

Android Development

- The `onCreate()` function is the entry point to this Android app and calls other functions to build the user interface. In Kotlin programs, the `main()` function is the entry point/starting point of execution. In Android apps, the `onCreate()` function fills that role.
- The `setContent()` function within the `onCreate()` function is used to define your layout through composable functions. All functions marked with the `@Composable` annotation can be called from the `setContent()` function or from other Composable functions. The annotation tells the Kotlin compiler that this function is used by Jetpack Compose to generate the UI.
- 1. You add the `@Composable` annotation before the function.
- 1. `@Composable` function names are capitalized.
- 1. `@Composable` functions can't return anything.

To set a different background color for your introduction, you'll need to surround your text with a `Surface`. A `Surface` is a container that represents a section of UI where you can alter the appearance, such as the background color or border.

A `Modifier` is used to augment or decorate a composable. One modifier you can use is the `padding` modifier, which adds space around the element (in this case, adding space around the text). This is accomplished by using the `Modifier.padding()` function.

Annotations are means of attaching extra information to code. This information helps tools like the Jetpack Compose compiler, and other developers understand the app's code.

An annotation is applied by prefixing its name (the annotation) with the `@` character at the beginning of the declaration you are annotating. Different code elements, including properties, functions, and classes, can be annotated

The scalable pixels (SP) is a unit of measure for the font size. UI elements in Android apps use two different units of measurement: density-independent pixels (DP), which you use later for the layout, and scalable pixels (SP). By default, the SP unit is the same size as the DP unit, but it resizes based on the user's preferred text size under phone settings.

```
// Don't copy.  
Row {  
    Text("First Column")  
    Text("Second Column")  
}
```

Notice in the previous code snippet that curly braces are used instead of parentheses in the `Row` composable function. This is called *Trailing Lambda Syntax*.

When you resize images that are larger than the Android system can handle, an out-of-memory error is thrown. For photographs and background images, such as the current image, the `androidparty.png`, you should place them in the `drawable-nodpi` folder, which stops the resizing behavior.

If the image is imported successfully, Android Studio adds the image to the list under the **Drawable** tab. This list includes all your images and icons for the app. You can now use this image in your app.

1. Switch back to the project view, click

View > Tool Windows > Project or click the **Project** tab on the far left.

An

`R` class is an automatically generated class by Android that contains the IDs of all resources in the project. In most cases, the resource ID is the same as the filename. For example, the image in the previous file hierarchy can be accessed with this code:

```
R.drawable.graphic
```

You use the

`ContentScale.Crop` parameter scaling, which scales the image uniformly to maintain the aspect ratio so that the width and height of the image are equal to, or larger than, the corresponding dimension of the screen.

```
// This is an example.  
Modifier.padding(  
    start = 16.dp,  
    top = 16.dp,  
    end = 16.dp,  
    bottom = 16.dp  
)
```

As the name suggests, if you use the `!!` not-null assertion, it means that you assert that the value of the variable isn't `null`, regardless of whether it is or isn't.

Unlike `?.` safe-call operators, the use of a `!!` not-null assertion operator may result in a `NullPointerException` error being thrown if the nullable variable is indeed `null`. Thus, it should be done only when the variable is always non-nullable or proper exception handling is set in place. When not handled, exceptions cause runtime errors. You learn about exception handling in later units of this course.

Composables are stateless by default, which means that they don't hold a value and can be recomposed any time by the system, which results in the value being reset. However, Compose provides a convenient way to avoid this. Composable functions can store an object in memory using the `remember` composable.

1. Make the `result` variable a `remember` composable.

The `remember` composable requires a function to be passed.

1. In the `remember` composable body, pass in a `mutableStateOf()` function and then pass the function a `1` argument.

The `mutableStateOf()` function returns an observable. You learn more about observables later, but for now this basically means that when the value of the `result` variable changes, a recomposition is triggered, the value of the result is reflected, and the UI refreshes.

`MainActivity.kt`

```
var result by remember { mutableStateOf(1) }
```

Styles and themes are a collection of attributes that specifies the appearance for a single UI element. A style can specify attributes such as font color, font size, background color, and much more which can be applied for the entire app. Later codelabs will cover how to implement these in your app. For now, this has already been done for you to make your app more beautiful.

You use the `State` and `MutableState` types in Compose to make state in your app observable, or tracked, by Compose. The `State` type is immutable, so you can only read the value in it, while the `MutableState` type is mutable. You can use the `mutableStateOf()` function to create an observable `MutableState`. It receives an initial value as a parameter that is wrapped in a `State` object, which then makes its `value` observable.

The value returned by the `mutableStateOf()` function:

- Holds state, which is the bill amount.

- Is mutable, so the value can be changed.
- Is observable, so Compose observes any changes to the value and triggers a recomposition to update the UI.

you should hoist the state when you need to:

- Share the state with multiple composable functions.
- Create a stateless composable that can be reused in your app.

When you extract state from a composable function, the resulting composable function is called *stateless*. That is, composable functions can be made *stateless* by extracting state from them.

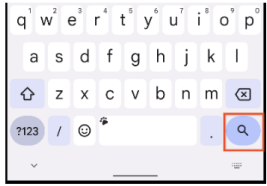
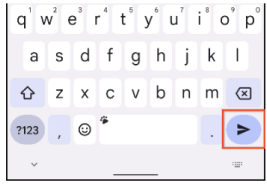
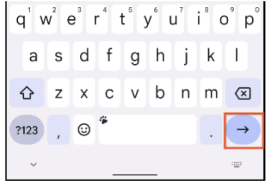
A *stateless* composable is a composable that doesn't have a state, meaning it doesn't hold, define, or modify a new state. On the other hand, a *stateful* composable is a composable that owns a piece of state that can change over time.

1. To denote that the

`label` parameter is expected to be a string resource reference, annotate the function parameter with the `@StringRes` annotation:

keyboard action button is a button at the end of the keyboard.

button is a button at the end of the keyboard. You can see some examples in this table:

Property	Action button on the keyboard
<code>ImeAction.Search</code> Used when the user wants to execute a search.	 A screenshot of an Android keyboard. The bottom row contains a shift key, a numeric keypad key, a spacebar, and a search button (magnifying glass icon) which is highlighted with a red box.
<code>ImeAction.Send</code> Used when the user wants to send the text in the input field.	 A screenshot of an Android keyboard. The bottom row contains a shift key, a numeric keypad key, a spacebar, and a send button (blue arrow icon) which is highlighted with a red box.
<code>ImeAction.Go</code> Used when the user wants to navigate to the target of the text in the input.	 A screenshot of an Android keyboard. The bottom row contains a shift key, a numeric keypad key, a spacebar, and a go button (blue arrow icon) which is highlighted with a red box.

When you want a property to have differing data types, subclassing is not the answer. Instead, Kotlin provides something called *generic types* that allow you to have a single property that can have differing data types, depending on the specific use case.

A generic data type is provided when instantiating a class, so it needs to be defined as part of the class signature. After the class name comes a left-facing angle bracket (`<`), followed by a placeholder name for the data type, followed by a right-facing angle bracket (`>`).

The placeholder name can then be used wherever you use a real data type within the class, such as for a property.

When a class is defined as a data class, the following methods are implemented.

- `equals()`
- `hashCode()`: you'll see this method when working with certain collection types.
- `toString()`

- `componentN(): component1(), component2(), etc.`
- `copy()`

Note: A data class needs to have at least one parameter in its constructor, and all constructor parameters must be marked with `val` or `var`. A data class also cannot be `abstract`, `open`, `sealed`, or `inner`.

- `List` is an interface that defines properties and methods related to a read-only ordered collection of items.
- `MutableList` extends the `List` interface by defining methods to modify a list, such as adding and removing elements.

Data classes are a type of class that only contain properties, they can provide some utility methods to work with those properties.

1. In Jetpack Compose, a scrollable list can be made using the

`LazyColumn` composable. The difference between a `LazyColumn` and a `Column` is that a `Column` should be used when you have a small number of items to display, as Compose loads them all at once. A `Column` can only hold a predefined, or fixed, number of composables. A `LazyColumn` can add content on demand, which makes it good for long lists and particularly when the length of the list is unknown.

A `LazyColumn` also provides scrolling by default, without additional code. Declare a `LazyColumn` composable inside of the `AffirmationList()` function. Pass the `modifier` object as an argument to the `LazyColumn`.

<https://material-foundation.github.io/material-theme-builder/>

`Scaffold` supports the `contentWindowInsets` parameter which can help to specify insets for the scaffold content. `WindowInsets` are the parts of your screen where your app can intersect with the system UI, these ones are to be passed to the content slot via the `PaddingValues` parameters.

Note: `Modifier.weight()` sets the UI element's width/height proportionally to the element's weight, relative to its weighted siblings (other child elements in the row or column).

Example: Consider three child elements in a row with weights `1f`, `1f`, and `2f`. All child elements have assigned weights in this case. The available space for the row is divided proportionally to the specified weight value, with more available space going to children with higher weight values. The child elements will distribute the weight as shown below:



In the above row, the first child composable has $\frac{1}{4}$ of the row's width, the second also has $\frac{1}{4}$ of the row's width, and the third has $\frac{1}{2}$ of the row's width.

If the children don't have assigned weights, (weight is an optional parameter), then the child composable's height/width would default to wrap content (wrapping the contents of what's inside the UI element).

each list item row contains a dog image, dog information, and an expand more button. You will add a `Spacer` composable before the expand more button with weight `1f` to properly align the button icon. Since the spacer is the only weighted child element in the row, it will fill the space remaining in the row after measuring the other unweighted child elements' width.

During its lifetime, an activity transitions through, and sometimes back to, various states. This transitioning of states is known as the activity lifecycle.

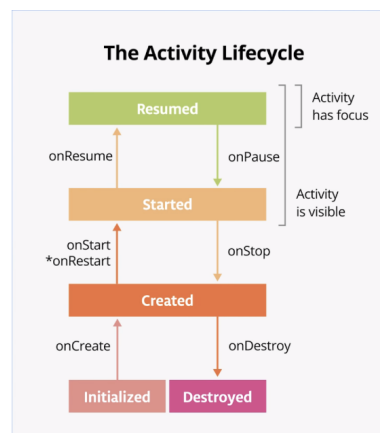
In Android, an activity is the entry point for interacting with the user.

In the past, one activity would display one screen in an app. With current best practices, one activity might display multiple screens by swapping them in and out as needed.

The activity lifecycle extends from the creation of the activity to its destruction, when the system reclaims that activity's resources. As a user navigates in and out of an activity, each activity transitions between different states in the activity lifecycle.

Similarly, the activity lifecycle consists of the different states that an activity can go through, from when the activity first initializes to its destruction, at which time the operating system (OS) reclaims its memory. Typically, the entry point of a program is the `main()` method. Android activities, however, begin with the `onCreate()` method; this method would be the equivalent of the egg stage in the above example. You have used activities already, many times throughout this course, and you might recognize the `onCreate()` method. As the user starts your app, navigates between activities, navigates inside and outside of your app, the activity changes state.

Often, you want to change some behavior, or run some code, when the activity lifecycle state changes. Therefore, the `Activity` class itself, and any subclasses of `Activity` such as `ComponentActivity`, implement a set of lifecycle callback methods. Android invokes these callbacks when the activity moves from one state to another, and you can override those methods in your own activities to perform tasks in response to those lifecycle state changes. The following diagram shows the lifecycle states along with the available overridable callbacks.



Note: The asterisk on the `onRestart()` method indicates that this method is not called every time the state transitions between **Created** and **Started**. It is only called if `onStop()` was called and the activity is subsequently restarted.

It's important to know when Android invokes the overridable callbacks and what to do in each callback method, but both of these diagrams are complex and can be confusing. In this codelab, instead of just reading what each state and callback means, you're going to do some detective work and build your understanding of the Android activity lifecycle.

The `onCreate()` lifecycle method is called once, just after the activity *initializes*—when the OS creates the new `Activity` object in memory.

After `onCreate()` executes, the activity is considered *created*.

Note: When you override the `onCreate()` method, you must call the superclass implementation to complete the creation of the Activity, so within it, you must immediately call `super.onCreate()`. The same is true for other lifecycle callback met

The `onStart()` lifecycle method is called just after `onCreate()`. After `onStart()` runs, your activity is visible on the screen. Unlike `onCreate()`, which is called only once to initialize your activity, `onStart()` can be called by the system many times in the lifecycle of your activity.

Note that `onStart()` is paired with a corresponding `onStop()` lifecycle method. If the user starts your app and then returns to the device's home screen, the activity is stopped and is no longer visible on screen.

1. Type `tag:MainActivity` into the search field to filter the log. Notice that both the `onCreate()` and `onStart()` methods were called one after the other, and that your activity is visible on screen.
2. Press the **Home** button on the device and then use the Recents screen to return to the activity. Notice that the activity resumes where it left off, with all the same values, and that `onStart()` is logged a second time to Logcat. Notice also that the `onCreate()` method is not called again.

When an activity starts from the beginning, you see all three of these lifecycle callbacks called in order:

- `onCreate()` when the system creates the app.
- `onStart()` makes the app visible on the screen, but the user is not yet able to interact with it.
- `onResume()` brings the app to the foreground, and the user is now able to interact with it.

Despite the name, the `onResume()` method is called at startup, even if there is nothing to resume.

The Android OS might close your activity if your code manually calls the activity's `finish()` method or if the user force-quits the app.

Your activity does not close down entirely every time the user navigates away from that activity:

- When your activity is no longer visible on screen, the status is known as putting the activity into the *background*. The opposite of this is when the activity is in the *foreground*, or onscreen.
- When the user returns to your app, that same activity is restarted and becomes visible again. This part of the lifecycle is called the app's *visible* lifecycle.

When `onPause()` is called, the app no longer has focus. After `onStop()`, the app is no longer visible on screen. Although the activity is stopped, the `Activity` object is still in memory in the background. The Android OS has not destroyed the activity. The user might return to the app, so Android keeps your activity resources around.

A *configuration change* occurs when the state of the device changes so radically that the easiest way for the system to resolve the change is to completely shut down and rebuild the activity. For example, if the user changes the device language, the whole layout might need to change to accommodate different text directions and string lengths. If the user plugs the device into a dock or adds a physical keyboard, the app layout may need to take advantage of a different display size or layout. And if the device orientation changes—if the device is rotated from portrait to landscape or back the other way—the layout might need to change to fit the new orientation. Let's look at how the app behaves in this scenario.

The last lifecycle callback to demonstrate is `onDestroy()`, which is called after `onStop()`. It is called just before the activity is destroyed. This can happen when the app's code calls `finish()`, or the system needs to destroy and recreate the activity because of a configuration change.

You use the `rememberSaveable` function to save values that you need if Android OS destroys and recreates the activity.

To save values during recompositions, you need to use `remember`.

Use `rememberSaveable` to save values during recompositions AND configuration changes.

The UI layer is made up of the following components:

- **UI elements:** components that render the data on the screen. You build these elements using Jetpack Compose.

- **State holders:** components that hold the data, expose it to the UI, and handle the app logic. An example state holder is ViewModel.

`ViewModel` stores the app-related data that isn't destroyed when the activity is destroyed and recreated by the Android framework. Unlike the activity instance, `ViewModel` objects are not destroyed. The app automatically retains `ViewModel` objects during configuration changes so that the data they hold is immediately available after the recomposition.

To implement `ViewModel` in your app, extend the `ViewModel` class, which comes from the architecture components library and stores app data within that class.

`StateFlow` is a data holder observable flow that emits the current and new state updates. Its `value` property reflects the current state value. To update state and send it to the flow, assign a new value to the `value` property of the `MutableStateFlow` class.

In Android, `StateFlow` works well with classes that must maintain an observable immutable state.

A `StateFlow` can be exposed from the `GameUiState` so that the composables can listen for UI state updates and make the screen state survive configuration changes.

In the `GameViewModel` class, add the following `_uiState` property.

```
import kotlinx.coroutines.flow.MutableStateFlow

// Game UI state
private val _uiState = MutableStateFlow(GameUiState())
```

To implement a backing property, you override the getter method to return a read-only version of your data. A backing property lets you return something from a getter other than the exact object.

In Compose, the only way to update the UI is by changing the state of the app. What you can control is your UI state. Every time the state of the UI changes, Compose recreates the parts of the UI tree that changed. Composables can accept state and expose events. For example,

a `TextField` / `OutlinedTextField` accepts a value and exposes a callback `onValueChange` that requests the callback handler to change the value.

A *unidirectional data flow* (UDF) is a design pattern in which state flows down and events flow up. By following unidirectional data flow, you can decouple composables that display state in the UI from the parts of your app that store and change state.

The UI update loop for an app using unidirectional data flow looks like the following:

- **Event:** Part of the UI generates an event and passes it upward—such as a button click passed to the ViewModel to handle—or an event that is passed from other layers of your app, such as an indication that the user session has expired.
- **Update state:** An event handler might change the state.
- **Display state:** The state holder passes down the state, and the UI displays it.

Note on `copy()` method: Use the `copy()` function to copy an object, allowing you to alter some of its properties while keeping the rest unchanged.

Example:

```
val jack = User(name = "Jack", age = 1)

val olderJack = jack.copy(age = 2)
```

The Navigation component has three main parts:

- **NavController:** Responsible for navigating between destinations—that is, the screens in your app.
- **NavGraph:** Maps composable destinations to navigate to.
- **NavHost:** Composable acting as a container for displaying the current destination of the NavGraph.
- **`navController`:** An instance of the `NavHostController` class. You can use this object to navigate between screens, for example, by calling the `navigate()` method to navigate to another destination. You can obtain the `NavHostController` by calling `rememberNavController()` from a composable function.

- `startDestination` : A string route defining the destination shown by default when the app first displays the `NavHost` . In the case of the Cupcake app, this should be the `Start` route.

1. Within the trailing lambda, get a reference to

`LocalContext.current` and store it in a variable named `context` . `Context` is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, etc. You can use this variable to get the strings from the list of resource IDs in the view model to display the list of flavors.

6. Make your layout adapt to different screen sizes

What are breakpoints?

You may wonder how you can show different layouts for the same app. The short answer is by using conditionals on different states, the way you did in the beginning of this codelab.

To create an adaptive app, you need the layout to change based on screen size. The measurement point where a layout changes is known as a breakpoint. Material Design created an [opinionated breakpoint range](#) that covers most Android screens.

Breakpoint range (dp)	Portrait	Landscape	Window size class	Columns	Minimum margins*
0-599	Handset	Phone**	Compact	4	8
600-839	Foldable Small tablet	Foldable Small tablet	Medium	12	12
840+	Large tablet	Large tablet	Expanded	12	32

*Margins and gutters are flexible and don't need to be equal in size.

**Phones in landscape are considered an exception in order to still fit within this category.

This breakpoint range table shows, for example, that if your app is currently running on a device with a screen size less than 600 dp, you should show the mobile layout.

Note: The breakpoints concept in adaptive layouts is different from the [breakpoints term in debugging](#).

The `WindowSizeClass` API introduced for Compose makes the implementation of Material Design breakpoints simpler.

Window Size Classes introduces three categories of sizes: Compact, Medium, and Expanded, for both width and height.

Coroutines allow the execution of a block of code to be suspended and then resumed later, so that other work can be done in the meantime. Coroutines make it easier to write **asynchronous** code, which means one task doesn't need to finish completely before starting the next task, enabling multiple tasks to run concurrently.

A **suspending** function is like a regular function, but it can be suspended and resumed again later. To do this, suspend functions can only be called from other suspend functions that make this capability available.

A suspending function may contain zero or more suspension points.

A **suspension point** is the place within the function where execution of the function can suspend. Once execution resumes, it picks up where it last left off in the code and proceeds with the rest of the function.

```
import kotlinx.coroutines.*

fun main() {
    runBlocking {
        println("Weather forecast")
        launch {
            printForecast()
        }
        launch {
            printTemperature()
        }
    }
}

suspend fun printForecast() {
    delay(1000)
    println("Sunny")
}
```

```

}

suspend fun printTemperature() {
    delay(1000)
    println("30\u00b0C")
}

```

The output is the same but you may have noticed that it is faster to run the program. Previously, you had to wait for the `printForecast()` suspend function to finish completely before moving onto the `printTemperature()` function.

Now `printForecast()` and `printTemperature()` can run concurrently because they are in separate coroutines.

`coroutineScope()` will only return once all its work, including any coroutines it launched, have completed.

The coroutine executing `getTemperature()` and the coroutine executing `getForecast()` are child coroutines of the same parent coroutine. The behavior you're seeing with exceptions in coroutines is due to structured concurrency. When one of the child coroutines fails with an exception, it gets propagated upwards. The parent coroutine is cancelled, which in turn cancels any other child coroutines (e.g. the coroutine running `getForecast()` in this case). Lastly, the error gets propagated upwards and the program crashes with the `AssertionError`.

Coroutines use dispatchers to determine the thread to use for its execution. A **thread** can be started, does some work (executes some code), and then terminates when there's no more work to be done.

When a user starts your app, the Android system creates a new process and a single thread of execution for your app, which is known as the **main thread**. The main thread handles many important operations for your app including Android system events, drawing the UI on the screen, handling user input events, and more. As a result, most of the code you write for your app will likely run on the main thread.

In the case of Android apps, you should only call blocking code on the main thread if it will execute fairly quickly. The goal is to keep the main thread unblocked, so that it can execute work immediately if a new event is triggered.

This main thread is the **UI thread** for your activities and is responsible for UI drawing and UI related events. When there's a change on the screen, the UI needs to be redrawn. For something like an animation on the screen, the UI needs to be redrawn frequently so that it appears like a smooth transition. If the main thread needs to execute a long-running block of work, then the screen won't update as frequently and the user will see an abrupt transition (known as "jank") or the app may hang or be slow to respond.

Hence we need to move any long-running work items off the main thread and handle it in a different thread. Your app starts off with a single main thread, but you can choose to create multiple threads to perform additional work. These additional threads can be referred to as worker threads. It's perfectly fine for a long-running task to block a worker thread for a long time, because in the meantime, the main thread is unblocked and can actively respond to the user.

There are some built-in dispatchers that Kotlin provides:

- **Dispatchers.Main:** Use this dispatcher to run a coroutine on the main Android thread. This dispatcher is used primarily for handling UI updates and interactions, and performing quick work.
- **Dispatchers.IO:** This dispatcher is optimized to perform disk or network I/O outside of the main thread. For example, read from or write to files, and execute any network operations.
- **Dispatchers.Default:** This is a default dispatcher used when calling `launch()` and `async()`, when no dispatcher is specified in their context. You can use this dispatcher to perform computationally-intensive work outside of the main thread. For example, processing a bitmap image file.

`LaunchedEffect()` composable runs the provided suspending function for as long as it remains in the composition. You can use the `LaunchedEffect()` composable function to accomplish all of the following:

- The `LaunchedEffect()` composable allows you to safely call suspend functions from composables.
- When the `LaunchedEffect()` function enters the Composition, it launches a coroutine with the code block passed as a parameter. It runs the provided suspend function as long as it remains in the composition. When a user clicks

the **Start** button in the RaceTracker app, the `LaunchedEffect()` enters the composition and launches a coroutine to update progress.

- The coroutine is canceled when the `LaunchedEffect()` exits the composition. In the app, if the user clicks the **Reset/Pause** button, `LaunchedEffect()` is removed from the composition and the underlying coroutines are canceled.

```
LaunchedEffect(playerOne, playerTwo) {  
    launch {playerOne.run() }  
    launch {playerTwo.run() }  
    raceInProgress = false // This will update the state immediately, without waiting for players to finish run() execution.  
}
```

The `raceInProgress` flag is updated immediately because:

- The `launch` builder function launches a coroutine to execute `playerOne.run()` and immediately returns to execute the next line in the code block.
- The same execution flow happens with the second `launch` builder function that executes `playerTwo.run()` function.
- As soon as the second `launch` builder returns, the `raceInProgress` flag is updated. This immediately changes the button text to **Start** and the race does not begin.

```
LaunchedEffect(playerOne, playerTwo) {  
    coroutineScope {  
        launch { playerOne.run() }  
        launch { playerTwo.run() }  
    }  
    raceInProgress = false  
}
```

- When the `LaunchedEffect()` block executes, the control is transferred to the `coroutineScope{..}` block.

- The `coroutineScope` block launches both coroutines concurrently and waits for them to finish execution.
- Once the execution is complete, the `raceInProgress` flag updates.

In Kotlin, object declarations are used to declare singleton objects. Singleton pattern ensures that one, and only one, instance of an object is created and has one global point of access to that object. Object initialization is thread-safe and done at first access.

Remember "lazy initialization" is when object creation is purposely delayed, until you actually need that object, to avoid unnecessary computation or use of other computing resources. Kotlin has first-class support for lazy instantiation.

A `viewModelScope` is the built-in coroutine scope defined for each `ViewModel` in your app. Any coroutine launched in this scope is automatically canceled if the `ViewModel` is cleared.

You can use `viewModelScope` to launch the coroutine and make the web service request in the background. Since the `viewModelScope` belongs to the `ViewModel`, the request continues even if the app goes through a configuration change.

In general a repository class:

- Exposes data to the rest of the app.
- Centralizes changes to data.
- Resolves conflicts between multiple data sources.
- Abstracts sources of data from the rest of the app.
- Contains business logic.

Many times, classes require objects of other classes to function. When a class requires another class, the required class is called a **dependency**.