COMP-304 Winter 2023

Review of Lecture 4

- **Declaration of Application resources**:
  - Declare resources, such as Strings, Integers, Booleans, Colors, Drawables, String Arrays, etc., in XML files
  - Common files to use:
    - **strings.xml**
    - **color.xml**
    - **dimens.xml**
    - **drawbles.xml**
  - Also, you can declare resources in Java code
- Introduction to **Android User Interface** elements
  - **View**, superclass of UI classes
  - **ViewGroup**, inherits from View, represents a container which holds views
  - Android Layout classes:
    - **LinearLayut**
    - **FrameLayout**
    - **RelativeLayout**
    - **ConstraintLayout**
    - **TableLayout**
    - **ScrollView**

Review of Lecture 4

- **RecyclerView**
  - more advanced and flexible version of ListView
  - The views in the list are represented by **view holder** objects (RecyclerView.ViewHolder)
  - The view holder objects are managed by an adapter, which you create by extending RecyclerView.Adapter
  - The adapter **creates view holders** as needed.
  - The adapter also **binds the view holders** to their data.

- **Declarative Data Binding**
  - A **support library** that allows you to bind UI components in your layouts to data sources in your app **using a declarative format**.
  - binding variables are defined inside a **data** element that is a sibling of the UI layout's root element.
  - Both activity layout and data element are wrapped in a layout tag.
  - supports **two-way data binding**.

Review of Lecture 4

- **Simple UI controls and event handling**:
  - Declare Android controls in XML files, in **res\layout** folder
  - Use Layout editor to create the UI
  - Use the toolbars in Layout editor to arrange UI controls in the screen
  - Drag UI controls from the palette to the layout
  - Use Attributes window to set the values for attributes of UI controls

- **Simple UI controls and event handling**:
  - TextView
  - EditText
  - Button
- Use findViewById method to instantiate UI objects in Kotlin code
  - android.widget package
  - Use onClick attribute to handle Button click events
  - Use Attributes window to **associate an event handler** method in the activity class with onClick attribute

Designing UI with Standard Views

Objectives:

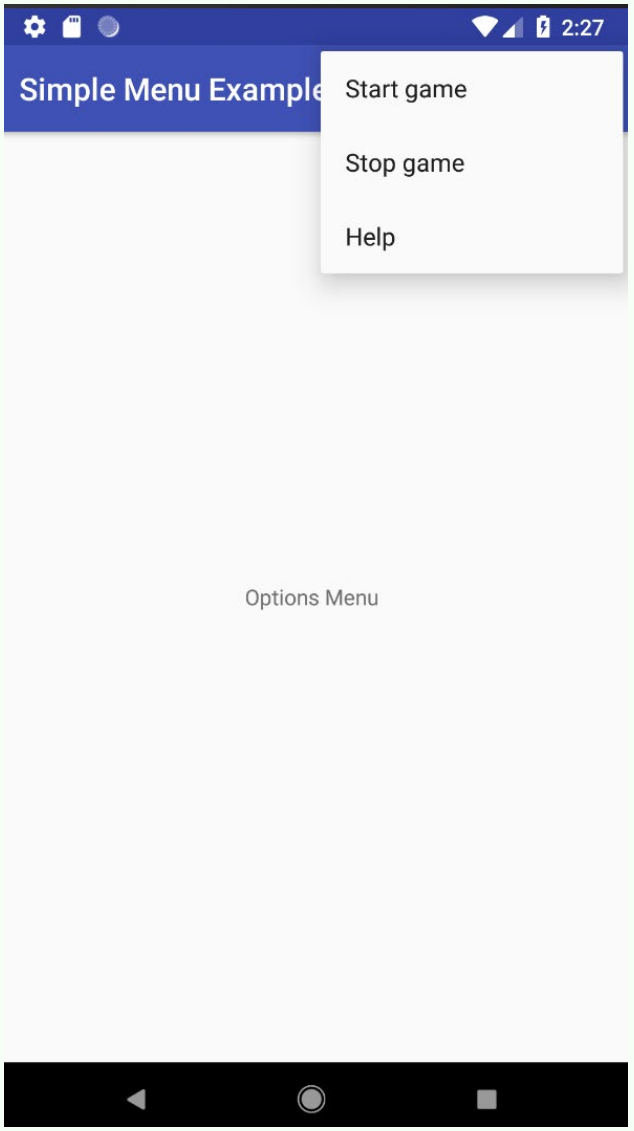
- Examine Android Views
- Utilize Menus in Android Apps
- Implement Check boxes, RadioGroup & RadioButton, ToggleButton, ImageButton, Spinner, Progress indicators:
  - Defining UI elements
  - Event Handling
  - Handling threads

Using Menus in Android Apps

- There are **three fundamental types of menus** or action presentations on all versions of Android:
  - The **options menu** is the primary collection of menu items for an activity.
    - It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."
  - A **context menu** is a floating menu that appears when the user performs a long-click on an element.
  - A **popup menu** displays a list of items in a vertical list that's anchored to the view that invoked the menu
- Each menu resource is stored as a specially formatted XML files in the /res/**menu** directory
- Here's an example of a simple menu resource file /res/menu/game_menu.xml that defines a short menu with three items in a specific order:

Options Menu

```
?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/start"
        android:title="@string/start">

    </item>

    <item
        android:id="@+id/stop"
        android:title="@string/stop">

    </item>

    <item
        android:id="@+id/help"
        android:title="@string/help">

    </item>

</menu>
```



Options Menu

- To access the preceding menu resource called /res/menu/game_menu.xml, simply override the method onCreateOptionsMenu() in your application:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    val inflater = menuInflater
    inflater.inflate(R.menu.game_menu, menu)
    return true
}
```

Options Menu

- **Handling the event** when a menu option item is selected:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle item selection
        when (item.getItemId()) {
            R.id.start -> Toast.makeText(this, "You selected start!", Toast.LENGTH_LONG).show()
            R.id.play -> Toast.makeText(this, "You selected Play!", Toast.LENGTH_LONG).show()
            R.id.playWell -> Toast.makeText(this, "You selected Play Well!", Toast.LENGTH_LONG)
                .show()
            R.id.stop -> Toast.makeText(this, "You selected stop!", Toast.LENGTH_SHORT).show()
            R.id.help -> Toast.makeText(this, "You selected help!", Toast.LENGTH_LONG).show()
            R.id.nothelp -> Toast.makeText(this, "You selected nothelp!",
Toast.LENGTH_LONG).show()
            R.id.someId -> Toast.makeText(this, "You selected someID!",
Toast.LENGTH_LONG).show()
            else -> return super.onOptionsItemSelected(item)
        }
        return true
    }
```

Context Menu

- **Register the View** to which the context menu should be associated by calling registerForContextMenu() and pass it the View.

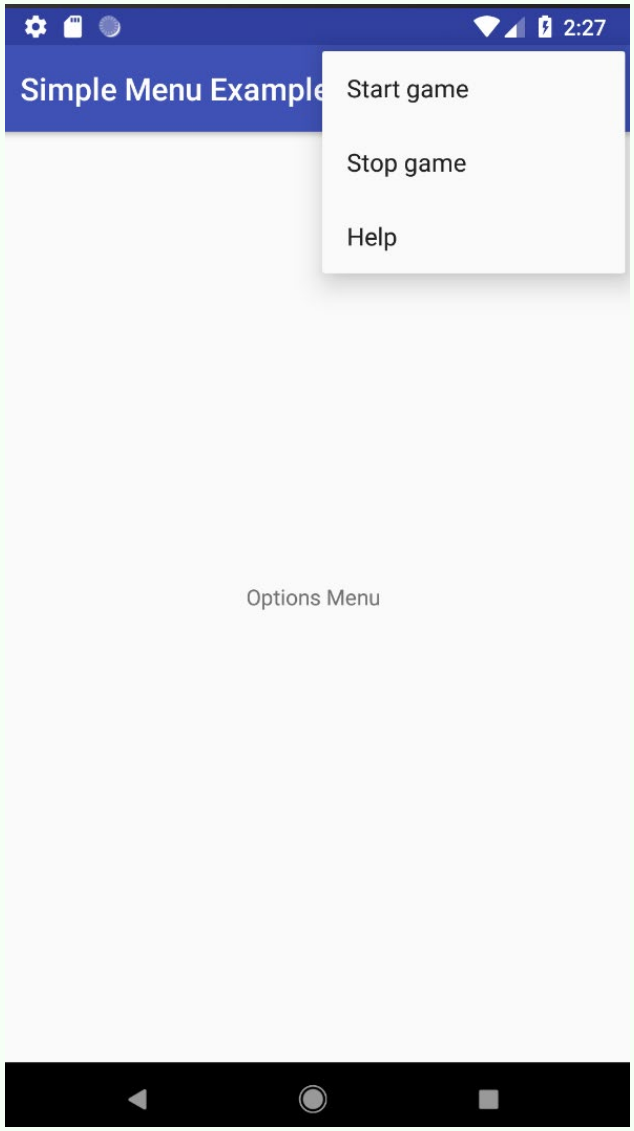
registerForContextMenu(textView)

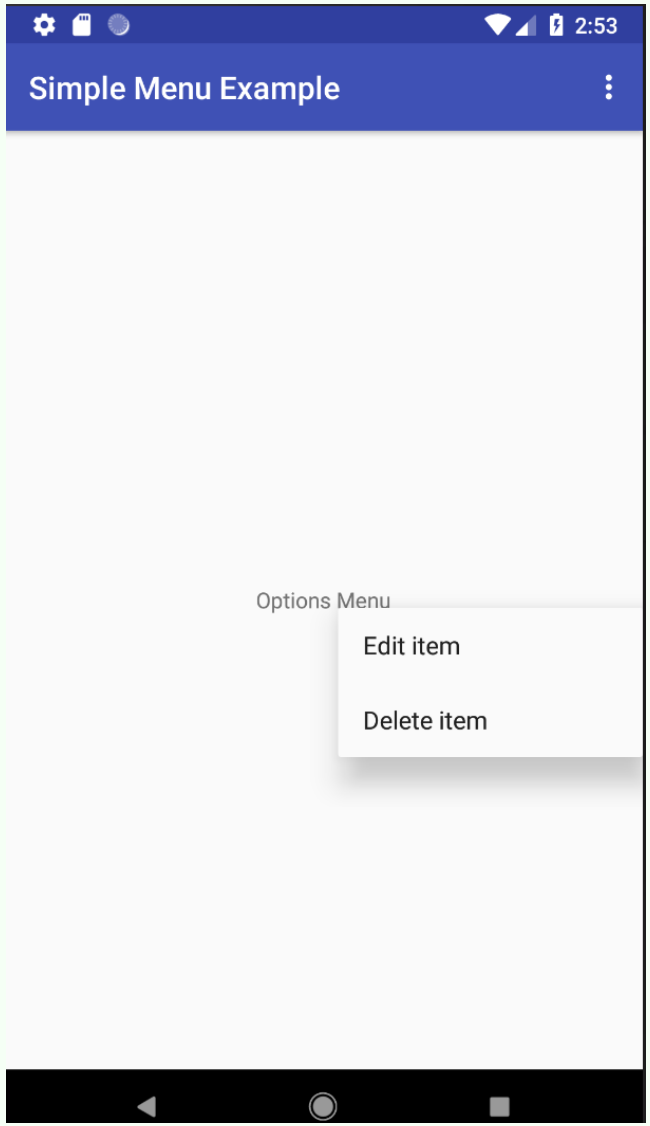
- **Implement** the onCreateContextMenu() method in your Activity or Fragment.

```
override fun onCreateContextMenu(
        menu: ContextMenu?, v: View?,
        menuInfo: ContextMenuInfo?
    ) {
        super.onCreateContextMenu(menu, v, menuInfo)
        val inflater = menuInflater
        inflater.inflate(R.menu.context_menu, menu)
       }
```

- **Implement onContextItemSelected()** in your activity, similarly to onOptionsItemSelected.
- See the SimpleMenuExample.

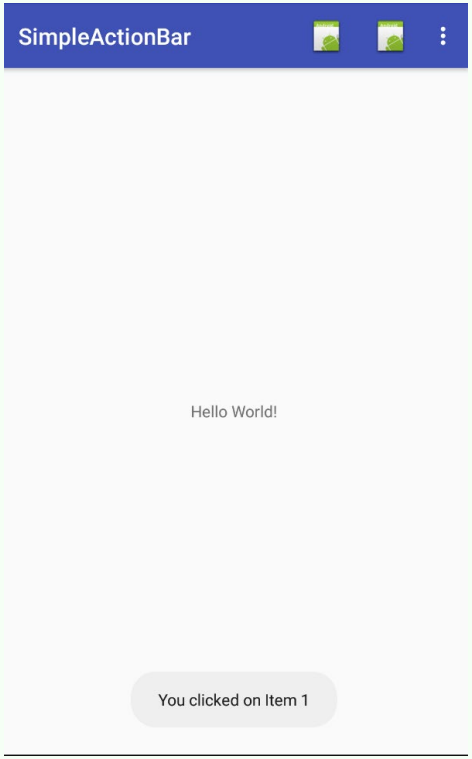
SimpleMenuExample app





Action Bar

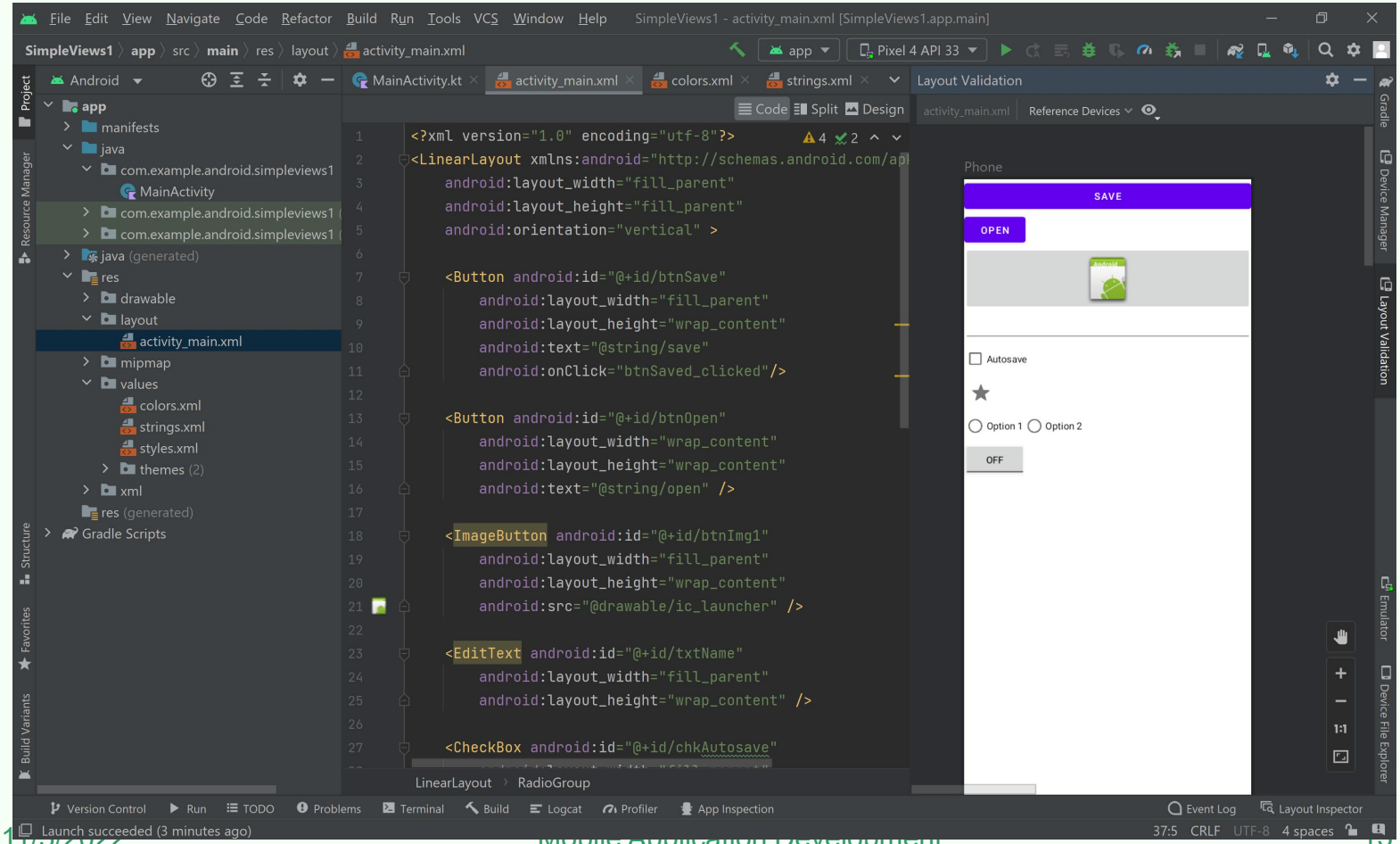
- Another newer feature introduced in Android 3 and 4 is the Action Bar.
- Located at the top of the device's screen, the Action Bar **displays the application icon together with the activity title**.
  - Optionally, on the right side of the Action Bar are *action items.*



- *SimpleActionBar example*

Designing UI

- Use design editor (drag and drop)
- XML definitions will be stored in layout files



Event Handling of Basic Views

1. **Create a reference to the control** using **findViewById** method:

```
val btnOpen: Button = findViewById<View>(R.id.btnOpen) as Button
```

2. **Register the control** with a proper listener:

```
btnOpen.setOnClickListener(object : View.OnClickListener {
```

3. **//Implement the event handler method**

```
        override fun onClick(v: View?) {
          DisplayToast("You have clicked the Open button")
        }
    })
```

- **Alternatively**, add the android:**onClick** attribute to the <Button> element in your XML layout.

Using Check Boxes

- The Android check box contains a **text** attribute that appears to the side of the check box.
- Here's an XML layout resource definition for a CheckBox control:

```
<CheckBox
android:id="@+id/checkbox"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Autosave" />
```

Event handling of Check Boxes

```
//1- create the check box reference
 val checkBox = findViewById<View>(R.id.chkAutosave) as CheckBox
//2- register the checkbox reference with a click listener
checkBox.setOnClickListener(object : View.OnClickListener {
//3- implement the event handler method
            override fun onClick(v: View) {
              if ((v as CheckBox).isChecked) DisplayToast("CheckBox is
checked") else DisplayToast(
                  "CheckBox is unchecked"
              )
          }
       })
```

- **Alternatively**, add the android:**onClick** attribute to the <CheckBox> element in your XML layout.

Using **RadioGroups and RadioButtons**

- The RadioButton controls are similar to CheckBox controls.
- They have a **text label** next to them, set via the text attribute, and they have a **state** (checked or unchecked)
- However, you should group **RadioButton** objects inside a **RadioGroup** that handles enforcing their combined states so that **only one RadioButton can be checked at a time**.

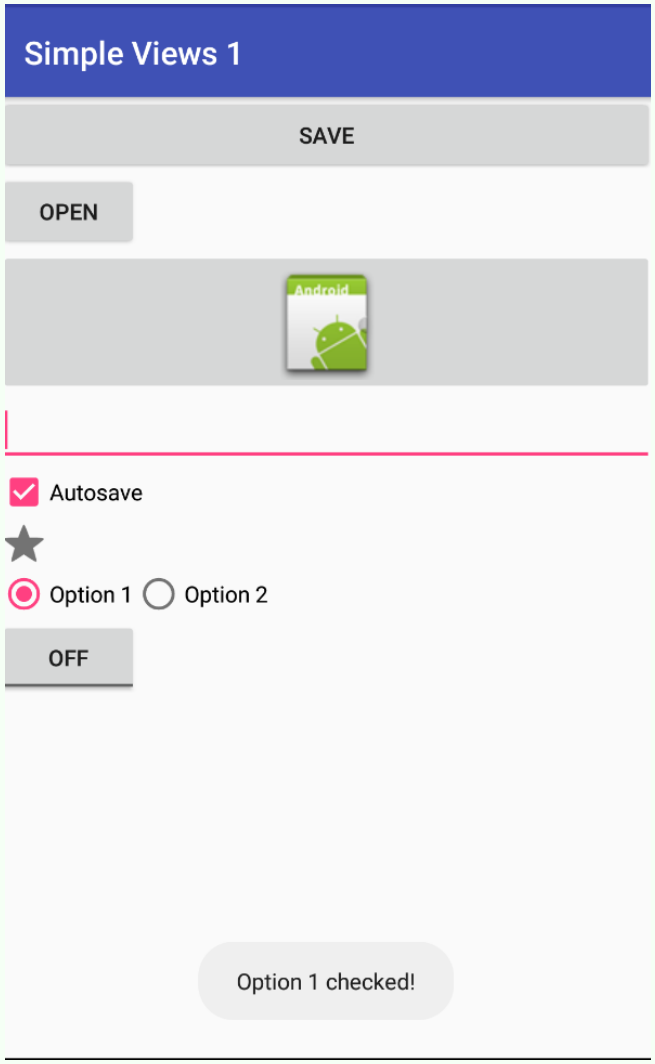
Using **RadioGroups and RadioButtons**

- The XML layout resource definition below shows a RadioGroup containing two RadioButton objects

```
<RadioGroup android:id="@+id/rdbGp1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <RadioButton android:id="@+id/rdb1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/option_1" />
        <RadioButton android:id="@+id/rdb2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/option_2" />
    </RadioGroup>
```

Using **RadioGroups and RadioButtons**

- You handle actions on these RadioButton objects through the RadioGroup object.
- The following example shows event handling of RadioButton objects for a click event.



Event handling of Radio buttons

```
//1- create the radio group reference
 val radioGroup = findViewById<View>(R.id.rdbGp1) as RadioGroup
//2- register the radio group reference with a click listener
 radioGroup.setOnCheckedChangeListener { group, checkedId ->
{
//3- implement the event handler method
radioGroup.setOnCheckedChangeListener { group, checkedId ->
        val rb1 = findViewById<View>(R.id.rdb1) as RadioButton
        if (rb1.isChecked) {
            DisplayToast("Option 1 checked!")
        } else {
            DisplayToast("Option 2 checked!")
        }
    }
```

- **Alernatively**, add the android:**onClick** attribute to the <RadioButton> element in your XML layout.

Toggle buttons

- A **Toggle** Button is similar to a check box in behavior but is usually used to show or alter the **on** or **off state** of something.
- Like the CheckBox, it **has a state** (checked or not).
- Unlike the CheckBox, it does not show text next to it. Instead, it **has two text fields**.
  - The first attribute is **textOn**, which is the text that displays on the button when its checked state is on.

  - The second attribute is **textOff**, which is the text that displays on the button when its checked state is off.

Toggle buttons

- The following layout code shows a definition for a toggle button that shows “Enabled” or “Disabled” based on the state of the button:

```
<ToggleButton
android:id=”@+id/toggle_button”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”Toggle”
android:textOff=”Disabled”
android:textOn=”Enabled” />
```

Event handling of Toggle buttons

```
//1- create the toggle button reference
 val toggleButton = findViewById<View>(R.id.toggle1) as ToggleButton
//2- register the radio group reference with a click listener
 toggleButton.setOnClickListener(object : View.OnClickListener {
//3- implement the event handler method
override fun onClick(v: View) {
            if ((v as ToggleButton).isChecked) DisplayToast("Toggle button
is On") else DisplayToast(
                "Toggle button is Off"
            )
        }
    })
  }
```

Validating Input

- **Using attributes** of UI controls:
  - **maxLength**

```
<EditText
  android:id="@+id/editText1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:maxLength="5“ />
```

- **Using Input Filters** programmatically:
  - Here is an example of an EditText control with two **built-in filters** that might be appropriate for a **two-letter state abbreviation**:

```
val text_filtered = findViewById<View>(R.id.input_filtered) as
  EditText
text_filtered.filters = arrayOf( AllCaps(), LengthFilter(2) )
```

Using **Spinner Controls**

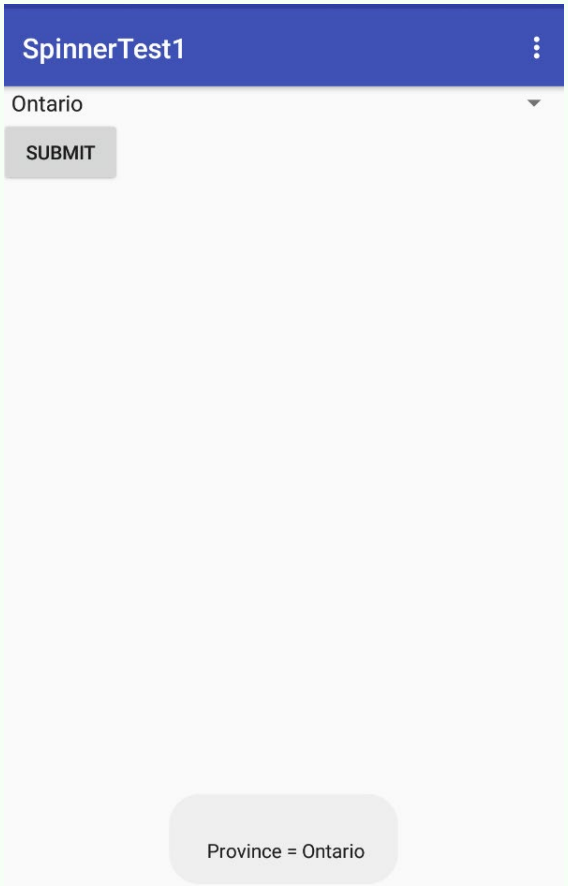
- Limit the choices available for users to type:
  - set the available choices in the layout definition by using **the entries attribute** with an array resource (specifically a string-array that is referenced as something such as @array/state/province-list):
- Here is an example of the XML layout definition for a Spinner control for choosing a color:

```
<Spinner
android:id=”@+id/Spinner01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:entries=”@array/colors”
android:prompt=”@string/spin_prompt” />
```

Spinner Control example

- Retrieve the selected View and extract the text directly:

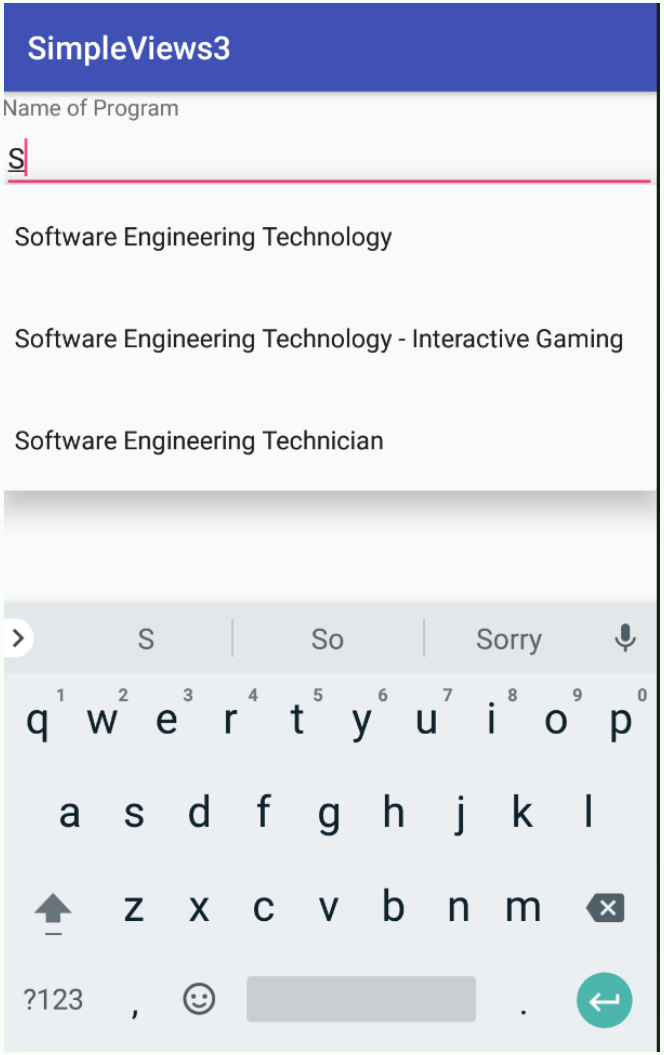
```
val spin = findViewById<View>(R.id.provinces_spinner) as
    Spinner
     //create a button object
     val submit: Button = findViewById<View>(R.id.submit) as
    Button
     //handle the  click event
     submit.setOnClickListener(object : View.OnClickListener
    {
        override fun onClick(v: View?) {
            //get the spinner view as text view
            val text_sel = spin.selectedView as TextView
            //get the text from the spinner view
            Toast.makeText(
                this@MainActivity, """
Province = ${text_sel.text}""", Toast.LENGTH_SHORT
            ).show()
        }
     })
```



Filtering choices with a **spinner** control

AutoCompleteTextView View

- The **AutoCompleteTextView** is a view that is similar to EditText (in fact it is a subclass of EditText), except that it **shows a list of completion suggestions automatically while the user is typing**.



AutoCompleteTextView View

- An **ArrayAdapter** object manages the array of strings that will be displayed by the **AutoCompleteTextView**.

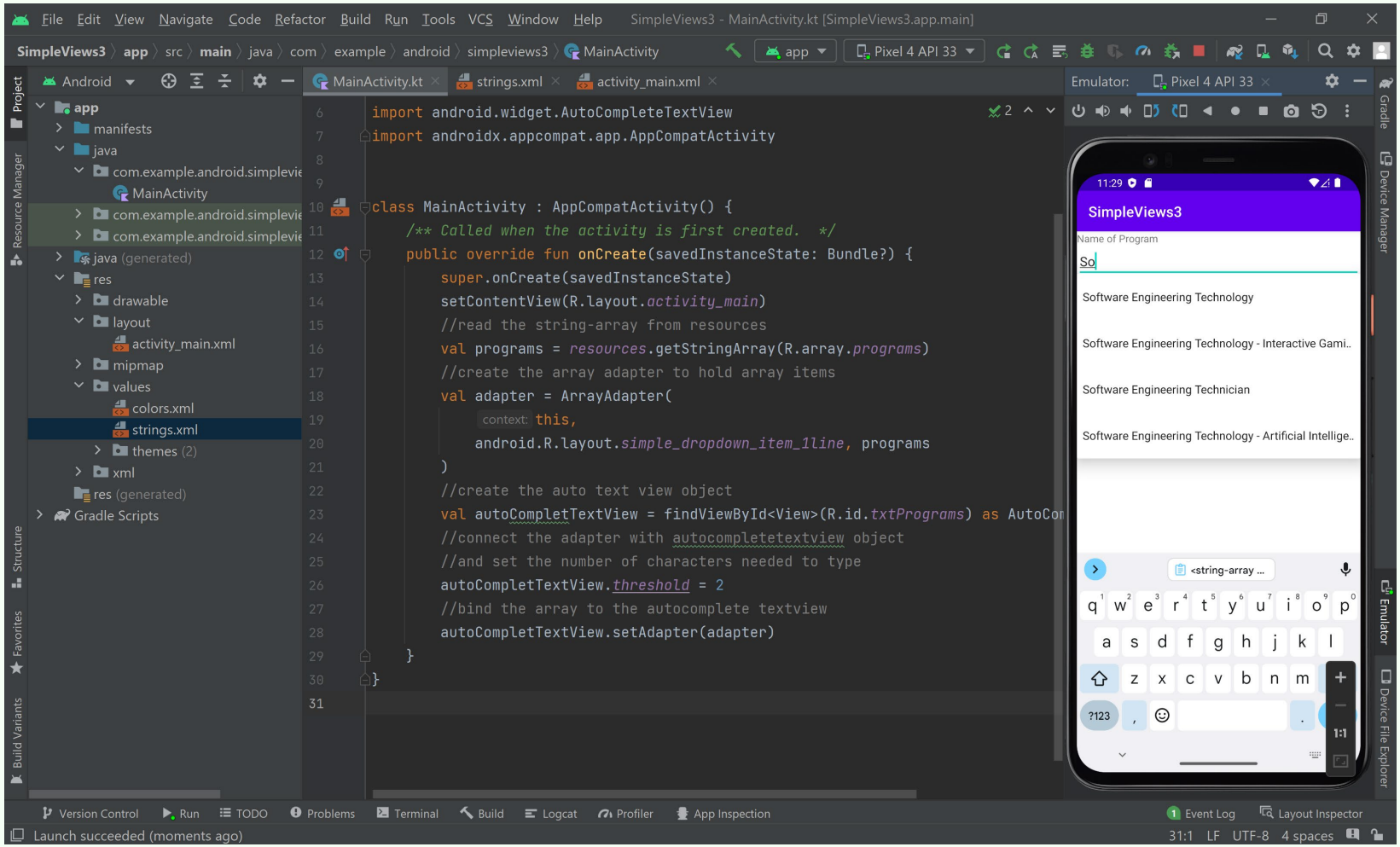
```
<string-array name = "programs">
        <item>Software Engineering Technology</item>
        <item>Software Engineering Technology - Interactive Gaming</item>
        <item>Health Informatics Technology</item>
        <item>Software Engineering Technician</item>
        <item>Mobile Apps Development</item>
        <item>Software Engineering Technology - Artificial Intelligence</item>
    </string-array>
```

- You set the AutoCompleteTextView to display in the ***simple_dropdown_item_1line*** mode:

AutoCompleteTextView View

```
class MainActivity : AppCompatActivity() {
    /** Called when the activity is first created.  */
    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //read the string-array from resources
        val programs = resources.getStringArray(R.array.programs)
        //create the array adapter to hold array items
        val adapter = ArrayAdapter(
            this,
            android.R.layout.simple_dropdown_item_1line, programs
        )
        //create the auto text view object
        val autoCompletTextView = findViewById<View>(R.id.txtPrograms) as
AutoCompleteTextView
        //connect the adapter with autocompletetextview object
        //and set the number of characters needed to type
        autoCompletTextView.threshold = 2
        //bind the array to the autocomplete textview
        autoCompletTextView.setAdapter(adapter)
    }
}
```

AutoCompleteTextView View



Using ImageButton

- An **ImageButton** is, for most purposes, almost exactly like a basic button.
  - Click actions are handled in the same way.
- The primary difference is that you can set its **src attribute to be an image**.
- Here is an example of an ImageButton definition in an XML layout resource file:

```
<ImageButton
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:id=”@+id/image_button”
android:src=”@drawable/droid” />
```

Progress Indicators

- A ProgressBar is a user interface element that indicates the **progress of an operation**.
- **Indeterminate** Progress
  - Use it when you do not know how long an operation will take.
  - is the default for progress bar and shows **a cyclic animation** without a specific amount of progress indicated.

```
<ProgressBar
    android:id="@+id/indeterminateBar"
    android:layout_width="wrap_content"
   android:layout_height="wrap_content"
    />
```

Progress Bar

- **The cyclic animation** shows that something is taking place.
- There **are three sizes** of this type of progress indicator
  - The default style is for a **medium-size** circular progress indicator.
  - Two other styles for indeterminate progress bar are **progressBarStyleLarge** and **progressBarStyleSmall**.
  - When the value reaches the maximum value, the indicators fade away so that they aren't visible.
- **Determinate** Progress
  - Use it when you want to show that a specific quantity of progress has occurred (file loading, etc.).

```
<ProgressBar
    android:id="@+id/determinateBar"
    style="@android:style/Widget.ProgressBar.Horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:progress="25"/>
```

Progress Indicators

- You can update the percentage of progress displayed by using progress property:

  progressBar!!.**progress** = progressStatus

  - Setting the progress to 75 shows the indicator at 75 percent complete.
  - **Use incrementProgressBy(int)** to increase the current progress completed by a specified amount.
  - By default, the progress bar is full when the progress value reaches 100.
  - You can adjust this default by setting the **max** property:

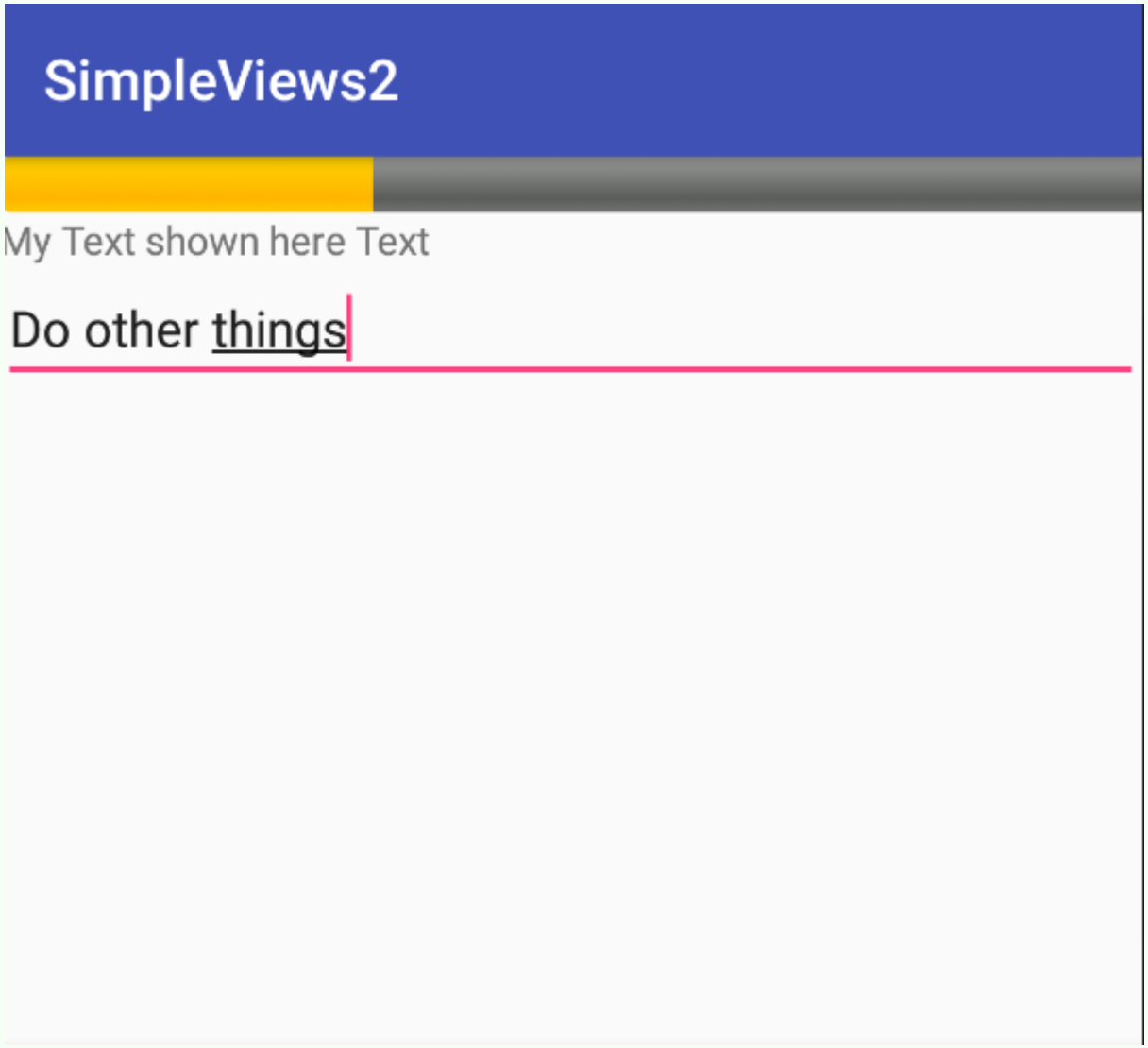
    progressBar!!.max = 1000

Android UI thread

- Android's main thread launched by the system is called **UI thread**. Android requires:
  - **Do not block the UI thread** – create other threads to do some time consuming work
  - **Do not access the Android UI toolkit from outside the UI thread** – to communicate with the UI thread from your new thread, just post a message to the Handler object created on the UI thread:

```
while (progressStatus < 5000) {
        // get the updated progess value
        progressStatus = doSomeWork()
        // Update the progress bar
        //you have to do that from within UI thread
        //by posting a message to Handler object
        handler.post(Runnable
        //this thread updates the progress status
        { //set the updated progress value
           progressBar!!.progress = progressStatus
        })
}
```

SimpleVews2 example

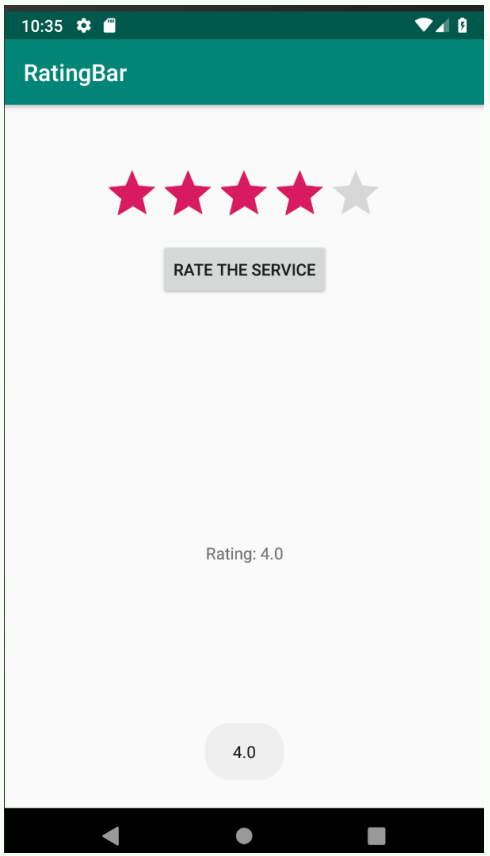


Displaying Rating Data with **RatingBar**

- **RatingBar** has a more specific purpose: showing ratings or getting a rating from a user:
- Here's an example of an XML layout resource definition for a RatingBar with four stars:

```
<RatingBar android:id="@+id/ratebar1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:numStars="4"
android:stepSize="0.5" />
```

- Here, users can choose any rating value between 0 and 4.0, but only in increments of 0.5, the **stepSize** value.



Displaying Rating Data with **RatingBar**

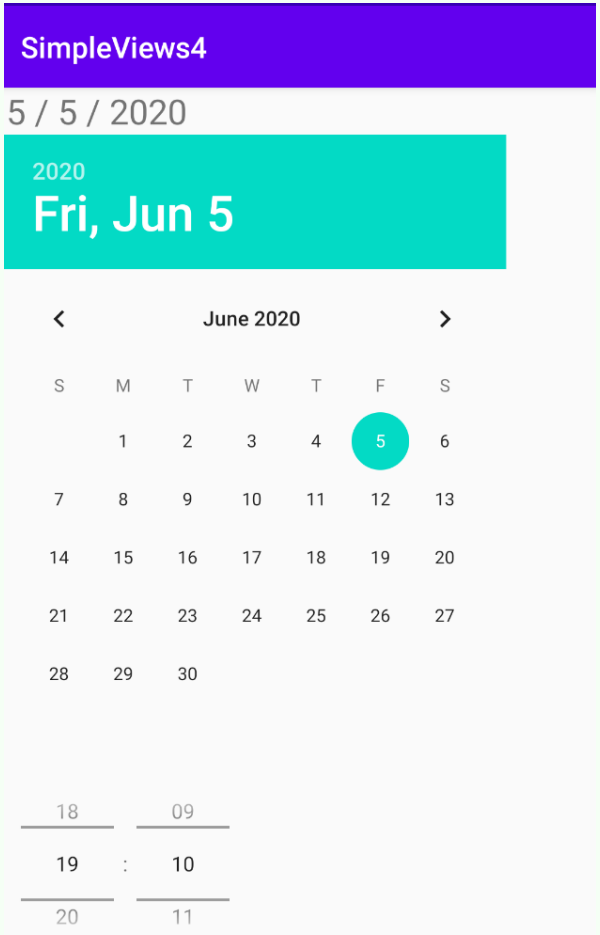
- To indicate that rating has changed, use RatingBar.**OnRatingBarChangeListener** class:

```
ratingBar = findViewById<View>(R.id.ratingBar) as RatingBar
ratingBar!!.onRatingBarChangeListener =
        OnRatingBarChangeListener {
          ratingBar, rating, fromTouch ->
           (findViewById<View>(R.id.textView) as TextView).text = "Rating:
    $rating"
        }
```

Getting Dates and Times from Users

- DatePicker control can be used to get a month, day, and year from the user
- The basic XML layout resource definition for a DatePicker follows:

```
<DatePicker
android:id="@+id/DatePicker01"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```



Getting Dates and Times from Users

- Your code can register to receive a method call when the date changes.

```
private val mDateSetListener: OnDateSetListener =
    //anonymous implementation of the OnDateSetListener
    OnDateSetListener { view, year, monthOfYear, dayOfMonth ->
        //event handler method
        yr = year
        month = monthOfYear
        day = dayOfMonth
        Toast.makeText(
            baseContext,
            ("You have selected : " + (month + 1) +
                    "/" + day + "/" + year),
            Toast.LENGTH_SHORT
        ).show()
    }
```

Getting Dates and Times from Users

```
private val mTimeSetListener: OnTimeSetListener =
    OnTimeSetListener { view, hourOfDay, minuteOfHour ->
        hour = hourOfDay
        minute = minuteOfHour
        val timeFormat = SimpleDateFormat("hh:mm aa")
        val date = Date(0, 0, 0, hour, minute)
        //format the date as string
        val strDate: String = timeFormat.format(date)
        //display it
        Toast.makeText(
            baseContext,
            "You have selected $strDate",
            Toast.LENGTH_SHORT
        ).show()
    }
```

Working with Styles

- Android user interface designers can **group layout element attributes together** in styles
- Styles are tagged with the <style> tag and should be stored in the /res/values/ directory.
- Style resources are defined in XML and compiled into the application binary at build time
- Styles in Android share a similar philosophy to cascading stylesheets in web design—they allow you to **separate the design from the content**.

Working with Styles

- For example, by using a style, you can take this layout XML:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="@string/hello" />
```

And turn it into this:

```
<TextView
    style = ”@style/mandatory_text_field_style”
    android:text="@string/hello" />
```

Working with Styles

- Here's an example of a simple style resource file /res/values/styles.xml containing two styles:
  - one for mandatory form fields, and one for optional form fields on TextView and EditText objects:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="mandatory_text_field_style">
    <item name="android:textColor">#000000</item>
    <item name="android:textSize">14pt</item>
    <item name="android:textStyle">bold</item>
    </style>
    <style name="optional_text_field_style">
    <item name="android:textColor">#0F0F0F</item>
    <item name="android:textSize">12pt</item>
    <item name="android:textStyle">italic</item>
    </style>
</resources>
```

Working with Styles

- Here's the **styles.xml** file again; this time, the color and text size fields are available in the other resource files: colors.xml and dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="mandatory_text_field_style">
    <item name="android:textColor">@color/mand_text_color</item>
    <item name="android:textSize">@dimen/important_text</item>
    <item name="android:textStyle">bold</item>
    </style>
    <style name="optional_text_field_style">
    <item name="android:textColor">@color/opt_text_color</item>
    <item name="android:textSize">@dimen/unimportant_text</item>
    <item name="android:textStyle">italic</item>
    </style>
</resources>
```

Working with Styles

- You can set each control's style attribute by referencing it as:

  style="@style/name_of_style"

  For example:

```
<TextView
android:id="@+id/TextView01"
style="@style/mandatory_text_field_style"
android:layout_height="wrap_content"
android:text="@string/mand_label"
android:layout_width="wrap_content" />
```

Working with Themes

- **Themes** are like styles, but instead of being applied to one layout element at a time, they **are applied to all elements of a given activity**.
- Themes are defined in exactly the same way as styles.
  - Themes use the <style> tag and should be stored in the /res/values directory.
- The only difference is that instead of applying that named style to a layout element, you define it as the **theme attribute of an activity in the AndroidManifest.xml** file

Working with Themes

- To set a theme for all the activities of your application, open the AndroidManifest.xml file and edit the <application> tag to include the android:theme attribute with the style name. For example:

  <**application** android:theme="@style/CustomTheme">

- To apply a theme to just one Activity in your application, add the android:theme attribute to the <activity> tag instead:

<**activity** android:theme="@android:style/CustomTheme">

- You can inherit built-in themes. For example:

  <activity android:theme="@android:style/Theme.**Dialog**">

will **make your activity to look like a dialog box**.

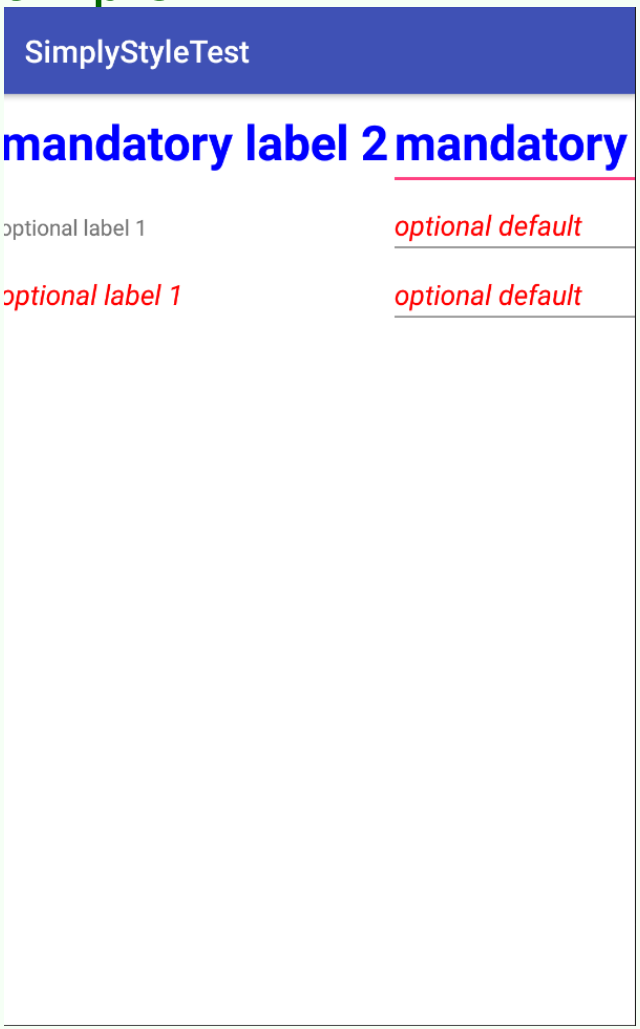
Working with Themes

- You can override whatever styles you want.
- For example, you can change the activity background color as follows:

```
<style name="AppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">
    ...
    <item
name="android:windowBackground">@color/activit
yBackground</item>
</style>
```

Styles Example

- SimpleStyles example



References

- Textbook
- Android Documentation:
  - https://developer.android.com/guide/topics/ui/menus
  - https://developer.android.com/guide/topics/ui/controls/button
  - https://developer.android.com/guide/topics/ui/controls/checkbox
  - https://developer.android.com/guide/topics/ui/controls/radiobutton
  - https://developer.android.com/guide/topics/ui/controls/togglebutton
  - https://developer.android.com/guide/topics/ui/controls/spinner
  - https://developer.android.com/guide/topics/ui/controls/pickers
  - https://developer.android.com/guide/topics/ui/look-and-feel/themes
- Lauren Darcey, Shane Conder: Introduction to Android Application Development: Android Essentials (5th Edition)