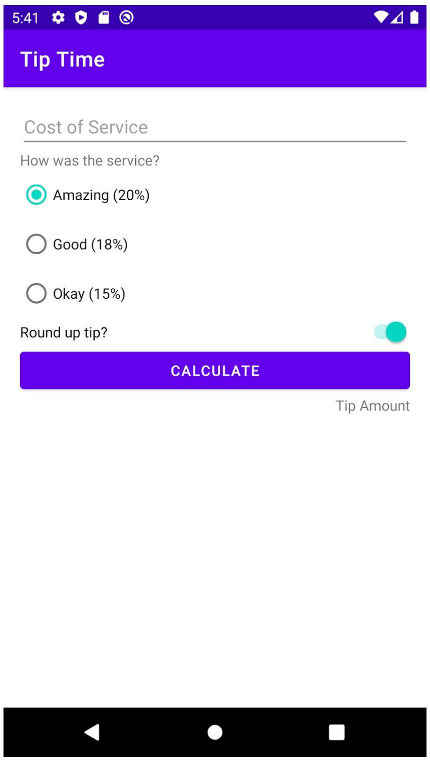
CP3406_CP5307 Codelab 3.2: Coding the Tip Calculator App

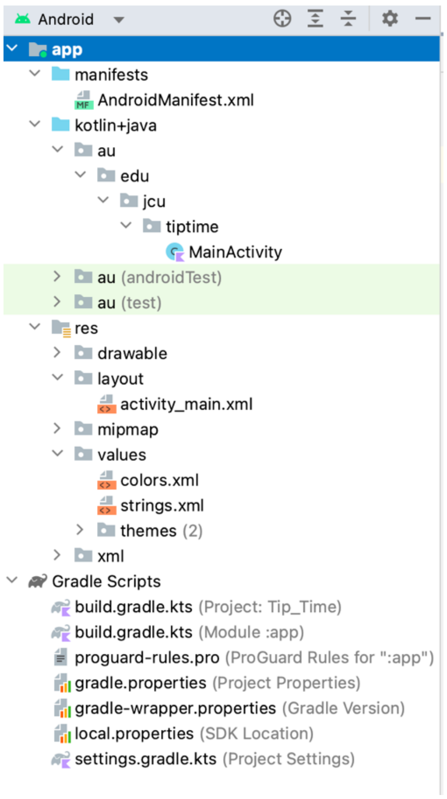
Contents



1. Overview

In this codelab, you will write code so that the tip calculator functions with the UI you created in the previous codelab. So far, we have built a UI for the **Tip Time** app. A **Cost of Service** `EditText` allows the user to enter the cost of the service. A list of `RadioButtons` allows the user to select the tip percentage, and a `Switch` allows the user to choose whether the tip should be rounded up or not. The amount of the tip is shown in a `TextView`. Finally, there is a `Button` that doesn't do anything, but should tell the app to get the data from the other fields and calculate the tip. In this codelab, we'll address that feature.

Structure of the app project



```
Android
app
  manifests
    AndroidManifest.xml
  kotlin+java
    au
      edu
        jcu
          tiptime
            MainActivity
    au (androidTest)
    au (test)
  res
    drawable
    layout
      activity_main.xml
    mipmap
    values
      colors.xml
      strings.xml
      themes (2)
    xml
Gradle Scripts
  build.gradle.kts (Project: Tip_Time)
  build.gradle.kts (Module :app)
  proguard-rules.pro (ProGuard Rules for ":app")
  gradle.properties (Project Properties)
  gradle-wrapper.properties (Gradle Version)
  local.properties (SDK Location)
  settings.gradle.kts (Project Settings)
```

In your IDE, an app project consists of several parts, including the Kotlin code, XML designs and other resources such as images and stromgs. Before making changes to the app, we recommend you learn how it works.

1. Opens the **Tip Time** project on Android Studio.
2. If the **Project** window is not displayed, select the **Project** tab on the left side of Android Studio.
3. If you haven't selected yet, choose the **Android** view from the drop-down menu and take a look at the folders/files. You have

- The **kotlin+java** folder for Kotlin files (or Java files)
- `MainActivity`: Class in which the entire Kotlin code will go for the logic of the tip calculator
- **res** folder for app resources
- `activity_main.xml`: design of Your Android app
- `strings.xml`: Contains string resources from your Android app
- **Gradle Scripts** folder

Gradle is the automated build system Android Studio uses. Every time you change the code, add a resource or make other changes to your app, Gradle determines what's changed and take the necessary steps to re-compile your app. Also, it helps to install your app on the emulator or on a physical device, and control its execution.

There are other folders and other files that are part of the compilation of your app, but these are the main ones you will work with in this codelab and the following codelabs.

Note: As you may already know, you can also develop Android apps into the Java programming language.

2. View Binding

To calculate the tip, your code must access all elements of the UI to read the user input. You may remember from previous codeblabs that your code should find a reference to an element `View`, for example, `Button` or `TextView` before you can call the methods in the element `View` or access their attributes. The Android framework provides a method, `findViewById()`, which does exactly what you need; given the ID of an element `View` it returns a reference to it. This approach works, but as youadd more views to your app and the UI becomes more complex, the use of `findViewById()` can become more cumbersome.

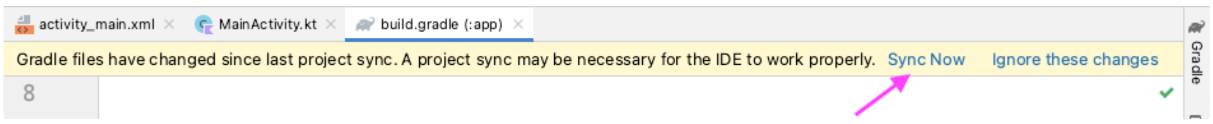
For reasons of convenience, Android also provides a function called *view binding*. With a little more work at first, view binding makes it much easier and faster to call methods on the views of your UI. To use view binding, you will need to enable it in Gradle and make some code changes.

How to enable view binding

4. Opens the file `build.gradle.kts` from the app (**Gradle Scripts > build.gradle.kts (Module: app)**)
5. Within the braces of the section `android`, adds the following lines:

```
buildFeatures {
    viewBinding = true
}
```

6. Please note the message **Gradle files have changed since last project sync.**
7. Press **Sync Now**.

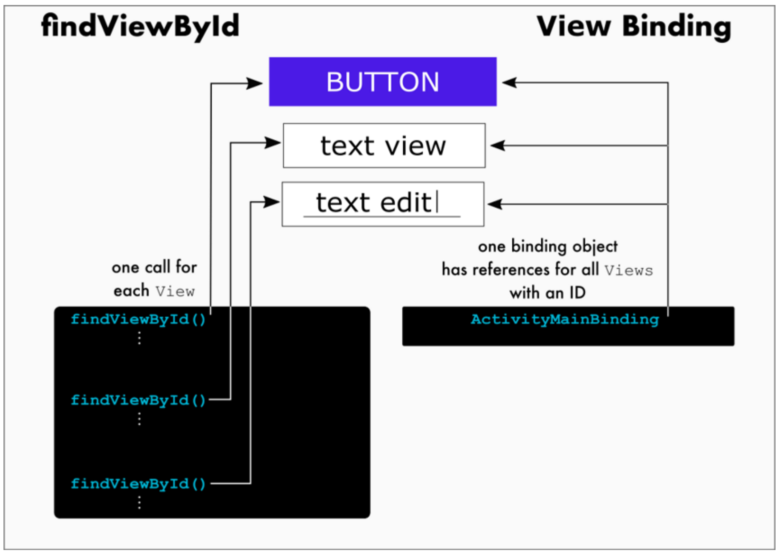


Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. Sync Now Ignore these changes

After a few moments, you should see a message at the bottom of the Android Studio window: **Gradle sync finished**. You can now close the file `build.gradle.kts`.

How to initialize a binding object

In previous codelabs, we explored the `onCreate()` method in the class `MainActivity`. It is one of the first things to be called when your app is started and the `MainActivity` is initialized. Instead of calling `findViewById()` for each `View` in your app, we will use a single binding object.



1. Open up. `MainActivity.kt`.
2. Replace the existing code for the class `MainActivity` with the following code to setup the class `MainActivity` to use view binding:

```
class MainActivity : AppCompatActivity() {
    lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

3. The following line declares a class level variable. It is defined at this level because it willbe used in various class methods in `MainActivity`.

```
lateinit var binding: ActivityMainBinding
```

The keyword `lateinit` is something new. It's a promise that your code will initialize the variable before using it. Otherwise, your app will fail.

4. This line creates (inflates) the object `binding` that you will use to access `Views` from the design `activity_main.xml`.

```
binding = ActivityMainBinding.inflate(layoutInflater)
```

5. Now to set the content view of the activity, instead of passing the layout resource ID, `R.layout.activity_main`, we specify the root of the view hierarchy in your app, `binding.root`.

```
setContentView(binding.root)
```

You may remember the idea of the parent views and child views; the root connects to all of these.

Now, when you need a reference to an element `View` in your app, you can get it from the object `binding` instead of calling `findViewById()`. The object `binding` automatically defines references for each `View` in your app that has an ID. The use of view binding is so concise that you often won't need to even create a variable to reference a `View`; you can just use it directly from the binding object.

```
// Old way with findViewById()
val myButton: Button = findViewById(R.id.my_button)
myButton.text = "A button"
// Better way with view binding
val myButton: Button = binding.myButton
myButton.text = "A button"
// Best way with view binding and no extra variable
binding.myButton.text = "A button"
```

It's great, isn't it?

Note: The name of the view binding class is generated by converting the name of the XML file according to the initial capital letter convention (PascalCase), and adding the word "Binding" at the end. Similarly, the reference for each view is generated by removing the underscores and converting the name of the view to the camelCase. For example, according to the initial capital letter convention, `activity_main.xml` becomes `ActivityMainBinding` and you can access `@id/text_view` like `binding.textView`.

3. How to calculate the tip

The process to calculate the tip begins when the user presses the **Calculate** button. This process includes checking the UI to determine the cost of the service and the tip percentage that the user wants to leave. With this information, you will calculate the full amount of the service charge, and the amount of the tip will be displayed.

How to add a click listener to the button

The first step involves adding a click listener object to specify what the **Calculate** button should do when the user presses it.

1. In `MainActivity.kt`, in `onCreate()`, after the call to `setContentView()`, configure a click listener on the **Calculate** button and have it call a method `calculateTip()`.

```
binding.calculateButton.setOnClickListener{ calculateTip() }
```

2. Still in class `MainActivity`, but outside the object `onCreate()`, add a new function called `calculateTip()`.

```
fun calculateTip() {
}
```

Here you will add the code to get the user inputs from the UI and calculate the tip.

`MainActivity.kt`

```
class MainActivity : AppCompatActivity() {
     lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        binding.calculateButton.setOnClickListener{ calculateTip() }
    }
    fun calculateTip() {
    }
}
```

How to get the cost of the service

To calculate the tip, the first thing you need is the cost of the service. The text is stored in the `EditText`, but you'll need it as a number so you that can use it in the calculations. You may remember the type `Int` from ther codelabs, but an element `Int` can only contain integers. To use a decimal number in your app, use the data type `Double` instead of `Int`. You can read more about the types of numerical data in Kotlin. Kotlin provides a method for converting an object `String` to `Double`, which is called `toDouble()`.

1. First, get the text for the cost of the service. In the method `calculateTip()`, get the **Cost of Service** text attribute `EditText` and assign it to a variable called `stringInTextField`. Remember, you can access the UI element from `binding` according to its resource ID name in the camelCase format.

```
val stringInTextField = binding.costOfService.text
```

Note the element `.text` at the end. The first part, `binding.costOfService` refers to the UI element for the cost of the service. The `.text` at the end indicates to take the object `EditText` and obtain the property `text` from it. This is known as *chaining* and is a very common pattern in Kotlin.

2. Next, convert the text into a decimal number. Call the method `toDouble()` on `stringInTextField` and store it in a variable called `cost`.

```
val cost = stringInTextField.toDouble()
```

Except, that doesn't seem to work. You should call `toDouble()` on an object of type `String`. Turns out the attribute `text` of an `EditText` is an `Editable`, because it represents a type of text that can be changed. Fortunately, you can easily convert an `Editable` to `String` if you call `toString()`.

3. Call the method `toString()` on `binding.costOfService.text` to make it a `String`:

```
val stringInTextField = binding.costOfService.text.toString()
```

Now, `stringInTextField.toDouble()` will work.

At this point, the method `calculateTip()` should contain the following code:

```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDouble()
}
```

How to get the tip percentage

So far, you have the cost of the service. Now, you need the tip percentage, which the user selected from a. `RadioGroup` of `RadioButtons`.

1. In `calculateTip()`, get the attribute `checkedRadioButtonId` of `tipOptionsRadioGroup` and store in a variable called `selectedId`.

```
val selectedId = binding.tipOptions.checkedRadioButtonId
```

Now we know that the selected `RadioButton` is one of `R.id.option_twenty_percent`, `R.id.option_eighteen_percent` or `R.id.fifteen_percent`, but we still need the corresponding percentage. You could write a series of `if/else` statements, but it's much easier to use the `when` expression.

2. Add the following lines to get the tip percentage.

```
val tipPercentage = when (selectedId) {
    R.id.option_twenty_percent -> 0.20
    R.id.option_eighteen_percent -> 0.18
    else -> 0.15
}
```

At this point, the method `calculateTip()` should look as follows:

```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDouble()

    val  selectedId  =  binding.tipOptions.checkedRadioButtonId
    val tipPercentage = when  (selectedId)  {
        R.id.option_twenty_percent  ->  0.20
        R.id.option_eighteen_percent  ->  0.18
        else  ->  0.15
    }
}
```

How to calculate tip and round it up

Now that you have the cost of the service and the percentage to tip, calculating the final tip is simple: the tip = cost of the service * percentage tip. That value may also be rounded up if the user selects that option.

1. In `calculateTip()` after the other code you added, multiply `tipPercentage` by `cost` and assign the result to a variable called `tip`.

```
var tip = tipPercentage * cost
```

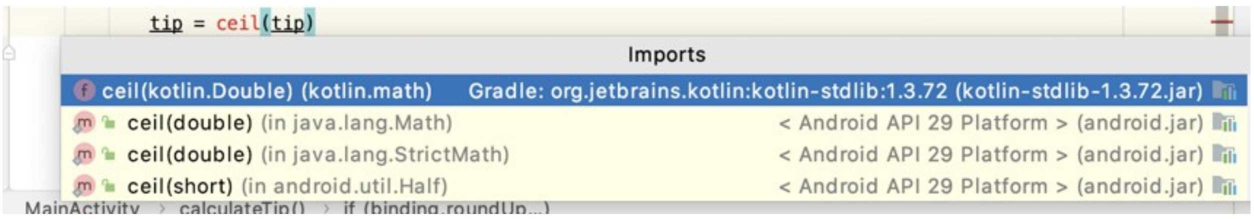
Please note that we used `var` instead of `val`, since you might have to round the value if the user selected that option, so this value could potentially change.

In the case of the `Switch`, you can use the attribute `isChecked` to see if the switch is "activated."

2. Assign the attribute `isChecked` to a variable named `roundUp`.

```
val  roundUp  =  binding.roundUpSwitch.isChecked
```

The term "rounding" involves adjusting a decimal number up or down to the nearest integer value; but, in this case, you just want to round up or reach the limit. You can use the function `ceil()` to do it. There are several functions with that name, but the one you need is defined in `kotlin.math`. You could add a an `import`, but, in this case, it's easier just to inform Android Studio what your intention is with the use of `kotlin.math.ceil()`.



```
tip = ceil(tip)
Imports
ceil(kotlin.Double) (kotlin.math)    Gradle: org.jetbrains.kotlin:kotlin-stdlib:1.3.72 (kotlin-stdlib-1.3.72.jar)
ceil(double) (in java.lang.Math)                   < Android API 29 Platform > (android.jar)
ceil(double) (in java.lang.StrictMath)             < Android API 29 Platform > (android.jar)
ceil(short) (in android.util.Half)                 < Android API 29 Platform > (android.jar)
MainActivity  >  calculateTip()  >  if (binding.roundUp...)
```

If you wanted to use multiple math functions, it would be easier to add an `import`.

3. Add an `if` statement which assigns the rounded up value to `tip` if `roundup` is true.

```
if  (roundUp)  {
    tip  =  kotlin.math.ceil(tip)
}
```

At this point, the method `calculateTip()` should look as follows:

```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDouble()
    val selectedId = binding.tipOptions.checkedRadioButtonId
    val tipPercentage = when (selectedId) {
        R.id.option_twenty_percent -> 0.20
        R.id.option_eighteen_percent -> 0.18
        else -> 0.15
    }
    var tip = tipPercentage * cost
    val roundUp = binding.roundUpSwitch.isChecked
    if (roundUp) {
        tip = kotlin.math.ceil(tip)
    }
}
```

How to format the tip

Your app is almost working. You calculated the tip; now, you just have to format it and show it.

As you might expect, Kotlin provides methods to format different types of numbers. However, the amount of the tip is a little different: it represents a currency value. Different countries use different currencies and have different rules to format decimal numbers. For example, in US dollars, 1234.56 would have the format $1,234.56, but in euros, it would be 1,234.56. Fortunately, the Android framework provides methods togive currency formats to the numbers so you don't need to know or remember all the possibilities. The system applies thecurrency format automatically according to the language and other configurations that the user has chosen on the device. Learn more about NumberFormat in the Android documentation.

1. In `calculateTip()` after your other code, call `NumberFormat.getCurrencyInstance()`.

```
NumberFormat.getCurrencyInstance()
```

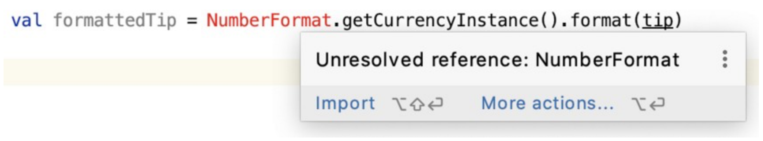
This way, you'll get a number formatter that you can use to apply the currency format to numbers.

2. With the number formatter, you can chain a call to the method `format()` with `tip` and assigns the result to a variable called `formattedTip`.

```
val formattedTip = NumberFormat.getCurrencyInstance().format(tip)
```

3. Please note that `NumberFormat` is displaying an error, as Android Studio can't automatically determine which version of `NumberFormat` you're referring to.

4. Place the cursor over `NumberFormat`and choose **Import** in the pop-up window that appears.



```
val formattedTip = NumberFormat.getCurrencyInstance().format(tip)
```

Unresolved reference: NumberFormat

5. From the list of possible imports, choose **NumberFormat (java.text)**. Android Studio adds a an `import` at the top of the file `MainActivity`, and the error is gone.

How to display the tip

Now, it's time to display the tip in the `TextView`. You could assign `formattedTip` to the attribute `text`, but it would also be good to label what the amount stands for. In U.S. or Australian English, the tip amount could be displayed: **$12.34** (the $ at the beginning), but in other languages it is possible that the number should appear at the beginning or even in the middle of the text with the currency symbol being displayed at the end. We can prepare for translating our app into different languages by using a string template with a string parameter.

1. Open up. `strings.xml`. Change the string named `tip_amount` to `Tip Amount: %s`.

```
<string name="tip_amount">Tip Amount: %s</string>
```

The element `%s` is where the formatted currency will be inserted.

2. Now, configure the text of `tipResult`. Back in the method `calculateTip()` in `MainActivity.kt`, set the output text with the following method:

```
binding.tipResult.text = getString(R.string.tip_amount, formattedTip)
```

At this point, the method `calculateTip()` should look as follows:

```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDouble()
    val selectedId = binding.tipOptions.checkedRadioButtonId
    val tipPercentage = when (selectedId) {
        R.id.option_twenty_percent -> 0.20
        R.id.option_eighteen_percent -> 0.18
        else -> 0.15
    }
    var tip = tipPercentage * cost
    val roundUp = binding.roundUpSwitch.isChecked
    if (roundUp) {
        tip = kotlin.math.ceil(tip)
    }
    val formattedTip = NumberFormat.getCurrencyInstance().format(tip)
    binding.tipResult.text = getString(R.string.tip_amount, formattedTip)
}
```

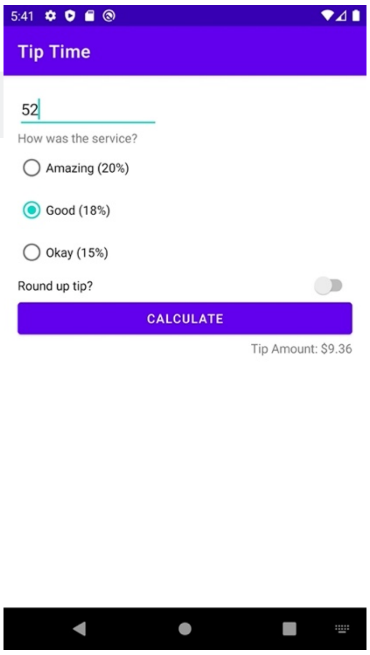
You're almost done. When you develop your app (look at the preview in design view), it's useful to have a default value displayed for that element `TextView`.

3. Open up. `activity_main.xml`
4. Look for the element `tip_resultTextView`.
5. Remove the line with the attribute `android:text`.

```
android:text="@string/tip_amount"
```

6. Add a line for the attribute `tools:text` that is `Tip Amount: $10`.

```
tools:text="Tip Amount: $10"
```



Since it's only displayed to the developer in Android Studio, you can provide a hardcoded value. The user won't see this.

7. Verify that the text of the tools appears in the **design editor**.
8. Run your app. Enter an amount for the cost and select some options. Then press the **Calculate** button.

Congratulations. It works. If you don't get the right amount for the tip, go back to step 1 of this section and check that you've made all the necessary changes to the code.

4. Debugging with Logcat

Now it's time to do more tests.

Think about how the information moves through the app in the method `calculateTip()` and what could go wrong at each step.

For example, what could happen in this line:

```
val cost = stringInTextField.toDouble()
```

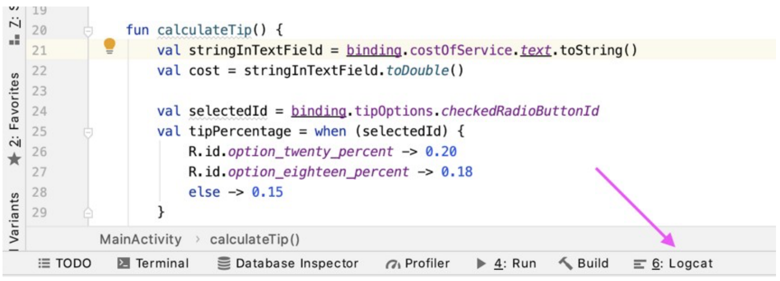
What if `stringInTextField` has text that doesn't represent a number? What would happen if the user did not enter any text at all and the element `stringInTextField` was empty?

1. Run your app on the emulator, but instead of using **Run > Run 'app'**, use **Run > Debug 'app'**.
2. Try different cost combinations, tip amounts and round up or not, and verify that the expected results are displayed for each case when you press **Calculate**.
3. Now, try to erase the entire text from the **Cost of Service** field and press **Calculate**. Oh, wow. The program fails.

How to fix the fault

The first step in addressing a mistake involves figuring out what happened. Android Studio keeps track of what's going on in the system; you can use it to detect errors.

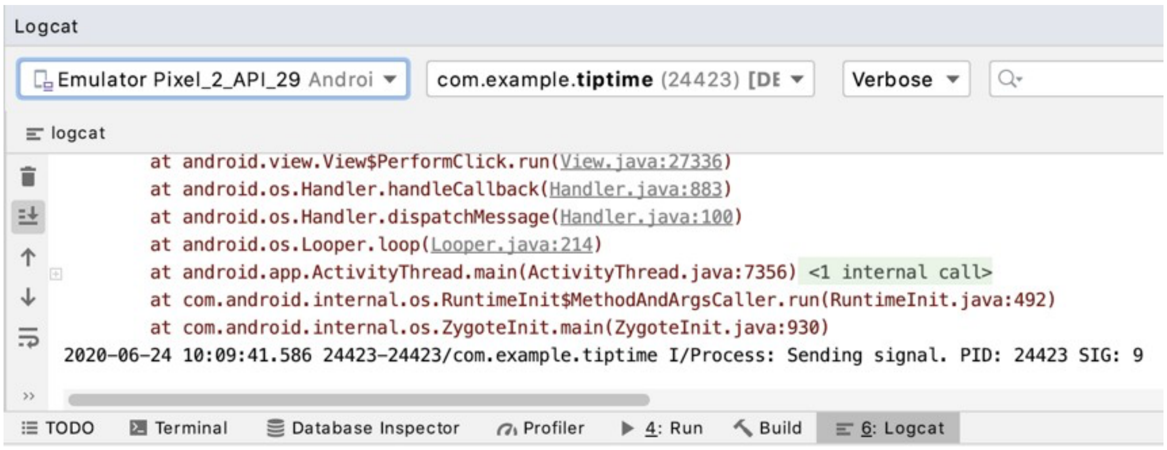
1. Press the **Logcat** button at the bottom of Android Studio or select **View > Tool Windows > Logcat** from menus.



```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDouble()

    val selectedId = binding.tipOptions.checkedRadioButtonId
    val tipPercentage = when (selectedId) {
        R.id.option_twenty_percent -> 0.20
        R.id.option_eighteen_percent -> 0.18
        else -> 0.15
    }
```

2. The **Logcat** window appears at the bottom of Android Studio, with some odd-looking text.



```
        at android.view.View$PerformClick.run(View.java:27336)
        at android.os.Handler.handleCallback(Handler.java:883)
        at android.os.Handler.dispatchMessage(Handler.java:100)
        at android.os.Looper.loop(Looper.java:214)
        at android.app.ActivityThread.main(ActivityThread.java:7356) <1 internal call>
        at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:492)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:930)
2020-06-24 10:09:41.586 24423-24423/com.example.tiptime I/Process: Sending signal. PID: 24423 SIG: 9
```

The text is a *stack trace*, a list of the methods called when the failure occurred.

3. Scroll up in the **Logcat** text until you find a line that includes the text `FATAL EXCEPTION`.

```
2020-06-24 10:09:41.564 24423-24423/com.example.tiptime E/AndroidRuntime:
FATAL EXCEPTION: main
    Process: com.example.tiptime, PID: 24423
    java.lang.NumberFormatException: empty String
        at
sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1842)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at
com.example.tiptime.MainActivity.calculateTip(MainActivity.kt:22)
        at
com.example.tiptime.MainActivity$onCreate$1.onClick(MainActivity.kt:17)
```

4. Read down until you find the line with `NumberFormatException`.

```
java.lang.NumberFormatException: empty String
```

To the right, it says. `empty String`. The exception type tells you that it was an error related to a number format, and the rest tells you the root of the problem: it found an empty `String` when it expected a `String` with a value.

5. Continue reading down and you'll see some calls to `parseDouble()`.
6. Underneath those calls, look for the line with `calculateTip`. Note that your class is also included: `MainActivity`.

```
at  com.example.tiptime.MainActivity.calculateTip(MainActivity.kt:22)
```

7. Pay attention to that line and you can see exactly where the code made the call: line 22 in `MainActivity.kt` (if you entered the code slighly differently, it could be a different number). That line is the one that converts the `String` to `Double` and assigns the result to the variable `cost`.

```
val cost = stringInTextField.toDouble()
```

8. See the Kotlin documentation for the method `toDouble()` that works on a `String`. The method is known as `String.toDouble()`.
9. The page reads "Exceptions: `NumberFormatException`- if the string is not a valid representation of a number."

An *exception* is how the system indicates that there is a problem. In this case, the problem is that `toDouble()` couldn't convert the empty `String` to a `Double`. Although `EditText` has the property `inputType=numberDecimal`, it is still possible to (not) enter some values that `toDouble()` can't process, like no value at all.

How to handle the empty value

It doesn't work to call `toDouble()` on an empty string or a string that does not represent a valid decimal number. Fortunately, Kotlin also provides a method called `toDoubleOrNull()` that handles this problem. It attempts to return a decimal number if possible, otherwise it returns `null` if it can’t.

Your app can verify whether it has obtained `null` from `toDoubleOrNull()` and do something different in that case to avoid the app from crashing.

1. In `calculateTip()`, change the line that declares the variable `cost` to call `toDoubleOrNull()` instead of calling `toDouble()`.

```
val cost = stringInTextField.toDoubleOrNull()
```

2. After that line, add a statement to check if `cost` is `null`, and if so, return from the method. The instruction `return` leaves the method without executing any further instructions.

```
if (cost == null) {
    return
}
```

3. Run your app again.
4. Don't enter any text in the **Cost of Service** field, and press **Calculate**. This time, the app doesn't fail. Well done. You found and you corrected the mistake.

How to control another case

Not all bugs may cause an app to crash. Consider situations in which the user may become confused by the information on the screen. For example, what will happen if the user performs the following?:

1. Enter a valid cost amount
2. Push **Calculate**
3. Clear the cost from the EditText
4. Push **Calculate** again

The first time, the tip will be calculated and shown as planned. The second time, the method will detect that there is no cost value added and return from `calculateTip()` early due to the validation you've just added. However, the app will continue to show the previously calculated tip. That could be confusing for the user, so let's add code in order to erase the tip amount if there is any problem.

1. To confirm this problem, introduce a valid cost and press **Calculate**. Then, delete the text and press **Calculate** again. The first value of the tip should continue to be displayed.
2. Inside the new validation code we added, before the `return`, add a line to set the attribute `Text` of `tipResult` to an empty string.

```
if (cost == null) {
    binding.tipResult.text = ""
    return
}
```

Now, the amount of the tip will be erased before it is returned from `calculateTip()`.

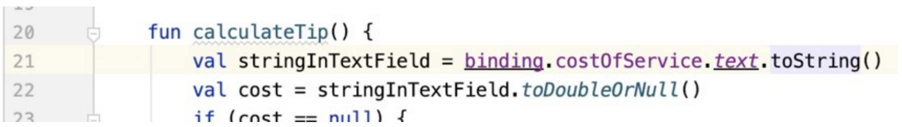
3. Run the app again and try the previous case. The first value of the tip should disappear when **you** press **Calculate** the second time.

Congratulations. You created an app to calculate tips for Android and test and handled some edge cases.

5. Implement coding best practices

Your tip calculator now works, but you can improve the code a little and help future maintainers of your code (including yourself) if you adopt some coding best practices.

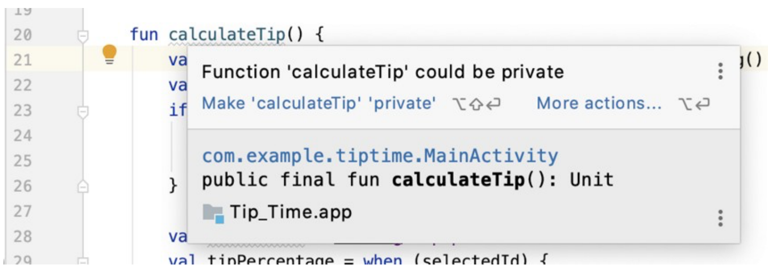
1. Open up. `MainActivity.kt`.
2. Observe the method `calculateTip()` and you might see that it is underlined with a grey wavy line.



```
fun calculateTip() {
    val stringInTextField = binding.costOfService.text.toString()
    val cost = stringInTextField.toDoubleOrNull()
    if (cost == null) {
```

3. Place the cursor over `calculateTip()` and you will see the message **Function 'calculateTip'**

could be private followed by the following suggestion: **Make 'calculateTip' private.**



```
fun calculateTip() {
    va
    va
    if
    }
    va
    val tipPercentage = when (selectedId) {
```

Function 'calculateTip' could be private

Make 'calculateTip' 'private' ⌥⇧↩   More actions... ⌥↩

```
com.example.tiptime.MainActivity
public final fun calculateTip(): Unit
```

Tip_Time.app

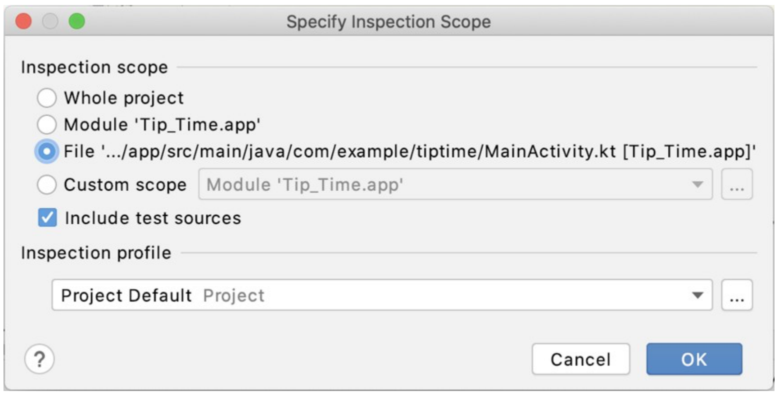
Remember from the previous codelabs that `private` indicates that the method or variable is only visible to the code within that class, in this case, `MainActivity`. There's no reason for this code to be used outside of `MainActivity`, so you can set it as `private`.

4. Choose **Make 'calculateTip' private'** or add the keyword `private` before `fun calculateTip()`. The gray line under `calculateTip()` will disappear.

How to inspect the code

The grey line is very subtle and easy to miss. You could manually look through the full file to search for more gray lines, but there's a simpler way to make sure you find all the suggestions.

1. With the element `MainActivity.kt` still open, select **Code > Inspect Code...** from the menu. A dialog box called **Specify Inspection Scope** will appear.



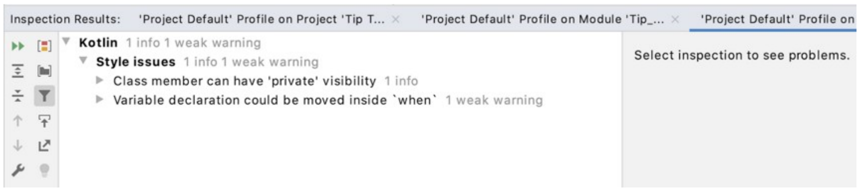
Specify Inspection Scope

Inspection scope

- ○ Whole project
- ○ Module 'Tip_Time.app'
- ● File '.../app/src/main/java/com/example/tiptime/MainActivity.kt [Tip_Time.app]'
- ○ Custom scope: Module 'Tip_Time.app'
- ☑ Include test sources

Inspection profile: Project Default Project

2. Choose the option that starts with **File** and press **OK**. In this way, inspection will be limited to `MainActivity.kt`.
3. A window with **Inspection Results** will appear at the bottom.
4. Click on the grey triangles next to **Kotlin** and then, next to **Style issues** until you see two messages. The first is this: **Class member can have 'private' visibility**.

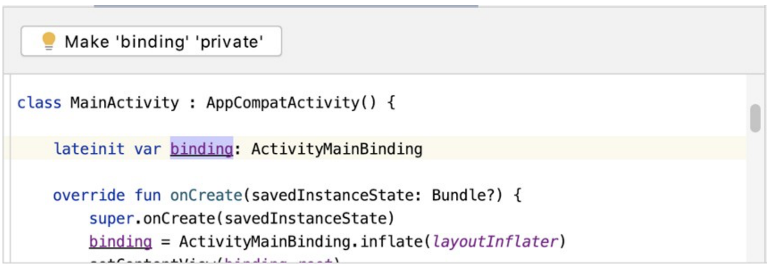


Inspection Results: 'Project Default' Profile on Project 'Tip T... × 'Project Default' Profile on Module 'Tip_... × 'Project Default' Profile on

```
▼ Kotlin 1 info 1 weak warning
  ▼ Style issues 1 info 1 weak warning
    ▶ Class member can have 'private' visibility 1 info
    ▶ Variable declaration could be moved inside `when` 1 weak warning
```

Select inspection to see problems.

5. Click on the grey triangles until you see the message **Property 'binding' could be private** and click on the message. Android Studio shows part of the code in `MainActivity` and highlights the variable `binding`.



Make 'binding' 'private'

```
class MainActivity : AppCompatActivity() {

    lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
```

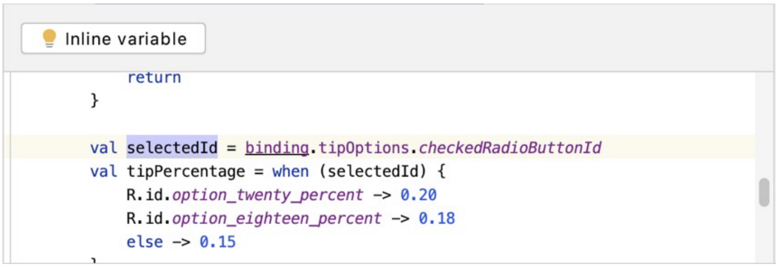
6. Press the **Make 'binding' private**

button. Android Studio removes the problem from **Inspection Results**.

7. If you look at `binding` in your code, you'll see that Android Studio added the keyword `private` before the statement.

```
private lateinit var binding: ActivityMainBinding
```

8. Click on the grey triangles in the results until you see the message **Variable could be inlined**. Android Studio again shows part of the code, but this time highlights the variable `selectedId`.



```
        return
    }

    val selectedId = binding.tipOptions.checkedRadioButtonId
    val tipPercentage = when (selectedId) {
        R.id.option_twenty_percent -> 0.20
        R.id.option_eighteen_percent -> 0.18
        else -> 0.15
```

9. If you look at the code, you'll see `selectedId` is only used twice: first, on the standout line where we get the value of `tipOptions.checkedRadioButtonId`, then, on the next line, in the element `when`.
10. Press the **Inline variable** button. Android Studio replaces `selectedId` in the expression `when` withthe value assigned in the previous line. Then, it completely removes the previous line, because it is no longer necessary.

```
val tipPercentage = when (binding.tipOptions.checkedRadioButtonId) {
    R.id.option_twenty_percent -> 0.20
    R.id.option_eighteen_percent -> 0.18
    else -> 0.15
}
```

That's very practical. The code has one line less and one less variable.

How to remove unnecessary variables

Android Studio has no further inspection results. However, if you look at the code carefully, you'll see a pattern similar to the one you've just changed: the variable `roundup` is assigned on a line, used on the next line and is not used anywhere else.

1. Copy the expression to the right of the element =from the line on which it is assigned `roundUp`.

```
val roundUp = binding.roundUpSwitch.isChecked
```

2. Replaces `roundup` in the next line with the expression you just copied, `binding.roundUpSwitch.isChecked`.

```
if (binding.roundUpSwitch.isChecked) {
    tip = kotlin.math.ceil(tip)
}
```

3. Clear the line with `roundup` because it's no longer necessary.

You did the same thing Android Studio helped you do with the variable `selectedId`. Again, your code has one less line and one less variable. They are small changes, but they help make the code more concise and readable. Imagine in a much larger project, if you didn't do this it would add up to many more lines and variables to manage!

Well done, you now have a functioning Tip Calculator app!

6. Summary

- The view binding feature allows you to write efficient code that interacts with the elements of the UI in your app.
- The `Double` data type in Kotlin can store a decimal number.
- Use the attribute `checkedRadioButtonId` of an element `RadioGroup`to find what `RadioButton` is selected.
- Use `NumberFormat.getCurrencyInstance()`to get a formatter to format numbers as currency.
- You can use string parameters, like `%s` to create dynamic strings that can be easily translated into other languages.
- Testing is important!
- In Android Studio, you can use **Logcat** to identify and fix problems, for example, when the app fails.
- A stack trace shows a list of the methods called, which may be useful if the code generates an exception.
- Exceptions indicate a problem that the code did not expect.
- Use **Analyze - Inspect Code** for suggestions that allow you to improve your code.