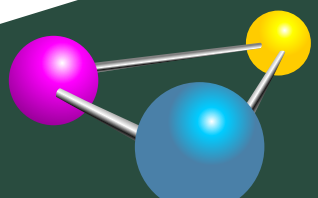
Version 1.0

digital updates at commonsware.com!

The Busy Coder's Guide to *Advanced* Android™ Development

Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Advanced Android Development

by Mark L. Murphy

The Busy Coder's Guide to Advanced Android Development
by Mark L. Murphy

Printing History:
Jul 2009: Version 1.0 ISBN: 978-0-9816780-1-6

Table of Contents

Welcome to the Warescription!

We hope you enjoy this digital book and its updates – keep tabs on the Warescription feed off the CommonsWare site to learn when new editions of this book, or other books in your Warescription, are available.

Each Warescription digital book is licensed for the exclusive use of its subscriber and is tagged with the subscribers name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the preface.

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

Some notes for first-generation Kindle users:

- You may wish to drop your font size to level 2 for easier reading
- Source code listings are incorporated as graphics so as to retain the monospace font, though this means the source code listings do not honor changes in Kindle font size

Preface

Welcome to the Book!

If you come to this book after having read its companion volume, The Busy Coder's Guide to Android Development, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and we appreciate your interest.

If you come to this book having learned about Android from other sources, thanks for joining the CommonsWare community! Android, while aimed at small devices, is a surprisingly vast platform, making it difficult for any given book, training, wiki, or other source to completely cover everything one needs to know. This book will hopefully augment your knowledge of the ins and outs of Android-dom and make it easier for you to create "killer apps" that use the Android platform.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

Prerequisites

This book assumes you have experience in Android development, whether from a CommonsWare resource or someplace else. In other words, you should have:

- A working Android development environment, whether it is based on Eclipse, another IDE, or just the command-line tools that accompany the Android SDK
- A strong understanding of how to create activities and the various stock widgets available in Android
- A working knowledge of the `Intent` system, how it serves as a message bus, and how to use it to launch other activities
- Experience in creating, or at least using, content providers and services

If you picked this book up expecting to learn those topics, you really need another source first, since this book focuses on other topics. While we are fans of The Busy Coder's Guide to Android Development, there are plenty of other books available covering the Android basics, blog posts, wikis, and, of course, the main Android site itself. A list of currently-available Android books can be found on the Android Programming knol.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance. Many chapters will reference material in The Busy Coder's Guide to Android Development, sometimes via the shorthand *BCG to Android* moniker.

In order to make effective use of this book, you will want to download the source code for it off of the book's page on the CommonsWare site.

Warescription

This book will be published both in print and in digital form. The digital versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to digital forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other digital formats will be added based on interest and the openness of the format.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in digital form. That way, your digital books are never out of date for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, including not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can exchange that copy for a free four-month Warescription.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare Web site.

Book Bug Bounty

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem in the current digital edition, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there. One coupon is given per email containing valid bug reports.

NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code License

The source code samples shown in this book are available for download from the CommonsWare Web site. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-

commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on June 1, **2013**. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.

Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

PART I – Advanced Widgets

CHAPTER 1

WebView, Inside and Out

Android uses the WebKit browser engine as the foundation for both its Browser application and the `WebView` embeddable browsing widget. The Browser application, of course, is something Android users can interact with directly; the `WebView` widget is something you can integrate into your own applications for places where an HTML interface might be useful.

In *BCG to Android*, we saw a simple integration of a `WebView` into an Android activity, with the activity dictating what the browsing widget displayed and how it responded to links.

Here, we will expand on this theme, and show how to more tightly integrate the Java environment of an Android application with the Javascript environment of WebKit.

Friends with Benefits

When you integrate a `WebView` into your activity, you can control what Web pages are displayed, whether they are from a local provider or come from over the Internet, what should happen when a link is clicked, and so forth. And between `WebView`, `WebViewClient`, and `WebSettings`, you can control a fair bit about how the embedded browser behaves. Yet, by default, the browser itself is just a browser, capable of showing Web pages and interacting with Web sites, but otherwise gaining nothing from being hosted by an Android application.

Except for one thing: `addJavascriptInterface()`.

The `addJavascriptInterface()` method on `WebView` allows you to inject a Java object into the `WebView`, exposing its methods, so they can be called by Javascript loaded by the Web content in the `WebView` itself.

Now you have the power to provide access to a wide range of Android features and capabilities to your `WebView`-hosted content. If you can access it from your activity, and if you can wrap it in something convenient for use by Javascript, your Web pages can access it as well.

For example, Google's Gears project offers a Geolocation API, so Web pages loaded in a Gears-enabled browser can find out where the browser is located. This information could be used for everything from fine-tuning a search to emphasize local content to serving up locale-tailored advertising.

We can do much of the same thing with Android and `addJavascriptInterface()`.

In the `WebView/GeoWeb1` project, you will find a fairly simple layout (`main.xml`):

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

All this does is host a full-screen `WebView` widget.

Next, take a look at the `GeoWebOne` activity class:

```
public class GeoWebOne extends Activity {
  private static String PROVIDER=LocationManager.GPS_PROVIDER;
```

```
  private WebView browser;
  private LocationManager myLocationManager=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    myLocationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE
);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb1.html");
  }

  @Override
  public void onResume() {
    super.onResume();
    myLocationManager.requestLocationUpdates(PROVIDER, 10000,
                                             100.0f,
                                             onLocationChange);
  }

  @Override
  public void onPause() {
    super.onPause();
    myLocationManager.removeUpdates(onLocationChange);
  }

  LocationListener onLocationChange=new LocationListener() {
    public void onLocationChanged(Location location) {
      // ignore...for now
    }

    public void onProviderDisabled(String provider) {
      // required for interface, not used
    }

    public void onProviderEnabled(String provider) {
      // required for interface, not used
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
      // required for interface, not used
    }
  };

  public class Locater {
    public double getLatitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);
```

```
      if (loc==null) {
        return(0);
      }

      return(loc.getLatitude());
    }

    public double getLongitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLongitude());
    }
  }
}
```

This looks a bit like some of the `WebView` examples in the *BCG to Android*'s chapter on integrating WebKit. However, it adds three key bits of code:

1. It sets up the `LocationManager` to provide updates when the device position changes, routing those updates to a do-nothing `LocationListener` callback object
2. It has a `Locater` inner class that provides a convenient API for accessing the current location, in the form of latitude and longitude values
3. It uses `addJavascriptInterface()` to expose a `Locater` instance under the name `locater` to the Web content loaded in the `WebView`

The Web page itself is referenced in the source code as `file:///android_asset/geoweb1.html`, so the `GeoWeb1` project has a corresponding `assets/` directory containing `geoweb1.html`:

```
<html>
<head>
<title>Android GeoWebOne Demo</title>
<script language="javascript">
  function whereami() {
    document.getElementById("lat").innerHTML=locater.getLatitude();
    document.getElementById("lon").innerHTML=locater.getLongitude();
  }
</script>
</head>
```

```
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami()">Update Location</a></p>
</body>
</html>
```

When you click the "Update Location" link, the page calls a `whereami()` Javascript function, which in turn uses the `locater` object to update the latitude and longitude, initially shown as "(unknown)" on the page.

If you run the application, initially, the page is pretty boring:

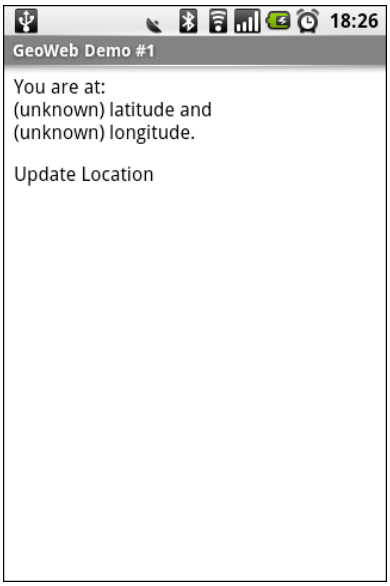


Figure 1. The GeoWebOne sample application, as initially launched

However, if you wait a bit for a GPS fix, and click the "Update Location" link...the page is still pretty boring, but it at least knows where you are:

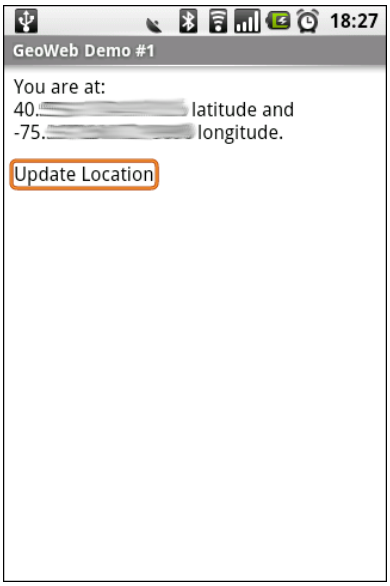


Figure 2. The GeoWebOne sample application, after clicking the Update Location link

Turnabout is Fair Play

Now that we have seen how Javascript can call into Java, it would be nice if Java could somehow call out to Javascript. In our example, it would be helpful if we could expose automatic location updates to the Web page, so it could proactively update the position as the user moves, rather than wait for a click on the "Update Location" link.

Well, as luck would have it, we can do that too. This is a good thing, otherwise, this would be a really weak section of the book.

What is unusual is how you call out to Javascript. One might imagine there would be an `executeJavascript()` counterpart to `addJavascriptInterface()`, where you could supply some Javascript source and have it executed within the context of the currently-loaded Web page.

Oddly enough, that is not how this is accomplished.

Instead, given your snippet of Javascript source to execute, you call `loadUrl()` on your `WebView`, as if you were going to load a Web page, but you put `javascript:` in front of your code and use that as the "address" to load.

If you have ever created a "bookmarklet" for a desktop Web browser, you will recognize this technique as being the Android analogue – the `javascript:` prefix tells the browser to treat the rest of the address as Javascript source, injected into the currently-viewed Web page.

So, armed with this capability, let us modify the previous example to continuously update our position on the Web page.

The layout for this new project (`WebView/GeoWeb2`) is the same as before. The Java source for our activity changes a bit:

```
public class GeoWebTwo extends Activity {
  private static String PROVIDER="gps";
  private WebView browser;
  private LocationManager myLocationManager=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    myLocationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE
);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb2.html");
  }

  @Override
  public void onResume() {
    super.onResume();
    myLocationManager.requestLocationUpdates(PROVIDER, 0,
                                             0,
                                             onLocationChange);
  }

  @Override
  public void onPause() {
    super.onPause();
    myLocationManager.removeUpdates(onLocationChange);
  }
```

```
  LocationListener onLocationChange=new LocationListener() {
    public void onLocationChanged(Location location) {
      StringBuilder buf=new StringBuilder("javascript:whereami(");

      buf.append(String.valueOf(location.getLatitude()));
      buf.append(",");
      buf.append(String.valueOf(location.getLongitude()));
      buf.append(")");

      browser.loadUrl(buf.toString());
    }

    public void onProviderDisabled(String provider) {
      // required for interface, not used
    }

    public void onProviderEnabled(String provider) {
      // required for interface, not used
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
      // required for interface, not used
    }
  };

  public class Locater {
    public double getLatitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLatitude());
    }

    public double getLongitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLongitude());
    }
  }
}
```

Before, the `onLocationChanged()` method of our `LocationListener` callback did nothing. Now, it builds up a call to a `whereami()` Javascript function, providing the latitude and longitude as parameters to that call. So, for

example, if our location were 40 degrees latitude and -75 degrees longitude, the call would be `whereami(40,-75)`. Then, it puts `javascript:` in front of it and calls `loadUrl()` on the `WebView`. The result is that a `whereami()` function in the Web page gets called with the new location.

That Web page, of course, also needed a slight revision, to accommodate the option of having the position be passed in:

```
<html>
<head>
<title>Android GeoWebTwo Demo</title>
<script language="javascript">
  function whereami(lat, lon) {
    document.getElementById("lat").innerHTML=lat;
    document.getElementById("lon").innerHTML=lon;
  }
</script>
</head>
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami(locater.getLatitude(), locater.getLongitude())">
Update Location</a></p>
</body>
</html>
```

The basics are the same, and we can even keep our "Update Location" link, albeit with a slightly different `onClick` attribute.

If you build, install, and run this revised sample on a GPS-equipped Android device, the page will initially display with "(unknown)" for the current position. After a fix is ready, though, the page will automatically update to reflect your actual position. And, as before, you can always click "Update Location" if you wish.

Gearing Up

In these examples, we demonstrate how WebView can interact with Java code, code that provides a service a little like one of those from Gears.

Of course, what would be really slick is if we could use Gears itself.

The good news is that Android is close on that front. Gears is actually baked into Android. However, it is only exposed by the Browser application, not via `WebView`. So, an end user of an Android device can leverage Gears-enabled Web pages.

For example, you could load the Geolocation sample application in your Android device's Browser application. Initially, you will get the standard "can we please use Gears?" security prompt:

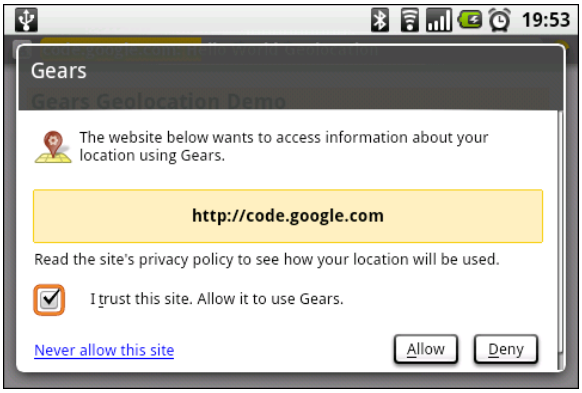


Figure 3. The Gears security prompt

Then, Gears will fire up the GPS interface (if enabled) and will fetch your location:

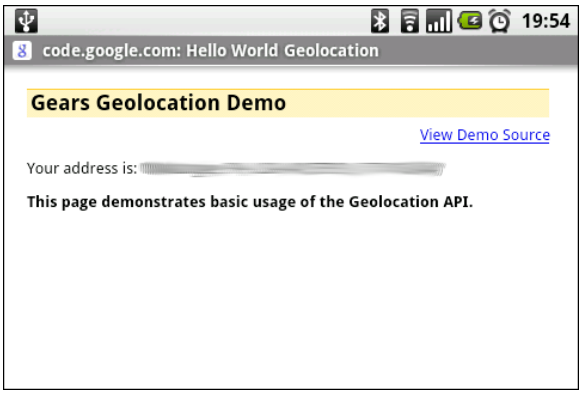


Figure 4. The Gears Geolocation sample application

Back To The Future

The core Android team has indicated that these sorts of capabilities will increase in future editions of the Android operating system. This could include support for more types of plugins, a richer Java-Javascript bridge, and so on.

You can also expect some improvements coming from the overall Android ecosystem. For example, the PhoneGap project is attempting to build a framework that supports creating Android applications solely out of Web content, using WebView as the front-end, supporting a range of Gears-like capabilities and more, such as accelerometer awareness.

CHAPTER 2

Crafting Your Own Views

One of the classic forms of code reuse is the GUI widget. Since the advent of Microsoft Windows – and, to some extent, even earlier – developers have been creating their own widgets to extend an existing widget set. These range from 16-bit Windows "custom controls" to 32-bit Windows OCX components to the innumerable widgets available for Java Swing and SWT, and beyond. Android lets you craft your own widgets as well, such as extending an existing widget with a new UI or new behaviors.

Getting Meta

One common way of creating your own widgets is to aggregate other widgets together into a reusable "meta" widget. Rather than worry about all the details of measuring the widget sizes and drawing its contents, you simply worry about creating the public interface for how one interacts with the widget.

In this section, we will look at the `Views/Meter` sample project. Here, we bundle a `ProgressBar` and two `ImageButton` widgets into a reusable `Meter` widget, allowing one to change a value by clicking "increment" and "decrement" buttons. In most cases, one would probably be better served using the built-in `SeekBar` widget. However, there are times when we only want people to change the value a certain amount at a time, for which the `Meter` is ideally suited. In fact, we will reuse the `Meter` in a later chapter when we show how to manipulate the various volume levels in Android.

The Widget Layout

The first step towards creating a reusable widget is to lay it out. In some cases, you may prefer to create your widget contents purely in Java, particularly if many copies of the widget will be created and you do not want to inflate them every time. However, otherwise, layout XML is simpler in many cases than the in-Java alternative.

Here is one such `Meter` layout (`res/layout/meter.xml` in `Views/Meter`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
>
  <ImageButton android:id="@+id/decr"
    android:layout_height="30px"
    android:layout_width="30px"
    android:src="@drawable/decr"
  />
  <ProgressBar android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="0px"
    android:layout_weight="1"
    android:layout_height="wrap_content"
  />
  <ImageButton android:id="@+id/incr"
    android:layout_height="30px"
    android:layout_width="30px"
    android:src="@drawable/incr"
  />
</LinearLayout>
```

All we do is line them up in a row, giving the `ProgressBar` any excess space (via `android:layout_width = "0px"` and `android:layout_weight = "1"`). We are using a pair of 16x16 pixel images from the Nuvola icon set for the increment and decrement button faces.

The Attribute Declarations

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute we specified on the `ImageButton` widgets in the layout

above. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file to specify them.

For example, here is the attributes file for `Meter`:

```
<resources>
  <declare-styleable name="Meter">
    <attr name="max" format="integer" />
    <attr name="incr" format="integer" />
    <attr name="decr" format="integer" />
  </declare-styleable>
</resources>
```

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute – in our case, we will call the widget `Meter`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the name of an attribute (e.g., `incr`) and what data type the attribute has (e.g., `integer`). The data type will help with compile-time validation and in getting any supplied values for this attribute parsed into the appropriate type at runtime.

Here, we indicate there are three attributes: `max` (indicating the highest value the `Meter` will go to), `incr` (indicating how much the value should increase when the increment button is clicked), and `decr` (indicating how much the value should decrease when the decrement button is clicked).

The Widget Implementation

There are many ways to go about actually implementing a widget. This section outlines one option: using a container class (specifically `LinearLayout`) and inflating the contents of your widget layout into the container.

The Constructor

To be usable inside of layout XML, you need to implement a constructor that takes two parameters:

1. A `Context` object, typically representing the `Activity` that is inflating your widget
2. An `AttributeSet`, representing the bundle of attributes included in the element in the layout being inflated that references your widget

In this constructor, after chaining to your superclass, you can do some basic configuration of your widget. Bear in mind, though, that you are not in position to configure the widgets that make up your aggregate widget – you need to wait until `onFinishInflate()` before you can do anything with those.

One thing you definitely want to do in the constructor, though, is use that `AttributeSet` to get the values of whatever attributes you defined in your `attrs.xml` file. For example, here is the constructor for `Meter`:

```
public Meter(final Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  this.setOrientation(HORIZONTAL);

  TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                           R.styleable.Meter,
                                           0, 0);

  max=a.getInt(R.styleable.Meter_max, 100);
  incrAmount=a.getInt(R.styleable.Meter_incr, 1);
  decrAmount=-1*a.getInt(R.styleable.Meter_decr, 1);

  a.recycle();
}
```

The `obtainStyledAttributes()` on `Context` allows us to convert the `AttributeSet` into useful values:

- It resolves references to other resources, such as strings
- It handles any styles that might be declared via the style attribute in the layout element that references your widget
- It finds the resources you declared via attrs.xml and makes them available to you

In the code shown above, we get our `TypedArray` via `obtainStyledAttributes()`, then call `getInt()` three times to get our values

out of the `TypedArray`. The `TypedArray` is keyed by `R.styleable` identifiers, so we use the three generated for us by the build tools for `max`, `incr`, and `decr`.

Note that you should call `recycle()` on the `TypedArray` when done – this makes this `TypedArray` available for immediate reuse, rather than forcing it to be garbage-collected.

Finishing Inflation

Your widget will also typically override `onFinishInflate()`. At this point, you can turn around and add your own contents, via Java code or, as shown below, by inflating a layout XML resource into yourself as a container:

```
@Override
protected void onFinishInflate() {
  super.onFinishInflate();

  ((Activity)getContext()).getLayoutInflater().inflate(R.layout.meter, this);

  bar=(ProgressBar)findViewById(R.id.bar);
  bar.setMax(max);

  ImageButton btn=(ImageButton)findViewById(R.id.incr);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      bar.incrementProgressBy(incrAmount);

      if (onIncr!=null) {
        onIncr.onClick(Meter.this);
      }
    }
  });

  btn=(ImageButton)findViewById(R.id.decr);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      bar.incrementProgressBy(decrAmount);

      if (onDecr!=null) {
        onDecr.onClick(Meter.this);
      }
    }
  });
}
```

Of course, once you have constructed or inflated your contents, you can configure them, particularly using the attributes you declared in attrs.xml and retrieved in your constructor.

Event Handlers and Other Methods

If you wish to expose events to the outside world – such as `Meter` exposing when the increment or decrement buttons are clicked – you need to do a few things:

- Choose or create an appropriate listener class or classes (e.g., `View.OnClickListener`)
- Hold onto instances of those classes as data members of the widget class
- Offer setters (and, optionally, getters) to define those listener objects
- Call those listeners when appropriate

For example, `Meter` holds onto a pair of `View.OnClickListener` instances:

```
private View.OnClickListener onIncr=null;
private View.OnClickListener onDecr=null;
```

It lets users of `Meter` define those listeners via getters:

```
public void setOnIncrListener(View.OnClickListener onIncr) {
  this.onIncr=onIncr;
}

public void setOnDecrListener(View.OnClickListener onDecr) {
  this.onDecr=onDecr;
}
```

And, as shown in the previous section, it passes along the button clicks to the listeners:

```
ImageButton btn=(ImageButton)findViewById(R.id.incr);

btn.setOnClickListener(new View.OnClickListener() {
```

```
  public void onClick(View v) {
    bar.incrementProgressBy(incrAmount);

    if (onIncr!=null) {
      onIncr.onClick(Meter.this);
    }
  }
});

btn=(ImageButton)findViewById(R.id.decr);

btn.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    bar.incrementProgressBy(decrAmount);

    if (onDecr!=null) {
      onDecr.onClick(Meter.this);
    }
  }
});
```

Note that we change the value passed in the `onClick()` method – our listener receives the `ImageButton`, but we pass the `Meter` widget on the outbound `onClick()` call. This is so we do not leak internal implementation of our widget. The users of `Meter` should neither know nor care that we have `ImageButton` widgets as part of the `Meter` internals.

Your widget may well require other methods as well, for widget-specific configuration or functionality, though `Meter` does not.

Using the Widget

Given all of that, using the `Meter` widget is not significantly different than using any other widget provided in the system...with a few minor exceptions.

In the layout, since your custom widget is not in the `android.widget` Java package, you need to fully-qualify the class name for the widget, as seen in the `main.xml` layout for the `Views/Meter` project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/com.commonsware.android.widget"
```

```
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:paddingTop="5px"
>
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Meter:"
  />
  <com.commonsware.android.widget.Meter
    android:id="@+id/meter"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    app:max="100"
    app:incr="1"
    app:decr="5"
  />
</LinearLayout>
```

You will also note that we have a new namespace (`xmlns:app = "http://schemas.android.com/apk/res/com.commonsware.android.widget"`), and that our custom attributes from above are in that namespace (e.g., `app:max`). The custom namespace is because our attributes are not official Android ones and so will not be recognized by the build tools in the `android:` namespace, so we have to create our own. The value of the namespace needs to be `http://schemas.android.com/apk/res/` plus the name of the package containing the styleable attributes (`com.commonsware.android.widget`).

With just the stock generated activity, we get the following UI: