IBM® WebSphere® Commerce V7 Feature Pack 4 – Lab exercise

Android app customization lab

What this exercise is about

In this tutorial, you will extend the existing sample Android native application. One of the features included in the desktop version of the Madisons store but not in the sample application is the concept of related products. You will learn to work with the application code by adding a button on the product display page to view related products.

In order to add this new capability, there are several areas of the Android application where you will update code. First, you will extend the Product object and JSON parsing code to capture the related product data. This information is returned by the REST service but is not currently stored in the application. Next, you will add a new button to the product display page to show the related products. To implement the new button, you will create a new activity and provide a layout for the related products page. Finally, you will add your new activity to the application manifest file.

This tutorial requires that you have an Android development environment set up. If you need instructions to complete this setup, you can find them in Appendix A of this document.

This tutorial should take approximately 90 minutes to complete.

What you should be able to do

After completing this exercise, you should be able to:

- Extend an Android business object
- Create and update page layouts
- Implement a new activity

Requirements

Before beginning this lab, ensure you have:

- Installed WebSphere Commerce V7 Fix Pack 5
- Installed WebSphere Commerce V7 Feature Pack 4
- Prepared an Android development environment
- Downloaded the AndroidAppCustomizationSnippets.txt file provided with this lab guide.

Part 1: Extend the Product object

In this section, you will extend the Product class to hold merchandising associations. The Android native application defines classes for each of the main business objects it operates on. The business object definitions are located in the com.ibm.commerce.android.nativeapp.mobile.objects package.

____ 1. Open the Product class.

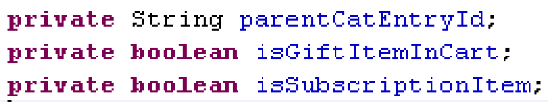
__ a. Launch Eclipse**.**

__ b. In the Package Explorer, navigate to the project containing the Android native application code.

__ c. Open the file **src > com.ibm.commerce.android.nativeapp.mobile.objects > Product.java**.

____ 2. Add new fields to store the merchandising association information for a product.

__ a. Scroll to the bottom of the existing field definitions



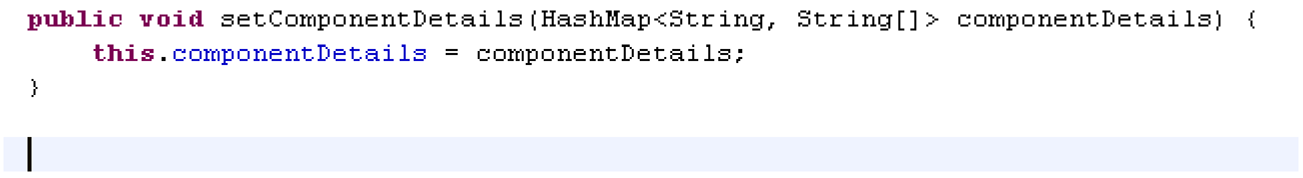
```
private String parentCatEntryId;
private boolean isGiftItemInCart;
private boolean isSubscriptionItem;
```

__ b. Add two new fields to the class as shown below. The first field stores the merchandising associations and the second stores the association type.

```
private Product[] relatedProducts;
private String massocType;
```

____ 3. Add getter and setter methods for the new fields.

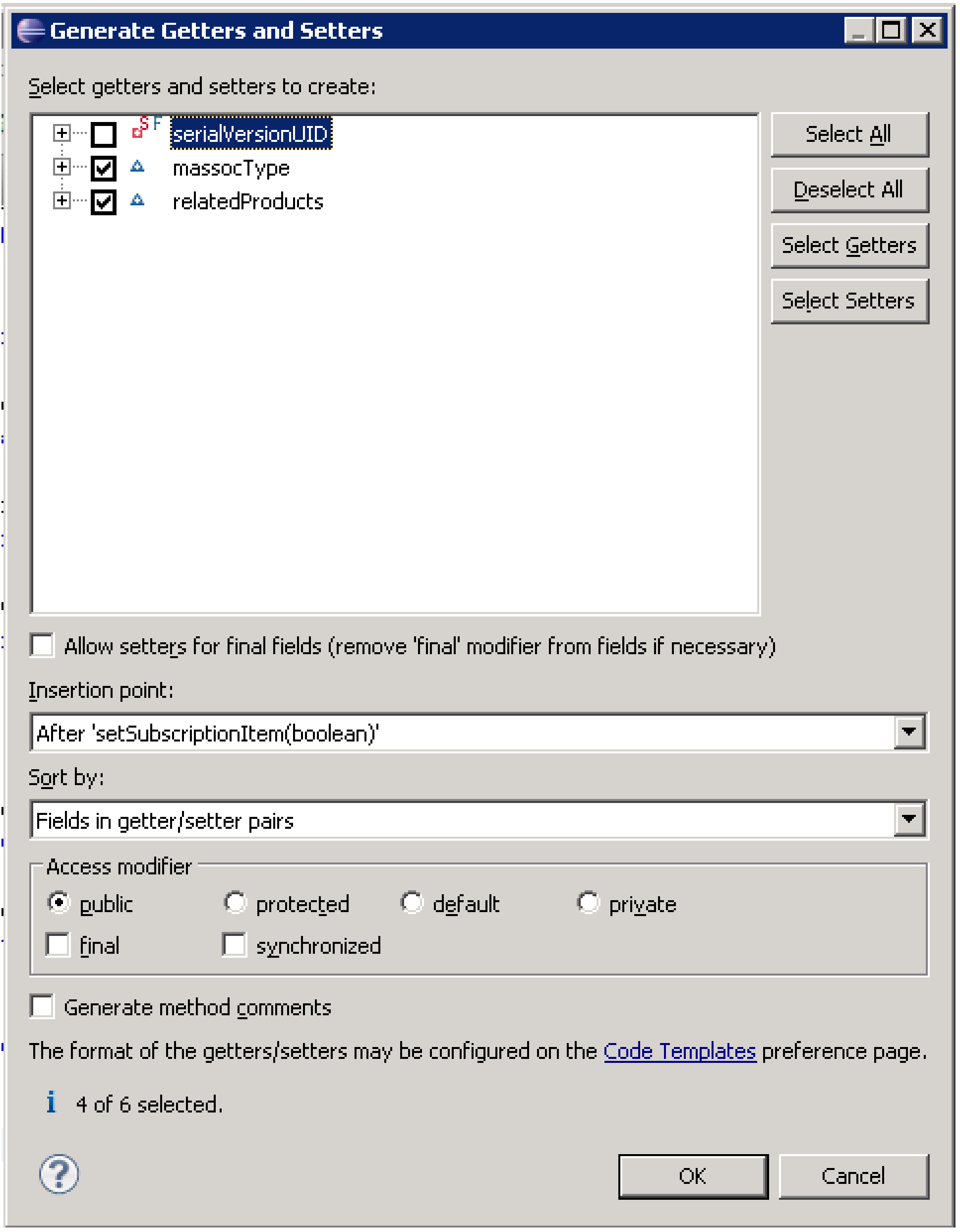
__ a. Scroll to the bottom on the class just below the last setter method.



```
public void setComponentDetails(HashMap<String, String[]> componentDetails) {
    this.componentDetails = componentDetails;
}
```

__ b. Right click and select **Source > Generate Getters and Setters** from the pop-up menu.

__ c. Select your two new fields and click **OK**.



Generate Getters and Setters

Select getters and setters to create:

- [ ] serialVersionUID
- [✓] massocType
- [✓] relatedProducts

- [ ] Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: After 'setSubscriptionItem(boolean)'

Sort by: Fields in getter/setter pairs

Access modifier: [✓ public] protected default private

- [ ] final
- [ ] synchronized
- [ ] Generate method comments

The format of the getters/setters may be configured on the Code Templates preference page.

4 of 6 selected.

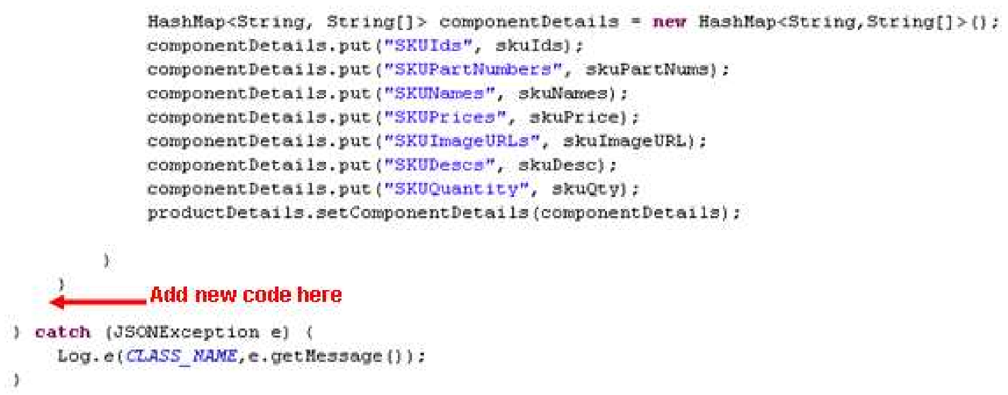
__ d. Save and close the file.

Part 2: Populate the new Product fields

In this section, you will extend the JSON parsing code to populate the new Product fields when products details are received. In the Android native application, the classes for handling REST services requests and responses are in the com.ibm.commerce.android.nativeapp.mobile.utils package.

____ 1. Using the Package Explorer, find and open the file **src > com.ibm.commerce.android.nativeapp.mobile.utils > JSONParser.java**.

____ 2. Locate the **getProductDetailsFromJSON** method. This is the method that populates the Product object when you navigate to the product display page in the store. Scroll to the bottom of the **try** block.



```
        HashMap<String, String[]> componentDetails = new HashMap<String,String[]>();
        componentDetails.put("SKUIds", skuIds);
        componentDetails.put("SKUPartNumbers", skuPartNums);
        componentDetails.put("SKUNames", skuNames);
        componentDetails.put("SKUPrices", skuPrice);
        componentDetails.put("SKUImageURLs", skuImageURL);
        componentDetails.put("SKUDescs", skuDesc);
        componentDetails.put("SKUQuantity", skuQty);
        productDetails.setComponentDetails(componentDetails);

    }
  }
} catch (JSONException e) {
  Log.e(CLASS_NAME,e.getMessage());
}
```

____ 3. Add the code below to populate the merchandising association information for the product. This information was already being passed in the service response, it just was not being captured. A text version of the code can be found in the snippets file provided with this lab.

```
// Get merchandising associations
JSONArray massocs = productObj.optJSONArray("MerchandisingAssociations");
if(massocs != null){
    int numOfAssoc = massocs.length();
    Product[] relatedProducts = new Product[numOfAssoc];
    for (int i = 0; i < numOfAssoc; i++) {
        JSONObject massoc = (JSONObject) massocs.get(i);
        Product relatedProduct = new Product();
        relatedProduct.setShortDesc(massoc.optString(ApplicationConstants.SHORT_DESCRIPTION_TAG));
        relatedProduct.setName(massoc.getString(ApplicationConstants.NAME_TAG));
        relatedProduct.setThumbNail(massoc.optString(ApplicationConstants.THUMBNAIL_TAG));
        relatedProduct.setPartNumber(massoc.getString(ApplicationConstants.PART_NUMBER_TAG));
        relatedProduct.setMassocType(massoc.getString("type"));
        JSONArray relProdPriceArray = massoc.optJSONArray("Price");
        if(relProdPriceArray != null){
            for(int temp=0; temp < relProdPriceArray.length(); temp++){
                JSONObject priceObj = relProdPriceArray.optJSONObject(temp);
                if(priceObj != null){
                    if(priceObj.getString("priceUsage").equalsIgnoreCase(ApplicationConstants.DISPLAY_TAG)){
                        relatedProduct.setListPrice(priceObj.getString(ApplicationConstants.PRICE_VALUE_TAG));
                    } else if(priceObj.getString("priceUsage").equalsIgnoreCase("Offer")){
                        relatedProduct.setContractPrice(priceObj.getString(ApplicationConstants.PRICE_VALUE_TAG));
                    }
                }
            }
        }
        relatedProducts[i] = relatedProduct;
    }
    productDetails.setRelatedProducts(relatedProducts);
```

____ 4. Save and close the file.

Part 3: Add a new button to the product display page

In this part of the lab, you will update the product display page so that shoppers can click a button and open a new page to see products related to the one they are currently viewing. In the Android native application, the page layout is specified in an XML file and the logic is provided by a Java Activity class. Layouts are stored in the resource folder and activities are in packages arranged by the store function they provide. For example catalog activities are in the com.ibm.commerce.android.nativeapp.mobile.activity.catalog package.

____ 1. Add a new text string for the button label.

__ a. Using the Package Explorer, find and open the file **res > values > strings.xml**. This is the main file for store labels. Translated versions of this file can be found in the corresponding locale-specific **values** folders. For example, values-fr.

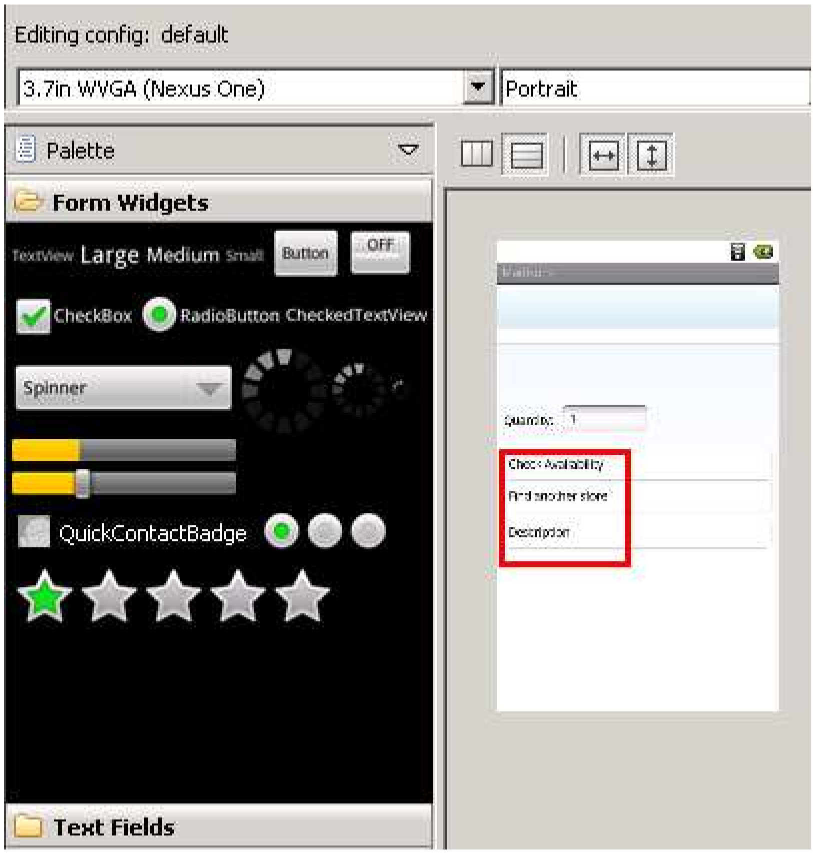
__ b. Add this string resource to the file:

```
<string name="RelatedProducts">Related Products</string>
```

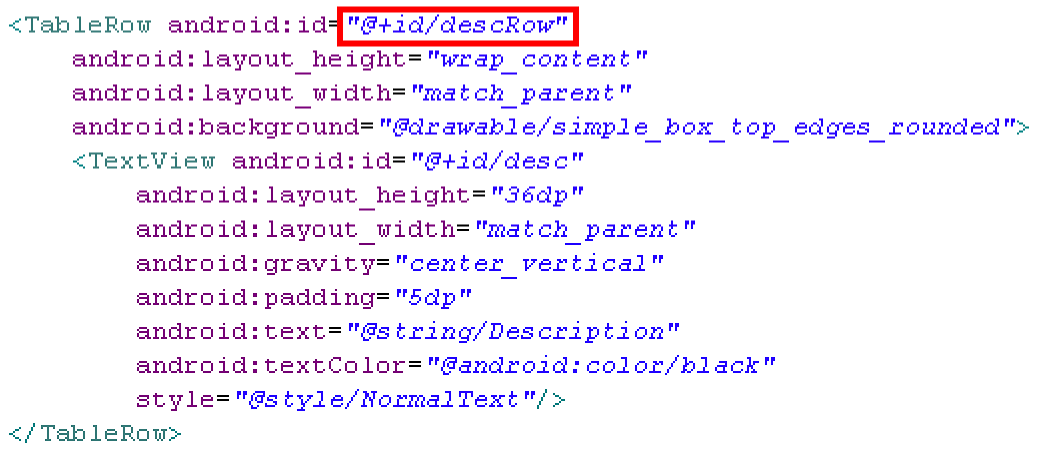
__ c. Save and close the file.

____ 2. Add the button to the product display layout.

__ a. Using the Package Explorer, find and open the file **res > layout > catalog_product_details.xml**. Layout files can be viewed in a graphical editor or in XML format. In the graphical view, you can see the three existing buttons on the page for check availability, find another store and description.

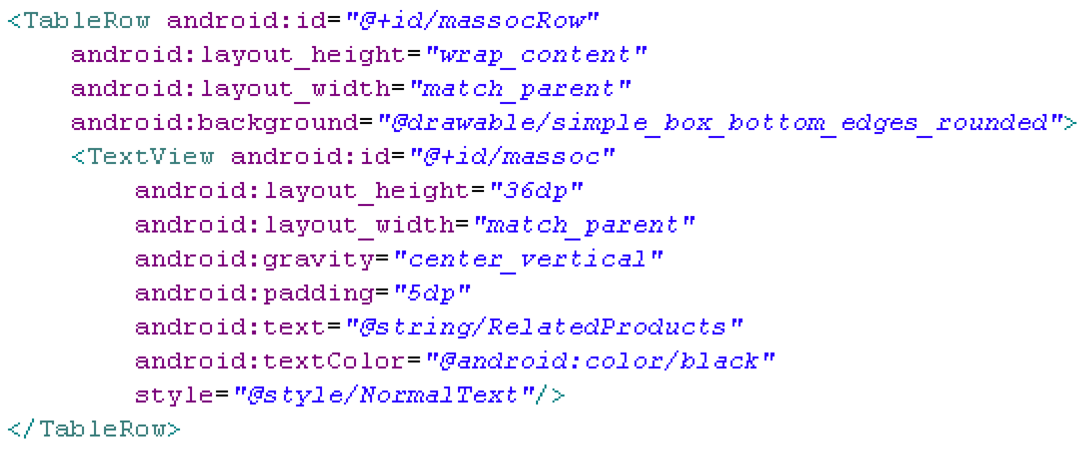


__ b. Switch to the XML view of the file and locate the section that displays the **Description** button as shown in the screen capture below. You can find this section easily by searching for the string **id/descRow**.



```
<TableRow android:id="@+id/descRow"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:background="@drawable/simple_box_top_edges_rounded">
    <TextView android:id="@+id/desc"
        android:layout_height="36dp"
        android:layout_width="match_parent"
        android:gravity="center_vertical"
        android:padding="5dp"
        android:text="@string/Description"
        android:textColor="@android:color/black"
        style="@style/NormalText"/>
</TableRow>
```

__ c. Add a new row for related products to the table immediately below the description using the code shown below. The text version can be found in the snippets file.



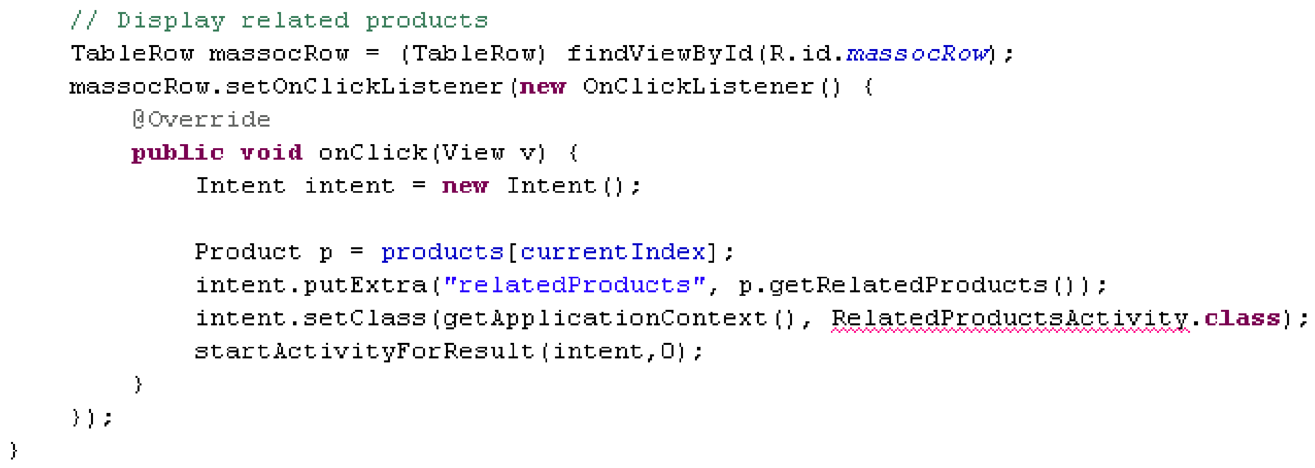
```
<TableRow android:id="@+id/massocRow"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:background="@drawable/simple_box_bottom_edges_rounded">
    <TextView android:id="@+id/massoc"
        android:layout_height="36dp"
        android:layout_width="match_parent"
        android:gravity="center_vertical"
        android:padding="5dp"
        android:text="@string/RelatedProducts"
        android:textColor="@android:color/black"
        style="@style/NormalText"/>
</TableRow>
```

__ d. Save and close the file.

____ 3. Add logic to the button to display the related products.

__ a. Using the Package Explorer, find and open the file **src > com.ibm.commerce.android.nativeapp.mobile.activity.catalog > ProductDetails.java**. This class implements an Android activity that provides the product details displayed on the page and defines the page interactions.

__ b. Scroll to the bottom of the **onCreate**() method and add the code below just before the closing bracket. This code adds a listener to the new table row you just added to the page layout. When a shopper clicks the related products button, a new intent is created. The intent specifies the action to take (the activity class) and stores data that will available to the activity. The text version can be found in the snippets file. You can learn more about intents on the Android developer site: http://developer.android.com/guide/topics/intents/intents-filters.html



```
        // Display related products
        TableRow massocRow = (TableRow) findViewById(R.id.massocRow);
        massocRow.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent();

                Product p = products[currentIndex];
                intent.putExtra("relatedProducts", p.getRelatedProducts());
                intent.setClass(getApplicationContext(), RelatedProductsActivity.class);
                startActivityForResult(intent,0);
            }
        });
    }
```

__ c. Save and close the file. You can ignore the error about the RelatedProductsActivity class. You will create this in the next part.

Part 4: Create the related products display page

In this part of the lab, you will create a new XML layout and activity class to display the related products. Before completing this part, you might find it helpful to review layouts and activities on the Android developer site.

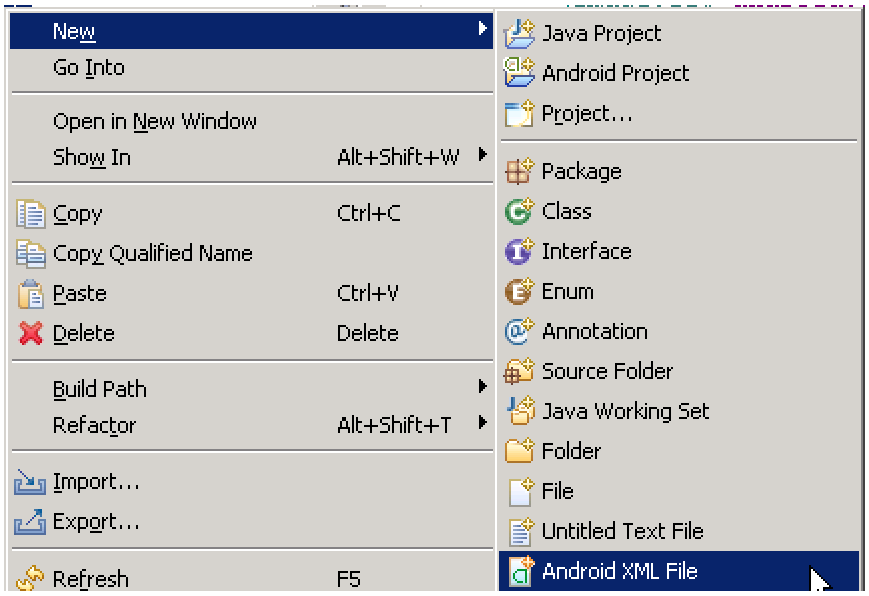
Layouts: http://developer.android.com/guide/topics/ui/declaring-layout.html

Activities: http://developer.android.com/guide/topics/fundamentals/activities.html

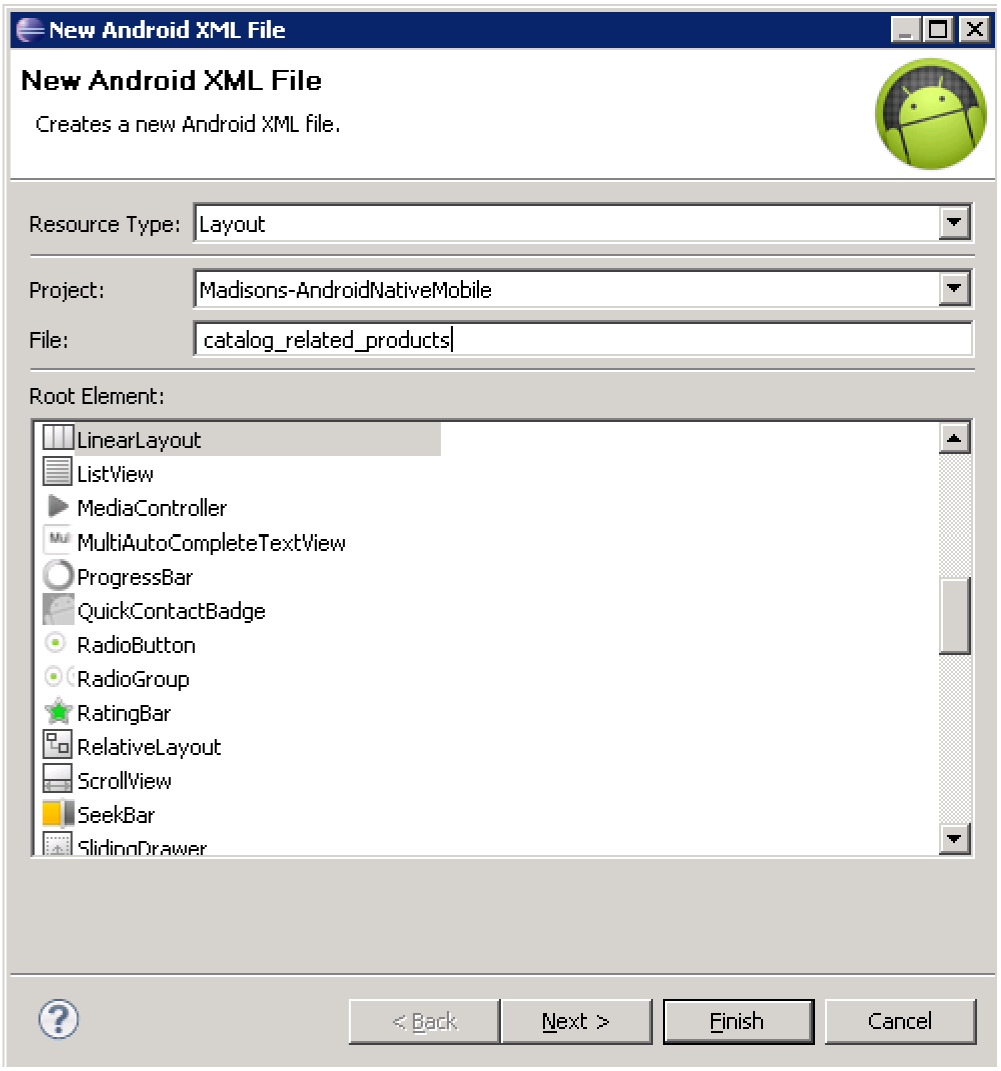
____ 1. Create the related products page layout.

__ a. Using the Package Explorer, navigate to **res > layout.**

__ b. Right click and select **New > Android XML File**.

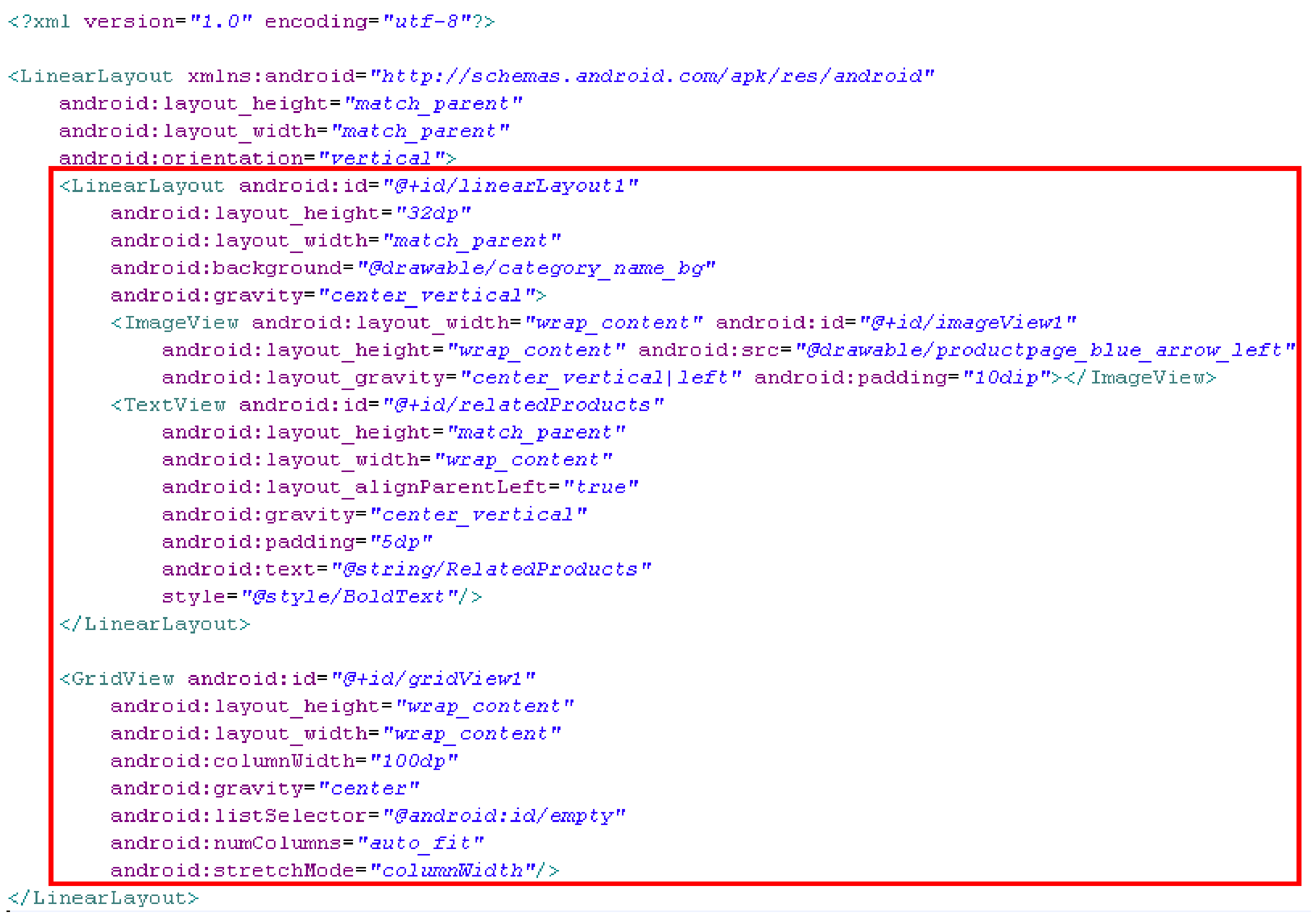


__ c. In the file creation dialog, name the file **catalog_related_products.xml**. Notice that **Layout** is already selected as the type of resource to create and **LinearLayout** is selected by default. Keep these default values and click **Finish**.



__ d. When the new file opens, switch to the XML view.

__ e. Inside the LinearLayout element that was created by default, add the layout definition for your page as shown in the highlighted area below. The text version can be found in the snippets file. The first section defines the page header with an arrow to go back to the previous page, followed by the page title. Notice the title is reusing the resource string you created in the previous part. The second section defines the related product display area. Products are shown in a grid layout.



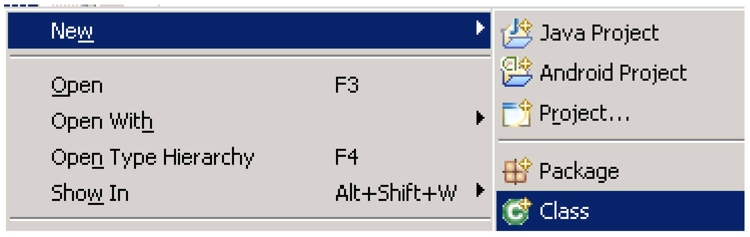
```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="vertical">
    <LinearLayout android:id="@+id/linearLayout1"
        android:layout_height="32dp"
        android:layout_width="match_parent"
        android:background="@drawable/category_name_bg"
        android:gravity="center_vertical">
        <ImageView android:layout_width="wrap_content" android:id="@+id/imageView1"
            android:layout_height="wrap_content" android:src="@drawable/productpage_blue_arrow_left"
            android:layout_gravity="center_vertical|left" android:padding="10dip"></ImageView>
        <TextView android:id="@+id/relatedProducts"
            android:layout_height="match_parent"
            android:layout_width="wrap_content"
            android:layout_alignParentLeft="true"
            android:gravity="center_vertical"
            android:padding="5dp"
            android:text="@string/RelatedProducts"
            style="@style/BoldText"/>
    </LinearLayout>

    <GridView android:id="@+id/gridView1"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:columnWidth="100dp"
        android:gravity="center"
        android:listSelector="@android:id/empty"
        android:numColumns="auto_fit"
        android:stretchMode="columnWidth"/>
</LinearLayout>
```

__ f. Save the file. Before you close it, switch back to the **Graphical Layout** tab to see your new page layout.

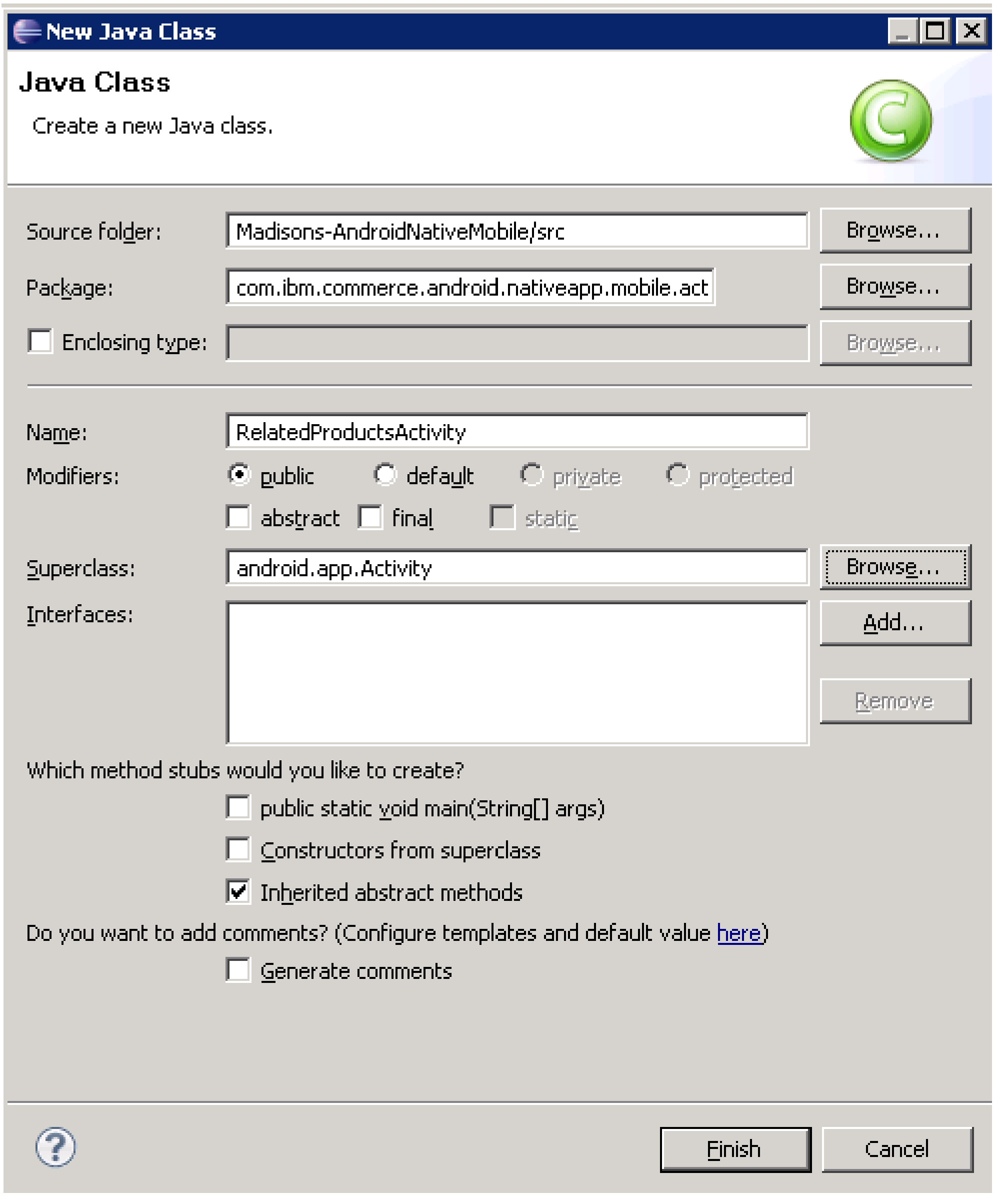
____ 2. Create a new activity for the related products page.

__ a. Using the Package Explorer, navigate to **src > com.ibm.commerce.android.nativeapp.mobile.activity.catalog**

__ b. Right click and select **New > Class**.



__ c. Name the class **RelatedProductsActivity** and set the superclass to **android.app.Activity**. Click **Finish**.



Source folder: Madisons-AndroidNativeMobile/src

Package: com.ibm.commerce.android.nativeapp.mobile.act

☐ Enclosing type:

Name: RelatedProductsActivity

Modifiers: [✓ public] default private protected

☐ abstract ☐ final ☐ static

Superclass: android.app.Activity

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

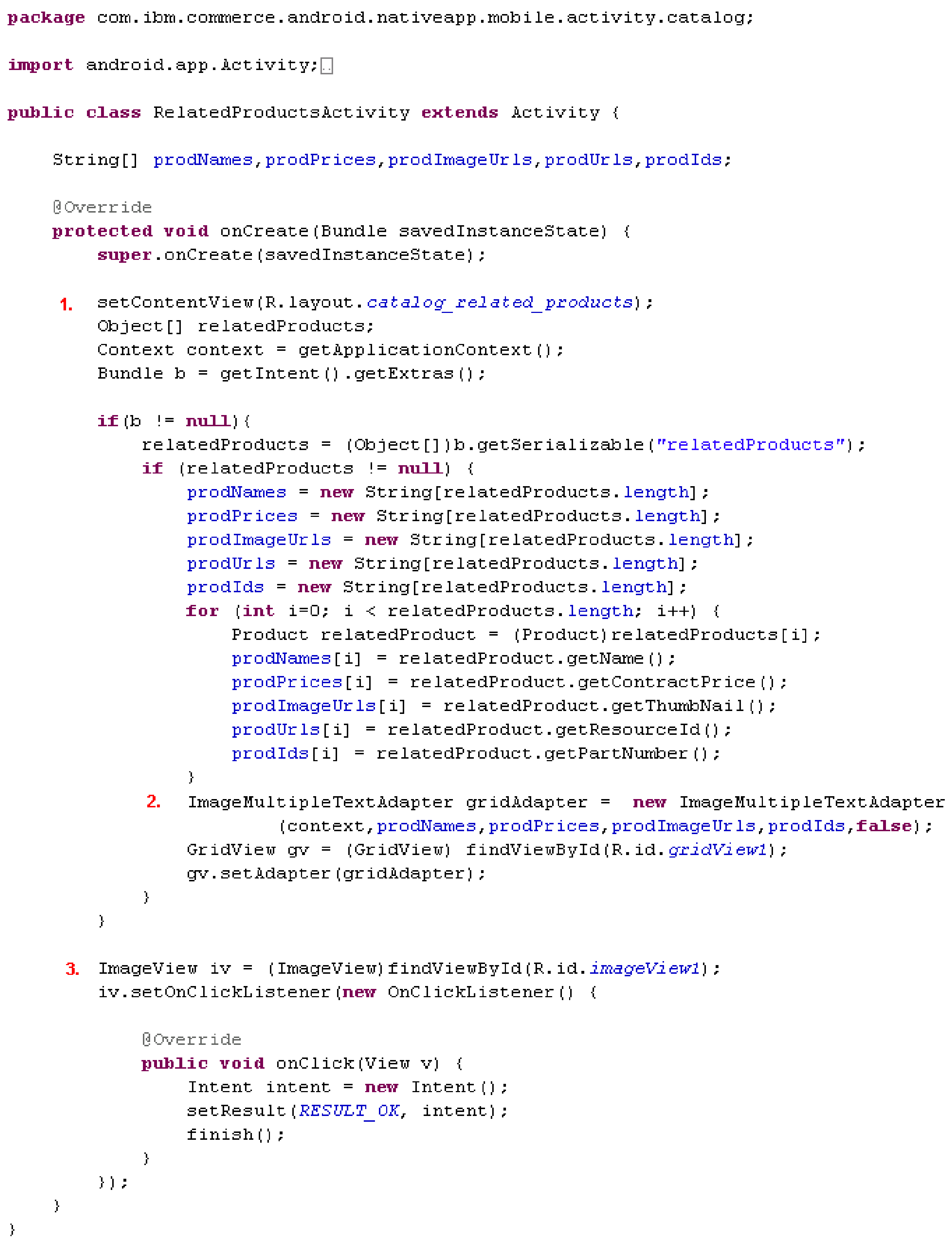
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

__ d. Implement the class as shown in the screen capture below. The full text for the class can be found in the snippets file. Three areas are marked with red numbers below. The number one indicates where the layout for this activity is defined. The R class is auto-generated. The **catalog_related_products** field maps to the **catalog_related_products.xml** file you created in step 1. The number two indicates where the related product information is associated with the grid view in the layout. A helper class is used to handle the layout. Helper UI classes are found in the com.ibm.commerce.android.nativeapp.mobile.ui.utils package. The number three shows the onClick listener that is added to the back arrow image to return to the product display page. When the back arrow is clicked, the intent that was created to launch the related products page is marked complete and the application returns to the product display page.



```
package com.ibm.commerce.android.nativeapp.mobile.activity.catalog;

import android.app.Activity;

public class RelatedProductsActivity extends Activity {

    String[] prodNames,prodPrices,prodImageUrls,prodUrls,prodIds;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

 1.     setContentView(R.layout.catalog_related_products);
        Object[] relatedProducts;
        Context context = getApplicationContext();
        Bundle b = getIntent().getExtras();

        if(b != null){
            relatedProducts = (Object[])b.getSerializable("relatedProducts");
            if (relatedProducts != null) {
                prodNames = new String[relatedProducts.length];
                prodPrices = new String[relatedProducts.length];
                prodImageUrls = new String[relatedProducts.length];
                prodUrls = new String[relatedProducts.length];
                prodIds = new String[relatedProducts.length];
                for (int i=0; i < relatedProducts.length; i++) {
                    Product relatedProduct = (Product)relatedProducts[i];
                    prodNames[i] = relatedProduct.getName();
                    prodPrices[i] = relatedProduct.getContractPrice();
                    prodImageUrls[i] = relatedProduct.getThumbNail();
                    prodUrls[i] = relatedProduct.getResourceId();
                    prodIds[i] = relatedProduct.getPartNumber();
                }
        2.      ImageMultipleTextAdapter gridAdapter =  new ImageMultipleTextAdapter
                        (context,prodNames,prodPrices,prodImageUrls,prodIds,false);
                GridView gv = (GridView) findViewById(R.id.gridView1);
                gv.setAdapter(gridAdapter);
            }
        }

 3.     ImageView iv = (ImageView)findViewById(R.id.imageView1);
        iv.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                Intent intent = new Intent();
                setResult(RESULT_OK, intent);
                finish();
            }
        });
    }
}
```

__ e. Save and close the file.

____ 3. Register the new activity in the manifest file.

__ a. Open the file **AndroidManifest.xml** and scroll to the bottom on the file**.**

__ b. Just before the **</application>** tag, add your new activity.

```
<activity
   android:name="com.ibm.commerce.android.nativeapp.mobile.
activity.catalog.RelatedProductsActivity"
   android:configChanges="locale|orientation|keyboard"
   android:screenOrientation="portrait">
</activity>
```

__ c. Save and close the file.

Part 5: Test your new page

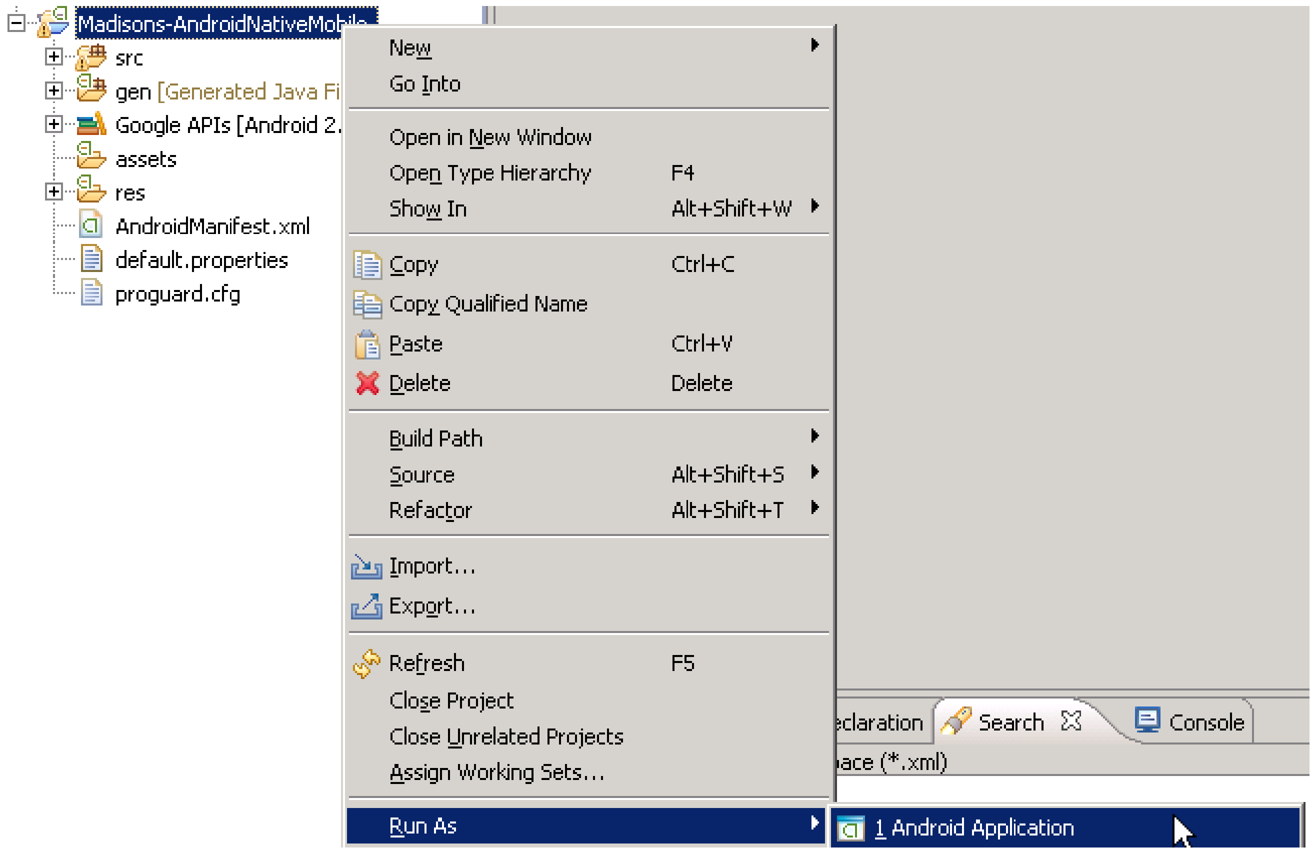
In this part of the lab, you will launch the Android native application and navigate to the new related products page.

____ 1. Launch the application.

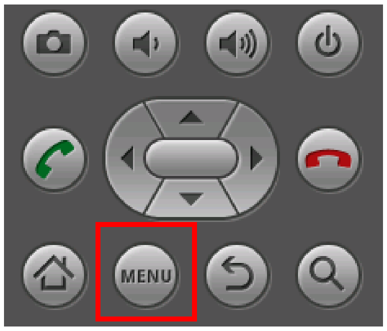
__ a. If you have previously run the Android native application on the emulator, you must uninstall it before you can test your changes. Within the emulator, select **Settings > Applications > Manage Applications** from the main menu to uninstall the existing application.

__ b. Start your WebSphere Commerce test server.

__ c. In Eclipse, right click the native application project and select **Run As > Android Application** from the pop-up menu.

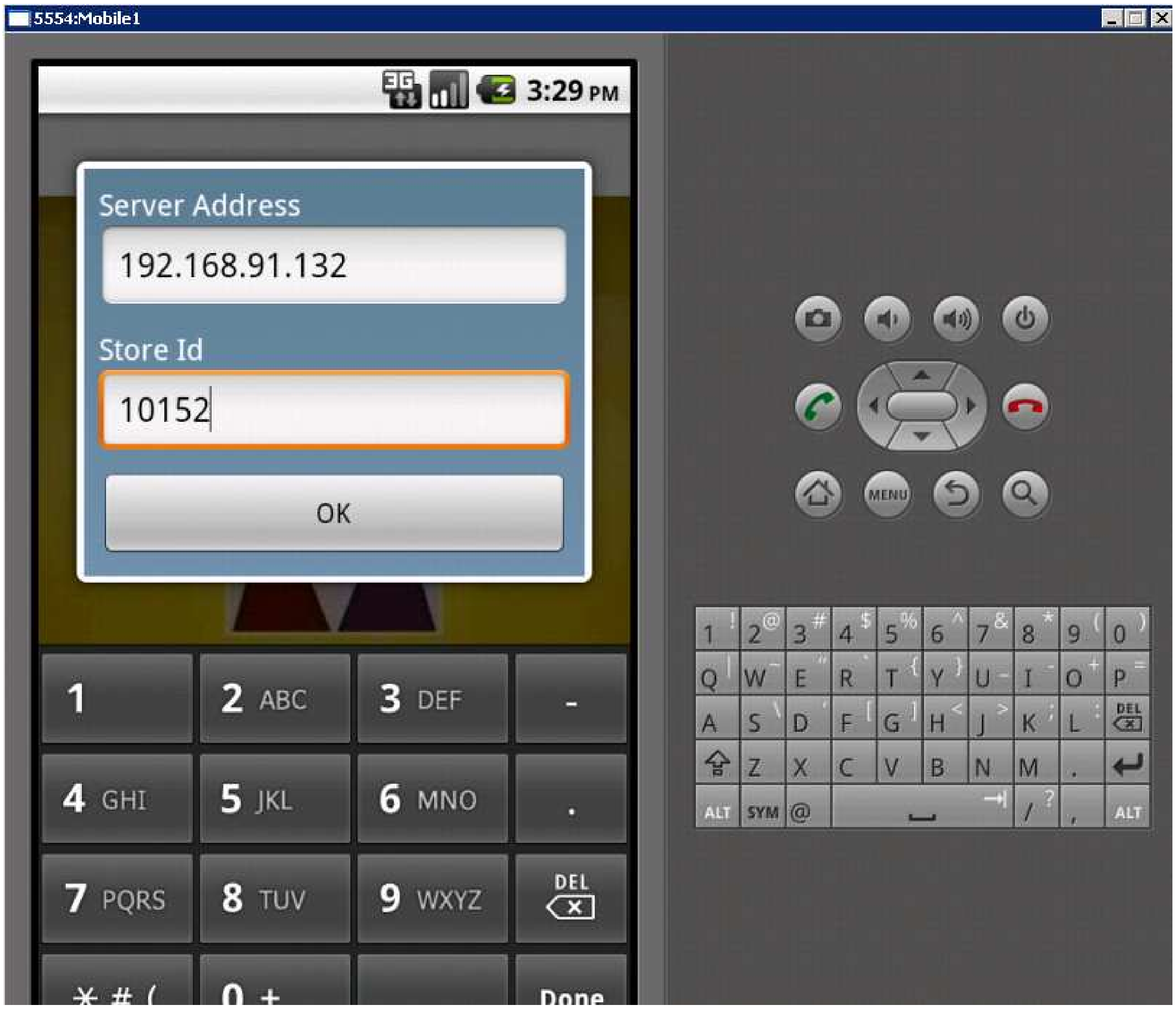


__ d. When the emulator has finished starting up, click the **Menu** button to switch the view to the running application.



____ 2. Test your changes.

__ a. The first time you launch the application, you are asked to provide connection details. Enter the IP address for the machine you are testing on and the store ID for your Madisons store. The screen capture below shows an example. Your values might be different.

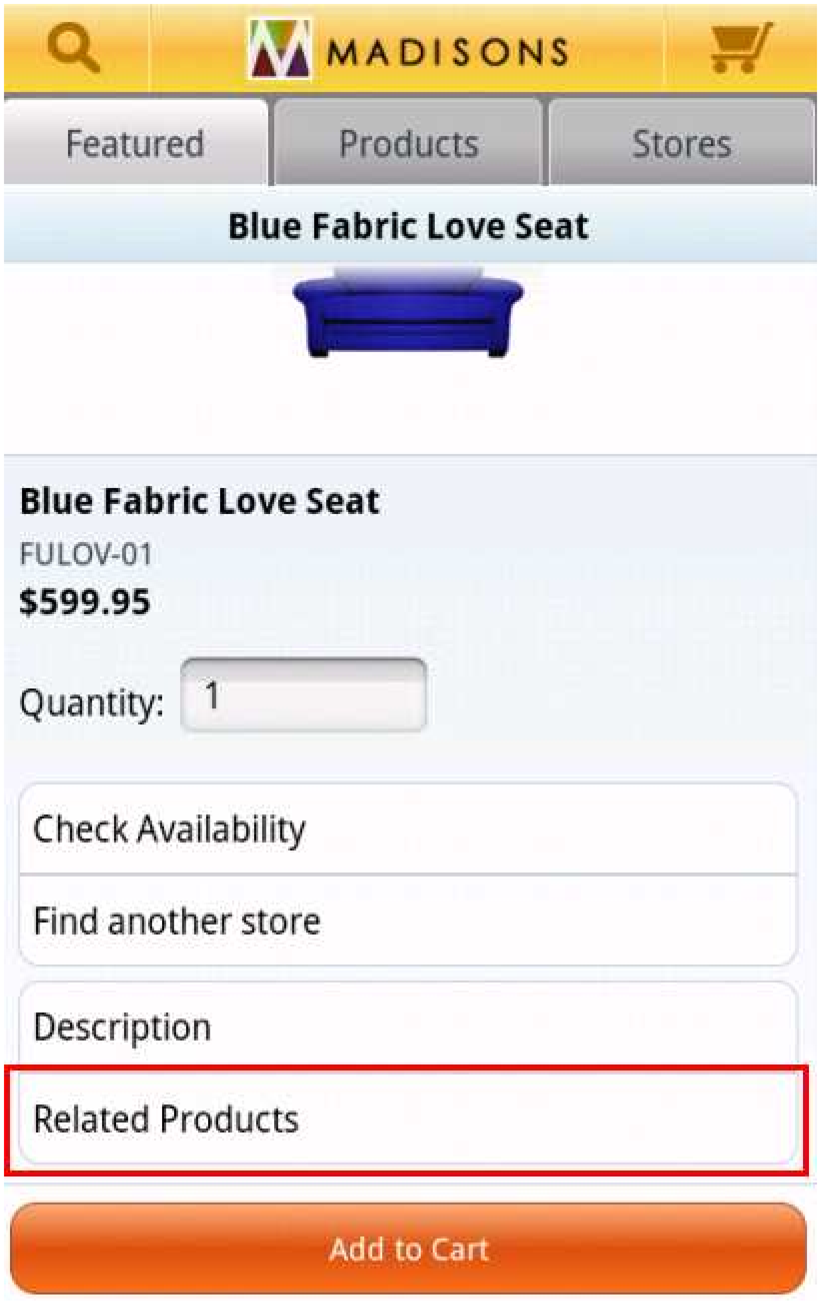

5554:Mobile1

Server Address: 192.168.91.132

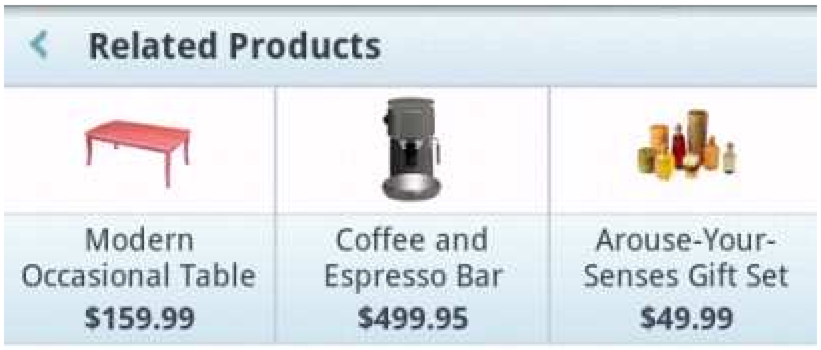
Store Id: 10152

__ b. On the application home page, select the **Blue Fabric Love Seat**.

__ c. Once the product display page loads, you should see the new **Related Products** button.



__ d. Click the Related Products button and you should see the new page you created.



__ e. Select the back arrow in the upper left to return to the product display page.

Part 6: What you did in this exercise

In this tutorial you learned how to work with the sample Android native application. You extended the product business object and created and modified layouts and activities to add a new page to the application.

You should now understand how to complete these tasks:

- Extend an Android business object
- Create and update page layouts
- Implement a new activity

Appendix A: Setting up an Android development environment

This section will cover the steps required to test the sample Android applications in the Android SDK emulator. This is an optional setup step that can aid in testing if you do not have an Android smart phone to test the native and hybrid applications.

____ 1. Install Eclipse.

__ a. The Android Development Tools are packaged as an Eclipse plug-in. This plug-in is not compatible with the IBM Rational tools and should be installed on a base Eclipse platform.

__ b. Download and install Eclipse from here: http://eclipse.org/downloads/

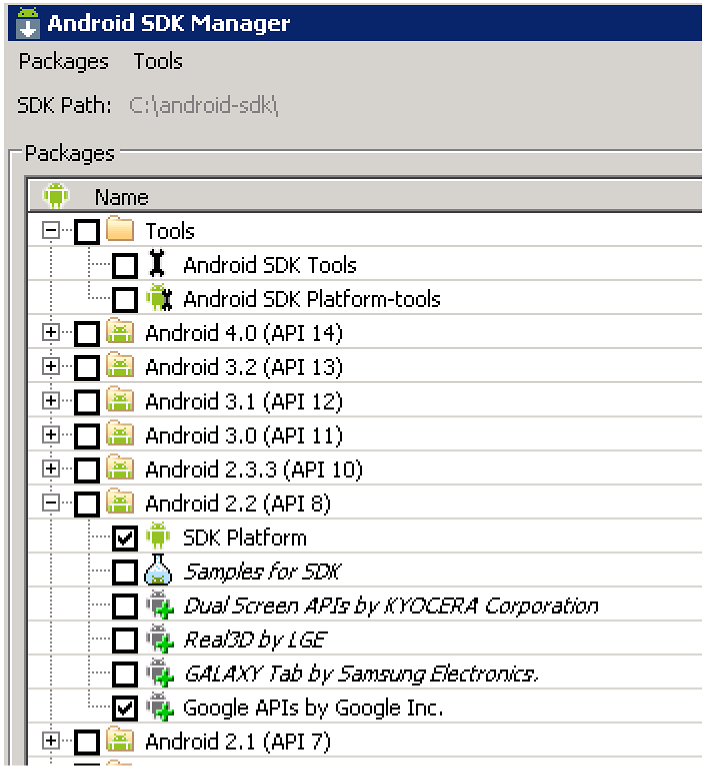
____ 2. Install the Android SDK.

__ a. Download the SDK starter package from the Android developers site: http://developer.android.com/sdk/index.html

__ b. Follow the instructions here http://developer.android.com/sdk/installing.html to install the SDK and the ADT plug-in.

1) Make note of the directory where you installed the SDK. It is referred to later as <AndroidSDK_InstallDir>.

2) When you reach step 4, **Adding Platforms and Other Components**, install the Android API 8 platform and the Google APIs as shown in the screen capture below.



SDK Path: C:\android-sdk\

Packages

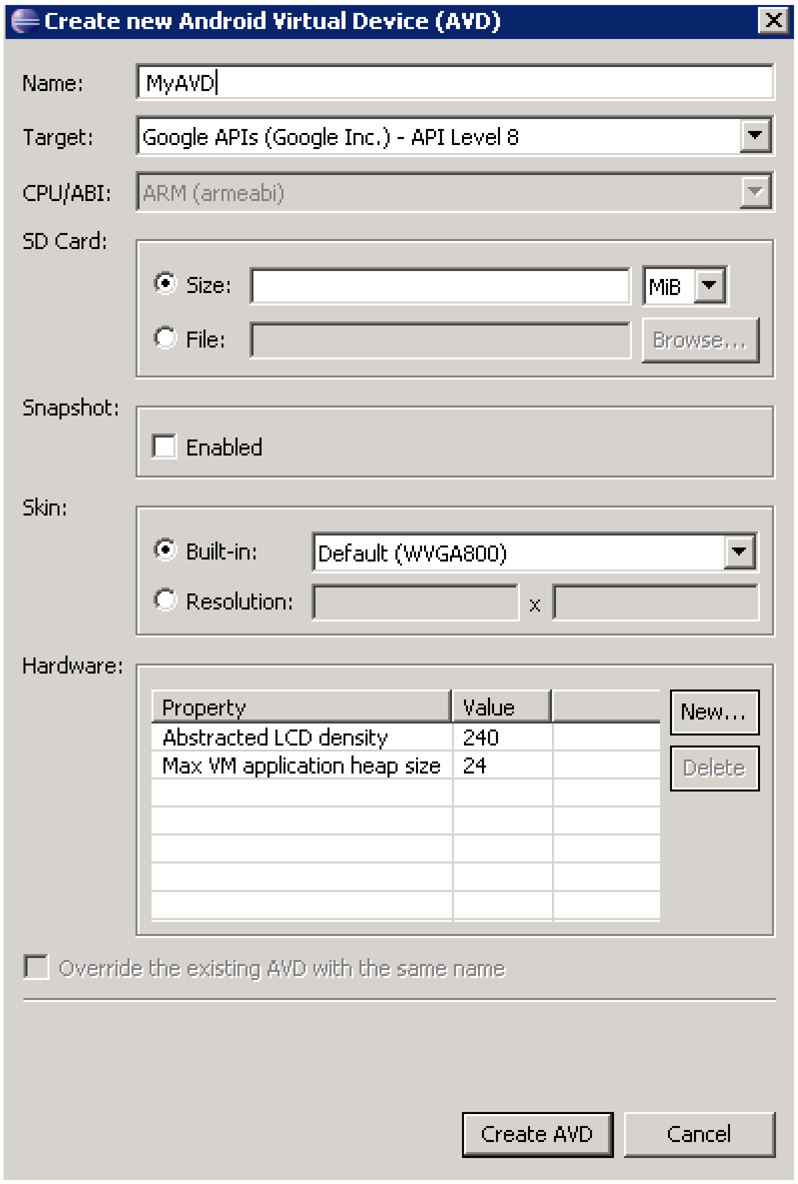
Name
☐ Tools
☐ Android SDK Tools
☐ Android SDK Platform-tools
☐ Android 4.0 (API 14)
☐ Android 3.2 (API 13)
☐ Android 3.1 (API 12)
☐ Android 3.0 (API 11)
☐ Android 2.3.3 (API 10)
☐ Android 2.2 (API 8)
☑ SDK Platform
☐ *Samples for SDK*
☐ *Dual Screen APIs by KYOCERA Corporation*
☐ *Real3D by LGE*
☐ *GALAXY Tab by Samsung Electronics.*
☑ Google APIs by Google Inc.
☐ Android 2.1 (API 7)

____ 3. Create an Android virtual device (AVD).

__ a. Launch the AVD Manager from **Window > AVD Manager.**

__ b. Select **New…** to create a new virtual device.

__ c. Give your AVD a name and select **Google APIs (Google Inc.) – API Level 8** from the target drop down.



Name: MyAVD

Target: Google APIs (Google Inc.) - API Level 8

CPU/ABI: ARM (armeabi)

Size: MiB

File:

Enabled

Built-in: Default (WVGA800)

Resolution: x

Property	Value
Abstracted LCD density	240
Max VM application heap size	24

Override the existing AVD with the same name

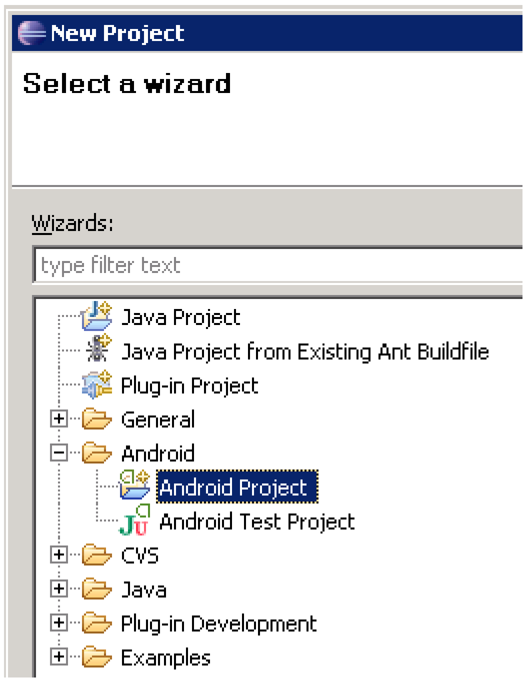
__ d. Click **Create AVD**.

____ 4. Import the sample Android hybrid application.

__ a. Create a new Android project for the hybrid application code.

1) From the Eclipse menu, select **File > New > Project…**

2) Expand the **Android** folder and select **Android Project.**


New Project

Select a wizard

Wizards:

type filter text

- Java Project
- Java Project from Existing Ant Buildfile
- Plug-in Project
- General
- Android
  - Android Project
  - Android Test Project
- CVS
- Java
- Plug-in Development
- Examples

3) Complete the Android Project properties with values similar to those shown in the table.

Field	Value
Project name	Madisons-AndroidHybridMobile
Build Target	Google APIs 2.2 API 8
Package name	com.ibm.commerce.android.hybrid.mobile
Create Activity	unchecked

4) Click **Finish** to create the project.

__ b. Extract the sample application code.

1) Navigate to **<WC_installdir>\components\store-enhancements\samples\Android**

2) Expand the compressed file **Madisons-AndroidHybridMobile.zip** to a temporary directory.

__ c. Import the sample application into Eclipse.

1) In Eclipse, right click the **MadisonsHybrid** project you created and select **Import…**

2) In the Import dialog, expand **General** and select **File System**. Click **Next >**.

3) Browse to the location where you expanded the compressed file.

4) Select the **Madisons-AndroidHybridMobile** folder to import all files. Click **Finish**.

5) When prompted to overwrite file, select **Yes to All**.

__ d. Verify the Java compiler level.

1) Right click the **Madisons-AndroidHybridMobile** project and select **Properties.**

2) Select **Java Compiler** from the properties list on the left.

3) Ensure the **Compiler compliance level** drop down is set to **1.6**.

4) Click **OK** to save your changes and **Yes** when prompted to rebuild the project.

____ 5. Import the sample Android native application. This process is similar to importing the hybrid application. The high level steps are provided below. For more details, review step 1 above.

__ a. Create a new Android project for the native application code.

1) Set the package name to com.ibm.commerce.android.nativeapp.mobile.activity

__ b. Extract the sample application code.

1) Sample code is located in **<WC_installdir>\components\store-enhancements\samples\Android**

__ c. Import the sample application into Eclipse.

__ d. Verify the Java compiler level is set to 1.6.