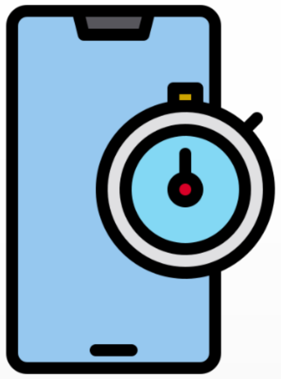
# Starting Another Activity in Java Android App. Java Programming Language Fundamentals (part 2)

Dmytro Zubov, PhD
dmytro.zubov@ucentralasia.org

## Naryn, 9:06am, October 8, 2022

## Lessons learnt last time

- What is Firebase?
- History of Firebase
- Why use Firebase?
- General architecture
- Services
- Adding Firebase Realtime Database to Android app

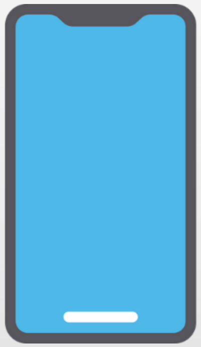


## What we gonna discuss today?

- Start another activity in the Java Android app
- Cast operator
- Conditional operators
- Equality operators
- Logical operators
- Multiplicative operators
- Relational operators
- Shift operators
- If-Else statement
- Switch statement
- Loop statements
- Try-catch exception handler
- Break statement and Labeled Break statement
- Continue statement and Labeled Continue statement

## Start another activity in the Java Android app

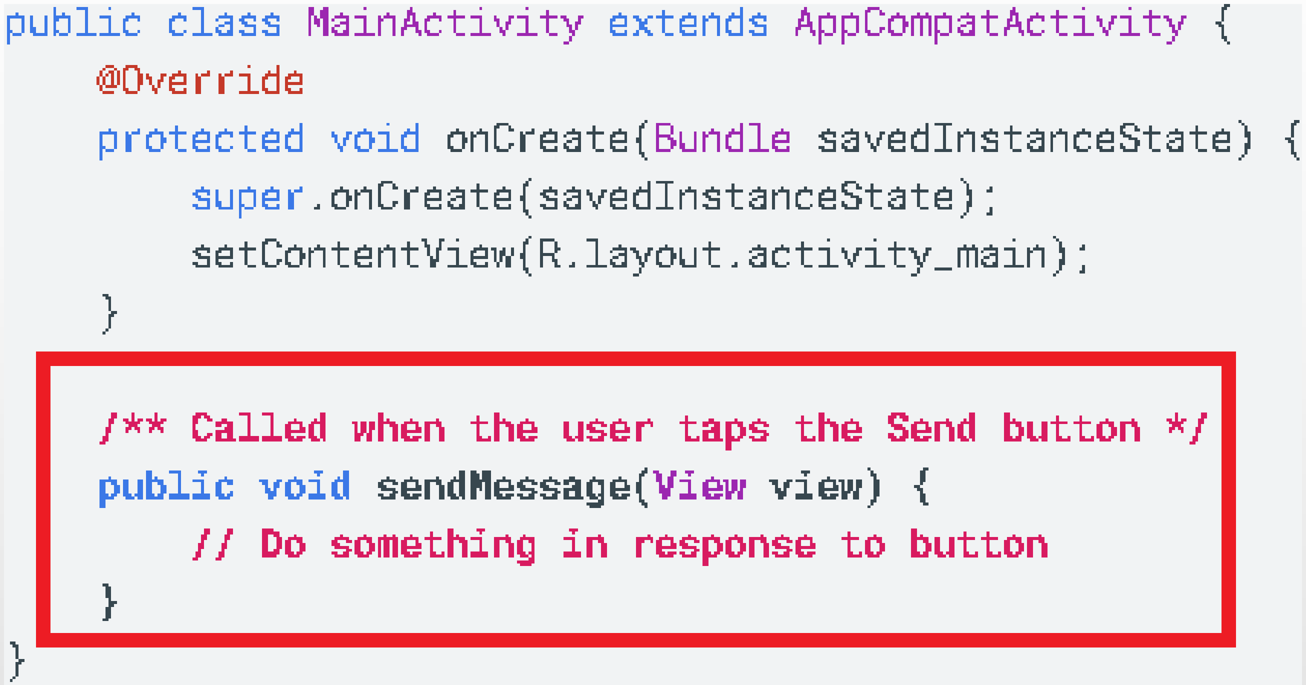
- Imagine we have an app that shows an activity that consists of a single screen with `textView` and `editText` fields, a Send button at least
- In this example, we add some code to the MainActivity that starts a new activity to display a message when the user taps the Send button



https://developer.android.com/training/basics/firstapp/starting-activity

# Start another activity in the Java Android app (cont.)

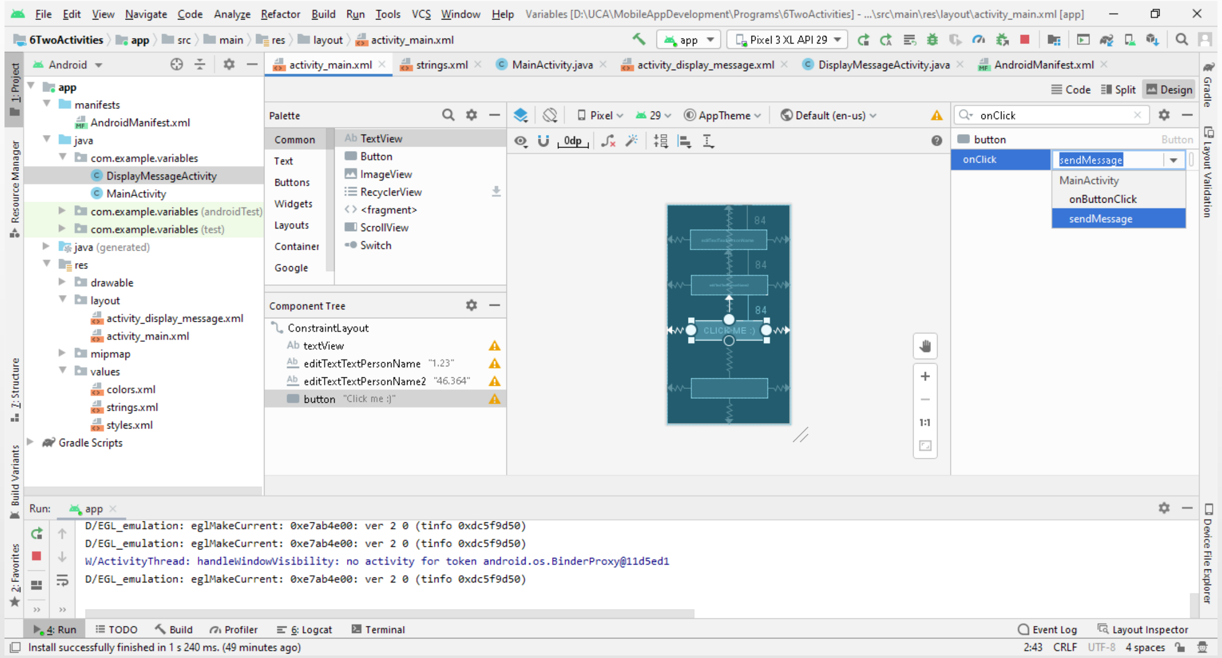
- Respond to the Send button
  - In the file `app>java>com.example.variables>MainActivity`, add the method `sendMessage()`:



```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** Called when the user taps the Send button */
    public void sendMessage(View view) {
        // Do something in response to button
    }
}
```

# Start another activity in the Java Android app (cont.)

- Return to the `activity_main.xml` file to call the method from the button:
  - Select the button in the `Layout Editor`
  - In the `Attributes` window, locate the `onClick` property and select `sendMessage [MainActivity]` from its drop-down list



button — Button

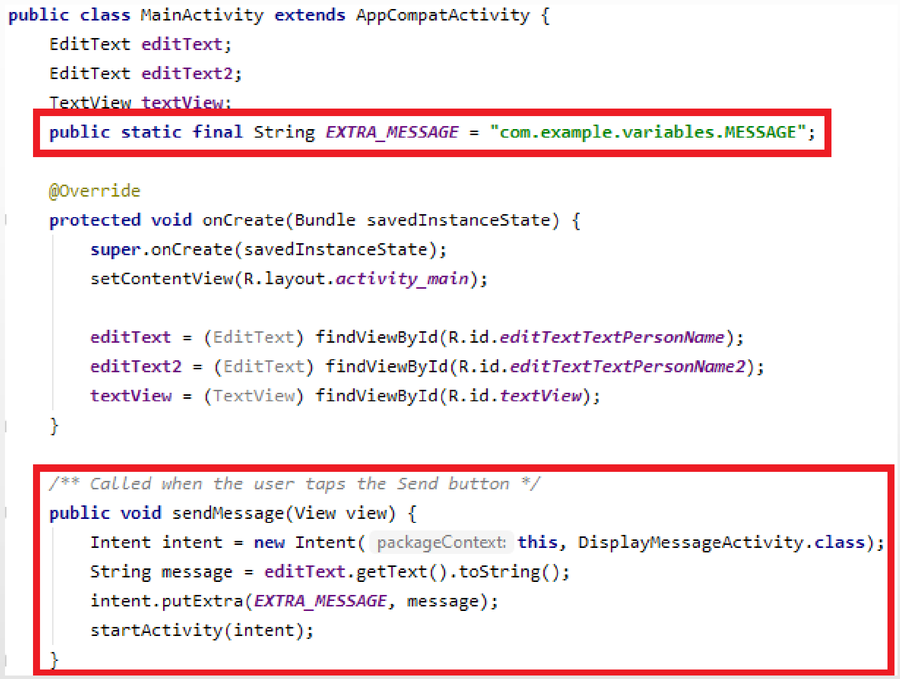
onClick: sendMessage

```
D/EGL_emulation: eglMakeCurrent: 0xe7ab4e00: ver 2 0 (tinfo 0xdc5f9d50)
D/EGL_emulation: eglMakeCurrent: 0xe7ab4e00: ver 2 0 (tinfo 0xdc5f9d50)
W/ActivityThread: handleWindowVisibility: no activity for token android.os.BinderProxy@11d5ed1
D/EGL_emulation: eglMakeCurrent: 0xe7ab4e00: ver 2 0 (tinfo 0xdc5f9d50)
```

Install successfully finished in 1 s 240 ms. (49 minutes ago)

## Start another activity in the Java Android app (cont.)

- Build an intent
  - An Intent is an object that provides runtime binding between separate components, such as two activities. The Intent represents an app's intent to do something. We can use intents for a wide variety of tasks, but in this example, our intent starts another activity.
  - In MainActivity, add the `EXTRA_MESSAGE` constant and the `sendMessage()` code, as shown:



```
public class MainActivity extends AppCompatActivity {
    EditText editText;
    EditText editText2;
    TextView textView;
    public static final String EXTRA_MESSAGE = "com.example.variables.MESSAGE";

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

        editText = (EditText) findViewById(R.id.editTextTextPersonName);
        editText2 = (EditText) findViewById(R.id.editTextTextPersonName2);
        textView = (TextView) findViewById(R.id.textView);
    }

    /** Called when the user taps the Send button */
    public void sendMessage(View view) {
        Intent intent = new Intent( packageContext: this, DisplayMessageActivity.class);
        String message = editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
```

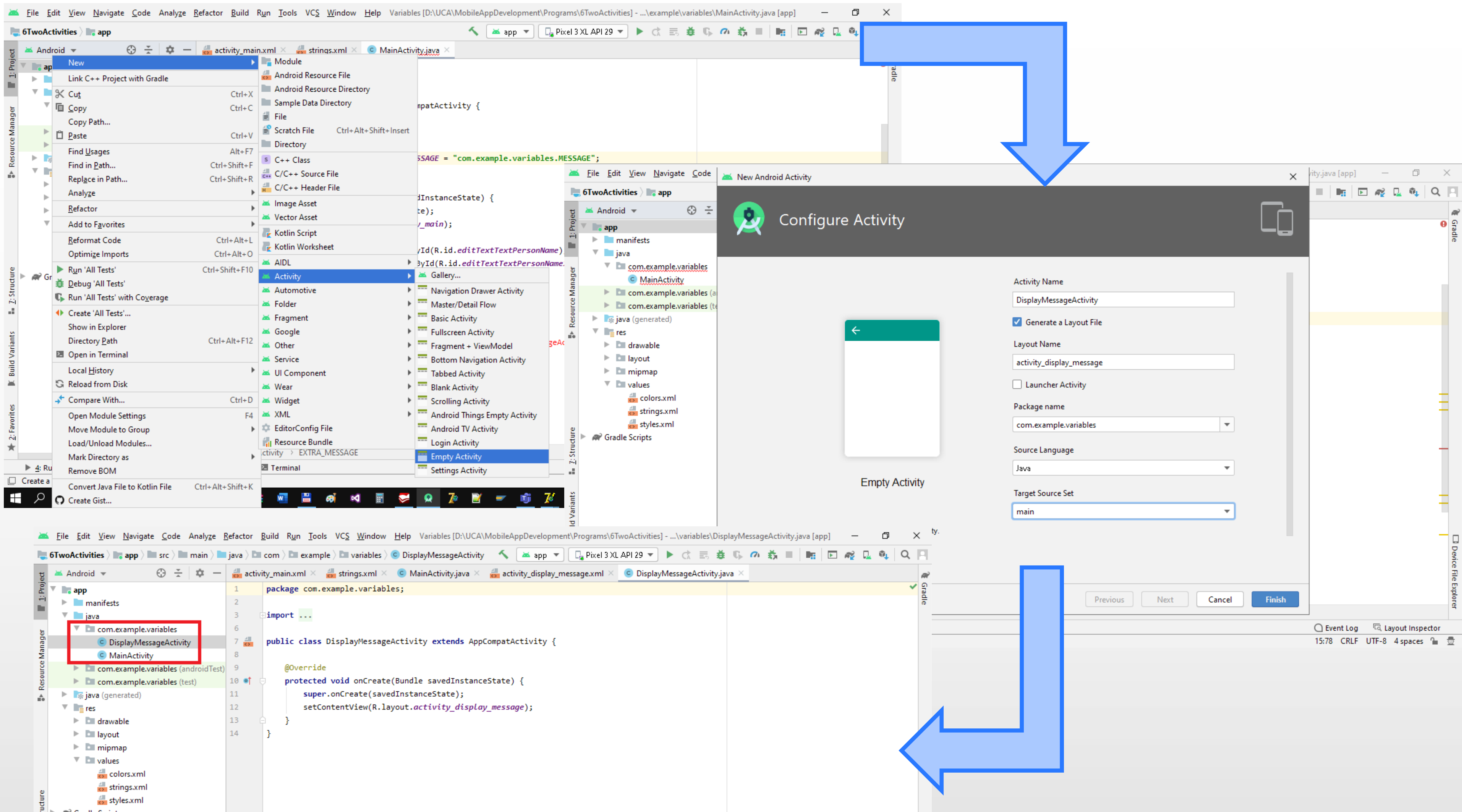
## Start another activity in the Java Android app (cont.)

- An error still remains for `DisplayMessageActivity`, but that's okay now. We fix it later.
- Here's what's going on in `sendMessage()`:
  - The Intent constructor takes two parameters, a `Context` and a `Class`
  - The `Context` parameter is used first because the `Activity` class is a subclass of `Context`
  - The `Class` parameter of the app component, to which the system delivers the `Intent`, is, in this case, the activity to start
  - The `putExtra()` method adds the value of `EditText` to the intent. An `Intent` can carry data types as key-value pairs called extras.
  - Our key is a public constant EXTRA_MESSAGE because the next activity uses the key to retrieve the text value. It's a good practice to define keys for intent extras with our app's package name as a prefix. This ensures that the keys are unique, in case our app interacts with other apps.
  - The `startActivity()` method starts an instance of the `DisplayMessageActivity` that's specified by the Intent. Next, we need to create that class.

## Start another activity in the Java Android app (cont.)

- Create the second activity
  - In the `Project` window, right-click the app folder and select `New>Activity>Empty Activity`
  - In the `Configure Activity` window, enter `"DisplayMessageActivity"` for Activity Name. Leave all other properties set to their defaults and click Finish.
- Android Studio automatically does three things:
  - Creates the `DisplayMessageActivity` file
  - Creates the layout file `activity_display_message.xml`, which corresponds with the `DisplayMessageActivity` file
  - Adds the required `<activity>` element in `AndroidManifest.xml`
- If we run the app and tap the button on the first activity, the second activity starts but is empty. This is because the second activity uses the empty layout provided by the template.

# Start another activity in the Java Android app (cont.)



Activity Name: DisplayMessageActivity

[✓] Generate a Layout File

Layout Name: activity_display_message

[ ] Launcher Activity

Package name: com.example.variables

Source Language: Java

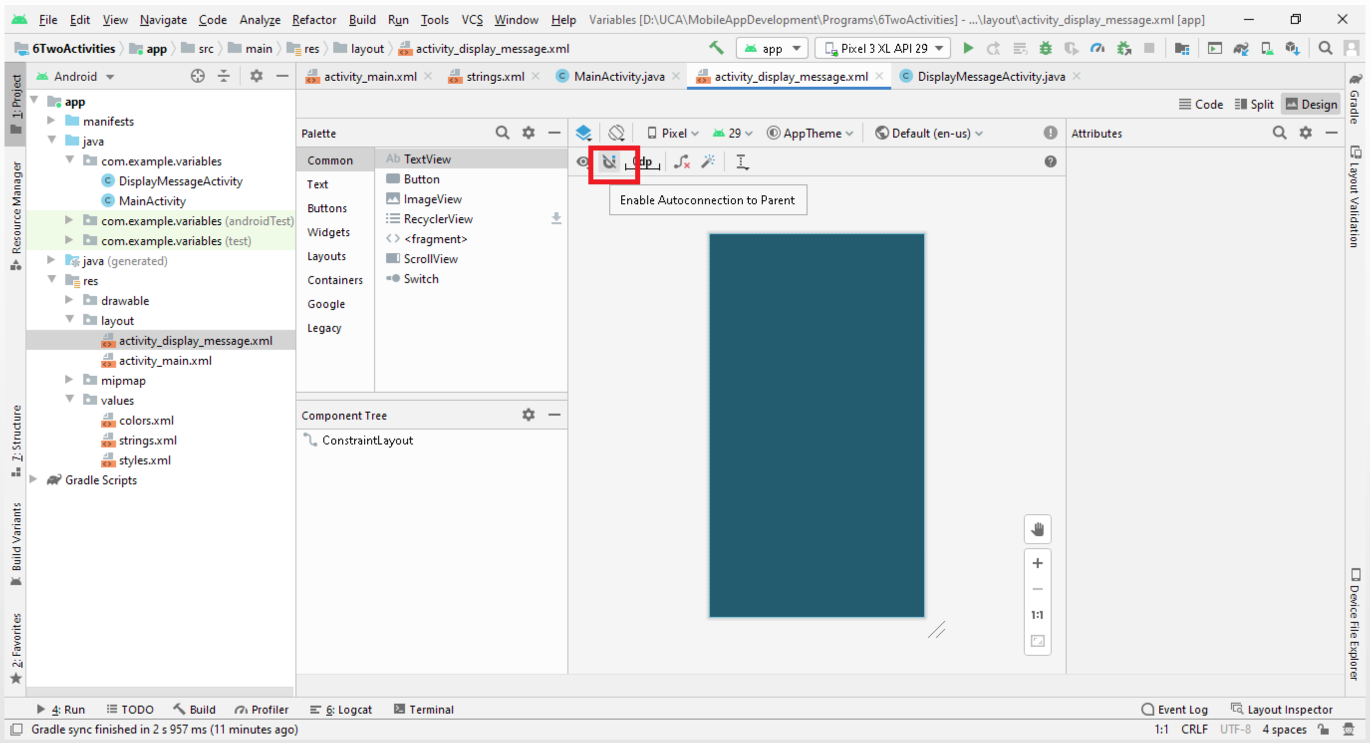
Target Source Set: main

Variables [D:\UCA\MobileAppDevelopment\Programs\6TwoActivities] - ...\variables\DisplayMessageActivity.java [app]

```
package com.example.variables;

import ...

public class DisplayMessageActivity extends AppCompatActivity {

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

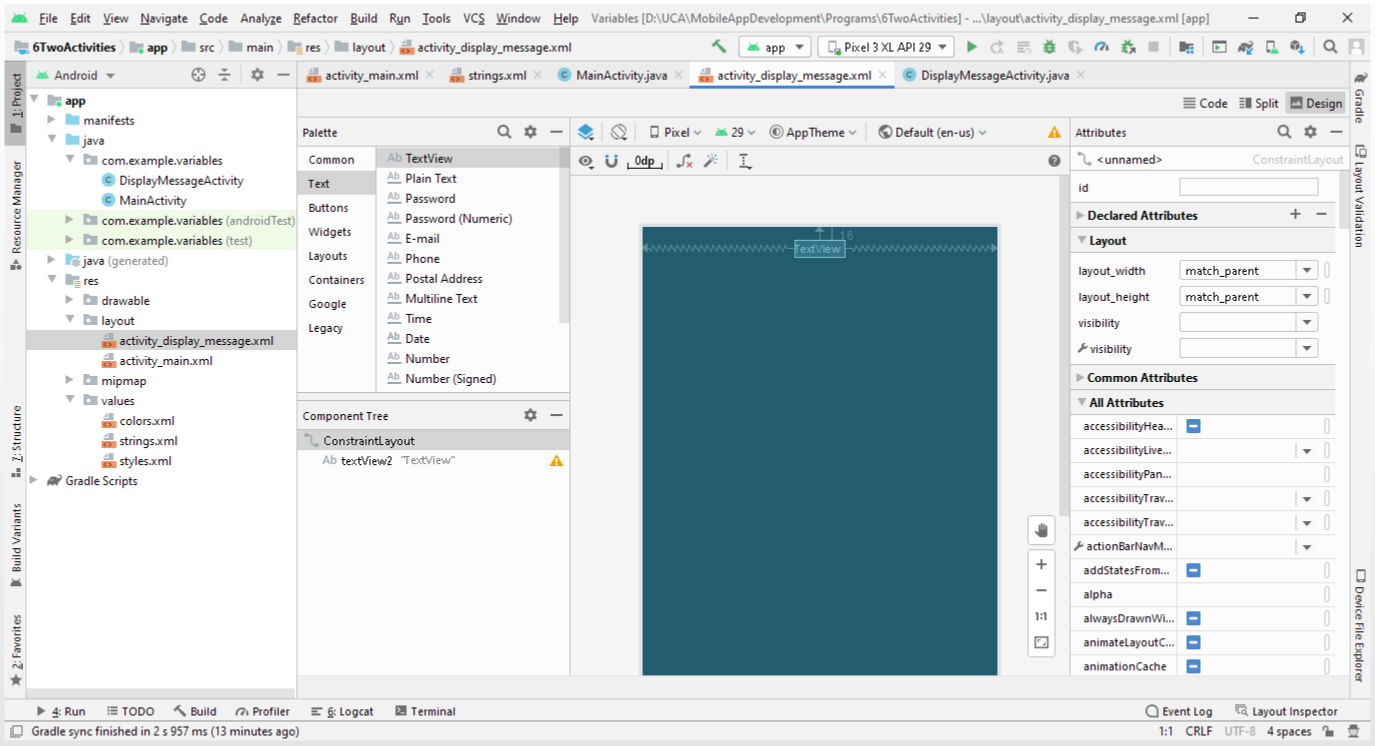
# Start another activity in the Java Android app (cont.)

- Add a text view:
  - Open the file `app>res>layout>activity_display_message.xml`
  - Click `Enable Autoconnection to Parent` in the toolbar. This enables Autoconnect


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Variables [D:\UCA\MobileAppDevelopment\Programs\6TwoActivities] - ...\layout\activity_display_message.xml [app]
6TwoActivities app src main res layout activity_display_message.xml
app
Pixel 3 XL API 29
Android
activity_main.xml strings.xml MainActivity.java activity_display_message.xml DisplayMessageActivity.java
Code Split Design
app
manifests
java
com.example.variables
DisplayMessageActivity
MainActivity
com.example.variables (androidTest)
com.example.variables (test)
java (generated)
res
drawable
layout
activity_display_message.xml
activity_main.xml
mipmap
values
colors.xml
strings.xml
styles.xml
Gradle Scripts
Palette
Common
Text
Buttons
Widgets
Layouts
Containers
Google
Legacy
TextView
Button
ImageView
RecyclerView
<fragment>
ScrollView
Switch
Pixel 29 AppTheme Default (en-us)
Attributes
0dp
Enable Autoconnection to Parent
Component Tree
ConstraintLayout
4: Run TODO Build Profiler 6: Logcat Terminal
Event Log Layout Inspector
Gradle sync finished in 2 s 957 ms (11 minutes ago)
1:1 CRLF UTF-8 4 spaces
1: Project Resource Manager 7: Structure Build Variants 2: Favorites
Gradle Layout Validation Device File Explorer

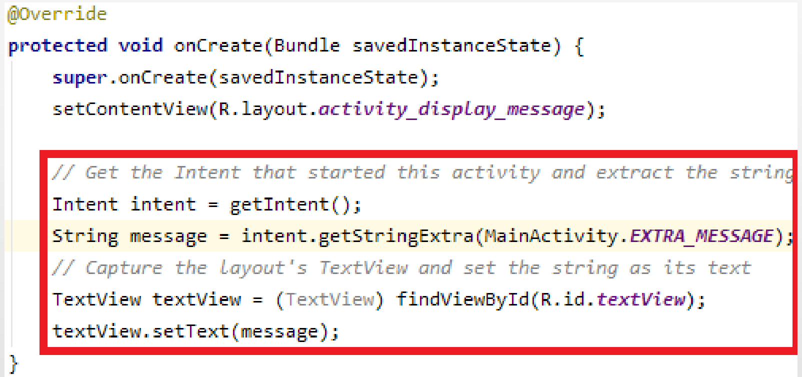
## Start another activity in the Java Android app (cont.)

- In the `Palette` panel, click `Text`, drag a `TextView` into the layout, and drop it near the top-center of the layout so that it snaps to the vertical line that appears. Autoconnect adds left and right constraints in order to place the view in the horizontal center.



## Start another activity in the Java Android app (cont.)

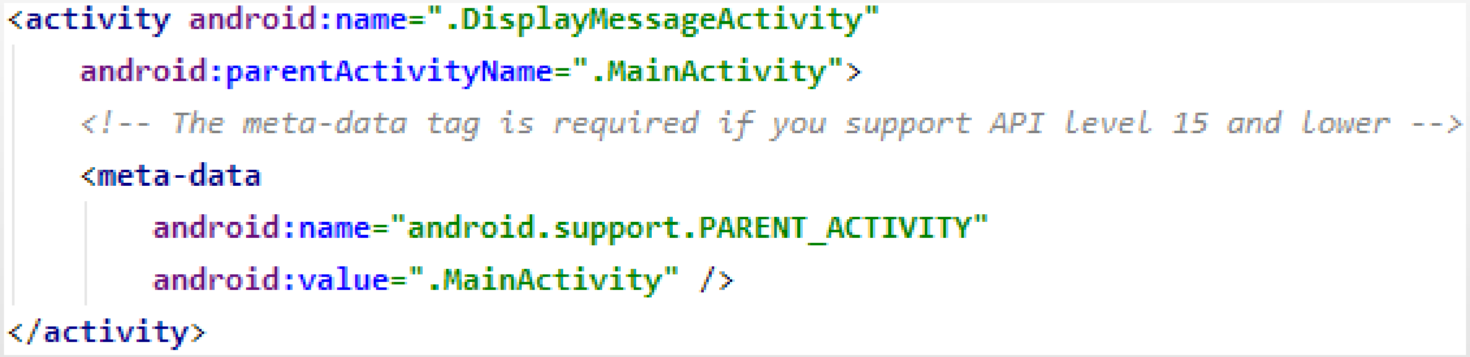
  - Create one more constraint from the top of the text view to the top of the layout
- Optionally, we can make some adjustments to the text style if we expand `textAppearance` in the `Common Attributes` panel of the `Attributes` window, and change attributes such as `textSize` and `textColor`
- Display the message
  - In `DisplayMessageActivity`, add the following code to the `onCreate()` method:



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

    // Get the Intent that started this activity and extract the string
    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
    // Capture the layout's TextView and set the string as its text
    TextView textView = (TextView) findViewById(R.id.textView);
    textView.setText(message);
}
```

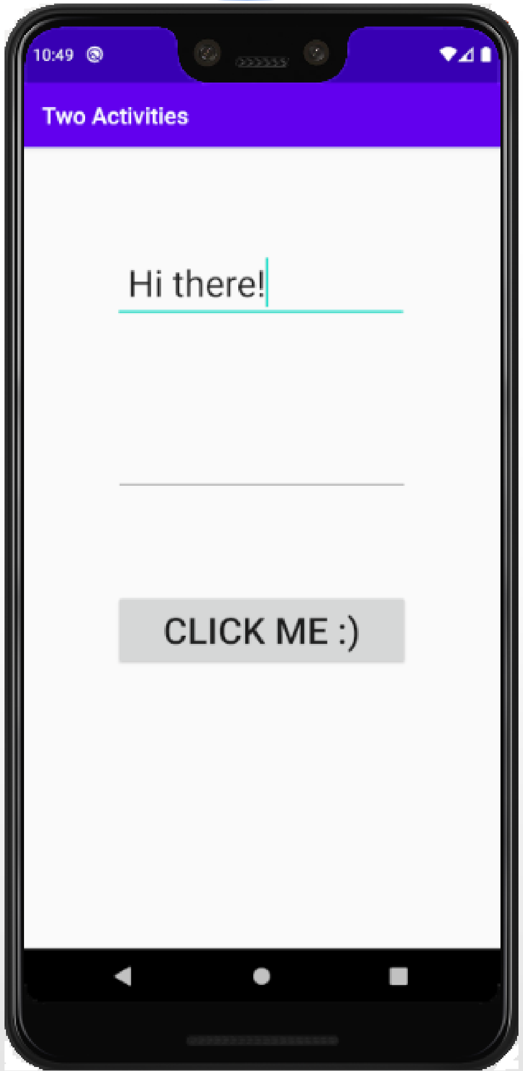
## Start another activity in the Java Android app (cont.)

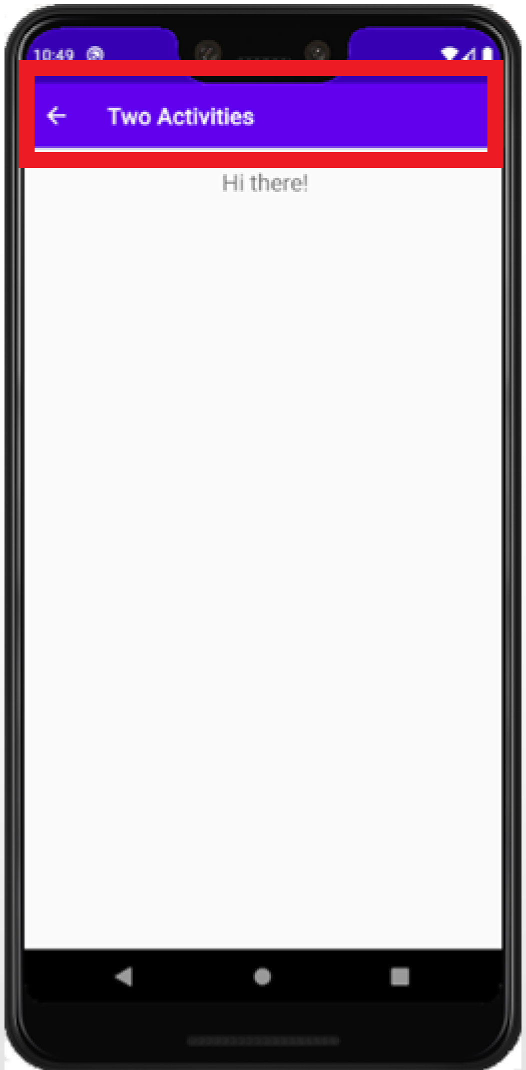
- Add upward navigation
  - **Each screen in our app** that is not the main entry point, which are all the screens that aren't the home screen, **must provide navigation that directs the user to the logical parent screen in the app's hierarchy**. To do this, add an **Up** button in the `app bar`.
  - To add an **Up** button, we need to declare which activity is the logical parent in the `AndroidManifest.xml` file. Open the file at `app>manifests>AndroidManifest.xml`, locate the `<activity>` tag for `DisplayMessageActivity`, and replace it with the following:



```
<activity android:name=".DisplayMessageActivity"
    android:parentActivityName=".MainActivity">
    <!-- The meta-data tag is required if you support API level 15 and lower -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>
```

# Start another activity in the Java Android app (cont.)





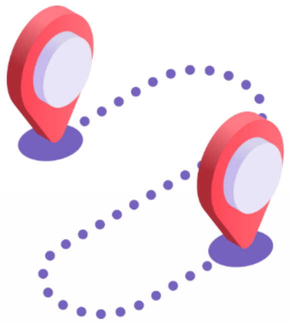
## Cast operator

- ***The cast operator – (type) – attempts to convert the type of its operand to type***. This operator exists because the compiler will not allow to convert a value from one type to another in which information will be lost without specifying the intention to do so (via the cast operator).
  - For instance, when presented with short *s*=1.65+3; the compiler can report an error because attempting to convert a 64-bit double precision floating-point value to a 16-bit signed short integer results in the loss of the fraction .65, so *s* would contain 4 instead of 4.65.
  - Recognizing that information loss might not always be a problem, Java permits us to state our intention explicitly by casting to the target type. For instance, short **s=(short)1.65+3**; tells the compiler that we want 1.65+3 to be converted to a short integer, and we understand that the fraction will disappear.

## Cast operator (cont.)

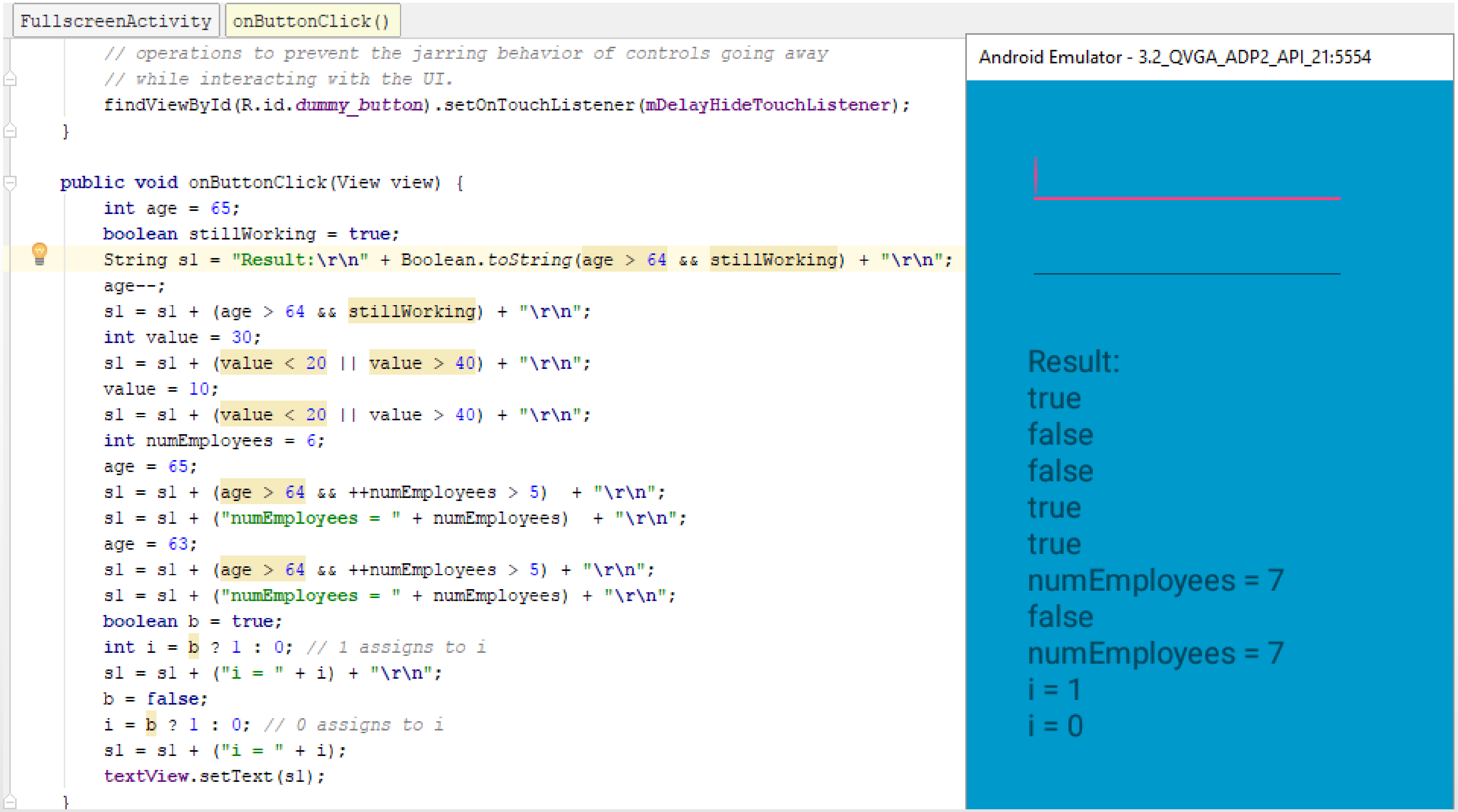
- *Java supports the following primitive-type conversions via **cast** operators*:
  - Byte integer to character
  - Short integer to byte integer or character
  - Character to byte integer or short integer
  - Integer to byte integer, short integer, or character
  - Long integer to byte integer, short integer, character, or integer
  - Floating-point to byte integer, short integer, character, integer, or long integer
  - Double precision floating-point to byte integer, short integer, character, integer, long integer, or floating-point

## Conditional operators



- **The conditional operators are conditional AND (&&), conditional OR (||), and conditional (?:).** First two operators always evaluate their left operand (a Boolean expression that evaluates to true or false) and conditionally evaluate their right operand (another Boolean expression). The third operator evaluates one of two operands based on a third Boolean operand.

# Conditional operators (cont.)



```
        // operations to prevent the jarring behavior of controls going away
        // while interacting with the UI.
        findViewById(R.id.dummy_button).setOnTouchListener(mDelayHideTouchListener);
    }

    public void onButtonClick(View view) {
        int age = 65;
        boolean stillWorking = true;
        String sl = "Result:\r\n" + Boolean.toString(age > 64 && stillWorking) + "\r\n";
        age--;
        sl = sl + (age > 64 && stillWorking) + "\r\n";
        int value = 30;
        sl = sl + (value < 20 || value > 40) + "\r\n";
        value = 10;
        sl = sl + (value < 20 || value > 40) + "\r\n";
        int numEmployees = 6;
        age = 65;
        sl = sl + (age > 64 && ++numEmployees > 5)  + "\r\n";
        sl = sl + ("numEmployees = " + numEmployees)  + "\r\n";
        age = 63;
        sl = sl + (age > 64 && ++numEmployees > 5) + "\r\n";
        sl = sl + ("numEmployees = " + numEmployees) + "\r\n";
        boolean b = true;
        int i = b ? 1 : 0; // 1 assigns to i
        sl = sl + ("i = " + i) + "\r\n";
        b = false;
        i = b ? 1 : 0; // 0 assigns to i
        sl = sl + ("i = " + i);
        textView.setText(sl);
    }
```

```
Result:
true
false
false
true
true
numEmployees = 7
false
numEmployees = 7
i = 1
i = 0
```

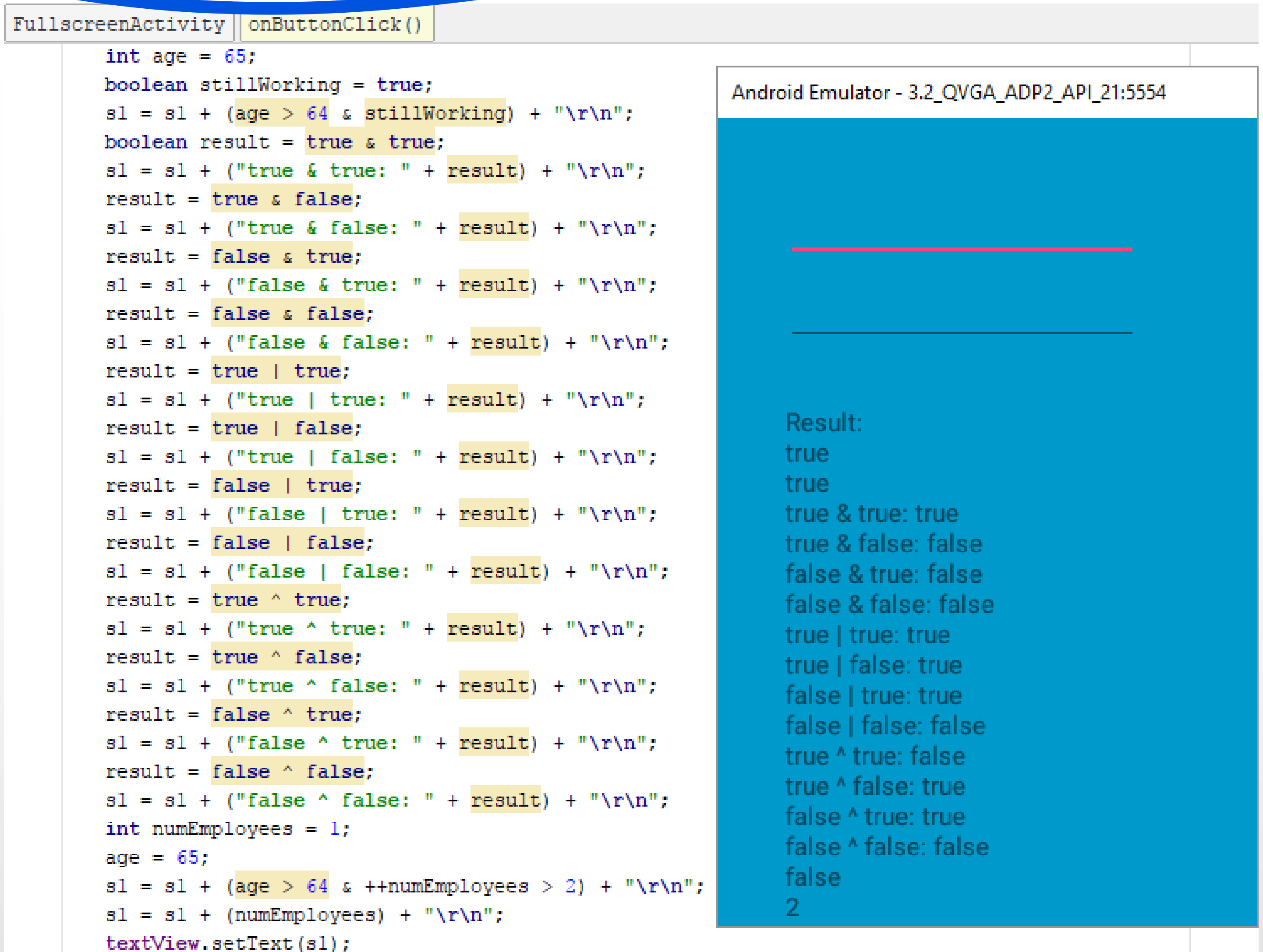
## Equality operators

- **The equality operators are equality (==) and inequality (!=).** These operators compare their operands to determine whether they are equal or unequal. The first operator returns true when equal and the second operator returns true when unequal.
    - For instance, each of 2 == 2 and 2 != 3 evaluates to true, whereas each of 2 == 4 and 4 != 4 evaluates to false.

## Logical operators

- The logical operators consist of logical AND (&), logical complement (!), logical exclusive OR (^), and logical inclusive OR (|). Although these operators are similar to their bitwise counterparts, whose operands must be integer/character, **the operands passed to the logical operators must be Boolean**.
    - For example, !false returns true. Also, when confronted with age>64 & stillWorking, logical AND evaluates both subexpressions; *there's no short-circuiting*. This same pattern holds for logical exclusive OR and logical inclusive OR.

# Logical operators (cont.)



```
int age = 65;
boolean stillWorking = true;
sl = sl + (age > 64 & stillWorking) + "\r\n";
boolean result = true & true;
sl = sl + ("true & true: " + result) + "\r\n";
result = true & false;
sl = sl + ("true & false: " + result) + "\r\n";
result = false & true;
sl = sl + ("false & true: " + result) + "\r\n";
result = false & false;
sl = sl + ("false & false: " + result) + "\r\n";
result = true | true;
sl = sl + ("true | true: " + result) + "\r\n";
result = true | false;
sl = sl + ("true | false: " + result) + "\r\n";
result = false | true;
sl = sl + ("false | true: " + result) + "\r\n";
result = false | false;
sl = sl + ("false | false: " + result) + "\r\n";
result = true ^ true;
sl = sl + ("true ^ true: " + result) + "\r\n";
result = true ^ false;
sl = sl + ("true ^ false: " + result) + "\r\n";
result = false ^ true;
sl = sl + ("false ^ true: " + result) + "\r\n";
result = false ^ false;
sl = sl + ("false ^ false: " + result) + "\r\n";
int numEmployees = 1;
age = 65;
sl = sl + (age > 64 & ++numEmployees > 2) + "\r\n";
sl = sl + (numEmployees) + "\r\n";
textView.setText(sl);
```

Android Emulator - 3.2_QVGA_ADP2_API_21:5554

```
Result:
true
true
true & true: true
true & false: false
false & true: false
false & false: false
true | true: true
true | false: true
false | true: true
false | false: false
true ^ true: false
true ^ false: true
false ^ true: true
false ^ false: false
false
2
```

## Multiplicative operators

- **The multiplicative operators consist of multiplication (*), division (/), and remainder (%).**
  - Multiplication returns the product of its operands (such as 6*4 returns 24), integer division returns the quotient of dividing its left operand by its right operand (such as 6/4 returns 1), and remainder returns the remainder of dividing its left operand by its right operand (such as 6%4 returns 2).

## Relational operators

- **The relational operators consist of larger than (>), larger than or equal to (>=), less than (<), less than or equal to (<=), and type checking (instanceof).** First four operators compare their operands and return true when the left operand is larger than, larger than or equal to, less than, or less than or equal to the right operand. For instance, each of 5.0>3, 2>=2, 16.1<303.3, and 54.0<=54.0 evaluates to true.
- The type-checking operator (instanceof) is used to determine if an object belongs to a specific type, returning true when this is the case.

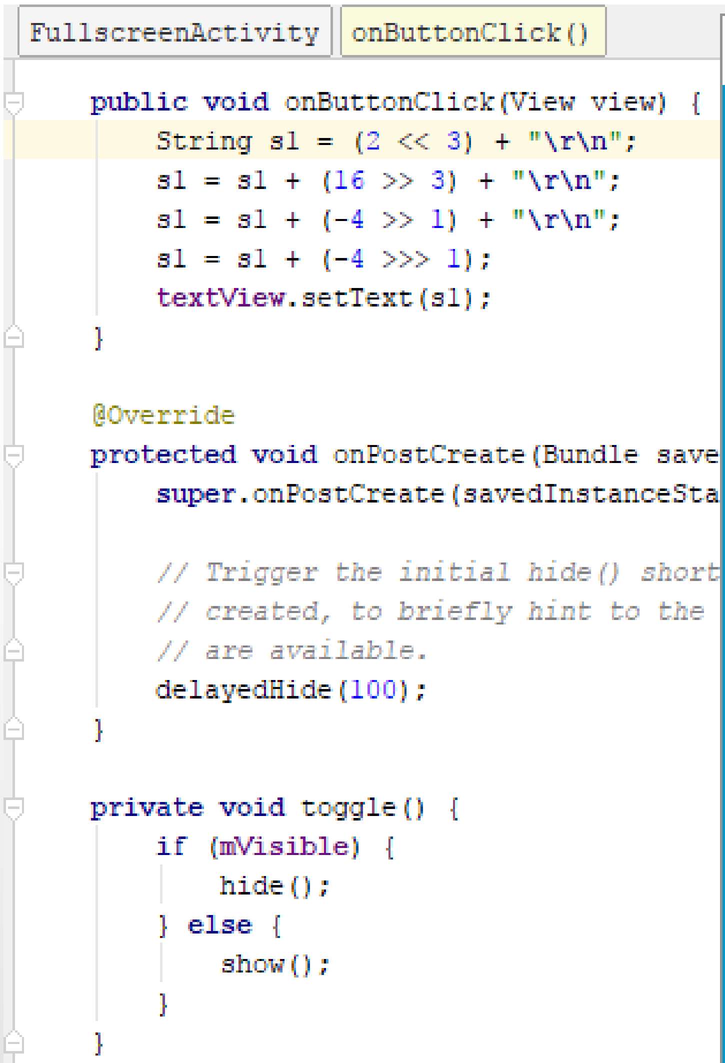
```
"abc" instanceof String
```

returns true because "abc" is a String object.

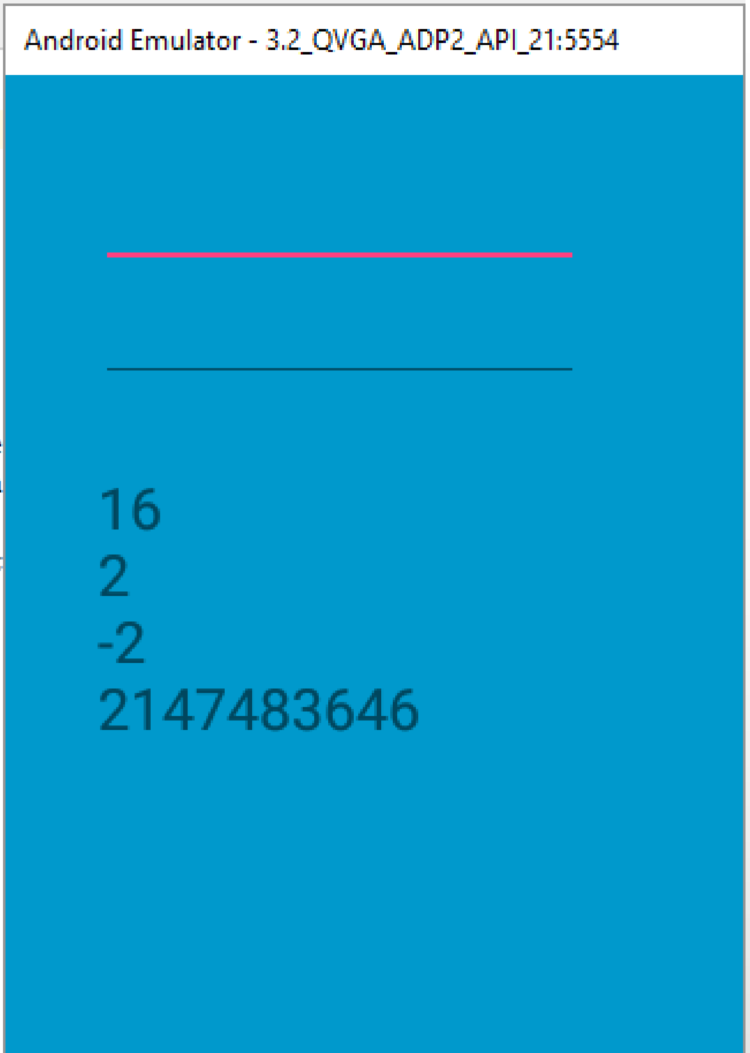
## Shift operators

- **The shift operators consist of left shift (<<), signed right shift (>>), and unsigned right shift (>>>).** Left shift shifts the binary representation of its left operand leftward by the number of positions specified by its right operand.
  - Each left shift is equivalent to multiplying by 2. For example, 2<<3 shifts binary representation left by three positions; the result is equivalent to multiplying 2 by 8. Each right shift is equivalent to dividing by 2. For instance, 16>>3 shifts binary representation right by three positions; the result is equivalent to dividing 16 by 8 (result is 2).
- *Signed right shift includes the sign bit in the shift, whereas unsigned right shift ignores the sign bit*. As a result, signed right shift preserves negative numbers, but unsigned right shift doesn't. For example, -4>>1 (the equivalent of -4/2) evaluates to -2, whereas –4>>>1 evaluates to 2147483646.

## Shift operators (cont.)


FullscreenActivity | onButtonClick()

```
public void onButtonClick(View view) {
    String s1 = (2 << 3) + "\r\n";
    s1 = s1 + (16 >> 3) + "\r\n";
    s1 = s1 + (-4 >> 1) + "\r\n";
    s1 = s1 + (-4 >>> 1);
    textView.setText(s1);
}

@Override
protected void onPostCreate(Bundle save
    super.onPostCreate(savedInstanceSta

    // Trigger the initial hide() short
    // created, to briefly hint to the
    // are available.
    delayedHide(100);
}

private void toggle() {
    if (mVisible) {
        hide();
    } else {
        show();
    }
}
```


Android Emulator - 3.2_QVGA_ADP2_API_21:5554

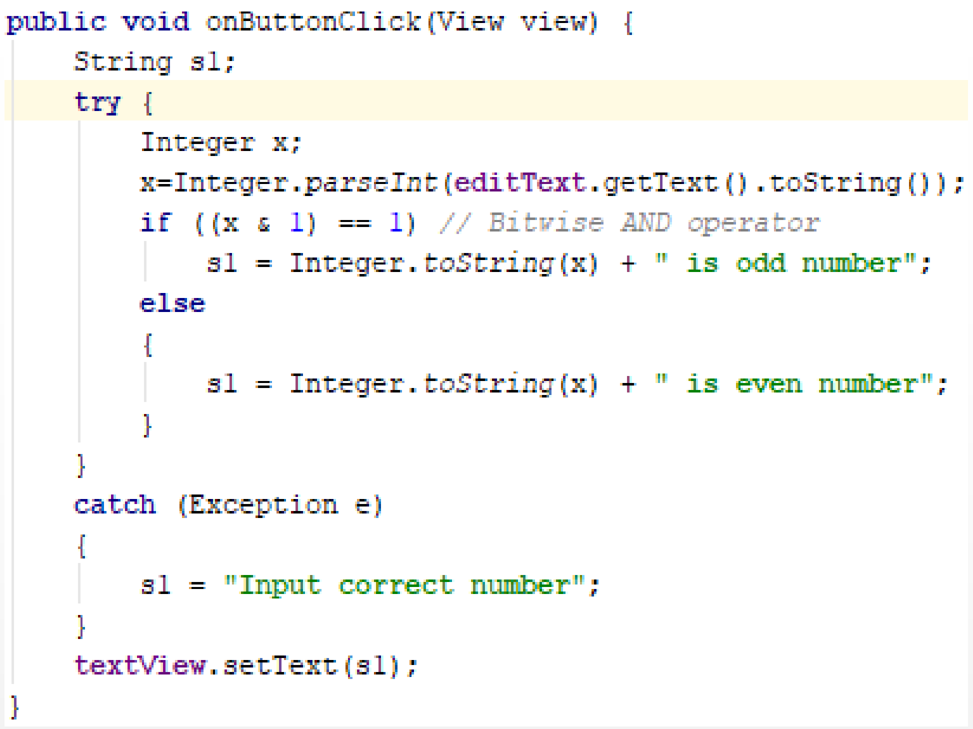
```
16
2
-2
2147483646
```

## If-Else statement

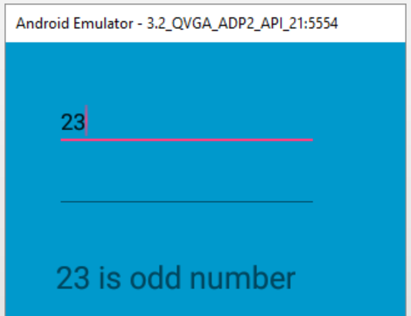
- The **if-else** statement evaluates a Boolean expression and executes one of two statements depending on whether this expression evaluates to true or false. This statement has the following syntax:

```
if (Boolean expression)
     statement1
else
     statement2
```

## If-Else statement (cont.)



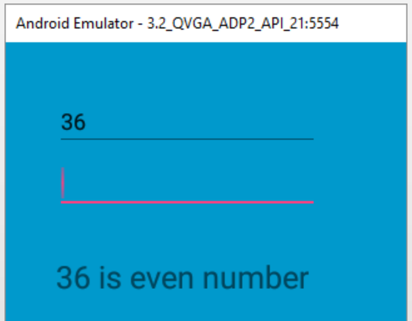
```
public void onButtonClick(View view) {
    String s1;
    try {
        Integer x;
        x=Integer.parseInt(editText.getText().toString());
        if ((x & 1) == 1) // Bitwise AND operator
            s1 = Integer.toString(x) + " is odd number";
        else
        {
            s1 = Integer.toString(x) + " is even number";
        }
    }
    catch (Exception e)
    {
        s1 = "Input correct number";
    }
    textView.setText(s1);
}
```



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

23

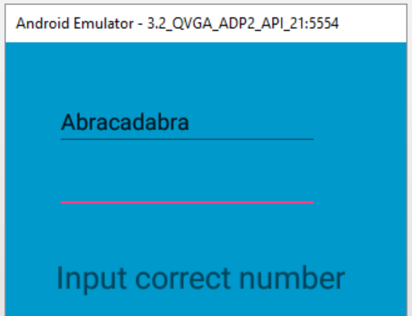
23 is odd number



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

36

36 is even number



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

Abracadabra

Input correct number

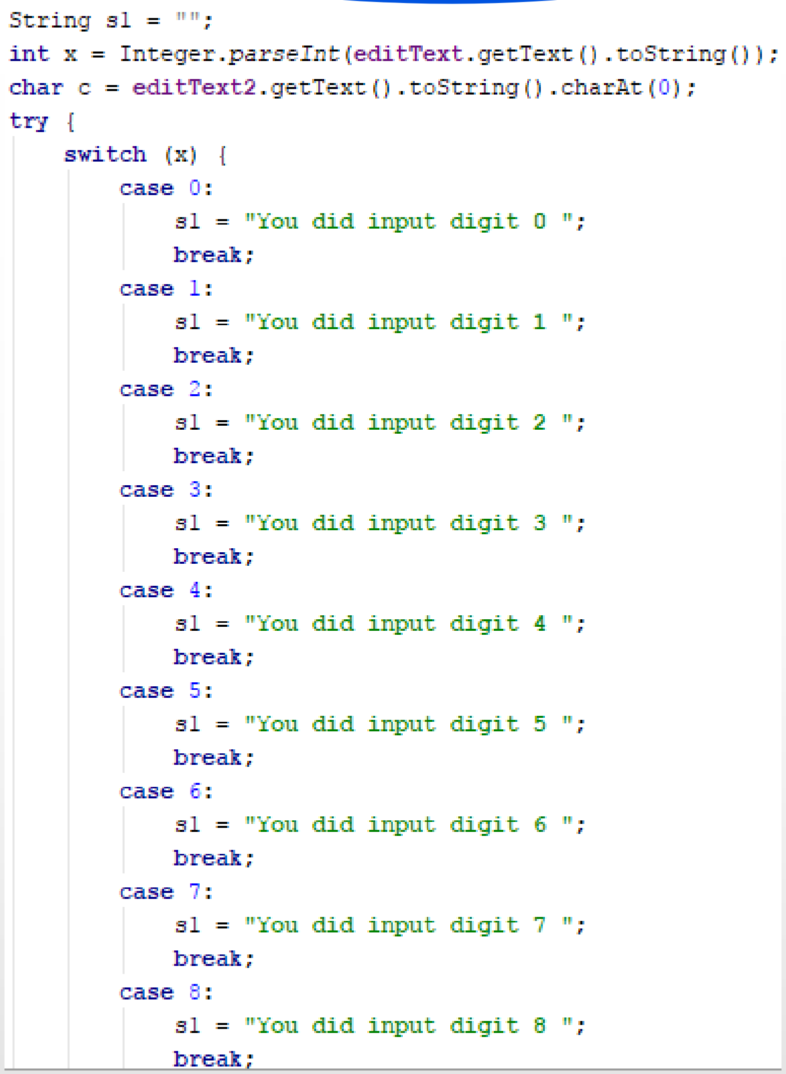
## Switch statement

- The switch statement lets us choose from several execution paths more efficiently than with equivalent chained if-else statements:

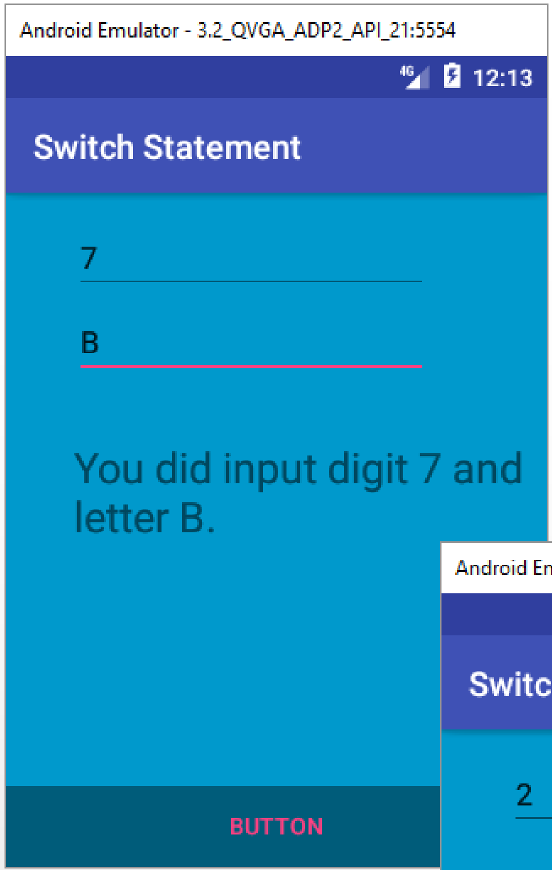
```
switch (selector expression)
{
        case value1: statement1 [break;]
        case value2: statement2 [break;]
        ...
        case valueN: statementN [break;]
        [default: statement]
}
```

  - Switch consists of reserved word *switch*, followed by a selector expression in parentheses, followed by a body of cases. The selector expression is any expression that evaluates to an integer or character value.

# Switch statement (cont.)



```java
String s1 = "";
int x = Integer.parseInt(editText.getText().toString());
char c = editText2.getText().toString().charAt(0);
try {
    switch (x) {
        case 0:
            s1 = "You did input digit 0 ";
            break;
        case 1:
            s1 = "You did input digit 1 ";
            break;
        case 2:
            s1 = "You did input digit 2 ";
            break;
        case 3:
            s1 = "You did input digit 3 ";
            break;
        case 4:
            s1 = "You did input digit 4 ";
            break;
        case 5:
            s1 = "You did input digit 5 ";
            break;
        case 6:
            s1 = "You did input digit 6 ";
            break;
        case 7:
            s1 = "You did input digit 7 ";
            break;
        case 8:
            s1 = "You did input digit 8 ";
            break;
```



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

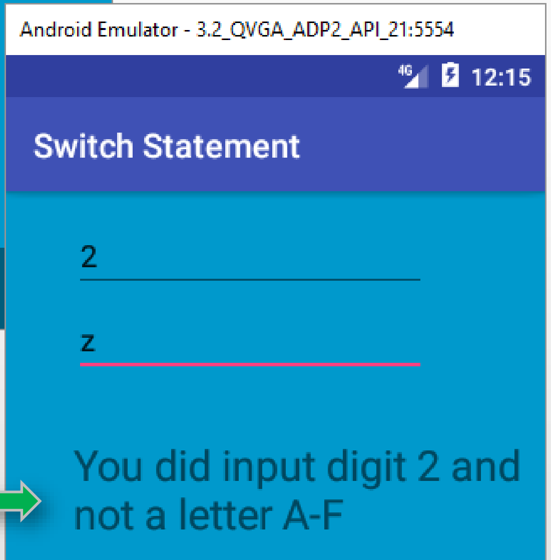
12:13

Switch Statement

7

B

You did input digit 7 and letter B.



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

12:15

Switch Statement

2

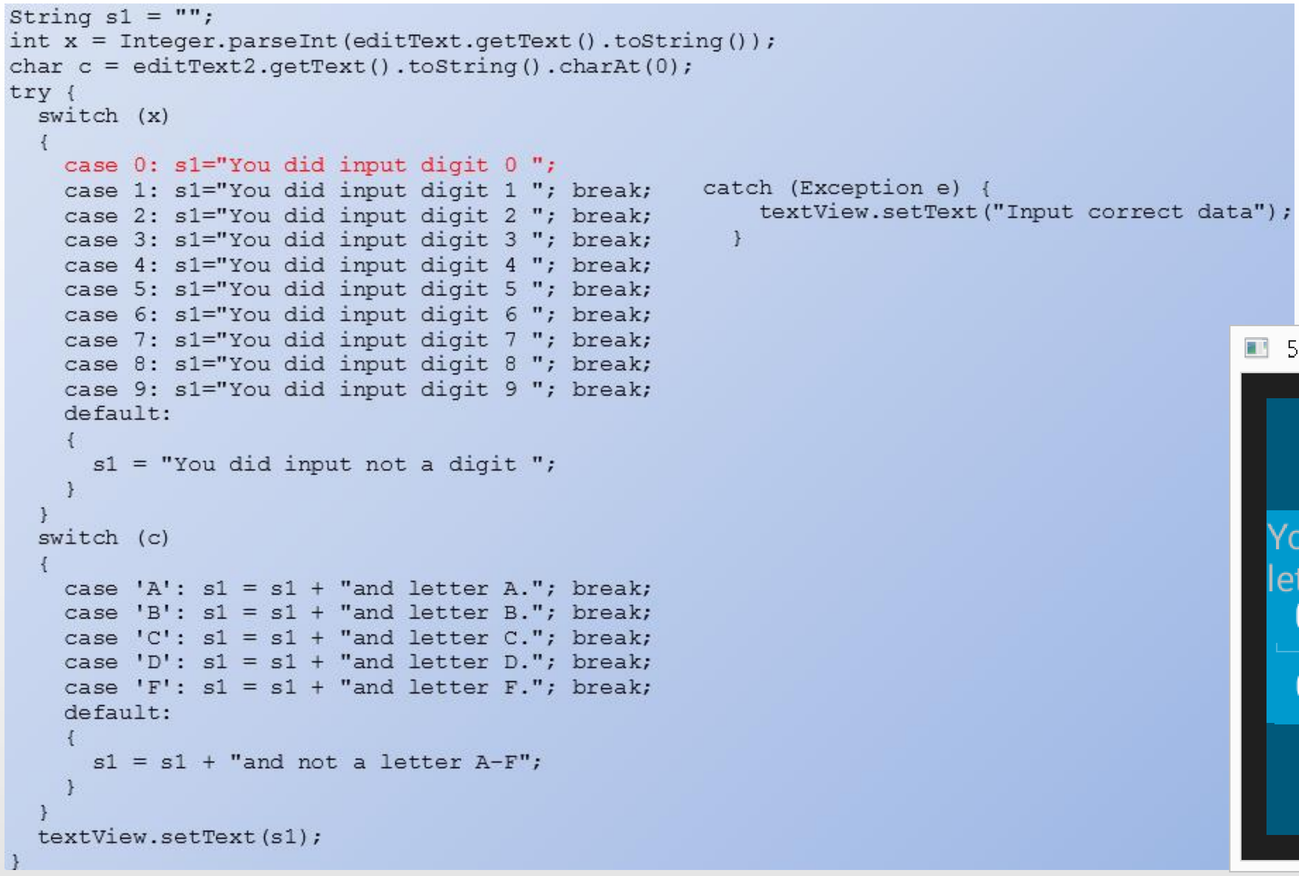
z

You did input digit 2 and not a letter A-F

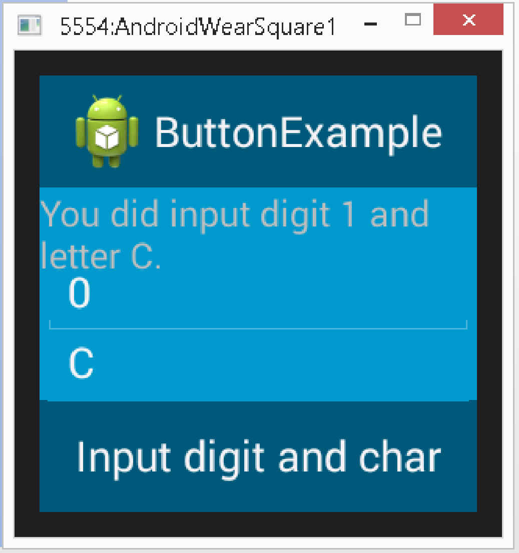
We see this screenshot, but we do not have the Java code for this message :( What Java code should we have?

## Switch statement (cont.)

- Why do we have the output "You did input **digit 1** and letter C" if we did input **digit 0** and letter C?



```
String s1 = "";
int x = Integer.parseInt(editText.getText().toString());
char c = editText2.getText().toString().charAt(0);
try {
  switch (x)
  {
    case 0: s1="You did input digit 0 ";
    case 1: s1="You did input digit 1 "; break;    catch (Exception e) {
    case 2: s1="You did input digit 2 "; break;        textView.setText("Input correct data");
    case 3: s1="You did input digit 3 "; break;      }
    case 4: s1="You did input digit 4 "; break;
    case 5: s1="You did input digit 5 "; break;
    case 6: s1="You did input digit 6 "; break;
    case 7: s1="You did input digit 7 "; break;
    case 8: s1="You did input digit 8 "; break;
    case 9: s1="You did input digit 9 "; break;
    default:
    {
      s1 = "You did input not a digit ";
    }
  }
  switch (c)
  {
    case 'A': s1 = s1 + "and letter A."; break;
    case 'B': s1 = s1 + "and letter B."; break;
    case 'C': s1 = s1 + "and letter C."; break;
    case 'D': s1 = s1 + "and letter D."; break;
    case 'F': s1 = s1 + "and letter F."; break;
    default:
    {
      s1 = s1 + "and not a letter A-F";
    }
  }
  textView.setText(s1);
}
```



5554:AndroidWearSquare1

ButtonExample

You did input digit 1 and letter C.

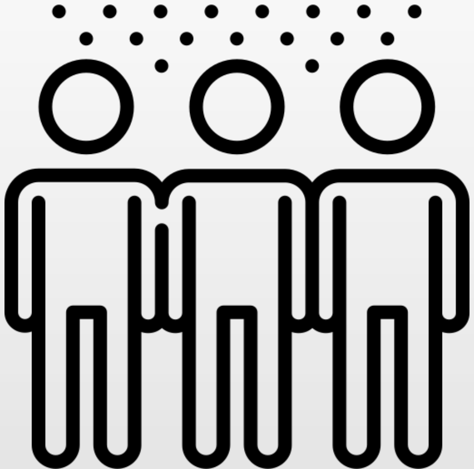
0

C

Input digit and char

## Loop statements: for, while, do-while

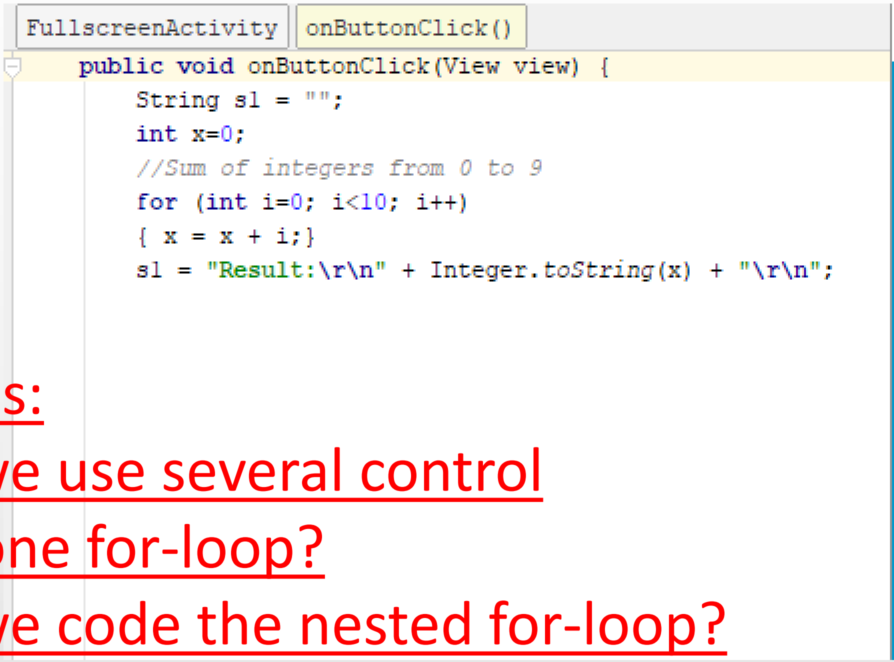
- Java provides three kinds of loop statements: *for*, *while*, and *do-while*.
- The *break*, *labeled break*, *continue*, and *labeled continue* statements are used in Java loops



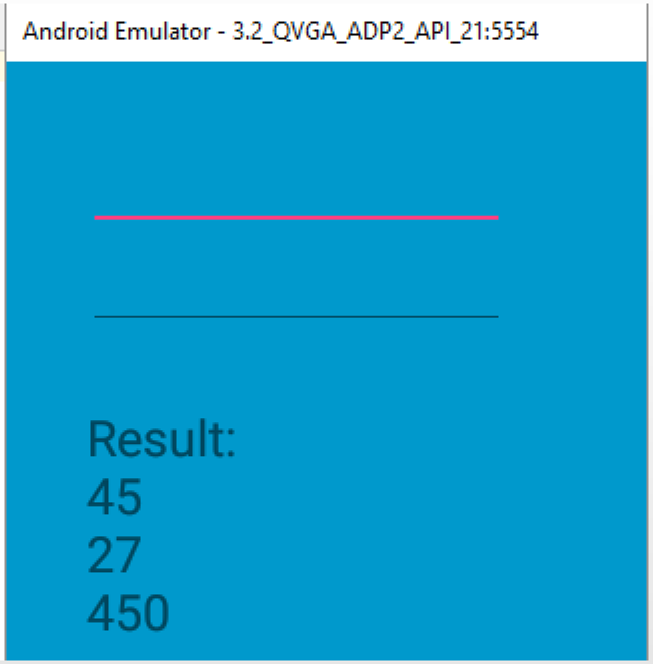
## Loop statements: for statement

- The ***for*** *statement* lets us loop over a statement a specific number of times or even indefinitely:

```
for ([initialize]; [test]; [update])
      statement
```



```
FullscreenActivity   onButtonClick()
    public void onButtonClick(View view) {
        String s1 = "";
        int x=0;
        //Sum of integers from 0 to 9
        for (int i=0; i<10; i++)
        { x = x + i;}
        s1 = "Result:\r\n" + Integer.toString(x) + "\r\n";
```



Android Emulator - 3.2_QVGA_ADP2_API_21:5554

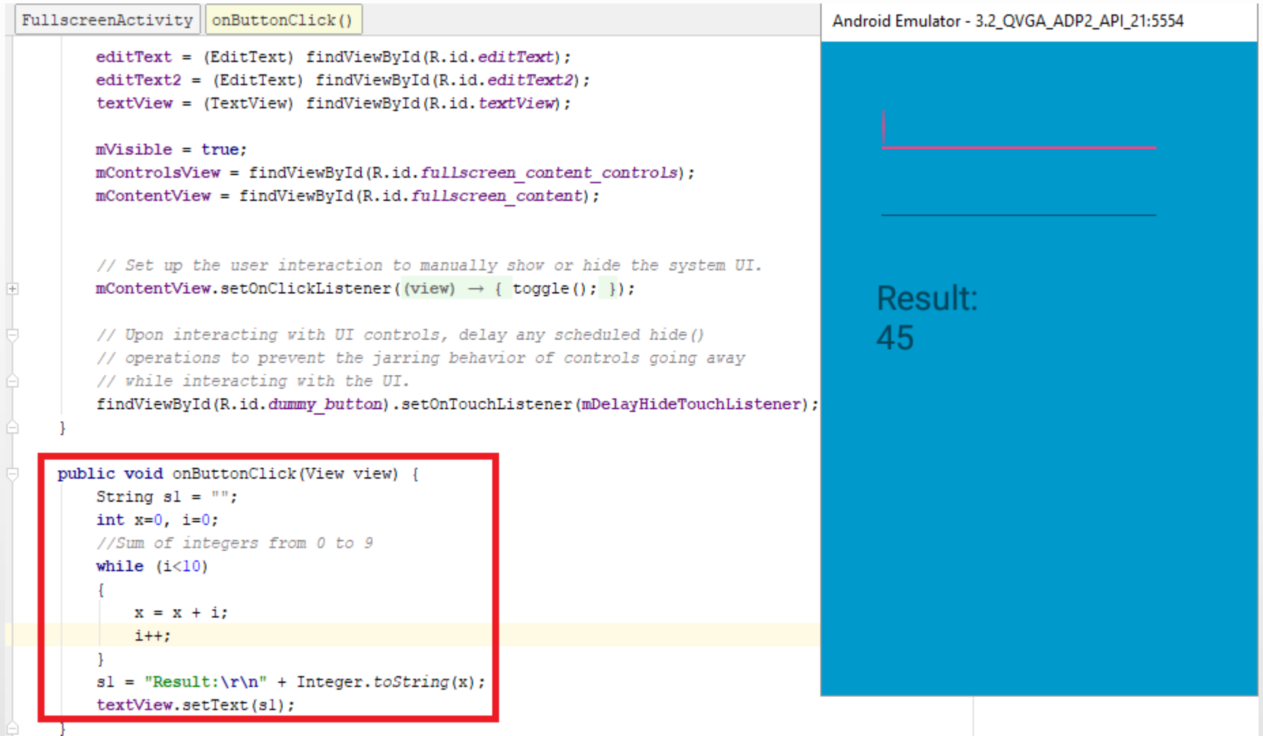
```
Result:
45
27
450
```

The questions:
1. How can we use several control variables in one for-loop?
2. How can we code the nested for-loop?

## Loop statements: while statement

- The ***while*** *statement* repeatedly executes another statement while its Boolean expression evaluates to true:

```
while (Boolean expression)
      statement
```



FullscreenActivity | onButtonClick()

```
        editText = (EditText) findViewById(R.id.editText);
        editText2 = (EditText) findViewById(R.id.editText2);
        textView = (TextView) findViewById(R.id.textView);

        mVisible = true;
        mControlsView = findViewById(R.id.fullscreen_content_controls);
        mContentView = findViewById(R.id.fullscreen_content);


        // Set up the user interaction to manually show or hide the system UI.
        mContentView.setOnClickListener((view) → { toggle(); });

        // Upon interacting with UI controls, delay any scheduled hide()
        // operations to prevent the jarring behavior of controls going away
        // while interacting with the UI.
        findViewById(R.id.dummy_button).setOnTouchListener(mDelayHideTouchListener);
    }

    public void onButtonClick(View view) {
        String s1 = "";
        int x=0, i=0;
        //Sum of integers from 0 to 9
        while (i<10)
        {
            x = x + i;
            i++;
        }
        s1 = "Result:\r\n" + Integer.toString(x);
        textView.setText(s1);
    }
```

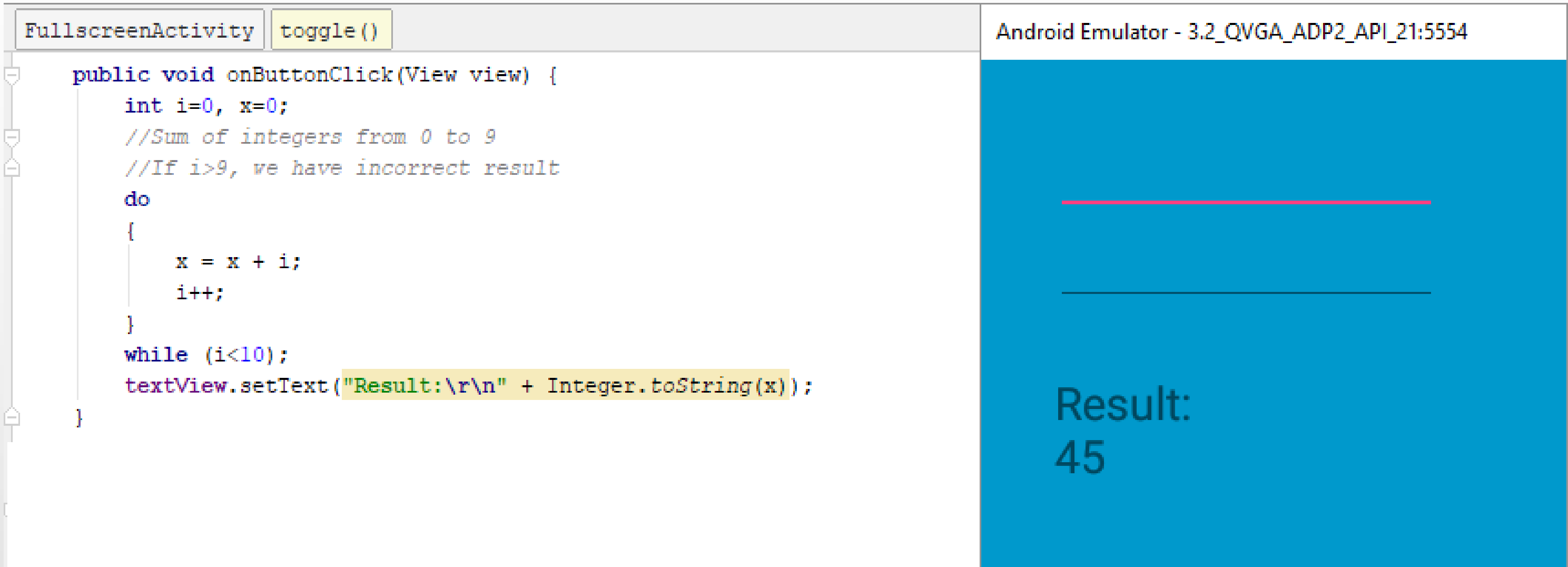
Android Emulator - 3.2_QVGA_ADP2_API_21:5554

```
Result:
45
```

## Loop statements: do-while statement

- The *do-while statement* repeatedly executes a statement while its Boolean expression evaluates to true:

```
do
      statement
while (Boolean expression);
```



FullscreenActivity | toggle()

```
public void onButtonClick(View view) {
    int i=0, x=0;
    //Sum of integers from 0 to 9
    //If i>9, we have incorrect result
    do
    {
        x = x + i;
        i++;
    }
    while (i<10);
    textView.setText("Result:\r\n" + Integer.toString(x));
}
```

Android Emulator - 3.2_QVGA_ADP2_API_21:5554

```
Result:
45
```

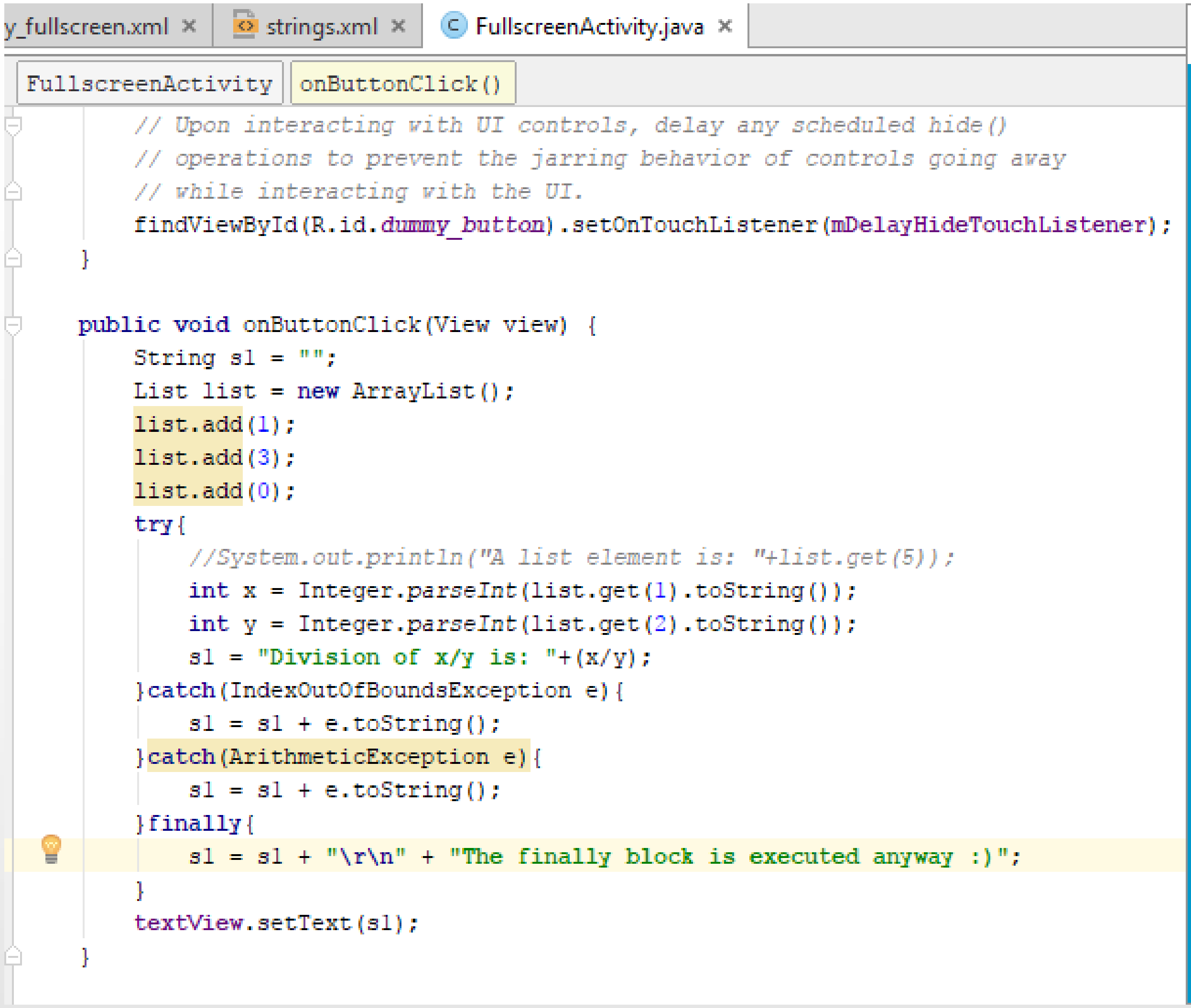
## Try-catch exception handler

- ***try-catch*** block of code is called exception handler. Also, the exception handler may optionally include the *finally* block.
- *More than one exceptions may arise during the execution of a block of code;* thus, more than one catch blocks can be associated with a single *try*:

```
try {}
catch (TypeOfException e) {}
catch (TypeOfException e) {}
. . . .
finally block
```

- The *finally* block is optional. The *finally* block encloses a block of code that will be definitely executed regardless of the thrown exception.

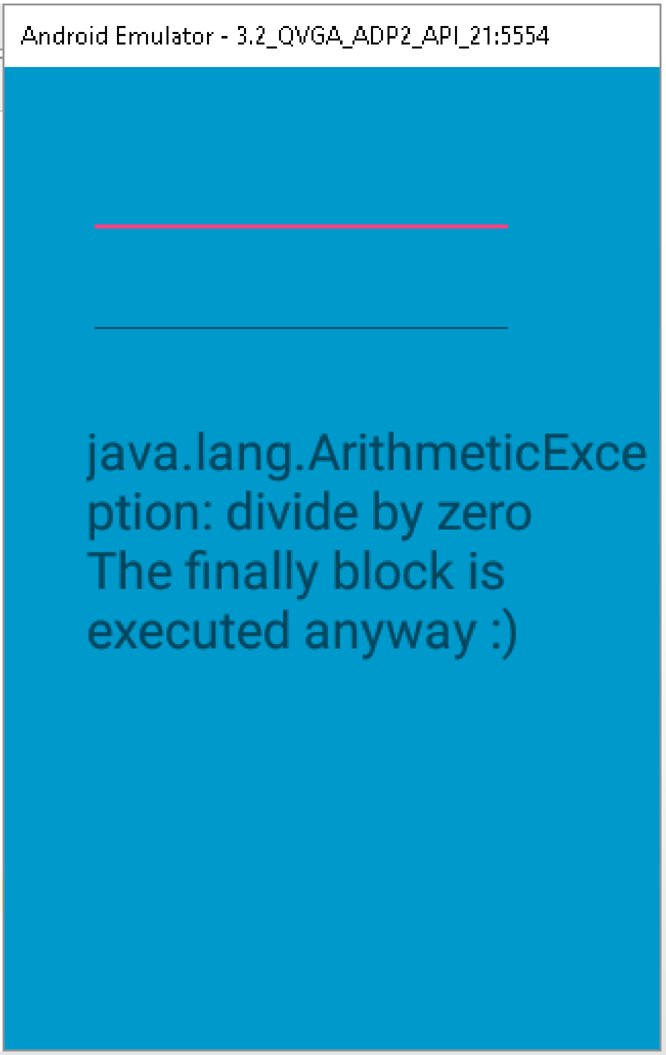
# Try-catch exception handler (cont.)



y_fullscreen.xml × strings.xml × FullscreenActivity.java ×

FullscreenActivity onButtonClick()

```
        // Upon interacting with UI controls, delay any scheduled hide()
        // operations to prevent the jarring behavior of controls going away
        // while interacting with the UI.
        findViewById(R.id.dummy_button).setOnTouchListener(mDelayHideTouchListener);
    }

    public void onButtonClick(View view) {
        String s1 = "";
        List list = new ArrayList();
        list.add(1);
        list.add(3);
        list.add(0);
        try{
            //System.out.println("A list element is: "+list.get(5));
            int x = Integer.parseInt(list.get(1).toString());
            int y = Integer.parseInt(list.get(2).toString());
            s1 = "Division of x/y is: "+(x/y);
        }catch(IndexOutOfBoundsException e){
            s1 = s1 + e.toString();
        }catch(ArithmeticException e){
            s1 = s1 + e.toString();
        }finally{
            s1 = s1 + "\r\n" + "The finally block is executed anyway :)";
        }
        textView.setText(s1);
    }
```

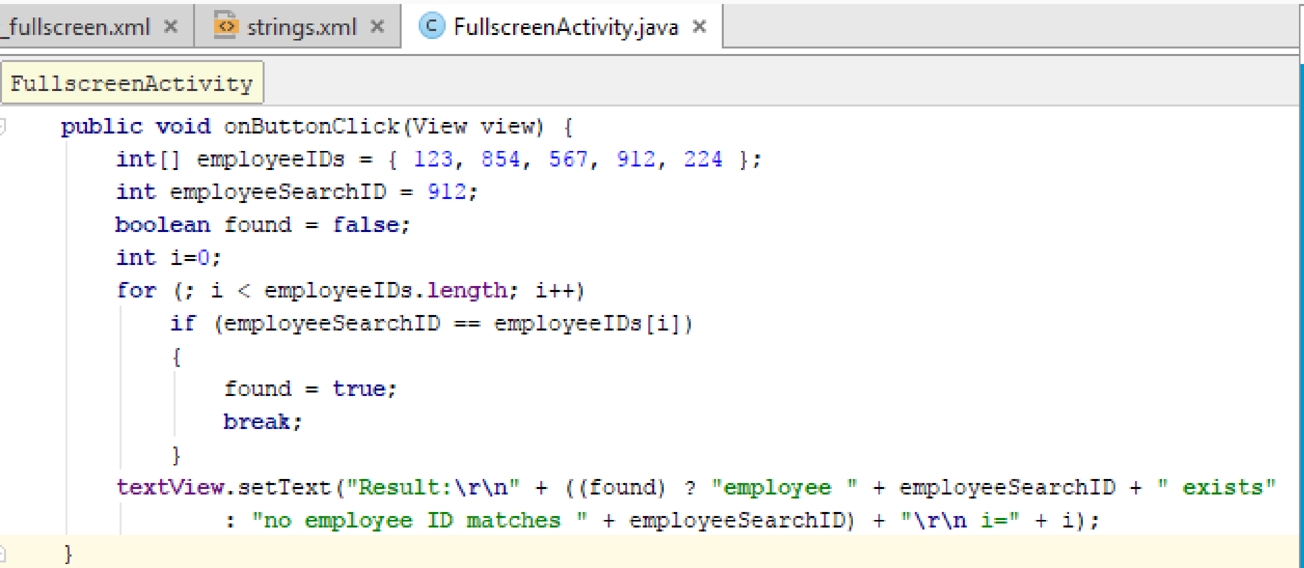


Android Emulator - 3.2_QVGA_ADP2_API_21:5554

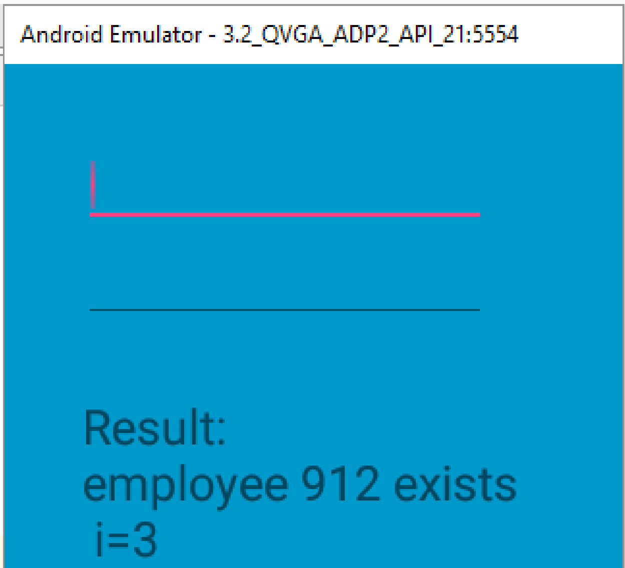
```
java.lang.ArithmeticExce
ption: divide by zero
The finally block is
executed anyway :)
```

## Break statement and Labeled Break statement

- The ***break*** statement transfers execution to the first statement following a *switch* statement or a loop. This statement consists of reserved word *break* followed by a semicolon.
  - The *break* statement is also useful in the context of a finite loop. For example, consider a scenario where an array of values is searched for a specific value, and we want to exit the loop when this value is found.



```
public void onButtonClick(View view) {
    int[] employeeIDs = { 123, 854, 567, 912, 224 };
    int employeeSearchID = 912;
    boolean found = false;
    int i=0;
    for (; i < employeeIDs.length; i++)
        if (employeeSearchID == employeeIDs[i])
        {
            found = true;
            break;
        }
    textView.setText("Result:\r\n" + ((found) ? "employee " + employeeSearchID + " exists"
            : "no employee ID matches " + employeeSearchID) + "\r\n i=" + i);
}
```



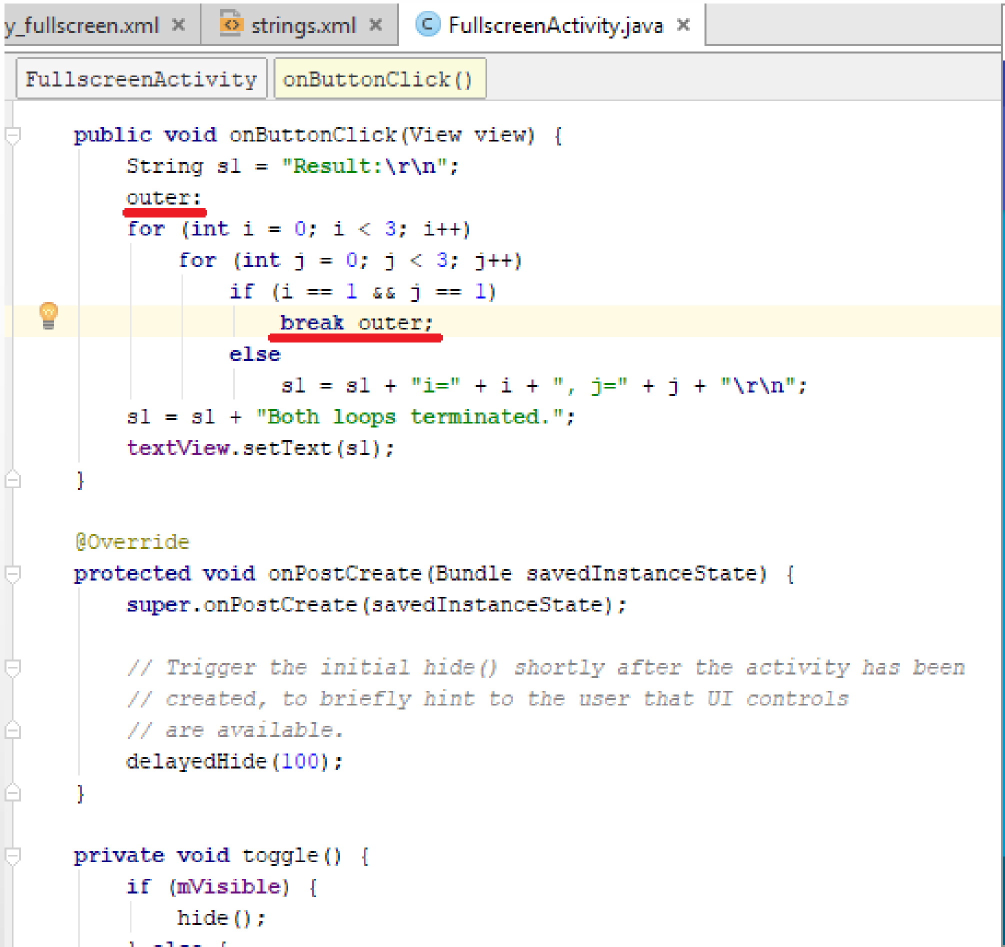
```
Result:
employee 912 exists
 i=3
```

## Break statement and Labeled Break statement (cont.)

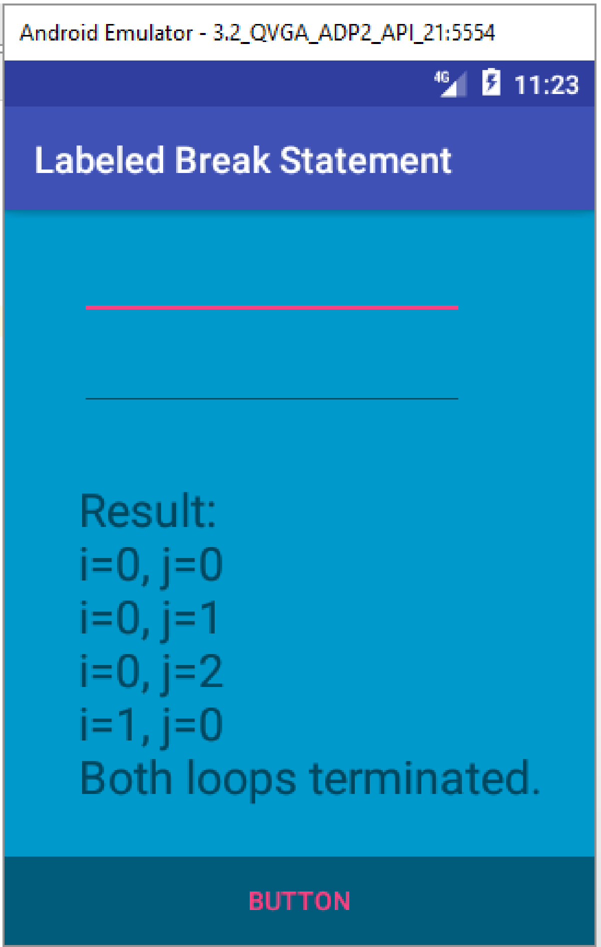
- The **labeled break** statement transfers execution to the first statement following the loop that's prefixed by a label (an identifier followed by a colon). It consists of reserved word *break*, followed by an identifier for which the matching label must exist. Furthermore, the label must immediately precede a loop statement.
  - The labeled break is useful for breaking out of nested loops. The following example reveals the labeled break statement transferring execution to the first statement that follows the **outer** for loop:

## Break statement and Labeled Break statement (cont.)

- An example of **labeled break** statement



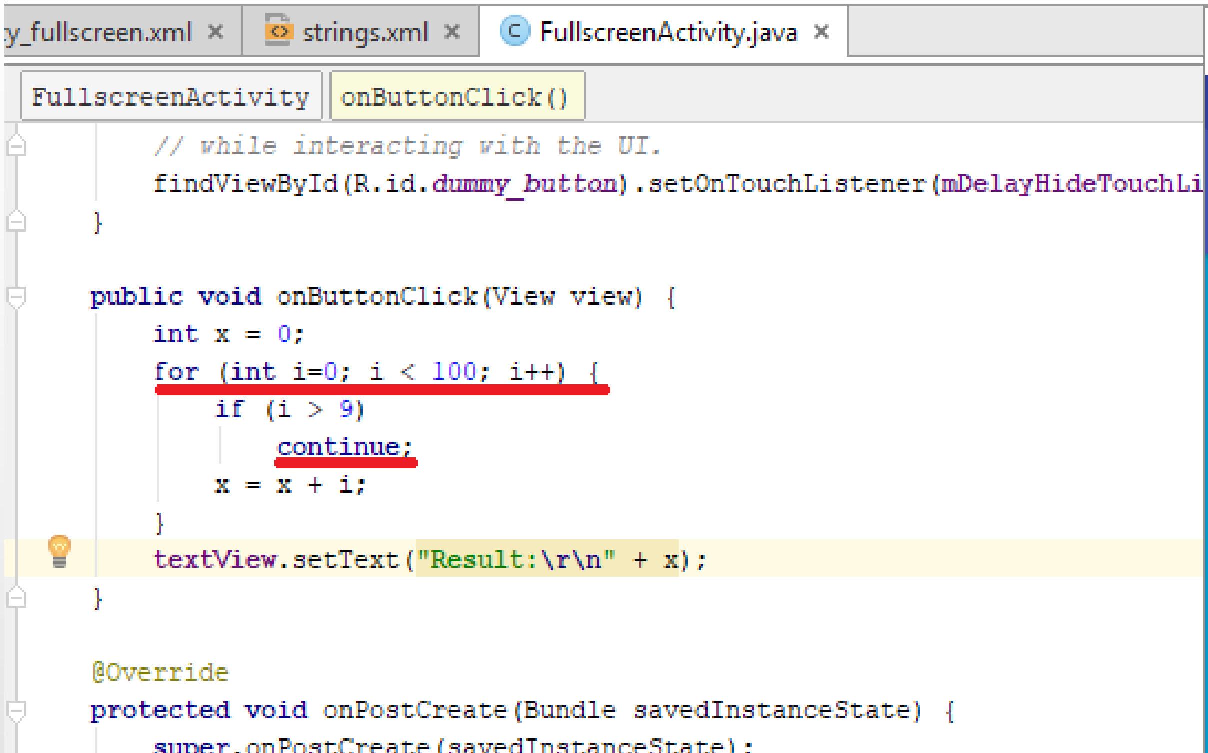
```
public void onButtonClick(View view) {
    String s1 = "Result:\r\n";
    outer:
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (i == 1 && j == 1)
                break outer;
            else
                s1 = s1 + "i=" + i + ", j=" + j + "\r\n";
    s1 = s1 + "Both loops terminated.";
    textView.setText(s1);
}

@Override
protected void onPostCreate(Bundle savedInstanceState) {
    super.onPostCreate(savedInstanceState);

    // Trigger the initial hide() shortly after the activity has been
    // created, to briefly hint to the user that UI controls
    // are available.
    delayedHide(100);
}

private void toggle() {
    if (mVisible) {
        hide();
```



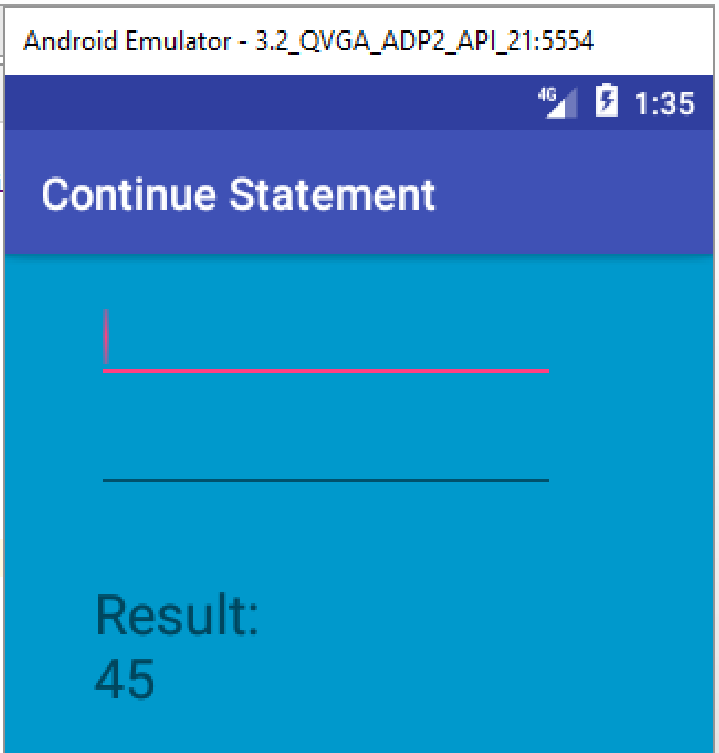
```
Result:
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
Both loops terminated.
```

# Continue statement

- The ***continue* statement** skips the remainder of the current loop iteration, re-evaluates the loop's Boolean expression, and performs another iteration (if true) or terminates the loop (if false). *Continue* consists of reserved word *continue* followed by a semicolon.



```
        // while interacting with the UI.
        findViewById(R.id.dummy_button).setOnTouchListener(mDelayHideTouchLi
    }

    public void onButtonClick(View view) {
        int x = 0;
        for (int i=0; i < 100; i++) {
            if (i > 9)
                continue;
            x = x + i;
        }
        textView.setText("Result:\r\n" + x);
    }

    @Override
    protected void onPostCreate(Bundle savedInstanceState) {
        super.onPostCreate(savedInstanceState);
```



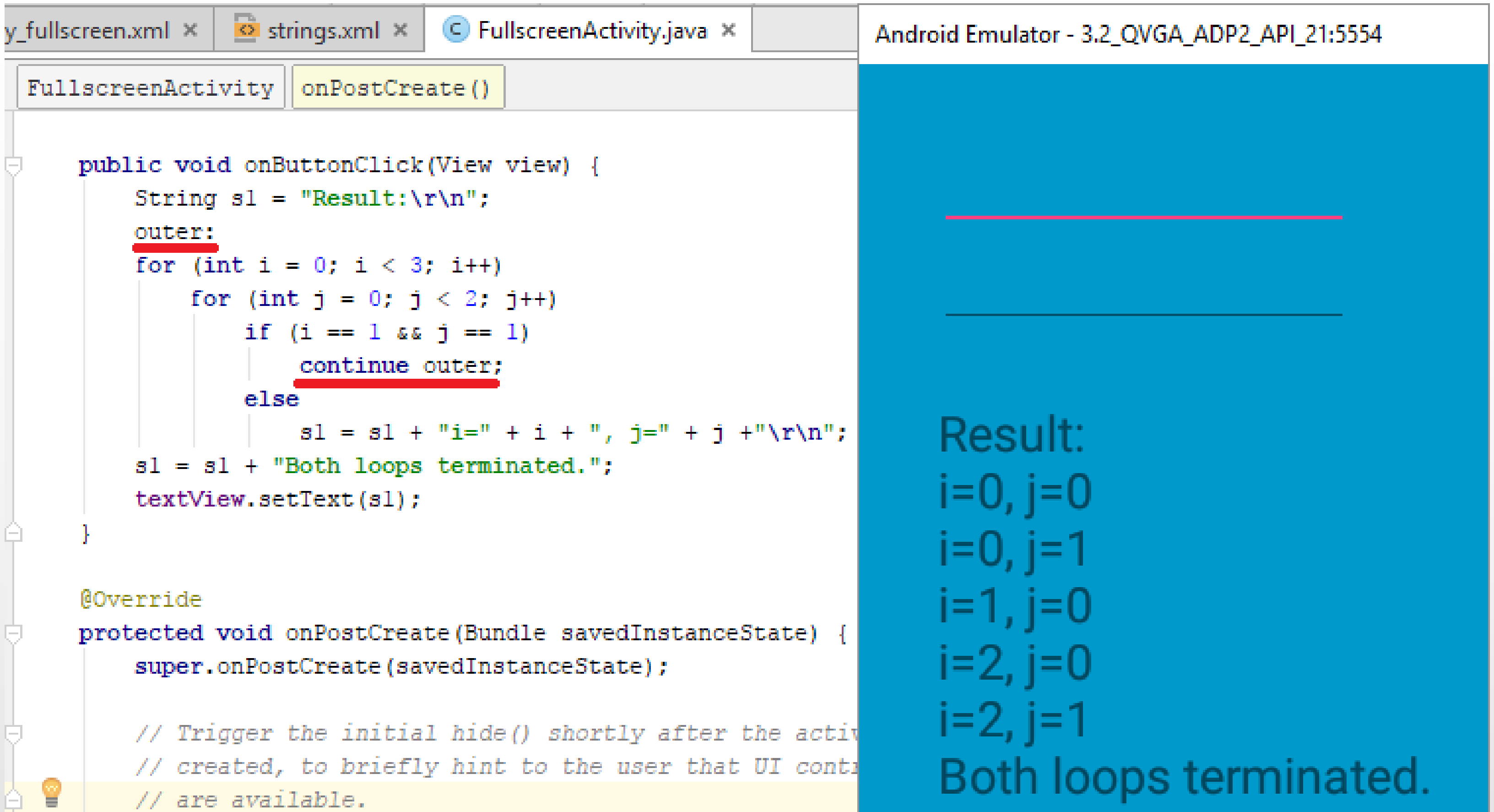
```
Result:
45
```

## Labeled Continue statement

- The **labeled continue** statement skips the remaining iterations of one or more nested loops and transfers execution to the labeled loop. It consists of reserved word *continue*, followed by an identifier for which a matching label must exist. Furthermore, the label must immediately precede a loop statement.

## Labeled Continue statement

- Labeled continue is useful for breaking out of nested loops while still continuing to execute the labeled loop



```
public void onButtonClick(View view) {
    String s1 = "Result:\r\n";
    outer:
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 2; j++)
            if (i == 1 && j == 1)
                continue outer;
            else
                s1 = s1 + "i=" + i + ", j=" + j +"\r\n";
    s1 = s1 + "Both loops terminated.";
    textView.setText(s1);
}

@Override
protected void onPostCreate(Bundle savedInstanceState) {
    super.onPostCreate(savedInstanceState);

    // Trigger the initial hide() shortly after the activ
    // created, to briefly hint to the user that UI contr
    // are available.
```

```
Result:
i=0, j=0
i=0, j=1
i=1, j=0
i=2, j=0
i=2, j=1
Both loops terminated.
```

Do you have any questions or comments?

Thank you

for your attention !

In this presentation:

• Some icons were downloaded from flaticon.com and iconscout.com