ATLS 4120: Mobile Application Development
Week 11: Android UI Controls

Widgets

The Android SDK comes with many widgets to build your user interface. We've already looked at text views, edit texts, buttons, and image views. Today we're going to look at 4 more.

Material Design Selection controls https://material.io/guidelines/components/selection-controls.html#selection-controls-switch

Switch

https://developer.android.com/guide/topics/ui/controls/togglebutton

- A switch is a two state toggle switch that can select between two options https://developer.android.com/reference/android/widget/Switch.html
- The **android:textOn** and **android:textOff** attributes determine the text you want to display depending on the state of the switch
- The **isChecked()** method returns a boolean – true if it's on, false if it's off
- Added in API 14
- Responds to the **OnCheckedChange** event

CheckBox

https://developer.android.com/guide/topics/ui/controls/checkbox.html

- Check boxes let you display multiple options that the user can check or not https://developer.android.com/reference/android/widget/CheckBox.html
- Each check box is independent of the others
- The **isChecked()** method returns a boolean – true if it's checked, false if it's not
- Responds to the **onClick** event

RadioButton

https://developer.android.com/guide/topics/ui/controls/radiobutton.html

- Radio buttons let you display multiple options next to each other https://developer.android.com/reference/android/widget/RadioButton.html
- Responds to the **onClick** event
- RadioGroup is a container for radio buttons
  - Users can select only ONE radio button in a radio group
  - The **getCheckedRadioButtonId()** method returns the id (integer) of the chosen button
    - -1 means no button was chosen

Spinner

https://developer.android.com/guide/topics/ui/controls/spinner.html

- A spinner presents a drop-down list of values from which only one can be selected https://developer.android.com/reference/android/widget/Spinner.html
- Use if you don't need to show the options next to each other (radio buttons)
- You can store the values as an array in strings.xml
- The **getSelectedItem()** method returns the String of the selected item
- Responds to the **onItemSelected** event

Properties

Widgets that are descendants from the View class have some commonly used properties available

- android:id
    - Gives the component a unique identifying name
    - Lets you access the widget in your code
    - Lets you refer to the widget in your layout
- android:text
- the text displayed in that component
    - string resources should be used for these
- All controls will have layout_width and layout_height

Snackbar
https://developer.android.com/training/snackbar
A snackbar provides a quick popup message to the user. The current activity remains visible and interactive while the Snackbar is displayed. Snackbars automatically disappear after a short time.

- An action for the user to respond to can be added to a snackbar

Snackbars supercede Toasts and are preferred although Toasts are still supported.

Taco
From the Welcome screen chose Start a new Android Studio Project (File | New | New Project)
Select a Project Template: In the Phone and Tablet tab pick Empty activity.
Name: Taco
Package name: the fully qualified name for the project
Save location: the directory for your project (make sure there is a directory with the name of the project after the location)
Language: Kotlin
Minimum SDK: API 21: Android 5.0 Lollipop (21 is the minimum API for Material Design)
(Help me choose will show you the cumulative distribution of the API levels)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

Open the activity_main.xml layout file.

TextView
In Design mode move the textview to the top. We'll use this as a heading so change the textAppearance to Large or Display1. (Higher numbers are larger, opposite from HTML.)

Update the textView to use a string resource called heading with the value "Taco Tuesday".

Radio Buttons
Add 3 radio buttons to pick the taco filling
Radio buttons belong in a group so add a radio button group first and make the orientation horizontal.
Give it an id of radioGroup.
It will need horizontal and vertical constraints if it doesn't have any.

Add 3 radio buttons into the radio group going across in a row.
Notice they're each added with layout_weight of 1. This will make them be equally spaced across the group.
Update their ids to be radioButton1, radioButton2, and radioButton3.
Add three string resources for them – chicken, steak, and veggie.

<u>Button</u>
Add a button and check that it has an id (button).
Add string resource called tacoButton with the value "Create taco".
Add needed constraints leaving some room between the radio buttons and the button because we'll be adding more views above the button.
For the onClick event assign it the name of the method you want it to call – createTaco.
Create this method in the MainActivity.kt file either manually or through the XML lightbulb shortcut. It must have the name createTaco and take a parameter of type View since it will be called by a Button which is a subclass of View.

```
    fun createTaco(view: View) {}
}
```

<u>TextView</u>
Add another TextView below the button that we'll use for our output. I updated the id to be more descriptive – messageTextView.
Add missing constraints. I also added a start and end margin of 16. If you want the text centered set the gravity attribute to center.
I changed textAppearance to medium.
Remove the default text as we'll be setting this programmatically.

<u>Android Extensions</u>
Kotlin Android Extensions are a Kotlin plugin that allows even easier access to our views from our Kotlin code. The plugin will allow you to automatically access the views in the layout file as variables without having to define them using findViewById().
So instead of having to do:

```
val resultTextView=findViewById<TextView>(R.id. messageTextView)
resultTextView.text
```

You can just textView2 directly and access its properties

```
messageTextView.text
```

To add the Android Extension open the build.gradle module app file and at the top in the plugins section add: `id 'kotlin-android-extensions'`

You'll see a message at the top that your gradle file has changed. Click Sync Now at the top right.

In your Kotlin file the first time you access a view by its id you will get an error telling you to import

```
import kotlinx.android.synthetic.main.activity_main.*
```

Either click import or add this at the top with the other import statements manually.
The error should go away and now you can skip all the findViewById() calls and just access the views directly using their ids.

The plugin is creating some code behind the scenes to make the views accessible. The first time a view is accessed it calls findViewById and then creates a local view cache so in subsequent calls the view will be recovered from the cache. So this is not only easier, it's faster as well.

<u>MainActivity.kt</u>
Let's update createTaco(View) to get the selected radio button and use its text in a String we write to the messageTextView.

```
fun createTaco(view: View) {
    //radio buttons
    val fillingId = radioGroup.checkedRadioButtonId
    val filling = findViewById<RadioButton>(fillingId).text

    //textview
    messageTextView.text = "You'd like $filling tacos"
}
```

We can just access the view with the id radioGroup without calling findViewById() because of the Android Extension.
checkedRadioButtonId returns an Int for the radio button that's selected.
Then we use that Int to find the view with that id and access its text property which returns the String value of its string resource.

Run your app and make sure it works.
There is an issue, can you find it?

Snackbar
checkedRadioButtonId returns -1 if no button is selected and trying to access the text property for a view with the id -1, which doesn't exist, will crash the app.
So let's reorganize our code to handle this and present a snackbar with a message if the user didn't select a taco filling.
A Snackbar needs a view to attach itself to so go into the activity_main.xml layout and in the component tree select Constraint Layout and give it an id of root_layout.
Update createTaco(View)

```
var filling : CharSequence = ""
//radio buttons
val fillingId = radioGroup.checkedRadioButtonId

if (fillingId == -1){
    //snackbar
    val fillingSnackbar = Snackbar.make(root_layout, "Please select a 
filling", Snackbar.LENGTH_SHORT)
    fillingSnackbar.show()
} else {
    filling = findViewById<RadioButton>(fillingId).text

    //textview
    messageTextView.text = "You'd like $filling tacos"
}
```

We add an if statement to test if no radio button is selected and therefore the id is -1.
We create a Snackbar that will show its message on the main view with a short duration.
Snackbar has other methods but we'll just keep it simple.

We also need filling to have a default value so I moved it to the top of the function and made it a variable since it will change. It needs to be of type CharSequence because that's the type of the text property.
I also move the message assignment of the TextView into the else so the user will only see the Snackbar message if they haven't picked a filling. The rest of our logic will be there as well.

Check boxes
Let's add 4 checkboxes in a row. Give them ids checkBox1- checkBox4
Replace their default text with string resources for taco toppings – salsa, cheese, guacamole, and sour cream.
Select them all, right click Align | Top Edges.
Then I selected them all again, right click Chain |Create horizontal chain. I set the chain mode to spread by cycling through the chain modes.
The left most checkbox will need a constraint to the left, and the right most checkbox a constraint to the right.
Then add a vertical constraint for one of them and all constraint conditions should be satisfied.

MainActivity.kt
Add a variable for the list of toppings that we'll build.
```
var toppinglist = "" //String
```

Add logic to the else statement in createTaco(View)

```
if (checkBox1.isChecked){
    toppinglist += " " + checkBox1.text
}
if (checkBox2.isChecked){
    toppinglist += " " + checkBox2.text
}
if (checkBox3.isChecked){
    toppinglist += " " + checkBox3.text
}
if (checkBox4.isChecked){
    toppinglist += " " + checkBox4.text
}
if (toppinglist.isNotEmpty()){
    toppinglist = "with" + toppinglist
}
messageTextView.text = "You'd like $filling tacos $toppinglist"
```

We test to see if each checkbox is checked and build a string of toppings.

You can also use conditional expressions for the if statements. These require an else condition.
```
toppinglist = (if (toppinglist.isNotEmpty()) "with$toppinglist" else
"").toString()
```

Spinner
Add a spinner to the right of the toggle button (in the palette containers section, or search)
Notice it's been given the id "spinner".

We’ll store the values for the spinner in an array.
Just like we defined strings we can define a string-array as a resource in strings.xml. This works when the items in the array won’t change. We’ll deal with changing lists next semester.

```
<string-array name="location">
    <item>the Hill</item>
    <item>29th St.</item>
    <item>Pearl St.</item>
</string-array>
```

In activity_main.xml set the spinner’s entries attribute to use this reference @array/locations.

Fix the spinner’s missing constraints. Make the layout_width “wrap_content” if you want it to be the size needed for the content. “match_parent” will make it the width of the device.
Be aware of what part of the layout the spinner’s content will cover when the user taps it.

MainActivity.kt
Add logic to the else statement in createTaco(View)

```
val location = "at " + spinner.selectedItem
```

We call selectedItem on the spinner to get the item that was selected. Notice it returns type Any! Any is the root of the Kotlin class hierarchy. Every Kotlin class has Any as a superclass.
When you see a type defined with a “!” after it that means it maybe be mutable or not. T! means “T or T?”

Select lists present a UX dilemma. If you provide a list where the first option is one of the choices, there’s no good way to know if the user really picked that or if it’s there as the default. If the first option is blank it doesn’t provide an indication of what that control is for and you’ll need text next to it to describe it. Or you can use the first option as a description to the user and then add logic to check for that option and treat it as no selection.
I’ve kept this example simple but we’ll look at another approach next week.
Which do you think is better?

Switch
Add a switch to the right of the spinner.
Note that the default id is switch1 and not switch. Why do you think we can’t use switch as an id?
Since we’re using the Kotlin extension plug-in where we can use our ids as variables, let’s make the id more descriptive and change it to glutenSwitch.
Add a string resource called gluten_free with the value Gluten-Free.
Add needed constraints.
Note: The Switch component does not implement the Material Switch, instead it extends the AppCompat Switch. Therefore our styling doesn’t apply to the Switch unless you specify the Material Switch.

```
<com.google.android.material.switchmaterial.SwitchMaterial … />
```

MainActivity.kt
Add logic to the else statement in createTaco(View)

Add a switch to the right of the spinner.
Note that the default id is switch1 and not switch. Why do you think we can’t use switch as an id?

Since we're using the Kotlin extension plug-in where we can use our ids as variables, let's make the id more descriptive and change it to glutenSwitch.
Add a string resource called gluten_free with the value gluten-free and assign that resource to the text attribute. The text attribute controls the text displayed in the switch label.
Add needed constraints.

MainActivity.kt
Update createTaco(View)

```
if (glutenSwitch.isChecked){
    filling = glutenSwitch.text.toString() + " $filling"
}
```

If you end up with a lot of space in the layout above the button, update the vertical constraints. Instead of a bottom constraint I used a top constraint to the View above it, the spinner or switch.

Scroll View
You can add a ScrollView to your layout so your layout will scroll if needed.
A ScrollView can only contain a **single child element** so we're going to make it our root layout, wrapping our constraint layout and all the views in it, in a scroll view.
If you add a ScrollView in design mode it goes inside the constraint layout which is not what we want so I'm going to add it directly in the XML. You can also add it in design mode and then go into the XML and move it around. Here's the end result:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android=http://schemas.android.com/apk/res/android
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/root_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

… all your views

</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

Note the XML tag is outside of all other tags.
We've moved the XML namespace tags that start with xmlns, to the ScrollView as it needs to be in the root view.
Then you have to make the layout_height "wrap_content" so it automatically grows with the content.