

Practical 2: Interactive UI

The user interface (UI) that appears on a screen of an Android device consists of a hierarchy of objects called *views* — every element of the screen is a View. The View class represents the basic building block for all UI components, and the base class for classes that provide interactive UI components such as buttons, checkboxes, and text entry fields. Commonly used View subclasses described over several lessons include:

- TextView for displaying text.
- EditText to enable the user to enter and edit text.
- Button and other clickable elements (such as RadioButton, CheckBox, and Spinner) to provide interactive behavior.
- ScrollView and RecyclerView to display scrollable items.
- ImageView for displaying images.
- ConstraintLayout and LinearLayout for containing other View elements and positioning them.

The Java code that displays and drives the UI is contained in a class that extends Activity. An Activity is usually associated with a layout of UI views defined as an XML (eXtended Markup Language) file. This XML file is usually named after its Activity and defines the layout of View elements on the screen.

For example, the MainActivity code in the Hello World app displays a layout defined in the activity_main.xml layout file, which includes a TextView with the text "Hello World".

In more complex apps, an Activity might implement actions to respond to user taps, draw graphical content, or request data from a database or the internet. You learn more about the Activity class in another lesson.

In this practical you learn how to create your first interactive app—an app that enables user interaction. You create an app using the Empty Activity template. You also learn how to use the layout editor to design a layout, and how to edit the layout in XML. You need to develop these skills so you can complete the other practical in this course.

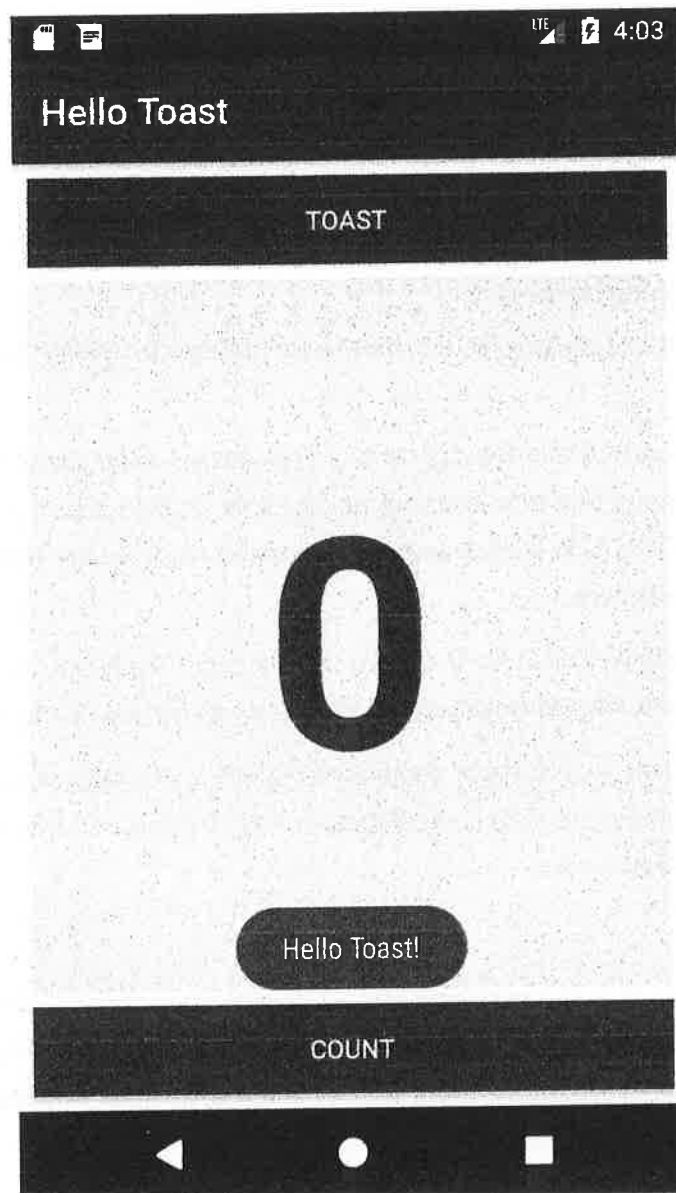
- Create an app and add two Button elements and a TextView to the layout.
- Manipulate each element in the ConstraintLayout to constrain them to the margins and other elements.
- Change UI element attributes.
- Edit the app's layout in XML.

- Extract hardcoded strings into string resources.
- Implement click-handler methods to display messages on the screen when the user taps each Button.

App overview

The HelloToast app consists of two Button elements and one TextView. When the user taps the first Button, it displays a short message (a Toast) on the screen. Tapping the second Button increases a "click" counter displayed in the TextView, which starts at zero.

Here's what the finished app looks like:



Task 1: Create and explore a new project

In this practical, you design and implement a project for the HelloToast app. A link to the solution code is provided at the end.

1.1 Create the Android Studio project

1. Start Android Studio and create a new project with the following parameters:

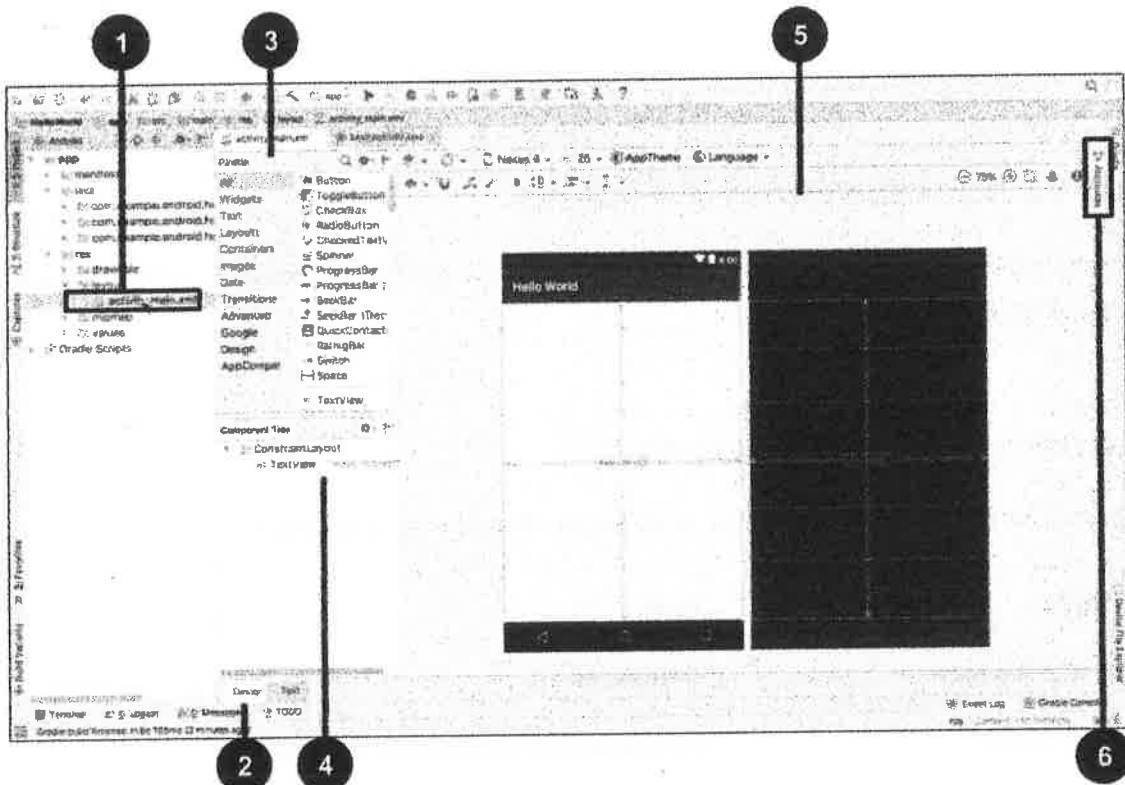
Attribute	Value
Application Name	Hello Toast
Company Name	com.example.android (or your own domain)
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Template	Empty Activity
Generate Layout file box	Selected
Backwards Compatibility box	Selected

2. Select **Run > Run app** or click the **Run icon**  in the toolbar to build and execute the app on the emulator or your device.

1.2 Explore the layout editor

Android Studio provides the layout editor for quickly building an app's layout of user interface (UI) elements. It lets you drag elements to a visual design and blueprint view, position them in the layout, add constraints, and set attributes. *Constraints* determine the position of a UI element within the layout. A constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline.

Explore the layout editor, and refer to the figure below as you follow the numbered steps:



1. In the **app > res > layout** folder in the **Project > Android** pane, double-click the **activity_main.xml** file to open it, if it is not already open.
2. Click the **Design** tab if it is not already selected. You use the **Design** tab to manipulate elements and the layout, and the **Text** tab to edit the XML code for the layout.
3. The **Palettes** pane shows UI elements that you can use in your app's layout.
4. The **Component tree** pane shows the view hierarchy of UI elements. View elements are organized into a tree hierarchy of parents and children, in which a child inherits the attributes of its parent. In the figure above, the **TextView** is a child of the **ConstraintLayout**. You will learn about these elements later in this lesson.
5. The design and blueprint panes of the layout editor showing the UI elements in the layout. In the figure above, the layout shows only one element: a **TextView** that displays "Hello World".
6. The **Attributes** tab displays the **Attributes** pane for setting properties for a UI element.

Tip: See [Building a UI with Layout Editor](#) for details on using the layout editor, and [Meet Android Studio](#) for the full Android Studio documentation.

Task 2: Add View elements in the layout editor


In this task you create the UI layout for the HelloToast app in the layout editor using the ConstraintLayout features. You can create the constraints manually, as shown later, or automatically using the **Autoconnect** tool.



2.1 Examine the element constraints

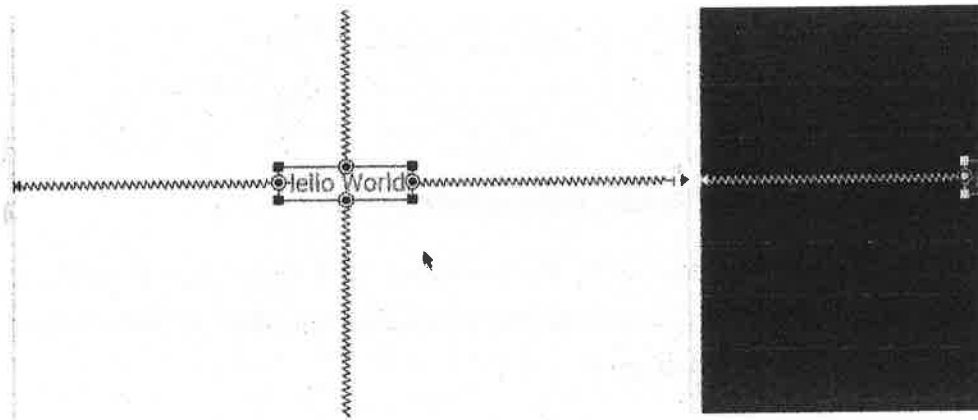
Follow these steps:

1. Open `activity_main.xml` from the **Project > Android** pane if it is not already open. If the **Design** tab is not already selected, click it.

If there is no blueprint, click the **Select Design Surface** button  in the toolbar and choose **Design + Blueprint**.

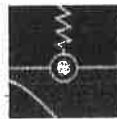
2. The **Autoconnect** tool  is also located in the toolbar. It is enabled by default. For this step, ensure that the tool is not disabled.

3. Click the zoom in  90%  button to zoom into the design and blueprint panes for a close-up look.
4. Select **TextView** in the Component Tree pane. The "Hello World" TextView is highlighted in the design and blueprint panes and the constraints for the element are visible.
5. Refer to the animated figure below for this step. Click the circular handle on the right side of the TextView to delete the horizontal constraint that binds the view to the right side of the layout. The TextView jumps to the left side because it is no longer constrained to the right side. To add back the horizontal constraint, click the same handle and drag a line to the right side of the layout.

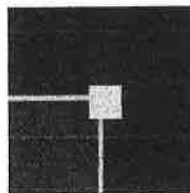


In the blueprint or design panes, the following handles appear on the `TextView` element:

- **Constraint handle:** To create a constraint as shown in the animated figure above, click a constraint handle, shown as a circle on the side of an element. Then drag the handle to another constraint handle, or to a parent boundary. A zigzag line represents the constraint.



- **Resizing handle:** To resize the element, drag the square resizing handles. The handle changes to an angled corner while you are dragging it.

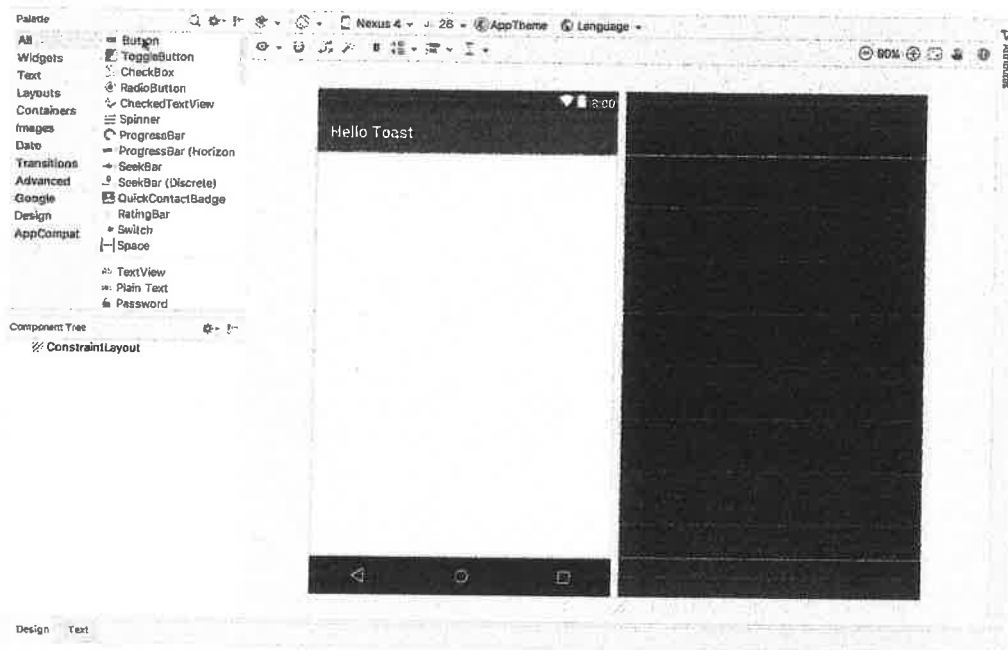


2.2 Add a Button to the layout

When enabled, the **Autoconnect** tool automatically creates two or more constraints for a UI element to the parent layout. After you drag the element to the layout, it creates constraints based on the element's position.

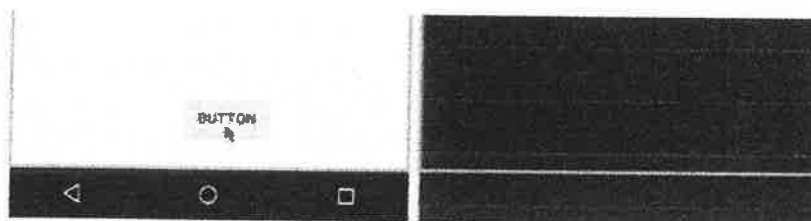
Follow these steps to add a Button:

1. Start with a clean slate. The `TextView` element is not needed, so while it is still selected, press the **Delete** key or choose **Edit > Delete**. You now have a completely blank layout.
2. Drag a **Button** from the **Palette** pane to any position in the layout. If you drop the Button in the top middle area of the layout, constraints may automatically appear. If not, you can drag constraints to the top, left side, and right side of the layout as shown in the animated figure below.




2.3 Add a second Button to the layout

1. Drag another **Button** from the **Palette** pane to the middle of the layout as shown in the animated figure below. Autoconnect may provide the horizontal constraints for you (if not, you can drag them yourself).
2. Drag a vertical constraint to the bottom of the layout (refer to the figure below).



You can remove constraints from an element by selecting the element and hovering your

pointer over it to show the  button. Click this button to remove *all* constraints on the selected element. To clear a single constraint, click the specific handle that sets the constraint.

To clear all constraints in the entire layout, click the **Clear All Constraints** tool in the toolbar. This tool is useful if you want to redo all the constraints in your layout.

Task 3: Change UI element attributes

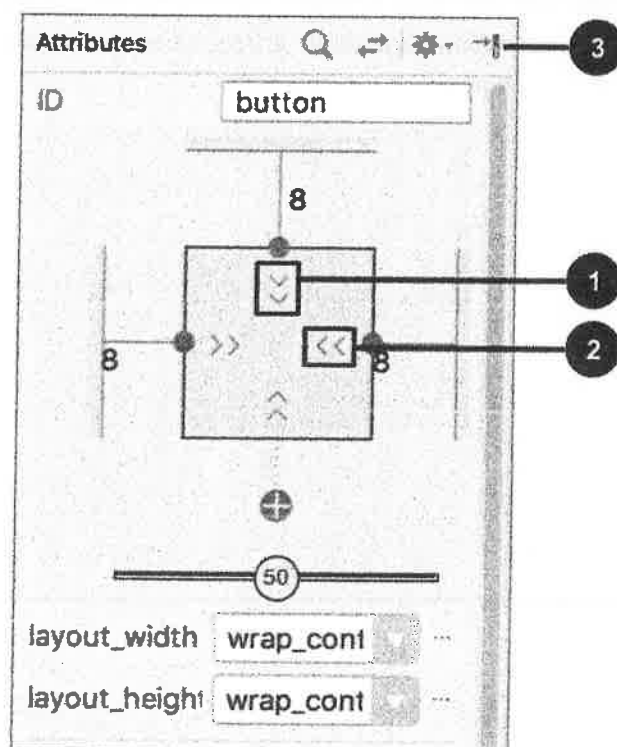
The **Attributes** pane offers access to all of the XML attributes you can assign to a UI element. You can find the attributes (known as *properties*) common to all views in the [View class documentation](#).

In this task you enter new values and change values for important Button attributes, which are applicable to most View types.

3.1 Change the Button size

The layout editor offers resizing handles on all four corners of a View so you can resize the View quickly. You can drag the handles on each corner of the View to resize it, but doing so hardcodes the width and height dimensions. Avoid hardcoding sizes for most View elements, because hardcoded dimensions can't adapt to different content and screen sizes.

Instead, use the **Attributes** pane on the right side of the layout editor to select a sizing mode that doesn't use hardcoded dimensions. The **Attributes** pane includes a square sizing panel called the *view inspector* at the top. The symbols inside the square represent the height and width settings as follows:

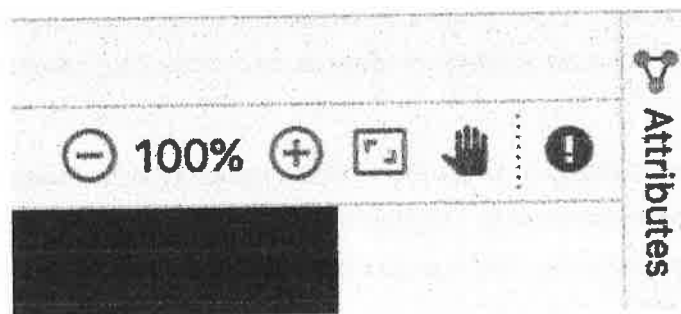


In the figure above:

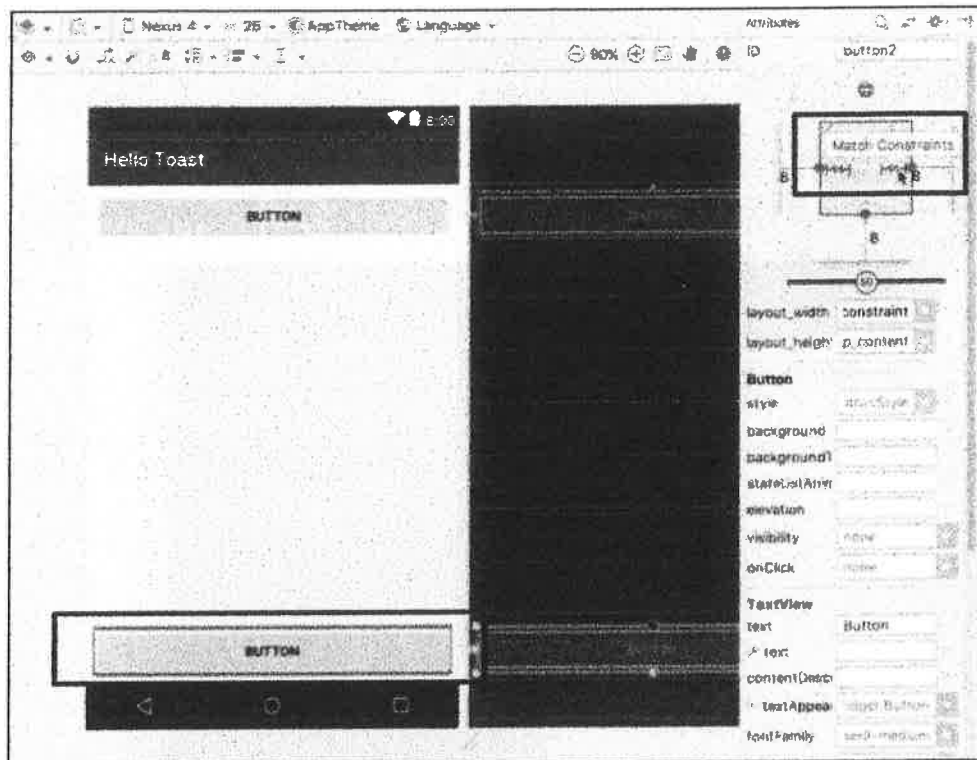
1. **Height control.** This control specifies the `layout_height` attribute and appears in two segments on the top and bottom sides of the square. The angles indicate that this control is set to `wrap_content`, which means the View will expand vertically as needed to fit its contents. The "8" indicates a standard margin set to 8dp.
2. **Width control.** This control specifies the `layout_width` and appears in two segments on the left and right sides of the square. The angles indicate that this control is set to `wrap_content`, which means the View will expand horizontally as needed to fit its contents, up to a margin of 8dp.
3. **Attributes pane close button.** Click to close the pane.

Follow these steps:

1. Select the top Button in the **Component Tree** pane.
2. Click the **Attributes** tab on the right side of the layout editor window.



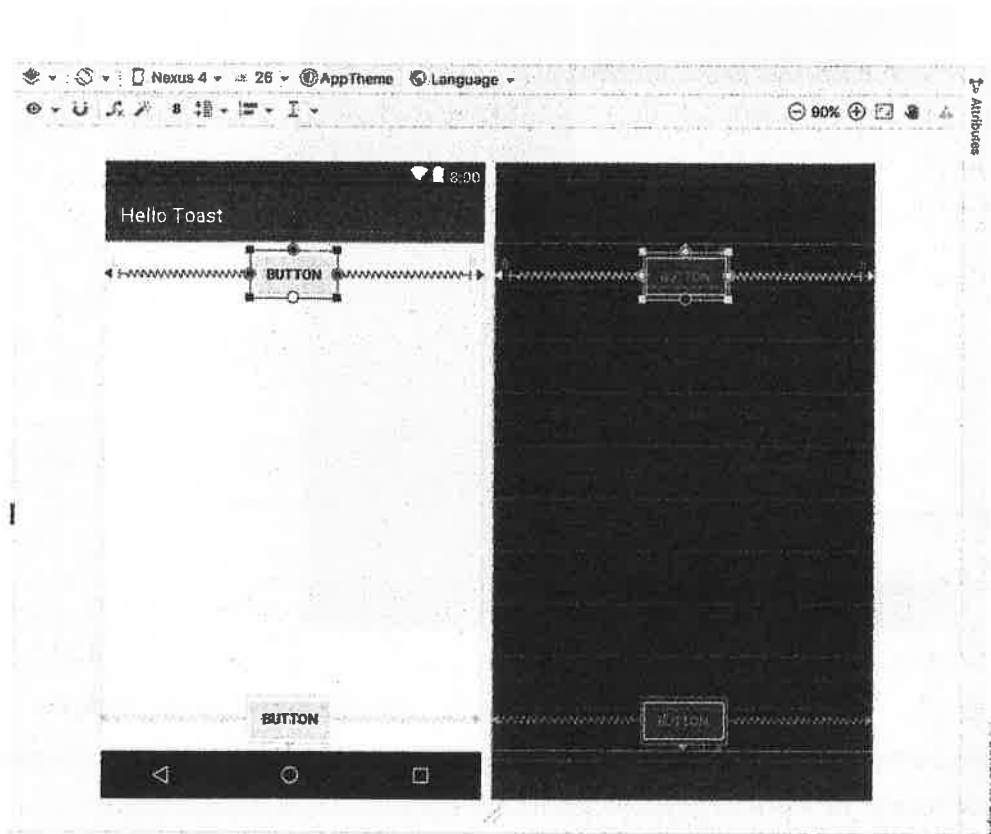
3. Click the width control twice—the first click changes it to **Fixed** with straight lines, and the second click changes it to **Match Constraints** with spring coils, as shown in the animated figure below.



As shown in the previous steps, the `layout_width` and `layout_height` attributes in the **Attributes** pane change as you change the height and width controls in the inspector. These attributes can take one of three values for the layout, which is a `ConstraintLayout`:

- The `match_constraint` setting expands the View element to fill its parent by width or height—up to a margin, if one is set. The parent in this case is the `ConstraintLayout`. You learn more about `ConstraintLayout` in the next task.
- The `wrap_content` setting shrinks the View element's dimensions so it is just big enough to enclose its content. If there is no content, the View element becomes invisible.
- To specify a fixed size that adjusts for the screen size of the device, use a fixed number of density-independent pixels (dp units). For example, 16dp means 16 density-independent pixels.

Tip: If you change the `layout_width` attribute using its popup menu, the `layout_width` attribute is set to zero because there is no set dimension. This setting is the same as `match_constraint`—the view can expand as much as possible to meet constraints and margin settings.



As a result of changing the width control, the `layout_width` attribute in the **Attributes** pane shows the value `match_constraint` and the Button element stretches horizontally to fill the space between the left and right sides of the layout.

4. Select the second Button, and make the same changes to the `layout_width` as in the previous step, as shown in the figure below.

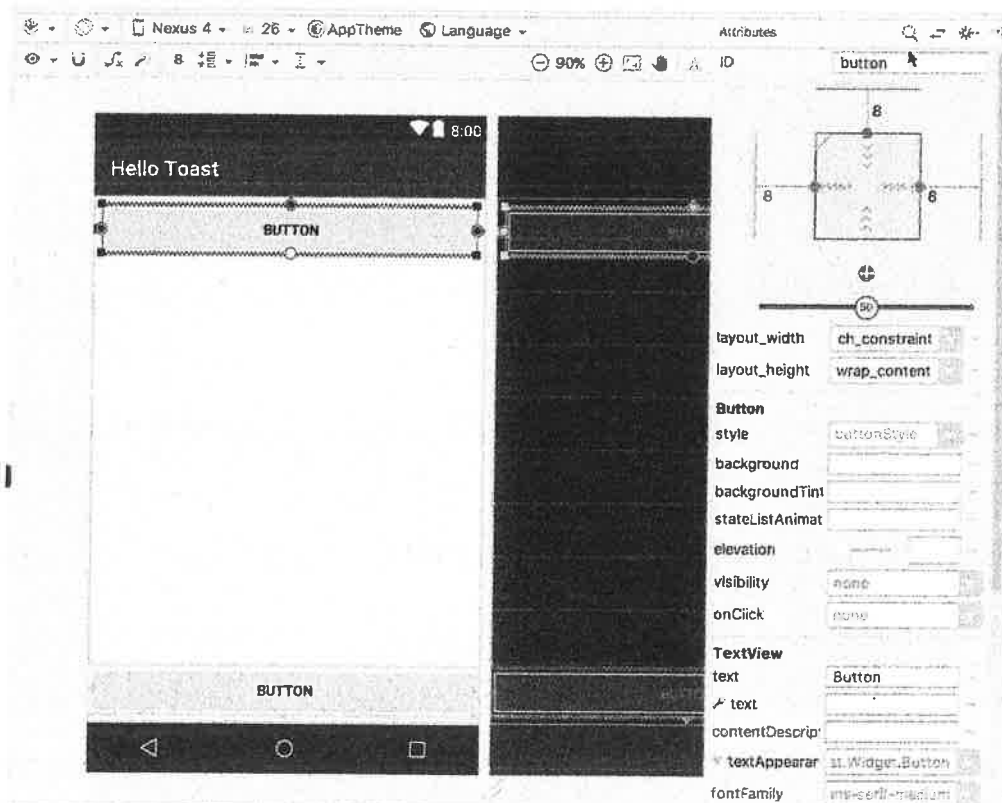
3.2 Change the Button attributes

To identify each View uniquely within an Activity layout, each View or View subclass (such as Button) needs a unique ID. And to be of any use, the Button elements need text. View elements can also have backgrounds that can be colors or images.

The **Attributes** pane offers access to all of the attributes you can assign to a View element. You can enter values for each attribute, such as the `android:id`, `background`, `textColor`, and `text` attributes.

The following animated figure demonstrates how to perform these steps:

1. After selecting the first Button, edit the ID field at the top of the **Attributes** pane to `button_toast` for the `android:id` attribute, which is used to identify the element in the layout.
2. Set the background attribute to `@color/colorPrimary`. (As you enter `@c`, choices appear for easy selection.)
3. Set the `textColor` attribute to `@android:color/white`.
4. Edit the text attribute to `Toast`.



5. Perform the same attribute changes for the second Button, using `button_count` as the ID, `Count` for the text attribute, and the same colors for the background and text as the previous steps.

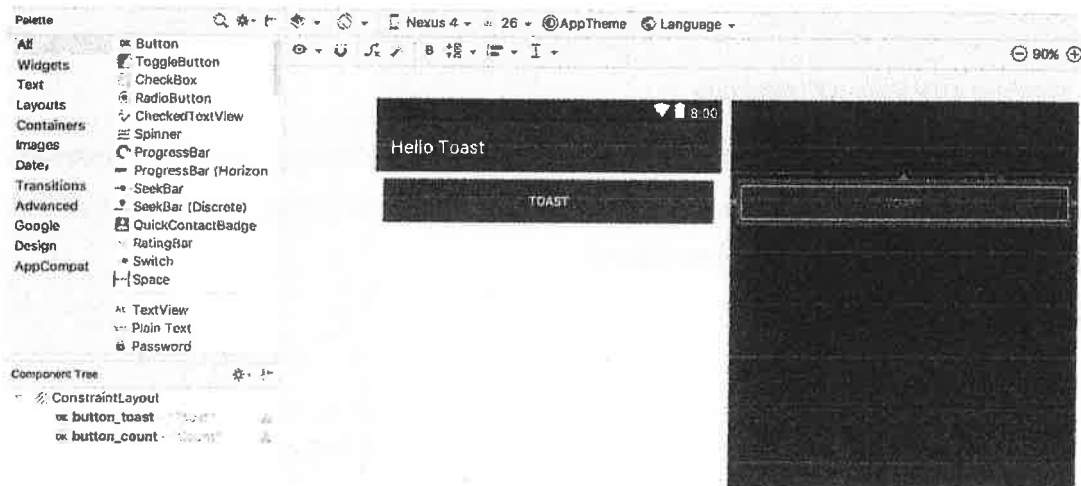
The `colorPrimary` is the primary color of the theme, one of the predefined theme base colors defined in the `colors.xml` resource file. It is used for the app bar. Using the base colors for other UI elements creates a uniform UI. You will learn more about app themes and Material Design in another lesson.

Task 4: Add a `TextView` and set its attributes

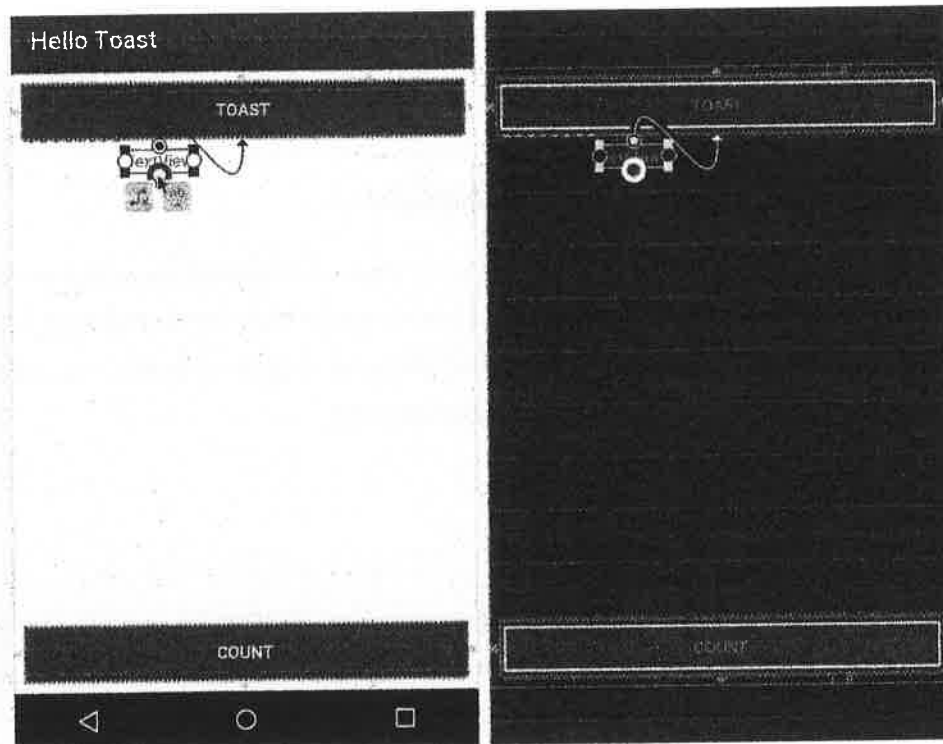
One of the benefits of `ConstraintLayout` is the ability to align or otherwise constrain elements relative to other elements. In this task you will add a `TextView` in the middle of the layout, and constrain it horizontally to the margins and vertically to the two `Button` elements. You will then change the attributes for the `TextView` in the **Attributes** pane.

4.1 Add a `TextView` and constraints

1. As shown in the animated figure below, drag a `TextView` from the **Palette** pane to the upper part of the layout, and drag a constraint from the top of the `TextView` to the handle on the bottom of the **Toast Button**. This constrains the `TextView` to be underneath the `Button`.



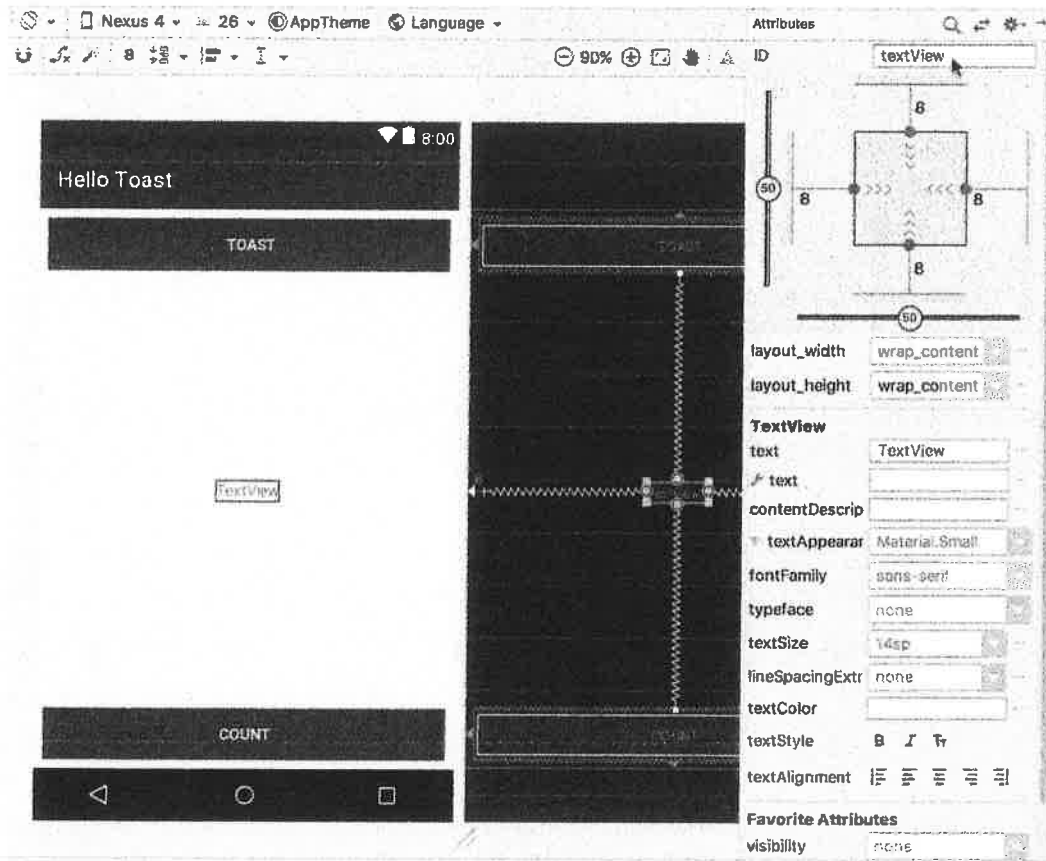
2. As shown in the animated figure below, drag a constraint from the bottom of the `TextView` to the handle on the top of the **Count** Button, and from the sides of the `TextView` to the sides of the layout. This constrains the `TextView` to be in the middle of the layout between the two `Button` elements.



4.2 Set the TextView attributes

With the `TextView` selected, open the **Attributes** pane, if it is not already open. Set attributes for the `TextView` as shown in the animated figure below. The attributes you haven't encountered yet are explained after the figure:

1. Set the ID to `show_count`.
2. Set the text to `0`.
3. Set the `textSize` to `160sp`.
4. Set the `textStyle` to **B** (bold) and the `textAlignment` to `ALIGNCENTER` (center the paragraph).
5. Change the horizontal and vertical view size controls (`layout_width` and `layout_height`) to `match_constraint`.
6. Set the `textColor` to `@color/colorPrimary`.
7. Scroll down the pane and click **View all attributes**, scroll down the second page of attributes to `background`, and then enter `#FFFF00` for a shade of yellow.
8. Scroll down to `gravity`, expand `gravity`, and select `center_ver` (for center-vertical).



- **textSize:** The text size of the TextView. For this lesson, the size is set to 160sp. The sp stands for *scale-independent pixel*, and like dp, is a unit that scales with the screen density and user's font size preference. Use dp units when you specify font sizes so that the sizes are adjusted for both the screen density and the user's preference.
- **textStyle and textAlignment:** The text style, set to **B** (bold) in this lesson, and the text alignment, set to **ALIGN_CENTER** (center the paragraph).
- **gravity:** The gravity attribute specifies how a View is aligned within its *parent* View or ViewGroup. In this step, you center the TextView to be centered vertically within the parent ConstraintLayout.

You may notice that the background attribute is on the first page of the **Attributes** pane for a Button, but on the second page of the **Attributes** pane for a TextView.

The **Attributes** pane changes for each type of View: The most popular attributes for the View type appear on the first page, and the rest are listed on the second page. To return

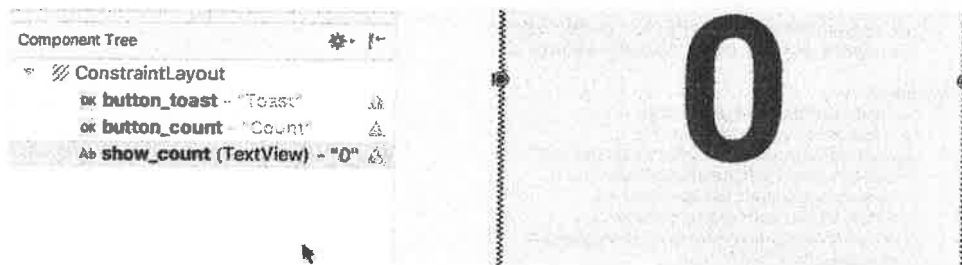
to the first page of the **Attributes** pane, click the



icon in the toolbar at the top of the pane.

Task 5: Edit the layout in XML

The Hello Toast app layout is nearly finished! However, an exclamation point appears next to each UI element in the Component Tree. Hover your pointer over these exclamation points to see warning messages, as shown below. The same warning appears for all three elements: **hardcoded strings should use resources.**



The easiest way to fix layout problems is to edit the layout in XML. While the layout editor is a powerful tool, some changes are easier to make directly in the XML source code.

5.1 Open the XML code for the layout

For this task, open the `activity_main.xml` file if it is not already open, and click

the **Text** tab at the bottom of the layout editor.

The XML editor appears, replacing the design and blueprint panes. As you can see in the figure below, which shows part of the XML code for the layout, the warnings are highlighted—the hardcoded strings `"Toast"` and `"Count"`. (The hardcoded `"0"` is also highlighted but not shown in the figure.) Hover your pointer over the hardcoded string `"Toast"` to see the warning message.

activity_main.xml MainActivity.java

```

1  android.support.constraint.ConstraintLayout Button
2  <?xml version="1.0" encoding="utf-8"?>
3  <android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
4  xmlns:app="http://schemas.android.com/apk/res-auto"
5  xmlns:tools="http://schemas.android.com/tools"
6  android:layout_width="match_parent"
7  android:layout_height="match_parent"
8  tools:context="com.example.android.hellotoast.MainActivity">
9
10     <Button
11         android:id="@+id/button_toast"
12         android:layout_width="0dp"
13         android:layout_height="wrap_content"
14         android:layout_marginEnd="8dp"
15         android:layout_marginStart="8dp"
16         android:layout_marginTop="8dp"
17         android:background="@color/colorPrimary"
18         android:text="Toast"
19         android:textColor="@android:color/white"
20         app:layout_constraintStart_toStartOf="parent"
21         app:layout_constraintTop_toTopOf="parent" />
22
23     <Button
24         android:id="@+id/button_count"
25         android:layout_width="0dp"
26         android:layout_height="wrap_content"
27         android:layout_marginBottom="8dp"
28         android:layout_marginEnd="8dp"
29         android:layout_marginStart="8dp"
30         android:background="@color/colorPrimary"
31         android:text="Count"
32         android:textColor="@android:color/white"
33         app:layout_constraintBottom_toBottomOf="parent"
34         app:layout_constraintEnd_toEndOf="parent"
35         app:layout_constraintStart_toStartOf="parent" />
36
37     <TextView
38         android:id="@+id/show_count"
39         android:layout_width="0dp"
40         android:layout_height="0dp"
41         android:layout_marginBottom="8dp"

```

Hardcoded string "Toast", should use @string resource more... (%F1)

5.2 Extract string resources

Instead of hard-coding strings, it is a best practice to use string resources, which represent the strings. Having the strings in a separate file makes it easier to manage them, especially if you use these strings more than once. Also, string resources are mandatory for translating and localizing your app, because you need to create a string resource file for each language.

1. Click once on the word "Toast"(the first highlighted warning).
2. Press **Alt-Enter** in Windows or **Option-Enter** in macOS and choose **Extract string resource** from the popup menu.
3. Enter **button_label_toast** for the **Resource name**.
4. Click **OK**. A string resource is created in the **values/res/string.xml** file, and the string in your code is replaced with a reference to the resource:

@string/button_label_toast

5. Extract the remaining strings: **button_label_count** for "Count", and **count_initial_value** for "0".

6. In the **Project > Android** pane, expand **values** within **res**, and then double-click **strings.xml** to see your string resources in the `strings.xml` file:

```
<resources>
  <string name="app_name">Hello Toast</string>
  <string name="button_label_toast">Toast</string>
  <string name="button_label_count">Count</string>
  <string name="count_initial_value">0</string>
</resources>
```

7. You need another string to use in a subsequent task that displays a message. Add to the `strings.xml` file another string resource named `toast_message` for the phrase "Hello Toast!":

```
<resources>
  <string name="app_name">Hello Toast</string>
  <string name="button_label_toast">Toast</string>
  <string name="button_label_count">Count</string>
  <string name="count_initial_value">0</string>
  <string name="toast_message">Hello Toast!</string> - + rfp manually
</resources>
```

Tip: The string resources include the app name, which appears in the app bar at the top of the screen if you start your app project using the Empty Template. You can change the app name by editing the `app_name` resource.

Task 6: Add onClick handlers for the buttons

In this task, you add a Java method for each Button in MainActivity that executes when the user taps the Button.

6.1 Add the onClick attribute and handler to each Button

A *click handler* is a method that is invoked when the user clicks or taps on a clickable UI element. In Android Studio you can specify the name of the method in the `onClick` field in the **Design** tab's **Attributes** pane. You can also specify the name of the handler method in the XML editor by adding the `android:onClick` property to the Button. You will use the latter method because you haven't yet created the handler methods, and the XML editor provides an automatic way to create those methods.

1. With the XML editor open (the Text tab), find the Button with the `android:id` set to `button_toast`:

```
<Button
    android:id="@+id/button_toast"
    android:layout_width="0dp"
    ...
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

2. Add the `android:onClick` attribute to the end of the `button_toast` element after the last attribute and before the `/>` end indicator:

```
    android:onClick="showToast" />
```

3. Click the red bulb icon that appears next to attribute. Select **Create click handler**, choose **MainActivity**, and click **OK**.

If the red bulb icon doesn't appear, click the method name ("`showToast`"). Press **Alt-Enter** (**Option-Enter** on the Mac), select **Create 'showToast(view)' in MainActivity**, and click **OK**.

This action creates a placeholder method stub for the `showToast()` method in MainActivity, as shown at the end of these steps.

4. Repeat the last two steps with the button_count Button: Add the android:onClick attribute to the end, and add the click handler:

```
android:onClick="countUp" />
```

The XML code for the UI elements within the ConstraintLayout now looks like this:

```
<Button
    android:id="@+id/button_toast"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:background="@color/colorPrimary"
    android:text="@string/button_label_toast"
    android:textColor="@android:color/white"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:onClick="showToast"/>

<Button
    android:id="@+id/button_count"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:background="@color/colorPrimary"
    android:text="@string/button_label_count"
    android:textColor="@android:color/white"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    android:onClick="countUp" />

<TextView
    android:id="@+id/show_count"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
```


Follow these steps to edit the **Toast** Button click handler:

1. Locate the newly created `showToast()` method.

```
public void showToast(View view) {  
}
```

2. To create an instance of a **Toast**, call the `makeText()` factory method on the **Toast** class.

```
public void showToast(View view) {  
    Toast toast = Toast.makeText(  
}
```

This statement is incomplete until you finish all of the steps.

3. Supply the context of the app Activity. Because a **Toast** displays on top of the Activity UI, the system needs information about the current Activity. When you are already within the context of the Activity whose context you need, use this as a shortcut.

```
    Toast toast = Toast.makeText(this,
```

4. Supply the message to display, such as a string resource (the `toast_message` you created in a previous step). The string resource `toast_message` is identified by `R.string`.

```
    Toast toast = Toast.makeText(this, R.string.toast_message,
```

5. Supply a duration for the display. For example, `Toast.LENGTH_SHORT` displays the toast for a relatively short time.

```
    Toast toast = Toast.makeText(this, R.string.toast_message,  
                                Toast.LENGTH_SHORT);
```

The duration of a **Toast** display can be either `Toast.LENGTH_LONG` or `Toast.LENGTH_SHORT`. The actual lengths are about 3.5 seconds for the long **Toast** and 2 seconds for the short **Toast**.

6. Show the **Toast** by calling `show()`. The following is the entire `showToast()` method:

```
public void showToast(View view) {  
    class - Toast toast = Toast.makeText(this, R.string.toast_message,  
                                Toast.LENGTH_SHORT);  
    toast.show();  
}
```

Handwritten annotations:
- "attribute with method" points to `toast`
- "resource" points to `R.string.toast_message`
- "show method" points to `toast.show()`
- "Object reference" points to the `toast` variable in the class declaration.

```

        android:background="#FFFF00"
        android:gravity="center_vertical"
        android:text="@string/count_initial_value"
        android:textAlignment="center"
        android:textColor="@color/colorPrimary"
        android:textSize="160sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toTopOf="@+id/button_count"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button_toast" />

```

5. If `MainActivity.java` is not already open, expand `java` in the `Project > Android view`, expand `com.example.android.hellotoast`, and then double-click **MainActivity**. The code editor appears with the code in `MainActivity`:

```

package com.example.android.hellotoast;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void showToast(View view) {

    }

    public void countUp(View view) {

    }
}

```

6.2 Edit the Toast Button handler

You will now edit the `showToast()` method—the **Toast** Button click handler in `MainActivity`—so that it shows a message. A **Toast** provides a way to show a simple message in a small popup window. It fills only the amount of space required for the message. The current activity remains visible and interactive. A Toast can be useful for testing interactivity in your app—add a Toast message to show the result of tapping a Button or performing an action.

6.3 Edit the Count Button handler

You will now edit the `countUp()` method—the **Count** Button click handler in `MainActivity`—so that it displays the current count after **Count** is tapped. Each tap increases the count by one.

The code for the handler must:

- Keep track of the count as it changes.
- Send the updated count to the `TextView` to display it.

Follow these steps to edit the **Count** Button click handler:

1. Locate the newly created `countUp()` method.

```
public void countUp(View view) {  
}
```

2. To keep track of the count, you need a private member variable. Each tap of the **Count** button increases the value of this variable. Enter the following, which will be highlighted in red and show a red bulb icon:

```
public void countUp(View view) {  
    mCount++;  
}
```

If the red bulb icon doesn't appear, select the `mCount++` expression. The red bulb eventually appears.

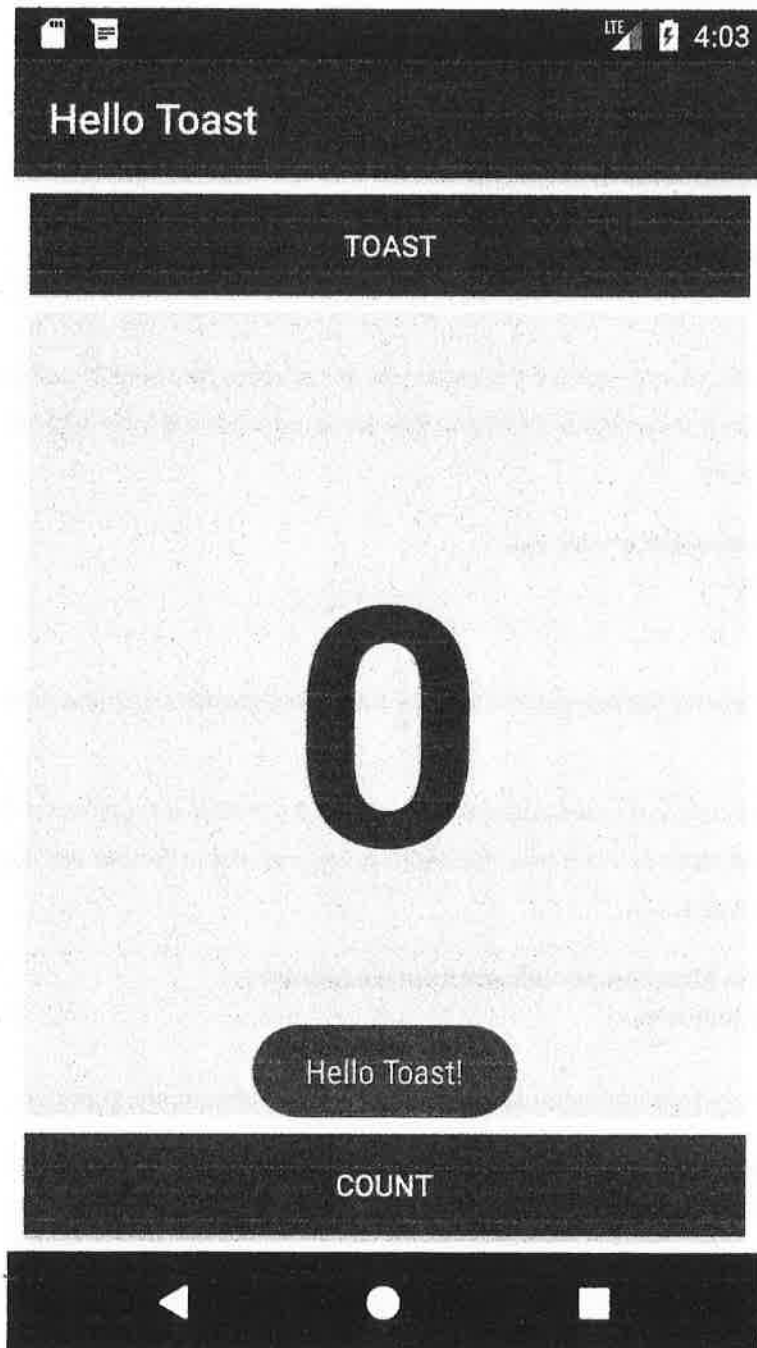
3. Click the red bulb icon and choose **Create field 'mCount'** from the popup menu. This creates a private member variable at the top of `MainActivity`, and Android Studio assumes that you want it to be an integer (`int`):

```
public class MainActivity extends AppCompatActivity {  
    private int mCount;  
}
```

4. Change the private member variable statement to initialize the variable to zero:

```
public class MainActivity extends AppCompatActivity {  
    private int mCount = 0;  
}
```

Run the app and verify that the Toast message appears when the **Toast** button is tapped.



5. Along with the variable above, you also need a private member variable for the reference of the show_count TextView, which you will add to the click handler. Call this variable mShowCount:

```
public class MainActivity extends AppCompatActivity {  
    private int mCount = 0;  
    private TextView mShowCount;
```

6. Now that you have mShowCount, you can get a reference to the TextView using the ID you set in the layout file. In order to get this reference only once, specify it in the onCreate() method. As you learn in another lesson, the onCreate() method is used to *inflate the layout*, which means to set the content view of the screen to the XML layout. You can also use it to get references to other UI elements in the layout, such as the TextView. Locate the onCreate() method in MainActivity:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
}
```

7. Add the findViewById statement to the end of the method:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    mShowCount = (TextView) findViewById(R.id.show_count);  
}
```

A View, like a string, is a resource that can have an id. The findViewById call takes the ID of a view as its parameter and returns the View. Because the method returns a View, you have to cast the result to the view type you expect, in this case (TextView).

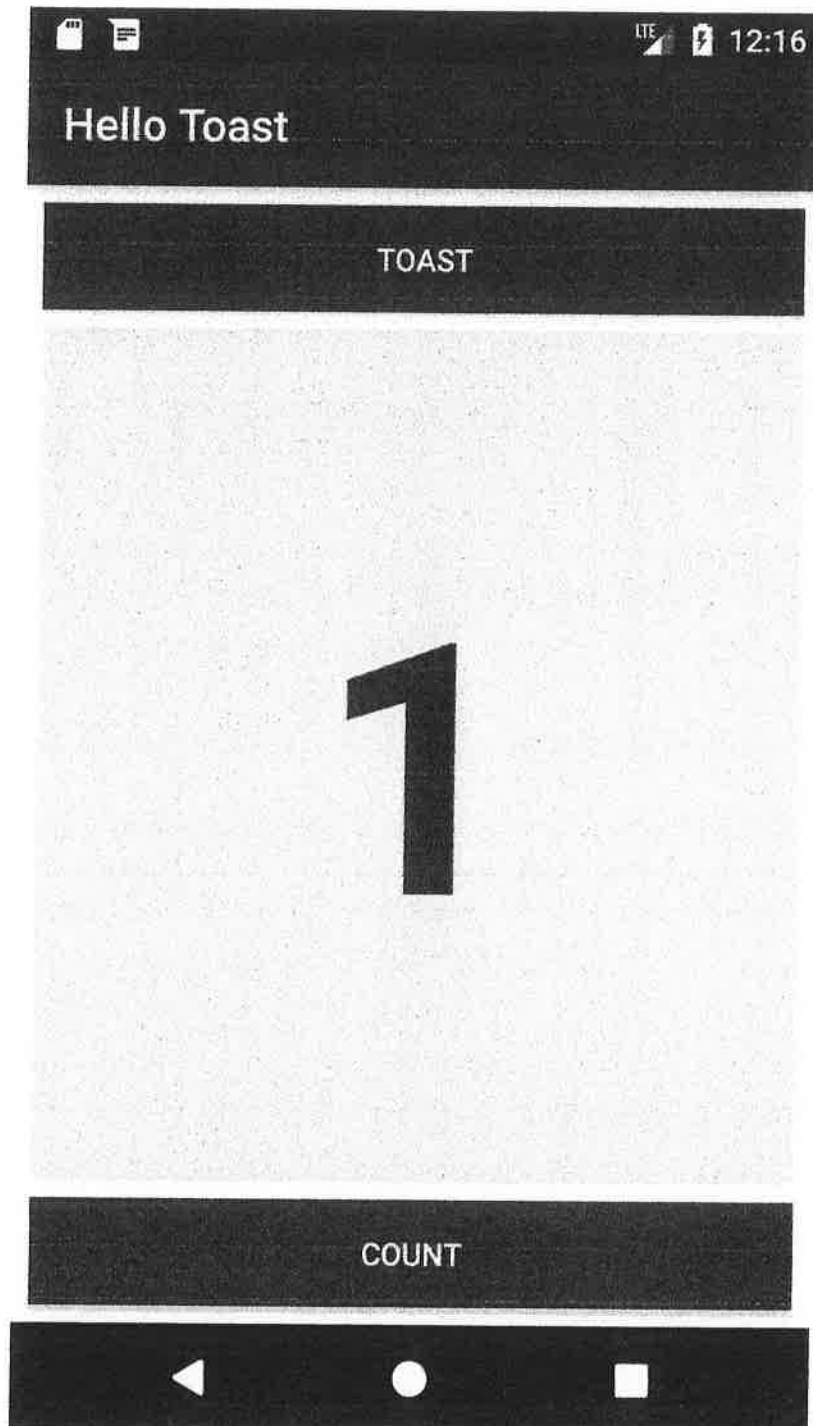
8. Now that you have assigned to mShowCount the TextView, you can use the variable to set the text in the TextView to the value of the mCount variable. Add the following to the countUp() method:

```
if (mShowCount != null)  
    mShowCount.setText(Integer.toString(mCount));
```

The entire `countUp()` method now looks like this:

```
public void countUp(View view) {  
    ++mCount;  
    if (mShowCount != null)  
        mShowCount.setText(Integer.toString(mCount));  
}
```

9. Run the app to verify that the count increases when you tap the **Count** button.



Tip: For an in-depth tutorial on using `ConstraintLayout`, see the Codelab [Using `ConstraintLayout` to design your views](#).