UNIT 2 : MODULE 1

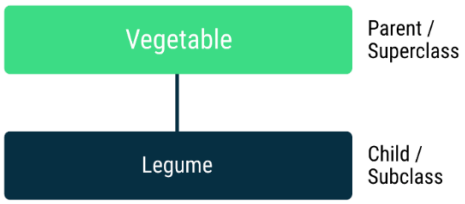
2. What is a class hierarchy?

It is natural for humans to classify items that have similar properties and behavior into groups and to even form some type of hierarchy among them. For example, you can have a broad category like vegetables, and within that you can have a more specific type like legumes. Within legumes, you can have even more specific types like peas, beans, lentils, chickpeas, and soybeans for example.

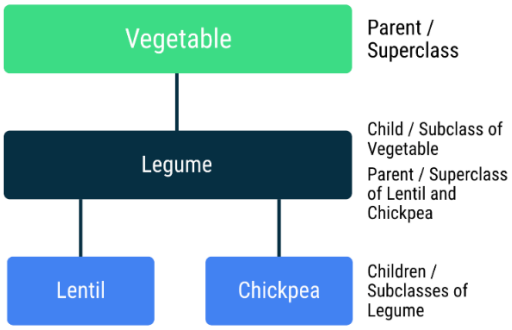
This can be represented as a hierarchy because legumes contain or *inherit* all the properties of vegetables (e.g. they are plants and edible). Similarly, peas, beans, and lentils all have the properties of legumes plus their own unique properties.

Let's look at how you would represent this relationship in programming terms. If you make `Vegetable` a class in Kotlin, you can create `Legume` as a *child* or *subclass* of the `Vegetable` class. That means all the properties and methods of the `Vegetable` class are inherited by (meaning also available in) the `Legume` class.

You can represent this in a *class hierarchy* diagram as shown below. You can refer to `Vegetable` as the *parent* or *superclass* of the `Legume` class.



You could continue and expand the class hierarchy by creating subclasses of `Legume` such as `Lentil` and `Chickpea`. This makes `Legume` both a child or subclass of `Vegetable` as well as a parent or superclass of `Lentil` and `Chickpea`. `Vegetable` is the *root* or *top-level (*or *base)* class of this hierarchy.



Note: Terminology summary

Here is a summary of words used in this codelab and what they mean in the context of Kotlin classes.

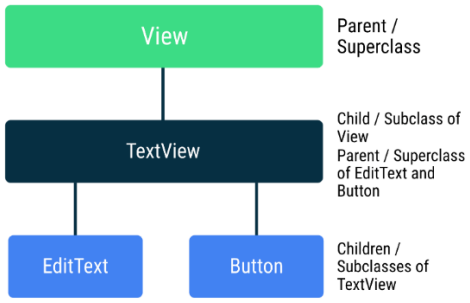
- **Class hierarchy.** An arrangement where classes are organized in a hierarchy of parents and children. Hierarchy diagrams are usually drawn with the parents shown above children.
- **Child or subclass.** Any class that is below another class in the hierarchy.
- **Parent or superclass or base class.** Any class with one or more child classes.

- **Root or top-level class.** The class at the top (or root) of the class hierarchy.
- **Inheritance.** When a child class includes (or inherits) all the properties and methods of its parent class. This allows you to share and reuse code, which makes programs easier to understand and maintain.

Inheritance in Android Classes

While you can write Kotlin code without using classes, and you did in previous codelabs, many parts of Android are provided to you in the form of classes, including activities, views, and view groups. Understanding class hierarchies is therefore fundamental to Android app development and allows you to take advantage of features provided by the Android framework.

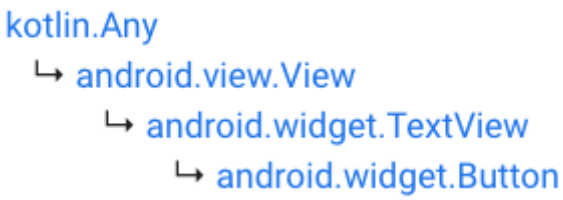
For example, there is a `View` class in Android that represents a rectangular area on the screen and is responsible for drawing and event handling. The `TextView` class is a subclass of the `View` class, which means that `TextView` inherits all the properties and functionality from the `View` class, plus adds specific logic for displaying text to the user.



Taking it a step further, the `EditText` and `Button` classes are children of the `TextView` class. They inherit all the properties and methods of the `TextView` and `View` classes, plus add their own specific logic. For example, `EditText` adds its own functionality of being able to edit text on the screen.

Instead of having to copy and paste all the logic from the `View` and `TextView` classes into the `EditText` class, the `EditText` can just subclass the `TextView` class, which in turn subclasses the `View` class. Then the code in the `EditText` class can focus specifically on what makes this UI component different from other views.

On the top of a documentation page for an Android class on the developer.android.com website, you can see the class hierarchy diagram. If you see `kotlin.Any` at the top of hierarchy, it's because in Kotlin, all classes have a common superclass Any. Learn more here.



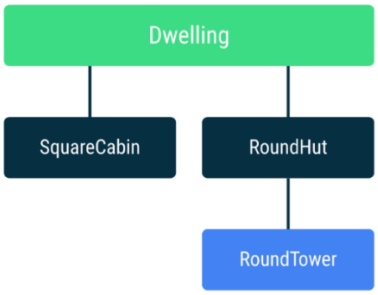
As you can see, learning to leverage inheritance among classes can make your code easier to write, reuse, read, and test.

3. Create a base class

Class hierarchy of dwellings

In this codelab, you are going to build a Kotlin program that demonstrates how class hierarchies work, using dwellings (shelters in which people live) with floor space, stories, and residents as an example.

Below is a diagram of the class hierarchy you are going to build. At the root, you have a `Dwelling` that specifies properties and functionality that is true for all dwellings, similar to a blueprint. You then have classes for a square cabin (`SquareCabin`), round hut (`RoundHut`), and a round tower (`RoundTower`) which is a `RoundHut` with multiple floors.



The classes that you will implement:

- `Dwelling`: a base class representing a non-specific shelter that holds information that is common to all dwellings.
- `SquareCabin`: a square cabin made of wood with a square floor area.
- `RoundHut`: a round hut that is made of straw with a circular floor area, and the parent of `RoundTower`.
- `RoundTower`: a round tower made of stone with a circular floor area and multiple stories.

Create an abstract Dwelling class

Any class can be the base class of a class hierarchy or a parent of other classes.

An "abstract" class is a class that cannot be instantiated because it is not fully implemented. You can think of it as a sketch. A sketch incorporates the ideas and plans for something, but not usually enough information to build it. You use a sketch (abstract class) to create a blueprint (class) from which you build the actual object instance.

A common benefit of creating a superclass is to contain properties and functions that are common to all its subclasses. If the values of properties and implementations of functions are not known, make the class abstract. For example, `Vegetables` have many properties common to all vegetables, but you can't create an instance of a non-specific vegetable, because you don't know, for example, its shape or color. So `Vegetable` is an abstract class that leaves it up to the subclasses to determine specific details about each vegetable.

The declaration of an abstract class starts with the `abstract` keyword.

Note: You'll come across a number of abstract classes in Android that you'll need to subclass within future codelabs of this course.

`Dwelling` is going to be an abstract class like `Vegetable`. It is going to contain properties and functions that are common to many types of dwellings, but the exact values of properties and details of implementation of functions are not known.

1. Go to the Kotlin Playground at https://developer.android.com/training/kotlinplayground.
2. In the editor, delete `println("Hello, world!")` inside the `main()` function.
3. Then add this code, below the `main()` function to create an `abstract` class called `Dwelling`.

```
abstract class Dwelling(){
}
```

Add a property for building material

In this `Dwelling` class, you define things that are true for all dwellings, even if they may be different for different dwellings. All dwellings are made of some building material.

4. Inside `Dwelling`, create a `buildingMaterial` variable of type `String` to represent the building material. Since the building material won't change, use `val` to make it an immutable variable.

```
val buildingMaterial: String
```

5. Run your program and you get this error.

```
Property must be initialized or be abstract
```

The `buildingMaterial` property does not have a value. In fact, you CAN'T give it a value, because a non-specific building isn't made of anything specific. So, as the error message indicates, you can prefix the declaration of `buildingMaterial` with the `abstract` keyword, to indicate that it is not going to be defined here.

6. Add the `abstract` keyword onto the variable definition.

```
abstract val buildingMaterial: String
```

7. Run your code, and while it does not do anything, it now compiles without errors.
8. Make an instance of `Dwelling` in the `main()` function and run your code.

```
val dwelling = Dwelling()
```

9. You'll get an error because you cannot create an instance of the abstract `Dwelling` class.

```
Cannot create an instance of an abstract class
```

10. Delete this incorrect code.

Your finished code so far:

```
abstract class Dwelling(){
    abstract val buildingMaterial: String
}
```

Add a property for capacity

Another property of a dwelling is the capacity, that is, how many people can live in it.

All dwellings have a capacity that doesn't change. However, the capacity cannot be set within the `Dwelling` superclass. It should be defined in subclasses for specific types of dwellings.

1. In `Dwelling`, add an `abstract` integer `val` called `capacity`.

```
abstract val capacity: Int
```

Add a private property for number of residents

All dwellings will have a number of `residents` who reside in the dwelling (which may be less than or equal to the `capacity`), so define the `residents` property in the `Dwelling` superclass for all subclasses to inherit and use.

1. You can make `residents` a parameter that is passed into the constructor of the `Dwelling` class. The `residents` property is a `var`, because the number of residents can change after the instance has been created.

```
abstract class Dwelling(private var residents: Int) {
```

Notice that the `residents` property is marked with the `private` keyword. Private is a visibility modifier in Kotlin meaning that the `residents` property is only visible to (and can be used inside) this class. It cannot be accessed from elsewhere in your program. You can mark properties or methods with the private keyword. Otherwise when no visibility modifier is specified, the properties and methods are `public` by default and accessible from other parts of your program. Since the number of people who live in a dwelling is usually private information (compared to information about the building material or the capacity of the building), this is a reasonable decision.

With both the `capacity` of the dwelling and the number of current `residents` defined, you can create a function `hasRoom()` to determine whether there is room for another resident in the dwelling. You can define and implement the `hasRoom()` function in the `Dwelling` class because the formula for calculating whether there is room is the same for all dwellings. There is room in a `Dwelling` if the number of `residents` is less than the `capacity`, and the function should return `true` or `false` based on this comparison.

2. Add the `hasRoom()` function to the `Dwelling` class.

```
fun hasRoom(): Boolean {
    return residents < capacity
}
```

3. You can run this code and there should be no errors. It doesn't do anything visible yet.

Your completed code should look like this:

```
abstract class Dwelling(private var residents: Int) {

    abstract val buildingMaterial: String
    abstract val capacity: Int
```

```
    fun hasRoom(): Boolean {
        return residents < capacity
    }
}
```

4. Create subclasses

Create a SquareCabin subclass

1. Below the `Dwelling` class, create a class called `SquareCabin`.

   ```
   class SquareCabin
   ```

2. Next, you need to indicate that `SquareCabin` is related to `Dwelling`. In your code, you want to indicate that `SquareCabin` extends from `Dwelling` (or is a subclass to `Dwelling`) because `SquareCabin` will provide an implementation for the abstract parts of `Dwelling`.

   Indicate this inheritance relationship by adding a colon (`:`) after the `SquareCabin` class name, followed by a call to initialize the parent `Dwelling` class. Don't forget to add parentheses after the `Dwelling` class name.

   ```
   class SquareCabin : Dwelling()
   ```

3. When extending from a superclass, you must pass in the required parameters expected by the superclass. `Dwelling` requires the number of `residents` as input. You could pass in a fixed number of residents like `3`.

   ```
   class SquareCabin : Dwelling(3)
   ```

   However, you want your program to be more flexible and allow for a variable number of residents for `SquareCabins`. Hence make `residents` a parameter in the `SquareCabin` class definition. Do not declare `residents` as `val`, because you are reusing a property already declared in the parent class `Dwelling`.

   ```
   class SquareCabin(residents: Int) : Dwelling(residents)
   ```

   Note: A lot happens under the hood with these class definitions.

   In the class header, you see **`class SquareCabin(residents: Int) ...`**

   This is actually shorthand for **`class SquareCabin constructor(residents: Int) ...`**

   The **`constructor`** is called when you create an object instance from a class. For example when you call **`SquareCabin(4)`**, the **`constructor`** of **`SquareCabin`** is called to initialize an object instance.

   The **`constructor`** builds your instance from all the information in the class, including the passed in arguments. When a class inherits properties and functions from a parent, the **`constructor`** calls the **`constructor`** of the parent class to finish initializing the object instance.

   Hence when you create an instance using **`SquareCabin(4)`**, the **`constructor`** for **`SquareCabin`** is executed and because of the inheritance relationship, the **`Dwelling`** constructor also gets executed. The **`Dwelling`** class definition specifies that its constructor requires a **`residents`** parameter, so that's why you

see **`residents`** passed to the **`Dwelling`** constructor in the **`SquareCabin`** class definition. You will learn more about constructors in later codelabs.

4. Run your code.
5. This will cause errors. Take a look:

Class 'SquareCabin' is not abstract and does not implement abstract base class member public abstract val buildingMaterial: String defined in Dwelling

When you declare abstract functions and variables, it is like a promise that you will give them values and implementations later. For a variable, it means that any subclass of that abstract class needs to give it a value. For a function, it means that any subclass needs to implement the function body.

In the `Dwelling` class, you defined an `abstract` variable `buildingMaterial`. `SquareCabin` is a subclass of `Dwelling`, so it must provide a value for `buildingMaterial`. Use the `override` keyword to indicate that this property was defined in a parent class and is about to be overridden in this class.

6. Inside the `SquareCabin` class, `override` the `buildingMaterial` property and assign it the value `"Wood"`.
7. Do the same for the `capacity`, saying 6 residents can live in a `SquareCabin`.

```
class SquareCabin(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Wood"
    override val capacity = 6
}
```

Your finished code should look like this.

```
abstract class Dwelling(private var residents: Int) {
    abstract val buildingMaterial: String
    abstract val capacity: Int

    fun hasRoom(): Boolean {
        return residents < capacity
    }
}

class SquareCabin(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Wood"
    override val capacity = 6
}
```

To test your code, create an instance of `SquareCabin` in your program.

Use SquareCabin

1. Insert an empty `main()` function before the `Dwelling` and `SquareCabin` class definitions.

```
fun main() {
}
```

```
abstract class Dwelling(private var residents: Int) {
    ...
}
class SquareCabin(residents: Int) : Dwelling(residents) {
    ...
}
```

2. Within the `main()` function, create an instance of `SquareCabin` called `squareCabin` with 6 residents. Add print statements for the building material, the capacity, and the `hasRoom()` function.

```
fun main() {
    val squareCabin = SquareCabin(6)

    println("\nSquare Cabin\n============")
    println("Capacity: ${squareCabin.capacity}")
    println("Material: ${squareCabin.buildingMaterial}")
    println("Has room? ${squareCabin.hasRoom()}")
}
```

Notice that the `hasRoom()` function was not defined in the `SquareCabin` class, but it was defined in the `Dwelling` class. Since `SquareCabin` is a subclass to `Dwelling` class, the `hasRoom()` function was inherited for free. The `hasRoom()` function can now be called on all instances of `SquareCabin`, as seen in the code snippet as `squareCabin.hasRoom()`.

3. Run your code, and it should print the following.

```
Square Cabin

============

Capacity: 6

Material: Wood

Has room? false
```

You created `squareCabin` with `6` residents, which is equal to the `capacity`, so `hasRoom()` returns `false`. You could experiment with initializing `SquareCabin` with a smaller number of `residents`, and when you run your program again, `hasRoom()` should return `true`.

Use with to simplify your code

In the `println()` statements, every time you reference a property or function of `squareCabin`, notice how you have to repeat `squareCabin.` This becomes repetitive and can be a source of errors when you copy and paste print statements.

When you are working with a specific instance of a class and need to access multiple properties and functions of that instance, you can say "do all the following operations with this instance object" using a `with` statement. Start with the keyword `with`, followed by the instance name in parentheses, followed by curly braces which contain the operations you want to perform.

```
with (instanceName) {
    // all operations to do with instanceName
}
```

1. In the `main()` function, change your print statements to use `with`.
2. Delete `squareCabin.` in the print statements.

```
with(squareCabin) {
    println("\nSquare Cabin\n============")
    println("Capacity: ${capacity}")
    println("Material: ${buildingMaterial}")
    println("Has room? ${hasRoom()}")
}
```

3. Run your code again to make sure it runs without errors and shows the same output.

```
Square Cabin

============

Capacity: 6

Material: Wood

Has room? false
```

This is your completed code:

```
fun main() {
    val squareCabin = SquareCabin(6)

    with(squareCabin) {
        println("\nSquare Cabin\n============")
        println("Capacity: ${capacity}")
        println("Material: ${buildingMaterial}")
        println("Has room? ${hasRoom()}")
    }
}

abstract class Dwelling(private var residents: Int) {
    abstract val buildingMaterial: String
    abstract val capacity: Int

    fun hasRoom(): Boolean {
        return residents < capacity
    }
}
```

```
class SquareCabin(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Wood"
    override val capacity = 6
}
```

Create a RoundHut subclass

1. In the same way as the `SquareCabin`, add another subclass, `RoundHut`, to `Dwelling`.
2. Override `buildingMaterial` and give it a value of `"Straw"`.
3. Override `capacity` and set it to 4.

```
class RoundHut(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Straw"
    override val capacity = 4

}
```

4. In `main()`, create an instance of `RoundHut` with 3 residents.

```
val roundHut = RoundHut(3)
```

5. Add the code below to print information about `roundHut`.

```
with(roundHut) {
    println("\nRound Hut\n==========")
    println("Material: ${buildingMaterial}")
    println("Capacity: ${capacity}")
    println("Has room? ${hasRoom()}")
}
```

6. Run your code and your output for the whole program should be:

```
Square Cabin

============

Capacity: 6

Material: Wood

Has room? false



Round Hut
```

=========

Material: Straw

Capacity: 4

Has room? true

You now have a class hierarchy that looks like this, with `Dwelling` as the root class and `SquareCabin` and `RoundHut` as subclasses of `Dwelling`.

Create a RoundTower subclass

The final class in this class hierarchy is a round tower. You can think of a round tower as a round hut made of stone, with multiple stories. So, you can make `RoundTower` a subclass of `RoundHut`.

1. Create a `RoundTower` class that is a subclass of `RoundHut`. Add the `residents` parameter to the constructor of `RoundTower`, and then pass that parameter to the constructor of the `RoundHut` superclass.
2. Override the `buildingMaterial` to be `"Stone"`.
3. Set the `capacity` to `4`.

```
class RoundTower(residents: Int) : RoundHut(residents) {
    override val buildingMaterial = "Stone"
    override val capacity = 4
}
```

1. Run this code and you get an error.

This type is final, so it cannot be inherited from

This error means that the `RoundHut` class cannot be subclassed (or inherited from). By default, in Kotlin, classes are final and cannot be subclassed. You are only allowed to inherit from `abstract` classes or classes that are marked with the `open` keyword. Hence you need to mark the `RoundHut` class with the `open` keyword to allow it to be inherited from.

Note: You do not need to use the **`open`** keyword when defining abstract classes. Example: **`Dwelling`** class does not need to be marked with the **`open`** keyword. It is assumed that you will want to subclass an abstract class in order to implement the abstract properties and functions.

4. Add the `open` keyword at the start of the `RoundHut` declaration.

```
open class RoundHut(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Straw"
    override val capacity = 4
```

```
}
```

5. In `main()`, create an instance of `roundTower` and print information about it.

```
 val roundTower = RoundTower(4)


with(roundTower) {
    println("\nRound Tower\n==========")
    println("Material: ${buildingMaterial}")
    println("Capacity: ${capacity}")
    println("Has room? ${hasRoom()}")
}
```

Here is the complete code.

```
fun main() {
    val squareCabin = SquareCabin(6)
    val roundHut = RoundHut(3)
    val roundTower = RoundTower(4)

    with(squareCabin) {
        println("\nSquare Cabin\n============")
        println("Capacity: ${capacity}")
        println("Material: ${buildingMaterial}")
        println("Has room? ${hasRoom()}")
    }

    with(roundHut) {
        println("\nRound Hut\n=========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
    }

    with(roundTower) {
        println("\nRound Tower\n==========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
    }
}


abstract class Dwelling(private var residents: Int) {
    abstract val buildingMaterial: String
    abstract val capacity: Int
```

```
    fun hasRoom(): Boolean {
      return residents < capacity
    }
  }

  class SquareCabin(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Wood"
    override val capacity = 6
  }

  open class RoundHut(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Straw"
    override val capacity = 4
  }

  class RoundTower(residents: Int) : RoundHut(residents) {
    override val buildingMaterial = "Stone"
    override val capacity = 4
  }
```

6. Run your code. It should now work without errors and produce the following output.

```
Square Cabin

============

Capacity: 6

Material: Wood

Has room? false



Round Hut

=========

Material: Straw

Capacity: 4

Has room? true



Round Tower

==========
```

Material: Stone

Capacity: 4

Has room? false

Add multiple floors to RoundTower

`RoundHut`, by implication, is a single-story building. Towers usually have multiple stories (floors).

Thinking of the capacity, the more floors a tower has, the more capacity it should have.

You can modify `RoundTower` to have multiple floors, and adjust its capacity based on the number of floors.

1. Update the `RoundTower` constructor to take an additional integer parameter `val floors` for the number of floors. Put it after `residents`. Notice that you don't need to pass this to the parent `RoundHut` constructor because `floors` is defined here in `RoundTower` and `RoundHut` has no `floors`.

```
class RoundTower(
    residents: Int,
    val floors: Int) : RoundHut(residents) {

    ...
}
```

Note: When you have multiple parameters and your class or function definition gets too long to comfortably fit on one line, you can break it up into multiple lines as shown in the code above.

2. Run your code. There is an error when creating `roundTower` in the `main()` method, because you are not supplying a number for the `floors` argument. You could add the missing argument.

   Alternatively, in the class definition of `RoundTower`, you can add a default value for `floors` as shown below. Then, when no value for `floors` is passed into the constructor, the default value can be used to create the object instance.

3. In your code, add `= 2` after the declaration of `floors` to assign it a default value of 2.

```
class RoundTower(
    residents: Int,
    val floors: Int = 2) : RoundHut(residents) {

    ...
}
```

4. Run your code. It should compile because `RoundTower(4)` now creates a `RoundTower` object instance with the default value of 2 floors.
5. In the `RoundTower` class, update the `capacity` to multiply it by the number of floors.

```
override val capacity = 4 * floors
```

6. Run your code and notice that the `RoundTower` capacity is now 8 for 2 floors.

Here is your finished code.

```
fun main() {

    val squareCabin = SquareCabin(6)
    val roundHut = RoundHut(3)
    val roundTower = RoundTower(4)

    with(squareCabin) {
        println("\nSquare Cabin\n============")
        println("Capacity: ${capacity}")
        println("Material: ${buildingMaterial}")
        println("Has room? ${hasRoom()}")
    }

    with(roundHut) {
        println("\nRound Hut\n=========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
    }

    with(roundTower) {
        println("\nRound Tower\n==========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
    }
}

abstract class Dwelling(private var residents: Int) {
    abstract val buildingMaterial: String
    abstract val capacity: Int

    fun hasRoom(): Boolean {
      return residents < capacity
    }
}

class SquareCabin(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Wood"
    override val capacity = 6
}

open class RoundHut(residents: Int) : Dwelling(residents) {
    override val buildingMaterial = "Straw"
```

```
    override val capacity = 4
}

class RoundTower(
    residents: Int,
    val floors: Int = 2) : RoundHut(residents) {

    override val buildingMaterial = "Stone"
    override val capacity = 4 * floors
}
```

5. Modify classes in the hierarchy

Calculate the floor area

In this exercise, you will learn how you can declare an abstract function in an abstract class and then implement its functionality in the subclasses.

All dwellings have floor area, however, depending on the shape of the dwelling, it's calculated differently.

Define floorArea() in Dwelling class

1. First add an `abstract` `floorArea()` function to the `Dwelling` class. Return a `Double`. Double is a data type, like `String` and `Int`; it is used for floating point numbers, that is, numbers that have a decimal point followed by a fractional part, such as 5.8793.)

```
abstract fun floorArea(): Double
```

All abstract methods defined in an abstract class must be implemented in any of its subclasses. Before you can run your code, you need to implement `floorArea()` in the subclasses.

Implement floorArea() for SquareCabin

Like with `buildingMaterial` and `capacity`, since you are implementing an `abstract` function that's defined in the parent class, you need to use the `override` keyword.

1. In the `SquareCabin` class, start with the keyword `override` followed by the actual implementation of the `floorArea()` function as shown below.

```
override fun floorArea(): Double {
}
```

2. Return the calculated floor area. The area of a rectangle or square is the length of its side multiplied by the length of its other side. The body of the function will `return length * length`.

```
override fun floorArea(): Double {
    return length * length
```

```
}
```

The length is not a variable in the class, and it is different for every instance, so you can add it as a constructor parameter for the `SquareCabin` class.

3. Change the class definition of `SquareCabin` to add a `length` parameter of type `Double`. Declare the property as a `val` because the length of a building doesn't change.

```
class SquareCabin(residents: Int, val length: Double) : Dwelling(residents) {
```

`Dwelling` and therefore all its subclasses have `residents` as a constructor argument. Because it's the first argument in the `Dwelling` constructor, It is a best practice to also make it the first argument in all subclass constructors and put the arguments in the same order in all the class definitions. Hence insert the new `length` parameter after the `residents` parameter.

4. In `main()` update the creation of the `squareCabin` instance. Pass in `50.0` to the `SquareCabin` constructor as the `length`.

```
val squareCabin = SquareCabin(6, 50.0)
```

5. Inside the `with` statement for `squareCabin`, add a print statement for the floor area.

```
println("Floor area: ${floorArea()}")
```

Your code won't run, because you also have to implement `floorArea()` in `RoundHut`.

Implement floorArea() for RoundHut

In the same way, implement the floor area for `RoundHut`. `RoundHut` is also a direct subclass of `Dwelling`, so you need to use the `override` keyword.

The floor area of a circular dwelling is PI * radius^2, or PI * radius * radius.

`PI` is a mathematical value. It is defined in a math library. A library is a predefined collection of functions and values defined outside a program that a program can use. In order to use a library function or value, you need to tell the compiler that you are going to use it. You do this by importing the function or value into your program. To use `PI` in your program, you need to import `kotlin.math.PI`.

1. Import `PI` from the Kotlin math library. Put this at the top of the file, before `main()`.

```
import kotlin.math.PI
```

2. Implement the `floorArea()` function for `RoundHut`.

```
override fun floorArea(): Double {
    return PI * radius * radius
}
```

Warning: If you do not import kotlin.math.PI, you will get an error, so import this library before using it. Alternatively, you could write out the fully qualified version of PI, as in kotlin.math.PI * radius * radius, and then the import statement is not needed.

3. Update the RoundHut constructor to pass in the radius.

```
open class RoundHut(
  val residents: Int,
  val radius: Double) : Dwelling(residents) {
```

4. In main(), update the initialization of roundHut by passing in a radius of 10.0 to the RoundHut constructor.

```
val roundHut = RoundHut(3, 10.0)
```

5. Add a print statement inside the with statement for roundHut.

```
println("Floor area: ${floorArea()}")
```

Implement floorArea() for RoundTower

Your code doesn't run yet, and fails with this error:

```
Error: No value passed for parameter 'radius'
```

In RoundTower, in order for your program to compile, you don't need to implement floorArea() as it gets inherited from RoundHut, but you need to update the RoundTower class definition to also have the same radius argument as its parent RoundHut.

1. Change the constructor of RoundTower to also take the radius. Put the radius after residents and before floors. It is recommended that variables with default values are listed at the end. Remember to pass radius to the parent class constructor.

```
class RoundTower(
   residents: Int,
   radius: Double,
   val floors: Int = 2) : RoundHut(residents, radius) {
```

2. Update the initialization of roundTower in main().

```
val roundTower = RoundTower(4, 15.5)
```

3. And add a print statement that calls floorArea().

```
println("Floor area: ${floorArea()}")
```

4. You can now run your code!

5. Notice that the calculation for the `RoundTower` is not correct, because it is inherited from `RoundHut` and does not take into account the number of `floors`.
6. In `RoundTower`, `override floorArea()` so you can give it a different implementation that multiplies the area with the number of floors. Notice how you can define a function in an abstract class (`Dwelling`), implement it in a subclass (`RoundHut`) and then override it again in a subclass of the subclass (`RoundTower`). It's the best of both worlds - you inherit the functionality you want, and can override the functionality you don't want.

```
override fun floorArea(): Double {
    return PI * radius * radius * floors
}
```

This code works, but there's a way to avoid repeating code that is already in the `RoundHut` parent class. You can call the `floorArea()` function from the parent `RoundHut` class, which returns `PI * radius * radius`. Then multiply that result by the number of `floors`.

7. In `RoundTower`, update `floorArea()` to use the superclass implementation of `floorArea()`. Use the `super` keyword to call the function that is defined in the parent.

```
override fun floorArea(): Double {
    return super.floorArea() * floors
}
```

8. Run your code again and `RoundTower` outputs the correct floor space for multiple floors.

Here is your finished code:

```
import kotlin.math.PI

fun main() {

    val squareCabin = SquareCabin(6, 50.0)
    val roundHut = RoundHut(3, 10.0)
    val roundTower = RoundTower(4, 15.5)

    with(squareCabin) {
        println("\nSquare Cabin\n============")
        println("Capacity: ${capacity}")
        println("Material: ${buildingMaterial}")
        println("Has room? ${hasRoom()}")
        println("Floor area: ${floorArea()}")
    }

    with(roundHut) {
        println("\nRound Hut\n=========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
```

```
        println("Floor area: ${floorArea()}")
    }

    with(roundTower) {
        println("\nRound Tower\n==========")
        println("Material: ${buildingMaterial}")
        println("Capacity: ${capacity}")
        println("Has room? ${hasRoom()}")
        println("Floor area: ${floorArea()}")
    }
 }

abstract class Dwelling(private var residents: Int) {

    abstract val buildingMaterial: String
    abstract val capacity: Int

    fun hasRoom(): Boolean {
        return residents < capacity
}

    abstract fun floorArea(): Double
}

class SquareCabin(residents: Int,
    val length: Double) : Dwelling(residents) {

    override val buildingMaterial = "Wood"
    override val capacity = 6

    override fun floorArea(): Double {
      return length * length
    }
}

open class RoundHut(val residents: Int,
    val radius: Double) : Dwelling(residents) {

    override val buildingMaterial = "Straw"
    override val capacity = 4

    override fun floorArea(): Double {
        return PI * radius * radius
    }
}

class RoundTower(residents: Int, radius: Double,
    val floors: Int = 2) : RoundHut(residents, radius) {
```

```
    override val buildingMaterial = "Stone"
    override val capacity = 4 * floors

    override fun floorArea(): Double {
        return super.floorArea() * floors
    }
}
```

The output should be:

```
Square Cabin

============

Capacity: 6

Material: Wood

Has room? false

Floor area: 2500.0


Round Hut

=========

Material: Straw

Capacity: 4

Has room? true

Floor area: 314.1592653589793


Round Tower

==========

Material: Stone

Capacity: 8

Has room? true

Floor area: 1509.5352700498956
```

Note: For the area values, it would be a nicer user experience to only show a couple of decimal places. This is out of scope for this codelab, but you could print the floor area using this: println("Floor area: %.2f".format(floorArea()))

Allow a new resident to get a room

Add the ability for a new resident to get a room with a `getRoom()` function that increases the number of residents by one. Since this logic is the same for all dwellings, you can implement the function in `Dwelling`, and this makes it available to all subclasses and their children. Neat!

Notes:

- Use an `if` statement that only adds a resident if there is capacity left.
- Print a message for the outcome.
- You can use `residents++` as a shorthand for `residents = residents + 1` to add 1 to the `residents` variable.

1. Implement the `getRoom()` function in the `Dwelling` class.

```
fun getRoom() {
    if (capacity > residents) {
        residents++
        println("You got a room!")
    } else {
        println("Sorry, at capacity and no rooms left.")
    }
}
```

2. Add some print statements to the `with` statement block for `roundHut` to observe what happens with `getRoom()` and `hasRoom()` used together.

```
println("Has room? ${hasRoom()}")
getRoom()
println("Has room? ${hasRoom()}")
getRoom()
```

Output for these print statements:

```
Has room? true

You got a room!

Has room? false

Sorry, at capacity and no rooms left.
```

See Solution code for details.

Note: This is an example of why inheritance is so powerful. You can call the **`getRoom()`** function on all the subclasses of **`Dwelling`** without additional code in those classes.

Fit a carpet into a round dwelling

Let's say you need to know what size carpet to get for your `RoundHut` or `RoundTower`. For the `SquareCabin`, the max carpet is the same as the floor area from the length, so no extra calculations are needed. Put the function into `RoundHut` to make it available to all round dwellings.

1. First import the `sqrt()` function from the `kotlin.math` library.

```
import kotlin.math.sqrt
```

2. Implement the `calculateMaxCarpetSize()` function in the `RoundHut` class. The formula for fitting a rectangle into a circle is the square root of the diameter squared divided by 2: `sqrt (diameter * diameter / 2)`

```
fun calculateMaxCarpetSize(): Double {
    val diameter = 2 * radius
    return sqrt(diameter * diameter / 2)
}
```

3. The `calculateMaxCarpetSize()` method can now be called on `RoundHut` and `RoundTower` instances. Add print statements to `roundHut` and `roundTower` in the `main()` function.

```
println("Carpet size: ${calculateMaxCarpetSize()}")
```

See Solution code for details.

Note: Notice that you added **`calculateMaxCarpetSize()`** to **`RoundHut`**, and **`RoundTower`** inherits it. However, you cannot call this function on the **`squareCabin`** instance because it doesn't define it or have a superclass that defines it.

Congratulations! You have created a complete class hierarchy with properties and functions, learning everything you need to create more useful classes!

7. Summary

In this codelab you learned how to:

- Create a class hierarchy, that is a tree of classes where children inherit functionality from parent classes. Properties and functions are inherited by subclasses.
- Create an `abstract` class where some functionality is left to be implemented by its subclasses. An `abstract` class can therefore not be instantiated.
- Create subclasses of an `abstract` class.
- Use `override` keyword to override properties and functions in subclasses.
- Use the `super` keyword to reference functions and properties in the parent class.
- Make a class `open` so that it can be subclassed.
- Make a property `private`, so it can only be used inside the class.
- Use the `with` construct to make multiple calls on the same object instance.
- Import functionality from the `kotlin.math` library

In this pathway, you'll build a simple version of a tip calculator as an Android app.

Developers will often work in this way—getting a simple version of the app ready and partially functioning (even if it doesn't look very good), and then making it fully functional and visually polished later.

You will be using these UI elements that are provided by Android:

- `EditText` - for entering and editing text
- `TextView` - to display text like the service question and tip amount
- `RadioButton` - a selectable radio button for each tip option
- `RadioGroup` - to group the radio button options
- `Switch` - an on/off toggle for choosing whether to round up the tip or not

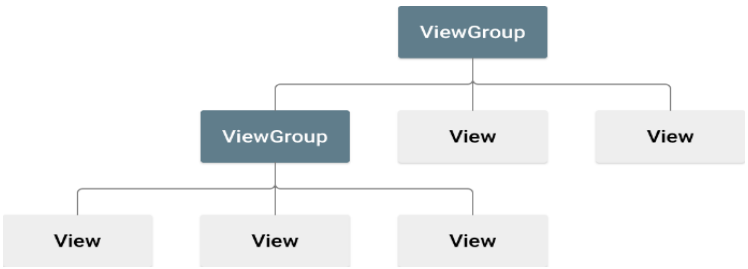
3. Read and understand XML

Instead of using the **Layout Editor** which you're already familiar with, you will build the layout of your application by modifying the XML that describes the UI. Learning how to understand and modify UI layouts using XML will be important for you as an Android developer.

You will be looking at, and editing the XML file that defines the UI layout for this app. XML stands for *eXtensible Markup Language*, which is a way of describing data using a text-based document. Because XML is extensible and very flexible, it's used for many different things, including defining the UI layout of Android apps. You may recall from earlier codelabs that other resources like strings for your app are also defined in an XML file called `strings.xml`.

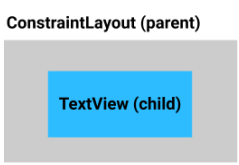
The UI for an Android app is built as a containment hierarchy of components (widgets), and the on-screen layouts of those components. Note those layouts are UI components themselves.

You describe the view hierarchy of UI elements on the screen. For example, a `ConstraintLayout` (the parent) can contain `Buttons`, `TextViews`, `ImageViews`, or other views (the children). Remember, `ConstraintLayout` is a subclass of `ViewGroup`. It allows you to position or size child views in a flexible manner.



Containment hierarchy of an Android app

NOTE: The visible UI hierarchy is based on containment, ie, one component has one or more components inside of it. This is unrelated to the hierarchy of classes and subclasses that you learned earlier. The terms parent and child are sometimes used, but the context here is talking about parent views (viewgroups) containing children views, which in turn can contain children views.

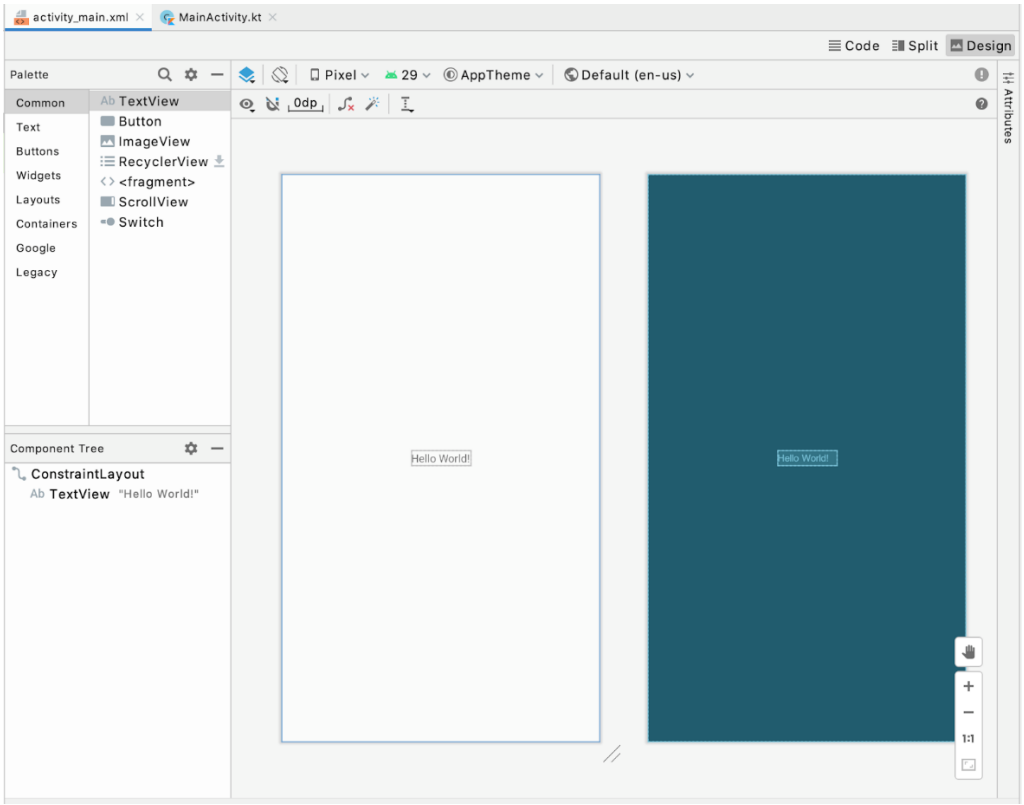


Each UI element is represented by an XML *element* in the XML file. Each element starts and ends with a tag, and each tag starts with a `<` and ends with a `>`. Just as you can set attributes on UI elements using the **Layout Editor (design)**, the XML elements can have *attributes*, too. Simplified, the XML for the UI elements above might be something like this:

```
<ConstraintLayout>
  <TextView
    text="Hello World!">
  </TextView>
</ConstraintLayout>
```

Let's look at a real example.

1. Open `activity_main.xml` (**res** > **layout** > **activity_main.xml**)
2. You might notice the app shows a `TextView` with "Hello World!" within a `ConstraintLayout`, as you have seen in previous projects created from this template.



3. Find the options for **Code**, **Split**, and **Design** views in the upper right of the Layout Editor.
4. Select the **Code** view.

This is what the XML in `activity_main.xml` looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hello World!"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintLeft_toLeftOf="parent"
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

There's a *lot* more going on than in the simplified example, but Android Studio does some things to help make the XML more readable, just as it does with your Kotlin code.

5. Notice the indentation. Android Studio does this automatically to show you the hierarchy of elements. The `TextView` is indented because it is contained in the `ConstraintLayout`. The `ConstraintLayout` is the parent, and the `TextView` is the child. The attributes for each element are indented to show that they're part of that element.
6. Notice the color coding—some things are in blue, some in green, and so on. Similar parts of the file are drawn in the same color to help you match them up. In particular, notice that Android Studio draws the start and end of tags of elements in the same color. (Note: the colors used in the codelab may not match what you see in Android Studio.)

XML tags, elements and attributes

Here's a simplified version of the `TextView` element so you can look at some of the important parts:

```
<TextView
   android:text="Hello World!"
/>
```

The line with `<TextView` is the start of the tag, and the line with `/>` is the end of the tag. The line with `android:text="Hello World!"` is an attribute of the tag. It represents text that will be displayed by the `TextView`. These 3 lines are a commonly used shorthand called an *empty-element tag*. It would mean the same thing if you wrote it with a separate *start-tag* and *end-tag*, like this:

```
<TextView
   android:text="Hello World!"
></TextView>
```

It's also common with an empty-element tag to write it on as few lines as possible and combine the end of the tag with the line before it. So you might see an empty-element tag on two lines (or even one line if it had no attributes):

```
<!-- with attributes, two lines -->
<TextView
   android:text="Hello World!" />
```

The `ConstraintLayout` element is written with separate start and end tags, because it needs to be able to hold other elements inside it. Here's a simplified version of the `ConstraintLayout` element with the `TextView` element inside it:

```
<androidx.constraintlayout.widget.ConstraintLayout>
   <TextView
      android:text="Hello World!" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

If you wanted to add another `View` as a child of the `ConstraintLayout`, like a `Button` below the `TextView`, it would go after the end of the `TextView` tag `/>` and before the end tag of the `ConstraintLayout`, like this:

```
<androidx.constraintlayout.widget.ConstraintLayout>
   <TextView
      android:text="Hello World!" />
   <Button
      android:text="Calculate" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

More about XML for layouts

1. Look at the tag for the `ConstraintLayout`, and notice that it says `androidx.constraintlayout.widget.ConstraintLayout` instead of just `ConstraintLayout` like the `TextView`. This is because `ConstraintLayout` is part of Android Jetpack, which contains libraries of code which offers additional functionality on top of the core Android platform. Jetpack has useful functionality you can take advantage of to make building apps easier. You'll recognize this UI component is part of Jetpack because it starts with "androidx".
2. You may have noticed the lines that begin with `xmlns:` followed by `android`, `app`, and `tools`.

   ```
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   ```

   The `xmlns` stands for XML namespace, and each line defines a *schema*, or vocabulary for attributes related to those words. The `android:` namespace, for example, marks attributes that are defined by the Android system. All of the attributes in the layout XML begins with one of those namespaces.

3. Whitespace between XML elements doesn't change the meaning to a computer, but it can help make the XML easier for people to read.

   Android Studio will automatically add some whitespace and indenting for readability. You'll learn later how to have Android Studio make sure your XML follows coding style conventions.

4. You can add comments to XML, just like you would with Kotlin code. Start `<!--` and end with `-->`.

   ```
   <!-- this is a comment in XML -->
   ```

   ```
   <!-- this is a
   ```

```
multi-line
Comment.
And another
Multi-line comment -->
```

5. Note the first line of the file:

```
<?xml version="1.0" encoding="utf-8"?>
```

This indicates that the file is an XML file, but not every XML file includes this.

Note: If there's a problem with the XML for your app, Android Studio will flag the problem by drawing the text in red. If you move the mouse over the red text, Android Studio will show more information about the problem. If the problem isn't obvious, look at the indenting and color coding, and they may give you a clue to what is wrong.

4. Build the layout in XML

1. Still in `activity_main.xml`, switch to the **Split** screen view to see the XML beside the **Design Editor**. The **Design Editor** allows you to preview your UI layout.
2. It is up to personal preference which view you use, but for this codelab, use the **Split** view so you can see both the XML that you edit and the changes those edits make in the **Design Editor**.
3. Try clicking on different lines, one below the `ConstraintLayout`, and then one below the `TextView`, and notice that the corresponding view is selected in the **Design Editor**. The reverse works, too—for example, if you click on the `TextView` in the **Design Editor**, the corresponding XML is highlighted.

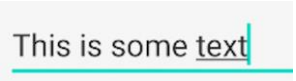
Delete the TextView

1. You don't need the `TextView` now, so delete it. Be sure to delete everything from the `<TextView` to the closing `/>`.
2. Add 16dp of padding to the `ConstraintLayout` so the UI won't be crowded against the edge of the screen.

Padding is similar to margins, but it adds space to the inside of the `ConstraintLayout`, instead of adding space to the outside.

Add cost of service text field

In this step you'll add the UI element to allow the user to enter the cost of service into the app. You'll use an `EditText` element, which lets a user enter or modify text in an app.



1. Look at the EditText documentation, and examine the sample XML.
2. Find a blank space between the opening and closing tags of the `ConstraintLayout`.
3. Copy and paste the XML from the documentation into that space in your layout in Android Studio.

Your layout file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <EditText
      android:id="@+id/plain_text_input"
      android:layout_height="wrap_content"
      android:layout_width="match_parent"
      android:inputType="text"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

You may not understand all of this yet, but it will be explained in the following steps.

4. Notice `EditText` is underlined in red.
5. Hover the pointer over it, and you'll see a "view is not constrained" error, which should look familiar from earlier codelabs. Recall that children of a `ConstraintLayout` need constraints so the layout knows how to arrange them.

6. Add these constraints to the `EditText` to anchor it to the top left corner of the parent.

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
```

If you're writing in English or another language that's written left-to-right (LTR), the starting edge is the left. But some languages like Arabic are written right-to-left (RTL), so the starting edge is the right. That's why the constraint uses "start", so that it can work with either LTR or RTL languages. Similarly, constraints use "end" instead of right.

Note: The name of the constraints follows the form **`layout_constraint<Source>_to<Target>Of`**, where **`<Source>`** refers to the current view. **`<Target>`** refers to the edge of the target view that the current view is being constrained to, either the parent container or another view.

Review the EditText attributes

Double check all the `EditText` attributes that you pasted in to make sure it works for how it will be used in your app.

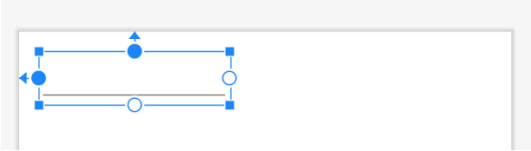
1. Find the `id` attribute, which is set to `@+id/plain_text_input`.
2. Change the `id` attribute to a more appropriate name, `@+id/cost_of_service`.

Note: A resource ID is a unique resource name for the element. When you add a `View` or other resource with the **Layout Editor**, Android Studio automatically assigns resource IDs for them. When you manually type out the XML, you need to explicitly declare the resource ID yourself. New view IDs in your XML file must be defined with the `@+id` prefix, which tells Android Studio to add that ID as a new resource ID.

Choose descriptive names for resources so you know what they refer to, but they should be all in lowercase letters, and multiple words should be separated with an underscore.

When you refer to resource IDs in your app code, use `R.<type>.<name>`; for example, `R.string.roll`.
For `View` IDs, the `<type>` is `id`, for example, `R.id.button`.

3. Look at the layout_height attribute. It's set to wrap_content which means the height will be as tall as the content inside it. That's OK, because there will only be 1 line of text.
4. Look at the layout_width attribute. It's set to match_parent, but you can't set match_parent on a child of ConstraintLayout. Furthermore, the text field doesn't need to be so wide. Set it to be a fixed width of 160dp, which should be plenty of space for the user to enter in a cost of service.



5. Notice the inputType attribute—it's something new. The value of the attribute is "text", which means the user can type in any text characters into the field on screen (alphabetical characters, symbols, etc.)

   android:inputType="text"

   However, you want them to only type numbers into the EditText, because the field represents a monetary value.

6. Erase the word text, but leave the quotes.
7. Start typing number in its place. After typing "n", Android Studio shows a list of possible completions that include “n”.
6. Choose numberDecimal, which limits them to numbers with a decimal point.

   android:inputType="numberDecimal"

   To see other options for input types, see Specify the input method type in the developer documentation.

   There's one more change to make, because it's helpful to display some hint as to what the user should enter into this field.

9. Add a hint attribute to the EditText describing what the user should enter in the field.

   android:hint="Cost of Service"

   You will see the **Design Editor** update, too.

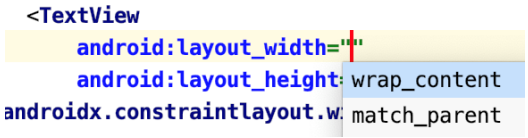
   Nice job! It doesn't do much yet, but you've got a good start and have edited some XML. The XML for your layout should look something like this.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".MainActivity">

    <EditText
        android:id="@+id/cost_of_service"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:hint="Cost of Service"
        android:inputType="numberDecimal"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Add the service question

In this step, you'll add a `TextView` for the question prompt, "How was the service?" Try typing this out without copy/pasting. The suggestions from Android Studio should help you.

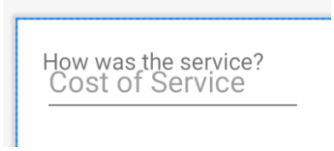
1. After the close of the `EditText` tag, `/>`, add a new line and start typing `<TextView`
2. Select `TextView` from the suggestions, and Android Studio will automatically add the `layout_width` and `layout_height` attributes for the `TextView`.
3. Choose `wrap_content` for both, since you only need the `TextView` to be as big as the text content inside it.



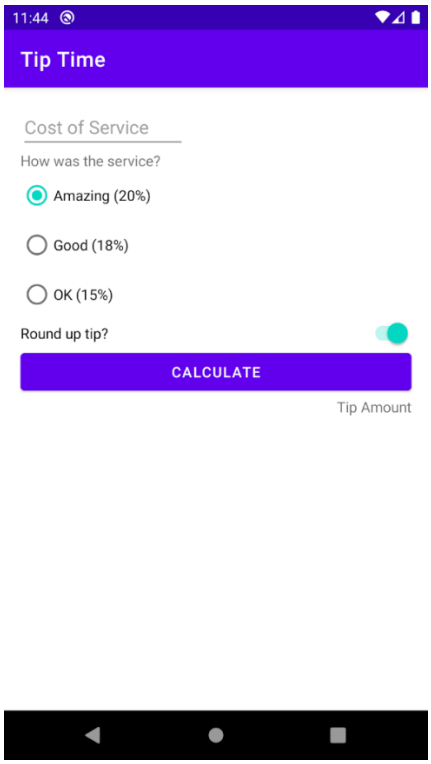
4. Add the `text` attribute with `"How was the service?"`

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="How was the service?"
```

5. Close the tag with `/>`.
6. Notice in the **Design Editor** that the `TextView` overlaps the `EditText`.



That doesn't look right, so you'll add constraints on the `TextView` next. Think about what constraints you need. Where should the `TextView` be positioned horizontally and vertically? You can check the app screenshot to help you.



Vertically, you want the `TextView` to be below the cost of service text field. Horizontally, you want the `TextView` aligned to the starting edge of the parent.

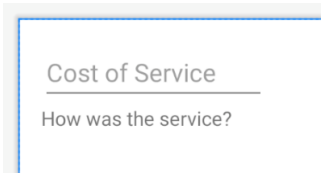
7. Add a horizontal constraint to the `TextView` to constrain its starting edge to the starting edge of the parent.

   app:layout_constraintStart_toStartOf="parent"

8. Add a vertical constraint to the `TextView` to constrain the top edge of the `TextView` to the bottom edge of the cost of service `View`.

   app:layout_constraintTop_toBottomOf="@id/cost_of_service"

   Note that there's no plus in `@id/cost_of_service` because the ID is already defined.



It doesn't look the best, but don't worry about that for now. You just want to make sure all the necessary pieces are on screen and the functionality works. You'll fix how it looks in the following codelabs.

9. Add a resource ID on the `TextView`. You'll need to refer to this view later as you add more views and constrain them to each other.

   android:id="@+id/service_question"

5. Add tip options

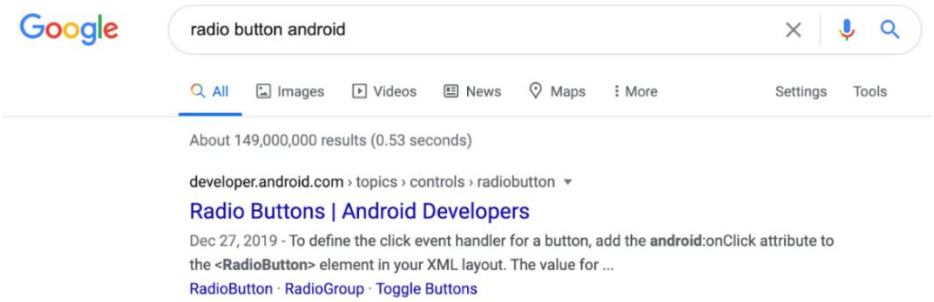
Next you'll add radio buttons for the different tip options the user can choose from.

There should be three options:

- Amazing (20%)
- Good (18%)
- Okay (15%)

If you're not sure how to do this, you can do a Google search. This is a great tool that developers use when they're stuck.

1. Do a Google search for `radio button android`. The top result is a guide from the Android developers site on how to use radio buttons—perfect!

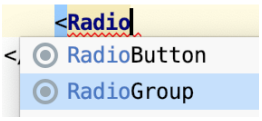


2. Skim through the Radio Buttons guide.

   Reading the description, you can confirm that you can use a `RadioButton` UI element in your layout for each radio button you need. Furthermore, you also need to group the radio buttons within a `RadioGroup` because only one option can be selected at a time.

   There is some XML that looks like it would fit your needs. Read through it and see how `RadioGroup` is the parent view and the `RadioButtons` are child views within it.

3. Go back to your layout in Android Studio to add the `RadioGroup` and `RadioButton` to your app.
4. After the `TextView` element, but still within the `ConstraintLayout`, start typing out `<RadioGroup`. Android Studio will provide helpful suggestions to help you complete your XML.



5. Set the `layout_width` and `layout_height` of the `RadioGroup` to `wrap_content`.
6. Add a resource ID set to `@+id/tip_options`.
7. Close the start tag with `>`.
8. Android Studio adds `</RadioGroup>`. Like the `ConstraintLayout`, the `RadioGroup` element will have other elements inside it, so you might want to move it to its own line.

9. Constrain the `RadioGroup` below the service question (vertically) and to the start of the parent (horizontally).
10. Set the `android:orientation` attribute to `vertical`. If you wanted the `RadioButtons` in a row, you would set the orientation to `horizontal`.

    The XML for the `RadioGroup` should look like this:

```
<RadioGroup
    android:id="@+id/tip_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/service_question">

</RadioGroup>
```

Add RadioButtons

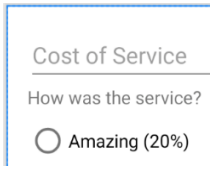
1. After the last attribute of the `RadioGroup`, but before the `</RadioGroup>` end tag, add a `RadioButton`.

```
<RadioGroup
    android:id="@+id/tip_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/service_question">

    <!-- add RadioButtons here -->

</RadioGroup>
```

2. Set the `layout_width` and `layout_height` to `wrap_content`.
3. Assign a resource ID of `@+id/option_twenty_percent` to the `RadioButton`.
4. Set the text to `Amazing (20%)`.

5. Close the tag with `/>`.

```
<RadioGroup
  android:id="@+id/tip_options"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:layout_constraintTop_toBottomOf="@id/service_question"
  app:layout_constraintStart_toStartOf="parent"
  android:orientation="vertical">

  <RadioButton
     android:id="@+id/option_twenty_percent"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Amazing (20%)" />

</RadioGroup>
```

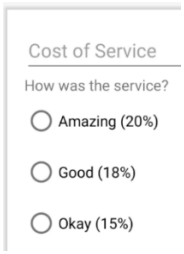


Now that you've added one `RadioButton`, can you modify the XML to add 2 more radio buttons for the `Good (18%)` and `Okay (15%)` options?

This is what the XML for the `RadioGroup` and `RadioButtons` looks like:

```
<RadioGroup
  android:id="@+id/tip_options"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:layout_constraintTop_toBottomOf="@id/service_question"
  app:layout_constraintStart_toStartOf="parent"
  android:orientation="vertical">

  <RadioButton
     android:id="@+id/option_twenty_percent"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Amazing (20%)" />

  <RadioButton
     android:id="@+id/option_eighteen_percent"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Good (18%)" />

  <RadioButton
```

```
    android:id="@+id/option_fifteen_percent"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="OK (15%)" />

</RadioGroup>
```



Add a default selection

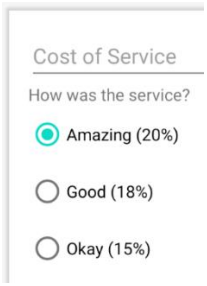
Currently, none of the tip options are selected. It would be nice to select one of the radio button options by default.

There's an attribute on the `RadioGroup` where you can specify which button should be checked initially. It's called `checkedButton`, and you set it to the resource ID of the radio button you want selected

1. On the `RadioGroup`, set the `android:checkedButton` attribute to `@id/option_twenty_percent`.

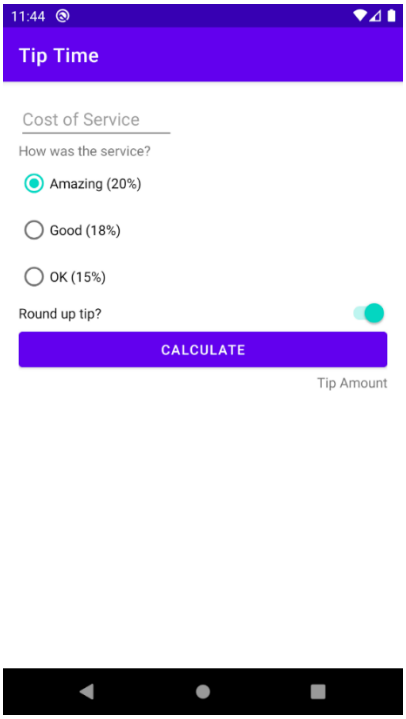
```
<RadioGroup
  android:id="@+id/tip_options"
  android:checkedButton="@id/option_twenty_percent"
  ...
```

Notice in the **Design Editor** that the layout has updated. The 20% tip option is selected by default—cool! Now it's starting to look like a tip calculator!



6. Complete the rest of the layout

You're on the last part of the layout now. You'll add a `Switch`, `Button`, and a `TextView` to display the tip amount.



Add a Switch for rounding up the tip

Next, you'll use a Switch widget to allow the user to select yes or no for rounding up the tip.

You want the `Switch` to be as wide as the parent, so you might think the width should be set to `match_parent`. As noted earlier, you can't set `match_parent` on UI elements in a `ConstraintLayout`. Instead, you need to constrain the start and end edges of the view, and set the width to `0dp`. Setting the width to `0dp` tells the system not to calculate the width, just try to match the constraints that are on the view.

Note: You cannot use **`match_parent`** for any view in a **`ConstraintLayout`**. Instead use **`0dp`**, which means match constraints.

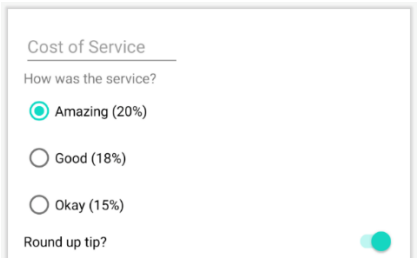
1. Add a `Switch` element after the XML for the `RadioGroup`.
2. As noted above, set the `layout_width` to `0dp`.
3. Set the `layout_height` to `wrap_content`. This will make the `Switch` view as tall as the content inside.
4. Set the `id` attribute to `@+id/round_up_switch`.
5. Set the `text` attribute to `Round up tip?`. This will be used as a label for the `Switch`.
6. Constrain the start edge of the `Switch` to the start edge of the parent, and the end to the end of the parent.
7. Constrain the top of the `Switch` to the bottom of the `tip_options`.
8. Close the tag with `/>`.

It would be nice if the switch was turned on by default, and there's an attribute for that, `android:checked`, where the possible values are `true` (on) or `false` (off).

9. Set the `android:checked` attribute to `true`.

Putting that all together, the XML for the `Switch` element looks like this:

```
<Switch
  android:id="@+id/round_up_switch"
  android:layout_width="0dp"
  android:layout_height="wrap_content"
  android:checked="true"
  android:text="Round up tip?"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toBottomOf="@id/tip_options" />
```



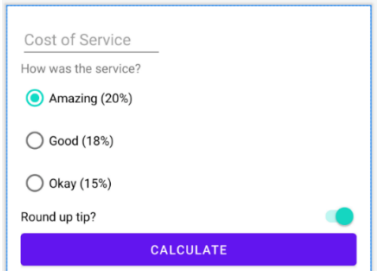
Add the Calculate button

Next you'll add a `Button` so the user can request that the tip be calculated. You want the button to be as wide as the parent, so the horizontal constraints and width are the same as they were for the `Switch`.

1. Add a `Button` after the `Switch`.
2. Set the width to `0dp`, as you did for the `Switch`.
3. Set the height to `wrap_content`.
4. Give it a resource ID of `@+id/calculate_button`, with the text `"Calculate"`.
5. Constrain the top edge of `Button` to the bottom edge of the **Round up tip?** `Switch`.
6. Constrain the start edge to the start edge of the parent and the end edge to the end edge of the parent.
7. Close the tag with `/>`.

Here's what the XML for the **Calculate** `Button` looks like.

```
<Button
  android:id="@+id/calculate_button"
  android:layout_width="0dp"
  android:layout_height="wrap_content"
  android:text="Calculate"
  app:layout_constraintTop_toBottomOf="@id/round_up_switch"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintEnd_toEndOf="parent" />
```



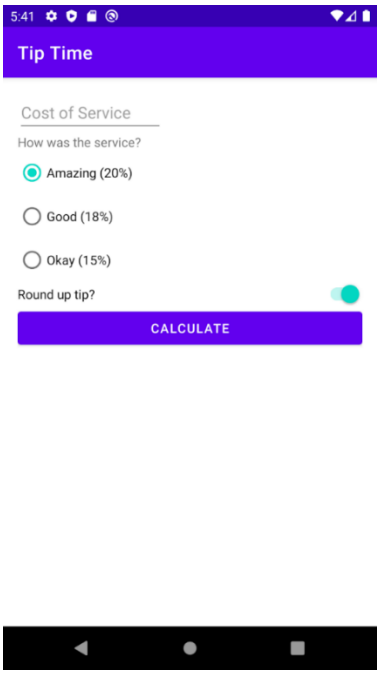
Add tip result

You're almost done with the layout! In this step you'll add a `TextView` for the tip result, putting it below the **Calculate** button, and aligned with the end instead of the start like the other UI elements.

1. Add a `TextView` with a resource ID named `tip_result` and the text `Tip Amount`.
2. Constrain the ending edge of the `TextView` to the ending edge of the parent.
3. Constrain the top edge to the bottom edge of the **Calculate** button.

```
<TextView
   android:id="@+id/tip_result"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintTop_toBottomOf="@id/calculate_button"
   android:text="Tip Amount" />
```

Run the app



Great job, especially if this is your first time working with XML!

Note that the app may not look exactly the same as the screenshot because the templates may have changed in a later version of Android Studio. The **Calculate** button doesn't do anything yet, but you can type in the cost, select the tip percentage, and toggle the option to round up the tip or not. You'll make the **Calculate** button work in the next codelab, so be sure to come back for it!

7. Adopt good coding practices

Extract the strings

You may have noticed the warnings about hard-coded strings. Recall from the earlier codelabs that extracting strings to a resource file makes it easier to translate your app to other languages and to reuse strings. Go through `activity_main.xml` and extract all the string resources.

1. Click on a string; hover over on the yellow light bulb icon that appears, then click on the triangle next to it; choose **Extract String Resource**. The default names for the string resources are fine. If you want, for the tip options you can use `amazing_service`, `good_service`, and `okay_service` to make the names more descriptive.

Now verify the string resources you just added.

2. If the **Project** window isn't showing, click the **Project** tab on the left side of the window.
3. Open **app > res > values > strings.xml** to see all the UI string resources.

```
<resources>
    <string name="app_name">Tip Time</string>
    <string name="cost_of_service">Cost of Service</string>
    <string name="how_was_the_service">How was the service?</string>
    <string name="amazing_service">Amazing (20%)</string>
    <string name="good_service">Good (18%)</string>
    <string name="ok_service">Okay (15%)</string>
    <string name="round_up_tip">Round up tip?</string>
    <string name="calculate">Calculate</string>
    <string name="tip_amount">Tip Amount</string>
</resources>
```

Reformat the XML

Android Studio provides various tools to tidy up your code and make sure it follows recommended coding conventions.

1. In `activity_main.xml`, choose **Edit > Select All**.
2. Choose **Code > Reformat Code**.

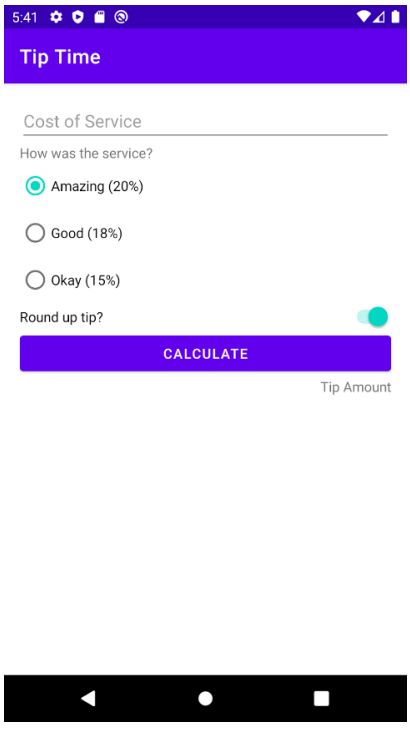
This will make sure the indenting is consistent, and it may reorder some of the XML of UI elements to group things, for example, putting all the `android:` attributes of one element together.

9. Summary

- XML (Extensible Markup Language) is a way of organizing text, made of tags, elements, and attributes.
- Use XML to define the layout of an Android app.
- Use `EditText` to let the user input or edit text.
- An `EditText` can have a hint to tell the user what is expected in that field.
- Specify the `android:inputType` attribute to limit what type of text the user can input into an `EditText` field.
- Make a list of exclusive options with `RadioButtons`, grouped with a `RadioGroup`.
- A `RadioGroup` can be vertical or horizontal, and you can specify which `RadioButton` should be selected initially.
- Use a `Switch` to let the user toggle between two options.
- You can add a label to a `Switch` without using a separate `TextView`.
- Each child of a `ConstraintLayout` needs to have vertical and horizontal constraints.
- Use "start" and "end" constraints to handle both Left to Right (LTR) and Right to Left (RTL) languages.
- Names of the constraint attributes follow the form `layout_constraint<Source>_to<Target>Of`.
- To make a `View` as wide as the `ConstraintLayout` it's in, constrain the start and end to the start and end of the parent, and set the width to 0dp.

2. Starter app overview

The **Tip Time** app from the last codelab has all the UI needed for a tip calculator, but no code to calculate the tip. There's a **Calculate** button, but it doesn't work yet. The **Cost of Service** `EditText` allows the user to enter the cost of the service. A list of `RadioButtons` lets the user select the tip percentage, and a `Switch` allows the user to choose whether the tip should be rounded up or not. The tip amount is displayed in a `TextView`, and finally a **Calculate** `Button` will tell the app to get the data from the other fields and calculate the tip amount. That's where this codelab picks up.



App project structure

An app project in your IDE consists of a number of pieces, including Kotlin code, XML layouts, and other resources like strings and images. Before making changes to the app, it's good to learn your way around.

1. Open the **Tip Time** project in Android Studio.
2. If the **Project** window isn't showing, select the **Project** tab on the left side of Android Studio.
3. If it's not already selected, choose the **Android** view from the dropdown.

- **java** folder for Kotlin files (or Java files)
- `MainActivity` - class where all of the Kotlin code for the tip calculator logic will go
- **res** folder for app resources
- `activity_main.xml` - layout file for your Android app
- `strings.xml` - contains string resources for your Android app
- **Gradle Scripts** folder

Gradle is the automated build system used by Android Studio. Whenever you change code, add a resource, or make other changes to your app, Gradle figures out what has changed and takes the necessary steps to rebuild your app. It also installs your app in the emulator or physical device and controls its execution.

There are other folders and files involved in building your app, but these are the main ones you'll work with for this codelab and the following ones.

Note: You can also write Android apps in the Java programming language.

3. View binding

In order to calculate the tip, your code will need to access all of the UI elements to read the input from the user. You may recall from earlier codelabs that your code needs to find a reference to a `View` like a `Button` or `TextView` before your code can call methods on the `View` or access its attributes. The Android framework provides a method, `findViewById()`, that does exactly what you need—given the ID of a `View`, return a reference to it. This approach works, but as you add more views to your app and the UI becomes more complex, using `findViewById()` can become cumbersome.

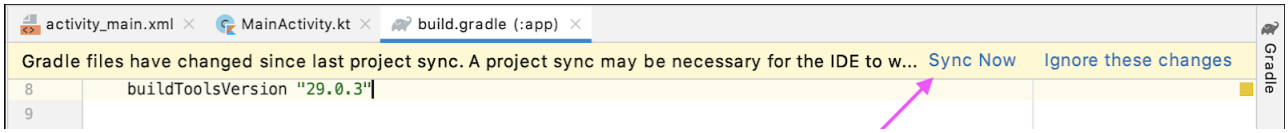
For convenience, Android also provides a feature called *view binding*. With a little more work up front, view binding makes it much easier and faster to call methods on the views in your UI. You'll need to enable view binding for your app in Gradle, and make some code changes.

Enable view binding

1. Open the app's `build.gradle` file (**Gradle Scripts > build.gradle (Module: Tip_Time.app)**)
2. In the `android` section, add the following lines:

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

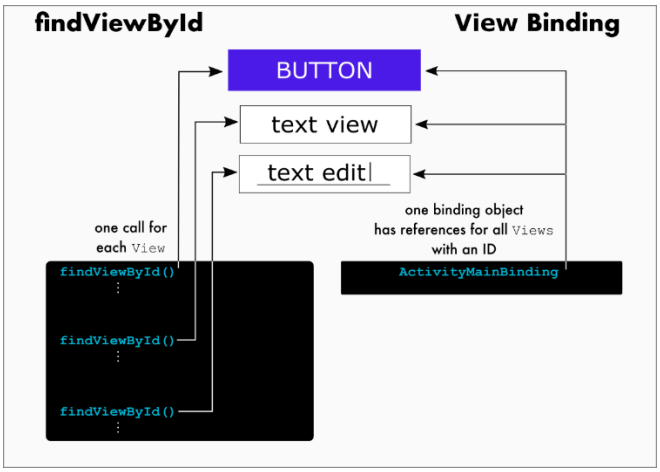
3. Note the message **Gradle files have changed since last project sync.**
4. Press **Sync Now**.



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

Initialize the binding object

In earlier codelabs, you saw the `onCreate()` method in the `MainActivity` class. It's one of the first things called when your app starts and the `MainActivity` is initialized. Instead of calling `findViewById()` for each `View` in your app, you'll create and initialize a binding object once.



1. Open `MainActivity.kt` (**app > java > com.example.tiptime > MainActivity**).
2. Replace all of the existing code for `MainActivity` class with this code to setup the `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. This line declares a top-level variable in the class for the binding object. It's defined at this level because it will be used across multiple methods in `MainActivity` class.

```
lateinit var binding: ActivityMainBinding
```

The `lateinit` keyword is something new. It's a promise that your code will initialize the variable before using it. If you don't, your app will crash.

4. This line initializes the `binding` object which you'll use to access `Views` in the `activity_main.xml` layout.

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

5. Set the content view of the activity. Instead of passing the resource ID of the layout, `R.layout.activity_main`, this specifies the root of the hierarchy of views in your app, `binding.root`.

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

You may recall the idea of parent views and child views; the root connects to all of them.

Now when you need a reference to a `View` in your app, you can get it from the `binding` object instead of calling `findViewById()`. The `binding` object automatically defines references for every `View` in your app that has an ID. Using view binding is so much more concise that often you won't even need to create a variable to hold the reference for a `View`, 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"
```

How cool is that?

Note: The name of the binding class is generated by converting the name of the XML file to camel case and adding the word "Binding" to the end. Similarly, the reference for each view is generated by removing underscores and converting the view name to camel case. For example, **`activity_main.xml`** becomes **`ActivityMainBinding`**, and you can access **`@id/text_view`** as **`binding.textView`**.

4. Calculate the tip

Calculating the tip starts with the user tapping the **Calculate** button. This involves checking the UI to see how much the service cost and the percentage tip that the user wants to leave. Using this information you calculate the total amount of the service charge and display the amount of the tip.

Add click listener to the button

The first step is to add a click listener to specify what the **Calculate** button should do when the user taps it.

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

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

2. Still inside `MainActivity` class but outside `onCreate()`, add a helper method called `calculateTip()`.

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

This is where you'll add the code to check 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() {

    }
}
```

Get the cost of service

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

1. First, get the text for the cost of service. In the `calculateTip()` method, get the text attribute of the **Cost of Service** `EditText`, and assign it to a variable called `stringInTextField`. Remember that you can access the UI element using the `binding` object, and that you can reference the UI element based on its resource ID name in camel case.

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

Notice the `.text` on the end. The first part, `binding.costOfService`, references the UI element for the cost of service. Adding `.text` on the end says to take that result (an `EditText` object), and get the `text` property from it. This is known as *chaining*, and is a very common pattern in Kotlin.

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

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

That doesn't work, though—`toDouble()` needs to be called on a `String`. It turns out that the `text` attribute of an `EditText` is an `Editable`, because it represents text that can be changed. Fortunately, you can convert an `Editable` to a `String` by calling `toString()` on it.

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

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

Now `stringInTextField.toDouble()` will work.

At this point, the `calculateTip()` method should like this:

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

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 `checkedRadioButtonId` attribute of the `tipOptions` `RadioGroup`, and assign it to a variable called `selectedId`.

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

Now you know which `RadioButton` was selected, one of `R.id.option_twenty_percent`, `R.id.option_eighteen_percent`, or `R.id.fifteen_percent`, but you need the corresponding percentage. You could write a series of `if/else` statements, but it's much easier to use a `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 `calculateTip()` method should like this:

```
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
    }
}
```

Calculate the tip and round it up

Now that you have the cost of service and the tip percentage, calculating the tip is straightforward: the tip is the cost times the tip percentage, tip = cost of service * tip percentage. Optionally, that value may be rounded up.

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

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

Note the use of `var` instead of `val`. This is because you may need to round up the value if the user selected that option, so the value might change.

```
For a `Switch` element, you can check the `isChecked` attribute to see if the switch is "on".
```

2. Assign the `isChecked` attribute of the round up switch to a variable called `roundUp`.

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

The term rounding means adjusting a decimal number up or down to the closest integer value, but in this case, you only want to round up, or find the ceiling. You can use the `ceil()` function to do that. There are several functions with that name, but the one you want is defined in `kotlin.math`. You could add an `import` statement, but in this case it's simpler to just tell Android Studio which you mean by using `kotlin.math.ceil()`.

If there were several math functions you wanted to use, it would be easier to add an `import` statement.

3. Add an `if` statement that assigns the ceiling of the tip to the `tip` variable if `roundUp` is true.

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

Format the tip

Your app is almost working. You've calculated the tip, now you just need to format it and display it.

As you might expect, Kotlin provides methods for formatting different types of numbers. But the tip amount is a little different—it represents a currency value. Different countries use different currencies, and have different rules for formatting decimal numbers. For example, in U.S. dollars, 1234.56 would be formatted as $1,234.56, but in Euros, it would be formatted €1.234,56. Fortunately, the Android framework provides methods for formatting numbers as currency, so you don't need to know all the possibilities. The system automatically formats currency based on the language and other settings that the user has chosen on their phone. Read more about NumberFormat in the Android developer documentation.

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

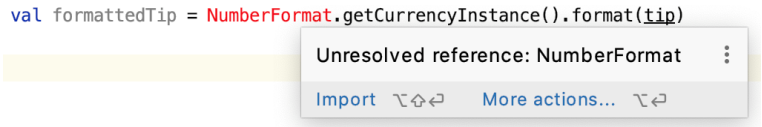
```
NumberFormat.getCurrencyInstance()
```

This gives you a number formatter you can use to format numbers as currency.

2. Using the number formatter, chain a call to the `format()` method with the `tip`, and assign the result to a variable called `formattedTip`.

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

3. Notice that `NumberFormat` is drawn in red. This is because Android Studio can't automatically figure which version of `NumberFormat` you mean.
4. Hover the pointer over `NumberFormat`, and choose **Import** in the popup that appears.



5. In the list of possible imports, choose **NumberFormat (java.text)**. Android Studio adds an `import` statement at the top of the `MainActivity` file, and `NumberFormat` is no longer red.

Display the tip

Now it's time to display the tip in the tip amount `TextView` element of your app. You could just assign `formattedTip` to the `text` attribute, but it would be nice to label what the amount represents. In the U.S. with English, you might have it display **Tip Amount: $12.34**, but in other languages the number might need to appear at the beginning or even the middle of the string. The Android framework provides a mechanism for this called *string parameters*, so someone translating your app can change where the number appears if needed.

1. Open `strings.xml` (**app > res > values > strings.xml**)
2. Change the `tip_amount` string from `Tip Amount` to `Tip Amount: %s`.

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

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

3. Now set the text of the `tipResult`. Back in the `calculateTip()` method in `MainActivity.kt`, call `getString(R.string.tip_amount, formattedTip)` and assign that to the `text` attribute of the tip result `TextView`.

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

At this point, the `calculateTip()` method should like this:

```
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 there. When developing your app (and viewing the preview), it's useful to have a placeholder for that `TextView`.

4. Open `activity_main.xml` (**app > res > layout > activity_main.xml**)
5. Find the `tip_result` `TextView`.
6. Remove the line with the `android:text` attribute.

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

7. Add a line for the `tools:text` attribute set to `Tip Amount: $10`.

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

Because this is just a placeholder, you don't need to extract the string into a resource. It won't appear when you run your app.

8. Note that the tools text appears in the **Layout Editor**.
9. Run your app. Enter an amount for the cost and select some options, then press the **Calculate** button.

5. Test and debug

You've run the app at various steps to make sure it does what you want, but now it's time for some additional testing.

For now, think about how the information moves through your app in the `calculateTip()` method, and what could go wrong at each step.

For example, what would happen in this line:

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

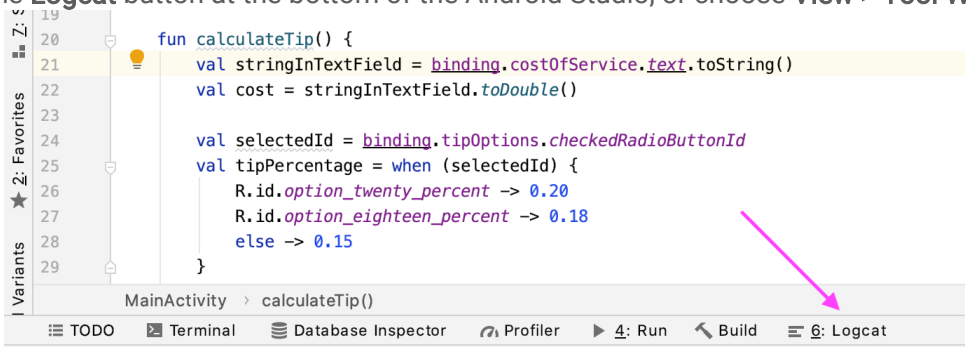
if `stringInTextField` didn't represent a number? What would happen if the user didn't enter any text and `stringInTextField` was empty?

1. Run your app in the emulator, but instead of using **Run > Run 'app'**, use **Run > Debug 'app'**.
2. Try some different combinations of cost, tip amount, and rounding up the tip or not and verify that you get the expected result for each case when you tap **Calculate**.
3. Now try deleting all the text in the **Cost of Service** field and tap **Calculate**. Uh, oh...your program crashes.

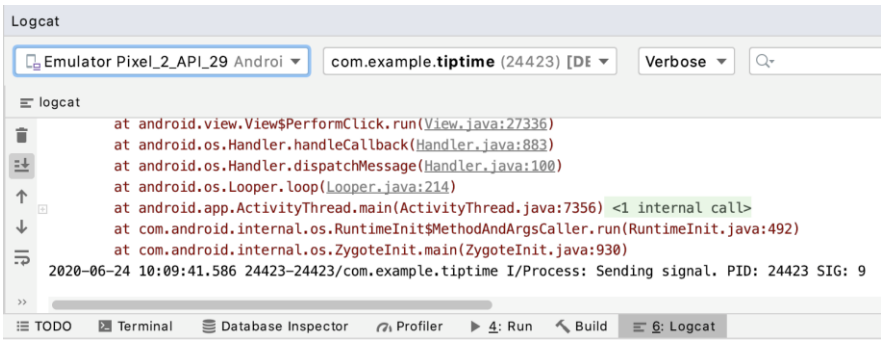
Debug the crash

The first step in dealing with a bug is to find out what happened. Android Studio keeps a log of what is happening in the system which you can use to find out what went wrong.

1. Press the **Logcat** button at the bottom of the Android Studio, or choose **View > Tool Windows > Logcat** in the menus.



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



The text is a *stack trace*, a list of which methods were being called when the crash occurred.

3. Scroll upward in the **Logcat** text until you find a line which 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 downward until you find the line with `NumberFormatException`.

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

   To the right it says `empty String`. The type of the exception tells you it was something to do with a number format, and the rest you the basis of the problem: an empty `String` was found when it should have been a `String` with a value.

5. Continue reading downward, and you'll see some calls to `parseDouble()`.
6. Below those calls, find the line with `calculateTip`. Note that it includes your `MainActivity` class, too.

   ```
   at com.example.tiptime.MainActivity.calculateTip(MainActivity.kt:20)
   ```

7. Look carefully at that line, and you can see exactly where in your code the call was made, line 20 in `MainActivity.kt`. (If you typed your code differently, it may be a different number.) That line converts the `String` to a `Double` and assigns the result to the `cost` variable.

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

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

   An *exception* is the system's way of saying there is a problem. In this case, the problem is that `toDouble()` couldn't convert the empty `String` into a `Double`. Even though the `EditText` has `inputType=numberDecimal`, it's still possible to enter some values that `toDouble()` can't handle, like an empty string.

Learn about null

Calling `toDouble()` on a string that is empty or a string that doesn't represent a valid decimal number doesn't work. Fortunately Kotlin also provides a method called `toDoubleOrNull()` which handles these problems. It returns a decimal number if it can, or it returns `null` if there's a problem.

Null is a special value that means "no value". It's different from a `Double` having a value of `0.0` or an empty `String` with zero characters, `""`. `Null` means there is no value, no `Double` or no `String`. Many methods expect a value and may not know how to handle `null` and will stop, which means the app crashes, so Kotlin tries to limit where `null` is used. You'll learn more about this in future lessons.

Your app can check for `null` being returned from `toDoubleOrNull()` and do things differently in that case so the app doesn't crash.

1. In `calculateTip()`, change the line that declares the `cost` variable 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 to return from the method. The `return` instruction means exit the method without executing the rest of the instructions. If the method needed to return a value, you would specify it with a `return` instruction with an expression.

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

3. Run your app again.
4. With no text in the **Cost of Service** field, tap **Calculate**. This time your app doesn't crash! Good job—you found and fixed the bug!

Handle another case

Not all bugs will cause your app to crash—sometimes the results might be potentially confusing to the user.

Here's another case to consider. What will happen if the user:

1. enters a valid amount for the cost
2. taps **Calculate** to calculate the tip
3. deletes the cost
4. taps **Calculate** again?

The first time the tip will be calculated and displayed as expected. The second time, the `calculateTip()` method will return early because of the check you just added, but the app will still show the previous tip amount. This might be confusing to the user, so add some code to clear the tip amount if there's a problem.

1. Confirm this problem is what happens by entering a valid cost and tapping **Calculate**, then deleting the text, and tapping **Calculate** again. The first tip value should still be displayed.
2. Inside the `if` just added, before the `return` statement, add a line to set the `text` attribute of `tipResult` to an empty string.

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

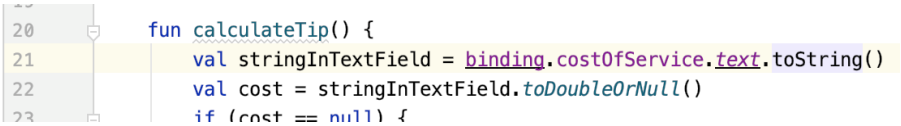
This will clear the tip amount before returning from `calculateTip()`.

3. Run your app again, and try the above case. The first tip value should go away when you tap **Calculate** the second time.

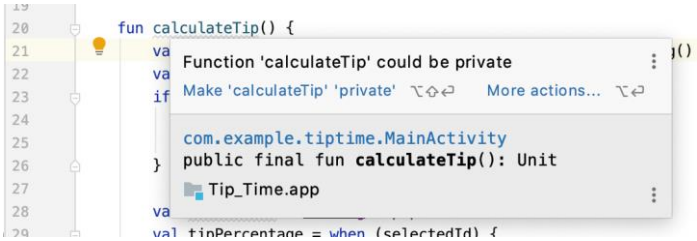
6. Adopt good coding practices

Your tip calculator works now, but you can make code a little better and make it easier to work with in the future by adopting good coding practices.

1. Open `MainActivity.kt` (**app > java > com.example.tiptime > MainActivity**)
2. Look at the beginning of the `calculateTip()` method, and you might see that it is underlined with a wavy, grey line.



3. Hover the pointer over `calculateTip()`, you'll see a message, **Function 'calculateTip' could be private** with a suggestion below to **Make 'calculateTip' 'private'**.



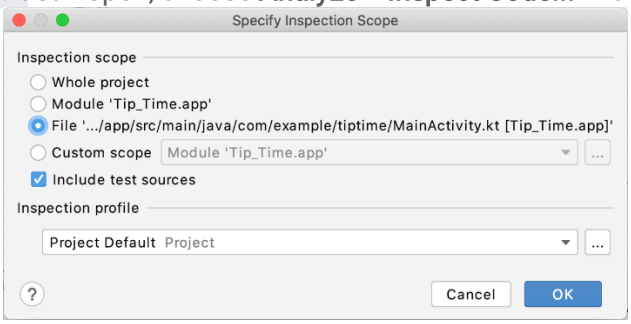
Recall from earlier codelabs that `private` means the method or variable is only visible to code within that class, in this case, `MainActivity` class. There's no reason for code outside `MainActivity` to call `calculateTip()`, so you can safely make it `private`.

4. Choose **Make 'calculateTip' 'private'**, or add the `private` keyword before `fun calculateTip()`. The grey line under `calculateTip()` disappears.

Inspect the code

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

1. With `MainActivity.kt` still open, choose **Analyze > Inspect Code...** in the menus. A dialog box called **Specify Inspection Scope** appears.



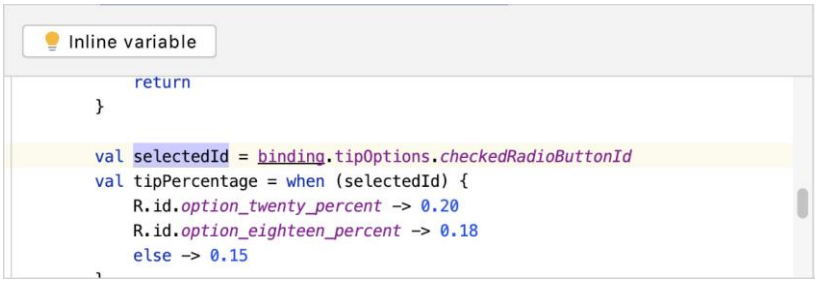
2. Choose the option that starts with **File** and press **OK**. This will limit the inspection to just `MainActivity.kt`.
3. A window with **Inspection Results** appears at the bottom.
4. Click on the grey triangles next to **Kotlin** and then next to **Style issues** until you see two messages. The first says **Class member can have 'private' visibility**.

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

6. Press the **Make 'binding' 'private'** button. Android Studio removes the issue from the **Inspection Results**.
7. If you look at `binding` in your code, you'll see that Android Studio has added the keyword `private` before the declaration.

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

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



9. If you look at your code, you'll see that `selectedId` is only used twice: first in the highlighted line where it is assigned the value of `tipOptions.checkedRadioButtonId`, and in the next line in the `when`.
10. Press the **Inline variable** button. Android Studio replaces `selectedId` in the `when` expression with the value that was assigned in the line before. And then it removes the previous line completely, because it's no longer needed!

```
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 pretty cool! Your code has one less line, and one less variable.

Removing unnecessary variables

Android Studio doesn't have any more results from the inspection. However, if you look at your code closely, you'll see a similar pattern to what you just changed: the `roundUp` variable is assigned on one line, used on the next line, and not used anywhere else.

1. Copy the expression to the right of the `=` from the line where `roundUp` is assigned.

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

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

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

3. Delete the line with `roundUp`, because it isn't needed anymore.

You just did the same thing that Android Studio helped you do with the `selectedId` variable. Again your code has one less line and one less variable. These are small changes, but they help make your code more concise and more readable.

(Optional) Eliminate repetitive code

Once your app is running correctly, you can look for other opportunities to clean up your code and make it more concise. For example, when you don't enter a value in cost of service, the app updates `tipResult` to be an empty string `""`. When there is a value, you use `NumberFormat` to format it. This functionality can be applied elsewhere in the app, for example, to display a tip of `0.0` instead of the empty string.

To reduce duplication of very similar code, you can extract these two lines of code to their own function. This helper function can take as input a tip amount as a `Double`, formats it, and updates the `tipResult` `TextView` on screen.

1. Identify the duplicated code in `MainActivity.kt`. These lines of code could be used multiple times in the `calculateTip()` function, once for the `0.0` case, and once for the general case.

```
val formattedTip = NumberFormat.getCurrencyInstance().format(0.0)
binding.tipResult.text = getString(R.string.tip_amount, formattedTip)
```

2. Move the duplicated code to its own function. One change to the code is to take a parameter tip so that the code works in multiple places.

```
private fun displayTip(tip : Double) {
    val formattedTip = NumberFormat.getCurrencyInstance().format(tip)
    binding.tipResult.text = getString(R.string.tip_amount, formattedTip)
}
```

3. Update your `calculateTip()` function to use the `displayTip()` helper function and check for `0.0`, too.

MainActivity.kt

```
private fun calculateTip() {
    ...

    // If the cost is null or 0, then display 0 tip and exit this function early.
    if (cost == null || cost == 0.0) {
        displayTip(0.0)
        return
    }

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

    // Display the formatted tip value on screen
    displayTip(tip)
}
```

Note

Even though the app is functioning now, it's not ready for production yet. You need to do more testing. And you need to add some visual polish and follow Material Design guidelines. You'll also learn to change the app theme and app icon in the following codelabs.

8. Summary

- View binding lets you more easily write code that interacts with the UI elements in your app
- The `Double` data type in Kotlin can store a decimal number
- Use the `checkRadioButtonId` attribute of a `RadioGroup` to find which `RadioButton` is selected
- Use `NumberFormat.getCurrencyInstance()` to get a formatter to use for formatting numbers as currency
- You can use string parameters like `%s` to create dynamic strings that can still be easily translated into other languages
- Testing is important!
- You can use **Logcat** in Android Studio to troubleshoot problems like the app crashing
- A stack trace shows a list of methods that were called. This can be useful if the code generates an exception.
- Exceptions indicate a problem that code didn't expect
- `Null` means "no value"
- Not all code can handle `null` values, so be careful using it
- Use **Analyze > Inspect Code** for suggestions to improve your code

9. More Codelabs to improve your UI

Great job on getting the tip calculator to work! You'll notice that there are still ways to improve the UI to make the app look more polished. If you're interested, check out these additional codelabs to learn more about how to change the app theme and app icon, as well as how to follow best practices from the Material Design guidelines for the Tip Time app!

- Change the app color theme
- Change the app icon
- Create a more polished UI with Material Design

UNIT 2 : MODULE 2

1. Before you begin

Material is a design system created by Google to help developers build high-quality digital experiences for Android and other platforms. The full Material system includes design guidelines on visual, motion, and interaction design for your app, but this codelab will focus on changing the color theme for your Android app.

The codelab uses the Empty Activity app template, but you can use whatever Android app you're working on. If you're taking this as part of the Android Basics course, you can use the Tip Time app.

What you'll learn

- How to select effective colors for your app according to Material Design principles
- How to set colors as part of your app theme
- The RGB components of a color
- How to apply a style to a `View`
- Change the look of an app using a Theme
- Understand the importance of color contrast

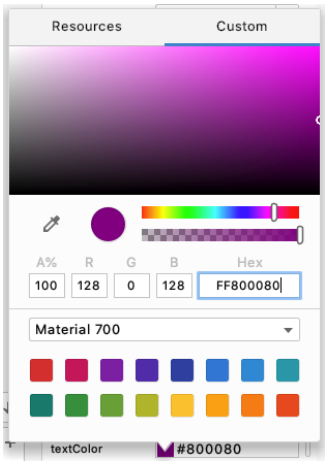
2. Design and color

Material Design

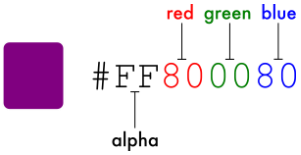
Material Design is inspired by the physical world and its textures, including how objects reflect light and cast shadows. It provides guidelines on how to build your app UI in a readable, attractive, and consistent manner. Material Theming allows you to adapt Material Design for your app, with guidance on customizing colors, typography, and shapes. Material Design comes with a built-in, baseline theme that can be used as-is. You can then customize it as little, or as much, as you like to make Material work for your app.

A bit about color

Color is all around us, both in the real world and the digital realm. The first thing to know is not everyone sees color the same way, so it is important to choose colors for your app so users can effectively use your app. Choosing colors with enough color contrast is just one part of building more accessible apps.



A Color can be represented as 3 hexadecimal numbers, #00-#FF (0-255), representing the red, green, and blue (RGB) components of that color. The higher the number, the more of that component there is.



Note that a color can also be defined including an alpha value #00-#FF, which represents the transparency (#00 = 0% = fully transparent, #FF = 100% = fully opaque). When included, the alpha value is the first of 4 hexadecimal numbers (ARGB). If an alpha value is not included, it is assumed to be #FF = 100% opaque.

You don't need to generate the hexadecimal numbers yourself, though. There are tools available to help you pick colors which will generate the numbers for you.

Some examples you may have seen in the `colors.xml` file of your Android app include 100% black (R=#00, G=#00, B=#00) and 100% white (R=#FF, G=#FF, B=#FF):

```
<color name="black">#FF000000</color>
<color name="white">#FFFFFFFF</color>
```

3. Themes

A *style* can specify attributes for a `View`, such as font color, font size, background color, and much more.

A *theme* is a collection of styles that's applied to an entire app, activity, or view hierarchy—not just an individual `View`. When you apply a theme to an app, activity, view, or view group, the theme is applied to that element and all of its children. Themes can also apply styles to non-view elements, such as the status bar and window background.

Add Material dependencies to your project

This app uses Material Components and themes from the Material Design Components library. Older versions of Android Studio (prior to version 4.1) do not have the required dependency included by default for projects created from the templates. Add this Material library dependency to your project, if it's not already present.

1. In the **Project** window, open **Gradle Scripts** > **build.gradle (Module: app)**. There are two `build.gradle` files, one for the project and one for the app. Make sure you select the correct one in the app module.
2. In `build.gradle`, locate the `dependencies` block near the bottom of the file.

3. Locate the line that reads

   ```
   implementation 'androidx.appcompat:appcompat:1.2.0'
   ```

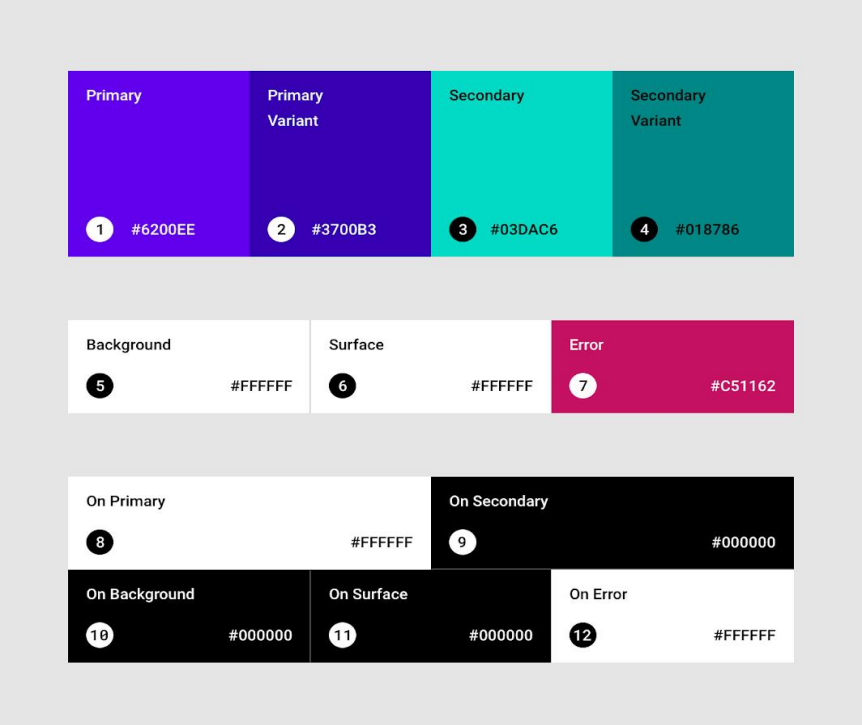
   and change it to

   ```
   implementation 'com.google.android.material:material:1.2.0'
   ```

4. Re-sync the project by clicking the **Sync Now** prompt that appears when you edit the gradle file.

Learn about theme colors

Different parts of the UI for Android apps use different colors. To help you use color in a meaningful way in your app, and apply it consistently throughout, the theme system groups colors into 12 named attributes related to color to be used by text, icons, and more. Your theme doesn't need to specify all of them; you will be choosing the primary and secondary colors, as well as the colors for text and icons drawn on those colors.



The "On" colors are used for text and icons drawn on the different surfaces.

#	Name	Theme Attribute
1	Primary	`colorPrimary`
2	Primary Variant	`colorPrimaryVariant`
3	Secondary	`colorSecondary`
4	Secondary Variant	`colorSecondaryVariant`
5	Background	`colorBackground`

6	Surface	`colorSurface`
7	Error	`colorError`
8	On Primary	`colorOnPrimary`
9	On Secondary	`colorOnSecondary`
10	On Background	`colorOnBackground`
11	On Surface	`colorOnSurface`
12	On Error	`colorOnError`

Take a look at the colors defined in the default theme.

1. In Android Studio, open `themes.xml` (**app > res > values > themes.xml**).
2. Notice the theme name, `Theme.TipTime`, which is based on your app name.

```
<style name="Theme.TipTime" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
```

3. Note that line of XML also specifies a parent theme, `Theme.MaterialComponents.DayNight.DarkActionBar`. `DayNight` is a predefined theme in the Material Components library. `DarkActionBar` means that the action bar uses a dark color. Just as a class inherits attributes from its parent class, a theme inherits attributes from its parent theme.

   Note: Theme color attributes that aren't defined in a theme will use the color from the parent theme.

4. Look through the items in the file, and note that the names are similar to those in the diagram above: `colorPrimary`, `colorSecondary`, and so on. If you are using Android Studio 4.0, copy the following text into your `themes.xml` file.

`themes.xml`

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.TipTime"
parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryDark">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor"
tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```

Not all color theme attributes are defined. Colors that are not defined will inherit the color from the parent theme.

5. Also notice that Android Studio draws a small sample of the color in the left margin.

```
<!-- Primary brand color. -->
<item name="colorPrimary">@color/purple_500</item>
<item name="colorPrimaryVariant">@color/purple_700</item>
<item name="colorOnPrimary">@color/white</item>
<!-- Secondary brand color. -->
<item name="colorSecondary">@color/teal_200</item>
<item name="colorSecondaryVariant">@color/teal_700</item>
<item name="colorOnSecondary">@color/black</item>
```

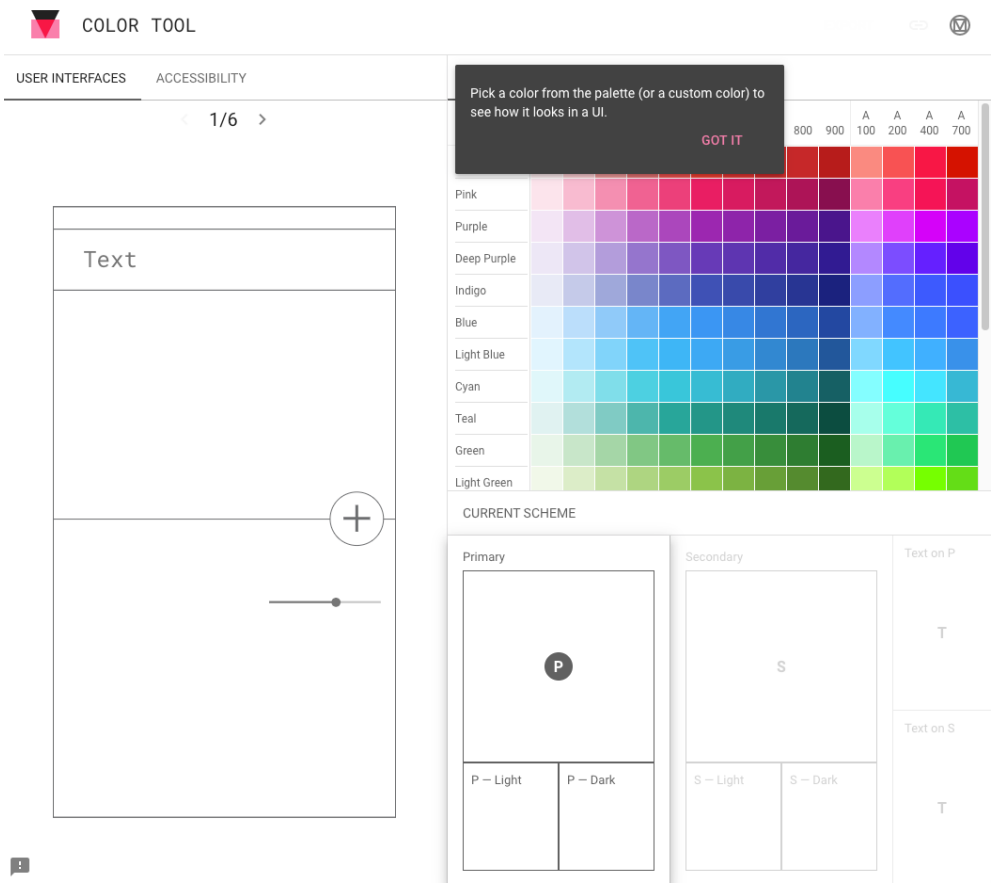
6. Finally, note that the colors are specified as color resources, for example, `@color/purple_500`, rather than using an RGB value directly.

7. Open `colors.xml` (**app > res > values > colors.xml**), and you will see the hex values for each color resource. Recall that the leading `#FF` is the alpha value, and means the color is 100% opaque.
`colors.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

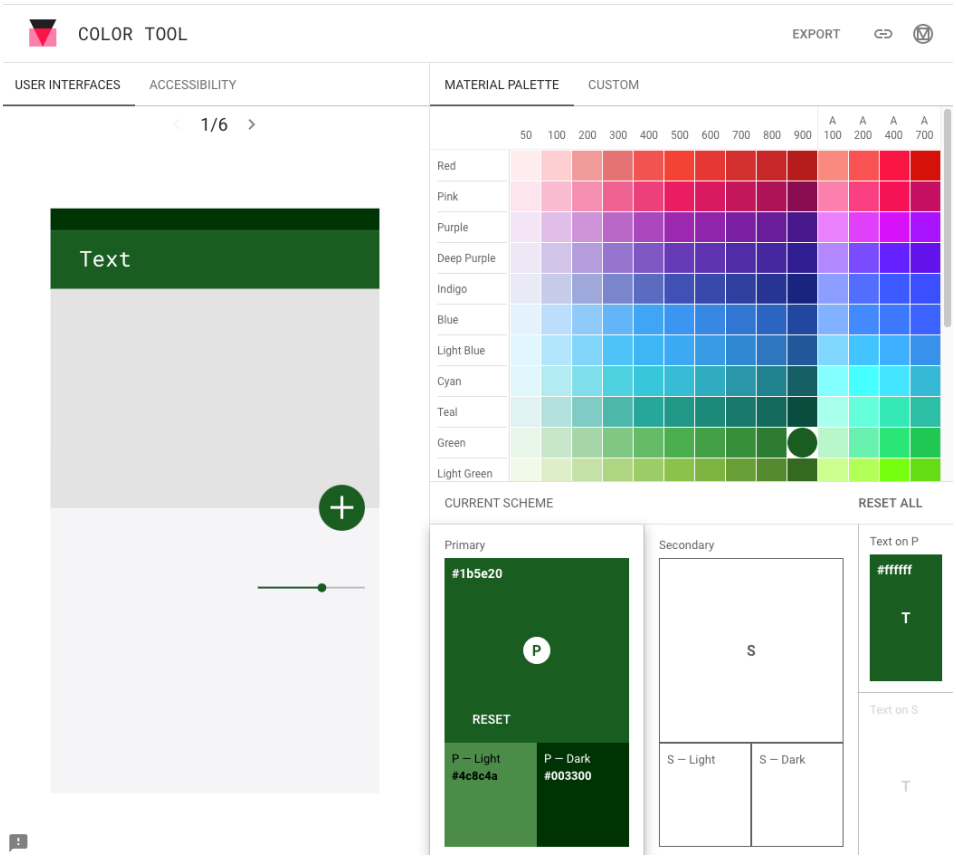
4. Pick app theme colors

Now that you have some idea of the theme attributes, it's time to pick some colors! The easiest way to do this is with the Color Tool, a web-based app provided by the Material team. The tool provides a palette of predefined colors, and lets you easily see how they look when used by different UI elements.



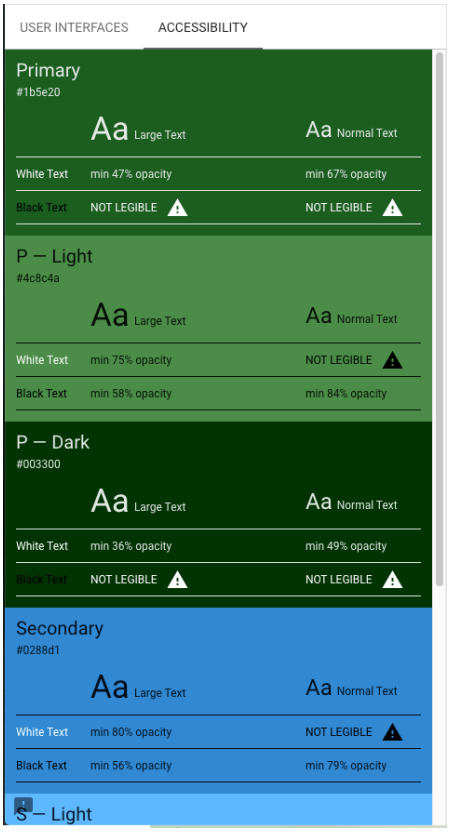
Pick some colors

1. Start by selecting a **Primary** color (`colorPrimary`), for example, **Green 900**.
   The color tool shows what that will look like in an app mockup, and also selects **Light** and **Dark** variants.



2. Tap on the **Secondary** section and choose a secondary color (`colorSecondary`) that you like, for example, **Light Blue 700**.
   The color shows what that will look like in the app mockup, and again selects **Light** and **Dark** variants.
3. Note that the app mockup includes 6 different mock "screens". Look at what your color choices look like on the different screens by tapping the arrows above the mockup.

4. The color tool also has an **Accessibility** tab to let you know if your colors are clear enough to read when used with black or white text. Part of making your app more accessible is ensuring the color contrast is high enough: 4.5:1 or higher for small text, and 3.0:1 or higher for larger text. Read more about color contrast.



5. For `primaryColorVariant` and `secondaryColorVariant`, you can pick either the light or dark variant suggested.

Note: You can also use the Material palette generator to select a secondary color. You can choose a primary color, and it will suggest colors that are complementary, analogous, or triadic.

Add the colors to your app

Defining resources for colors makes it easier to consistently reuse the same colors in different parts of your app.

1. In Android Studio, open `colors.xml` (**app > res > values > colors.xml**)
2. After the existing colors, define a color resource named `green` using the value selected above, `#1B5E20`.

```
<color name="green">#1B5E20</color>
```

3. Continue defining resources for the other colors. Most of these are from the color tool. Note that the values for `green_light` and `blue_light` are different from what the tool shows; you'll use them in a later step.

Green	#1B5E20

green_dark	#003300
green_light	#A5D6A7
Blue	#0288D1
blue_dark	#005B9F
blue_light	#81D4FA

There are already color resources defined for black and white, so you don't need to define those.

Use the colors in your theme

Now that you have names for the colors you selected, it's time to use them in your theme.

1. Open `themes.xml` (**app > res > values > themes > themes.xml**)
2. Change `colorPrimary` to the primary color you selected, `@color/green`.
3. Change `colorPrimaryVariant` to `@color/green_dark`.
4. Change `colorSecondary` to `@color/blue`.
5. Change `colorSecondaryVariant` to `@color/blue_dark`.
6. Check that **Text on P** and **Text on S** are still white (`#FFFFFF`) and black (`#000000`). If you're using the color tool on your own and select other colors, you may need to define additional color resources.
7. Run your app in the emulator or on a device and see what your app looks like with the new theme.

5. Dark theme

The app template included a default light theme, and also included a dark theme variant. A dark theme uses darker, more subdued colors, and:

- Can reduce power usage by a significant amount (depending on the device's screen technology).
- Improves visibility for users with low vision and those who are sensitive to bright light.
- Makes it easier for anyone to use a device in a low-light environment.

Choosing colors for dark theme

Colors for a dark theme still need to be readable. Dark themes use a dark surface color, with limited color accents. To help ensure readability, the primary colors are usually less saturated versions of the primary colors for the light theme.

To provide more flexibility and usability in a dark theme, it's recommended to use lighter tones (200-50) in a dark theme, rather than your default color theme (saturated tones ranging from 900-500). Earlier you picked green 200 and light blue 200 as light colors. For your app, you'll use the light colors as the main colors, and the primary colors as the variants.

Update the dark version of your theme

1. Open `themes.xml (night)` (**app > res > values > themes > themes.xml (night)**)

   Note: This **`themes.xml`** file is different from the previous **`themes.xml`** file This file contains the dark theme version of the theme. The resources in this file will be used when **Dark theme** on the device is on.

2. Change `colorPrimary` to the light variant of the primary color you selected, `@color/green_light`.
3. Change `colorPrimaryVariant` to `@color/green`.
4. Change `colorSecondary` to `@color/blue_light`.
5. Change `colorSecondaryVariant` to `@color/blue_light`.

   When you're done, your `themes.xml (night)` file should look like this:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Application theme for dark theme. -->
    <style name="Theme.TipTime"
parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/green_light</item>
        <item name="colorPrimaryVariant">@color/green</item>
        <item name="colorOnPrimary">@color/black</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/blue_light</item>
        <item name="colorSecondaryVariant">@color/blue_light</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor"
tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```

6. At this point, the original colors defined in `colors.xml`, for example, `purple_200`, aren't used anymore so you can delete them.

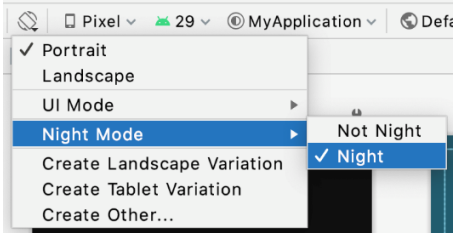
Try dark theme

You can see what your theme looks like in dark mode by enabling dark mode on your device.

Note: Dark theme requires a device or emulator running API 28 (Android 9) or API 29 (Android 10) or higher.

Congratulations! You've successfully made a new app theme, with both a light theme and dark theme.

Note: You can also preview what your theme looks like in the Android Studio **Design Editor**. The **Orientation for Preview** menu contains settings for **Night** or **Not Night**.



7. Summary

- Use the Material Color Tool to select colors for your app theme.
- Alternatively, you can use the Material palette generator to help select a color palette.
- Declare color resources in the `colors.xml` file to make it easier to reuse them.
- Dark theme can reduce power usage and make your app easier to read in low light.

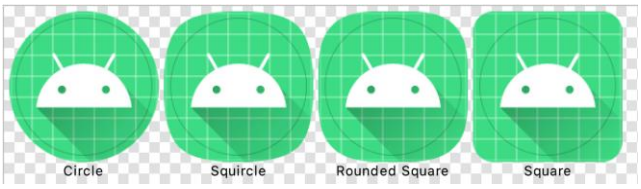
1. Before you begin

An app icon is an important way to distinguish your app. It also appears in a number of places including the Home screen, All Apps screen, and the Settings app.

You may also hear the app icon referred to as a launcher icon. Launcher refers to the experience when you hit the Home button on an Android device to view and organize your apps, add widgets and shortcuts, and more.

If you've tried different Android devices, you may have noticed that the Launcher experience can look different depending on the device manufacturer. Sometimes device manufacturers will create a custom Launcher experience that's signature to their brand. As part of that, different manufacturers may show app icons in a different shape than the circular icon shape shown above.

They could display all the app icons in a square shape, rounded square, or squircle (between a square and circle) for example.



Regardless of the shape the device manufacturers choose, the goal is for all the app icons on a single device to have a uniform shape for a more consistent user experience.

That's why the Android platform introduced support for adaptive icons (as of API level 26). By implementing an adaptive icon for your app, your app will be able to accommodate a large range of devices by displaying a high-quality app icon appropriately.

This codelab will provide you with image source files for a Tip Calculator launcher icon to practice with. You will use a tool in Android Studio called Image Asset Studio to generate all versions of the launcher icons needed. Afterwards, you can take what you learned and apply it to changing the app icon for other apps!

Prerequisites

- Able to navigate the files of a basic Android project, including the resource files
- Able to install an Android app from Android Studio on the emulator or a physical device

What you'll learn

- How to change the launcher icon of an app
- How to use Image Asset Studio in Android Studio to generate launcher icon assets
- What is an adaptive icon and why it's made up of two layers

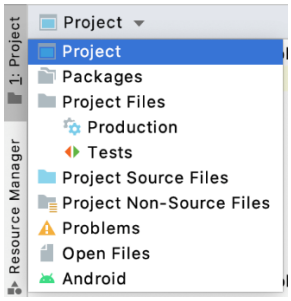
3. Launcher Icons

The goal is for your launcher icon to look fantastic (crisp and clear) regardless of the device model or screen density. Specifically, screen pixel density refers to how many pixels per inch (or dpi, dots per inch) are on the screen. For a medium-density device (mdpi), there are 160 dots per inch on the screen while an extra-extra-extra-high-density device (xxxhdpi) has 640 dots per inch on the screen.

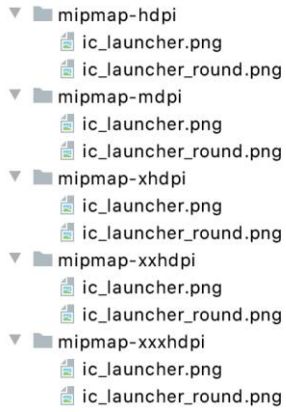
To account for devices across a range of screen densities, you'll need to provide different versions of your app icon.

Explore launcher icon files

1. To see what this looks like, open up your project in Android Studio. If your app was started from a template, then you should have default launcher icons that are already provided by Android Studio.
2. In the **Project window**, switch to the **Project** view. This will show you the files in your project according to the file structure of how those files are saved on your computer.



3. Navigate to the resources directory (**app > src > main > res**) and expand out some of the `mipmap` folders. These `mipmap` folders are where you should put the launcher icon assets for your Android app.



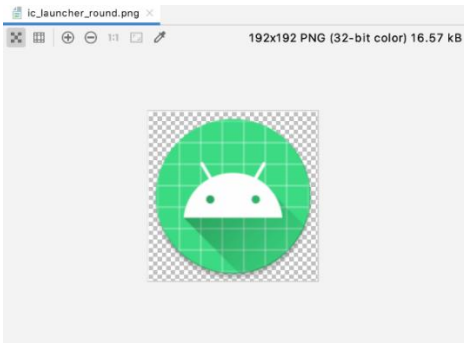
`mdpi`, `hdpi`, `xhdpi`, etc.. are density qualifiers that you can append onto the name of a resource directory (like `mipmap`) to indicate that these are resources for devices of a certain screen density. Here's a list of density qualifiers on Android:

- `mdpi` - resources for medium-density screens (~160 dpi)
- `hdpi` - resources for high-density screens (~240 dpi)
- `xhdpi` - resources for extra-high-density screens (~320 dpi)
- `xxhdpi` - resources for extra-extra-high-density screens (~480dpi)
- `xxxhdpi` - resources for extra-extra-extra-high-density screens (~640dpi)
- `nodpi` - resources that are not meant to be scaled, regardless of the screen's pixel density
- `anydpi` - resources that scale to any density

Note: You may wonder why launcher icon assets are located in **`mipmap`** directories separate from other app assets located in **`drawable`** directories. This is because some launchers may display your app icon at a larger size than what's provided by the device's default density bucket. For example, on a hdpi device, a certain device launcher may want to use the xhdpi version of the app icon instead.

4. If you click on the image files, you'll see a preview. The `ic_launcher.png` files contain the square version of the icon, while the `ic_launcher_round.png` files contain the circular version of the icon. Both are provided in each resource directory.

   For example, this is what **res > mipmap-xxxhdpi > ic_launcher_round.png** looks like. Also notice the size of the asset is in the top right. This image is 192px x 192px in size.



On the other hand, this is what **res > mipmap-mdpi > ic_launcher_round.png** looks like. It's only 48px x 48px in size.

As you can see, these bitmap image files are composed of a fixed grid of pixels. They were created for a certain screen resolution. Hence the quality can degrade as you resize them.

If you scale down a bitmap image, it will probably look okay because you are getting rid of pixel information. If you scale up a bitmap image significantly, it may appear blurry because Android will need to guess at and fill in the missing pixel information.

Note: To avoid a blurry app icon, be sure to provide different bitmap images of an app icon for each density bucket (`mdpi`, `hdpi`, `xhdpi`, etc...) Note that device screen densities won't be precisely 160dpi, 240dpi, 320dpi, etc... Based on the current screen density, Android will select the resource at the closest larger density bucket and then scale down.

Now that you have some background context on launcher icons, you'll learn about adaptive icons next.

4. Adaptive Icons

Foreground and Background Layers

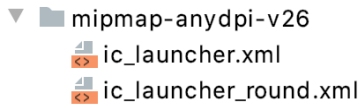
As of the Android 8.0 release (API level 26), there is support for adaptive launcher icons, which allows for more flexibility and interesting visual effects when it comes to app icons. For developers, that means that your app icon is made up of two layers: a foreground and a background layer.

In the above example, the white Android icon is in the foreground layer, while the blue and white grid is in the background layer. The foreground layer will be stacked on top of the background layer. Then a mask (circular mask in this case) will be applied on top to produce a circular shaped app icon.

Explore adaptive icon files

Look at the default adaptive icon files provided by the project template in Android Studio.

1. In the **Project window** of Android Studio, look for and expand out the **res > mipmap-anydpi-v26** resource directory.

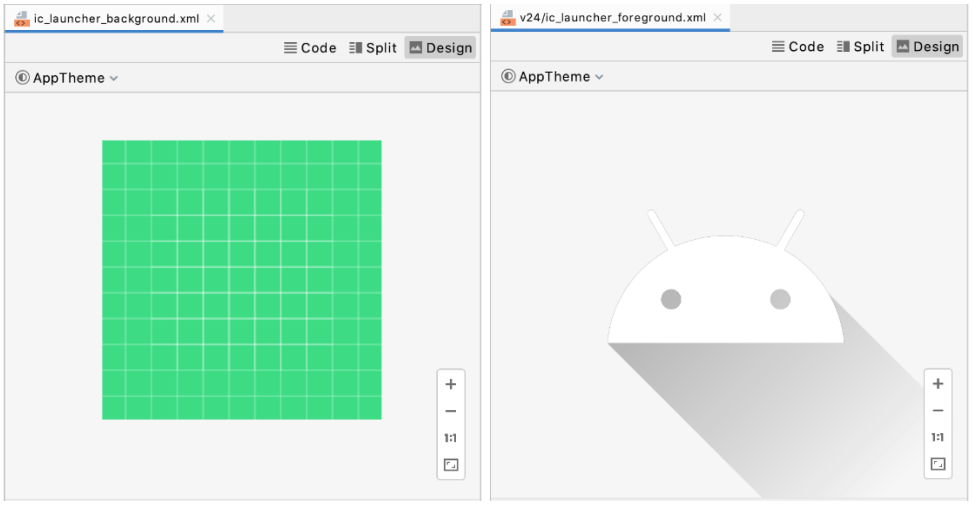


Note: Adaptive icons were added in API level 26 of the platform, so the adaptive icons should be declared in a `mipmap` resource directory that has the `-v26` resource qualifier on it. That means the resources in this directory will only be applied on devices that are running API 26 (Android 8.0) or higher. The resource files in this directory are ignored on devices running older versions of the platform.

2. Open up one of the XML files, for example `ic_launcher.xml`. You should see this:

```
<?xml version="1.0" encoding="utf-8"?>
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
    <background android:drawable="@drawable/ic_launcher_background" />
    <foreground android:drawable="@drawable/ic_launcher_foreground" />
</adaptive-icon>
```

3. Notice how the `<adaptive-icon>` element is used to declare the `<background>` and `<foreground>` layers of the app icon by providing resource drawables for each.
4. Go back to the **Project** view and look for where the drawables are declared: **drawable > ic_launcher_background.xml** and **drawable-v24 > ic_launcher_foreground.xml**.
5. Switch to **Design** view to see a preview of each (background on left, foreground on right).



6. These are both vector drawable files. They don't have a fixed size in pixels. If you switch to **Code** view, you can see the XML declaration for the vector drawable using the `<vector>` element.

While a vector drawable and a bitmap image both describe a graphic, there are important differences.

A bitmap image doesn't understand much about the image that it holds, except for the color information at each pixel. On the other hand, a vector graphic knows how to draw the shapes that define an image. These instructions are composed of a set of

points, lines, and curves along with color information. The advantage is that a vector graphic can be scaled for any canvas size for any screen density, without losing quality.

A **vector drawable** is Android's implementation of vector graphics, intended to be flexible enough on mobile devices. You can define them in XML with these possible elements. Instead of providing versions of a bitmap asset for all density buckets, you only need to define the image once. Thus, reducing the size of your app and making it easier to maintain.

Note: there are tradeoffs to using a vector drawable versus a bitmap image. For example, icons can be ideal as vector drawables because they are made up of simple shapes, while a photograph would be harder to describe as a series of shapes. It would be more efficient to use a bitmap asset in that case.

Now it's time to move onto actually changing the app icon.

Download new assets

Next, download these 2 new assets that will enable you to create an adaptive icon for a Tip Calculator app. You don't need to worry about understanding every detail of the vector drawable files. Their contents can get auto-generated for you from design tools.

1. Download `ic_launcher_background.xml`, the vector drawable for the background layer. If your browser shows the file instead of downloading it, select **FIle > Save Page As...** to save it to your computer.
2. Download `ic_launcher_foreground.xml`, the vector drawable for the foreground layer.

Note that there are certain requirements on these foreground and background layer assets, such as both must be 108dp x 108dp in size. More details on the requirements here or you can see the design guidance on Android icons on the Material site.

Because the edges of your icon could get clipped depending on the shape of the mask from the device manufacturer, it's important to put the key information of your icon in a " safe zone" which is a circle of diameter 66 dp in the center of the layer. The content outside of that safe zone should not be essential (e.g. background color) where it's okay if it gets clipped.

6. Change the app icon

Go back to Android Studio to use the new assets.

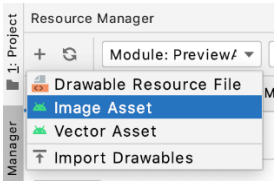
1. First delete the old drawable resources that have the Android icon and green grid background. In the **Project view**, right click on the file and choose **Delete**.

Delete:

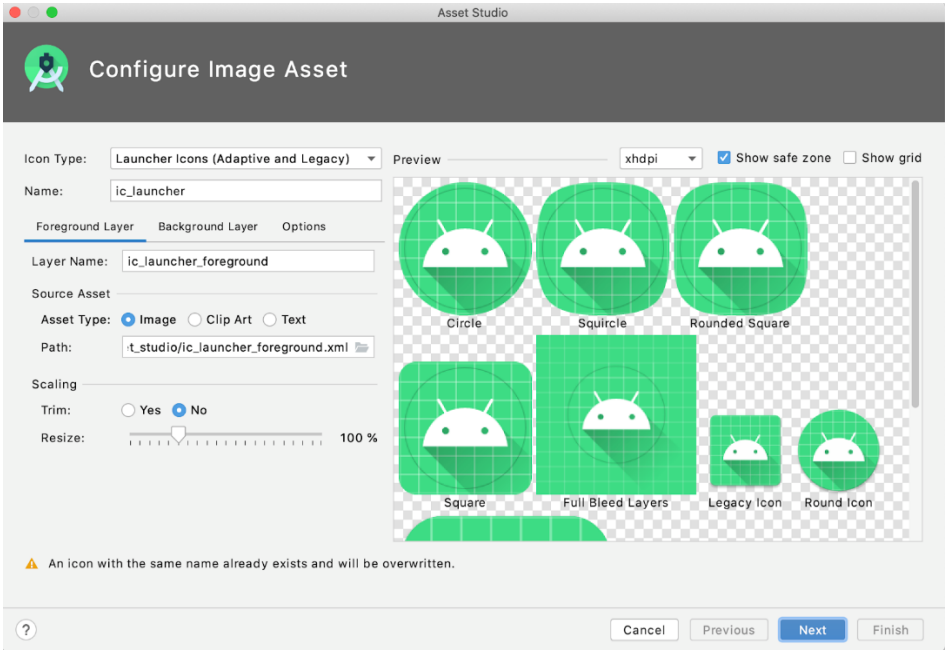
drawable/ic_launcher_background.xml
drawable-v24/ic_launcher_foreground.xml

You can uncheck the box **Safe delete (with usage search)** and click **OK**.

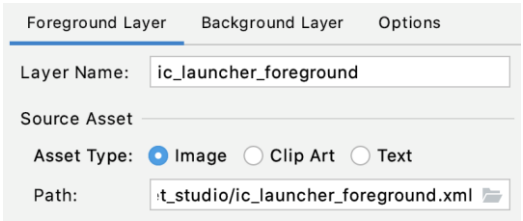
2. Create a new **Image Asset**. You could right click on the **res** directory and choose **New > Image Asset.** Or you can click on the **Resource Manager** tab, click the **+** icon, and select **Image Asset.**



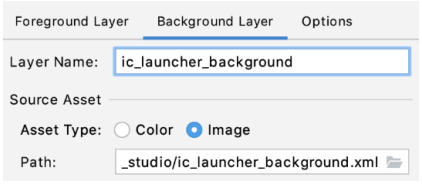
3. Android Studio's **Image Asset Studio** tool opens.
4. Leave the default settings:
5. Icon Type: Launcher Icons (Adaptive and Legacy)
6. Name: ic_launcher



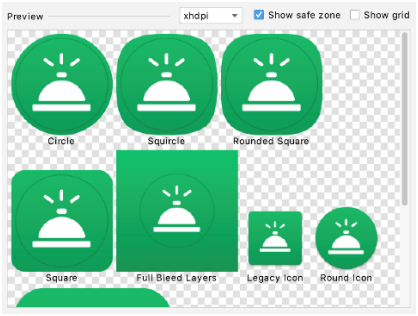
5. With the **Foreground Layer tab** already selected, go down to the **Source Asset** subsection. On the **Path** field, click the folder icon.
6. A prompt pops up to browse your computer and select a file. Find the location of the new `ic_launcher_foreground.xml` file you just downloaded on your computer. It may be in the downloads folder of your computer. Once you've found it, click **Open**.
7. The **Path** is now updated with the location of the new foreground vector drawable. Leave **Layer Name** as **ic_launcher_foreground** and **Asset Type** as **Image**.



8. Next switch to the **Background Layer tab** of the interface. Leave defaults as-is. Click the folder icon of the **Path**.
9. Find the location of the `ic_launcher_background.xml` file you just downloaded. Click **Open**.

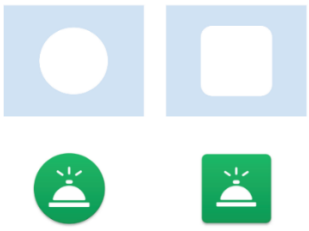


10. The preview should be updating as you select the new resource files. This is what it should look like with the new foreground and background layers.



By representing your app icon in two layers, device manufacturers (called original equipment manufacturers or OEMs for short) can create different shapes depending on the Android device, as shown in the preview above. The OEM provides a mask that gets applied to all app icons on the device.

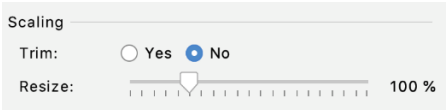
This mask gets applied on top of the foreground and background layers of your app icon. Example of a circular mask and square mask below.



When a circular mask is applied to both layers of your app icon, the result is a circular icon with a blue grid background and an Android in it (left image above). Alternatively, you could apply a square mask to produce the app icon in the above right.

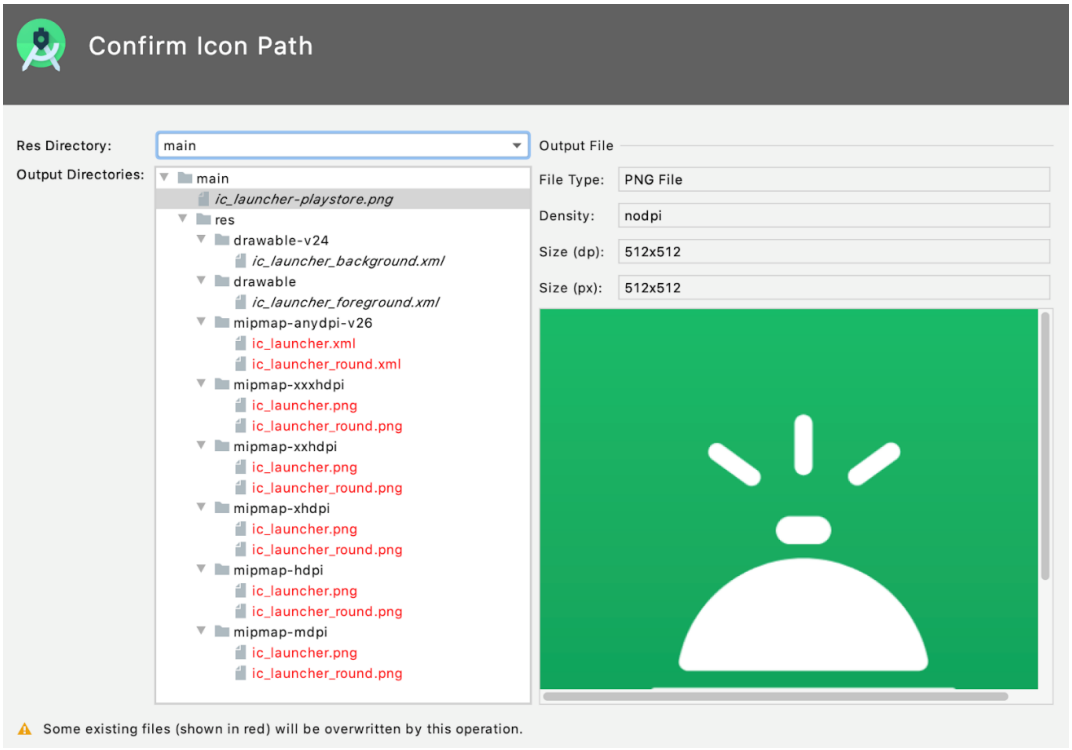
Having two layers also allows for interesting visual effects because the two layers can move independently or be scaled. For some fun examples of how the visual effects can look, check out this blogpost under Design Considerations. Because you can't tell in advance what device your user will have or what mask the OEM will apply onto your icon, you need to set up your adaptive icon so important information doesn't get cut off.

11. Make sure the main contents of the foreground layer (the service bell icon in this case) are contained within the safe zone and not clipped by the different mask shapes. If important content is clipped or appearing too small, then you can use the **Resize** slider bar under the **Scaling** section of each layer. In this case, no resizing is needed, so you can leave it at 100%.



12. Click **Next**.

13. This step is to **Confirm Icon Path**. You can click the individual files to see the preview. There's also a warning at the bottom that some existing files will be overwritten (shown in red). That is okay because those old files were for the previous app icon.



14. The defaults are fine, so click **Finish**.
15. Verify all the generated assets look correct in the `mipmap` folders. Examples:

Great work! Now you'll do one more change.

Move vector drawable files into -v26 directory

Depending on the min SDK of your app, you may notice that the foreground asset is located in the `drawable-v24` folder, while the background asset is in the `drawable` folder. The reason is because the foreground asset contains a gradient, which was available starting in the Android 7.0 release (also known as API version 24, hence the `-v24` resource qualifier). The background asset doesn't contain a gradient, so that can go in the base `drawable` folder.

Instead of having the foreground and background assets in two separate `drawable` folders, move both vector drawable files into a `-v26` resource directory. Since these assets are only used for adaptive icons, these two drawables are only needed on API 26 and above. This folder structure will make it easier to find and manage your adaptive icon files.

drawable-anydpi-v26
  ic_launcher_background.xml
  ic_launcher_foreground.xml
mipmap-anydpi-v26
  ic_launcher.xml

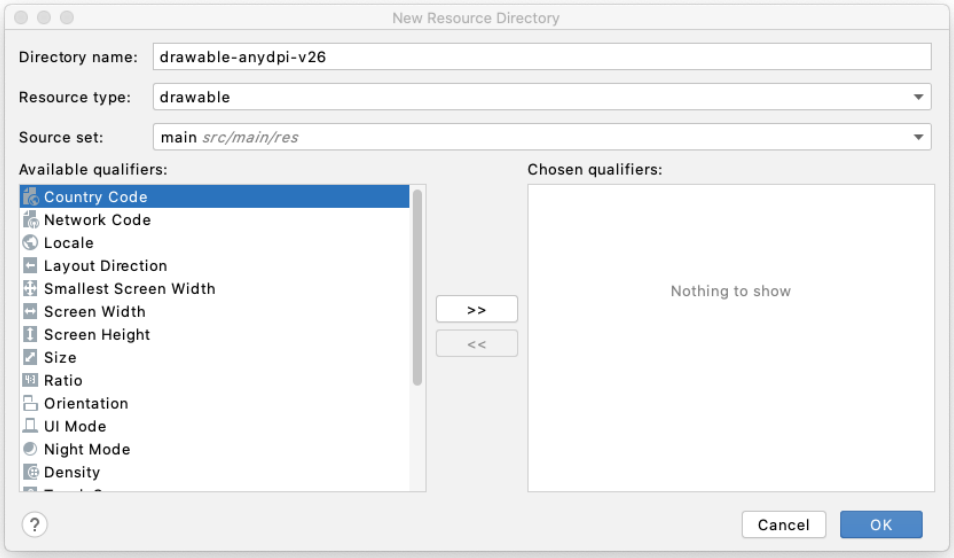
ic_launcher_round.xml

1. First create the `drawable-anydpi-v26` directory. Right click on the **res** folder. Select **New > Android Resource Directory.**
2. **New Resource Directory** dialog will appear. Select these options:

   Directory name: drawable-anydpi-v26

   Resource type: drawable (Select from dropdown)

   Source set: main (leave default value as-is)

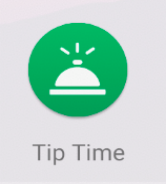


   Click **OK**. In the **Project** view, verify the new resource directory **res > drawable-anydpi-v26** has been created.

3. Left click on the `ic_launcher_foreground.xml` file and drag it from the **drawable** folder into the **drawable-anydpi-v26** folder. Recall that putting a resource in an "any dpi" directory indicates that it's a resource that can scale to any density.
4. Left click on the `ic_launcher_background.xml` and drag it from the **drawable-v24** folder into the **drawable-anydpi-v26** folder.
5. Delete the `drawable-v24` folder if it's empty now. Right click on the folder and then select **Delete**.
6. Click through all the `drawable` and `mipmap` files in your project. Make sure the preview of these icons look correct.

Test your app

1. Test that your new app icon appears. Run the app on your device (emulator or physical device).
2. Hit the home button on your device.
3. Swipe up to show the All Apps list.
4. Look for the app you just updated. You should see the new app icon displayed.



Note: Depending on your device model, you may see a launcher icon of a different shape. But nevertheless, it should show your foreground layer on top of your background layer with some type of mask applied to it.

Nice job! The new app icon looks great.

Adaptive and legacy launcher icons

Now that your adaptive icon works well, you may be wondering why you can't get rid of all the app icon bitmap images. You still need those files so that your app icon appears high-quality on older versions of Android, which is referred to as backwards compatibility.

On devices running Android 8.0 or higher (API version 26 and above):

Adaptive icons can be used (combination of foreground vector drawable, background vector drawable, with an OEM mask applied on top of it). These are the relevant files in your project:

res/drawable-anydpi-v26/ic_lancher_background.xml
res/drawable-anydpi-v26/ic_launcher_foreground.xml
res/mipmap-anydpi-v26/ic_launcher.xml
res/mipmap-anydpi-v26/ic_launcher_round.xml

On devices running anything below Android 8.0 (but above the minimum required API level of our app):

Legacy launcher icons will be used (the bitmap images in the `mipmap` folders of different density buckets). These are the relevant files in your project:

res/mipmap-mdpi/ic_launcher.png
res/mipmap-mdpi/ic_launcher_round.png
res/mipmap-hdpi/ic_launcher.png
res/mipmap-hdpi/ic_launcher_round.png
res/mipmap-xhdpi/ic_launcher.png
res/mipmap-xhdpi/ic_launcher_round.png
res/mipmap-xxdpi/ic_launcher.png
res/mipmap-xxdpi/ic_launcher_round.png
res/mipmap-xxxdpi/ic_launcher.png
res/mipmap-xxxdpi/ic_launcher_round.png

Essentially, Android will fall back to the bitmap images on older devices without adaptive icon support.

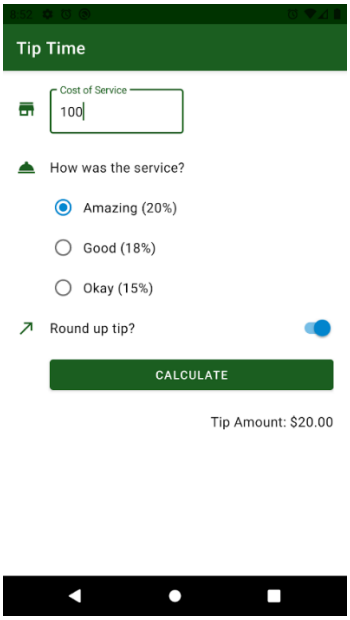
Congratulations, you have completed all the steps for changing an app icon!

8. Summary

- Place app icon files in the `mipmap` resource directories.
- Provide different versions of an app icon bitmap image in each density bucket (`mdpi`, `hdpi`, `xhdpi`, `xxhdpi`, `xxxhdpi`) for backwards compatibility with older versions of Android.
- Add resource qualifiers onto resource directories to specify resources that should be used on devices with a certain configuration (e.g. `v26`).
- Vector drawables are Android's implementation of vector graphics. They are defined in XML as a set of points, lines, and curves along with associated color information. Vector drawables can be scaled for any density without loss of quality.
- Adaptive icons were introduced to the Android platform in API 26. They are made up of a foreground and background layer that follow specific requirements, so that your app icon looks high-quality on a range of devices with different OEM masks.
- Use Image Asset Studio in Android Studio to create legacy and adaptive icons for your app.

1. Before you begin

As you've learned in earlier codelabs, Material is a design system created by Google with guidelines, components, and tools that support the best practices of user interface design. In this codelab, you will update the tip calculator app (from previous codelabs) to have a more polished user experience, as seen in the final screenshot below. You'll also test the app in some additional scenarios to ensure the user experience is as smooth as possible.



Prerequisites

- Familiar with common UI widgets such as `TextView`, `ImageView`, `Button`, `EditText`, `RadioButton`, `RadioGroup`, and `Switch`
- Familiar with `ConstraintLayout` and positioning child views by setting constraints
- Comfortable with modifying XML layouts
- Aware of the difference between bitmap images and vector drawables
- Can set theme attributes in a theme

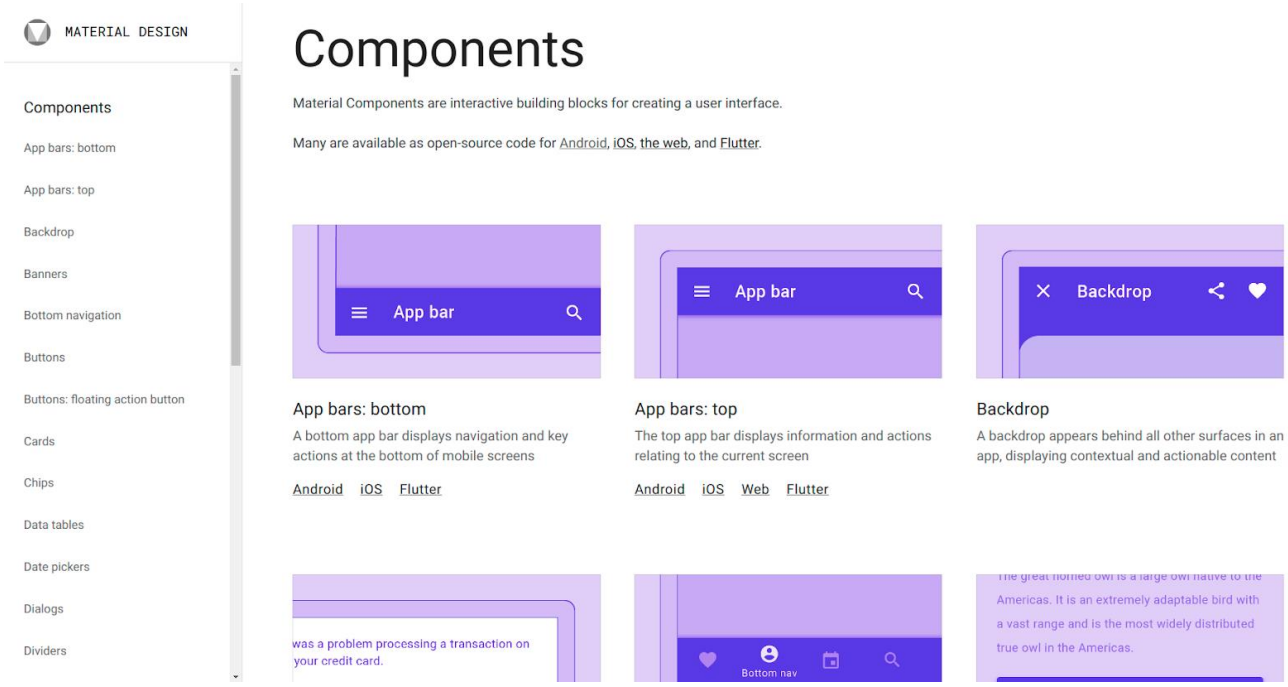
- Able to turn on Dark theme on a device
- Have previously modified the app's `build.gradle` file for project dependencies

What you'll learn

- How to use Material Design Components in your app
- How to use Material icons in your app by importing them from Image Asset Studio
- How to create and apply new styles
- How to set other theme attributes aside from color

3. Material Components

Material Components are common UI widgets that make it easier to implement Material styling in your app. The documentation provides information for how to use and customize the Material Design Components. There are general Material Design guidelines for each component, and Android platform-specific guidance for the components that are available on Android. The labeled diagrams give you enough information to recreate a component if it happens to not exist on your chosen platform.



By using Material Components, your app will operate in a more consistent way alongside other apps on the user's device. That way the UI patterns learned in one app can be carried over to the next one. Hence users will be able to learn how to use your app much faster. It's recommended to use Material Components whenever possible (as opposed to the non-Material widgets). They are also more flexible and customizable, as you will learn in this next task.

The Material Design Components (MDC) library needs to be included as a dependency in your project. If you are using Android Studio 4.1 or newer, this line should already be present in your project by default. In your app's `build.gradle` file, make sure this dependency is included with the latest version of the library. For more details, see the Get started page on the Material site.

app/build.gradle

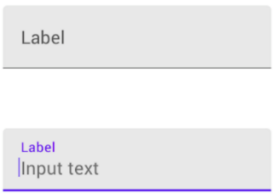
```
dependencies {
    ...
    implementation 'com.google.android.material:material:<version>'
}
```

Text Fields

In your tip calculator app, at the top of the layout, you currently have an `EditText` field for the cost of service. This `EditText` field works, but it doesn't follow the recent Material Design guidelines on how text fields should look and behave.

For any new component that you want to use, start by learning about it on the Material site. From the guide on Text Fields, there are two types of text fields:

Filled text field



Outlined text field

To create a text field as shown above, use a `TextInputLayout` with an enclosed `TextInputEditText` from the MDC library. The Material text field can be easily customized to:

- Display input text or a label that's always visible
- Display an icon in the text field
- Display helper or error messages

In the first task of this codelab, you'll be replacing the cost of service `EditText` with a Material text field (which is composed of a `TextInputLayout` and `TextInputEditText`).

1. With the **Tip Time** app open in Android Studio, go to the `activity_main.xml` layout file. It should contain a `ConstraintLayout` with the tip calculator layout.
2. To see an example of what the XML looks like for a Material text field, go back to the Android guidance for Text fields. You should see snippets like this one:

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/textField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/label">

    <com.google.android.material.textfield.TextInputEditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
    />

</com.google.android.material.textfield.TextInputLayout>
```

3. After seeing this example, insert a Material text field as the first child of the `ConstraintLayout` (before the `EditText` field). You'll get rid of the `EditText` field in a later step.

   You can type this into Android Studio and use autocomplete to make it easier. Or you can copy the example XML from the documentation page and paste it into your layout like this. Notice how the `TextInputLayout` has a child view, the `TextInputEditText`. Remember that ellipsis (...) are used to abbreviate snippets, so that you can focus on the lines of XML that have actually changed.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ...>

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/textField"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/label">

        <com.google.android.material.textfield.TextInputEditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
        />

    </com.google.android.material.textfield.TextInputLayout>

    <EditText
        android:id="@+id/cost_of_service" ... />

    ...
```

   You are expected to see errors on the `TextInputLayout` element. You haven't properly constrained this view yet in the parent `ConstraintLayout`. Also the string resource isn't recognized. You'll fix these errors in the coming steps.

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/textField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/label">
```

4. Add vertical and horizontal constraints onto the text field, to properly position it within the parent `ConstraintLayout`. Since you haven't deleted the `EditText` yet, cut and paste the following attributes from the `EditText` and place them onto the `TextInputLayout`: the constraints, resource ID `cost_of_service`, layout width of `160dp`, layout height of `wrap_content`, and the hint text `@string/cost_of_service`.

   ...

   ```
   <com.google.android.material.textfield.TextInputLayout
     android:id="@+id/cost_of_service"
     android:layout_width="160dp"
     android:layout_height="wrap_content"
     android:hint="@string/cost_of_service"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent">

     <com.google.android.material.textfield.TextInputEditText
       android:layout_width="match_parent"
       android:layout_height="wrap_content"/>

   </com.google.android.material.textfield.TextInputLayout>
   ```

   ...

   You may see an error that the `cost_of_service` ID is the same as the resource ID of the `EditText`, but you can ignore this error for now. (`EditText` will be removed in a couple steps).

5. Next make sure the `TextInputEditText` element has all the appropriate attributes. Cut and paste over the input type from the `EditText` onto the `TextInputEditText`. Change the `TextInputEditText` element resource ID to `cost_of_service_edit_text`.

   ```
   <com.google.android.material.textfield.TextInputLayout ... >

     <com.google.android.material.textfield.TextInputEditText
       android:id="@+id/cost_of_service_edit_text"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:inputType="numberDecimal" />

   </com.google.android.material.textfield.TextInputLayout>
   ```

   Width of `match_parent` and height of `wrap_content` is fine as-is. When setting a width of `match_parent`, the `TextInputEditText` will have the same width as its parent `TextInputLayout` which is `160dp`.

6. Now that you have copied over all relevant information from the `EditText`, go ahead and delete the `EditText` from the layout.

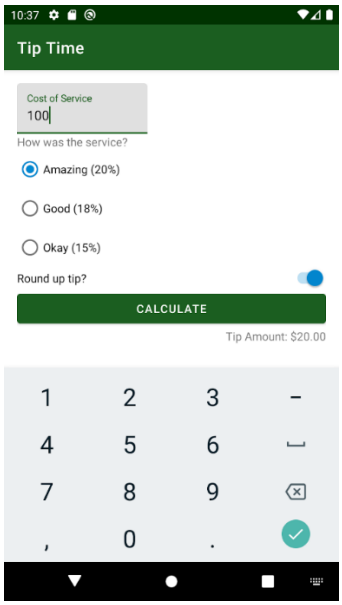
7. In the **Design** view of your layout, you should see this preview. The cost of service field now looks like a Material text field.

8. You can't run the app yet because there's an error in your `MainActivity.kt` file in the `calculateTip()` method. Recall from an earlier codelab that with view binding enabled for your project, Android creates properties in a binding object based on the resource ID name. The field we retrieved the cost of service from has changed in the XML layout, so the Kotlin code needs to be updated accordingly.

   You will now be retrieving the user input from the `TextInputEditText` element with resource ID `cost_of_service_edit_text`. In the `MainActivity`, use `binding.costOfServiceEditText` to access the text string stored within it. The rest of the `calculateTip()` method can stay the same.

```
private fun calculateTip() {
    // Get the decimal value from the cost of service text field
    val stringInTextField = binding.costOfServiceEditText.text.toString()
    val cost = stringInTextField.toDoubleOrNull()

    ...
}
```

9. Great work! Now run the app and test that it still works. Notice how the "Cost of Service" label now appears above your input as you type. The tip should still calculate as expected.



Switches

In the Material Design guidelines, there's also guidance on switches. A switch is a widget where you can toggle a setting on or off.

1. Check out the Android guidance for Material switches. You will learn about the `SwitchMaterial` widget (from the MDC library), which provides Material styling for switches. If you keep scrolling through the guide, you will see some example XML.
2. To use `SwitchMaterial`, you must explicitly specify `SwitchMaterial` in your layout and use the fully qualified path name.

   In the `activity_main.xml` layout, change the XML tag from `Switch` to `com.google.android.material.switchmaterial.SwitchMaterial`.

   ...

   ```
   <com.google.android.material.switchmaterial.SwitchMaterial
     android:id="@+id/round_up_switch"
     android:layout_width="0dp"
     android:layout_height="wrap_content" ... />
   ```

   ...

3. Run the app to verify it still compiles. There happens to be no visible change in the app. However, an advantage to using `SwitchMaterial` from the MDC library (instead of `Switch` from the Android platform), is that when the library's implementation for `SwitchMaterial` gets updated (e.g. the Material Design guidelines change), then you will get the updated widget for free without any changes required on your part. This helps future-proof your app.

   At this point, you've seen two examples of how your UI can benefit from using out-of-the-box Material Design Components and how that brings your app closer in line with Material guidelines. Remember that you can always explore other Material Design Components provided on Android at this site.

4. Icons

Icons are symbols that can help users understand a user interface by visually communicating the intended function. They often take inspiration from objects in the physical world that a user is expected to have experienced. Icon design often reduces the level of detail to the minimum required to be familiar to a user. For example, a pencil in the physical world is used for writing so its icon counterpart usually indicates creating, adding, or editing an item.


Photo by Angelina Litvin on Unsplash

Sometimes icons are linked to obsolete physical world objects as is the case with the floppy disk icon. This icon is the ubiquitous indication of saving a file or database record; however, while floppy disks were popularized in the 1970s, they ceased to be common after 2000. But its continual use today speaks to how a strong visual can transcend the lifetime of its physical form.

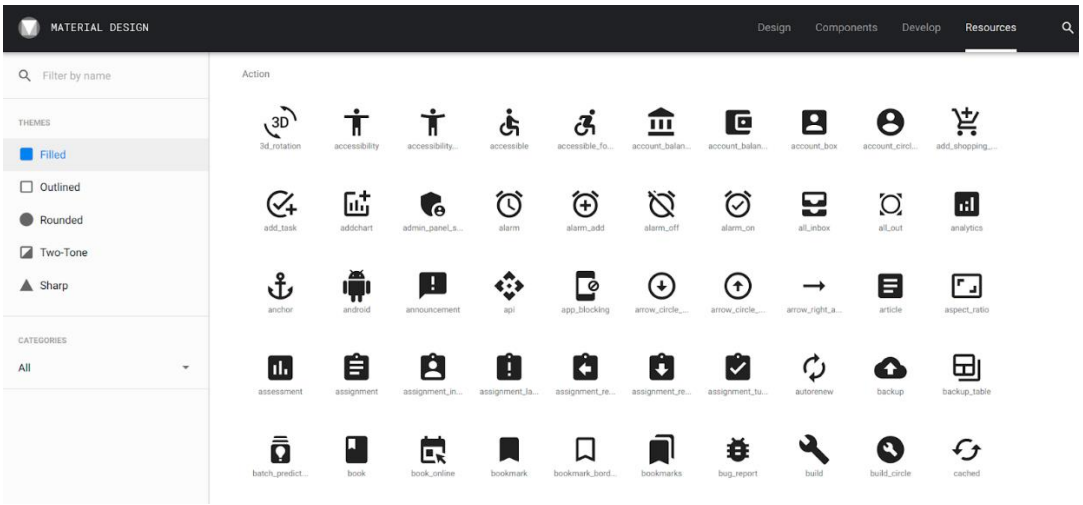

Photo by Vincent Botta on Unsplash

Representing icons in your app

For icons in your app, instead of providing different versions of a bitmap image for different screen densities, the recommended practice is to use vector drawables. Vector drawables are represented as XML files that store the instructions on how to create an image rather than saving the actual pixels that make up that image. Vector drawables can be scaled up or down without any loss of visual quality or increase in file size.

Provided Icons

Material Design provides a number of icons arranged in common categories for most of your needs. See list of icons.



These icons can also be drawn using one of five themes (Filled, Outlined, Rounded, Two-Tone, and Sharp) and can be tinted with colors.

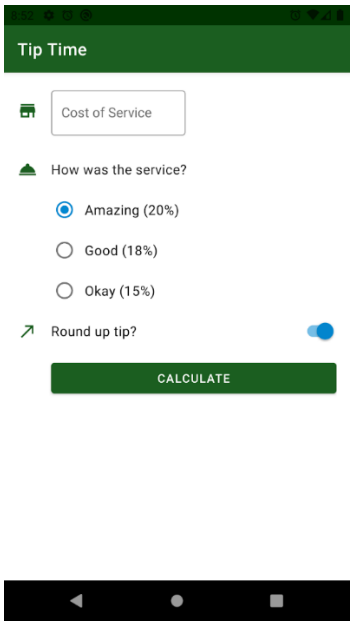
Filled	Outlined	Rounded	Two-Tone	Sharp
				

Adding Icons

In this task, you'll add three vector drawable icons to the app:

1. Icon next to the cost of service text field
2. Icon next to the service question
3. Icon next to the round up tip prompt

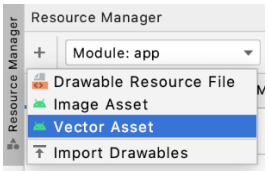
Below is a screenshot of the final version of the app. After you add the icons, you'll adjust the layout to accommodate the placement of these icons. Notice how the fields and calculate button get shifted over to the right, with the addition of the icons.



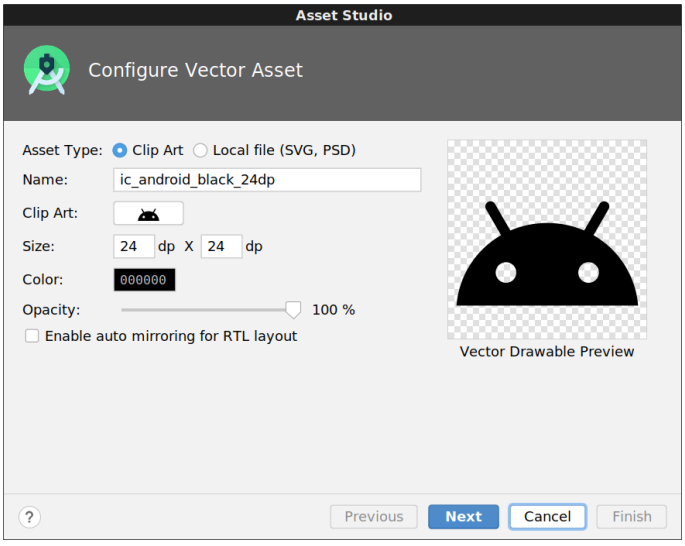
Add vector drawable assets

You can create these icons as vector drawables directly from **Asset Studio** in Android Studio.

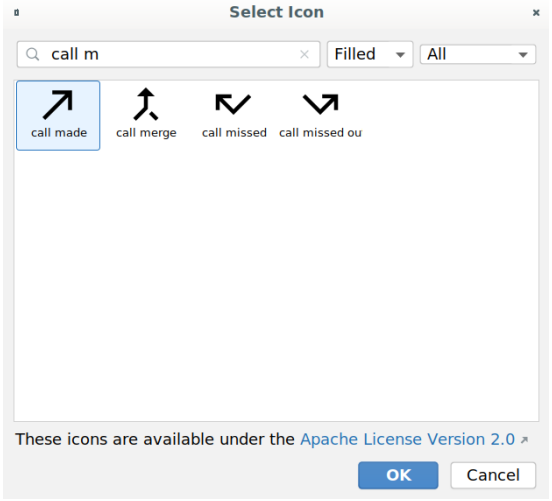
1. Open the **Resource Manager** tab located on the left of the application window.
2. Click the + icon and select **Vector Asset.**



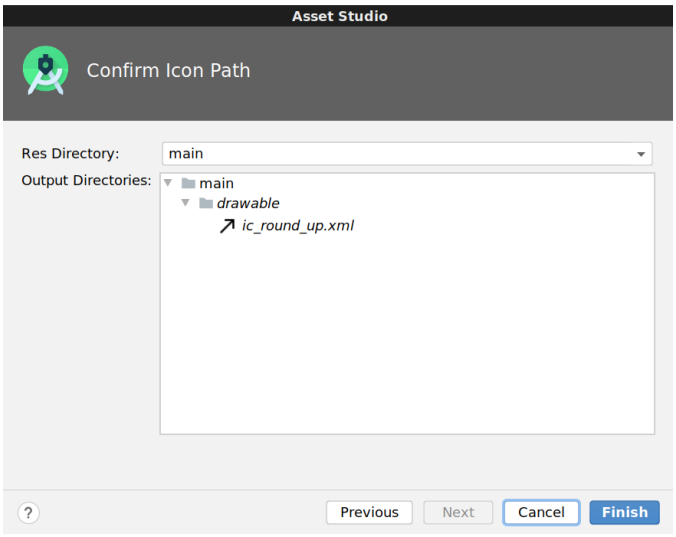
3. For the **Asset Type**, make sure the radio button labeled **Clip Art** is selected.



4. Click the button next to **Clip Art:** to select a different clip art image. In the prompt that appears, type "call made" into the window that appears. You'll be using this arrow icon for the round up tip option. Select it and click **OK**.



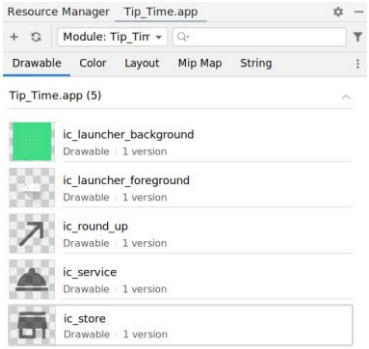
5. Rename the icon to `ic_round_up`. (It's recommended to use the prefix ic_ when naming your icon files.) You can leave the **Size** as 24 dp x 24 dp and **Color** as black 000000.
6. Click **Next**.
7. Accept the default directory location and click **Finish**.



8. Repeat steps 2 - 7 for the other two icons:

- **Service question icon:** Search for "room service" icon, save it as `ic_service`.
- **Cost of service icon:** Search for "store" icon, save it as `ic_store`.

9. Once you're done, the **Resource Manager** will look like the below screenshot. You will also have these three vector drawables (`ic_round_up`, `ic_service`, and `ic_store`) listed in your `res/drawable` folder.



Support older Android versions

You just added vector drawables to your app, but it's important to note that support for vector drawables on the Android platform wasn't added until Android 5.0 (API level 21).

Based on how you set up the project, the minimum SDK version for the Tip Time app is API 19. That means the app can run on Android devices that are running Android platform version 19 or higher.

To make your app work on these older versions of Android (known as backwards compatibility), add the `vectorDrawables` element to your app's `build.gradle` file. This enables you to use vector drawables on versions of the platform less than API 21, versus converting to PNGs when the project is built. See more details here.

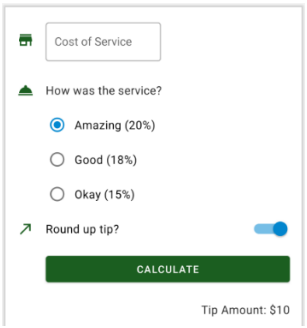
`app/build.gradle`

```
android {
 defaultConfig {
  ...
  vectorDrawables.useSupportLibrary = true
  }
  ...
}
```

With your project configured properly, you can now move onto adding the icons into the layout.

Insert icons and position elements

You'll be using `ImageViews` to display icons in the app. This is how your final UI will appear.



1. Open the `activity_main.xml` layout.
2. First position the store icon next to the cost of service text field. Insert a new `ImageView` as the first child of the `ConstraintLayout`, before the `TextInputLayout`.

```
<androidx.constraintlayout.widget.ConstraintLayout
  ...>

  <ImageView
     android:layout_width=""
     android:layout_height=""

  <com.google.android.material.textfield.TextInputLayout
     android:id="@+id/cost_of_service"
```

...

3. Set up the appropriate attributes on the `ImageView` to hold the `ic_store` icon. Set the ID to `icon_cost_of_service`. Set the `app:srcCompat` attribute to the drawable resource `@drawable/ic_store`, and you'll see a preview of the icon next to that line of XML. Also set `android:importantForAccessibility="no"` since this image is used for decorative purposes only.

```
<ImageView
  android:id="@+id/icon_cost_of_service"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:importantForAccessibility="no"
  app:srcCompat="@drawable/ic_store" />
```

It is expected that there will be an error on the `ImageView` because the view is not constrained yet. You'll fix that next.

4. Position the `icon_cost_of_service` in two steps. First add constraints onto the `ImageView` (this step), and then update constraints on the `TextInputLayout` next to it (step 5). This diagram shows how the constraints should be set up.

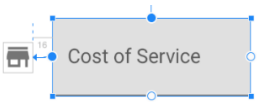
On the `ImageView`, you want its starting edge to be constrained to the starting edge of the parent view (`app:layout_constraintStart_toStartOf="parent"`).

The icon appears centered vertically compared to the text field beside it, so constrain the top of this `ImageView` (`layout_constraintTop_toTopOf`) to the top of the text field. Constrain the bottom of this `ImageView` (`layout_constraintBottom_toBottomOf`) to the bottom of the text field. To refer to the text field, use the resource ID `@id/cost_of_service`. The default behavior is that when two constraints are applied to a widget in the same dimension (such as a top and bottom constraint), the constraints are applied equally. The result is that the icon gets vertically centered, in relation to the cost of service field.

```
<ImageView
  android:id="@+id/icon_cost_of_service"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:importantForAccessibility="no"
  app:srcCompat="@drawable/ic_store"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="@id/cost_of_service"
  app:layout_constraintBottom_toBottomOf="@id/cost_of_service" />
```

The icon and text field are still overlapping in the **Design** view. That will be fixed in the next step.

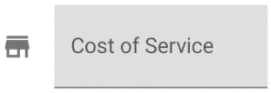
5. Before the addition of the icon, the text field was positioned at the start of parent. Now it needs to be shifted over to the right. Update the constraints on the `cost_of_service` text field in relation to `icon_cost_of_service`.



The starting edge of the `TextInputLayout` should be constrained to the ending edge of the `ImageView` (`@id/icon_cost_of_service`). To add some spacing between the two views, add a start margin of `16dp` on the `TextInputLayout`.

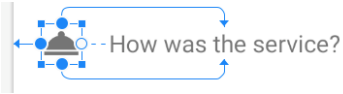
```
<com.google.android.material.textfield.TextInputLayout
   android:id="@+id/cost_of_service"
   ...
   android:layout_marginStart="16dp"
   app:layout_constraintStart_toEndOf="@id/icon_cost_of_service">

   <com.google.android.material.textfield.TextInputEditText ... />

</com.google.android.material.textfield.TextInputLayout>
```

After all these changes, the icon should be positioned correctly next to the text field.



6. Next insert the service bell icon next to the "How was the service?" `TextView`. While you could declare the `ImageView` anywhere within the `ConstraintLayout`, your XML layout will be easier to read if you insert the new `ImageView` in the XML layout after the `TextInputLayout`, but before the `service_question` `TextView`.

   For the new `ImageView`, assign it a resource ID of `@+id/icon_service_question`. Set the appropriate constraints on the `ImageView` and the service question `TextView`.

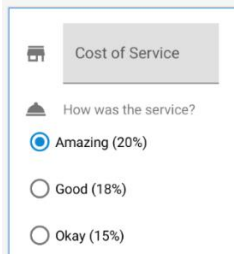


   Also add a `16dp` top margin to the `service_question TextView` so there's more vertical space between the service question and the cost of service text field above it.

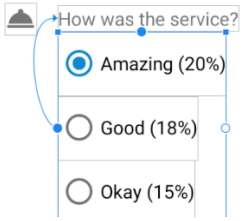
```
...

  <ImageView
     android:id="@+id/icon_service_question"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:importantForAccessibility="no"
     app:srcCompat="@drawable/ic_service"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="@id/service_question"
     app:layout_constraintBottom_toBottomOf="@id/service_question" />

  <TextView
```

```
        android:id="@+id/service_question"
        ...
        android:layout_marginTop="16dp"
        app:layout_constraintStart_toStartOf="@id/cost_of_service"
        app:layout_constraintTop_toBottomOf="@id/cost_of_service"/>

    ...
```

7. At this point the **Design** view should look like this. The cost of service field and service question (and their respective icons) look great, but the radio buttons now look out of place. They aren't vertically aligned with the content above it.



8. Improve the positioning of the radio buttons by shifting them to the right, underneath the service question. That means updating a `RadioGroup` constraint. Constrain the starting edge of the `RadioGroup` to the starting edge of the `service_question` `TextView`. All other attributes on the `RadioGroup` can remain the same.



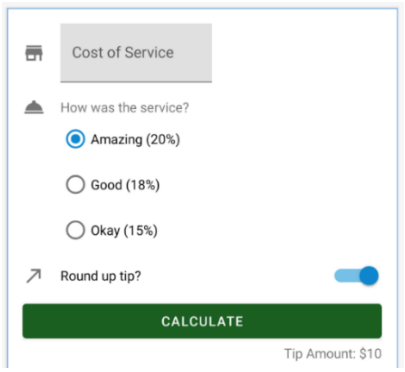

```
    ...

    <RadioGroup
      android:id="@+id/tip_options"
      ...
      app:layout_constraintStart_toStartOf="@id/service_question">

    ...
```

9. Then proceed with adding the `ic_round_up` icon to the layout next to the "Round up tip?" switch. Try doing this on your own and if you get stuck, you can consult the XML below. You can assign the new `ImageView` a resource ID of `icon_round_up`.
10. In the layout XML, insert a new `ImageView` after the `RadioGroup` but before the `SwitchMaterial` widget.
11. Assign the `ImageView` a resource ID of `icon_round_up` and set the `srcCompat` to the drawable of the icon `@drawable/ic_round_up`. Constraint the start of the `ImageView` to the start of the parent, and also vertically center the icon relative to the `SwitchMaterial`.

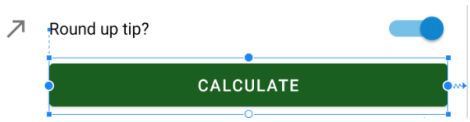
12. Update `SwitchMaterial` to be next to the icon and have a `16dp` start margin. This is what the resulting XML should look like for `icon_round_up` and `round_up_switch`.

```
...

  <ImageView
     android:id="@+id/icon_round_up"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:importantForAccessibility="no"
     app:srcCompat="@drawable/ic_round_up"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="@id/round_up_switch"
     app:layout_constraintBottom_toBottomOf="@id/round_up_switch" />

   <com.google.android.material.switchmaterial.SwitchMaterial
     android:id="@+id/round_up_switch"
     ...
     android:layout_marginStart="16dp"
     app:layout_constraintStart_toEndOf="@id/icon_round_up" />

...
```

13. The **Design** view should look like this. All three icons are correctly positioned.



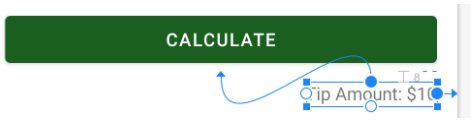
14. If you compare this with the final app screenshot, you'll notice the calculate button is also shifted over to align vertically with the cost of service field, service question, radio button options, and round up tip question. Achieve this look by constraining the start of the calculate button to the start of the `round_up_switch`. Also add `8dp` of vertical margin between the calculate button and the switch above it.




```
...

<Button
  android:id="@+id/calculate_button"
```

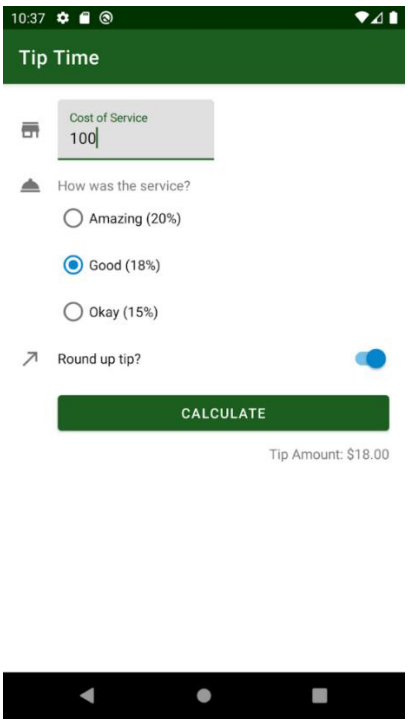
```
    ...
    android:layout_marginTop="8dp"
    app:layout_constraintStart_toStartOf="@id/round_up_switch" />

...
```

15. Last but not least, position `tip_result` by adding `8dp` of top margin to the `TextView`.




```
...

<TextView
    android:id="@+id/tip_result"
    ...
    android:layout_marginTop="8dp" />

...
```

16. That was a lot of steps! Great job on working through them step-by-step. It requires a lot of attention to detail to get elements aligned correctly in the layout, but it makes the end result look much better! Run the app and it should look like the below screenshot. By vertically aligning and increasing spacing between elements, they are not as crowded together.



You're not done yet! You may have noticed that the font size and color of the service question and tip amount look different than the text in the radio buttons and switch. Let's make these consistent in the next task by using styles and themes.

5. Styles and Themes

A style is a collection of view attributes values for a single type of widget. For example, a `TextView` style can specify font color, font size, and background color, to name a few. By extracting these attributes into a style, you can easily apply the style to multiple views in the layout and maintain it in a single place.

In this task, you will first create styles for the text view, radio button, and switch widgets.

Create Styles

1. Create a new file named `styles.xml` in the **res > values** directory if one doesn't already exist. Create it by right-clicking on the **values** directory and selecting **New > Values Resource File**. Call it `styles.xml`. The new file will have the following contents.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
</resources>
```

2. Create a new `TextView` style so that text appears consistent throughout the app. Define the style once in `styles.xml` and then you can apply it to all the `TextViews` in the layout. While you could define a style from scratch, you can extend from an existing `TextView` style from the MDC library.

   When styling a component, you should generally extend from a parent style of the widget type you are using. This is important for two reasons. First, it makes sure all important default values are set on your component, and secondly, your style will continue to inherit any future changes to that parent style.

   You can name your style anything you'd like, but there is a recommended convention. If you inherit from a parent Material style, then name your style in a parallel way by substituting `MaterialComponents` with your app's name (`TipTime`). This moves your changes into its own namespace which eliminates the possibility for future conflicts when Material Components introduces new styles. Example:

   Your style name: `Widget.TipTime.TextView` Inherits from parent style: `Widget.MaterialComponents.TextView`

   Add this to your `styles.xml` file in between the `resources` opening and closing tags.

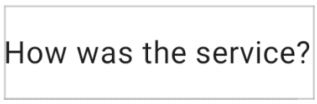
```
<style name="Widget.TipTime.TextView" parent="Widget.MaterialComponents.TextView">
</style>
```

3. Set up your `TextView` style so that it overrides the following attributes: `android:minHeight`,`android:gravity`,and `android:textAppearance`.

   `android:minHeight` sets a minimum height of 48dp on the `TextView`. The smallest height for any row should be 48dp according to the Material Design guidelines.

   You can center the text in the `TextView` vertically by setting the `android:gravity` attribute. (See screenshot below.) Gravity controls how the content within a view will position itself. Since the actual text content doesn't take up the full 48dp in height, the value `center_vertical` centers the text within the `TextView` vertically (but does not change its

horizontal position). Other possible gravity values include `center`, `center_horizontal`, `top`, and `bottom`. Feel free to try out the other gravity values to see the effect on the text.



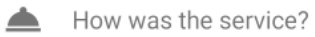
Set the text appearance attribute value to `?attr/textAppearanceBody1`. TextAppearance is a set of pre-made styles around text size, fonts, and other properties of text. For other possible text appearances that are provided by Material, see this list of type scales.

```
<style name="Widget.TipTime.TextView" parent="Widget.MaterialComponents.TextView">
    <item name="android:minHeight">48dp</item>
    <item name="android:gravity">center_vertical</item>
    <item name="android:textAppearance">?attr/textAppearanceBody1</item>
</style>
```

4. Apply the `Widget.TipTime.TextView` style to the `service_question` `TextView` by adding a style attribute on each `TextView` in `activity_main.xml`.

```
<TextView
    android:id="@+id/service_question"
    style="@style/Widget.TipTime.TextView"
    ... />
```

Before the style, the `TextView` looked like this with small font size and gray font color:



After adding the style, the `TextView` looks like this. Now this `TextView` looks more consistent with the rest of the layout.

5. Apply that same `Widget.TipTime.TextView` style to the `tip_result` `TextView`.

```
<TextView
    android:id="@+id/tip_result"
    style="@style/Widget.TipTime.TextView"
    ... />
```

Note: If you specify an attribute in a style (e.g. set **`android:textSize`** to be **`18sp`**), and also specify that same attribute in the layout file (e.g. set **`android:textSize`** to be **`14sp`**), then the value you set in the layout (**`14sp`**) will actually be applied to what you see on the screen.

6. The same text style should be applied to the text label in the switch. However, you can't set a `TextView` style onto a `SwitchMaterial` widget. `TextView` styles can only be applied on `TextViews`. Hence create a new style for the switch. The attributes are the same in terms of `minHeight`, `gravity`, and `textAppearance`. What's different here is the style name and parent because you're now inheriting from the `Switch` style from the MDC library. Your name for the style should also mirror the name of the parent style.

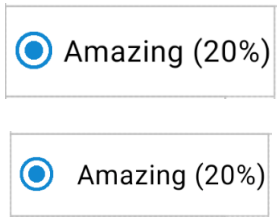
   Your style name: `Widget.TipTime.CompoundButton.Switch`. Inherits from parent style: `Widget.MaterialComponents.CompoundButton.Switch`

```
<style name="Widget.TipTime.CompoundButton.Switch" parent="Widget.MaterialComponents.CompoundButton.Switch">
  <item name="android:minHeight">48dp</item>
  <item name="android:gravity">center_vertical</item>
  <item name="android:textAppearance">?attr/textAppearanceBody1</item>
</style>
```

   You could also specify additional attributes specific to switches in this style, but in your app, there's no need to.

7. The radio button text is the last place you want to make sure the text appears visually consistent. You can't apply a `TextView` style or `Switch` style onto a `RadioButton` widget. Instead, you must create a new style for radio buttons. You can extend from the MDC library's `RadioButton` style.

   While you are creating this style, also add some padding between the radio button text and the circle visual. `paddingStart` is a new attribute you haven't used yet. Padding is the amount of space between the contents of a view and the bounds of the view. The `paddingStart` attribute sets the padding only at the start of the component. See the difference between 0dp and 8dp of `paddingStart` on a radio button.




```
<style name="Widget.TipTime.CompoundButton.RadioButton"
parent="Widget.MaterialComponents.CompoundButton.RadioButton">
  <item name="android:paddingStart">8dp</item>
  <item name="android:textAppearance">?attr/textAppearanceBody1</item>
</style>
```

8. (Optional) Create a `dimens.xml` file to improve manageability of frequently-used values. You can create the file in the same manner you did for the `styles.xml` file above. Select the values directory, right click and select **New > Values Resource File**.

   In this small app, you've repeated the minimum height setting twice. That's certainly manageable for now, but would quickly get out of control if we had 4, 6, 10 or more components sharing that value. Remembering to change all of them individually is tedious and error-prone. You can create another helpful resource file in **res > values** called `dimens.xml` that holds common dimensions that you can name. By standardizing common values as named dimensions, we make it easier to manage our app. TipTime is small so we won't be using it outside this optional step. However, with more complex apps in a production environment where you might work with a design team, `dimens.xml` will easily allow you to change these values more often.

dimens.xml

```
<resources>
  <dimen name="min_text_height">48dp</dimen>
</resources>
```

You would update styles.xml file to use @dimen/min_text_height instead of 48dp directly.

```
...
<style name="Widget.TipTime.TextView" parent="Widget.MaterialComponents.TextView">
   <item name="android:minHeight">@dimen/min_text_height</item>
   <item name="android:gravity">center_vertical</item>
   <item name="android:textAppearance">?attr/textAppearanceBody1</item>
</style>
...
```

Add these styles to your themes

You may have noticed that you haven't applied the new RadioButton and Switch styles onto the respective widgets yet. The reason is because you will be using theme attributes to set the radioButtonStyle and switchStyle in the app theme. Let's revisit what a theme is.

A theme is a collection of named resources (called theme attributes) that can be referenced later in styles, layouts, etc. You can specify a theme for an entire app, activity, or view hierarchy—not just an individual View. Previously you modified the app's theme in themes.xml by setting theme attributes like colorPrimary and colorSecondary, which gets used throughout the app and its components.

radioButtonStyle and switchStyle are other theme attributes you can set. The style resources that you provide for these theme attributes will be applied to every radio button and every switch in the view hierarchy that the theme applies to.

There's also a theme attribute for textInputStyle where the specified style resource will be applied to all text input fields within the app. To make a TextInputLayout appear like an outlined text field (as shown in the Material Design guidelines), there is an OutlinedBox style defined in the MDC library as Widget.MaterialComponents.TextInputLayout.OutlinedBox. This is the style you'll use.

1. Modify the themes.xml file so that the theme refers to the desired styles. Setting a theme attribute is done the same way you declared the colorPrimary and colorSecondary theme attributes in an earlier codelab. This time however, the relevant theme attributes are textInputStyle, radioButtonStyle, and switchStyle. You'll be using the styles you've created previously for the RadioButton and Switch along with the style for the Material OutlinedBox text field.

   Copy the following into res/values/themes.xml into the style tag for your app theme.

```
<item name="textInputStyle">@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox</item>
<item name="radioButtonStyle">@style/Widget.TipTime.CompoundButton.RadioButton</item>
```

```
<item name="switchStyle">@style/Widget.TipTime.CompoundButton.Switch</item>
```

2. This is what your `res/values/themes.xml` file should look like. You can add comments in the XML if you want (indicated by `<!-` and `-->`).

```
<resources xmlns:tools="http://schemas.android.com/tools">

    <!-- Base application theme. -->
    <style name="Theme.TipTime" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        ...
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Text input fields -->
        <item name="textInputStyle">@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox</item>
        <!-- Radio buttons -->
        <item name="radioButtonStyle">@style/Widget.TipTime.CompoundButton.RadioButton</item>
        <!-- Switches -->
        <item name="switchStyle">@style/Widget.TipTime.CompoundButton.Switch</item>
    </style>

</resources>
```

3. Be sure to make the same changes to the dark theme in **themes.xml (night)**. Your `res/values-night/themes.xml` file should look like this.

```
<resources xmlns:tools="http://schemas.android.com/tools">

    <!-- Application theme for dark theme. -->
    <style name="Theme.TipTime" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        ...
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Text input fields -->
        <item name="textInputStyle">@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox</item>
        <!-- For radio buttons -->
        <item name="radioButtonStyle">@style/Widget.TipTime.CompoundButton.RadioButton</item>
        <!-- For switches -->
        <item name="switchStyle">@style/Widget.TipTime.CompoundButton.Switch</item>
    </style>

</resources>
```

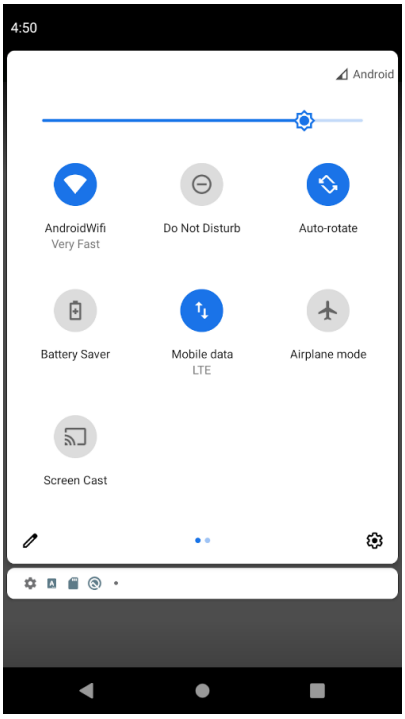
4. Run the app and see the changes. The `OutlinedBox` style looks much better for the text field, and all the text now looks consistent!

6. Enhance the user experience

As you near completion of your app, you should test your app not just with the expected workflow but in other user scenarios as well. You may find that some small code changes can improve the user experience in a large way.

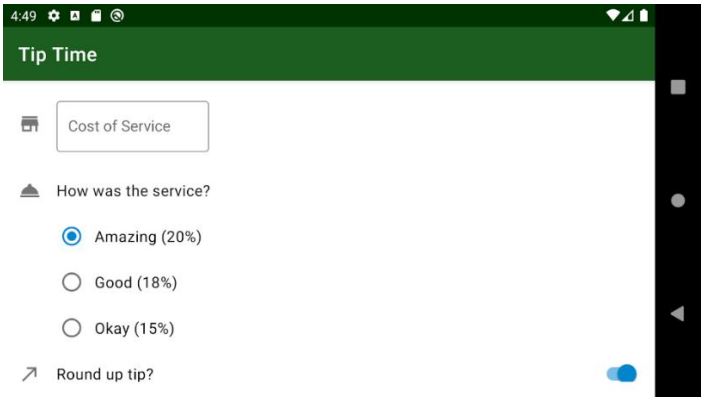
Rotating the device

1. Rotate your device to landscape mode. You may need to enable **Auto-rotate** setting first. (This is located under the device's Quick Settings or under **Settings > Display > Advanced > Auto-rotate screen** option.)



In the emulator, you can then use the emulator options (located on the upper right side adjacent the device) to rotate the screen to the right or left.

2. You'll notice some of the UI components including the **Calculate** button will get truncated. This clearly prevents you from using the app!



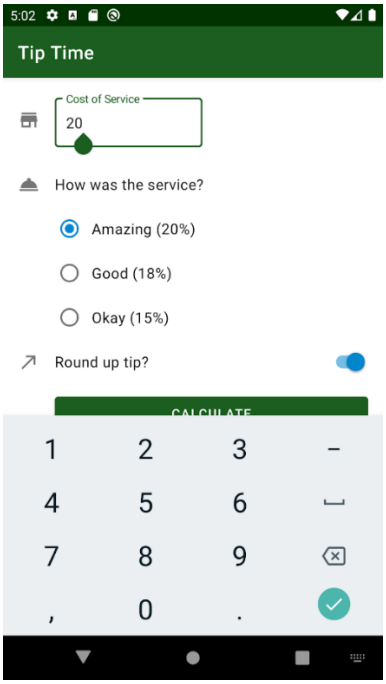
3. To solve this bug, add a `ScrollView` around the `ConstraintLayout`. Your XML will look something like this.

```
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_height="match_parent"
  android:layout_width="match_parent">

  <androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp"
    tools:context=".MainActivity">

    ...
  </ConstraintLayout>

</ScrollView>
```

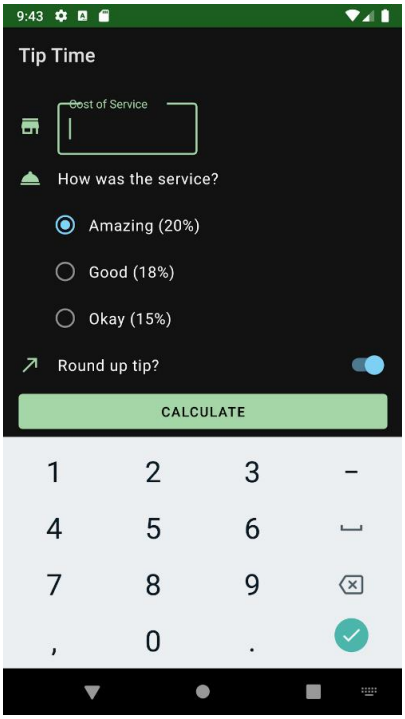
4. Run and test the app again. When you rotate the device to landscape mode, you should be able to scroll the UI to access the calculate button and see the tip result. This fix is not only useful for landscape mode, but also for other Android devices that may have different dimensions. Now regardless of the device screen size, the user can scroll the layout.

Hide keyboard on Enter key

You may have noticed that after you enter in a cost of service, the keyboard still stays up. It's a bit cumbersome to manually hide the keyboard each time to better access the calculate button. Instead, make the keyboard automatically hide itself when the Enter key is pressed.



For the text field, you can define a key listener to respond to events when certain keys are tapped. Every possible entry option on a keyboard has a key code associated with it, including the `Enter` key. Note that an onscreen keyboard is also known as a soft keyboard, as opposed to a physical keyboard.



In this task, set up a key listener on the text field to listen for when the `Enter` key is pressed. When that event is detected, hide the keyboard.

1. Copy and paste this helper method into your `MainActivity` class. You can insert it right before the closing brace of the `MainActivity` class. The `handleKeyEvent()` is a private helper function that hides the onscreen keyboard if the `keyCode` input parameter is equal to `KeyEvent.KEYCODE_ENTER`. The InputMethodManager controls if a soft keyboard is shown, hidden, and allows the user to choose which soft keyboard is displayed. The method returns true if the key event was handled, and returns false otherwise.

   `MainActivity.kt`

   ```
   private fun handleKeyEvent(view: View, keyCode: Int): Boolean {
     if (keyCode == KeyEvent.KEYCODE_ENTER) {
       // Hide the keyboard
       val inputMethodManager =
           getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
       inputMethodManager.hideSoftInputFromWindow(view.windowToken, 0)
       return true
     }
     return false
   }
   ```

2. Now attach a key listener on the `TextInputEditText` widget. Remember that you can access the `TextInputEditText` widget through the binding object as `binding.costOfServiceEditText`.

   Call the `setOnKeyListener()` method on the `costOfServiceEditText` and pass in an `OnKeyListener`. This is similar to how you set a click listener on the calculate button in the app with `binding.calculateButton.setOnClickListener { calculateTip() }`.

The code for setting a key listener on a view is a little more complex, but the general idea is that `OnKeyListener` has an `onKey()` method that gets triggered when a key press happens. The `onKey()` method takes in 3 input arguments: the view, the code for the key that was pressed, and a key event (which you won't use, so you can call it "_"). When the `onKey()` method is called, you should call your `handleKeyEvent()` method and pass along the view and key code arguments. The syntax for writing this out is: `view, keyCode, _ -> handleKeyEvent(view, keyCode)`. This is actually called a lambda expression, but you will learn more about lambdas in a later unit.

Add the code for setting up the key listener on the text field within the activity's `onCreate()` method. This is because you want your key listener to be attached as soon as the layout is created and before the user starts interacting with the activity.

`MainActivity.kt`

```
override fun onCreate(savedInstanceState: Bundle?) {
  ...

  setContentView(binding.root)

  binding.calculateButton.setOnClickListener { calculateTip() }

  binding.costOfServiceEditText.setOnKeyListener { view, keyCode, _ -> handleKeyEvent(view, keyCode)
  }
}
```

3. Test that your new changes work. Run the app and enter in a cost of service. Hit the Enter key on the keyboard and the soft keyboard should get hidden.

Test your app with Talkback enabled

As you've been learning throughout this course, you want to build apps that are accessible to as many users as possible. Some users may use Talkback to access and navigate your app. TalkBack is the Google screen reader included on Android devices. TalkBack gives you spoken feedback so that you can use your device without looking at the screen.

With Talkback enabled, ensure that a user can complete the use case of calculating tip within your app.

1. Enable Talkback on your device by following these instructions.
2. Return to the **Tip Time** app.
3. Explore your app with Talkback using these instructions. Swipe right to navigate through screen elements in sequence, and swipe left to go in the opposite direction. Double-tap anywhere to select. Verify that you can reach all elements of your app with swipe gestures.
4. Ensure that a Talkback user is able to navigate to each item on the screen, enter in a cost of service, change the tip options, calculate the tip, and hear the tip announced. Remember that no spoken feedback is provided for the icons since you marked those as `importantForAccessibility="no"`.

For more information on how to make your app more accessible, check out these principles.

(Optional) Adjust the tint of the vector drawables

In this optional task, you will tint the icons based on the primary color of the theme, so that the icons appear differently in light vs. dark theme (as seen below). This change is a nice addition to your UI to make the icons appear more cohesive with the app theme.

As we mentioned before, one of the advantages of `VectorDrawables` versus bitmap images is the ability to scale and tint them. Below we have the XML representing the bell icon. There are two specific color attributes to take notice of: `android:tint` and `android:fillColor`.

`ic_service.xml`

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
  android:width="24dp"
  android:height="24dp"
  android:viewportWidth="24"
  android:viewportHeight="24"
  android:tint="?attr/colorControlNormal">
 <path
    android:fillColor="@android:color/white"
    android:pathData="M2,17h20v2L2,19zM13.84,7.79c0.1,-0.24 0.16,-0.51 0.16,-0.79 0,-1.1 -0.9,-2 -2,-2s-2,0.9 -2,2c0,0.28
0.06,0.55 0.16,0.79C6.25,8.6 3.27,11.93 3,16h18c-0.27,-4.07 -3.25,-7.4 -7.16,-8.21z"/>
</vector>
```

If there is a tint present, it will override any `fillColor` directives for the drawable. In this case, the white color is overridden with the `colorControlNormal` theme attribute. `colorControlNormal` is the color of the "normal" (unselected/unactivated state) of a widget. Currently that's a gray color.

One visual enhancement we can make to the app is to tint the drawable based on the primary color of the app theme. For light theme, the icon will appear as `@color/green`, whereas in dark theme, the icon will appear as `@color/green_light`, which is the `?attr/colorPrimary`. Tinting the drawable based on the primary color of the app theme can make the elements in the layout appear more unified and cohesive. This also saves us from having to duplicate the set of icons for light theme and dark theme. There's only 1 set of vector drawables, and the tint will change based on the `colorPrimary` theme attribute.

1. Change the value of the `android:tint` attribute in `ic_service.xml`

   android:tint="?attr/colorPrimary"

   In Android Studio, that icon now has the proper tint.

The value that the `colorPrimary` theme attribute points to will differ depending on light vs. dark theme.

2. Repeat the same for changing the tint on the other vector drawables.

`ic_store.xml`

```
<vector ...
  android:tint="?attr/colorPrimary">
  ...
</vector>
```

`ic_round_up.xml`

```
<vector ...
  android:tint="?attr/colorPrimary">
  ...
</vector>
```

3. Run the app. Verify that the icons appear differently in light vs. dark themes.
4. As a final cleanup step, remember to reformat all XML and Kotlin code files in your app.

Congratulations, you have finally completed the tip calculator app! You should be very proud of what you've built. Hopefully this is the stepping stone for you to build even more beautiful and functional apps!

8. Summary

- Use Material Design Components where possible to adhere to Material Design guidelines and allow for more customization.
- Add icons to give users visual cues about how parts of your app will function.
- Use `ConstraintLayout` to position elements in your layout.
- Test your app for edges cases (e.g. rotating your app in landscape mode) and make improvements where applicable.
- Comment your code to help other people who are reading your code understand what your approach was.
- Reformat your code and clean up your code to make it as concise as possible.

UNIT 2 : MODULE 3

1. Before you begin

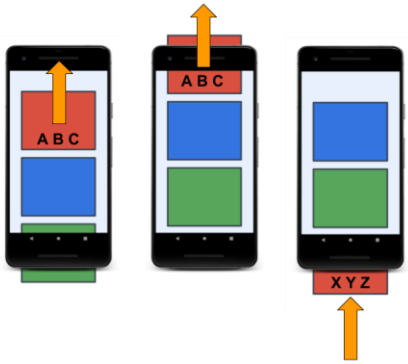
If you think about the apps you commonly use on your phone, almost every single app has at least one list. The call history screen, the contacts app, and your favorite social media app all display a list of data. As shown in the screenshot below, some of these apps display a simple list of words or phrases, where others display more complex items such as cards that include text and images. No matter what the content is, displaying a list of data is one of the most common UI tasks in Android.

1. Before you begin

If you think about the apps you commonly use on your phone, almost every single app has at least one list. The call history screen, the contacts app, and your favorite social media app all display a list of data. As shown in the screenshot below, some of these apps display a simple list of words or phrases, where others display more complex items such as cards that include text and images. No matter what the content is, displaying a list of data is one of the most common UI tasks in Android.

To help you build apps with lists, Android provides the `RecyclerView`. `RecyclerView` is designed to be very efficient, even with large lists, by reusing, or recycling, the views that have scrolled off the screen. When a list item is scrolled off the screen, `RecyclerView` reuses that view for the next list item about to be displayed. That means, the item is filled with new content that scrolls onto the screen. This `RecyclerView` behavior saves a lot of processing time and helps lists scroll more smoothly.

In the sequence shown below, you can see that one view has been filled with data, `ABC`. After that view scrolls off the screen, `RecyclerView` reuses the view for new data, `XYZ`.



In this codelab, you will build the Affirmations app. Affirmations is a simple app that displays ten positive affirmations as text in a scrolling list. Then, in the follow-up codelab, you will take it a step further, add an inspiring image to each affirmation, and polish the app UI.

What you'll learn

- How to use a `RecyclerView` to display a list of data.
- How to organize your code into packages
- How to use adapters with `RecyclerView` to customize how an individual list item looks.

3. Setting up the list of data

The next step in creating the Affirmations app is to add resources. You will add the following to your project.

- String resources to display as affirmations in the app.
- A source of data to provide a list of affirmations to your app.

Note: In most production Android projects, you would retrieve the affirmations data from a database or from a server. Networking and databases are beyond the scope of this codelab, so you will use a list of affirmations strings defined inside the app.

Add Affirmation strings

1. In the **Project** window, open **app > res > values > strings.xml**. This file currently has a single resource which is the name of the app.
2. In `strings.xml`, add the following affirmations as individual string resources. Name them `affirmation1`, `affirmation2`, and so on.

   The `strings.xml` file should look like this when you're done.

```
<resources>
    <string name="app_name">Affirmations</string>
    <string name="affirmation1">I am strong.</string>
    <string name="affirmation2">I believe in myself.</string>
    <string name="affirmation3">Each day is a new opportunity to grow and be a better version of myself.</string>
    <string name="affirmation4">Every challenge in my life is an opportunity to learn from.</string>
    <string name="affirmation5">I have so much to be grateful for.</string>
    <string name="affirmation6">Good things are always coming into my life.</string>
    <string name="affirmation7">New opportunities await me at every turn.</string>
    <string name="affirmation8">I have the courage to follow my heart.</string>
    <string name="affirmation9">Things will unfold at precisely the right time.</string>
    <string name="affirmation10">I will be present in all the moments that this day brings.</string>
</resources>
```

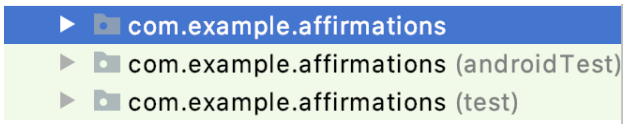
Now that you have added string resources, you can reference them in your code as `R.string.affirmation1` or `R.string.affirmation2`.

Create a new package

Organizing your code logically helps you and other developers understand, maintain, and extend it. In the same way that you can organize paperwork into files and folders, you can organize your code into files and packages.

What is a package?

1. In Android Studio, in the **Project** window (**Android**), take a look at your new project files under **app > java** for the Affirmations app. They should look similar to the screenshot below, which shows three packages, one for your code (**com.example.affirmations**), and two for test files (**com.example.affirmations (androidTest)** and **com.example.affirmations (test)**).



2. Notice that the name of the package consists of several words separated by a period.

There are two ways in which you can make use of packages.

- Create different packages for different parts of your code. For example, developers will often separate the classes that work with data and the classes that build the UI into different packages.
- Use code from other packages in your code. In order to use the classes from other packages, you need to define them in your build system's dependencies. It's also a standard practice to `import` them in your code so you can use their shortened names (eg, `TextView`) instead of their fully-qualified names (eg, `android.widget.TextView`). For example, you have already used `import` statements for classes such as `sqrt` (`import kotlin.math.sqrt`) and `View` (`import android.view.View`).

In the Affirmations app, in addition to importing Android and Kotlin classes, you will also organize your app into several packages. Even when you don't have a lot of classes for your app, it is a good practice to use packages to group classes by functionality.

Naming packages

A package name can be anything, as long as it is globally unique; no other published package anywhere can have the same name. Because there are a very large number of packages, and coming up with random unique names is hard, programmers use conventions to make it easier to create and understand package names.

- The package name is usually structured from general to specific, with each part of the name in lowercase letters and separated by a period. Important: The period is just part of the name. It does not indicate a hierarchy in code or mandate a folder structure!
- Because internet domains are globally unique, it is a convention to use a domain, usually yours or the domain of your business, as the first part of the name.
- You can choose the names of packages to indicate what's inside the package, and how packages are related to each other.
- For code examples like this one, `com.example` followed by the name of the app is commonly used.

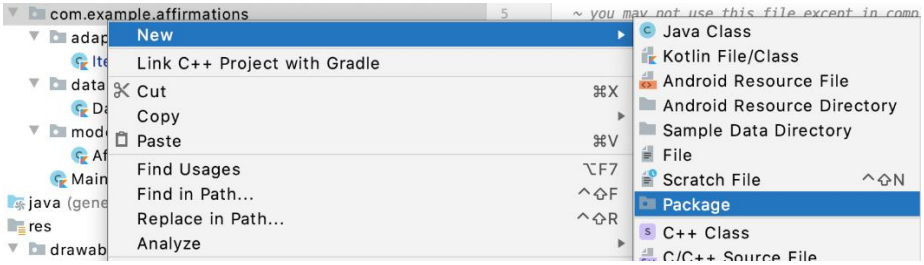
Here are some examples of predefined package names and their contents:

- `kotlin.math` - Mathematical functions and constants.
- `android.widget` - Views, such as `TextView`.

Note: While the names of packages (and their arrangement in the Android **Project** window of Android Studio as a hierarchy of folders) appear as a hierarchy, there is no actual hierarchy in the executable code. Just like the numbering system of a library categorizes and organizes books, they are still all on the same shelf, and you can take out any of them.

Create a package

1. In Android Studio, in the **Project** pane, right-click **app > java > com.example.affirmations** and select **New > Package**.



2. In the **New Package** popup, notice the suggested package name prefix. The suggested first part of the package name is the name of the package you right-clicked. While package names do not create a hierarchy of packages, reusing parts of the name is used to indicate a relationship and organization of the content!
3. In the popup, append **model** to the end of the suggested package name. Developers often use **model** as the package name for classes that model (or represent) the data.

4. Press **Enter**. This creates a new package under the **com.example.affirmations** (root) package. This new package will contain any data-related classes defined in your app.

Create the Affirmation data class

In this task, you'll create a class called `Affirmation`. An object instance of `Affirmation` represents one affirmation and contains the resource ID of the string with the affirmation.

5. Right-click on the **com.example.affirmations.model** package and select **New > Kotlin File/Class.**

6. In the popup, select **Class** and enter `Affirmation` as the name of the class. This creates a new file called `Affirmation.kt` in the `model` package.
7. Make `Affirmation` a data class by adding the `data` keyword before the class definition. This leaves you with an error, because data classes must have at least one property defined.

`Affirmation.kt`

```
package com.example.affirmations.model

data class Affirmation {
}
```

When you create an instance of `Affirmation`, you need to pass in the resource ID for the affirmation string. The resource ID is an integer.

8. Add a `val` integer parameter `stringResourceId` to the constructor of the `Affirmation` class. This gets rid of your error.

```
package com.example.affirmations.model

data class Affirmation(val stringResourceId: Int)
```

Create a class to be a data source

Data displayed in your app may come from different sources (e.g. within your app project or from an external source that requires connecting to the internet to download data). As a result, data may not be in the exact format that you need. The rest of the app should not concern itself with where the data originates from or in what format it is originally. You can and should hide away this data preparation in a separate `Datasource` class that prepares the data for the app.

Since preparing data is a separate concern, put the `Datasource` class in a separate **data** package.

1. In Android Studio, in the **Project** window, right-click **app > java > com.example.affirmations** and select **New > Package**.
2. Enter `data` as the last part of the package name.
3. Right click on the `data` package and select **new Kotlin File/Class.**
4. Enter `Datasource` as the class name.
5. Inside the `Datasource` class, create a function called `loadAffirmations()`.

   The `loadAffirmations()` function needs to return a list of `Affirmations`. You do this by creating a list and populating it with an `Affirmation` instance for each resource string.

6. Declare `List<Affirmation>` as the return type of the method `loadAffirmations()`.
7. In the body of `loadAffirmations()`, add a `return` statement.
8. After the `return` keyword, call `listOf<>()` to create a `List`.
9. Inside the angle brackets `<>`, specify the type of the list items as `Affirmation`. If necessary, import `com.example.affirmations.model.Affirmation`.
10. Inside the parentheses, create an `Affirmation`, passing in `R.string.affirmation1` as the resource ID as shown below.

```
Affirmation(R.string.affirmation1)
```

11. Add the remaining `Affirmation` objects to the list of all affirmations, separated by commas. The finished code should look like the following.

`Datasource.kt`

```
package com.example.affirmations.data

import com.example.affirmations.R
import com.example.affirmations.model.Affirmation
```

```
class Datasource {

    fun loadAffirmations(): List<Affirmation> {
        return listOf<Affirmation>(
            Affirmation(R.string.affirmation1),
            Affirmation(R.string.affirmation2),
            Affirmation(R.string.affirmation3),
            Affirmation(R.string.affirmation4),
            Affirmation(R.string.affirmation5),
            Affirmation(R.string.affirmation6),
            Affirmation(R.string.affirmation7),
            Affirmation(R.string.affirmation8),
            Affirmation(R.string.affirmation9),
            Affirmation(R.string.affirmation10)
        )
    }
}
```

[Optional] Display the size of the Affirmations list in a TextView

To verify that you can create a list of affirmations, you can call `loadAffirmations()` and display the size of the returned list of affirmations in the `TextView` that comes with your Empty Activity app template.

1. In `layouts/activity_main.xml`, give the `TextView` that comes with your template an `id` of `textview`.
2. In `MainActivity` in the `onCreate()` method after the existing code, get a reference to `textview`.

```
val textView: TextView = findViewById(R.id.textview)
```

3. Then add code to create and display the size of the affirmations list. Create a `Datasource`, call `loadAffirmations()`, get the size of the returned list, convert it to a string, and assign it as the `text` of `textView`.

```
textView.text = Datasource().loadAffirmations().size.toString()
```

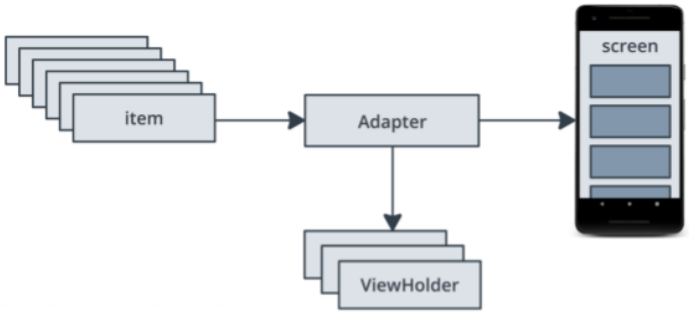
5. Delete the code you just added in `MainActivity`.

4. Adding a RecyclerView to your app

In this task, you will set up a `RecyclerView` to display the list of `Affirmations`.

There are a number of pieces involved in creating and using a `RecyclerView`. You can think of them as a division of labor. The diagram below shows an overview, and you will learn more about each piece as you implement it.

- **item** - One data item of the list to display. Represents one `Affirmation` object in your app.
- **Adapter** - Takes data and prepares it for `RecyclerView` to display.
- **ViewHolders** - A pool of views for `RecyclerView` to use and reuse to display affirmations.
- **RecyclerView** - Views on screen

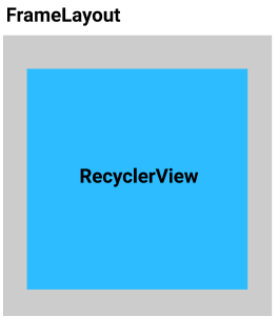


Add a RecyclerView to the layout

The Affirmations app consists of a single activity named `MainActivity`, and its layout file is called `activity_main.xml`. First, you need to add the `RecyclerView` to the layout of the `MainActivity`.

1. Open `activity_main.xml` (**app > res > layout > activity_main.xml**)
2. If you are not already using it, switch to **Split view**.
3. Delete the `TextView`.

The current layout uses `ConstraintLayout`. `ConstraintLayout` is ideal and flexible when you want to position multiple child views in a layout. Since your layout only has a single child view, `RecyclerView`, you can switch to a simpler `ViewGroup` called `FrameLayout` that should be used for holding a single child view.



4. In the XML, replace `ConstraintLayout` with `FrameLayout`.
5. Switch to **Design** view.
6. In the **Palette**, select **Containers**, and find the **RecyclerView**.
7. Drag a **RecyclerView** into the layout.
8. If it appears, read the **Add Project Dependency** popup and click **OK**. (If the popup doesn't appear, no action is needed.)
9. Wait for Android Studio to finish and the `RecyclerView` to appear in your layout.
10. If necessary, change the `layout_width` and `layout_height` attributes of the `RecyclerView` to `match_parent` so that the `RecyclerView` can fill the whole screen.
11. Set the resource ID of the `RecyclerView` to `recycler_view`.

`RecyclerView` supports displaying items in different ways, such as a linear list or a grid. Arranging the items is handled by a `LayoutManager`. The Android framework provides layout managers for basic item layouts. The Affirmations app displays items as a vertical list, so you can use the `LinearLayoutManager`.

12. Switch back to **Code** view. In the XML code, inside the `RecyclerView` element, add `LinearLayoutManager` as the layout manager attribute of the `RecyclerView`, as shown below.

    ```
    app:layoutManager="LinearLayoutManager"
    ```

    To be able to scroll through a vertical list of items that is longer than the screen, you need to add a vertical scrollbar.

13. Inside `RecyclerView`, add an `android:scrollbars` attribute set to `vertical`.

    ```
    android:scrollbars="vertical"
    ```

    The final XML layout should look like the following:

    `activity_main.xml`

    ```
    <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       xmlns:tools="http://schemas.android.com/tools"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       tools:context=".MainActivity">

       <androidx.recyclerview.widget.RecyclerView
          android:id="@+id/recycler_view"
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          android:scrollbars="vertical"
          app:layoutManager="LinearLayoutManager" />
    </FrameLayout>
    ```

14. Run your app.

    The project should compile and run without any issues. However, only a white background is displayed in your app because you are missing a crucial piece of code. Right now, you have the source of data and the `RecyclerView` added to your layout, but the `RecyclerView` has no information on how to display the `Affirmation` objects.

Implement an Adapter for the RecyclerView

Your app needs a way to take the data from `Datasource`, and format it so that each `Affirmation` can be displayed as an item in the `RecyclerView`.

Adapter is a design pattern that adapts the data into something that can be used by `RecyclerView`. In this case, you need an adapter that takes an `Affirmation` instance from the list returned by `loadAffirmations()`, and turns it into a list item view, so that it can be displayed in the `RecyclerView`.

When you run the app, `RecyclerView` uses the adapter to figure out how to display your data on screen. `RecyclerView` asks the adapter to create a new list item view for the first data item in your list. Once it has the view, it asks the adapter to provide the data to draw the item. This process repeats until the `RecyclerView` doesn't

need any more views to fill the screen. If only 3 list item views fit on the screen at once, the `RecyclerView` only asks the adapter to prepare those 3 list item views (instead of all 10 list item views).

In this step, you'll build an adapter which will adapt an `Affirmation` object instance so that it can be displayed in the `RecyclerView`.

Create the Adapter

An adapter has multiple parts, and you'll be writing quite a bit of code that's more complex than what you've done in this course so far. It's okay if you don't fully understand the details at first. Once you have completed this whole app with a `RecyclerView`, you'll be able to better understand how all the parts fit together. You'll also be able to reuse this code as a base for future apps that you create with a `RecyclerView`.

Create a layout for items

Each item in the `RecyclerView` has its own layout, which you define in a separate layout file. Since you are only going to display a string, you can use a `TextView` for your item layout.

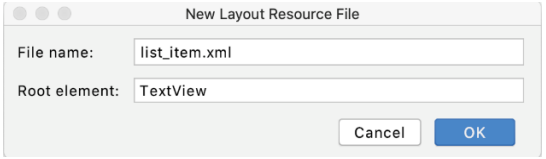
1. In **res > layout**, create a new empty **File** called `list_item.xml`.
2. Open `list_item.xml` in **Code** view.
3. Add a `TextView` with `id` `item_title`.
4. Add `wrap_content` for the `layout_width` and `layout_height`, as shown in the code below.

Notice that you don't need a `ViewGroup` around your layout, because this list item layout will later be inflated and added as a child to the parent `RecyclerView`.

`list_item.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/item_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Alternatively, you could have used **File > New > Layout Resource File**, with **File name** `list_item.xml` and `TextView` as the **Root element**. Then update the generated code to match the code above.



Create an ItemAdapter class

1. In Android Studio in the **Project** pane, right-click **app > java > com.example.affirmations** and select **New > Package**.
2. Enter `adapter` as the last part of the package name.
3. Right-click on the `adapter` package and select **New > Kotlin File/Class.**
4. Enter `ItemAdapter` as the class name, finish, and the `ItemAdapter.kt` file opens.

You need to add a parameter to the constructor of `ItemAdapter`, so that you can pass the list of affirmations into the adapter.

5. Add a parameter to the `ItemAdapter` constructor that is a `val` called `dataset` of type `List<Affirmation>`. Import `Affirmation`, if necessary.
6. Since the `dataset` will be only used within this class, make it `private`.

`ItemAdapter.kt`

```
import com.example.affirmations.model.Affirmation
class ItemAdapter(private val dataset: List<Affirmation>) {
}
```

The `ItemAdapter` needs information on how to resolve the string resources. This, and other information about the app, is stored in a `Context` object instance that you can pass into an `ItemAdapter` instance.

7. Add a parameter to the `ItemAdapter` constructor that is a `val` called `context` of type `Context`. Position it as the first parameter in the constructor.

```
class ItemAdapter(private val context: Context, private val dataset: List<Affirmation>) {
}
```

Create a ViewHolder

`RecyclerView` doesn't interact directly with item views, but deals with `ViewHolders` instead. A `ViewHolder` represents a single list item view in `RecyclerView`, and can be reused when possible. A `ViewHolder` instance holds references to the individual views within a list item layout (hence the name "view holder"). This makes it easier to update the list item view with new data. View holders also add information that `RecyclerView` uses to efficiently move views around the screen.

1. Inside the `ItemAdapter` class, before the closing curly brace for `ItemAdapter`, create an `ItemViewHolder` class.

```
class ItemAdapter(private val context: Context, private val dataset: List<Affirmation>) {
    class ItemViewHolder()
}
```

- Defining a class inside another class is called creating a **nested class**.
- Since `ItemViewHolder` is only used by `ItemAdapter`, creating it inside `ItemAdapter` shows this relationship. This is not mandatory, but it helps other developers understand the structure of your program.

2. Add a `private` `val` `view` of type `View` as a parameter to the `ItemViewHolder` class constructor.
3. Make `ItemViewHolder` a subclass of `RecyclerView`. `ViewHolder` and pass the `view` parameter into the superclass constructor.
4. Inside `ItemViewHolder`, define a `val` property `textView` that is of type `TextView`. Assign it the view with the ID `item_title` that you defined in `list_item.xml`.

6. Summary

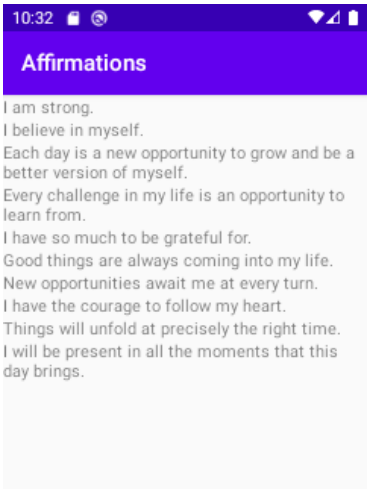
- `RecyclerView` widget helps you display a list of data.
- `RecyclerView` uses the adapter pattern to adapt and display the data.
- `ViewHolder` creates and holds the views for `RecyclerView`.
- `RecyclerView` comes with built in `LayoutManagers`. `RecyclerView` delegates how items are laid out to `LayoutManagers`.

To implement the adapter:

- Create a new class for the adapter, for example, `ItemAdapter`.
- Create a custom `ViewHolder` class that represents a single list item view. Extend from `RecyclerView.ViewHolder` class.
- Modify the `ItemAdapter` class to extend from the `RecyclerView.Adapter` class with the custom `ViewHolder` class.
- Implement these methods within the adapter: `getItemsCount()`, `onCreateViewHolder()`, and `onBindViewHolder()`.

1. Before you begin

In the previous codelab, you created an Affirmations app that displays a list of text in a `RecyclerView`.



In this follow-up codelab, you add an inspiring image to each affirmation of your app. You will display the text and image for each affirmation within a card, using the `MaterialCardView` widget from the Material Components for Android library. Then you will finish the app by polishing the UI to create a more cohesive and beautiful user experience. This is a screenshot of the completed app:

What you'll learn

- How to add images to the list of displayed affirmations in a `RecyclerView`.
- How to use `MaterialCardView` in a `RecyclerView` item layout.
- How to make visual changes in the UI to make the app look more polished.

2. Adding images to the list items

So far you have created an adapter `ItemAdapter` to display affirmation strings in a `RecyclerView`. Functionally this works great, but visually it is not very appealing. In this task, you will modify the list item layout and adapter code to display images with the affirmations.

Download the images

1. To start, open up the Affirmations app project in Android Studio from the previous codelab. If you don't have this project, go through the steps in the previous codelab to create that project. Then return here.
2. Next download the image files onto your computer. There should be ten images, one for each affirmation in your app. The files should be named from `image1.jpg` to `image10.jpg`.
3. Copy the images from your computer into your project's **res > drawable** folder (`app/src/main/res/drawable`) within Android Studio. Once these resources have been added to your app, you will be able to access these images from your code using their resource IDs, such as `R.drawable.image1`. (You may have to rebuild your code for Android Studio to find the image.)

Now the images are ready to use in the app.

Add support for images in the Affirmation class

In this step, you'll add a property in the `Affirmation` data class to hold a value for an image resource ID. That way a single `Affirmation` object instance will contain a resource ID for the text of the affirmation and a resource ID for the image of the affirmation.

1. Open the `Affirmation.kt` file within the `model` package.
2. Modify the constructor of the `Affirmation` class by adding another `Int` parameter named `imageResourceId`.

Using resource annotations

Both `stringResourceId` and `imageResourceId` are integer values. Although this looks okay, the caller could accidentally pass in the arguments in the wrong order (`imageResourceId` first instead of `stringResourceId`).

To avoid this, you can use Resource annotations. Annotations are useful because they add additional info to classes, methods, or parameters. Annotations are always declared with an @ symbol. In this case, add the `@StringRes` annotation to your string resource ID property and `@DrawableRes` annotation to your drawable resource ID property. Then you will get a warning if you supply the wrong type of resource ID.

1. Add the `@StringRes` annotation to `stringResourceId.`
2. Add the `@DrawableRes` annotation to `imageResourceId`.
3. Make sure the imports `androidx.annotation.DrawableRes` and `androidx.annotation.StringRes` are added at the top of your file after the package declaration.

`Affirmation.kt`

```
package com.example.affirmations.model
import androidx.annotation.DrawableRes
import androidx.annotation.StringRes
data class Affirmation(
  @StringRes val stringResourceId: Int,
  @DrawableRes val imageResourceId: Int
)
```

Initialize list of affirmations with images

Now that you've changed the constructor of the `Affirmation` class, you need to update the `Datasource` class. Pass in an image resource ID to each `Affirmation` object that is initialized.

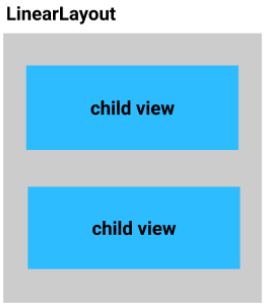
1. Open `Datasource.kt`. You should see an error for each instantiation of `Affirmation`.
2. For each `Affirmation`, add the resource ID of an image as an argument, such as `R.drawable.image1`.

`Datasource.kt`

```
package com.example.affirmations.data
import com.example.affirmations.R
import com.example.affirmations.model.Affirmation
class Datasource() {
    fun loadAffirmations(): List<Affirmation> {
        return listOf<Affirmation>(
            Affirmation(R.string.affirmation1, R.drawable.image1),
            Affirmation(R.string.affirmation2, R.drawable.image2),
            Affirmation(R.string.affirmation3, R.drawable.image3),
            Affirmation(R.string.affirmation4, R.drawable.image4),
            Affirmation(R.string.affirmation5, R.drawable.image5),
            Affirmation(R.string.affirmation6, R.drawable.image6),
            Affirmation(R.string.affirmation7, R.drawable.image7),
            Affirmation(R.string.affirmation8, R.drawable.image8),
            Affirmation(R.string.affirmation9, R.drawable.image9),
            Affirmation(R.string.affirmation10, R.drawable.image10)
        )
    }
}
```

Add an ImageView to the list item layout

To show an image for each affirmation in your list, you need to add an `ImageView` to your item layout. Because you now have two views (a `TextView` and `ImageView`), you need to place them as children views within a `ViewGroup`. To arrange the views in a vertical column, you can use a `LinearLayout`. `LinearLayout` aligns all child views in a single direction, vertically or horizontally.



1. Open **res > layout > list_item.xml**. Add a `LinearLayout` around the existing `TextView` and set the `orientation` property to `vertical`.
2. Move the `xmlns schema` declaration line from the `TextView` element to the `LinearLayout` element to get rid of the error.

list_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:id="@+id/item_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

3. Inside the `LinearLayout`, before the `TextView`, add an `ImageView` with a resource ID of `item_image`.
4. Set the `ImageView`'s width to `match_parent` and height to `194dp`. Depending on screen size, this value should show a few cards on screen at any given time.
5. Set the `scaleType` to `centerCrop`.
6. Set the `importantForAccessibility` attribute to `no` since the image is used for decorative purposes.

```
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="194dp"
        android:id="@+id/item_image"
        android:importantForAccessibility="no"
        android:scaleType="centerCrop" />
```

Update the ItemAdapter to set the image

1. Open `adapter/ItemAdapter.kt` (**app > java > adapter > ItemAdapter**)
2. Go to the `ItemViewHolder` class.
3. An `ItemViewHolder` instance should hold a reference to the `TextView` and a reference to the `ImageView` in the list item layout. Make the following change.

   Below the initialization of the `textView` property, add a `val` called `imageView`. Use `findViewById()` to find the reference to the `ImageView` with ID `item_image` and assign it to the `imageView` property.

   ItemAdapter.kt

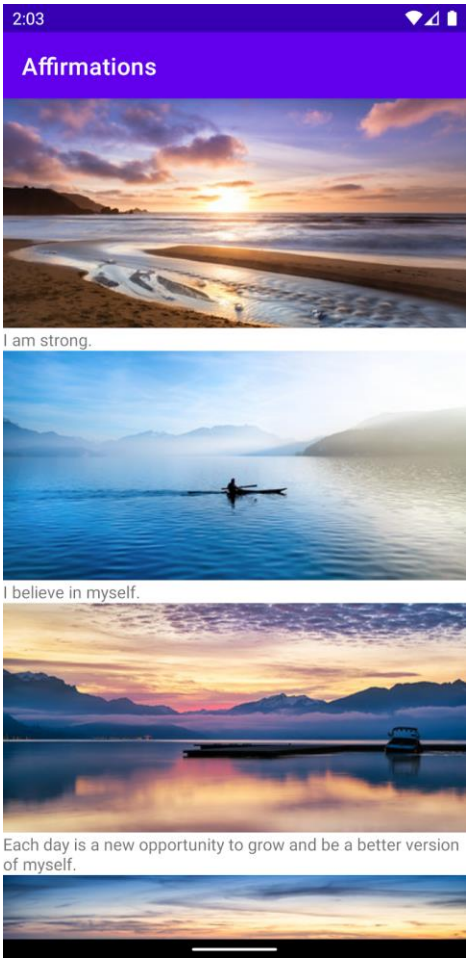
```
class ItemViewHolder(private val view: View): RecyclerView.ViewHolder(view) {
    val textView: TextView = view.findViewById(R.id.item_title)
    val imageView: ImageView = view.findViewById(R.id.item_image)
}
```

4. Find the `onBindViewHolder()` function in `ItemAdapter`.
5. Previously you set the affirmation's `stringResourceId` on to `textView` in the `ItemViewHolder`. Now set the affirmation item's `imageResourceId` onto the `ImageView` of the list item view.

```
    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
        val item = dataset[position]
```

```
        holder.textView.text = context.resources.getString(item.stringResourceId)
        holder.imageView.setImageResource(item.imageResourceId)
    }
```

6. Run the app and scroll through the list of affirmations.



The app looks much prettier with images! Yet you can still improve the app UI. In the next section you will make small adjustments to the app to improve the UI.

3. Polishing the UI

So far you've built a functional app that consists of a list of affirmation strings and images. In this section, you'll see how small changes in the code and XML can make the app look more polished.

Add padding

To start with, add some whitespace between the items in the list.

Tip: You can make layout changes in the XML as shown here, or you can make them in the **Attributes** panel in **Design** view, whichever you prefer.

1. Open `list_item.xml` (**app > res > layout > item_list.xml**) and add `16dp` padding to the existing `LinearLayout`.

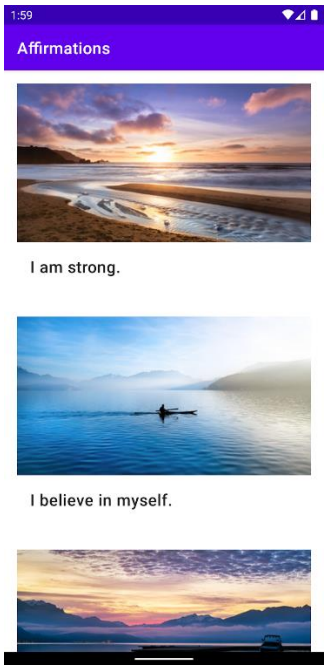
`list_item.xml`

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">
```

2. Add `16dp` padding to the `item_title` `TextView`.
3. In the `TextView`, set the `textAppearance` attribute to `?attr/textAppearanceHeadline6`. `textAppearance` is an attribute that allows you to define text-specific styling. For other possible predefined text appearance values, you can see the TextAppearances section in this blogpost on Common Theme Attributes.

```
<TextView
    android:id="@+id/item_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textAppearance="?attr/textAppearanceHeadline6" />
```

4. Run the app. Do you think the list looks better?



Use cards

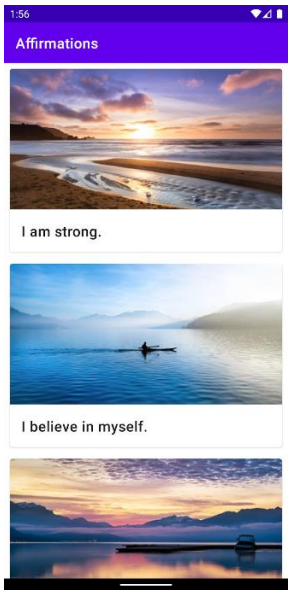
It is still hard to tell if an image belongs to the affirmation text above or below that image. To fix this you can use a **Card** view. A Card view provides an easy way to contain a group of views while providing a consistent style for the container. For more Material Design guidance on using cards, check out this guide on cards.

1. Add a `MaterialCardView` around the existing `LinearLayout`.
2. Once again, move the schema declaration from `LinearLayout` into `MaterialCardView`.
3. Set the `layout_width` of the `MaterialCardView` to `match_parent`, and the `layout_height` to `wrap_content`.
4. Add a `layout_margin` of `8dp`.

5. Remove the padding in the `LinearLayout`, so you don't have too much whitespace.
6. Now run the app again. Can you tell each affirmation apart better with `MaterialCardView`?

`list_item.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <ImageView
            android:id="@+id/item_image"
            android:layout_width="match_parent"
            android:layout_height="194dp"
            android:importantForAccessibility="no"
            android:scaleType="centerCrop" />

        <TextView
            android:id="@+id/item_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:padding="16dp"
            android:textAppearance="?attr/textAppearanceHeadline6" />

    </LinearLayout>
</com.google.android.material.card.MaterialCardView>
```



Change the app theme colors

The default app theme color may not be as calming as some other choices you could make. In this task, you will change the app theme color to blue. You can then change it again using your own ideas!

You can find pre-defined shades of blue from the Material Design color palette from this link.

In this codelab, you will be using the following colors from the Material Design palette:

- blue_200: `#FF90CAF9`
- blue_500: `#FF2196F3`
- blue_700: `#FF1976D2`

Add color resources

Define colors used within your app in a centralized place: the `colors.xml` file.

1. Open `colors.xml` (**res > color > colors.xml**).
2. Add new color resources to the file for the blue colors defined above.

```
<color name="blue_200">#FF90CAF9</color>
<color name="blue_500">#FF2196F3</color>
<color name="blue_700">#FF1976D2</color>
```

Change the theme colors

Now that you have new color resources, you can use them in your theme.

1. Open `themes.xml` (**res > values > themes > themes.xml**).
2. Find the `<!-- Primary brand color. -->` section.
3. Add or change `colorPrimary` to use `@color/blue_500`.
4. Add or change `colorPrimaryVariant` to use `@color/blue_700`.

```
<item name="colorPrimary">@color/blue_500</item>
<item name="colorPrimaryVariant">@color/blue_700</item>
```

5. Run the app. You should see the app bar color is changed to blue.

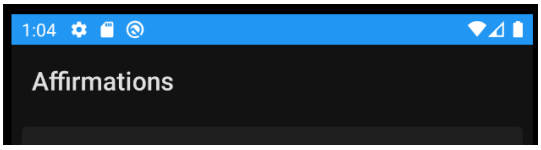
Update the dark theme colors

It's good to choose more desaturated colors for the dark theme of the app.

1. Open the dark theme `themes.xml` file (**themes > themes.xml (night)**).
2. Add or change the `colorPrimary` and `colorPrimaryVariant` theme attributes as follows.

```
<item name="colorPrimary">@color/blue_200</item>
<item name="colorPrimaryVariant">@color/blue_500</item>
```

3. Run your app.
4. In the **Settings** of your device, turn on the **Dark Theme**.
5. Your app switches to the **Dark Theme** and verify that it looks like the below screenshot.



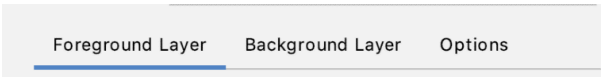
Note: app bar color**.** You may be wondering why your specified primary color for the dark theme is not showing in the app bar. The dark app bar is by design. In dark theme, the app bar and other large areas are by default shown with a dark background (`colorSurface`) instead of the primary color. This is because Material dark theme recommends less use of bright colors on large surfaces. Buttons or other small accents will show the defined primary color.

6. At this point, you can also remove unused colors in your `colors.xml` file (for example, the purple color resources used in the default app theme).

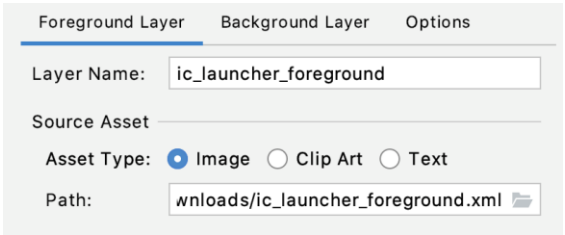
Change the app icon

As a final step, you'll update the app icon.

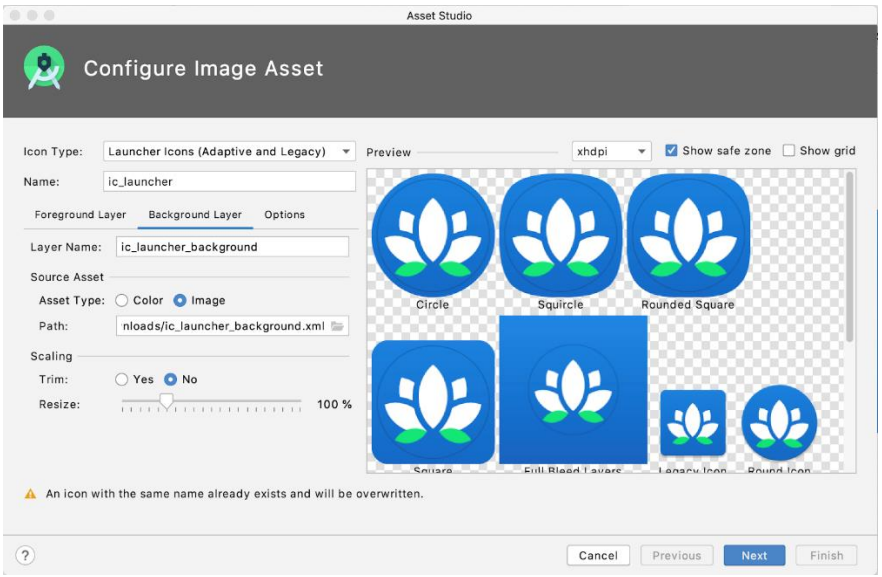
1. Download the app icon files `ic_launcher_foreground.xml` and `ic_launcher_background.xml`. If your browser shows the file instead of downloading it, select **FIle > Save Page As...** to save it to your computer.
2. Within Android Studio, delete two files: `drawable/ic_launcher_background.xml` and `drawable-v24/ic_launcher_foreground.xml` files since those are for the previous app icon. You can uncheck the box **Safe delete (with usage search)**.
3. Then right click on the **res > drawable** folder and select **New > Image Asset**.
4. In the **Configure Image Asset** window make sure **Foreground layer** is selected.



5. Below that, find the **Path** label.
6. Click the folder icon inside the **Path** text box.
7. Find and open the `ic_launcher_foreground.xml` file that you downloaded on your computer.



8. Switch to the **Background Layer** tab.
9. Click the **Browse** icon inside the **Path** text box.
10. Find and open the `ic_launcher_background.xml` file on your computer. No other changes are necessary.
11. Click **Next**.



12. In the **Confirm Icon Path** dialog, click **Finish**. It's OK to overwrite the existing icons.
13. For best practices, you can move the new vector drawables `ic_launcher_foreground.xml` and `ic_launcher_background.xml` into a new resource directory called `drawable-anydpi-v26`. Adaptive icons were introduced in API 26, so these resources will only be used on devices running API 26 and above (for any dpi).
14. Delete the `drawable-v24` directory if there's nothing left there.
15. Run your app and notice the beautiful new app icon in the app drawer!
16. As a last step, don't forget to reformat the Kotlin and XML files in the project so your code is cleaner and follows style guidelines.

    Congratulations! You created an inspiring Affirmations app.

5. Summary

- To display additional content in a `RecyclerView`, modify the underlying data model class and data source. Then update the list item layout and adapter to set that data onto the views.
- Use resource annotations to help ensure that the right type of resource ID is passed into a class constructor.
- Use the **Material Components for Android library** to have your app more easily follow the recommended Material Design guidelines.
- Use `MaterialCardView` to display content in a Material card.
- Small visual tweaks to your app in terms of color and spacing can make the app look more polished and consistent.