Advanced Mobile Application Development
Week 10: List Views and Adapters

Adapter Views

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses AdapterView such as spinners, list views, and grid views to populate the layout with views at runtime.
https://developer.android.com/guide/topics/ui/declaring-layout.html#AdapterViews

Adapters

Layouts that subclass the AdapterView class use an Adapter to bind data to its layout. Adapters provide a common interface into different kinds and sources of data. Android adapters are built to feed data from all sorts of sources(such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

Android has many different types of adapters

- **ArrayAdapter** binds arrays to an adapter view
  - can be used with any subclass of the **AdapterView** class
- To use an array adapter
  - Initialize the array adapter
  - Attach the array adapter to the list view using the **ListView setAdapter()** method
- The array adapter does the following:
  - Creates a view for each item in an array using the **toString()** method
  - Puts the contents in a text view
  - Displays each text view as a single row in the list view
- You can also define custom adapters

List Views

List Views are used to display a scrollable, vertical collection of data

- ListView class https://developer.android.com/reference/android/widget/ListView.html
- You can add a list view to your layout using the **<ListView>** element

List Activity

https://developer.android.com/reference/android/app/ListActivity.html
A **ListActivity** is a special type of activity that only has a list view in it.

- It has a default layout file so you don't need to create one
- The **ListActivity** class implements its own event listener so you don't need to create one

Use the **android:entries** property to display static data stored in a string-array in strings.xml
For nonstatic data stored in a Java array you need to use an adapter

Listeners

The **OnItemClickListener** is an event listener class in the **AdapterView** class

- The **onItemClick()** method is called when an item is clicked

To implement an event listener

- Create the **OnItemClickListener**
- Implement **onItemClick()**
- Attach the **OnItemClickListener** to the list view using the **ListView setOnItemClickListener()** method

Attaching the listener to the list view is what allows the listener to get notified when the user clicks on items in the list view.

Because the **ListActivity** class already implements an event listener you don't need to create one or bind it to the list view, you just need to implement the **ListActivity onListItemClick()** method

Passing Data
List activities are often used to display data that when a user clicks will start another activity with detail data
You do this by creating an intent to start the new activity in **onListItemClick()**

- pass the ID of the item clicked in the intent so the detail activity can get the correct detail data to populate its views
- call getItemAtPosition(position) to access the data associated with the selected item

Spring
Create a new project called Spring
Company name: a qualifier that will be appended to the package name
Package name: the fully qualified name for the project
Check Phone and Tablet, leave the rest unchecked
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Empty Activity template
Activity name: BulbMainActivity
Check Generate Layout File
Layout Name: activity_bulb_main
Uncheck Backwards Compatibility (uses Activity as the base class instead of AppCompatActivity)

Images
To add images go into the Project view.
Navigate to app/src/main/res
Drag images into the drawable folder (or copy/paste)
This project has 16 bulb images.
These are now available through the R class that Android automatically creates.
R.drawable.*imagename*

Bulb Layout
activity_bulb_main.xml
Make sure Autoconnect(magnet) is turned on.
Remove the textview
Add an ImageView to the top of the view and chose the bulbs image.
android:src="@drawable/bulbs"

If you want it right up to the top with no gap add
android:adjustViewBounds="true"

Having a content description for an image is optional but makes your app more accessible.
In your strings.xml add the text for this string
<**string name="bulbs**">Bulbs</**string**>
Set up the values for the list view in strings.xml
<**string-array name="bulb_types**">
  <**item**>Tulips</**item**>
  <**item**>Daffodils</**item**>

```
    <item>Iris</item>
</string-array>
```

Add to activity_bulb_main.xml
android:contentDescription="@string/bulbs"

Add a ListView in activity_bulb_main.xml and add
android:id="@+id/listView"
android:entries="@array/bulb_types"

If your ListView doesn't have any constraints it will be positioned at (0,0) which you don't want.
Select the ListView and chose Infer Constraints.
Now you should be able to run it and see your values in the list view.

We bind data to the list view using the android:entries attribute in our layout XML because the data is static. If your data is not static you use an adapter to act as a bridge between the data source and the list view.

ListView Listener
You can only use the android:onClick attribute in activity layouts for buttons, or any views that are subclasses of Button such as CheckBoxes and RadioButtons.
The ListView class isn't a subclass of Button, so using the android:onClick attribute won't work. To handle click events you have to implement OnItemClickListener.
In BulbMainActivity.java add to onCreate()

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bulb_main);
    //create listener
    AdapterView.OnItemClickListener itemClickListener = new AdapterView.OnItemClickListener(){
        public void onItemClick(AdapterView<?> listView, View view, int position, long id){
            String bulbtype = (String) listView.getItemAtPosition(position);
            //create new intent
            Intent intent = new Intent(BulbMainActivity.this, BulbCategoryActivity.class);
            //add bulbtype to intent
            intent.putExtra("bulbtype", bulbtype);
            //start intent
            startActivity(intent);
        }
    };
    //get the list view
    ListView listview = (ListView) findViewById(R.id.listView);
    //add listener to the list view
    listview.setOnItemClickListener(itemClickListener);
}
```

BulbCategoryActivity will give you an error because we haven't created it yet, we'll do that next.

Java class

We're going to create a custom Java class for the bulb data we'll be using in the next category activity.
In the java folder select the spring folder (not androidTest or test)
File | New | Java class
Name: Bulb
Kind: Class

We're going to create a Bulb class with two data members to store the bulb name and image resource id. We'll have getter and setter methods for both and a private utility method that choses the bulb.

```
public class Bulb {
   private String name;
   private int imageResourceID;

   //constructor
   private Bulb(String newname, int newID){
      this.name = newname;
      this.imageResourceID = newID;
   }

   public static final Bulb[] tulips = {
         new Bulb("Daydream", R.drawable.daydream),
         new Bulb("Apeldoorn Elite", R.drawable.apeldoorn),
         new Bulb("Banja Luka", R.drawable.banjaluka),
         new Bulb("Burning Heart", R.drawable.burningheart),
         new Bulb("Cape Cod", R.drawable.capecod)
   };

   public String getName(){
      return name;
   }

   public int getImageResourceID(){
      return imageResourceID;
   }

   //the string representation of a tulip is its name
   public String toString(){
      return this.name;
   }
}
```

For now we're putting our tulip array in our data model class and making it public so it can be accessed throughout our activities.
The static keyword means it belongs to the class instead of a specific instance. only one instance of a static field exists even if you create a million instances of the class or you don't create any. It will be shared by all instances.

The final keyword in respect to an array means that the array variable(which is actually a reference to an object) cannot be changed to refer to anything else after it's been initialized. The members of the array could be modified, but you can't change the array's reference.

You will eventually add two more arrays for daffodils and iris.

Tulip List

As our activity only needs to contain a single list view with no other GUI components, we can use a list activity, an activity that only contains a list. It's automatically bound to a default layout that contains a list view. So we don't need to create a layout or an event listener as the ListActivity class already implements one.

New | Activity | Empty
BulbCategoryActivity
Uncheck Generate Layout File
Uncheck Backwards Compatibility
Make sure the package name is your project.

In BulbCategoryActivity.java change the superclass from Activity to ListActivity.

Just as with normal activities, list activities need to be registered in the AndroidManifest.xml file. This is so they can be used within your app. When you create your activity, Android Studio does this for you.
<activity android:name=".BulbCategoryActivity"></activity>

We're going to use an array adapter to bind our tulips array to our new list activity. You can use an array adapter with any subclass of the AdapterView class, which means you can use it with list views, grid views, and spinners.

You use an array adapter by initializing the array adapter and attaching it to the list view.

```
import android.widget.ArrayAdapter;

private String bulbtype;

protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   Intent i = getIntent();
   bulbtype = i.getStringExtra("bulbtype");
   //get the list view
   ListView listBulbs = getListView();
   //define an array adapter
   ArrayAdapter<Bulb> listAdapter;
   //initialize the array adapter with the right list of bulbs
   switch (bulbtype){
      case "Tulips":
         listAdapter = new ArrayAdapter<Bulb>(this, android.R.layout.simple_list_item_1, Bulb.tulips);
         break;
      default: listAdapter = new ArrayAdapter<Bulb>(this, android.R.layout.simple_list_item_1,
Bulb.tulips);
```

```
    }
    //set the array adapter on the list view
    listBulbs.setAdapter(listAdapter);
}
```

simple_list_item_1 is a built-in layout resource that tells the array adapter to display each item in the array in a single text view.

Behind the scenes, the array adapter takes each item in the array, converts it to a String using its toString() method and puts each result into a text view. It then displays each text view as a single row in the list view.

Now you should be able to click on tulips and it will launch your new activity and show you the list of tulips. (Because we use tulips as the default, you will see the tulip list no matter which type you tap. Add to the case statement to handle the other types).

ListActivity Listener

In BulbMainActivity.java we created the OnItemClickListener event listener and implemented its onItemClick() method.

BulbCategoryActivity.java is a subclass of the ListActivity class which is a specific type of activity that's designed to work with list views. The ListActivity class already implements an on item click event listener so you don't need to create your own event listener, you just need to implement the onListItemClick() method.

```
import android.view.View;

@Override
public void onListItemClick(ListView listView, View view, int position, long id){
    Intent intent = new Intent(BulbCategoryActivity.this, BulbActivity.class);
    intent.putExtra("bulbid", (int) id);
    startActivity(intent);
}
```

BulbActivity will give you an error because we haven't created it yet, we'll do that next.
We use the id to pass which bulb was selected.

BulbActivity

Now let's create the BulbActivity activity.
New | Activity | Empty
Activity name: BulbActivity
Check Generate Layout File
Layout name: activity_bulb
Uncheck Backwards Compatibility and make sure launcher activity is unchecked.

Go into the activity_bulb layout file and add a TextView for the name. If no vertical constraint is added, add one.

In the TextView make its appearance large and change the id. You can add text for layout and testing.

```
android:textAppearance="?android:attr/textAppearanceLarge"
android:id="@+id/bulb_name"
```

Add an ImageView for the image. Chose an image for layout and testing purposes then you can remove the src property. If no vertical constraint is added, add one. Setting all the constraints to match_constraint allows you to set the aspect ratio to 1:1 so the image doesn't get stretched. Setting scaleType to fitCenter scales the image when you rotate the device.
Change the id to `android:id="@+id/bulbImageView"`
It will be asking you for a content description. Add a string resource and then add a content description.

```
<string name="bulb_image">Bulb picture</string>
android:contentDescription="@string/bulb_image"
```

If you run it at this point you should get to that view regardless of which tulip you tap.

Now let's populate the view with the correct data.

View Data
In BulbActivity.java we'll get the data sent from the intent to populate the view.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bulb);

    //get bulb id from the intent
    int bulbnum = (Integer)getIntent().getExtras().get("bulbid");
    Bulb bulb = Bulb.tulips[bulbnum];

    //populate image
    ImageView bulbImage = (ImageView)findViewById(R.id.bulbImageView);
    bulbImage.setImageResource(bulb.getImageResourceID());

    //populate name
    TextView bulbName = (TextView)findViewById(R.id.bulb_name);
    bulbName.setText(bulb.getName());
}
```

Action bar
The ActionBar, now known as the App Bar, is a consistent navigation element that is standard throughout modern Android applications. https://material.io/guidelines/layout/structure.html#structure-app-bar
The ActionBar can consist of:

- An application icon
- An "upward" navigation to logical parent
- An application or activity-specific title
- Primary action icons for an activity
- Consistent navigation (including navigation drawer)

Styles and Themes
A *style* is a collection of attributes that specify the appearance for a single view. A style can specify attributes such as font color, font size, background color, and more.

A *theme* is a type of style that's applied to an entire app, activity, or view hierarchy, not just an individual view. When you apply your style as a theme, every view in the app or activity applies each style attribute that it supports.

To add an action bar, you need to use a theme that includes an action bar.

In API 11 and above you can use the Holo theme or one of its subclasses. API 21 and above support's the newer Material theme, so we'll be using that.

Android's Material theme has dark and light themes. The light theme expects your App Bar to have a light background color, so it gives you a dark title and a dark overflow menu icon (the three vertical dots). The dark theme expects your App Bar to have a dark background color, so it gives you a light title and a light overflow menu icon.
https://developer.android.com/guide/topics/ui/look-and-feel/themes.html#MaterialTheme
The Material Design color palettes show this concept.
https://material.io/guidelines/style/color.html#color-color-palette
You can also create and test color schemes using this color tool
https://material.io/color/#!/?view.left=0&view.right=0
You can download Google's icons here https://material.io/icons/

You can also structure your app to use different themes based on the Android version of the device. The process is similar to what we did to have different layouts for portrait and landscape. You create a new values folder with the minimum version you want the new styles to be applied to. Then create another styles.xml file and save it in the new values directory that includes the resource version qualifier.
For example:

```
res/values/styles.xml        # themes for all versions
res/values-v21/styles.xml    # themes for API level 21+ only
```

Because the styles in the values/styles.xml file are available for all versions, your themes in values-v21/styles.xml can inherit them. So you can avoid duplicating styles by beginning with a "base" theme and then extending it in your version-specific styles.
https://developer.android.com/guide/topics/ui/look-and-feel/themes.html#Versions

Spring (cont.)
The activities we've created so far are passive activities that provide information and navigation. An active activity lets the user do or create something. Active activities are usually added to the action bar to help users quickly access those activities.

Let's add an action bar to our Spring app where we can access a (fake) bulb order form.

In the AndroidManifest.xml file (app/manifests) the **android:icon** attribute is used to assign an icon to the app. The icon is used as the launcher icon for the app, and if the theme you're using displays an icon in the action bar, it will use this icon.
The **android:label** attribute assigns a user-friendly label to the app or activity, depending on whether it's used in the `<application>` or `<activity>` attribute. The action bar displays the current activity's label. If the current activity has no label, it uses the app's label instead.

The **android:theme** attribute sets the theme. The @style prefix means that the theme is defined in a style resource file.

Open styles.xml (app/res/values)
You will probably have the following:

```
<style name="AppTheme" parent="android:Theme.Material.Light.DarkActionBar">
```

In colors.xml I updated my colors to the following:

```
<color name="colorPrimary">#E8F5E9</color>
<color name="colorPrimaryDark">#b6c2b7</color>
```

In styles.xml I changed and customized my style

```
<style name="AppTheme" parent="android:Theme.Material.Light">
    <!-- Customize your theme here. -->
    <item name="android:colorPrimary">@color/colorPrimary</item>
    <item name="android:colorPrimaryDark">@color/colorPrimaryDark</item>
</style>
```

I'm also going to use the ic_store icon from Google in my app bar.

Now let's add items to the action bar.
To add action items to the action bar you do three things:

1. Define the action items in a menu resource file
2. Get the activity to inflate the menu resource
3. Write the code to respond to a user clicking on the item

Menu Resource
To add a menu we need a menu folder in resources.
Right click on the res folder, chose new Directory and name it menu.
You should have a menu folder nested in the res folder.
Right click on the menu folder, chose new Menu resource file
File name: menu_main
Resource type: Menu
Source set: main
Directory name: menu

Now add a menu item in menu_main.xml

```
<menu xmlns:android= "http://schemas.android.com/apk/res/android">
<item
    android:id="@+id/create_order"
    android:title="@string/create_order"
    android:icon="@drawable/ic_store"
    android:showAsAction="ifRoom" />
</menu>
```

The **android:icon** property assigns an icon for an item that will be shown if there's room. If there's no icon it will show the title.
The **android:showAsAction** property defines how you want the item to appear – always in the main action bar or only if there's room. If there's not room it will go in the overflow area(displayed in three dots). Never will mean you'll always get the 3 dots and the item will be in there.
If you have multiple items in your menu the **android:orderInCategory** property would define the order in which they appear.

You can also define different menus for different activities if you want the items to be different based on the activity the user is in.
Add the string resource in the strings.xml file.
<string name="create_order">Order</string>

Now that we've added action items to the menu resource file, we need to add the items to the action bar in every activity's `onCreateOptionsMenu()` method where we want the menu to be accessible. It runs when the action bar's menu gets created and takes one parameter, a `Menu` object representing the action bar.

In BulbMainActivity.java implement onCreateOptionsMenu() If you just start typing the name of the method you'll be able to use autocomplete and it will also add the needed import statement.

```
import android.view.Menu;

public boolean onCreateOptionsMenu(Menu menu){
   //inflate menu to add items to the action bar
   getMenuInflater().inflate(R.menu.menu_main, menu);
   return super.onCreateOptionsMenu(menu);
}
```

For your activity to react when an action item in the action bar is clicked you need to implement the `onOptionsItemSelected()` method which is called when an item in the action bar is clicked. This will also import the MenuItem class.

```
import android.view.MenuItem;

public boolean onOptionsItemSelected(MenuItem item) {
//get the ID of the item on the action bar that was clicked
   switch (item.getItemId()){
      case R.id.create_order:
         //start order activity
         return true;
      default:
         return super.onOptionsItemSelected(item);
   }
}
```

The `onOptionsItemSelected()` method takes one attribute, a `MenuItem` object that represents the item on the action bar that was clicked.

Now when you run the app you should see the store icon in the action bar to indicate there's a menu and Order should be in the menu.
But you won't see this in the other views. You need to add the same method in all activities you want the menu to be a part of.

Order Activity
Create a new activity for the order using the Empty Activity template.
Activity name: OrderActivity
Check Generate Layout File
Layout name: activity_order
Uncheck Backwards Compatibility

Go into OrderActivity.java. This class should extend Activity. If it doesn't, change the superclass and add the import statement.
We're not going to add the methods to handle a menu because we're not going to add a menu to this activity.

Now go back to BulbMainActivity.java to start OrderActivity when the user clicks on Order in the menu.
Update the case statement in onOptionsItemSelected()

```
case R.id.create_order:
    //start order activity
    Intent intent = new Intent(this, OrderActivity.class);
    startActivity(intent);
    return true;
```

You'll need to implement `onCreateOptionsMenu()` and `onOptionsItemSelected()` in every activity you want the menu to be added to. So add them to BulbCategory.java and BulbActivity.java as well. Both will need the menu class imported and BulbActivity will need the Intent class imported as well.

```
import android.view.Menu;
import android.content.Intent;
```

I'll let you create the order form on your own.

Up Navigation

The action bar has an Up button that is used to navigate up the activity hierarchy. This is not the same as the back button which allows users to navigate back through the history of activities they've been to. The up button will take the user to that activity's parent activity. You define the parent activity in the AndroidManifest.xml file. We're going to make the parent of OrderActivity be BulbMainActivity.
In the AndroidManifest.xml file add a parent for the Order activity

```
<activity android:name=".OrderActivity"
    android:parentActivityName=".BulbMainActivity" >
</activity>
```

Now let's enable the Up button in the action bar in OrderActivity.java

Add to the onCreate() method.

```
//get reference to action bar
ActionBar actionBar = getActionBar();
//enable the up button
actionBar.setDisplayHomeAsUpEnabled(true);
```

This should import the Actionbar class.

```
import android.app.ActionBar;
```

Now when you run the app and go to the Order activity you'll see a back arrow in the action bar which should take you back to the main activity.
If you enable the up button and the activity doesn't have a parent defined the app will crash.