Revised: 12/19/2019

Event-Driven Programming II

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Android Programming
  - Creating an Activity
  - TextViews
  - EditTexts
  - Buttons
  - SeekBars
  - Running and Testing the App

- Exception Handling
  - Unchecked and Checked
  - try…catch statements
- Error Messages
  - Default Error Messages
  - Call Stack
- Uncaught Exceptions
- Handling Multiple Exceptions
- The Exception Object
- Finally Clauses
- Throwing Exceptions

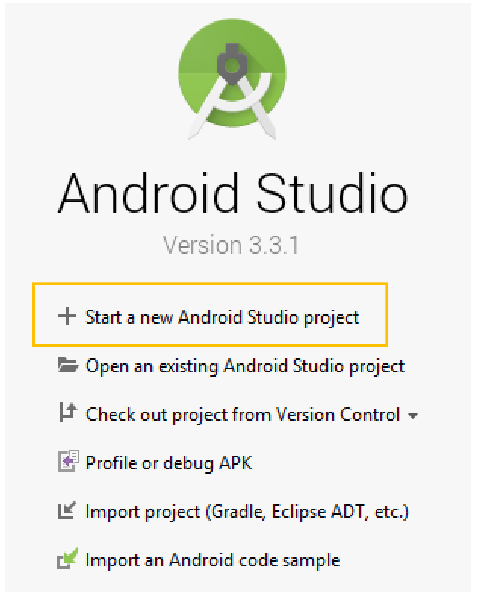
Colors/Fonts

Local Variable Names	– Brown
Primitive data types	– Fuchsia
Literals	– Blue
Keywords	– Orange
Object names	– Green
Operators/Punctuation	– Black
Field Names	– Lt Blue
Method Names	– Purple
Parameter Names	– Gold
Comments	– Gray
Package Names	– Pink

Source Code	– **Consolas**
Output	– `Courier New`

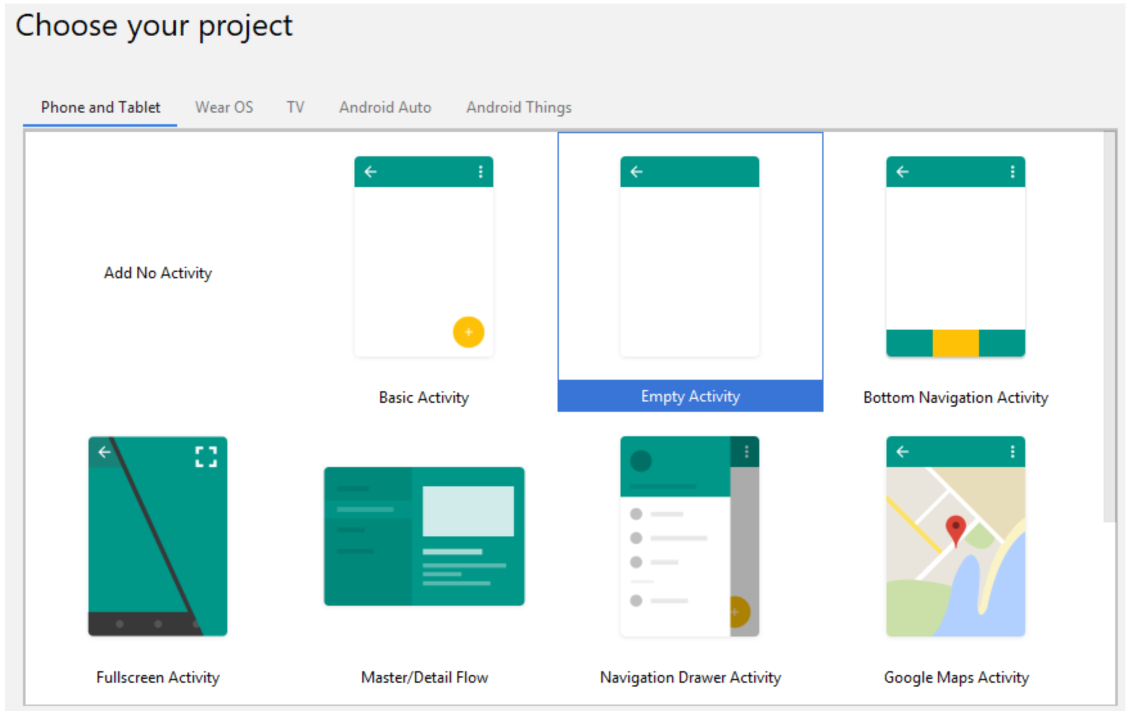
Create a new project in Android Studio

- Open Android Studio and start a new project.
  - If an existing project is already open, close it by…
    - Going to the **File** menu and selecting **Close Project**


Android Studio
Version 3.3.1
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample

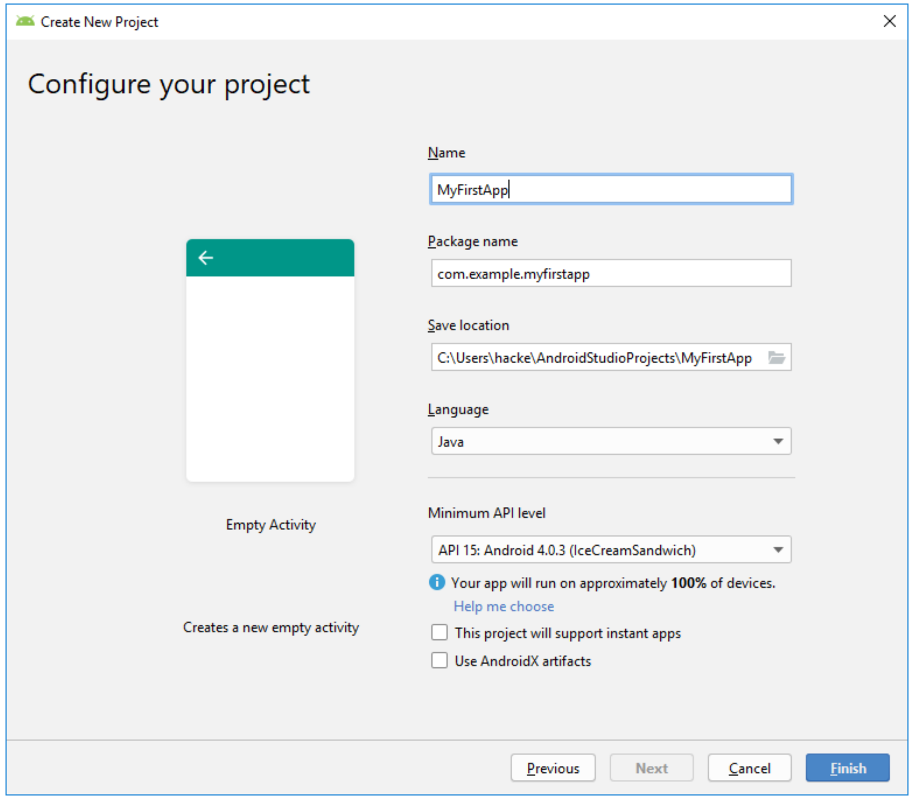
Create a new project in Android Studio

- Make sure **Empty Activity** is selected and click **Next**



Create a new project in Android Studio

- Name the project **MyFirstApp**
    - **Do not change any other settings.**
- Click the **Finish** button



Name: MyFirstApp

Package name: com.example.myfirstapp

Save location: C:\Users\hacke\AndroidStudioProjects\MyFirstApp

Language: Java

Minimum API level: API 15: Android 4.0.3 (IceCreamSandwich)

Your app will run on approximately **100%** of devices.

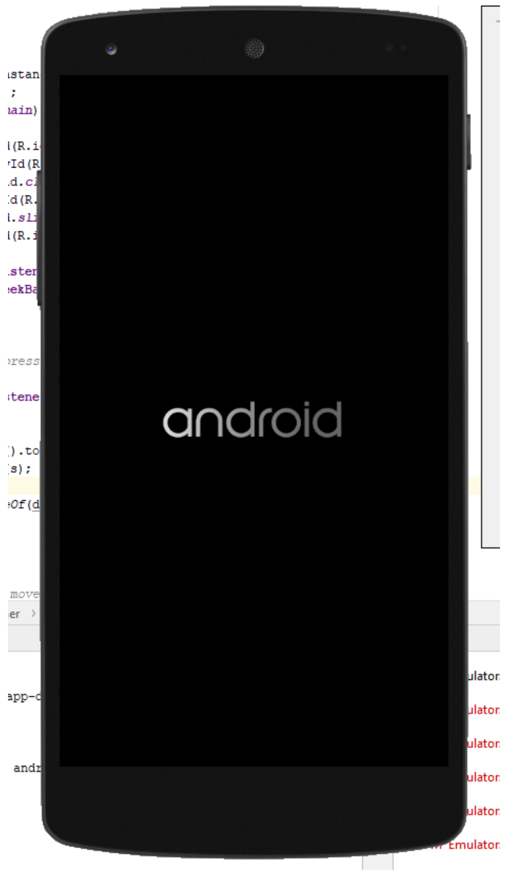
Help me choose

☐ This project will support instant apps

☐ Use AndroidX artifacts

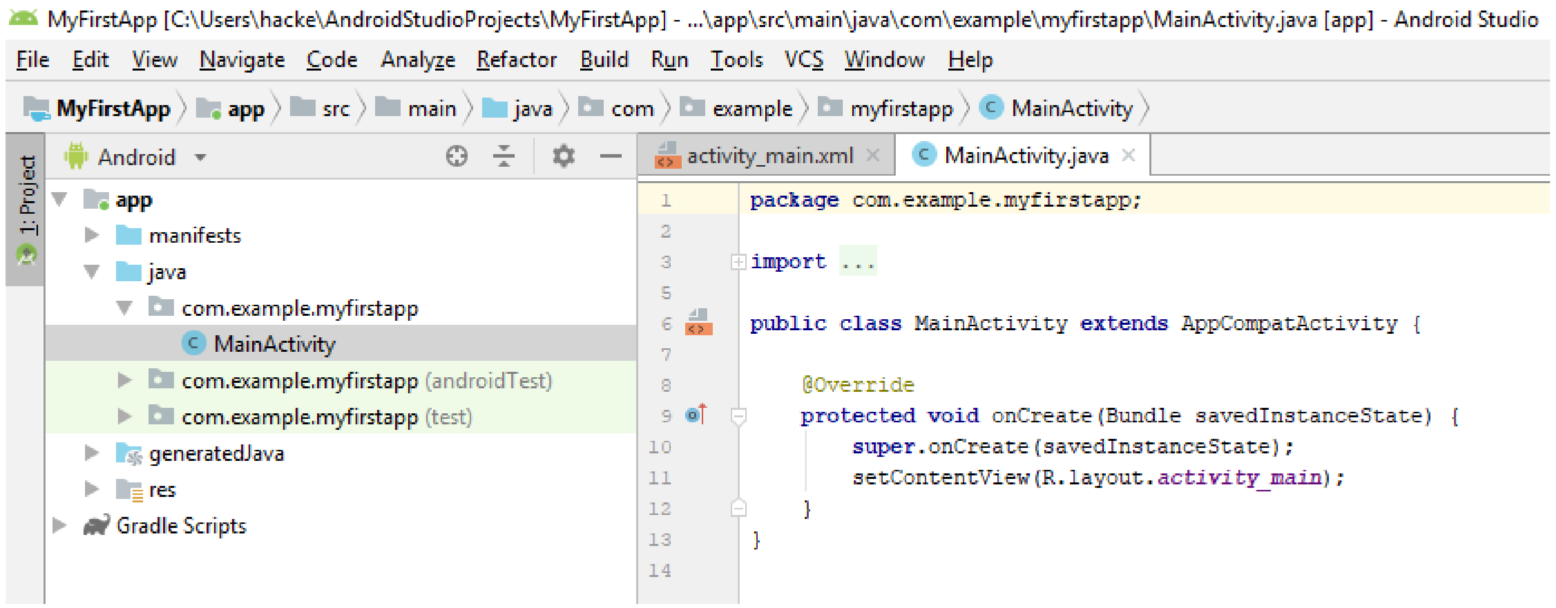
Start the AVD

- Start your Android Virtual Device now.
  - This way, we don't have to wait for it to load once we're ready to run our app.
- If you don't have an AVD (or don't know what one is)…
  - Go back complete the Android Virtual Devices instructions posted in this Canvas Module.



Source Code

- MainActivity.java will be automatically generated for us.
  - This is where we will put our source code for our app.


MyFirstApp [C:\Users\hacke\AndroidStudioProjects\MyFirstApp] - ...\app\src\main\java\com\example\myfirstapp\MainActivity.java [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyFirstApp app src main java com example myfirstapp MainActivity
1: Project
Android
app
manifests
java
com.example.myfirstapp
MainActivity
com.example.myfirstapp (androidTest)
com.example.myfirstapp (test)
generatedJava
res
Gradle Scripts
activity_main.xml
MainActivity.java

```
package com.example.myfirstapp;

import ...

public class MainActivity extends AppCompatActivity {

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

Activities

- An **Activity** is an Android object that provides a screen with which users can interact with.
    - Each activity is given a window in which to draw its user interface.
- An application usually consists of multiple activities.
    - The apps for this course will only consist of one activity.
- You can see that MainActivity.java is a subclass of something called AppCompatActivity.
    - (Which is a decendant/subclass of the Activity class)

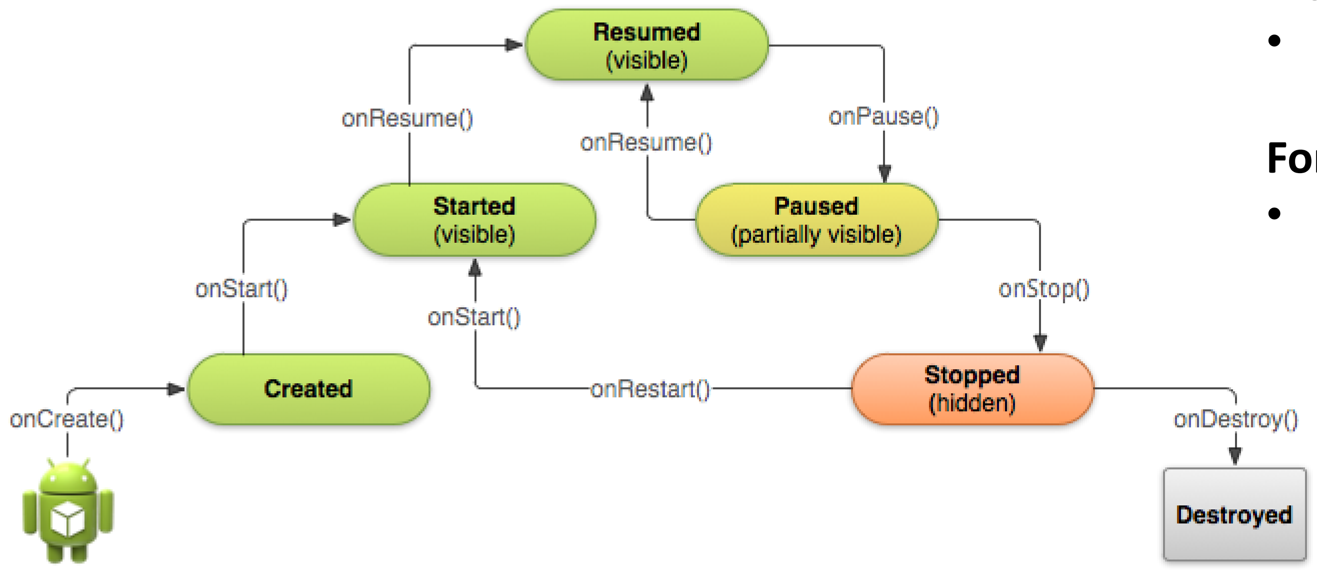
Activities

- In an application, one Activity may be specified as a “main activity”, that is first presented when launching the application.

- Each Activity is able to other activities.
  - When a new Activity starts, the previous activity is stopped and preserved in a stack.
  - Sort-of “put in the background”

- Activities use **callback methods** that the system calls when the activity transitions between various states of its **lifecycle**, such as when the activity is being created, stopped, resumed, or destroyed.

The Activity Lifecycle



Entire Lifetime

- Between onCreate() and onDestroy()

Visible Lifetime

- Between onStart() and onStop()

Foreground Lifetime

- Between onResume() and onPause()

Activities

- **onCreate()** The system will call this method when creating your Activity. In this method, you should initialize the essential components of your Activity.

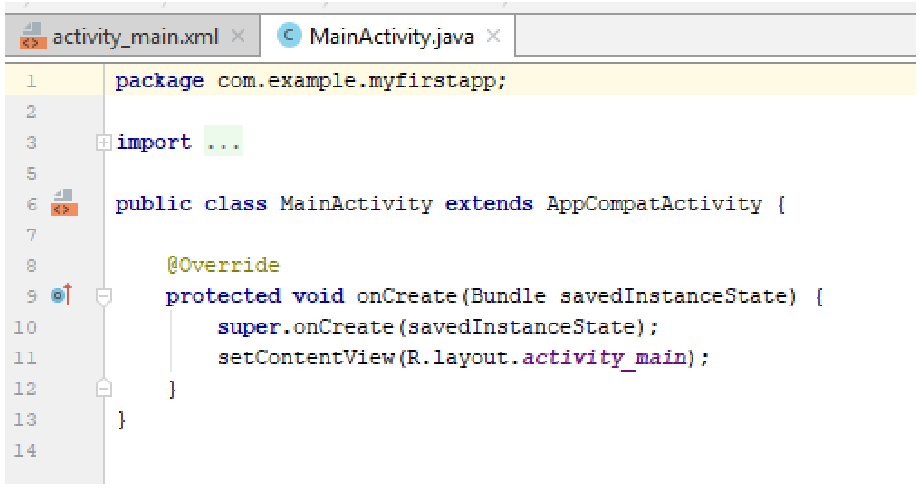
- **onPause()** The system will call this method at the first indication that the user is leaving the Activity (though it does not always mean the activity is being destroyed).
  - You should commit any changes that should be persisted beyond the current user session in this method.

Activities

- **onStart()** Called just before the Activity becomes visible to the user.
- **onResume()** Called just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack with user input going to it.
- **onStop()** Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.
- **onDestroy()** Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing, or because the system is temporarily destroying this instance of the Activity to save space.

Activities

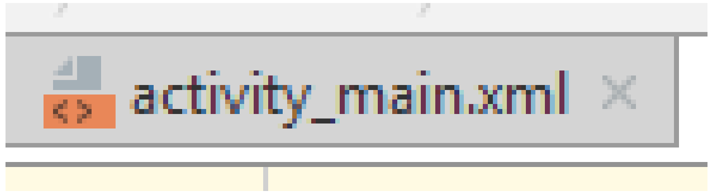
- For the apps you'll make in this course, you'll simply put any code in the onCreate() method.
  - Already generated for us.
  - Think of it as your "main method" like in a normal Java program.



```
package com.example.myfirstapp;

import ...

public class MainActivity extends AppCompatActivity {

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

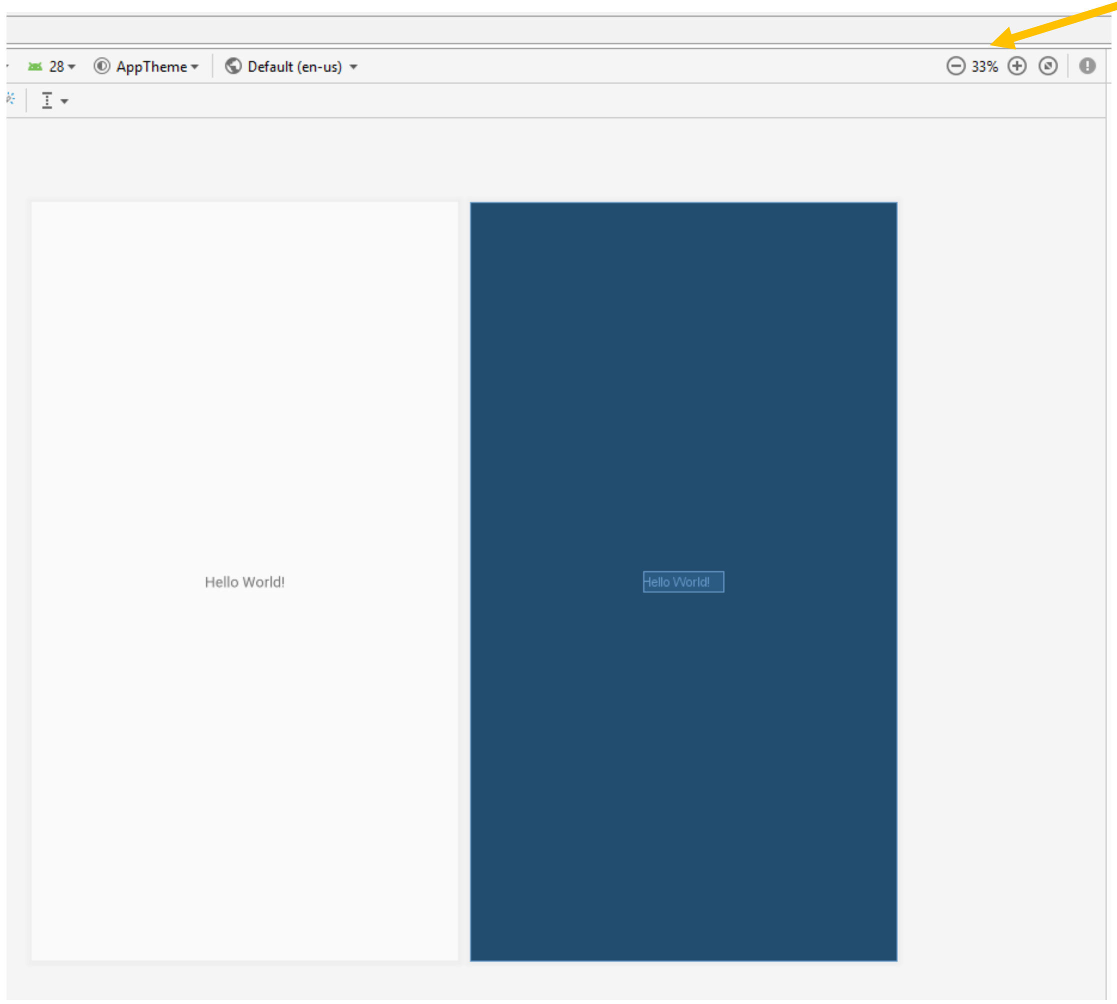
Designing the Interface

- Select the activity_main.xml tab in the editor panel.
  - The layout of the user interface is created with XML.
  - You will not need to write any XML for the apps in this course.



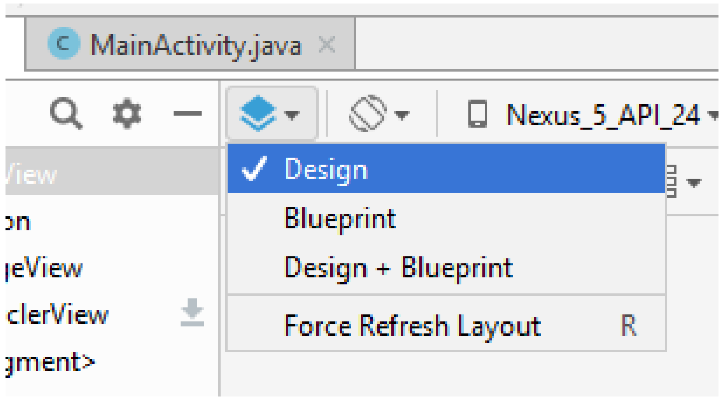
Designing the Interface

Zoom in/out



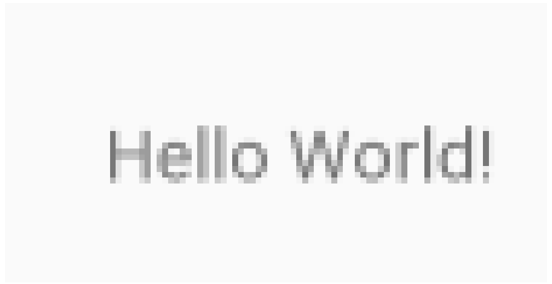
Designing the Interface

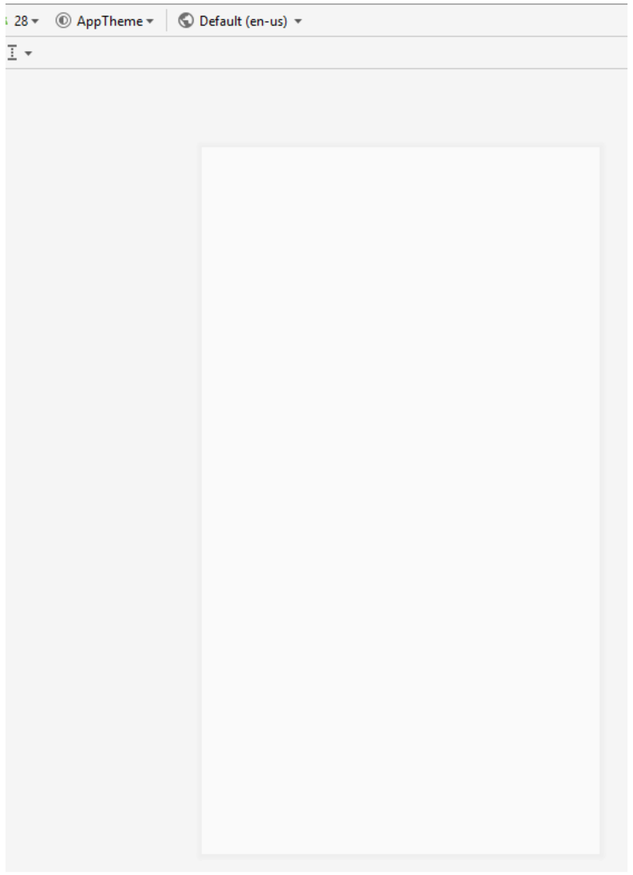
- Switch to Design view only.



Designing the Interface

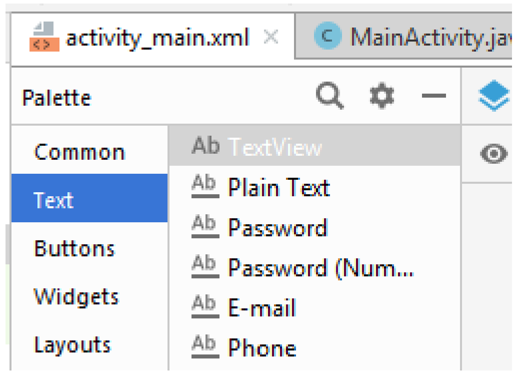
- A “Hello World” label was created for us.
  - Click on it and press the **delete** key.
- The interface should be blank.

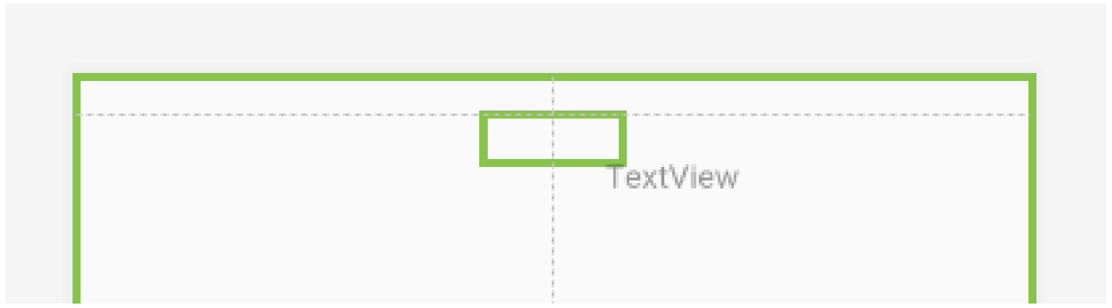




Adding a TextView

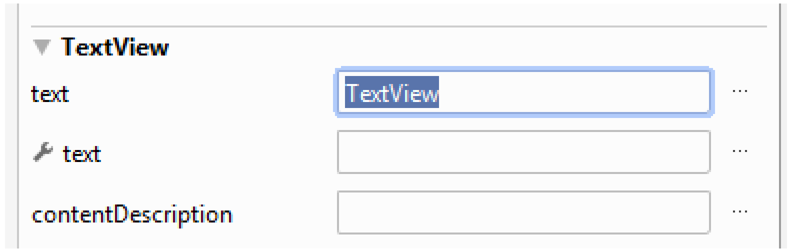
- A **TextView** is a graphical component (“widget”) that displays text.
  - TextViews are like Labels (See the Java GUI slides)
- From the Palette, select **Text**
  - Click and drag a **TextView** from the Palette to the top of the app’s interface.



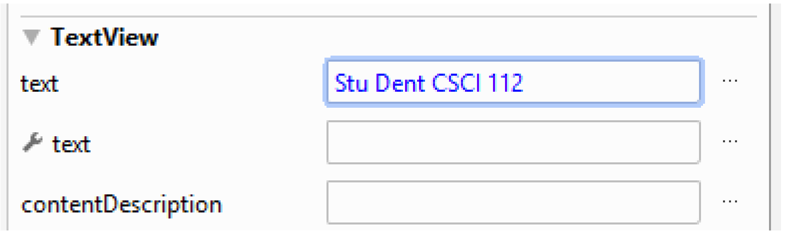


Adding a TextView

- On the right side of Android Studio, you'll see the Attributes pane.
  - This is where we can set and modify properties of the widgets we drop on the app's interface.
- Change the text from "TextView" to "Your Name CSCI 112"



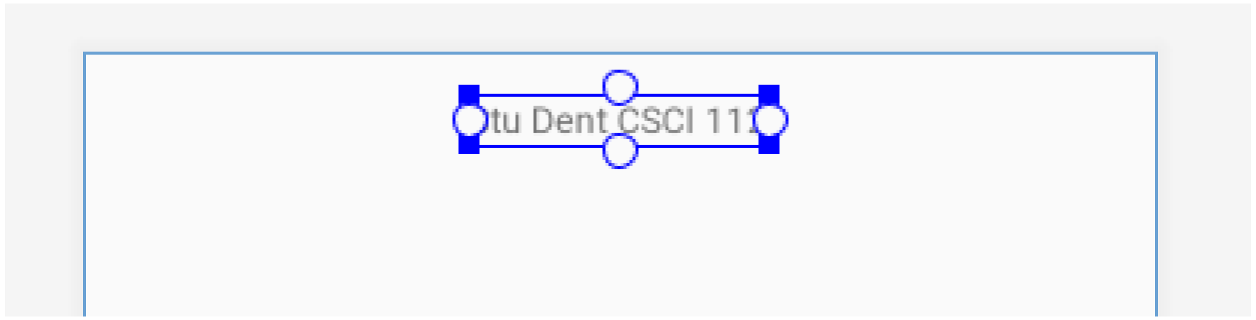
TextView
text: TextView
text:
contentDescription:



TextView
text: Stu Dent CSCI 112
text:
contentDescription:

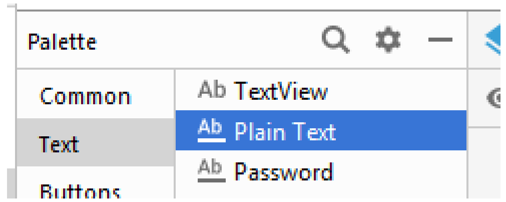
Adding a TextView

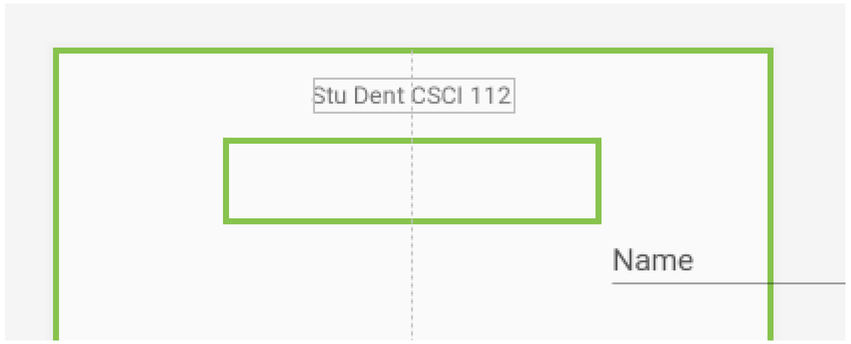
- Adjust the TextView (click and drag) so that is it centered in the interface.



Adding an EditText

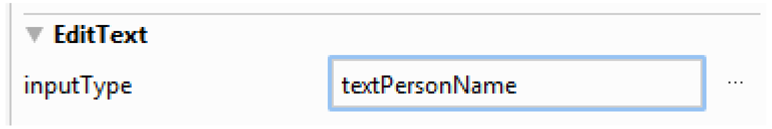
- An EditText is a widget that allows a user to type input.
  - EditTexts are like Text Fields (See the Java GUI slides)
- From the Palette, select **Text**
  - Click and drag a **Plain Text** from the Palette to underneath the TextView that displays your name.

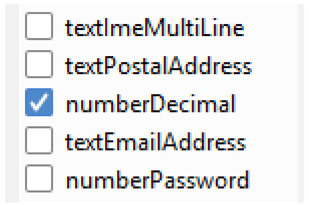


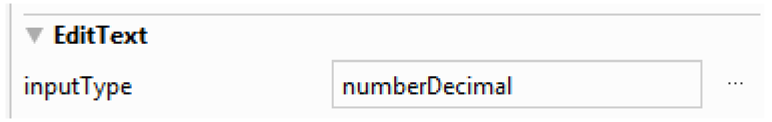


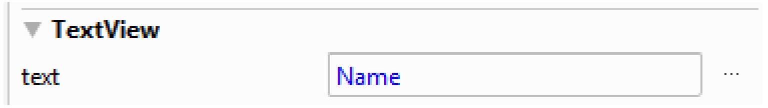
Adding an EditText

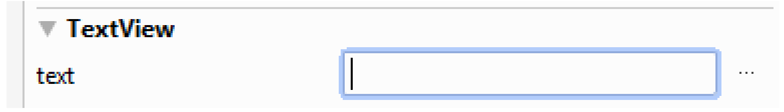
- In the attributes, change the inputType from "textPersonName" to "numberDecimal"
  - numberDecimal should be the only item checked.
- Change the text from "Name" to empty/blank.
  - Press Enter when done.


EditText
inputType textPersonName


- [ ] textImeMultiLine
- [ ] textPostalAddress
- [x] numberDecimal
- [ ] textEmailAddress
- [ ] numberPassword

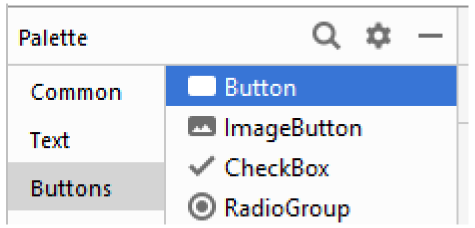

EditText
inputType numberDecimal

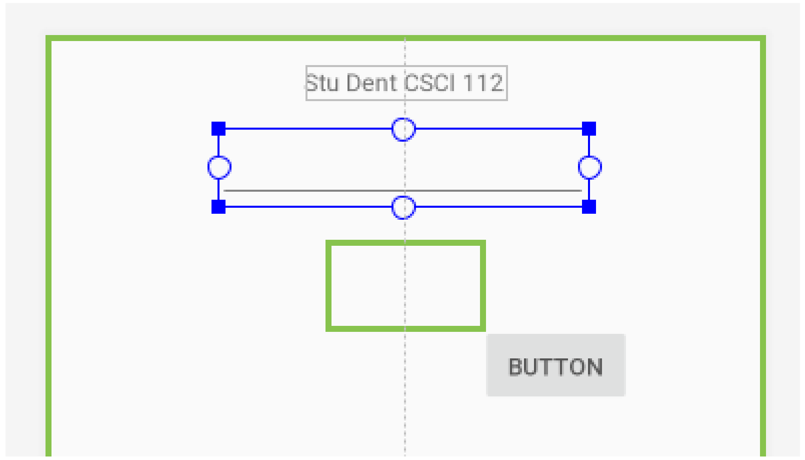

TextView
text Name


TextView
text

Adding a Button

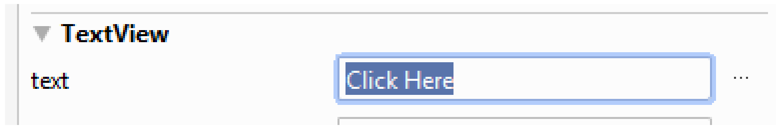
- An Button is a widget that a user presses to trigger some action.
- From the Palette, select **Buttons**
  - Click and drag a **Button** from the Palette to underneath the EditText.





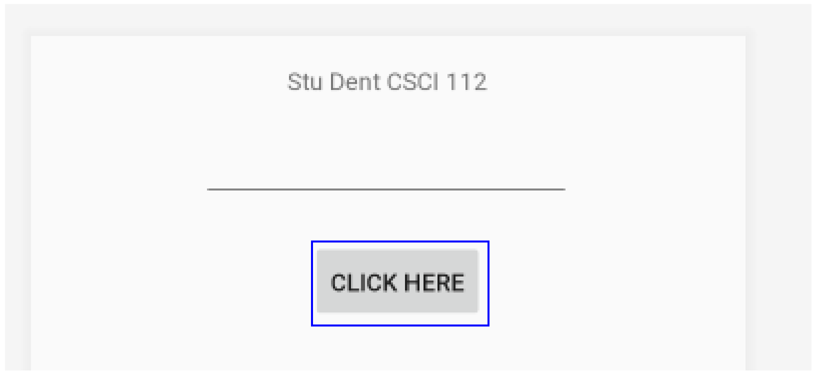
Adding a Button

- In the attributes, change the button's text to read "Click Here"



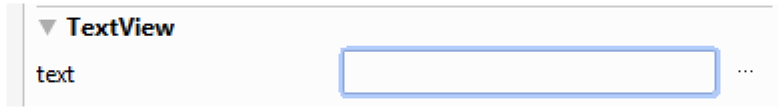
TextView

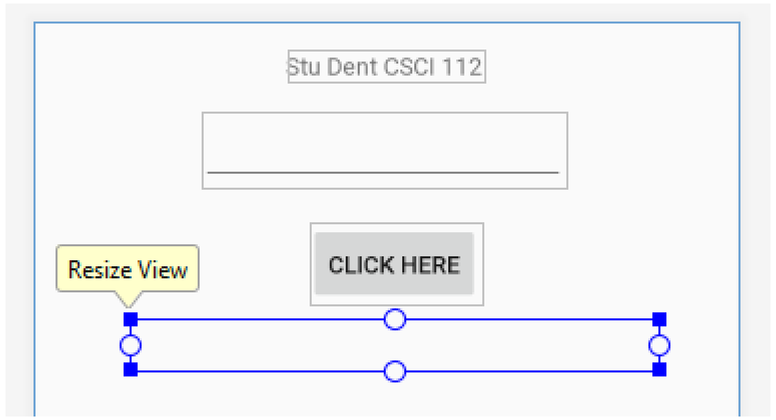
text: Click Here



Add a Second TextView

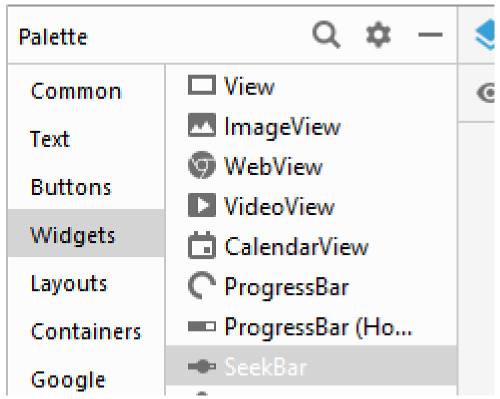
- Add a second TextView under the Button.
  - Click and drag the corners to resize the TextView (about the size shown below)
  - In the attributes, clear the text field so the TextView is blank.
  - Click and drag to recenter the empty TextView under the button.

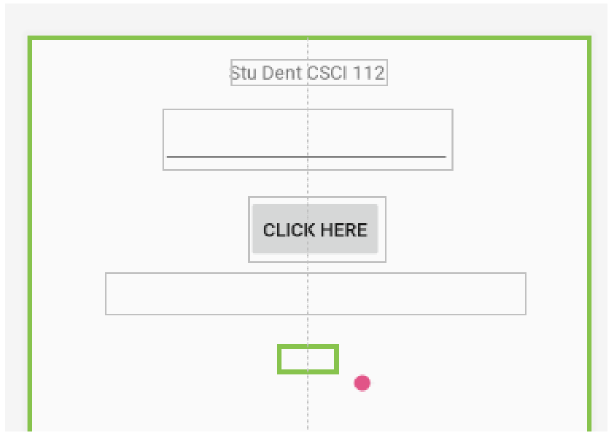




Adding a SeekBar

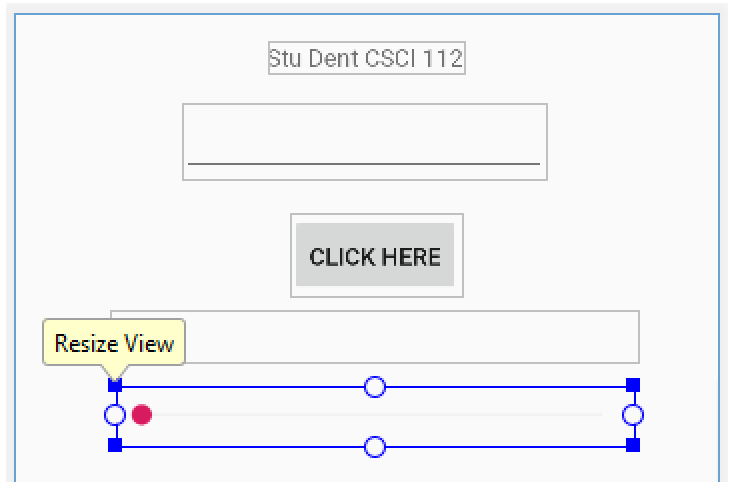
- An SeekBar is a widget that allows a user to select a value on a sliding scale.
  - SeekBars are like Sliders (See the Java GUI slides)
- From the Palette, select **Widgets**
  - Click and drag a **SeekBar** from the Palette to underneath the empty TextView.

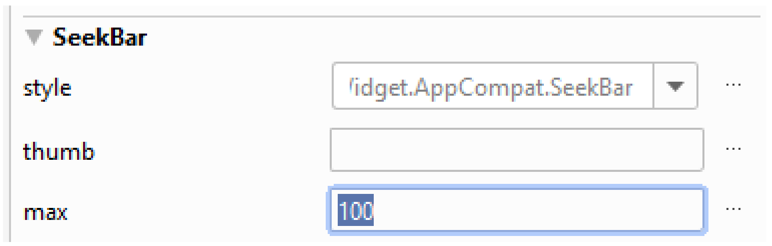




Adding a SeekBar

- Click and drag the corners to resize the SeekBar (about the size shown below)
- In the attributes, sent the max value to 100



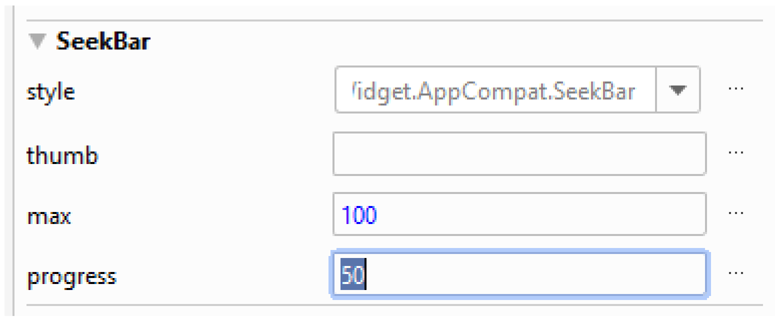


SeekBar

style	'idget.AppCompat.SeekBar
thumb	
max	100

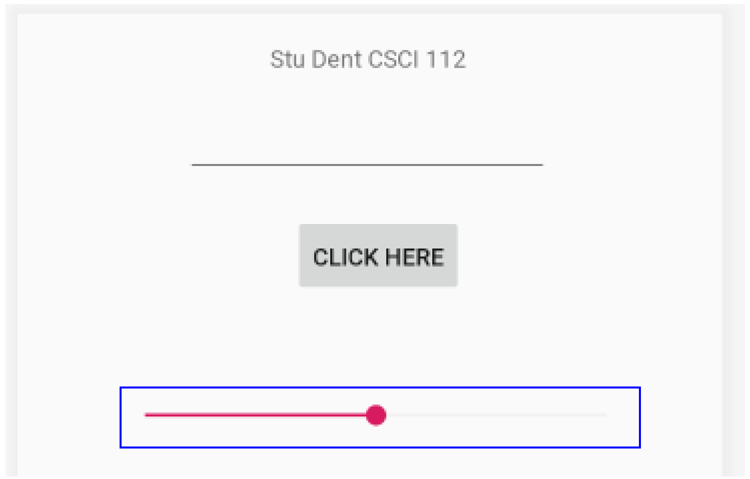
Adding a SeekBar

- In the attributes, sent the progress value to 50
  - Where the SeekBar slider starts



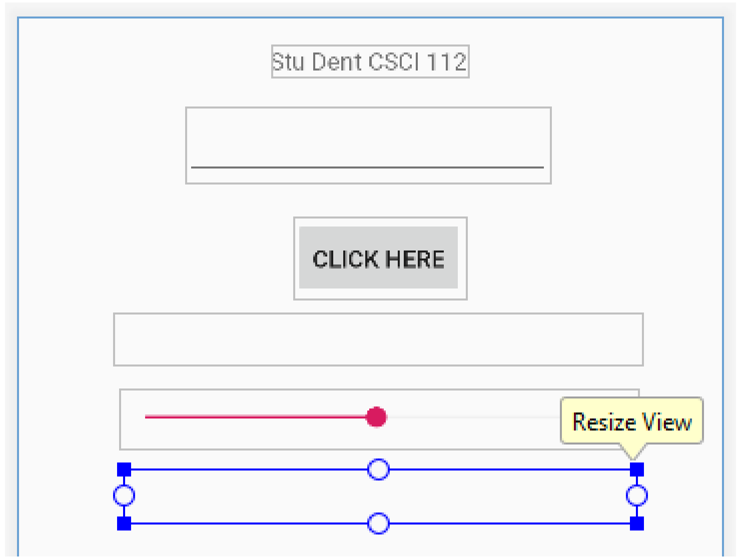
SeekBar

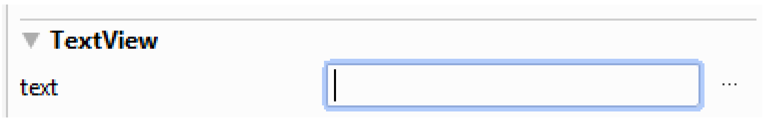
style	'idget.AppCompat.SeekBar
thumb	
max	100
progress	50



Add a Third TextView

- Add a third TextView under the SeekBar.
  - Click and drag the corners to resize the TextView (about the size shown below)
  - In the attributes, clear the text field so the TextView is blank.
  - Click and drag to recenter the empty TextView under the SeekBar.

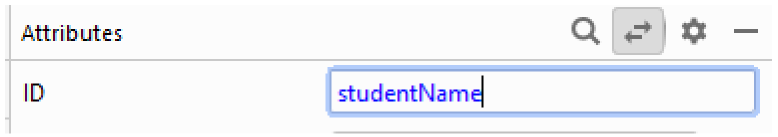




Fixing the IDs

- As we are adding widgets, they are assigned generated IDs

- Select the first TextView with your name.
    - In the attributes, change the ID to **studentName**

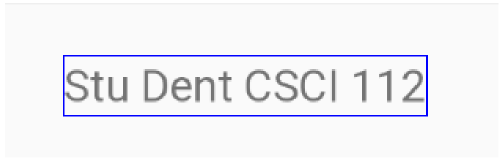

Attributes
ID studentName

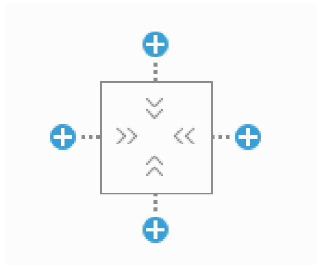
Fixing the IDs

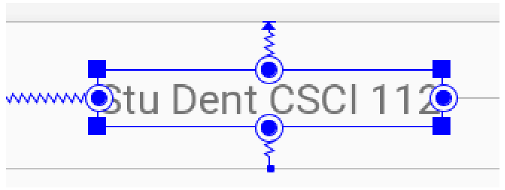
- Rename the IDs of the other widgets:
  - EditText – entryField
  - Button – clickButton
  - TextView (first blank one) – entryText
  - SeekBar – slider
  - TextView (second blank one) – sliderText

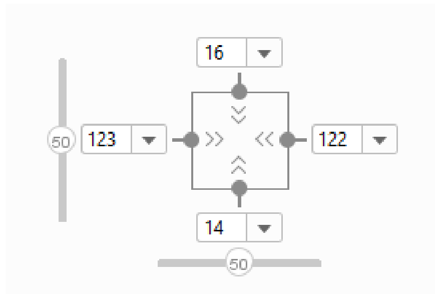
Set Positions

- Select each widget, one at a time.
  - In the attributes, click the + on all four sides.
  - Manually (drag and drop) to re-adjust any widgets that jump out of position.

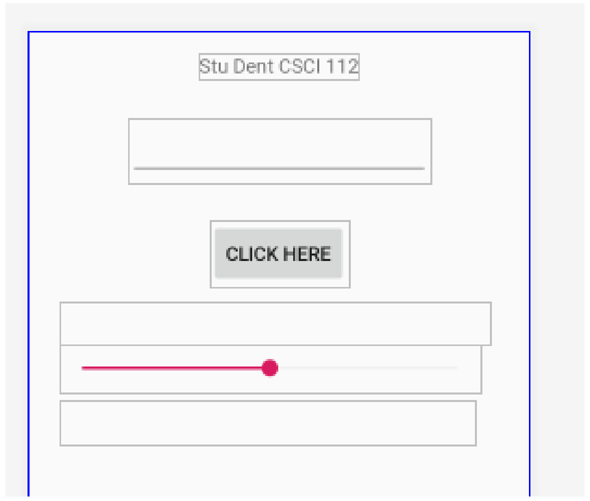








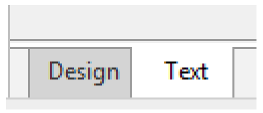
Ok if not exactly the same

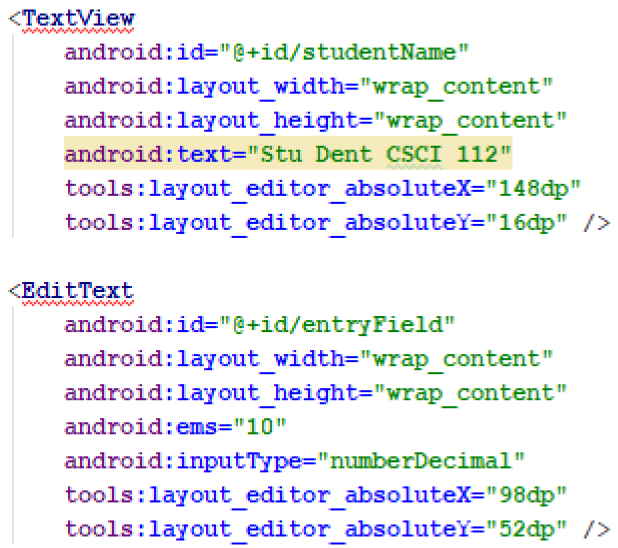


Should look close to this when finished

Viewing the XML behind the interface

- Click the **Text** tab on the bottom left of the interface designer.
  - You don't need to make any changes, but this is the XML "code" behind the interface.



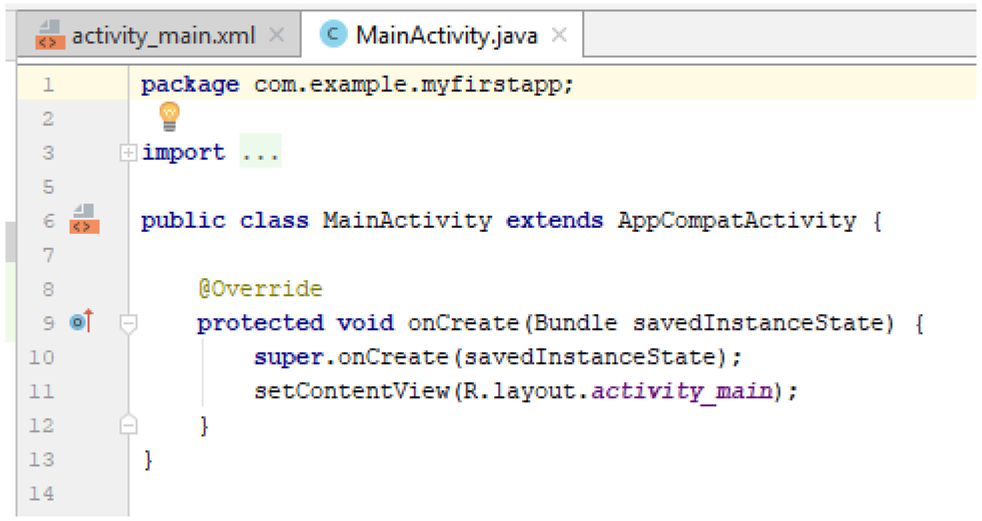


```
<TextView
    android:id="@+id/studentName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Stu Dent CSCI 112"
    tools:layout_editor_absoluteX="148dp"
    tools:layout_editor_absoluteY="16dp" />

<EditText
    android:id="@+id/entryField"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="numberDecimal"
    tools:layout_editor_absoluteX="98dp"
    tools:layout_editor_absoluteY="52dp" />
```

- Click the **Design** tab to go back.

Programming the App

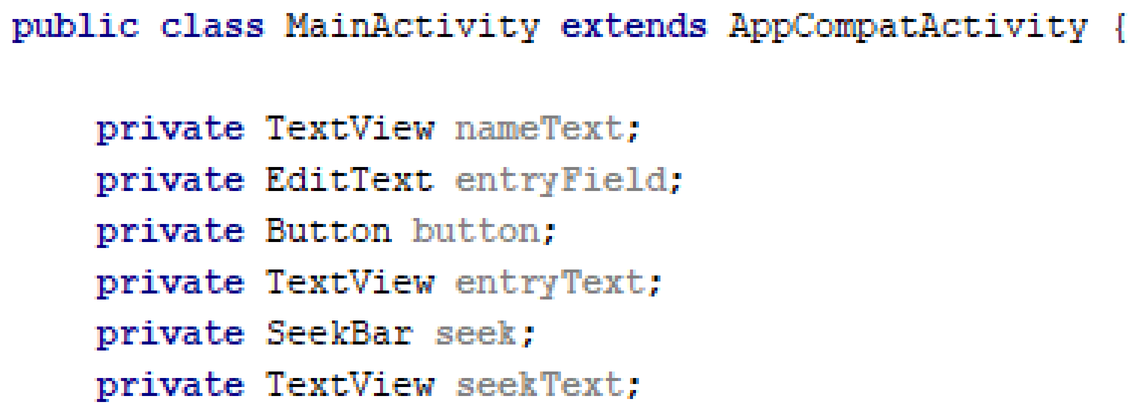
- Select the tab to switch back to MainActivity.java



```
package com.example.myfirstapp;

import ...

public class MainActivity extends AppCompatActivity {

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

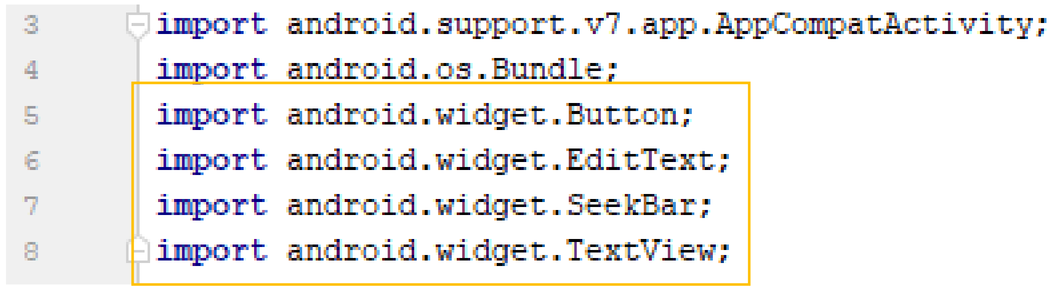
Programming the App

- We need to create variables for the widgets we added.
  - Added like normal fields in a class.



```
public class MainActivity extends AppCompatActivity {

    private TextView nameText;
    private EditText entryField;
    private Button button;
    private TextView entryText;
    private SeekBar seek;
    private TextView seekText;
```

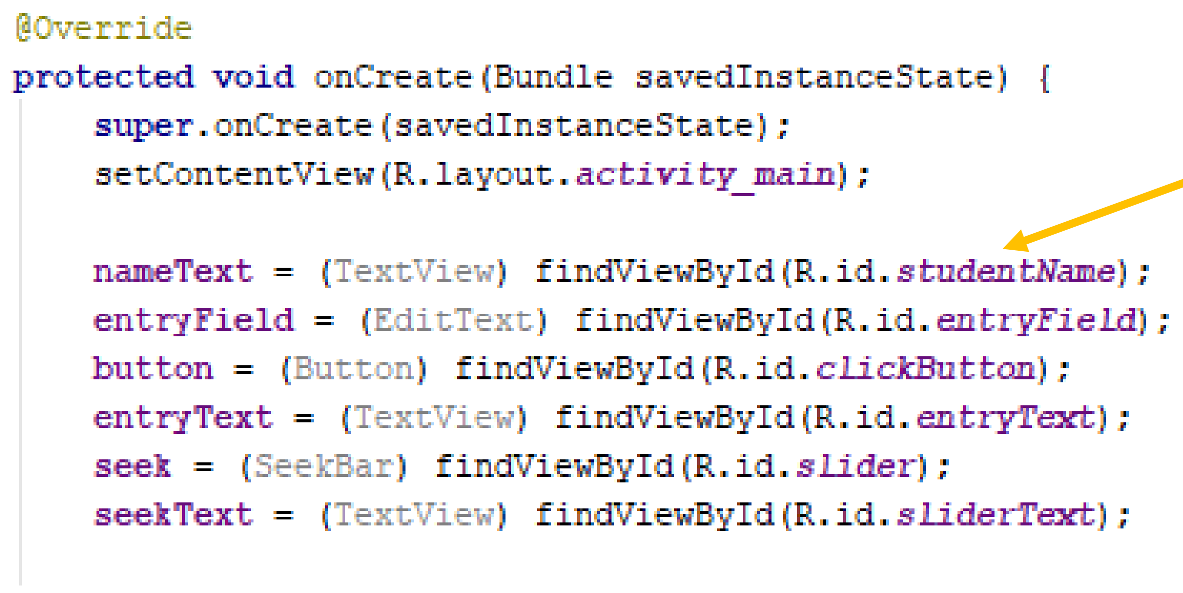
Additional imports



```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.SeekBar;
import android.widget.TextView;
```

Programming the App

- Instantiate the widgets in the onCreate method.



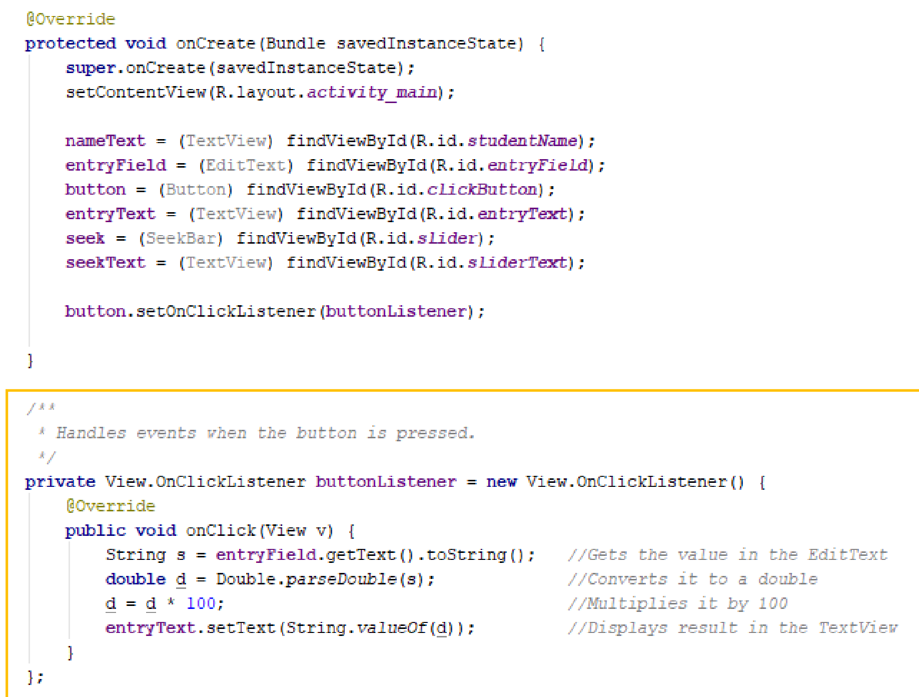
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nameText = (TextView) findViewById(R.id.studentName);
    entryField = (EditText) findViewById(R.id.entryField);
    button = (Button) findViewById(R.id.clickButton);
    entryText = (TextView) findViewById(R.id.entryText);
    seek = (SeekBar) findViewById(R.id.slider);
    seekText = (TextView) findViewById(R.id.sliderText);
```

The IDs we set

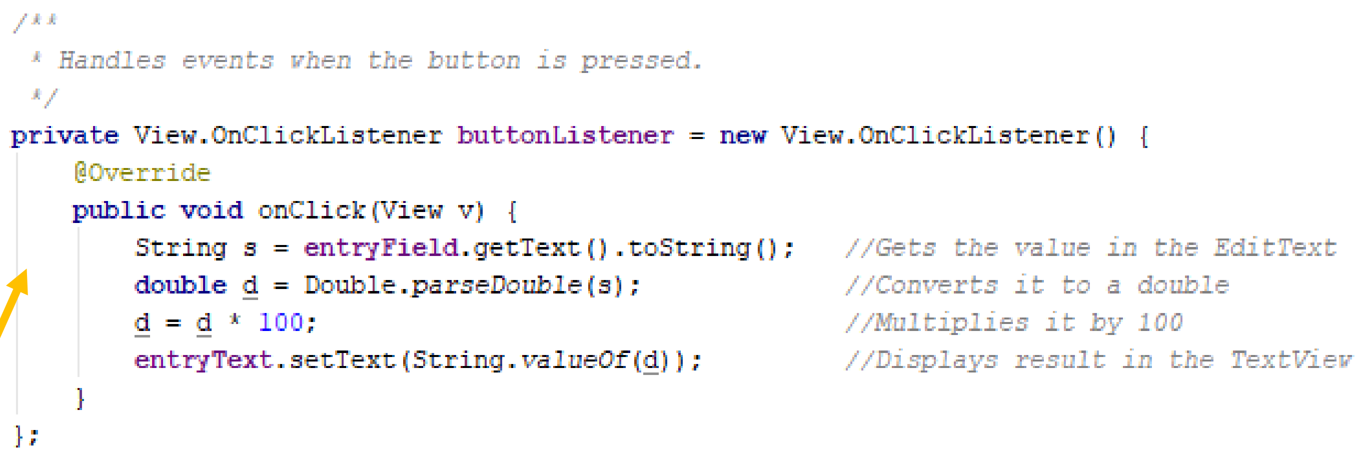
Programming the App

- Creating a Click Listener
  - We will need a special listener object to listen for when the button is clicked.
    - And to do something when it does get clicked.
    - We will have whatever is entered in the EditText be multiplied by 100 and then displayed in the first empty TextView.

Programming the App



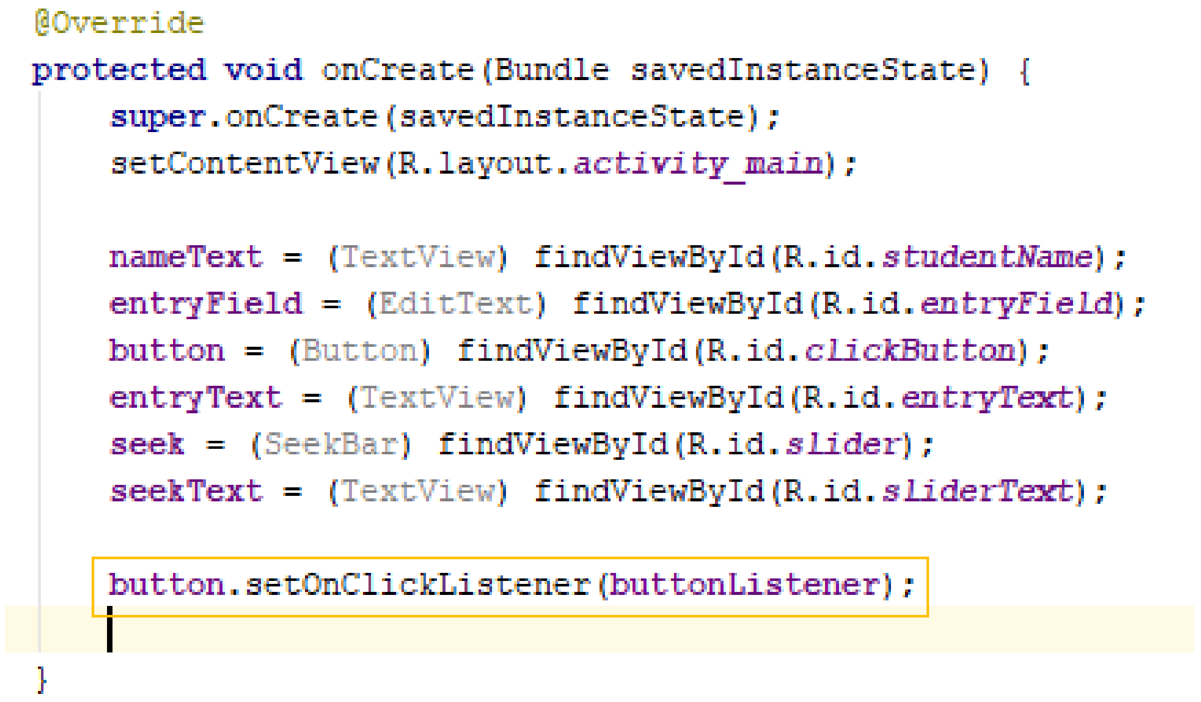
```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nameText = (TextView) findViewById(R.id.studentName);
    entryField = (EditText) findViewById(R.id.entryField);
    button = (Button) findViewById(R.id.clickButton);
    entryText = (TextView) findViewById(R.id.entryText);
    seek = (SeekBar) findViewById(R.id.slider);
    seekText = (TextView) findViewById(R.id.sliderText);

    button.setOnClickListener(buttonListener);

}
/**
 * Handles events when the button is pressed.
 */
private View.OnClickListener buttonListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String s = entryField.getText().toString();    //Gets the value in the EditText
        double d = Double.parseDouble(s);              //Converts it to a double
        d = d * 100;                                   //Multiplies it by 100
        entryText.setText(String.valueOf(d));          //Displays result in the TextView
    }
};
```



```java
/**
 * Handles events when the button is pressed.
 */
private View.OnClickListener buttonListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String s = entryField.getText().toString();    //Gets the value in the EditText
        double d = Double.parseDouble(s);              //Converts it to a double
        d = d * 100;                                   //Multiplies it by 100
        entryText.setText(String.valueOf(d));          //Displays result in the TextView
    }
};
```

Programming the App

- Now, we need to set this listener to the button.

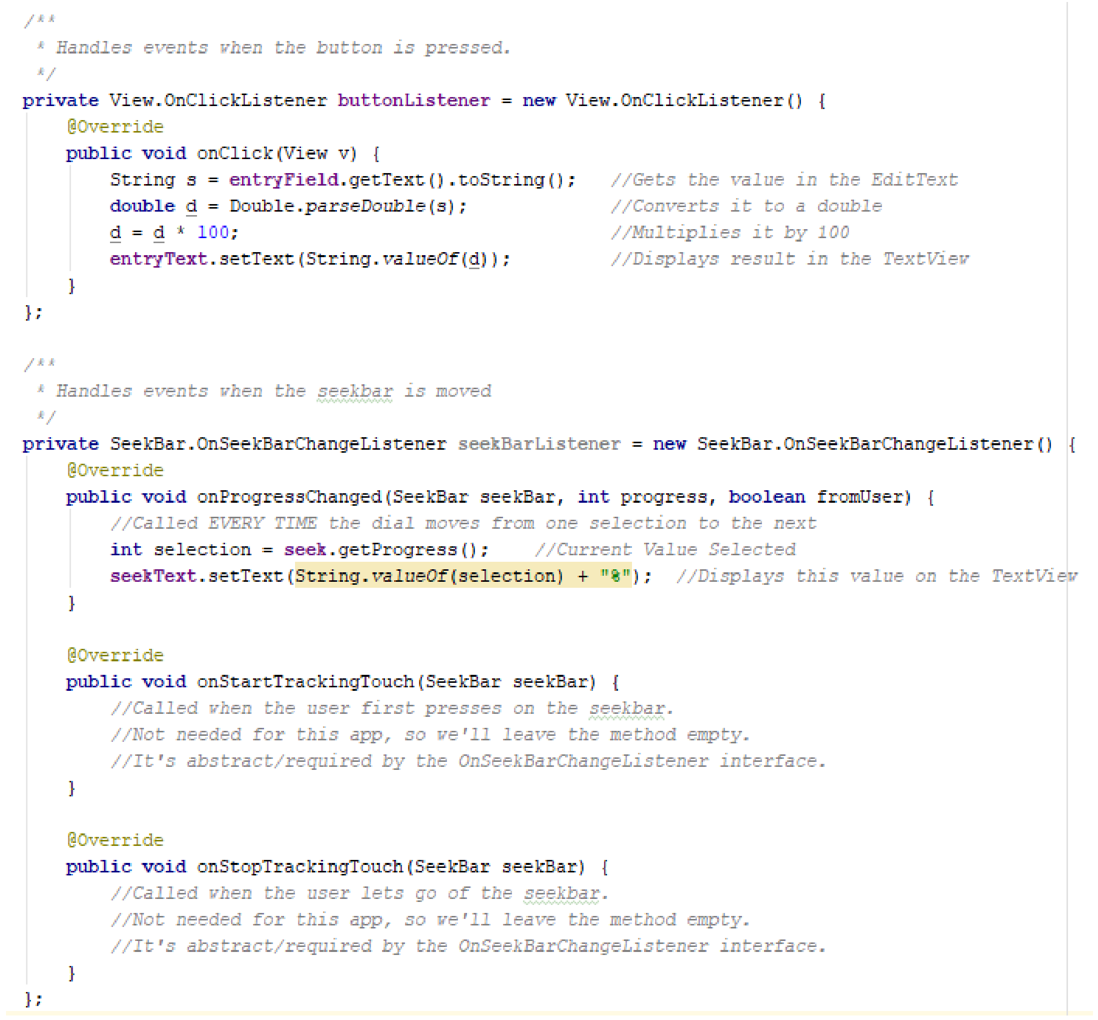


```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nameText = (TextView) findViewById(R.id.studentName);
    entryField = (EditText) findViewById(R.id.entryField);
    button = (Button) findViewById(R.id.clickButton);
    entryText = (TextView) findViewById(R.id.entryText);
    seek = (SeekBar) findViewById(R.id.slider);
    seekText = (TextView) findViewById(R.id.sliderText);

    button.setOnClickListener(buttonListener);

}
```

Programming the App

- Creating a Seek Bar Listener
  - We will need a special listener object to listen for when the SeekBar is changed.
    - And to do something when it does get changed.
    - We will have whatever is selected on the SeekBar be displayed in the second empty TextView.

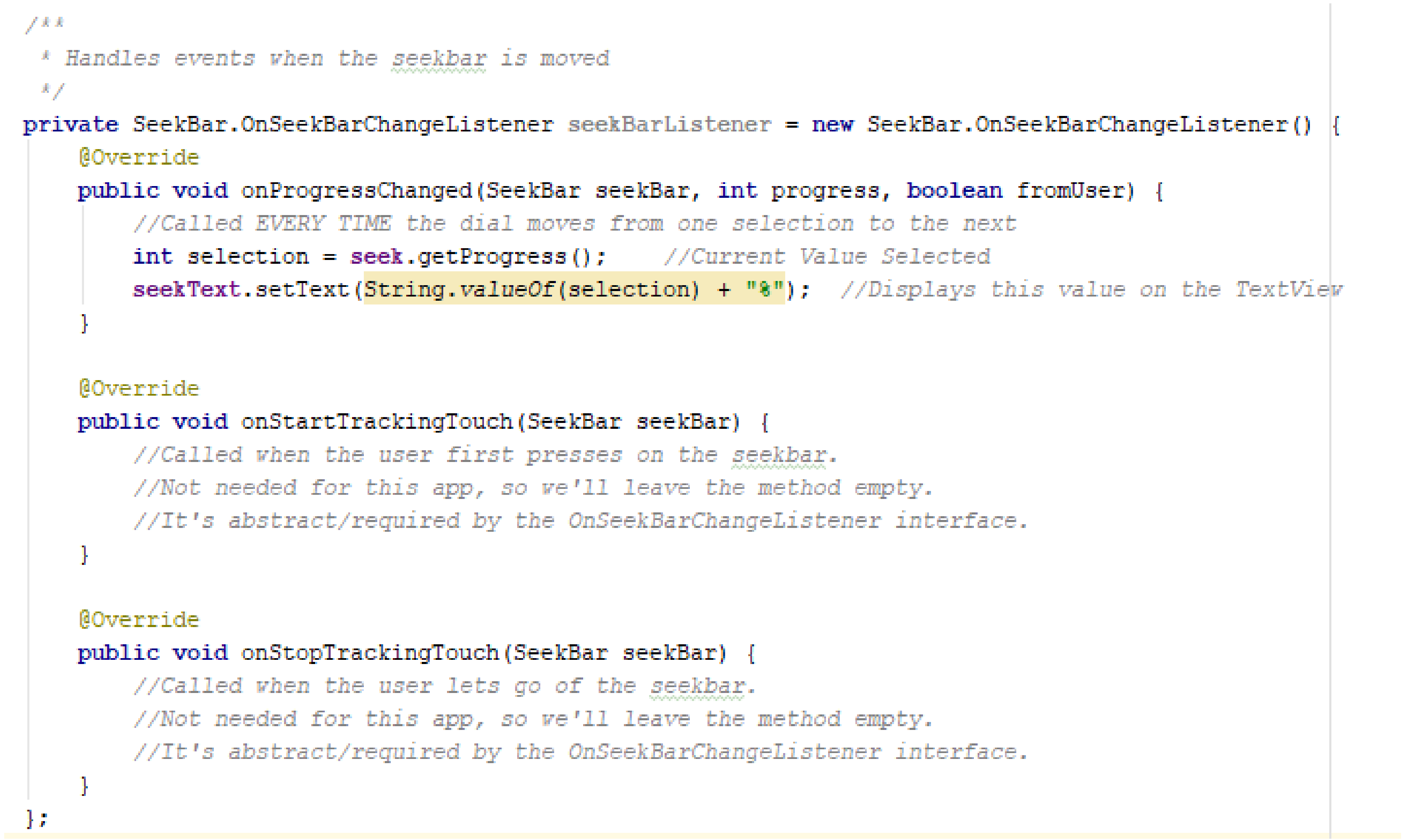
Programming the App



```java
/**
 * Handles events when the button is pressed.
 */
private View.OnClickListener buttonListener = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String s = entryField.getText().toString();   //Gets the value in the EditText
        double d = Double.parseDouble(s);              //Converts it to a double
        d = d * 100;                                   //Multiplies it by 100
        entryText.setText(String.valueOf(d));          //Displays result in the TextView
    }
};

/**
 * Handles events when the seekbar is moved
 */
private SeekBar.OnSeekBarChangeListener seekBarListener = new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {
        //Called EVERY TIME the dial moves from one selection to the next
        int selection = seek.getProgress();    //Current Value Selected
        seekText.setText(String.valueOf(selection) + "%");  //Displays this value on the TextView
    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
        //Called when the user first presses on the seekbar.
        //Not needed for this app, so we'll leave the method empty.
        //It's abstract/required by the OnSeekBarChangeListener interface.
    }

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {
        //Called when the user lets go of the seekbar.
        //Not needed for this app, so we'll leave the method empty.
        //It's abstract/required by the OnSeekBarChangeListener interface.
    }
};
```

See next slide for close-up

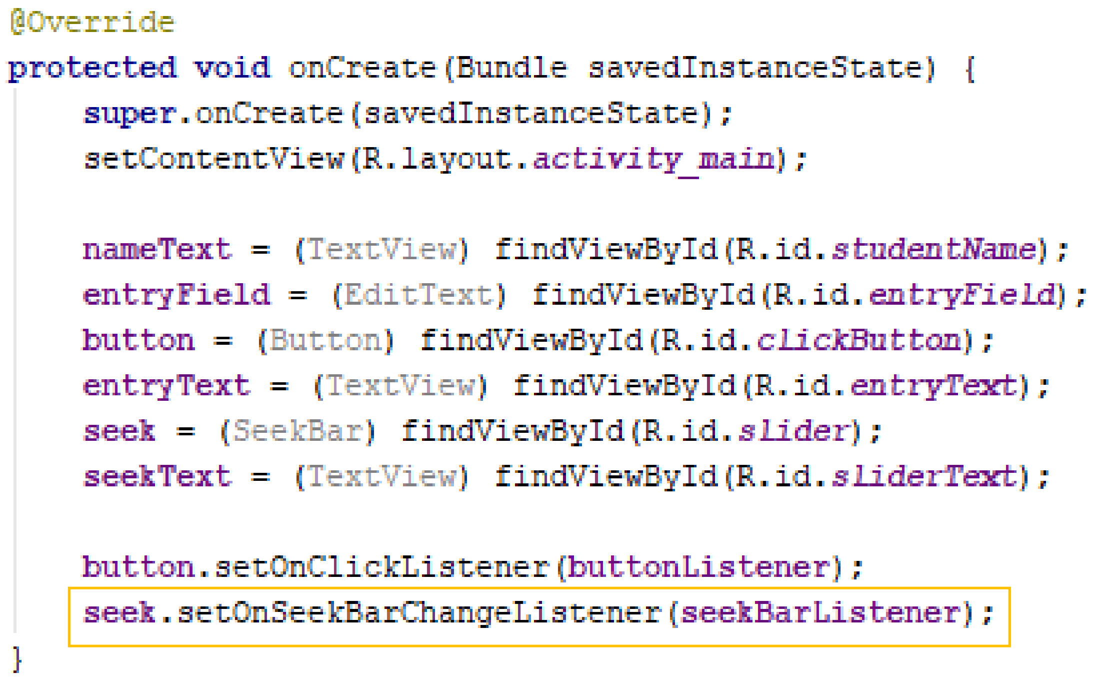
Programming the App



```
/**
 * Handles events when the seekbar is moved
 */
private SeekBar.OnSeekBarChangeListener seekBarListener = new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {
        //Called EVERY TIME the dial moves from one selection to the next
        int selection = seek.getProgress();    //Current Value Selected
        seekText.setText(String.valueOf(selection) + "%");  //Displays this value on the TextView
    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
        //Called when the user first presses on the seekbar.
        //Not needed for this app, so we'll leave the method empty.
        //It's abstract/required by the OnSeekBarChangeListener interface.
    }

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {
        //Called when the user lets go of the seekbar.
        //Not needed for this app, so we'll leave the method empty.
        //It's abstract/required by the OnSeekBarChangeListener interface.
    }
};
```

Programming the App

- Now, we need to set this listener to the SeekBar.

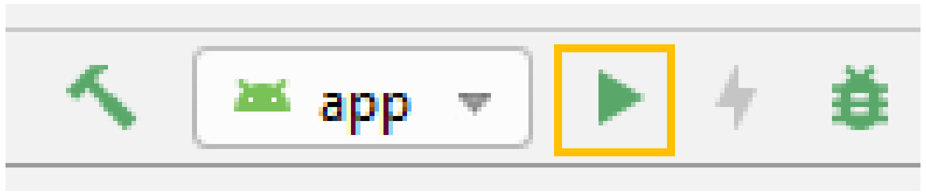


```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    nameText = (TextView) findViewById(R.id.studentName);
    entryField = (EditText) findViewById(R.id.entryField);
    button = (Button) findViewById(R.id.clickButton);
    entryText = (TextView) findViewById(R.id.entryText);
    seek = (SeekBar) findViewById(R.id.slider);
    seekText = (TextView) findViewById(R.id.sliderText);

    button.setOnClickListener(buttonListener);
    seek.setOnSeekBarChangeListener(seekBarListener);
}
```

Ready to Test

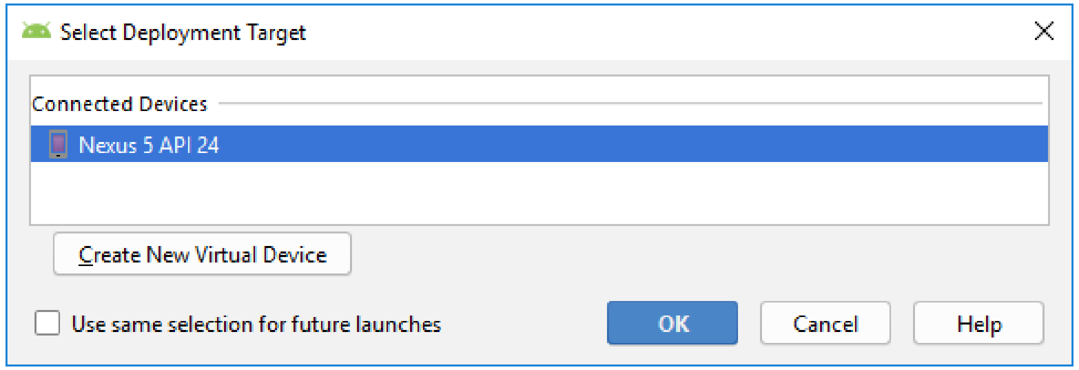
- With the interface set up and the two interactive widgets set up, we can test the app.

- Click the Play button in the upper right corner to run/test the app.
  - Be sure your AVD is ready to go.

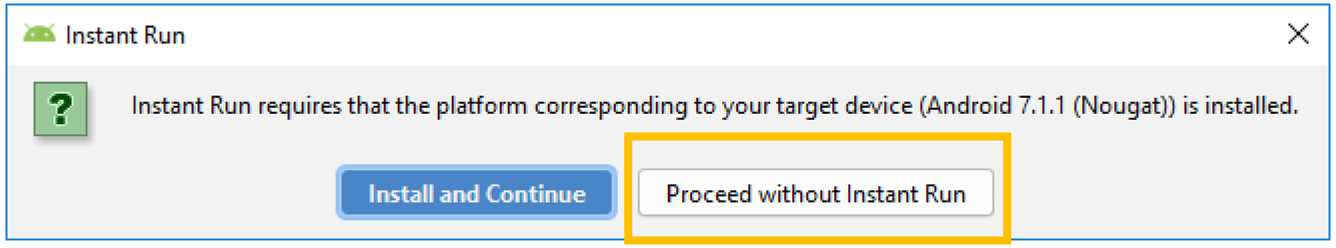


Ready to Test

- Make sure your running/connected AVD is selected and click OK.



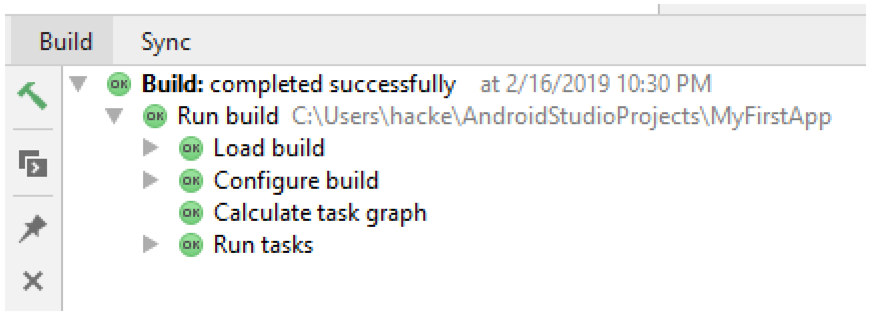
- Click **Proceed without Instant Run** if you see a similar message to this:



Instant Run requires that the platform corresponding to your target device (Android 7.1.1 (Nougat)) is installed.

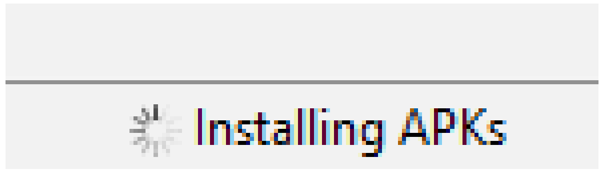
Ready to Test

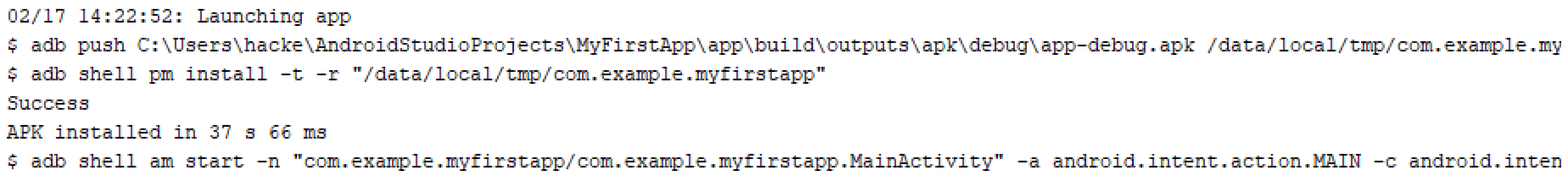
- Give it a minute or two to finish building and installing the app.



Build Sync

```
Build: completed successfully at 2/16/2019 10:30 PM
  Run build C:\Users\hacke\AndroidStudioProjects\MyFirstApp
    Load build
    Configure build
    Calculate task graph
    Run tasks
```

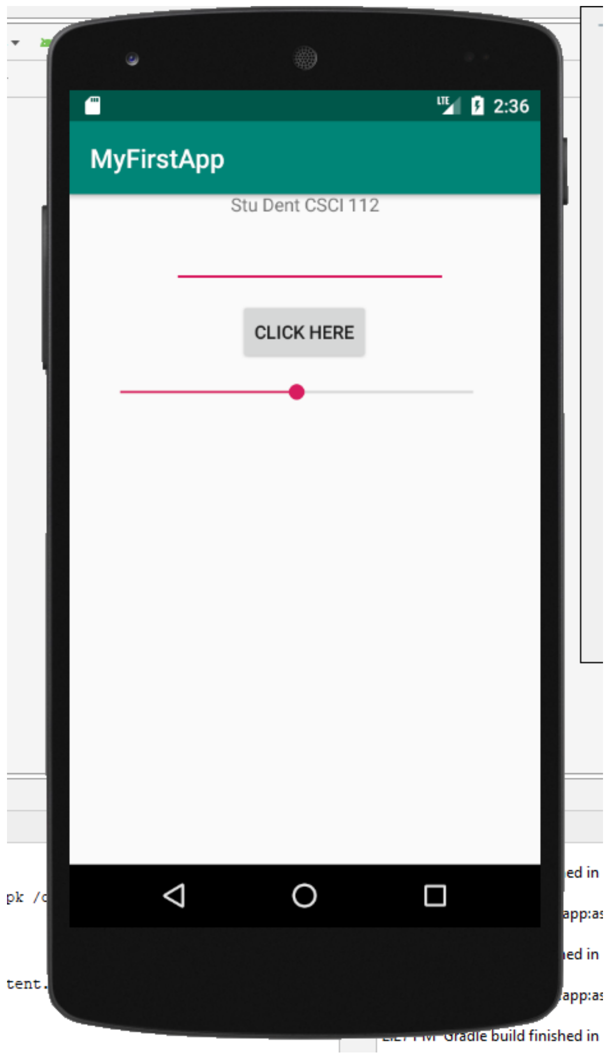




```
02/17 14:22:52: Launching app
$ adb push C:\Users\hacke\AndroidStudioProjects\MyFirstApp\app\build\outputs\apk\debug\app-debug.apk /data/local/tmp/com.example.my
$ adb shell pm install -t -r "/data/local/tmp/com.example.myfirstapp"
Success
APK installed in 37 s 66 ms
$ adb shell am start -n "com.example.myfirstapp/com.example.myfirstapp.MainActivity" -a android.intent.action.MAIN -c android.inten
```

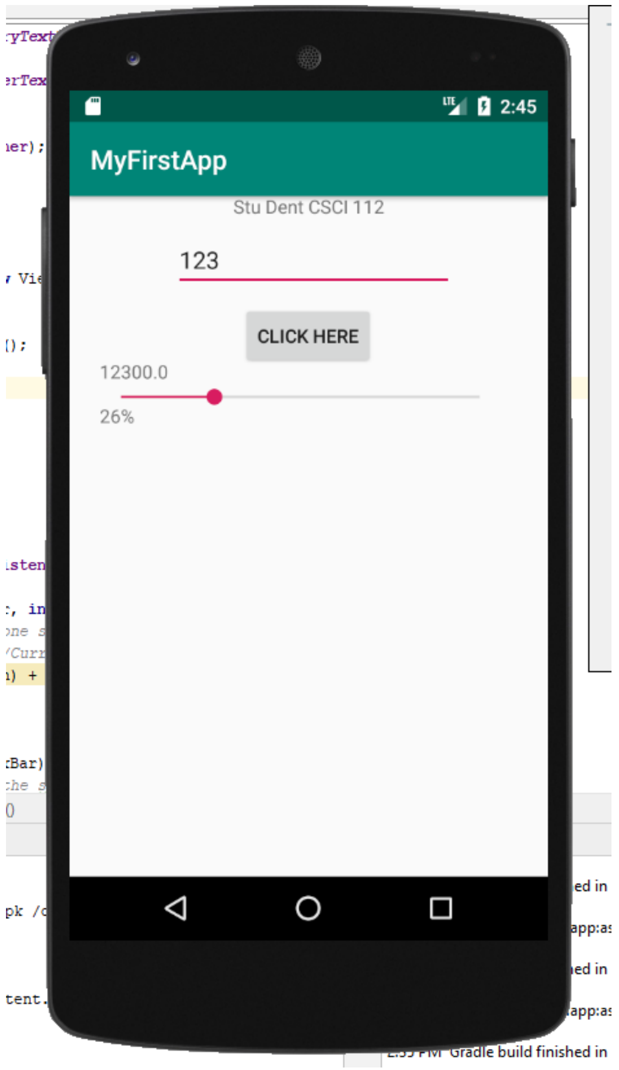
Ready to Test

- The app should be started automatically.



Ready to Test

- Test the features.
  - Enter numbers into the EditText.
    - Click the EditText.
      - The keypad should appear.
      - Enter the value 123
      - Click the Check Box icon in the keypad to close it.
    - Click the Button.
      - The number (multiplied by 100) should be displayed
  - Move the SeekBar's dial to 26%.



Exception Handling

- An Exception is an object that is generated as the result of an error or an unexpected event.
  - When that happens, we say that an exception has been ***thrown***.
- It is the programmer's responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.

Exception Handling

- An ***exception handler*** is a section of code that gracefully responds to exceptions.
    - The process of anticipating and responding to exceptions is called ***exception handling***.

- The ***default exception handler*** deals with any unhandled exceptions.
    - It prints an error message and stops the program.

Unchecked Exceptions

- Java has two types of exceptions, checked and unchecked.
- An ***unchecked exception*** is an exception that the compiler does not check for.
  - Executing the code below will result in an ArrayIndexOutOfBoundsException.

```
char[] letters = {'a', 'b', 'c'};
System.out.println(letters[3]);
```

  - The compiler doesn’t check to see if the index used is valid, yet the code will still compile.
  - The result will be an unchecked exception being thrown when the program runs.

Checked Exceptions

- A ***checked exception*** is an exception that the compiler *does* check for.
  - Checked exceptions have to be handled or the source code won't compile.

- Checked exceptions are identified by a method or constructor explicitly stating that they may throw an exception.
  - The method/constructor has a "throws" clause.

try…catch statements

- To handle possible exceptions, use a try…catch statement.

```
try {
  try block statements
}
catch(ExceptionType name) {
  catch block statements
}
```

- First, the keyword `try` indicates a block of code that will be attempted (the curly braces are required).
  - This block of code is known as a ***try block***.

try…catch statements

- The try block contains one or more statements that, when executed, can potentially throw an exception.
  - After the try block, at least one catch clause is required.

- The application will not halt if the try block throws an exception.
  - Instead, the code in the catch block will be executed.

catch clauses

- A catch clause begins with the keyword `catch`:

```
catch(ExceptionType name) {
  catch block statements
}
```

  - *ExceptionType* is the name/type of an exception object.
  - *name* is a variable name which will reference the exception object.
- The code that inside the catch clause is known as a ***catch block***.
  - The code in the catch block is executed if the try block throws that particular type of exception.

Handling Exceptions

- This code is designed to handle a ArrayIndexOutOfBoundsException, if it is thrown by statements in the try block.

```
try {
  values[99] = 34;
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Index does not exist in the array.");
}
```

- The Java Virtual Machine searches for a catch clause that can deal with the exception.

Default Error Messages

- Each exception object has a method named **getMessage** that can be used to retrieve the default error message for the exception.

```
try {
  value = new Integer.parseInt("abcd");
}
catch(NumberFormatException e) {
  System.out.println("An exception occurred:" + e.getMessage());
}
```

```
An exception occurred: For input string: "abcd"
```

Tracing the Call Stack

- The ***call stack*** is a list of the methods that are currently executing.
- A ***stack trace*** is a list of all the methods in the call stack.
- It can indicate:
  - The method that was executing when an exception occurred and
  - All of the methods that were called in order to execute that method.

Tracing the Call Stack

- Each exception object has a method named **printStackTrace** that will print the exception's stack trace to the console.

```
try {
    value = new Integer.parseInt("abcd");
}
catch(NumberFormatException e) {
    e.printStackTrace();
}
```

```
java.lang.NumberFormatException: For input string: "abcd"
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.lang.Double.parseDouble(Double.java:538)
    at SampleCode.TestProgram.main(b_ExceptionMessage.java:18)
```

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - Normal execution of that method stops.
  - The JVM searches for a compatible exception handler inside the method.

Uncaught Exceptions

- If there is no exception handler inside the method:
  - Control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the main method:
  - The main method must either handle the exception, or
  - The program is halted and the default exception handler handles the exception.

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- Multiple catch clauses can be written for each type of exception that could potentially be thrown.

Handling Multiple Exceptions

```
try {
  value = new Integer.parseInt(someString);
  numbers[7] = 15;
}
catch(NumberFormatException e) {
  System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid array index.");
}
```

Handling Multiple Exceptions

```
try {
  value = new Integer.parseInt(someString);
  numbers[7] = 15;
}
catch(NumberFormatException e) {
  System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid array index.");
}
```

The Exception Object

- The Exception object is the parent object of many types of exceptions.
  - In one way or another, all exceptions are traced back to the Exception object.

- Catching an Exception object allows us to catch every type of exception at once.
  - Which is sometimes good and sometimes bad.

The Exception Object

```
try {
  ...
}
catch(Exception e) {
  System.out.println("An exception occurred.");
}
```

- Regardless of the code in the try block, **any** exception thrown will be handled by the catch clause.
- Sure, the program won’t halt but we have no idea:
  - What caused the exception.
  - How the program should respond.

The Exception Object

- A better use of the Exception object is to make it the last catch clause.

- This will allow other catch clauses to handle specific types of exceptions.
  - The Exception catch will handle any other, possibly unforeseen, exceptions that are thrown.

The Exception Object

```
try {
  value = new Integer.parseInt(someString);
  numbers[7] = 15;
}
catch(NumberFormatException e) {
  System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid array index.");
}
catch(Exception e) {
  System.out.println("Unexpected exception.");
  e.printStackTrace();
}
```

finally clauses

- A try…catch statement may have an optional **`finally`** clause.
- If present, the finally clause must appear <u>after</u> all of the catch clauses.
- The ***finally block*** is one or more statements that are always executed after the try block has executed AND after any catch blocks have executed if an exception was thrown.
  - In other words, <u>the statements in the finally block execute whether an exception occurs or not.</u>

finally clauses

```java
try {
  values[99] = 34;
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Index does not exist in the array.");
}
finally {
  //Executes the code in here regardless if an exception
  //was thrown in the try block or not.
}
```

Throwing Exceptions

- If a method is unable to complete its function/purpose, the method should throw an exception.
- To throw an exception use the following statement:

```
throw new Exception("Your exception's message");
```

- The above statement throws a generic Exception object.
  - You can throw any type of exception you want (some may need to be imported):

```
throw new NumberFormatException("Your exception's message");
throw new IOException("Your exception's message");
```

- This creates a checked exception situation.
  - The exception you throw must be handled at some point up the call stack.

Throwing Exceptions

- If a method might throw an exception, the method header needs a ***throws clause***.
  - The class won't compile without it.
  - The exception type in the throws clause must match the type thrown.

```
private int validateGear(int g) throws Exception {
    if(g >= 1 && g <= 10) {
        return g; //Valid Value
    }
    throw new Exception("Gear value is invalid: " + g);
}
```

Throwing Exceptions

- Now, any method (or constructor) that calls the validateGear method must:
  - **Handle the exception itself**, or
  - Throw it up the call stack.

```
public void setCurrentGear(int currentGearIn) {
    try {
        currentGear = validateGear(currentGearIn);
    }
    catch(Exception e){
        //Maybe print an error message or set currentGear to 1
    }
}
```

Throwing Exceptions

- Now, any method (or constructor) that calls the validateGear method must:
  - Handle the exception itself, or
  - **Throw it up the call stack.**

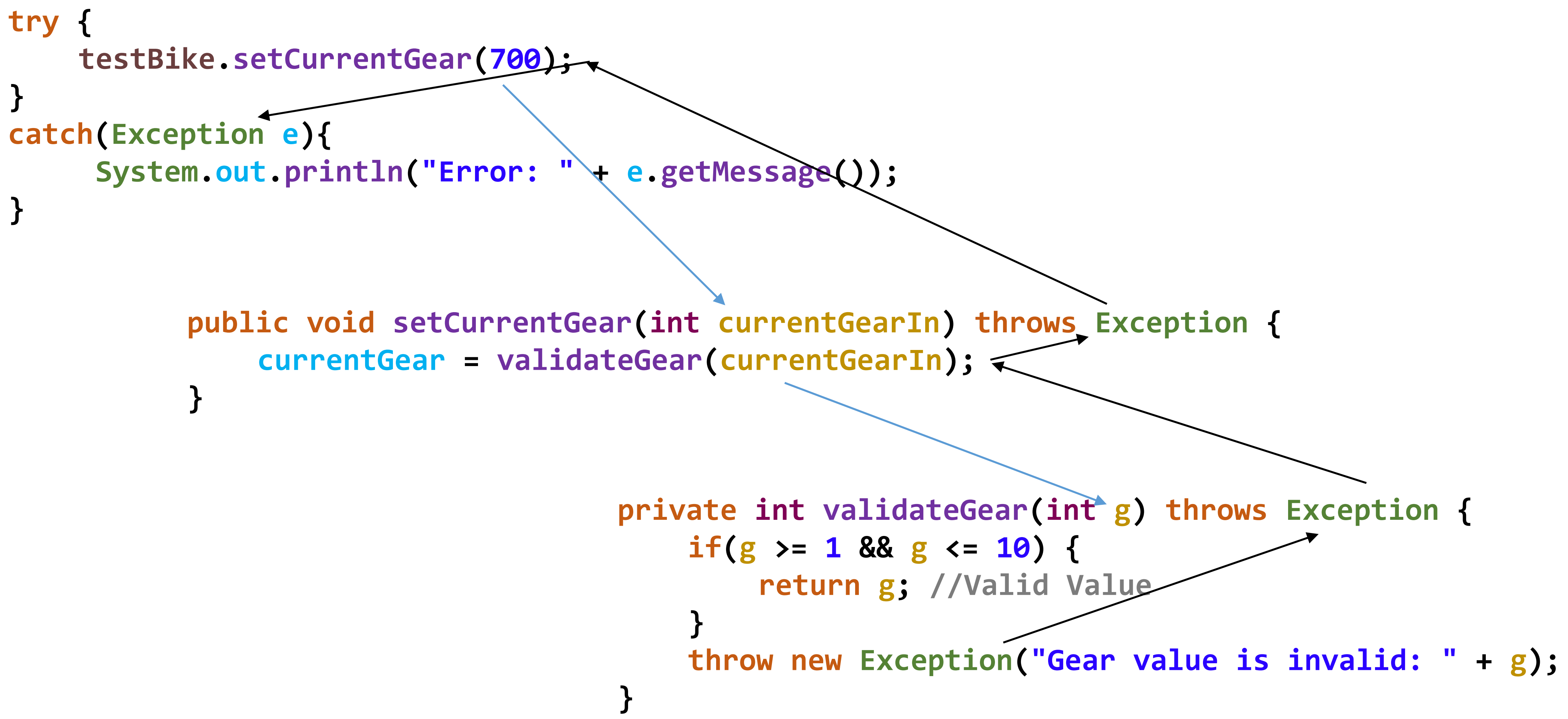
```
public void setCurrentGear(int currentGearIn) throws Exception {
    currentGear = validateGear(currentGearIn);
}
```

Throwing Exceptions

```java
public class BicycleTest {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();
        try {
            testBike.setCurrentGear(700);
        }
        catch(Exception e){
             System.out.println("Error: " + e.getMessage());
        }
        System.out.println("testBike is in gear " + testBike.getCurrentGear());
    }

}
```

```
Error: Gear value is invalid: 700
testBike is in gear 0
```

Throwing Exceptions (Call Stack)



```
try {
    testBike.setCurrentGear(700);
}
catch(Exception e){
     System.out.println("Error: " + e.getMessage());
}
```

```
public void setCurrentGear(int currentGearIn) throws Exception {
    currentGear = validateGear(currentGearIn);
}
```

```
private int validateGear(int g) throws Exception {
    if(g >= 1 && g <= 10) {
        return g; //Valid Value
    }
    throw new Exception("Gear value is invalid: " + g);
}
```