```
        sensorData=0;
    }

    private void calibrate(int iSeed)
    {
        // Do some calibration here
    }

    protected void seedCalibration(int iSeed)
    {
        calibrate(iSeed);
    }

    public int getSensorData()
    {
        // Check sensor here

        return sensorData;
    }
}
```


Conditionals

Java includes conditional statements, which can be used to execute snippets of code if, and only if, certain conditions are met. Typically, a conditional statement involves two sides. If the two sides are equivalent, the statement is true, otherwise it is false.

Java has all the typical conditional operators, such as:

- **==** equal to, as in (a == b)
- **!=** not equal to, as in (x != y)

```
        kitty.setName("Wookie");
    }

    Cat haveKitten()
    {
        Cat kitten = new Cat("Luke");
        return kitten;
    }
```

Finally, let's call these methods and see how they act upon Cat object instances:

```
Cat cat1 = new Cat("Jabba");
Cat cat2 = new Cat("Leia");

cat1.getName();     // Returns Jabba
cat2.getName();     // Returns Leia
messWithCat(cat1);
changeKitty(cat2);
Cat cat3 = haveKitten();
cat1.getName();     // Returns Jabba - note that object remains unchanged!
cat2.getName();     // Returns Wookie
cat3.getName();     // Returns Luke
```


Wrapping Up

You've just completed a crash-course of the Java programming language. While you may not be ready to write your first Java app, you should be able to work through the simplest of the Android sample application Java classes and determine what they're up to, at least in terms of the Java syntax of things. The first class you're going to want to look into for Android development is the Activity class. An Android application uses activities to define different runtime tasks, and therefore you must define an Activity to act as the entry point to your application. Now that you've got a handle on Java syntax, we highly recommend that you work through a beginner Android tutorial.

```
    if(nemo instanceof SaltwaterFish) {
        // Nemo is a Saltwater fish!
    }
}
```

Using instanceof in Android Development

So, when it comes to Android development, when is the `instanceof` feature useful? Well, for starters, the Android SDK classes are organized in typical object oriented fashion: hierarchically. For example, the classes such as Button, TextView, and CheckBox, which represent different types of user interface controls, are all derived from the same parent class: View. Therefore, if you wanted to create a method that took a View parameter, but had different behavior depending upon the specific type of control, you could use the `instanceof` mechanism to check the incoming parameter and determine exactly what kind of view control had been passed in.

For example, the following method takes a View parameter, allowing you to pass in any type of View, but specifically singles out TextView controls for special processing:

```
void checkforTextView(View v)
{
     if(v instanceof TextView)
    {
        // This is a TextView control
    } else {
        // This is not a TextView control
    }
}
```

In this example, we might continue by making a call to a method that is only valid for a TextView object and not the generic View object—in which case, we would likely cast the View parameter to a TextView prior to making such a call. If, however, we wanted to make a call that is available in all View objects, but behaves differently in TextView objects, there is no need to test for this. Java will handle calling the appropriate version of the method specific to

trying to use it. This allows the developer to leverage new APIs where available while still supporting the older devices—all in the same application.

Inspecting Classes

Java classes are represented at runtime using the Class (java.lang.Class) class. This class provides the starting point for all reflection APIs. Within this class, you'll find many methods for inspecting different aspects of a class, such as its fields, constructors, methods, permissions, and more. You can also use the Class method called forName() to load a non-primitive class (e.g. not int, but Integer) by name dynamically at runtime, instead of at compile time:

```
String sClassName = "android.app.NotificationManager";
try {
    Class classToInvestigate = Class.forName(sClassName);

    // Dynamically do stuff with this class
    // List constructors, fields, methods, etc.

} catch (ClassNotFoundException e) {
    // Class not found!
} catch (Exception e) {
    // Unknown exception
}
```


The class (in this case, NotificationManager) need not have the corresponding import statement in your code; you are not compiling in this class into your application. Instead, the class loader will load the class dynamically at runtime, if possible. You can then inspect this Class object and use the reflection techniques described in the rest of this tutorial.

Inspecting the Constructors Available Within a Class

You can inspect the constructors available within a given Class. To get just the constructors that are publicly available, use getConstructors(). However, if you want to inspect those methods specifically declared within the class, whether they are public or not, use getDeclaredConstructors() instead. Both methods return an array of Constructor (java.lang.reflect.Constructor) objects.

```
} catch (ClassNotFoundException e) {
    // Class not found
} catch (NoSuchFieldException e) {
    // Field does not exist, likely we are on Android 2.1 or older
    // provide alternative functionality to support older devices
} catch (SecurityException e) {
    // Access denied!
} catch (Exception e) {
    // Unknown exception
}
```

Once you have a valid Field object, you can get its name using the toGenericString() method. If you have the appropriate permissions, you can also access the value of that class field using the appropriate get() and set() methods.

Inspecting the Methods Available Within a Class

You can inspect the methods available within a given Class. To get just the methods that are publicly available, including inherited methods, use getMethods(). However, if you want to inspect those methods specifically declared within the class (without inherited ones), whether they are public or not, use getDeclaredMethods() instead. Both methods return an array of Method (java.lang.reflect.Method) objects.

For example, the following code iterates through the declared methods of a class:

```
Method[] aClassMethods = classToInvestigate.getDeclaredMethods();
for(Method m : aClassMethods)
{
    // Found a method m
}
```

Once you have a valid Method object, you can get its name using the toGenericString() method. You can also examine the parameters used by the method and the exceptions it can throw. Finally, if you have the appropriate permissions, you can also call the method using the invoke() method.

```
16            <item>@string/indigo</item>
17            <item>@string/violet</item>
18        </string-array>
19    </resources>
```

To load this array resource in your Activity class, use the getStringArray() method of the Resources object. For instance:

```
String aColors[] = getResources().getStringArray(R.array.colorsArray);
```

Challenge #1: **Warm-Up Challenge**

Now you're ready to get started. Load the string array from the resources, as discussed above. Then, iterate through the array's contents using a for() loop. Print each string to the Android LogCat debug log using the Log.v() method.

Extra points if you use the shorthand version of for() loops, discussed in Learn Java for Android Development: Working with Arrays.

Find the answer to this challenge in the challengeOne() method of the downloadable project.

Challenge #2: **Stretch Your Skills**

Iterate the same array as Challenge #1, but use a different iteration mechanism. For example, use a while() loop instead. Print each string to the Android LogCat debug log using the Log.v() method.

Find the answer to this challenge in the challengeTwo() method of the downloadable project.

Challenge #3: **Reverse!**

Iterate the same array backwards. Print each string to the Android LogCat debug log using the Log.v() method.

HINT: Challenge #2 can help.

Find the answer to this challenge in the challengeThree() method of the downloadable project.

instance for the inner class, simply use the this syntax. To access the this instance of the enclosing class, you need to tack on the name of the enclosing class, then a dot, then this. For example:

```
MyEnclosingClass.this
```

Take a look at the full implementation of the anonymous inner class, as discussed above, in the full context of its enclosing class, here called ClassChaosActivity:

```
package com.androidbook.classchaos;

import java.text.SimpleDateFormat;
import java.util.Date;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class ClassChaosActivity extends Activity {

        public static final String DEBUG_TAG = "MyLoggingTag";
        /** Called when the activity is first created. */
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                final TextView myTextview = (TextView)
findViewById(R.id.TextViewToShow);
                Button myButton = (Button) findViewById(R.id.ButtonToClick);
                myButton.setOnClickListener(new View.OnClickListener() {
                        public void onClick(View v) {
```

How Does Javadoc Work?

Javadoc documentation uses a combination of processing the source code (and inspecting types, parameters, etc.) and reading special comment tags that the developer provides as metadata associated with a section of code.

A Javadoc-style comment must come just before the code it is associated with. For example, a Javadoc comment for a class should be just above the class declaration and a comment for a method should be just above the method declaration. Each comment should begin with a short description, followed by an option longer description. Then you can include an number of different metadata tags, which must be supplied in a specific order. Some important tags include:

- @author – who wrote this code
- @version – when was it changed
- @param – describe method parameters
- @return – describe method return values
- @throws – describe exceptions thrown
- @see – link to other, related items (e.g. "See also…")
- @since – describe when code was introduced (e.g. API Level)
- @deprecated – describe deprecated item and what alternative to use instead

You can also create your own custom tags for use in documentation.

Generate Javadoc-style Comments in Eclipse

While you are writing code in Eclipse, you can generate a Javadoc –style comment by selecting the item you want to comment (a class name, method name, etc.) and pressing Alt-Shift-J (Cmd-Shift-J on a Mac). This will create a basic Javadoc-style comment for you to fill in the details.

Simple Javadoc Class Comments

Let's look at an example. Here's a simple Javadoc comment that describes a class:

```
/**
 * Activity for loading layout resources
 *
 * This activity is used to display different layout resources for a tutorial on user
interface design.
 *
 * @author LED
 * @version 2010.1105
 * @since 1.0
 */
public class LayoutActivity extends Activity {
```

Here's what it will look like when you generate the Javadoc documentation:

Working with the String Class

The String class is available as part of the java.lang package, which is included within the Android SDK for developers to use. The complete documentation for the String class can be found with the Android SDK documentation.

The String class represents an immutable (unchangeable) sequence of Unicode (16-bit encoding) characters, appropriate for storing characters in any language (English, German, Japanese, and so on).

So what does this have to do with Android development? Well, strings are used to store content displayed on application screens, or to store the input taken in from a user. Android developers are constantly loading, creating, and manipulating string data. So let's look at some of the stuff we can do with the String class.

Creating Strings

The String class has numerous constructors, for creating and instantiating string variables. String variables can be set to empty using the null keyword. You can also set its content from byte, character, or other String data. For example, here are some ways to create String variables for use within your applications (some are initialized from the variables, like uVowels and sVowelBuilder, defined earlier in this tutorial):

```
String strVowels1 = "aeiou";
String strVowels2 = new String("aeiou");
String strVowels3 = new String(sVowelBuilder);
String strVowels4 = new String(sbVowels);
String strVowels5 = new String(uVowels);
String strVowels6 = new String(abyteVowels2);
String strVowels7 = new String(abyteVowelsU);
String strVowels8 = new String("a" + "e" + "iou");
String strVowels9 = new String( new char[]{'\u0061',

'\u0065','\u0069','\u006F','\u0075'});
String strVowels10 = new String(new byte[]{ '\u0061',
```

What's a Developer to Do?

Pop quiz! Which of the following strings correctly represents the 4th month of the (Gregorian) calendar year: A, April, APR, Apr, 4, or 04? The answer? All of them. Already, working with dates and times seems a bit complicated, doesn't it? In fact, we did a quick perusal of all the Java reference books in our office (not insubstantial) and very few cover dates and times in any substantial way. But don't panic. Just accept the fact that, as a developer, you will have to expend a bit of effort towards satisfying two goals:

1. Your External Goal: To allow the user to work with the date and time formats they are most comfortable with, or at least familiar with.
2. Your Internal Goal: To keep your application code format-independent, so it works everywhere with little hassle.

Now let's talk a bit about each of these goals.

The External Goal: Be Flexible & Use Standard Controls

Have you ever noticed that few travel websites let the user manually enter date or time information? Instead, they rely upon calendar pickers and fixed day, month and year drop-downs, or, at minimum, enforce a specific date format (usually tied to your language/country, which they ask for in advance). From a user interface perspective, your apps should honor date and time nuances, at least to some extent. However, you should also carefully consider the methods in which the user can enter date and time information. By using standard date and time pickers in your Android apps, such as DatePicker and TimePicker, you effectively limit the data users can input to that which can be easily converted into appropriate date and time structures, without having to parse every known format (not to mention its typos).

- The Date class (java.util.Date) is a utility class for storing date and time in a way that can be reasonably manipulated without having to constantly think about time in terms of milliseconds.
- The Calendar class (java.util.Calendar) is a utility class for working with different calendars, as well as for manipulating date and time information in a variety of ways.
- The GregorianCalendar class (a subclass of java.util.Calendar) is used primarily for date manipulation in the Western hemisphere, were we use a 12-month calendar, with 7 days to a week, and two eras (BC and AD).

Determining the Current Date and Time

There are a number of ways to determine the current time on an Android device.

You can determine the raw date and time data using the static method provided in the System class (java.lang.System):

```
long msTime = System.currentTimeMillis();
Date curDateTime = new Date(msTime);
```

This method relies upon the time that the device *thinks* it is, which may or may not be reliable. Another way to determine the current date and time uses the default constructor for the Date class, which creates a Date object with the current date and time:

```
Date anotherCurDate = new Date();
```

There are yet other ways to determine the true exact time—computers frequently check known time-keeping servers to make sure everyone is “in sync”. This process is slightly beyond the scope of this tutorial, but suffice to say, just requesting and receiving the correct time takes some time, which must then be accounted for before synchronization can really be achieved.

(Note: Android devices that connect to cellular services tend to have locale accurate date and time information as this may be used in communications with the radio towers.)

Creating Date and Time Variables from Scratch