

A Quick Note On Terminology

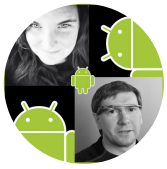
Although we have tried to be consistent with the terminology on nested and inner classes, the terminology is not always consistent from various online and offline resources. The following are a list of terms from the current Sun/Oracle documentation, which is as good as any for being authoritative on Java:

- **Nested Class**: a class defined inside another class
- **Static Nested Class**: a static class defined inside another class
- **Inner class**: a non-static nested class defined inside another class
- **Local Inner Class**: a class defined within a method
- **Anonymous Inner Class**: an unnamed class defined within a method

Confused? You can use the java.lang.Class methods called isLocalClass() and isAnonymous() class on instantiated classes to determine a couple of these properties. An Oracle blog entry attempts to clarify the situation a little, too, with a nice Venn diagram.

Wrapping Up

The Java programming language supports nested classes, allowing the developer great flexibility in defining objects. Inner classes can be used to organize class functionality or to define specialized behaviors that would otherwise require the developer to expose class data and functionality that really shouldn't be exposed. Static inner classes can be used to define fields and functionality that apply across all instances of a class. Finally, anonymous inner classes provide a useful shorthand for developers who want to create, define and use a custom object all at once.



Shane Conder & Lauren Darcey

Mobile developers Lauren Darcey and Shane Conder have coauthored numerous books on Android development. Our latest books include Sams Teach Yourself Android Application Development in 24 Hours (3rd Edition), Introduction to Android Application Development: Android Essentials (4th Edition), and Advanced Android Application Development (4th Edition). Lauren and Shane run a boutique consulting firm specializing in the development of commercial-grade Android applications for smartphones, tablets, wearables (i.e. Google Glass), and more. They can be reached on Google+, their blog at androidbook.blogspot.com and on Twitter @androidwireless.

Email Address

Update me weekly

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

QUICK LINKS - Explore popular categories

ENVATO TUTS+ +

JOIN OUR COMMUNITY +

HELP +

tuts+

26,219 Tutorials

1,147 Courses

25,575 Translations

Follow Envato Tuts+