UNIT 3 : MODULE 1

1. Introduction

So far, the apps you've worked on have had only one activity. In reality, many Android apps require multiple activities, with navigation between them.

In this codelab, you'll build out a dictionary app so that it uses multiple activities, uses intents to navigate between them, and passes data to other apps.

What you'll learn

How to:

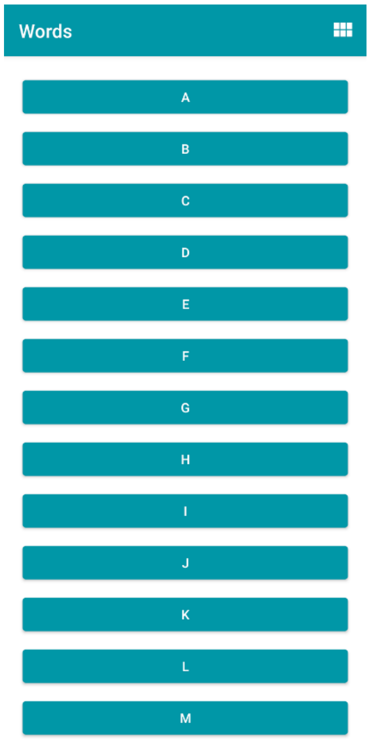
- Use an explicit intent to navigate to a specific activity.
- Use an implicit intent to navigate to content in another app.
- Add menu options to add buttons to the app bar.

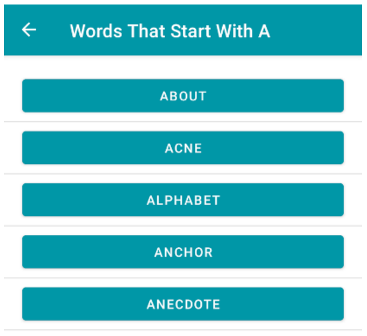
What you'll build

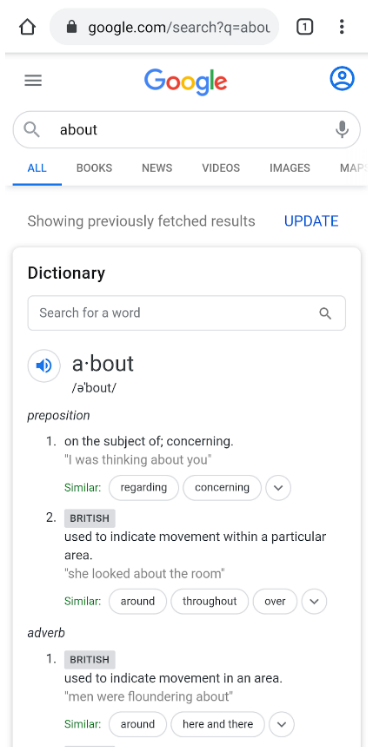
- Modify a dictionary app to implement navigation betwee n screens using intents and adding an options menu.

2. Starter code

On the next few steps, you'll be working on the Words app. The Words app is a simple dictionary app, with a list of letters, words for each letter, and the ability to look up definitions of each word in the browser.







Showing previously fetched results

Dictionary

a·bout
/əˈbout/

preposition

1. on the subject of; concerning.
   "I was thinking about you"
   Similar: regarding concerning
2. BRITISH
   used to indicate movement within a particular area.
   "she looked about the room"
   Similar: around throughout over

adverb

1. BRITISH
   used to indicate movement in an area.
   "men were floundering about"
   Similar: around here and there

There's a lot going on, but don't worry—you won't have to build an entire app just to learn about intents. Instead, you've been provided an incomplete version of the project, or starter project.

While all the screens are implemented, you can't yet navigate from one screen to the other. Your task is to use intents so that the entire project is working, without having to build everything from scratch.

Download the starter code for this codelab

This codelab provides starter code for you to extend with features taught in this codelab. Starter code may contain code that is familiar to you from previous codelabs. It may also contain code that is unfamiliar to you, and that you will learn about in later codelabs.

When you download the starter code from GitHub, note that the folder name is `android-basics-kotlin-words-app-starter`. Select this folder when you open the project in Android Studio.

Starter Code URL: https://github.com/google-developer-training/android-basics-kotlin-words-app/tree/starter

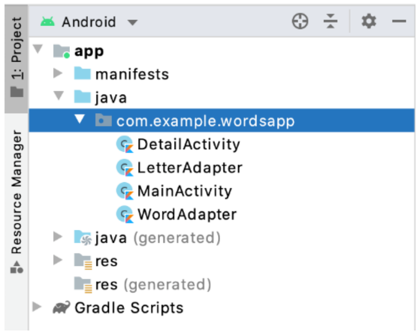
Branch name in GitHub:

starter

If you are familiar with git commands, note that the starter code is in a branch called "starter". After you clone the repo, check out the code from the `origin/starter` branch. If you haven't used git commands before, follow the below steps to download the code from GitHub.

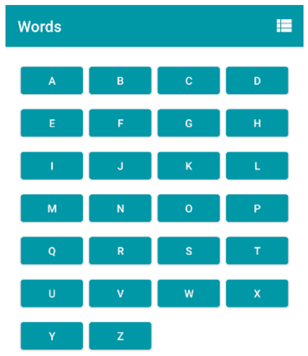
3. Words app overview

Before moving forward, take a moment to familiarize yourself with the project. All the concepts should be familiar from the previous unit. Right now, the app consists of two activities, each with a recycler view and an adapter.



You'll specifically be working with the following files:

1. `LetterAdapter` is used by the `RecyclerView` in `MainActivity`. Each letter is a button with an `onClickListener`, which is currently empty. This is where you'll handle button presses to navigate to the `DetailActivity`.
2. `WordAdapter` is used by the `RecyclerView` in `DetailActivity` to display a list of words. While you can't navigate to this screen quite yet, just know that each word also has a corresponding button with an `onClickListener`. This is where you'll add code that navigates to the browser, to show a definition for the word.
3. `MainActivity` will also need a few changes. This is where you'll implement the options menu to display a button that allows users to toggle between list and grid layouts.



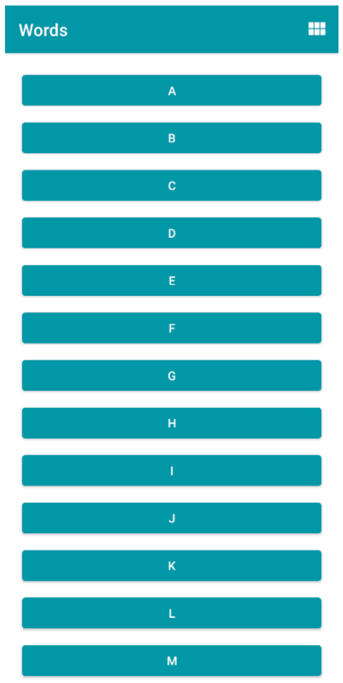
Once you feel comfortable with the project so far, continue to the next section where you'll learn about intents.

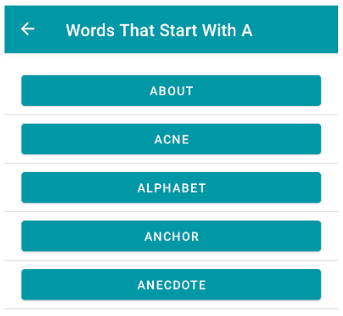
4. Introduction to Intents

Now that you've set up the initial project, let's discuss intents and how you can use them in your app.

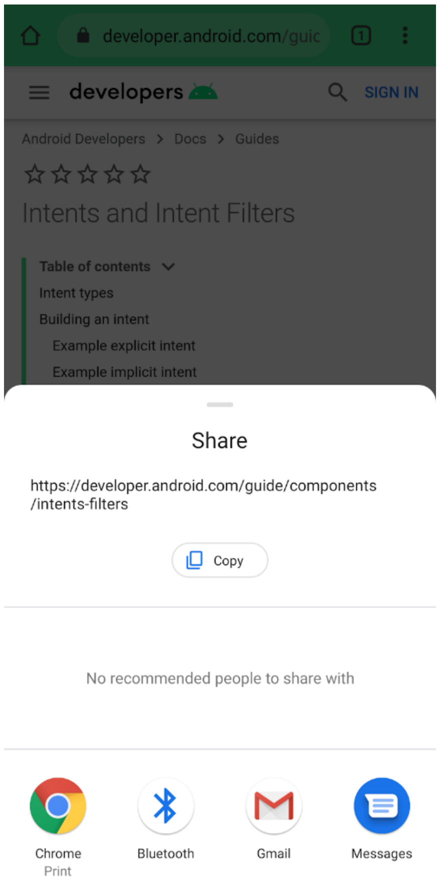
An *intent* is an object representing some action to be performed. The most common, but certainly not only, use for an intent is to launch an activity. There are two types of intents—**implicit** and **explicit**. An **explicit intent** is highly specific, where you know the exact activity that to be launched, often a screen in your own app.

An **implicit intent** is a bit more abstract, where you tell the system the type of action, such as opening a link, composing an email, or making a phone call, and the system is responsible for figuring out how to fulfill the request. You've probably seen both kinds of intents in action without knowing it. Generally, when showing an activity in your own app, you use an explicit intent.

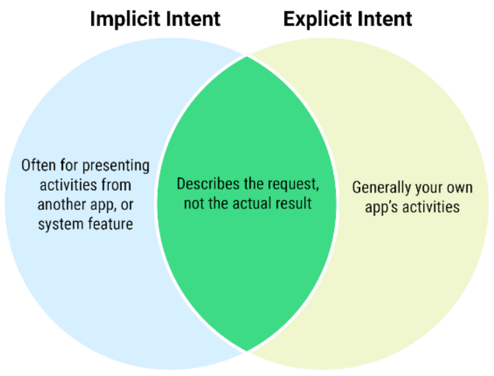




However, for actions that don't necessarily involve the current app—say, you found an interesting Android documentation page and want to share it with friends—you'd use an **implicit intent**. You might see a menu like this asking which app to use to share the page.


developer.android.com/guic
developers
SIGN IN
Android Developers > Docs > Guides
Intents and Intent Filters
Table of contents
Intent types
Building an intent
Example explicit intent
Example implicit intent
Share
https://developer.android.com/guide/components/intents-filters
Copy
No recommended people to share with
Chrome
Print
Bluetooth
Gmail
Messages

You use an explicit intent for actions or presenting screens in your own app and are responsible for the entire process. You commonly use implicit intents for performing actions involving other apps and rely on the system to determine the end result. You'll use both types of intents in the Words app.


Implicit Intent
Explicit Intent
Often for presenting activities from another app, or system feature
Describes the request, not the actual result
Generally your own app's activities

5. Set Up Explicit Intent

It's time to implement your first intent. On the first screen, when the user taps a letter, they should be taken to a second screen with a list of words. The `DetailActivity` is already implemented, so all that's needed is to launch it with an intent. Because your app knows exactly which activity should be launched, you use an explicit intent.

Creating and using an intent takes just a few steps.

1. Open `LetterAdapter.kt` and scroll down to `onBindViewHolder()`. Below the line to set the button text, set the `onClickListener` for `holder.button`.

```
holder.button.setOnClickListener {
}
```

2. Then get a reference to the `context`.

```
val context = holder.view.context
```

3. Create an `Intent`, passing in the context, as well as the class name of the destination activity.

```
val intent = Intent(context, DetailActivity::class.java)
```

   The name of the activity you want to show is specified with `DetailActivity::class.java`. An actual `DetailActivity` object is created behind the scenes.

4. Call the `putExtra` method, passing in "letter" as the first argument and the button text as the second argument.

```
intent.putExtra("letter", holder.button.text.toString())
```

   What's an extra? Remember that an intent is simply a set of instructions—there's no instance of the destination activity just yet. Instead, an extra is a piece of data, such as a number or string, that is given a name to be retrieved later. This is similar to passing an argument when you call a function. Because a `DetailActivity` can be shown for any letter, you need to tell it which letter to present.

   Also, why do you think it's necessary to call `toString()`? The button's text is already a string, right?

   Sort of. It's actually of type `CharSequence`, which is something called an *interface*. You don't need to know anything about Kotlin interfaces for now, other than it's a way to ensure a type, such as String, implements specific functions and properties. You can think of a `CharSequence` as a more generic representation of a string-like class. A button's `text` property could be a string, or it could be any object that is also a `CharSequence`. The `putExtra()` method, however, accepts a `String`, not just any `CharSequence`, hence the need to call `toString()`.

5. Call the `startActivity()` method on the context object, passing in the `intent`.

```
context.startActivity(intent)
```

   Now run the app and try tapping a letter. The detail screen is displayed! But no matter which letter the user taps, the detail screen always shows words for the letter A. You still have some work to do in the detail activity so that it shows words for whichever letter is passed as the `intent` extra.

6. Set Up DetailActivity

You've just created your first explicit intent! Now onto the detail screen.

In the onCreate method of DetailActivity, after the call to setContentView, replace the hard coded letter with code to get the letterId passed in from the intent.

```
val letterId = intent?.extras?.getString("letter").toString()
```

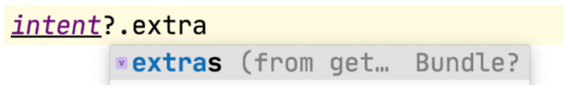
There's a lot going on here, so let's take a look at each part:

First, where does the intent property come from? It's not a property of DetailActivity, but rather, a property of any activity. It keeps a reference to the intent used to launch the activity.

The extras property is of type Bundle, and as you might have guessed, provides a way to access all extras passed into the intent.

Both of these properties are marked with a question mark. Why is this? The reason is that the intent and extras properties are nullable, meaning they may or may not have a value. Sometimes you may want a variable to be null. The intent property might not actually be an Intent (if the activity wasn't launched from an intent) and the extras property might not actually be a Bundle, but rather a value called null. In Kotlin, null means the absence of a value. The object may exist or it may be null. If your app tries to access a property or call a function on a null object, the app will crash.To safely access this value, you put a ? after the name. If intent is null, your app won't even attempt to access the extras property, and if extras is null, your code won't even attempt to call getString().

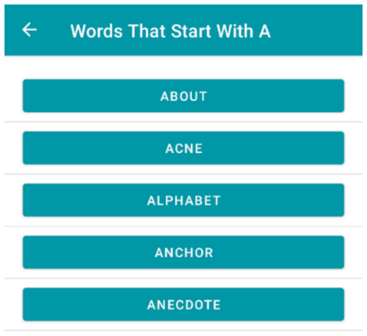
How do you know which properties require a question mark to ensure null safety? You can tell if the type name is followed by either a question mark or exclamation point.



```
intent?.extra
  extras (from get…  Bundle?
```

The final thing to note is the actual letter is retrieved with getString, which returns a String?, so toString() is called to ensure it's a String, and not null.

Now when you run the app and navigate to the detail screen, you should see the list of words for each letter.



Cleaning Up

Both the code to perform the intent, and retrieve the selected letter hardcode the name of the `extra`, "letter". While this works for this small example, it's not the best approach for large apps where you have many more intent extras to keep track of.

While you could just create a constant called "letter", this could get unwieldy as you add more intent extras to your app. And in which class would you put this constant? Remember that the string is used in both `DetailActivity` and `MainActivity`. You need a way to define a constant that can be used across multiple classes, while keeping your code organized.

Thankfully, there's a handy Kotlin feature that can be used to separate your constants and make them usable without a particular instance of the class called *companion objects*. A companion object is similar to other objects, such as instances of a class. However, only a single instance of a companion object will exist for the duration of your program, which is why this is sometimes called the *singleton pattern*. While there are numerous use cases for *singletons* beyond the scope of this codelab, for now, you'll use a companion object as a way to organize constants and make them accessible outside of the `DetailActivity`. You'll start by using a companion object to refactor the code for the "letter" extra.

1. In `DetailActivity`, just above `onCreate`, add the following.

```
companion object {
}
```

Notice how this is similar to defining a class, except you use the `object` keyword. There's also a keyword `companion`, meaning it's associated with the `DetailActivity` class, and we don't need to give it a separate type name.

2. Within the curly braces, add a property for the letter constant.

```
const val LETTER = "letter"
```

3. To use the new constant, update your hard coded letter call in `onCreate()` as follows.

```
val letterId = intent?.extras?.getString(LETTER).toString()
```

   Again, notice that you reference it with dot notation as usual, but the constant belongs to `DetailActivity`.

4. Switch over to `LetterAdapter`, and modify the call to `putExtra` to use the new constant.

```
intent.putExtra(DetailActivity.LETTER, holder.button.text.toString())
```

   All set! By refactoring, you just made your code easier to read, and easier to maintain. If this, or any other constant you add, ever needs to change, you only need to do so in one place.

   To learn more about companion objects, check out the Kotlin documentation on Object Expressions and Declarations.

7. Set Up Implicit Intent

In most cases, you'll present specific activities from your own app. However, there are some situations where you may not know which activity, or which app, you want to launch. On our detail screen, each word is a button that will show the user definition of the word.

For our example, you're going to use the dictionary functionality provided by a Google search. Instead of adding a new activity to your app, however, you're going to launch the device's browser to show the search page.

So perhaps you'd need an intent to load the page in Chrome, the default browser on Android?

Not quite.

It's possible that some users prefer a third party browser. Or their phone comes with a browser preinstalled by the manufacturer. Perhaps they have the Google search app installed—or even a third-party dictionary app.

You can't know for sure what apps the user has installed. Nor can you assume how they may want to look up a word. This is a perfect example of when to use an implicit intent. Your app provides information to the system on what the action should be, and the system figures out what to do with that action, prompting the user for any additional information as needed.

Do the following to create the implicit intent.

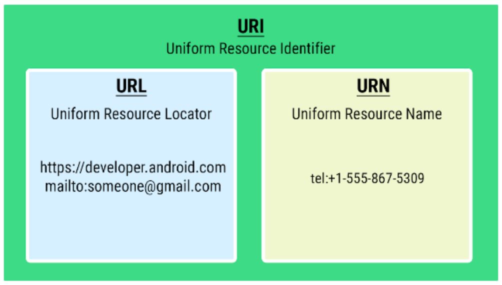
1. For this app, you'll perform a Google search for the word. The first search result will be a dictionary definition of the word. Since the same base URL is used for every search, it's a good idea to define this as its own constant. In `DetailActivity`, modify the companion object to add a new constant, `SEARCH_PREFIX`. This is the base URL for a Google search.

```
companion object {
   val LETTER = "letter"
   val SEARCH_PREFIX = "https://www.google.com/search?q="
}
```

2. Then, open up WordAdapter and in the onBindViewHolder() method, call setOnClickListener() on the button. Start by creating a URI for the search query. When calling parse() to create a URI from a String, you need to use string formatting so that the word is appended to the SEARCH_PREFIX.

```
holder.button.setOnClickListener {
    val queryUrl: Uri = Uri.parse("${DetailActivity.SEARCH_PREFIX}${item}")
}
```

If you're wondering what a *URI* is, it's not a typo, but stands for *Uniform Resource Identifier*. You may already know that a URL, or *Uniform Resource Locator*, is a string that points to a webpage. A URI is a more general term for the format. All URLs are URIs, but not all URIs are URLs. Other URIs, for example, an address for a phone number, would begin with tel:, but this is considered a URN or *Uniform Resource Name*, rather than a URL. The data type used to represent both is called URI.



Notice how there's no reference to any activity in your own app here. You simply provide a URI, without an indication of how it's ultimately used.

2. After defining queryUrl, initialize a new intent object

```
val intent = Intent(Intent.ACTION_VIEW, queryUrl)
```

Instead of passing in a context and an activity, you pass in Intent.ACTION_VIEW along with the URI.

ACTION_VIEW is a generic intent that takes a URI, in your case, a web address. The system then knows to process this intent by opening the URI in the user's web browser. Some other intent types include:

- CATEGORY_APP_MAPS – launching the maps app

- CATEGORY_APP_EMAIL – launching the email app
- CATEGORY_APP_GALLERY – launching the gallery (photos) app
- ACTION_SET_ALARM – setting an alarm in the background
- ACTION_DIAL – initiating a phone call

To learn more, visit the documentation for some [commonly used intents](#).

3. Finally, even though you aren't launching any particular activity in your app, you're telling the system to launch another app, by calling `startActivity()`and pass in the `intent`.

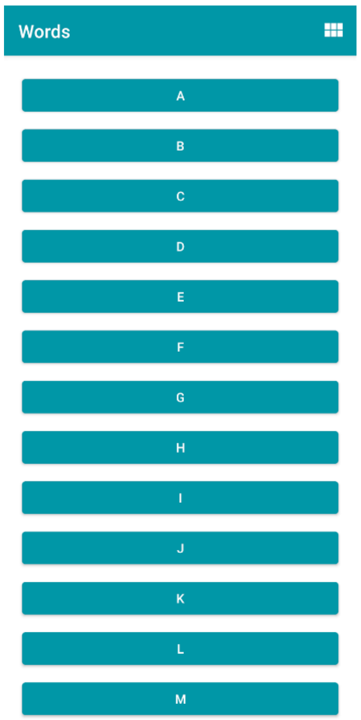
```
context.startActivity(intent)
```

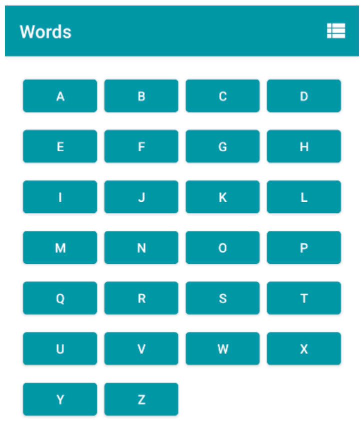
Now when you launch the app, navigate to the words list, and tap one of the words, your device should navigate to the URL (or present a list of options depending on your installed apps).

The exact behavior will differ among users, providing a seamless experience for everyone, without complicating your code.

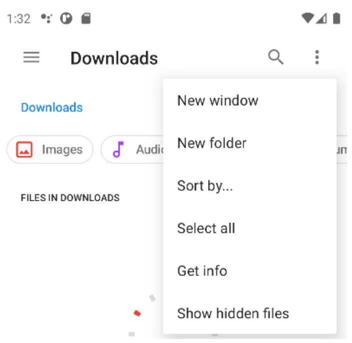
8. Set Up Menu and Icons

Now that you've made your app fully navigable by adding explicit and implicit intents, it's time to add a menu option so that the user can toggle between list and grid layouts for the letters.



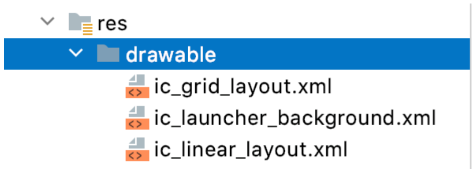


By now, you've probably noticed many apps have this bar at the top of the screen. This is called the app bar, and in addition to showing the app's name, the app bar can be customized and host lots of useful functionality, like shortcuts for useful actions, or overflow menus.

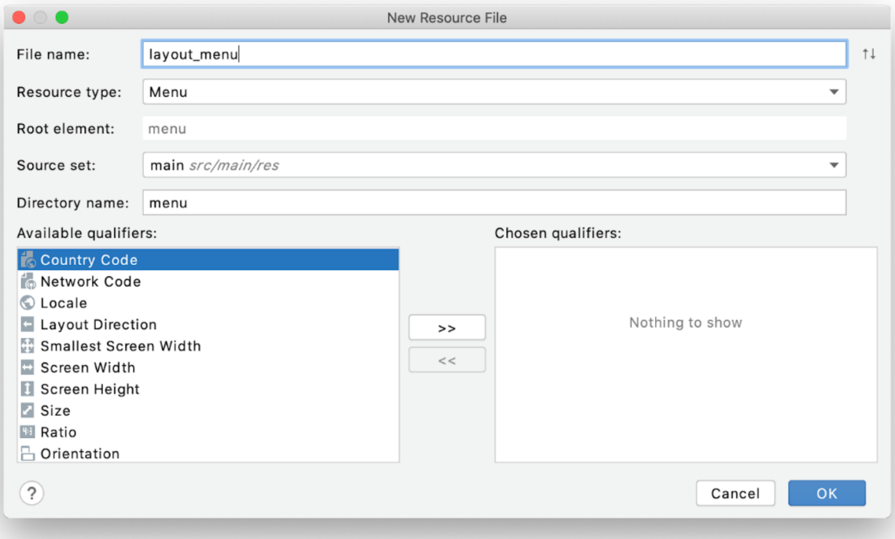


For this app, while we won't add a fully fledged menu, you'll learn how to add a custom button to the app bar, so that the user can change the layout.

1. First, you need to import two icons to represent the grid and list views. Add the clip art vector assets called "view module" (name it **ic_grid_layout**) and "view list" (name it **ic_linear_layout**). If you need a refresher on adding material icons, take a look at the instructions on this page.



2. You also need a way to tell the system what options are displayed in the app bar, and which icons to use. To do this, add a new resource file by right-clicking on the **res** folder and selecting **New > Android Resource File**. Set the **Resource Type** to `Menu` and the **File Name** to `layout_menu`.



New Resource File
File name: layout_menu
Resource type: Menu
Root element: menu
Source set: main *src/main/res*
Directory name: menu

3. Click **OK**.
4. Open **res/Menu/layout_menu**. Replace the contents of `layout_menu.xml` with the following:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <item android:id="@+id/action_switch_layout"
     android:title="@string/action_switch_layout"
```

```
    android:icon="@drawable/ic_linear_layout"
    app:showAsAction="always" />
</menu>
```

The structure of the menu file is fairly simple. Just like a layout starts with a layout manager to hold individual views, a menu xml file starts with a menu tag, which contains individual options.

Your menu just has one button, with a few properties:

- `id`: Just like views, the menu option has an id so that it can be referenced in code.
- `title`: This text won't actually be visible in your case, but may be useful for screen readers to identify the menu
- `icon`: The default is `ic_linear_layout`. However, this will be toggled on and off to show the grid icon, when the button is selected.
- `showAsAction`: This tells the system how to show the button. Because it's set to always, this button will always be visible in the app bar, and not become part of an overflow menu.

Of course, just having the properties set doesn't mean the menu will actually do anything.

You'll still need to add some code in `MainActivity.kt` to get the menu working.

9. Implement Menu button

To see your menu button in action, there are a few things to do in `MainActivity.kt`.

1. First, it's a good idea to create a property to keep track of which layout state the app is in. That will make it easier to toggle the layout button. Set the default value to `true`, as the linear layout manager will be used by default.

```
private var isLinearLayoutManager = true
```

2. When the user toggles the button, you want the list of items to turn into a grid of items. If you recall from learning about recycler views, there are many different layout managers, one of which, `GridLayoutManager` allows for multiple items on a single row.

```
private fun chooseLayout() {
    if (isLinearLayoutManager) {
        recyclerView.layoutManager = LinearLayoutManager(this)
    } else {
        recyclerView.layoutManager = GridLayoutManager(this, 4)
    }
    recyclerView.adapter = LetterAdapter()
}
```

Here you use an `if` statement to assign the layout manager. In addition to setting the `layoutManager`, this code also assigns the adapter. `LetterAdapter` is used for both list and grid layouts.

3. When you initially set up the menu in xml, you gave it a static icon. However, after toggling the layout, you should update the icon to reflect its new function—switching back to the list layout. Here, simply set the linear and grid layout icons, based on the layout the button will switch back to, next time it's tapped.

```
private fun setIcon(menuItem: MenuItem?) {
  if (menuItem == null)
     return
  // Set the drawable for the menu icon based on which LayoutManager is currently in use

  // An if-clause can be used on the right side of an assignment if all paths return a value.
  // The following code is equivalent to
  // if (isLinearLayoutManager)
  //     menu.icon = ContextCompat.getDrawable(this, R.drawable.ic_grid_layout)
  // else menu.icon = ContextCompat.getDrawable(this, R.drawable.ic_linear_layout)
  menuItem.icon =
     if (isLinearLayoutManager)
        ContextCompat.getDrawable(this, R.drawable.ic_grid_layout)
     else ContextCompat.getDrawable(this, R.drawable.ic_linear_layout)
}
```

The icon is conditionally set based on the `isLinearLayoutManager` property.

For your app to actually use the menu, you need to override two more methods.

- `onCreateOptionsMenu`: where you inflate the options menu and perform any additional setup
- `onOptionsItemSelected`: where you'll actually call `chooseLayout()` when the button is selected.

1. Override `onCreateOptionsMenu` as follows

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
  menuInflater.inflate(R.menu.layout_menu, menu)

  val layoutButton = menu?.findItem(R.id.action_switch_layout)
  // Calls code to set the icon based on the LinearLayoutManager of the RecyclerView
  setIcon(layoutButton)

  return true
}
```

Nothing fancy here. After inflating the layout, you call `setIcon()` to ensure the icon is correct, based on the layout. The method returns a `Boolean`—you return `true` here since you want the options menu to be created.

2. Implement as shown `onOptionsItemSelected` with just a few more lines of code.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  return when (item.itemId) {
     R.id.action_switch_layout -> {
        // Sets isLinearLayoutManager (a Boolean) to the opposite value
        isLinearLayoutManager = !isLinearLayoutManager
        // Sets layout and icon
        chooseLayout()
        setIcon(item)
        return true
     }
```

```
        //  Otherwise, do nothing and use the core event handling

        // when clauses require that all possible paths be accounted for explicitly,
        //  for instance both the true and false cases if the value is a Boolean,
        //  or an else to catch all unhandled cases.
        else -> super.onOptionsItemSelected(item)
    }
}
```

This is called any time a menu item is tapped so you need to be sure to check which menu item is tapped. You use a `when` statement, above. If the `id` matches the `action_switch_layout` menu item, you negate the value of `isLinearLayoutManager`. Then, call `chooseLayout()` and `setIcon()` to update the UI accordingly.

One more thing, before you run the app. Since the layout manager and adapter are now set in `chooseLayout()`, you should replace that code in `onCreate()` to call your new method. `onCreate()` should look like the following after the change.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)

   val binding = ActivityMainBinding.inflate(layoutInflater)
   setContentView(binding.root)

   recyclerView = binding.recyclerView
   // Sets the LinearLayoutManager of the recyclerview
   chooseLayout()
}
```

Now run your app and you should be able to toggle between list and grid views using the menu button.

11. Summary

- Explicit intents are used to navigate to specific activities in your app.
- Implicit intents correspond to specific actions (like opening a link, or sharing an image) and let the system determine how to fulfill the intent.
- Menu options allow you to add buttons and menus to the app bar.
- Companion objects provide a way to associate reusable constants with a type, rather than an instance of that type.

To perform an intent:

- Get a reference to the context.
- Create an `Intent` object providing either an activity or intent type (depending on whether it's explicit or implicit).
- Pass any needed data by calling `putExtra()`.
- Call `startActivity()` passing in the `intent` object.

1. Welcome

Introduction

In this codelab, you learn more about a fundamental part of Android: the *activity*. The *activity lifecycle* is the set of states an activity can be in during its lifetime. The lifecycle extends from when the activity is initially created to when it is destroyed and the system reclaims that activity's resources. As a user navigates between activities in your app (and into and out of your app), those activities each transition between different states in the activity lifecycle.

As an Android developer, you need to understand the activity lifecycle. If your activities do not correctly respond to lifecycle state changes, your app could generate strange bugs, confusing behavior for your users, or use too many Android system resources. Understanding the Android lifecycle, and responding correctly to lifecycle state changes, is critical to being a good Android citizen.

What you'll learn

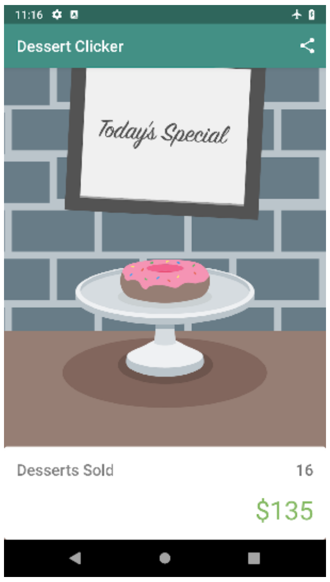
- How to print logging information to the Logcat.
- The basics of the `Activity` lifecycle, and the callbacks that are invoked when the activity moves between states.
- How to override lifecycle callback methods to perform operations at different times in the activity lifecycle.

What you'll do

- Modify a starter app called DessertClicker to add logging information that's displayed in the Logcat.
- Override lifecycle callback methods and log changes to the activity state.
- Run the app and note the logging information that appears as the activity is started, stopped, and resumed.
- Implement the `onSaveInstanceState()` method to retain app data that may be lost if the device configuration changes. Add code to restore that data when the app starts again.

2. App overview

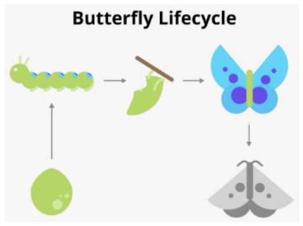
In this codelab, you work with a starter app called DessertClicker. In this app, each time the user taps a dessert on the screen, the app "purchases" the dessert for the user. The app updates values in the layout for the number of desserts that were purchased, and for the total amount the user spent.



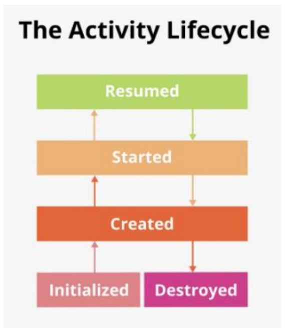
This app contains several bugs related to the Android lifecycle: For example, in certain circumstances, the app resets the dessert values to 0. Understanding the Android lifecycle will help you understand why these problems happen, and how to fix them.

3. Explore the lifecycle methods and add basic logging

Every activity has what is known as a *lifecycle*. This is an allusion to plant and animal lifecycles, like the lifecycle of this butterfly—the different states of the butterfly show its growth from birth to fully formed adulthood to death.



Similarly, the activity lifecycle is made up of the different states that an activity can go through, from when the activity is first initialized to when it is finally destroyed and its memory reclaimed by the system. As the user starts your app, navigates between activities, navigates inside and outside of your app, the activity changes state. The diagram below shows all the activity lifecycle states. As their names indicate, these states represent the status of the activity.



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 `AppCompatActivity`, 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.

It's important to know when these callbacks are invoked 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 what's going on.

Step 1: Examine the onCreate() method and add logging

To figure out what's going on with the Android lifecycle, it's helpful to know when the various lifecycle methods are called. This will help you hunt down where things are going wrong in DessertClicker.

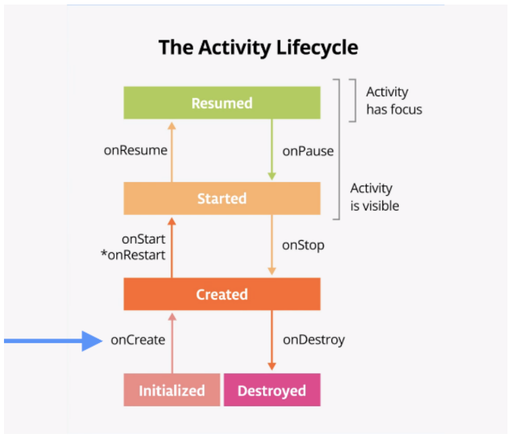
A simple way to do that is to use the Android logging functionality. Logging enables you to write short messages to a console while the app runs, and you can use it to show you when different callbacks are triggered.

1. Run the Dessert Clicker app, and tap several times on the picture of the dessert. Note how the value for **Desserts Sold** and the total dollar amount changes.

2. Open MainActivity.kt and examine the onCreate() method for this activity:

```
override fun onCreate(savedInstanceState: Bundle?) {
...
}
```

In the activity lifecycle diagram, you may have recognized the onCreate() method, because you've used this callback before. It's the one method every activity must implement. The onCreate() method is where you should do any one-time initializations for your activity. For example, in onCreate() you inflate the layout, define click listeners, or set up view binding.



The onCreate() lifecycle method is called once, just after the activity is initialized (when the new Activity object is created 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 methods.

3. In the onCreate() method, just after the call to super.onCreate(), add the following line.

```
Log.d("MainActivity", "onCreate Called")
```

4. Import the Log class if necessary (press Alt+Enter, or Option+Enter on a Mac, and select **Import**.) If you enabled auto imports, this should happen automatically.

```
import android.util.Log
```

The Log class writes messages to the **Logcat**. The **Logcat** is the console for logging messages. Messages from Android about your app appear here, including the messages you explicitly send to the log with the Log.d() method or other Log class methods.

There are three parts to this command:

- The *priority* of the log message, that is, how important the message is. In this case, the Log.d() method writes a debug message. Other methods in the Log class include Log.i() for informational messages, Log.e() for errors, Log.w() for warnings, or Log.v() for verbose messages.
- The log *tag* (the first parameter), in this case "MainActivity". The tag is a string that lets you more easily find your log messages in the Logcat. The tag is typically the name of the class.
- The actual log *message* (the second parameter), is a short string, which in this case is "onCreate called".

Note: A good convention is to declare a TAG constant in your class:

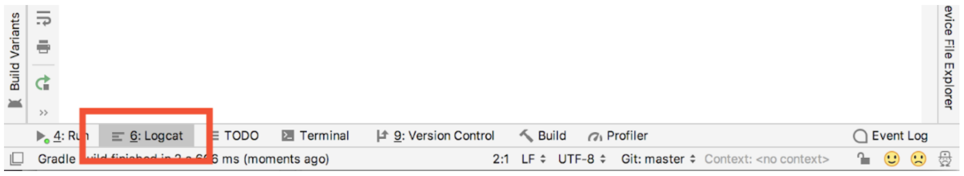
```
const val TAG = "MainActivity"
```

and use that in subsequent calls to the log methods, like below:

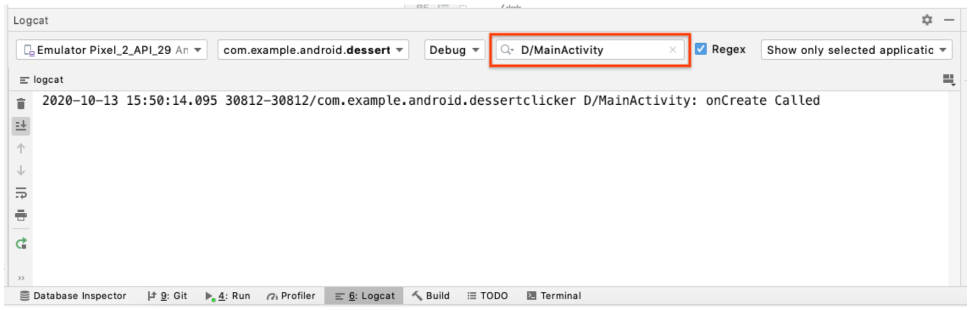
```
Log.d(TAG, "onCreate Called")
```

A *compile-time constant* is a value that won't change. Use `const` before a variable declaration to mark it as a compile-time constant.

5. Compile and run the DessertClicker app. You don't see any behavior differences in the app when you tap the dessert. In Android Studio, at the bottom of the screen, click the **Logcat** tab.



6. In the **Logcat** window, type `D/MainActivity` into the search field.



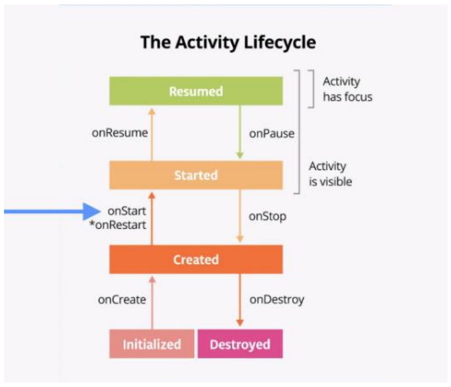
```
2020-10-13 15:50:14.095 30812-30812/com.example.android.dessertclicker D/MainActivity: onCreate Called
```

The Logcat can contain many messages, most of which aren't useful to you. You can filter the Logcat entries in many ways, but searching is the easiest. Because you used `MainActivity` as the log tag in your code, you can use that tag to filter the log. Adding `D/` at the start means that this is a debug message, created by `Log.d()`.

Your log message includes the date and time, the name of the package (`com.example.android.dessertclicker`), your log tag (with `D/` at the start), and the actual message. Because this message appears in the log, you know that `onCreate()` has been executed.

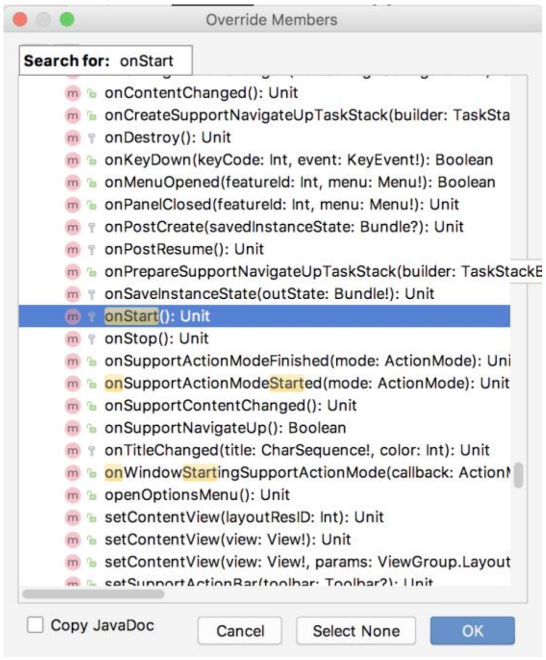
Step 2: Implement the onStart() method

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 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. In Android Studio, with MainActivity.kt open and the cursor within the MainActivity class, select **Code > Override Methods** or press Control+o (Command+o on Mac). A dialog appears with a huge list of all the methods you can override in this class.



```
onContentChanged(): Unit
onCreateSupportNavigateUpTaskStack(builder: TaskSta
onDestroy(): Unit
onKeyDown(keyCode: Int, event: KeyEvent!): Boolean
onMenuOpened(featureId: Int, menu: Menu!): Boolean
onPanelClosed(featureId: Int, menu: Menu!): Unit
onPostCreate(savedInstanceState: Bundle?): Unit
onPostResume(): Unit
onPrepareSupportNavigateUpTaskStack(builder: TaskStackB
onSaveInstanceState(outState: Bundle!): Unit
onStart(): Unit
onStop(): Unit
onSupportActionModeFinished(mode: ActionMode): Uni
onSupportActionModeStarted(mode: ActionMode): Unit
onSupportContentChanged(): Unit
onSupportNavigateUp(): Boolean
onTitleChanged(title: CharSequence!, color: Int): Unit
onWindowStartingSupportActionMode(callback: ActionM
openOptionsMenu(): Unit
setContentView(layoutResID: Int): Unit
setContentView(view: View!): Unit
setContentView(view: View!, params: ViewGroup.Layout
```

2. Start entering onStart to search for the right method. To scroll to the next matching item, use the down arrow. Choose onStart() from the list, and click **OK** to insert the boilerplate override code. The code looks like this:

```
override fun onStart() {
   super.onStart()
}
```

3. Add the following constant at the top level of the MainActivity.kt, that is above the class declaration, class MainActivity.

```
const val TAG = "MainActivity"
```

4. Inside the onStart() method, add a log message:

```
override fun onStart() {
   super.onStart()
   Log.d(TAG, "onStart Called")
}
```

5. Compile and run the DessertClicker app, and open the **Logcat** pane. Type D/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.
6. 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 usually not called again.

```
16:19:59.125 31107-31107/com.example.android.dessertclicker D/MainActivity: onCreate Called

16:19:59.372 31107-31107/com.example.android.dessertclicker D/MainActivity: onStart Called

16:20:11.319 31107-31107/com.example.android.dessertclicker D/MainActivity: onStart Called
```

Note: As you experiment with your device and observe the lifecycle callbacks, you might notice unusual behavior when you rotate your device. You'll learn about that behavior later in this codelab.

Step 3: Add more log statements

In this step, you implement logging for all the other lifecycle methods.

1. Override the remainder of the lifecycle methods in your `MainActivity`, and add log statements for each one. Here's the code:

```
override fun onResume() {
   super.onResume()
   Log.d(TAG, "onResume Called")
}

override fun onPause() {
   super.onPause()
   Log.d(TAG, "onPause Called")
}

override fun onStop() {
   super.onStop()
   Log.d(TAG, "onStop Called")
}

override fun onDestroy() {
   super.onDestroy()
   Log.d(TAG, "onDestroy Called")
}

override fun onRestart() {
   super.onRestart()
   Log.d(TAG, "onRestart Called")
}
```

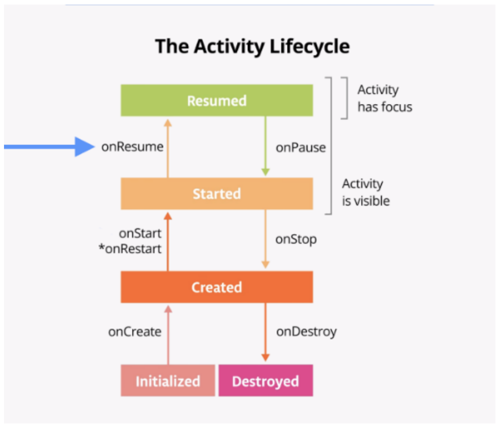
2. Compile and run DessertClicker again and examine Logcat. This time notice that in addition to `onCreate()` and `onStart()`, there's a log message for the `onResume()` lifecycle callback.

```
2020-10-16 10:27:33.244 22064-22064/com.example.android.dessertclicker D/MainActivity: onCreate Called

2020-10-16 10:27:33.453 22064-22064/com.example.android.dessertclicker D/MainActivity: onStart Called

2020-10-16 10:27:33.454 22064-22064/com.example.android.dessertclicker D/MainActivity: onResume Called
```

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

- `onCreate()` to create the app.
- `onStart()` to start it and make it visible on the screen.
- `onResume()` to give the activity focus and make it ready for the user to interact with it.

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



4. Explore lifecycle use cases

Now that the DessertClicker app is set up for logging, you're ready to start using the app in various ways, and ready to explore how the lifecycle callbacks are triggered in response to those uses.

Use case 1: Opening and closing the activity

You start with the most basic use case, which is to start your app for the first time, then close the app down completely.

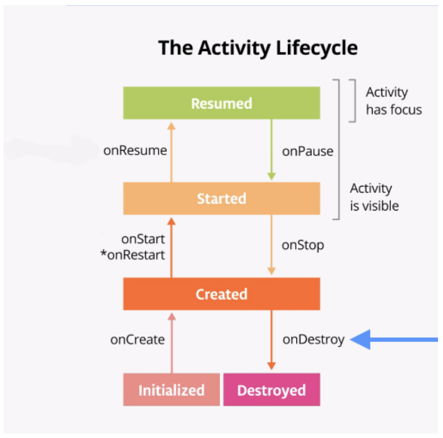
1. Compile and run the DessertClicker app, if it is not already running. As you've seen, the `onCreate()`, `onStart()`, and `onResume()` callbacks are called when the activity starts for the first time.

```
2020-10-16 10:27:33.244 22064-22064/com.example.android.dessertclicker D/MainActivity: onCreate Called
2020-10-16 10:27:33.453 22064-22064/com.example.android.dessertclicker D/MainActivity: onStart Called
2020-10-16 10:27:33.454 22064-22064/com.example.android.dessertclicker D/MainActivity: onResume Called
```

2. Tap the cupcake a few times.
3. Tap the **Back** button on the device. Notice in Logcat that `onPause()`, `onStop()`, and `onDestroy()` are called, in that order.

```
2020-10-16 10:31:53.850 22064-22064/com.example.android.dessertclicker D/MainActivity: onPause Called
2020-10-16 10:31:54.620 22064-22064/com.example.android.dessertclicker D/MainActivity: onStop Called
2020-10-16 10:31:54.622 22064-22064/com.example.android.dessertclicker D/MainActivity: onDestroy Called
```

In this case, using the **Back** button causes the activity (and the app) to be entirely closed. The execution of the `onDestroy()` method means that the activity was fully shut down and can be garbage-collected. *Garbage collection* refers to the automatic cleanup of objects that you'll no longer use. After `onDestroy()` is called, the system knows that those resources are discardable, and it starts cleaning up that memory.



Your activity may also be completely shut down if your code manually calls the activity's finish() method, or if the user force-quits the app. (For example, the user can force-quit or close the app in the recents screen.) The Android system may also shut down your activity on its own if your app has not been on-screen for a long time. Android does this to preserve battery, and to allow your app's resources to be used by other apps.

4. Return to the DessertClicker app by finding all open apps on the Overview screen. (Note this is also known as the Recents screen or recent apps.) Here's the Logcat:

```
2020-10-16 10:31:54.622 22064-22064/com.example.android.dessertclicker D/MainActivity: onDestroy Called
2020-10-16 10:38:00.733 22064-22064/com.example.android.dessertclicker D/MainActivity: onCreate Called
2020-10-16 10:38:00.787 22064-22064/com.example.android.dessertclicker D/MainActivity: onStart Called
2020-10-16 10:38:00.788 22064-22064/com.example.android.dessertclicker D/MainActivity: onResume Called
```

The activity was destroyed in the previous step, so when you return to the app, Android starts up a new activity and calls the onCreate(), onStart(), and onResume() methods. Notice that none of the DessertClicker logs from the previous activity has been retained.

Note: The key point here is that **`onCreate()`** and **`onDestroy()`** are only called once during the lifetime of a single activity instance: **`onCreate()`** to initialize the app for the very first time, and **`onDestroy()`** to clean up the resources used by your app.

The onCreate() method is an important step; this is where all your first-time initialization goes, where you set up the layout for the first time by inflating it, and where you initialize your variables.

Use case 2: Navigating away from and back to the activity

Now that you've started the app and completely closed it, you've seen most of the lifecycle states for when the activity gets created for the first time. You've also seen all the lifecycle states that the activity goes through when it gets completely shut down and destroyed. But as users interact with their Android devices, they switch between apps, return home, start new apps, and handle interruptions by other activities such as phone calls.

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, this is known as putting the activity into the *background*. (The opposite of this is when the activity is in the *foreground*, or on screen.)
- 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.

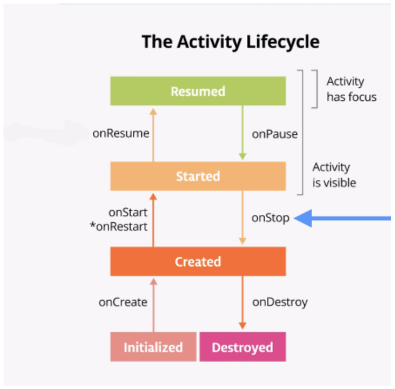
When your app is in the background, it generally should not be actively running, to preserve system resources and battery life. You use the Activity lifecycle and its callbacks to know when your app is moving to the background so that you can pause any ongoing operations. Then you restart the operations when your app comes into the foreground.

In this step, you look at the activity lifecycle when the app goes into the background and returns again to the foreground.

1. With the DessertClicker app running, click the cupcake a few times.
2. Press the **Home** button on your device and observe the Logcat in Android Studio. Returning to the home screen puts your app into the background rather than shutting down the app altogether. Notice that the `onPause()` method and `onStop()` methods are called, but `onDestroy()` is not.

```
2020-10-16 10:41:05.383 22064-22064/com.example.android.dessertclicker D/MainActivity: onPause Called
2020-10-16 10:41:05.966 22064-22064/com.example.android.dessertclicker D/MainActivity: onStop Called
```

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



3. Use the recents screen to return to the app. Notice in Logcat that the activity is restarted with `onRestart()` and `onStart()`, then resumed with `onResume()`.

```
2020-10-16 10:42:18.144 22064-22064/com.example.android.dessertclicker D/MainActivity: onRestart Called
2020-10-16 10:42:18.158 22064-22064/com.example.android.dessertclicker D/MainActivity: onStart Called
2020-10-16 10:42:18.158 22064-22064/com.example.android.dessertclicker D/MainActivity: onResume Called
```

   When the activity returns to the foreground, the `onCreate()` method is not called again. The activity object was not destroyed, so it doesn't need to be created again. Instead of `onCreate()`, the `onRestart()` method is called. Notice that this time when the activity returns to the foreground, the **Desserts Sold** number is retained.

4. Start at least one app other than DessertClicker so that the device has a few apps in its recents screen.
5. Bring up the recents screen and open another recent activity. Then go back to recent apps and bring DessertClicker back to the foreground.

   Notice that you see the same callbacks in Logcat here as when you pressed the Home button. `onPause()` and `onStop()` are called when the app goes into the background, and then `onRestart()`, `onStart()`, and `onResume()` when it comes back.

   Note: The important point here is that `onStart()` and `onStop()` are called multiple times as the user navigates to and from the activity.

These methods are called when the app is stopped and moved into the background, or when the app is started again when it returns to the foreground. If you need to do some work in your app during these cases, then override the relevant lifecycle callback method.

So what about `onRestart()`? The `onRestart()` method is much like `onCreate()`. Either `onCreate()` or `onRestart()` is called before the activity becomes visible. The `onCreate()` method is called only the first time, and `onRestart()` is called after that. The `onRestart()` method is a place to put code that you only want to call if your activity is **not** being started for the first time.

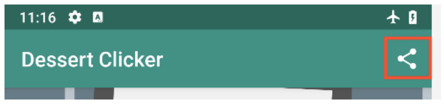
Use case 3: Partially hide the activity

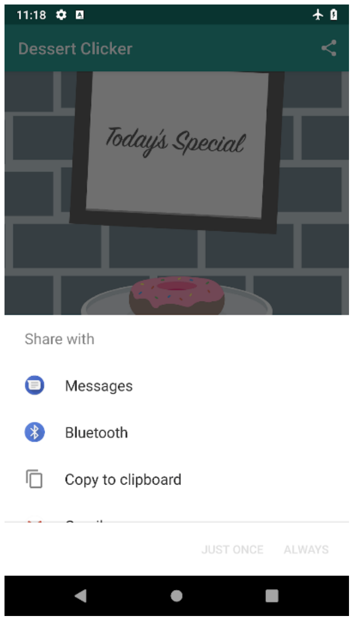
You've learned that when an app is started and `onStart()` is called, the app becomes visible on the screen. When the app is resumed and `onResume()` is called, the app gains the user focus, that is, the user can interact with the app. The part of the lifecycle in which the app is fully on-screen and has user focus is called the *interactive* lifecycle.

When the app goes into the background, the focus is lost after `onPause()`, and the app is no longer visible after `onStop()`.

The difference between focus and visibility is important because it is possible for an activity to be *partially* visible on the screen, but not have the user focus. In this step, you look at one case where an activity is partially visible, but doesn't have user focus.

1. With the DessertClicker app running, click the **Share** button in the top right of the screen.
2. The sharing activity appears in the lower half of the screen, but the activity is still visible in the top half.





2. Examine Logcat and note that only `onPause()` was called.

```
2020-10-16 11:00:53.857 22064-22064/com.example.android.dessertclicker D/MainActivity: onPause Called
```

In this use case, `onStop()` is not called, because the activity is still partially visible. But the activity does not have user focus, and the user can't interact with it—the "share" activity that's in the foreground has the user focus.

Why is this difference important? The interruption with only `onPause()` usually lasts a short time before returning to your activity or navigating to another activity or app. You generally want to keep updating the UI so the rest of your app doesn't appear to freeze.

Whatever code runs in `onPause()` blocks other things from displaying, so keep the code in `onPause()` lightweight. For example, if a phone call comes in, the code in `onPause()` may delay the incoming-call notification.

3. Click outside the share dialog to return to the app, and notice that `onResume()` is called.

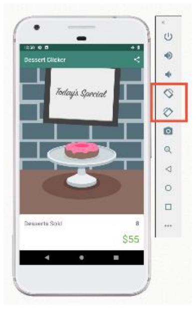
   Both `onResume()` and `onPause()` have to do with focus. The `onResume()` method is called when the activity has focus, and `onPause()` is called when the activity loses focus.

5. Explore configuration changes

There's another case in managing the activity lifecycle that is important to understand: how configuration changes affect the lifecycle of your activities.

A *configuration change* happens 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 may need to change to fit the new orientation. Let's look at how the app behaves in this scenario.

Data loss on device rotation

1. Compile and run your app, and open Logcat.
2. Rotate the device or emulator to landscape mode. You can rotate the emulator left or right with the rotation buttons, or with the `Control` and arrow keys (`Command` and arrow keys on a Mac). 
3. Examine the output in Logcat. Filter the output on `MainActivity`.

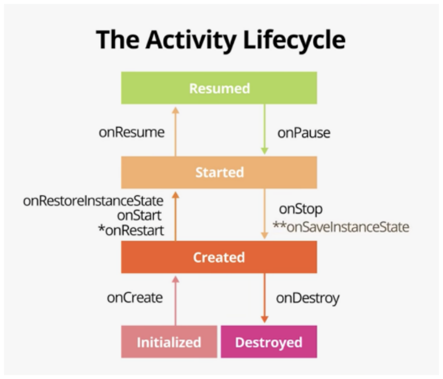
```
2020-10-16 11:03:09.618 23206-23206/com.example.android.dessertclicker D/MainActivity: onCreate Called
2020-10-16 11:03:09.806 23206-23206/com.example.android.dessertclicker D/MainActivity: onStart Called
2020-10-16 11:03:09.808 23206-23206/com.example.android.dessertclicker D/MainActivity: onResume Called
2020-10-16 11:03:24.488 23206-23206/com.example.android.dessertclicker D/MainActivity: onPause Called
2020-10-16 11:03:24.490 23206-23206/com.example.android.dessertclicker D/MainActivity: onStop Called
2020-10-16 11:03:24.493 23206-23206/com.example.android.dessertclicker D/MainActivity: onDestroy Called
2020-10-16 11:03:24.520 23206-23206/com.example.android.dessertclicker D/MainActivity: onCreate Called
2020-10-16 11:03:24.569 23206-23206/com.example.android.dessertclicker D/MainActivity: onStart Called
```

Notice that when the device or emulator rotates the screen, the system calls all the lifecycle callbacks to shut down the activity. Then, as the activity is re-created, the system calls all the lifecycle callbacks to start the activity.

4. When the device is rotated and the activity is shut down and re-created, the activity starts up with default values—the number of desserts sold and the revenue have reset to zeroes.

Use onSaveInstanceState() to save bundle data

The `onSaveInstanceState()` method is a callback you use to save any data that you might need if the `Activity` is destroyed. In the lifecycle callback diagram, `onSaveInstanceState()` is called after the activity has been stopped. It's called every time your app goes into the background.



Think of the `onSaveInstanceState()` call as a safety measure; it gives you a chance to save a small amount of information to a bundle as your activity exits the foreground. The system saves this data now because if it waited until it was shutting down your app, the system might be under resource pressure.

Note: Sometimes Android shuts down an entire app process, which includes every activity associated with the app. Android does this kind of shutdown when the system is stressed and in danger of visually lagging, so no additional callbacks or code is run at this point. Your app's process is simply shut down, silently, in the background. But to the user, it doesn't look like the app has been closed. When the user navigates back to an app that the Android system has shut down, Android restarts that app. You'll want to ensure that the user doesn't experience any data loss when this happens.

Saving the data each time ensures that updated data in the bundle is available to restore, if it is needed.

1. In `MainActivity`, override the `onSaveInstanceState()` callback, and add a log statement.

```
override fun onSaveInstanceState(outState: Bundle) {
   super.onSaveInstanceState(outState)

   Log.d(TAG, "onSaveInstanceState Called")
}
```

Note: There are two overrides for **`onSaveInstanceState()`**, one with just an **`outState`** parameter, and one that includes **`outState`** and **`outPersistentState`** parameters. Use the one shown in the code above, with the single **`outState`** parameter.

2. Compile and run the app, and click the **Home** button to put it into the background. Notice that the `onSaveInstanceState()` callback occurs just after `onPause()` and `onStop()`:

```
2020-10-16 11:05:21.726 23415-23415/com.example.android.dessertclicker D/MainActivity: onPause Called
2020-10-16 11:05:22.382 23415-23415/com.example.android.dessertclicker D/MainActivity: onStop Called
```

```
2020-10-16 11:05:22.393 23415-23415/com.example.android.dessertclicker D/MainActivity: onSaveInstanceState Called
```

3. At the top of the file, just before the class definition, add these constants:

```
const val KEY_REVENUE = "revenue_key"
const val KEY_DESSERT_SOLD = "dessert_sold_key"
```

   You will use these keys for both saving and retrieving data from the instance state bundle.

4. Scroll down to `onSaveInstanceState()`, and notice the `outState` parameter, which is of type `Bundle`.

   A Bundle is a collection of key-value pairs, where the keys are always strings. You can put simple data, such as `Int` and `Boolean` values, into the bundle. Because the system keeps this bundle in memory, it's a best practice to keep the data in the bundle small. The size of this bundle is also limited, though the size varies from device to device. If you store too much data, you risk crashing your app with the `TransactionTooLargeException` error. 5. In `onSaveInstanceState()`, put the `revenue` value (an integer) into the bundle with the `putInt()` method:

```
outState.putInt(KEY_REVENUE, revenue)
```

   The `putInt()` method (and similar methods from the `Bundle` class like `putFloat()` and `putString()` takes two arguments: a string for the key (the `KEY_REVENUE` constant), and the actual value to save.

6. Repeat the same process with the number of desserts sold:

```
outState.putInt(KEY_DESSERT_SOLD, dessertsSold)
```

Use onCreate() to restore bundle data

The Activity state can be restored in `onCreate(Bundle)` or `onRestoreInstanceState(Bundle)` (the `Bundle` populated by `onSaveInstanceState()` method will be passed to both lifecycle callback methods).

1. Scroll up to `onCreate()`, and examine the method signature:

```
override fun onCreate(savedInstanceState: Bundle) {
```

   Notice that `onCreate()` gets a `Bundle` each time it is called. When your activity is restarted due to a process shut down, the bundle that you saved is passed to `onCreate()`. If your activity was starting fresh, this `Bundle` in `onCreate()` is `null`. So if the bundle is not `null`, you know you're "re-creating" the activity from a previously known point.

   Note: If the activity is being re-created, the **`onRestoreInstanceState()`** callback is called after **`onStart()`**, also with the bundle. Most of the time, you restore the activity state in **`onCreate()`**. But because **`onRestoreInstanceState()`** is called after **`onStart()`**, if you ever need to restore some state after **`onCreate()`** is called, you can use **`onRestoreInstanceState()`**.

2. Add this code to `onCreate()`, just after the `binding` variable is set:

```
if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
}
```

The test for `null` determines whether there is data in the bundle, or if the bundle is `null`, which in turn tells you if the app has been started fresh or has been re-created after a shutdown. This test is a common pattern for restoring data from the bundle.

Notice that the key you used here (`KEY_REVENUE`) is the same key you used for `putInt()`. To make sure you use the same key each time, it is a best practice to define those keys as constants. You use `getInt()` to get data out of the bundle, just as you used `putInt()` to put data into the bundle. The `getInt()` method takes two arguments:

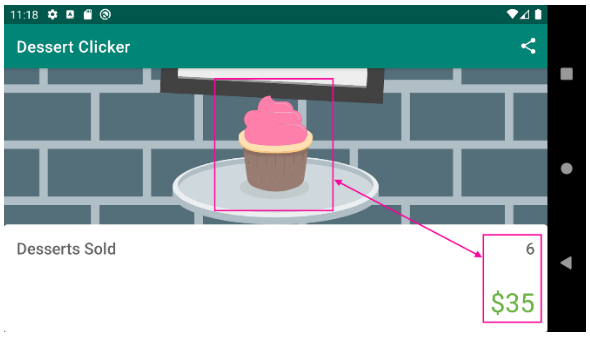
- A string that acts as the key, for example `"key_revenue"` for the revenue value.
- A default value in case no value exists for that key in the bundle.

The integer you get from the bundle is then assigned to the `revenue` variable, and the UI will use that value.

3. Add `getInt()` methods to restore the revenue and the number of desserts sold.

```
if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
   dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
}
```

4. Compile and run the app. Press the cupcake at least five times until it switches to a donut.
5. Rotate the device. Notice that this time the app displays the correct revenue and desserts sold values from the bundle. But also notice that the dessert has returned to a



cupcake. There's one more thing left to do to ensure that the app returns from a shutdown exactly the way it was left.

6. In `MainActivity`, examine the `showCurrentDessert()` method. Notice that this method determines which dessert image should be displayed in the activity based on the current number of desserts sold and the list of desserts in the `allDesserts` variable.

```
for (dessert in allDesserts) {
   if (dessertsSold >= dessert.startProductionAmount) {
       newDessert = dessert
   }
    else break
}
```

This method relies on the number of desserts sold to choose the right image. Therefore, you don't need to do anything to store a reference to the image in the bundle in `onSaveInstanceState()`. In that bundle, you're already storing the number of desserts sold.

7. In `onCreate()`, in the block that restores the state from the bundle, call `showCurrentDessert()`:

```
 if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
   dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
   showCurrentDessert()
}
```

8. Compile and run the app, and rotate the screen. Note now that both the values for desserts sold, total revenue, and the dessert image are correctly restored.

6. Summary

Activity lifecycle

- The *activity lifecycle* is a set of states through which an activity migrates. The activity lifecycle begins when the activity is first created and ends when the activity is destroyed.
- As the user navigates between activities and inside and outside of your app, each activity moves between states in the activity lifecycle.
- Each state in the activity lifecycle has a corresponding callback method you can override in your `Activity` class. The core set of lifecycle methods are: `onCreate()onStart()onPause()onRestart()onResume()onStop()onDestroy()`
- To add behavior that occurs when your activity transitions into a lifecycle state, override the state's callback method.
- To add skeleton override methods to your classes in Android Studio, select **Code > Override Methods** or press `Control+o` (`Command+o` on Mac)

Logging with Log

- The Android logging API, and specifically the `Log` class, enables you to write short messages that are displayed in the Logcat within Android Studio.
- Use `Log.d()` to write a debug message. This method takes two arguments: the log *tag*, typically the name of the class, and the log *message*, a short string.
- Use the **Logcat** window in Android Studio to view the system logs, including the messages you write.

Preserving activity state

- When your app goes into the background, just after `onStop()` is called, app data can be saved to a bundle. Some app data, such as the contents of an `EditText`, is automatically saved for you.
- The bundle is an instance of `Bundle`, which is a collection of keys and values. The keys are always strings.
- Use the `onSaveInstanceState()` callback to save other data to the bundle that you want to retain, even if the app was automatically shut down. To put data into the bundle, use the bundle methods that start with `put`, such as `putInt()`.
- You can get data back out of the bundle in the `onRestoreInstanceState()` method, or more commonly in `onCreate()`. The `onCreate()` method has a `savedInstanceState` parameter that holds the bundle.
- If the `savedInstanceState` variable is `null`, the activity was started without a state bundle and there is no state data to retrieve.
- To retrieve data from the bundle with a key, use the `Bundle` methods that start with `get`, such as `getInt()`.

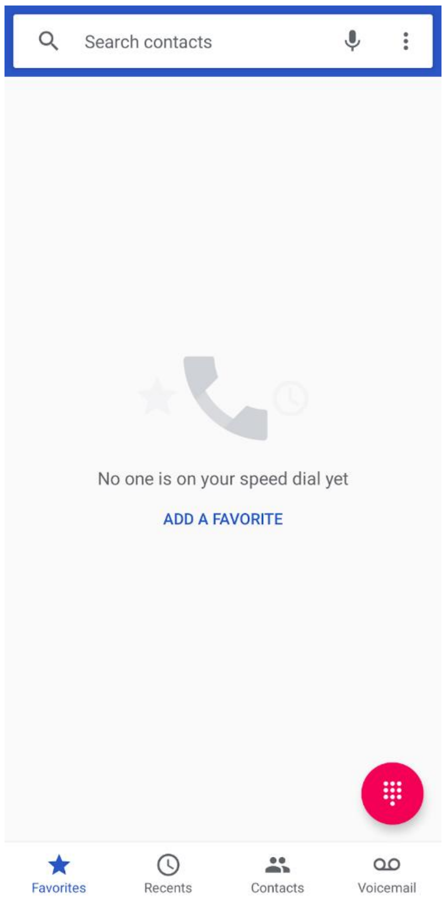
Configuration changes

- A *configuration change* happens when the state of the device changes so radically that the easiest way for the system to resolve the change is to destroy and rebuild the activity.
- The most common example of a configuration change is when the user rotates the device from portrait to landscape mode, or from landscape to portrait mode. A configuration change can also occur when the device language changes or a hardware keyboard is plugged in.
- When a configuration change occurs, Android invokes all the activity lifecycle's shutdown callbacks. Then Android restarts the activity from scratch, running all the lifecycle startup callbacks.
- When Android shuts down an app because of a configuration change, it restarts the activity with the state bundle that is available to `onCreate()`.
- As with process shutdown, save your app's state to the bundle in `onSaveInstanceState()`.

UNIT 3 : MODULE 1

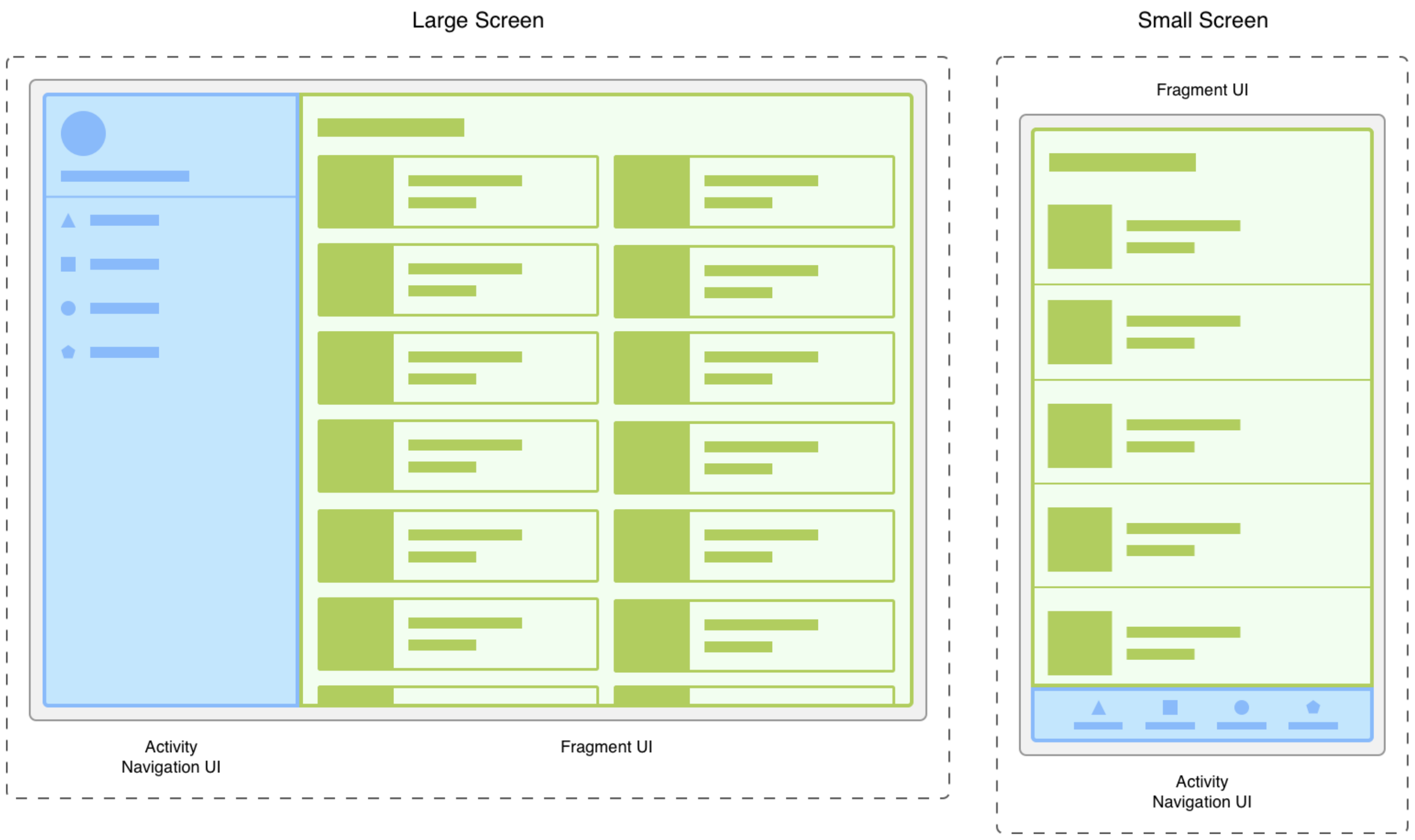
1. Before you begin

In the Activities and Intents codelab, you added intents in the Words app, to navigate between two activities. While this is a useful navigation pattern to know, it's only part of the story of making dynamic user interfaces for your apps. Many Android apps don't need a separate activity for every screen. In fact, many common UI patterns, such as tabs, exist within a single activity, using something called *fragments*.



A fragment is a reusable piece of UI; fragments can be reused and embedded in one or more activities. In the above screenshot, tapping on a tab doesn't trigger an intent to display the next screen. Instead, switching tabs simply swaps out the previous fragment with another fragment. All of this happens without launching another activity.

You can even show multiple fragments at once on a single screen, such as a master-detail layout for tablet devices. In the example below, both the navigation UI on the left and the content on the right can each be contained in a separate fragmen. Both fragments exist simultaneously in the same activity.



As you can see, fragments are an integral part of building high quality apps. In this codelab, you'll learn the basics of fragments, and convert the Words app to use them. You'll also learn how to use the Jetpack Navigation component and work with a new resource file called the **Navigation Graph** to navigate between fragments in the same host activity. By the end of this codelab, you'll come away with the foundational skills for implementing fragments in your next app.

Prerequisites

Before completing this codelab, you should know

- How to add resource XML files and Kotlin files to an Android Studio project.
- How the activity lifecycle works at a high level.
- How to override and implement methods in an existing class.
- How to create instances of Kotlin classes, access class properties, and call methods.
- Basic familiarity with nullable and non-nullable values and know how to safely handle null values.

What you'll learn

- How the fragment lifecycle differs from the activity lifecycle.

- How to convert an existing activity into a fragment.
- How to add destinations to a navigation graph, and pass data between fragments while using the Safe Args plugin.

What you'll build

- You'll modify the Words app to use a single activity and multiple fragments, and navigate between fragments with the Navigation Component.

2. Starter Code

In this codelab, you'll pick up where you left off with the Words app at the end of the Activities and Intents codelab. If you've already completed the codelab for activities and intents, feel free to use your code as a starting point. You can alternately download the code up until this point from GitHub.

Download the starter code for this codelab

This codelab provides starter code for you to extend with features taught in this codelab. Starter code may contain code that is familiar to you from previous codelabs. It may also contain code that is unfamiliar to you, and that you will learn about in later codelabs.

If you use the starter code from GitHub, note that the folder name is `android-basics-kotlin-words-app-activities`. Select this folder when you open the project in Android Studio.

Starter Code URL: https://github.com/google-developer-training/android-basics-kotlin-words-app/tree/activities

3. Fragments and the fragment lifecycle

A fragment is simply a reusable piece of your app's user interface. Like activities, fragments have a lifecycle and can respond to user input. A fragment is always contained within the view hierarchy of an activity when it is shown onscreen. Due to their emphasis on reusability and modularity, it's even possible for multiple fragments to be hosted simultaneously by a single activity. Each fragment manages its own separate lifecycle.

Fragment lifecycle

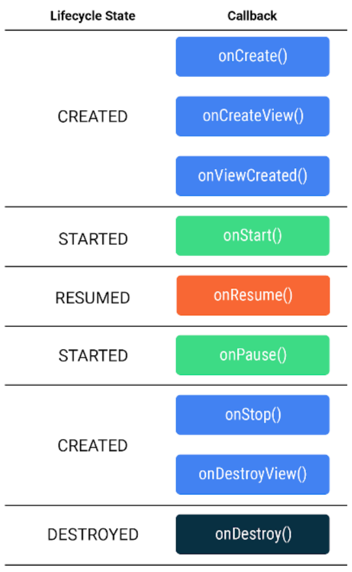
Like activities, fragments can be initialized and removed from memory, and throughout their existence, appear, disappear, and reappear onscreen. Also, just like activities, fragments have a lifecycle with several states, and provide several methods you can override to respond to transitions between them. The fragment lifecycle has five states, represented by the *Lifecycle.State* enum.

- INITIALIZED: A new instance of the fragment has been instantiated.
- CREATED: The first fragment lifecycle methods are called. During this state, the view associated with the fragment is also created.
- STARTED: The fragment is visible onscreen but does not have "focus", meaning it can't respond to user input.
- RESUMED: The fragment is visible and has focus.
- DESTROYED: The fragment object has been de-instantiated.

Also similar to activities, the [Fragment](#) class provides many methods that you can override to respond to lifecycle events.

- `onCreate()`: The fragment has been instantiated and is in the `CREATED` state. However, it's corresponding view has not been created yet.
- `onCreateView()`: This method is where you inflate the layout. The fragment has entered the `CREATED` state.
- `onViewCreated()`: This is called after the view is created. In this method, you would typically bind specific views to properties by calling `findViewById()`.
- `onStart()`: The fragment has entered the `STARTED` state.
- `onResume()`: The fragment has entered the `RESUMED` state and now has focus (can respond to user input).
- `onPause()`: The fragment has re-entered the `STARTED` state. The UI is visible to the user
- `onStop()`: The fragment has re-entered the `CREATED` state. The object is instantiated but is no longer presented on screen.
- `onDestroyView()`: Called right before the fragment enters the `DESTROYED` state. The view has already been removed from memory, but the fragment object still exists.
- `onDestroy()`: The fragment enters the `DESTROYED` state.

The chart below summarizes the fragment lifecycle, and the transitions between states.



Lifecycle State	Callback
CREATED	onCreate()
	onCreateView()
	onViewCreated()
STARTED	onStart()
RESUMED	onResume()
STARTED	onPause()
CREATED	onStop()
	onDestroyView()
DESTROYED	onDestroy()

The lifecycle states and callback methods are quite similar to those used for activities. However, keep in mind the difference with the `onCreate()` method. With activities, you would use this method to inflate the layout and bind views. However, in the fragment lifecycle, `onCreate()` is called before the view is created, so you can't inflate the layout here. Instead, you do this in `onCreateView()`. Then, after the view has been created, the `onViewCreated()` method is called, where you can then bind properties to specific views.

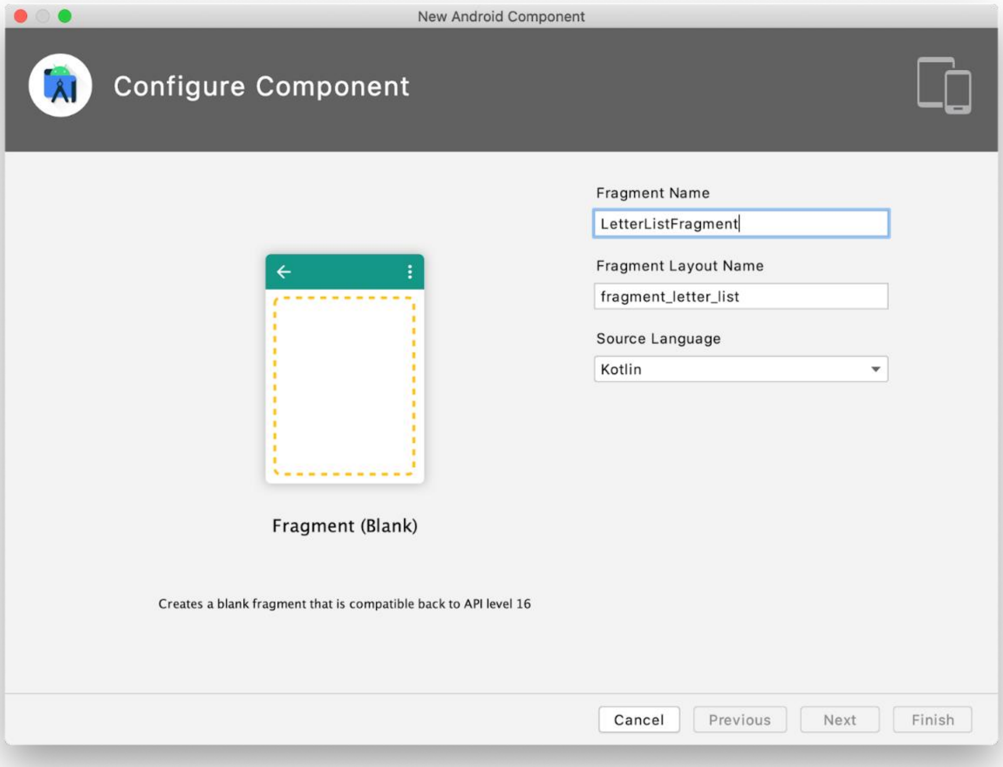
While that probably sounded like a lot of theory, you now know the basics of how fragments work, and how they're similar and different to activities. For the remainder of this codelab, you'll put that knowledge to work. First, you'll migrate the Words app you worked on previously to use a fragment based layout. Then, you'll implement navigation between fragments within a single activity.

4. Create Fragment and layout Files

As with activities, each fragment you add will consist of two files—an XML file for the layout and a Kotlin class to display data and handle user interactions. You'll add a fragment for both the letter list and the word list.

1. With **app** selected in the Project Navigator, add the following fragments (**File > New > Fragment > Fragment (Blank)**) and both a class and layout file should be generated for each.

- For the first fragment, set the **Fragment Name** to `LetterListFragment`. The **Fragment Layout Name** should populate as `fragment_letter_list`.



New Android Component

Configure Component

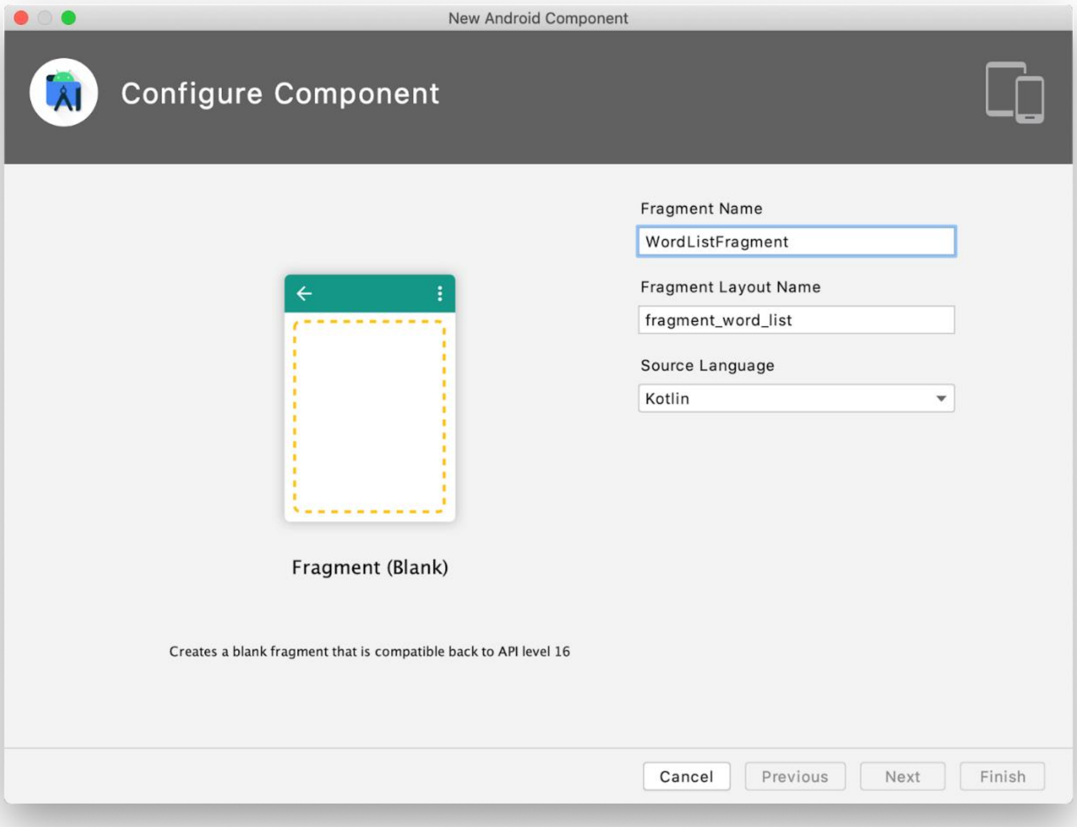
Fragment Name: LetterListFragment

Fragment Layout Name: fragment_letter_list

Source Language: Kotlin

Creates a blank fragment that is compatible back to API level 16

- For the second fragment, set the **Fragment Name** to `WordListFragment`. The **Fragment Layout Name** should populate as `fragment_word_list.xml`.



Fragment Name: WordListFragment
Fragment Layout Name: fragment_word_list
Source Language: Kotlin

Creates a blank fragment that is compatible back to API level 16

2. The generated Kotlin classes for both fragments contain a lot of boilerplate code commonly used when implementing fragments. However, as you're learning about fragments for the first time, go ahead and delete everything except the class declaration for `LetterListFragment` and `WordListFragment` from both files. We'll walk you through implementing the fragments from scratch so that you know how all of the code works. After deleting the boilerplate code, the Kotlin files should look as follows.

   LetterListFragment.kt

```kotlin
package com.example.wordsapp

import androidx.fragment.app.Fragment
class LetterListFragment : Fragment() {

}
```

   WordListFragment.kt

```kotlin
package com.example.wordsapp

import androidx.fragment.app.Fragment
class WordListFragment : Fragment() {

}
```

3. Copy the contents of `activity_main.xml` into `fragment_letter_list.xml` and the contents of `activity_detail.xml` into `fragment_word_list.xml`.
   Update `tools:context` in `fragment_letter_list.xml` to `.LetterListFragment` and `tools:context` in `fragment_word_list.xml` to `.WordListFragment`.

   After the changes, the fragment layout files should look as follows.

   fragment_letter_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".LetterListFragment">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recycler_view"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:clipToPadding="false"
       android:padding="16dp" />

</FrameLayout>
```

   fragment_word_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".WordListFragment">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recycler_view"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:clipToPadding="false"
       android:padding="16dp"
       tools:listitem="@layout/item_view" />

</FrameLayout>
```

5. Implement LetterListFragment

As with activities, you need to inflate the layout and bind individual views. There are just a few minor differences when working with the fragment lifecycle. We'll walk you through the process for setting up the `LetterListFragment`, and then you'll get the chance to do the same for `WordListFragment`.

To implement view binding in `LetterListFragment`, you first need to get a nullable reference to `FragmentLetterListBinding`. Binding classes like this are generated by Android Studio for each layout file, when the `viewBinding` property is enabled under the `buildFeatures` section of the **build.gradle** file. You just need to assign properties in your fragment class for each view in the `FragmentLetterListBinding`.

The type should be `FragmentLetterListBinding?` and it should have an initial value of `null`. Why make it nullable? Because you can't inflate the layout until `onCreateView()` is called. There's a period of time in-between when the instance of `LetterListFragment` is created (when its lifecycle begins with `onCreate()`) and when this property is actually usable. Also keep in mind that fragments' views can be created and destroyed several times throughout the fragment's lifecycle. For this reason you also need to reset the value in another lifecycle method, `onDestroyView()`.

1. In `LetterListFragment.kt`, start by getting a reference to the `FragmentLetterListBinding`, and name the reference `_binding`.

```
private var _binding: FragmentLetterListBinding? = null
```

Because it's nullable, every time you access a property of `_binding`, (e.g. `_binding?.someView`) you need to include the `?` for null safety. However, that doesn't mean you have to litter your code with question marks just because of one null value. If you're certain a value won't be null when you access it, you can append `!!` to its type name. Then you can access it like any other property, without the `?` operator.

NOTE: When making a variable nullable using `!!`, it's a good idea to limit its usage to only one or a few places where you know the value won't be null, just like you know `_binding` will have a value after it is assigned in `onCreateView()`. Accessing a nullable value in this manner is dangerous and can lead to crashes, so use sparingly, if at all.

2. Create a new property, called binding (without the underscore) and set it equal to `_binding!!`.

```
private val binding get() = _binding!!
```

Here, `get()` means this property is "get-only". That means you can **get** the value, but once assigned (as it is here), you can't assign it to something else.

NOTE: In Kotlin, and programming in general, you'll often encounter property names preceded by an underscore. This typically means that the property isn't intended to be accessed directly. In your case, you access the view binding in `LetterListFragment` with the binding property. However, the `_binding` property does not need to be accessed outside of `LetterListFragment`.

3. To display the options menu, override `onCreate()`. Inside `onCreate()` call `setHasOptionsMenu()` passing in `true`.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setHasOptionsMenu(true)
}
```

4. Remember that with fragments, the layout is inflated in `onCreateView()`. Implement `onCreateView()` by inflating the view, setting the value of `_binding`, and returning the root view.

```
override fun onCreateView(
   inflater: LayoutInflater, container: ViewGroup?,
   savedInstanceState: Bundle?
): View? {
   _binding = FragmentLetterListBinding.inflate(inflater, container, false)
   val view = binding.root
   return view
}
```

5. Below the `binding` property, create a property for the recycler view.

   ```
   private lateinit var recyclerView: RecyclerView
   ```

6. Then set the value of the `recyclerView` property in `onViewCreated()`, and call `chooseLayout()` like you did in `MainActivity`. You'll move the `chooseLayout()` method into `LetterListFragment` soon, so don't worry that there's an error.

   ```
   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
      recyclerView = binding.recyclerView
      chooseLayout()
   }
   ```

   Notice how the binding class already created a property for `recyclerView`, and you don't need to call `findViewById()` for each view.

7. Finally, in `onDestroyView()`, reset the `_binding` property to `null`, as the view no longer exists.

   ```
   override fun onDestroyView() {
      super.onDestroyView()
      _binding = null
   }
   ```

8. The only other thing to note is there are some subtle differences with the `onCreateOptionsMenu()` method when working with fragments. While the `Activity` class has a global property called `menuInflater`, Fragment does not have this property. The menu inflater is instead passed into `onCreateOptionsMenu()`. Also note that the `onCreateOptionsMenu()` method used with fragments doesn't require a return statement. Implement the method as shown:

   ```
   override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
      inflater.inflate(R.menu.layout_menu, menu)

      val layoutButton = menu.findItem(R.id.action_switch_layout)
      setIcon(layoutButton)
   }
   ```

9. Move the remaining code for `chooseLayout()`, `setIcon()`, and `onOptionsItemSelected()` from `MainActivity` as-is. The only other difference to note is that because unlike an activity, a fragment is not a [Context](). You can't pass in `this` (referring to the fragment object) as the layout manager's context. However, fragments provide a `context` property you can use instead. The rest of the code is identical to `MainActivity`.

```
private fun chooseLayout() {
  when (isLinearLayoutManager) {
    true -> {
      recyclerView.layoutManager = LinearLayoutManager(context)
      recyclerView.adapter = LetterAdapter()
    }
    false -> {
      recyclerView.layoutManager = GridLayoutManager(context, 4)
      recyclerView.adapter = LetterAdapter()
    }
  }
}

private fun setIcon(menuItem: MenuItem?) {
  if (menuItem == null)
    return

  menuItem.icon =
    if (isLinearLayoutManager)
      ContextCompat.getDrawable(this.requireContext(), R.drawable.ic_grid_layout)
    else ContextCompat.getDrawable(this.requireContext(), R.drawable.ic_linear_layout)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
  return when (item.itemId) {
    R.id.action_switch_layout -> {
      isLinearLayoutManager = !isLinearLayoutManager
      chooseLayout()
      setIcon(item)

      return true
    }
    else -> super.onOptionsItemSelected(item)
  }
}
```

10. Finally, copy over the `isLinearLayoutManager` property from `MainActivity`. Put this right below the declaration of the `recyclerView` property.

```
private var isLinearLayoutManager = true
```

11. Now that all the functionality has been moved to `LetterListFragment`, all the `MainActivity` class needs to do is inflate the layout so that the fragment is displayed in the view. Go ahead and delete everything except `onCreate()` from `MainActivity`. After the changes, `MainActivity` should contain only the following.

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

Your turn

That's it for migrating `MainActivity` to `LettersListFragment`. Migrating the `DetailActivity` is almost identical. Perform the following steps to migrate the code to `WordListFragment`.

1. Copy the companion object from `DetailActivity` to `WordListFragment`. Make sure the reference to `SEARCH_PREFIX` in `WordAdapter` is updated to reference `WordListFragment`.
2. Add a `_binding` variable The variable should be nullable and have `null` as its initial value.
3. Add a get-only variable called binding equal to the `_binding` variable.
4. Inflate the layout in `onCreateView()`, setting the value of `_binding` and returning the root view.
5. Perform any remaining setup in `onViewCreated()`: get a reference to the recycler view, set its layout manager and adapter, and add its item decoration. You'll need to get the letter from the intent. As fragments don't have an `intent` property and shouldn't normally access the intent of the parent activity. For now, you refer to `activity.intent` (rather than `intent` in `DetailActivity`) to get the extras.
6. Reset `_binding` to null in `onDestroyView`.
7. Delete the remaining code from `DetailActivity`, leaving only the `onCreate()` method.

Try to go through the steps on your own before moving on. A detailed walkthrough is available on the next step.

6. Convert DetailActivity to WordListFragment

Hopefully you enjoyed getting the chance to migrate `DetailActivity` to `WordListFragment`. This is almost identical to migrating `MainActivity` to `LetterListFragment`. If you got stuck at any point, the steps are summarized below.

1. First, copy the companion object to `WordListFragment`.

```
companion object {
  val LETTER = "letter"
  val SEARCH_PREFIX = "https://www.google.com/search?q="
}
```

2. Then in `LetterAdapter`, in the `onClickListener()` where you perform the intent, you need to update the call to `putExtra()`, replacing `DetailActivity.LETTER` with `WordListFragment.LETTER`.

```
intent.putExtra(WordListFragment.LETTER, holder.button.text.toString())
```

3. Similarly, in `WordAdapter` you need to update the `onClickListener()` where you navigate to the search results for the word, replacing `DetailActivity.SEARCH_PREFIX` with `WordListFragment.SEARCH_PREFIX`.

```
val queryUrl: Uri = Uri.parse("${WordListFragment.SEARCH_PREFIX}${item}")
```

4. Back in `WordListFragment`, you add a binding variable of type `FragmentWordListBinding?`.

```
private var _binding: FragmentWordListBinding? = null
```

5. You then create a get-only variable so that you can reference views without having to use `?`.

```
private val binding get() = _binding!!
```

6. Then you inflate the layout, assigning the `_binding` variable and returning the root view. Remember that for fragments you do this in `onCreateView()`, not `onCreate()`.

```
override fun onCreateView(
   inflater: LayoutInflater,
   container: ViewGroup?,
   savedInstanceState: Bundle?
): View? {
   _binding = FragmentWordListBinding.inflate(inflater, container, false)
   return binding.root
}
```

7. Next, you implement `onViewCreated()`. This is almost identical to configuring the `recyclerView` in `onCreateView()` in the `DetailActivity`. However, because fragments don't have direct access to the intent, you need to reference it with `activity.intent`. You have to do this in `onCreateView()` however, as there's no guarantee the activity exists earlier in the lifecycle.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
   val recyclerView = binding.recyclerView
   recyclerView.layoutManager = LinearLayoutManager(requireContext())
   recyclerView.adapter = WordAdapter(activity?.intent?.extras?.getString(LETTER).toString(), requireContext())

   recyclerView.addItemDecoration(
       DividerItemDecoration(context, DividerItemDecoration.VERTICAL)
   )
}
```

8. Finally, you can reset the `_binding` variable in `onDestroyView()`.

```
override fun onDestroyView() {
   super.onDestroyView()
   _binding = null
}
```

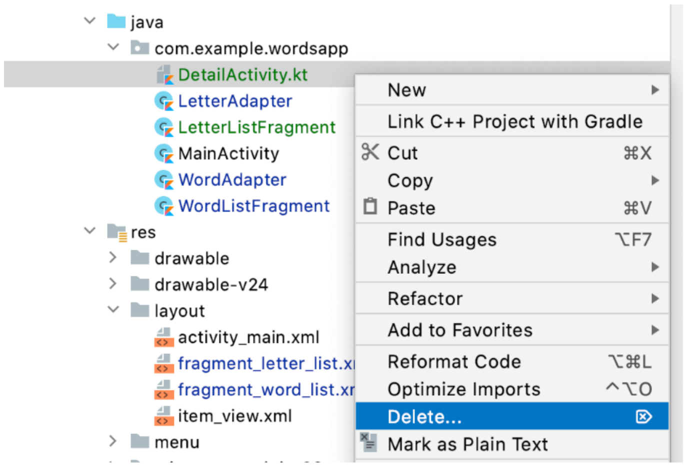
9. With all this functionality moved into WordListFragment, you can now delete the code from DetailActivity. All that should be left is the onCreate() method.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)

   val binding = ActivityDetailBinding.inflate(layoutInflater)
   setContentView(binding.root)
}
```

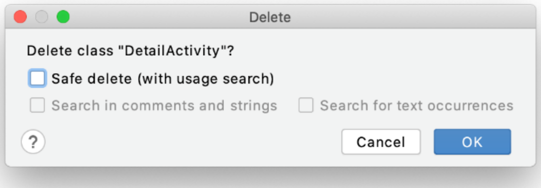
Remove DetailActivity

Now that you've successfully migrated the functionality of `DetailActivity` into `WordListFragment`, you no longer need `DetailActivity`. You can go ahead and delete both the `DetailActivity.kt` and `activity_detail.xml` as well as make a small change to the manifest.

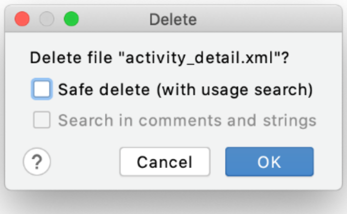
1. First, delete `DetailActivity.kt`



2. Make sure **Safe Delete** is Unchecked and click **OK**.


Delete
Delete class "DetailActivity"?
Safe delete (with usage search)
Search in comments and strings
Search for text occurrences
?
Cancel
OK

3. Next, delete `activity_detail.xml`. Again, make sure **Safe Delete** is unchecked.


Delete
Delete file "activity_detail.xml"?
Safe delete (with usage search)
Search in comments and strings
?
Cancel
OK

4. Finally, as `DetailActivity` no longer exists, remove the following from `AndroidManifest.xml`.

```
<activity
  android:name=".DetailActivity"
  android:parentActivityName=".MainActivity" />
```

After deleting the detail activity, you're left with two fragments (LetterListFragment and WordListFragment) and a single activity (MainActivity). In the next section, you'll learn about the Jetpack Navigation component and edit `activity_main.xml` so that it can display and navigate between fragments, rather than host a static layout.

7. Jetpack Navigation Component

Android Jetpack provides the *Navigation component* to help you handle any navigation implementation, simple or complex, in your app. The Navigation component has three key parts which you'll use to implement navigation in the **Words** app.

- Navigation Graph: The navigation graph is an XML file that provides a visual representation of navigation in your app. The file consists of *destinations* which correspond to individual activities and fragments as well as actions between them which can be used in code to navigate from one destination to another. Just like with layout files, Android Studio provides a visual editor to add destinations and actions to the navigation graph.
- `NavHost`: A `NavHost` is used to display destinations form a navigation graph within an activity. When you navigate between fragments, the destination shown in the `NavHost` is updated. You'll use a built-in implementation, called `NavHostFragment`, in your `MainActivity`.
- `NavController`: The `NavController` object lets you control the navigation between destinations displayed in the `NavHost`. When working with intents, you had to call startActivity to navigate to a new screen. With the Navigation component, you can call the `NavController`'s `navigate()` method to swap the fragment that's displayed. The `NavController` also helps you handle common tasks like responding to the system "up" button to navigate back to the previously displayed fragment.

Navigation Dependency

1. In the project-level `build.gradle` file, in **buildscript > ext**, below `material_version` set the `nav_version` equal to `2.3.1`.

```
buildscript {
   ext {
      appcompat_version = "1.2.0"
      constraintlayout_version = "2.0.2"
      core_ktx_version = "1.3.2"
      kotlin_version = "1.3.72"
      material_version = "1.2.1"
      nav_version = "2.3.1"
   }

   ...


}
```

2. In the app-level `build.gradle` file, add the following to the dependencies group.

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

Safe Args Plugin

When you first implemented navigation in the **Words** app, you used an explicit intent between the two activities. To pass data between the two activities, you called the `putExtra()` method, passing in the selected letter.

Before you start implementing the Navigation component into the **Words** app, you'll also add something called **Safe Args**—a Gradle plugin that will assist you with type safety when passing data between fragments.

Perform the following steps to integrate SafeArgs into your project.

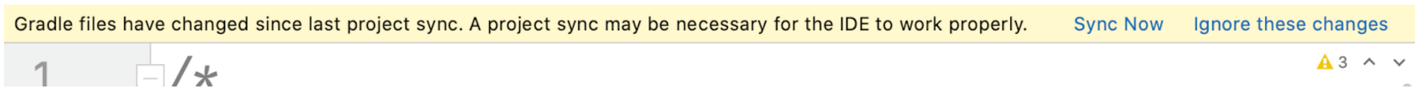
1. In the top-level `build.gradle` file, in **buildscript > dependencies**, add the following classpath.

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
```

2. In the app-level `build.gradle` file, within `plugins` at the top, add `androidx.navigation.safeargs.kotlin`.

```
plugins {
   id 'com.android.application'
   id 'kotlin-android'
   id 'kotlin-kapt'
   id 'androidx.navigation.safeargs.kotlin'
}
```

3. Once you've edited the Gradle files, you may see a yellow banner at the top asking you to sync the project. Click "**Sync Now**" and wait a minute or two while Gradle updates your project's dependencies to reflect your changes.


Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

Once syncing is complete, you're ready to move on to the next step where you'll add a navigation graph.

8. Using the Navigation Graph

Now that you have a basic familiarity with fragments and their lifecycle, it's time for things to get a bit more interesting. The next step is to incorporate the Navigation component. The *navigation component* simply refers to the collection of tools for implementing navigation, particularly between fragments. You'll be working with a new visual editor to help implement navigation between fragments; the Navigation Graph (or NavGraph for short).

What is a Navigation Graph?

The Navigation Graph (or NavGraph for short) is a virtual mapping of your app's navigation. Each screen, or fragment in your case, becomes a possible "destination" that can be navigated to. A `NavGraph` can be represented by an XML file showing how each destination relates to one another.

Behind the scenes, this actually creates a new instance of the `NavGraph` class. However, destinations from the navigation graph are displayed to the user by the `FragmentContainerView`. All you need to do is to create an XML file and define the possible destinations. Then you can use the generated code to navigate between fragments.

Use FragmentContainerView in MainActivity

Because your layouts are now contained in `fragment_letter_list.xml` and `fragment_word_list.xml`, your `activity_main.xml` file no longer needs to contain the layout for the first screen in your app. Instead, you'll repurpose `MainActivity` to contain a `FragmentContainerView` to act as the NavHost for your fragments. From this point forward, all the navigation in the app will take place within the `FragmentContainerView`.

1. Replace the content of the `FrameLayout` in **activity_main.xml** that is `androidx.recyclerview.widget.RecyclerView` with a `FragmentContainerView`. Give it an ID of `nav_host_fragment` and set its height and width to `match_parent` to fill the entire frame layout.

   Replace this:

   ```
   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recycler_view"
       ...
       android:padding="16dp" />
   ```

   With this:

   ```
   <androidx.fragment.app.FragmentContainerView
     android:id="@+id/nav_host_fragment"
     android:layout_width="match_parent"
     android:layout_height="match_parent" />
   ```

2. Below the id attribute, add a `name` attribute and set it to `androidx.navigation.fragment.NavHostFragment`. While you can specify a specific fragment for this attribute, setting it to `NavHostFragment` allows your `FragmentContainerView` to navigate between fragments.

   ```
   android:name="androidx.navigation.fragment.NavHostFragment"
   ```

3. Below the layout_height and layout_width attributes, add an attribute called `app:defaultNavHost` and set it equal to `"true"`. This allows the fragment container to interact with the navigation hierarchy. For example, if the system back button is pressed, then the container will navigate back to the previously shown fragment, just like what happens when a new activity is presented.

   ```
   app:defaultNavHost="true"
   ```

4. Add an attribute called `app:navGraph` and set it equal to `"@navigation/nav_graph"`. This points to an XML file that defines how your app's fragments can navigate to one another. For now, the Android studio will show you an unresolved symbol error. You will address this in the next task.

   ```
   app:navGraph="@navigation/nav_graph"
   ```

5. Finally, because you added two attributes with the app namespace, be sure to add the xmlns:app attribute to the FrameLayout.

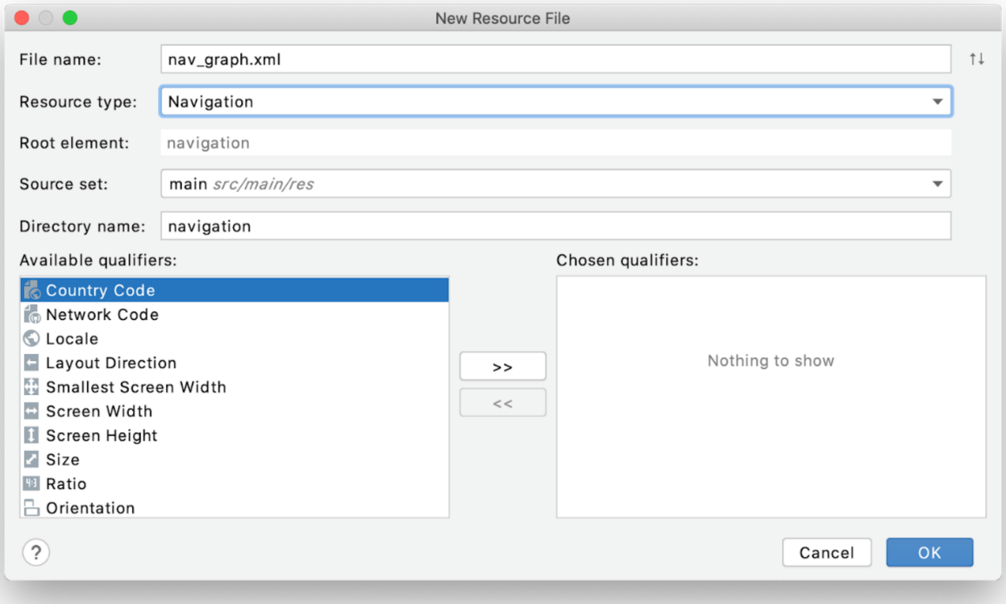
```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">
```

That's all the changes in activity_main.xml. Next up, you'll create the nav_graph file.

Set Up the Navigation Graph

Add a navigation graph file (**File** > **New** > **Android Resource File**) and filling the fields as follows.

- File Name: nav_graph.xml. This is the same as the name you set for the app:navGraph attribute.
- Resource Type: **Navigation**. The **Directory Name** should then automatically change to navigation. A new resource folder called "navigation" will be created.


New Resource File

File name: nav_graph.xml

Resource type: Navigation

Root element: navigation

Source set: main src/main/res

Directory name: navigation

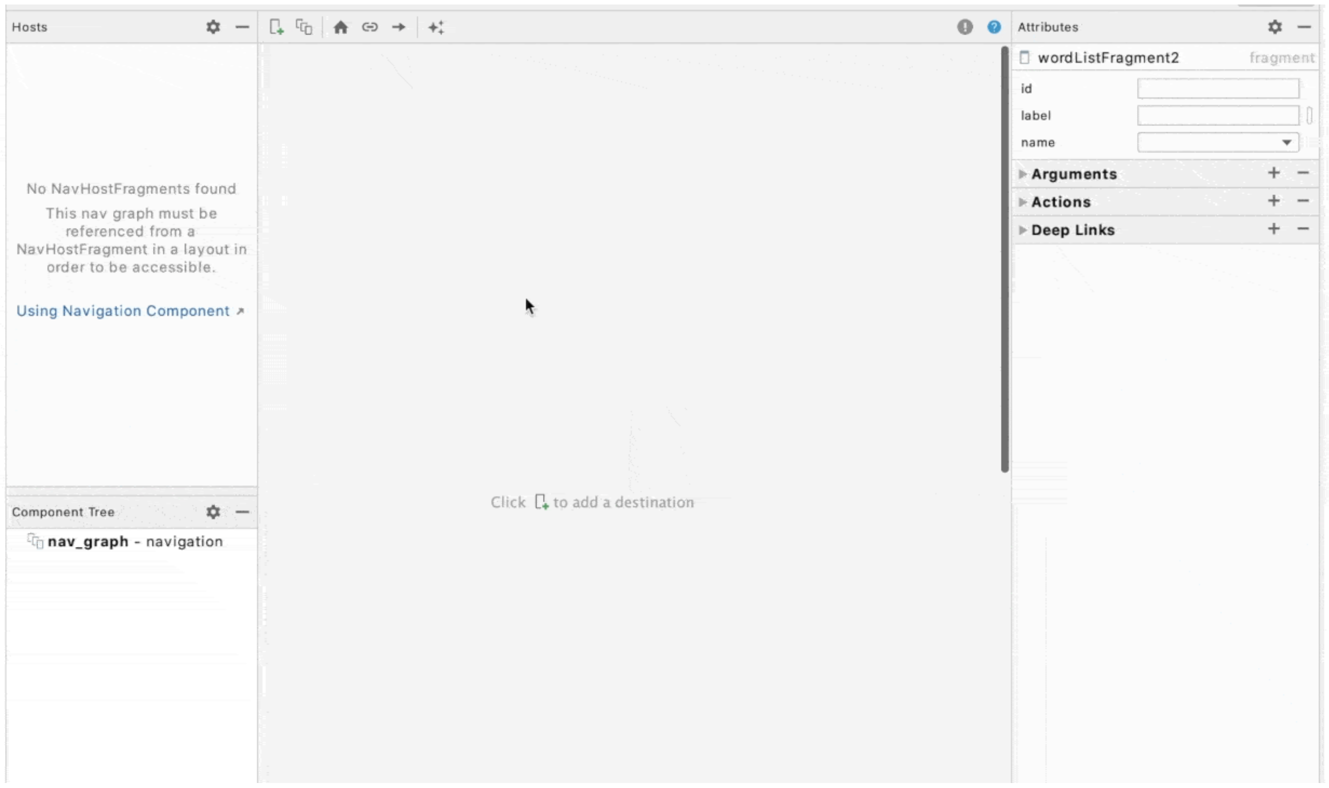
Available qualifiers:

Country Code
Network Code
Locale
Layout Direction
Smallest Screen Width
Screen Width
Screen Height
Size
Ratio
Orientation

Chosen qualifiers:

Nothing to show

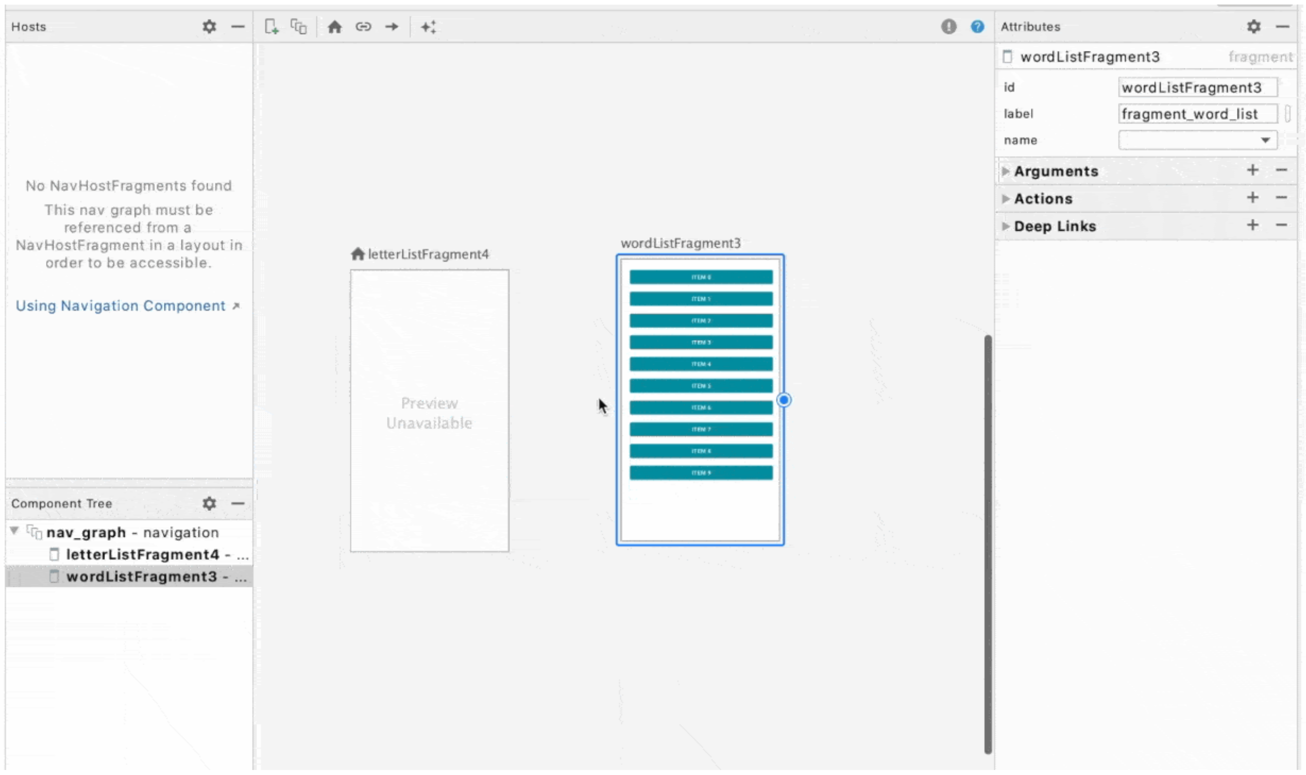
Upon creating the XML file, you're presented with a new visual editor. Because you've already referenced nav_graph in the FragmentContainerView's navGraph property, to add a new destination, click the new button in the top left of the screen and create a destination for each fragment (one for fragment_letter_list and one for fragment_word_list).


Hosts
No NavHostFragments found
This nav graph must be referenced from a NavHostFragment in a layout in order to be accessible.
Using Navigation Component
Component Tree
nav_graph - navigation
Click to add a destination
Attributes
wordListFragment2
fragment
id
label
name
Arguments
Actions
Deep Links

Once added, these fragments should appear on the navigation graph in the middle of the screen. You can also select a specific destination using the component tree that appears on the left.

Create a navigation action

To create a navigation action between the letterListFragment to the wordListFragment destinations, hover your mouse over the letterListFragment destination and drag from the circle that appears on the right onto the wordListFragment destination.



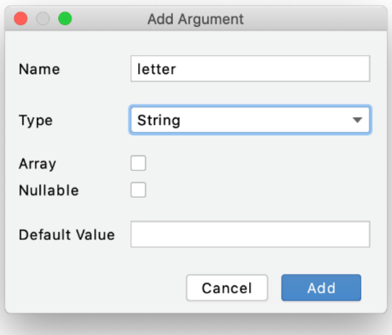
You should now see an arrow has been created to represent the action between the two destinations. Click on the arrow, and you can see in the attributes pane that this action has a name `action_letterListFragment_to_wordListFragment` that can be referenced in code.

Specify Arguments for WordListFragment

When navigating between activities using an intent, you specified an "extra" so that the selected letter could be passed to the `wordListFragment`. Navigation also supports passing parameters between destinations and plus does this in a type safe way.

Select the `wordListFragment` destination and in the attributes pane, under **Arguments**, click the plus button to create a new argument.

The argument should be called `letter` and the type should be `String`. This is where the Safe Args plugin you added earlier comes in. Specifying this argument as a string ensures that a `String` will be expected when your navigation action is performed in code.



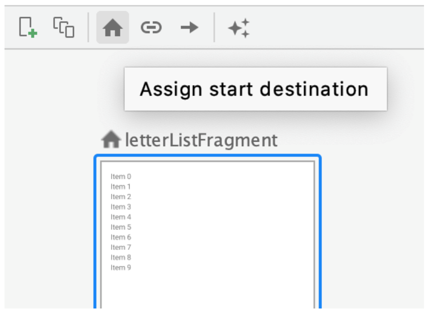
Add Argument

Name	letter
Type	String
Array	☐
Nullable	☐
Default Value	

Setting the Start Destination

While your NavGraph is aware of all the needed destinations, how will the `FragmentContainerView` know which fragment to show first? On the NavGraph, you need to set the letter list as a start destination.

Set the start destination by selecting `letterListFragment` and clicking the **Assign start destination** button.



That's all you need to do with the NavGraph editor for now. At this point, go ahead and build the project. This will generate some code based on your navigation graph so that you can use the navigation action you just created.

Perform the Navigation Action

Open up LetterAdapter.kt to perform the navigation action. This only requires two steps.

1. Delete the contents of the button's onClickListener(). Instead, you need to retrieve the navigation action you just created. Add the following to the onClickListener().

```
val action = LetterListFragmentDirections.actionLetterListFragmentToWordListFragment(letter = holder.button.text.toString())
```

You probably don't recognize some of these class and function names and that's because they've been automatically generated after you built the project. That's where the Safe Args plugin you added in the first step comes in—the actions created on the NavGraph are turned into code that you can use. The names, however, should be fairly intuitive. LetterListFragmentDirections lets you refer to all possible navigation paths starting from the letterListFragment. The function actionLetterListFragmentToWordListFragment()

is the specific action to navigate to the wordListFragment.

Once you have a reference to your navigation action, simply get a reference to your *NavController* (an object that lets you perform navigation actions) and call navigate() passing in the action.

```
holder.view.findNavController().navigate(action)
```

Configure MainActivity

The final piece of setup is in MainActivity. There are just a few changes needed in MainActivity to get everything working.

1. Create a navController property. This is marked as lateinit since it will be set in onCreate.

```
private lateinit var navController: NavController
```

2. Then, after the call to setContentView() in onCreate(), get a reference to the nav_host_fragment (this is the ID of your FragmentContainerView) and assign it to your navController property.

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
navController = navHostFragment.navController
```

3. Then in onCreate(), call setupActionBarWithNavController(), passing in navController. This ensures action bar (app bar) buttons, like the menu option in LetterListFragment are visible.

```
setupActionBarWithNavController(navController)
```

4. Finally, implement onSupportNavigateUp(). Along with setting defaultNavHost to true in the XML, this method allows you to handle the **up** button. However, your activity needs to provide the implementation.

```
override fun onSupportNavigateUp(): Boolean {
  return navController.navigateUp() || super.onSupportNavigateUp()
}
```

At this point, all the components are in-place to get navigation working with fragments. However, now that navigation is performed using fragments instead of the intent, the intent extra for the letter that you use in `WordListFragment` will no longer work. In the next step, you'll update `WordListFragment`, to get the `letter` argument.

NOTE: Because the **`navigateUp()`** function might fail, it returns a **`Boolean`** for whether or not it succeeds. However, you only need to call **`super.onSupportNavigateUp()`** if **`navigateUp()`** returns **`false`**. This works because of the **`||`** operator only requires one of the conditions to be true, so if **`navigateUp()`** returns **`true`**, the right side of the **`||`** expression is never executed. If, however, **`navigateUp()`** is false, then the implementation in the parent class is called. This is called short-circuit evaluation and is a nice little programming trick to know about.

9. Getting Arguments in WordListFragment

Previously, you referenced `activity?.intent` in `WordListFragment` to access the `letter` extra. While this works, this is not a best practice, since fragments can be embedded in other layouts, and in a larger app, it's much harder to assume which activity the fragment belongs to. Furthermore, when navigation is performed using `nav_graph` and safe arguments are used, there are no intents, so trying to access intent extras is simply not going to work.

Thankfully, accessing safe arguments is pretty straightforward, and you don't have to wait until `onViewCreated()` is called either.

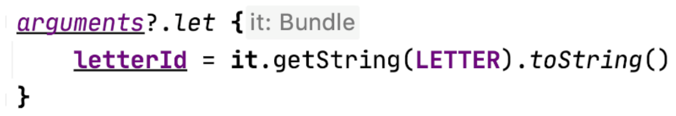
1. In `WordListFragment`, create a `letterId` property. You can mark this as lateinit so that you don't have to make it nullable.

```
private lateinit var letterId: String
```

2. Then override `onCreate()` (not `onCreateView()` or `onViewCreated()`!), add the following.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    arguments?.let {
        letterId = it.getString(LETTER).toString()
    }
}
```

Because it's possible for `arguments` to be optional, notice you call `let()` and pass in a lambda. This code will execute assuming `arguments` is not null, passing in the non null arguments for the `it` parameter. If `arguments` is `null`, however, the lambda will not execute.



```
arguments?.let { it: Bundle
    letterId = it.getString(LETTER).toString()
}
```

While not part of the actual code, Android Studio provides a helpful hint to make you aware of the `it` parameter.

What exactly is a `Bundle`? Think of it as a key-value pair used to pass data between classes, such as activities and fragments. Actually, you've already used a bundle when you called `intent?.extras?.getString()` when performing an intent in the first version of this app. Getting the string from arguments when working with fragments works exactly the same way.

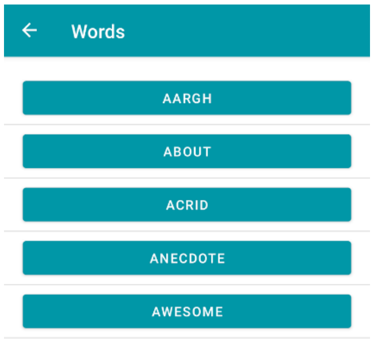
3. Finally, you can access the `letterId` when you set the recycler view's adapter. Replace `activity?.intent?.extras?.getString(LETTER).toString()` in `onViewCreated()` with `letterId`.

```
recyclerView.adapter = WordAdapter(letterId, requireContext())
```

You did it! Take a moment to run your app. It's now able to navigate between two screens, without any intents, and all in a single activity.

10. Update Fragment Labels

You've successfully converted both screens to use fragments. Before any changes were made, the app bar for each fragment had a descriptive title for each activity contained in the app bar. However, after converting to use fragments, this title is missing from the detail activity.

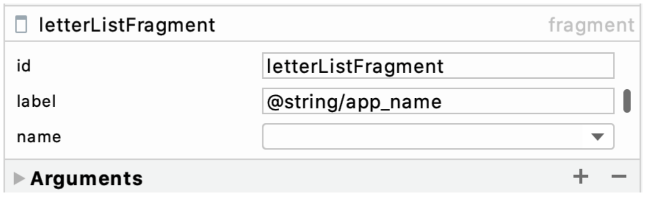


Fragments have a property called `"label"` where you can set the title which the parent activity will know to use in the app bar.

1. In `strings.xml`, after the app name, add the following constant.

```
<string name="word_list_fragment_label">Words That Start With {letter}</string>
```

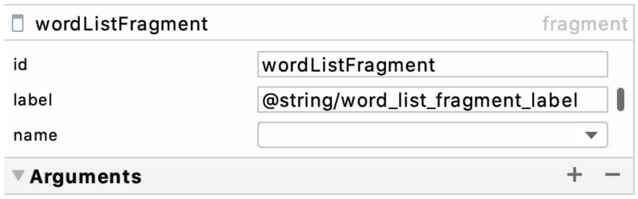
2. You can set the label for each fragment on the navigation graph. Go back into nav_graph.xml and select letterListFragment in the component tree, and in the attributes pane, set the label to the app_name string



letterListFragment — fragment

id	letterListFragment
label	@string/app_name
name	

3. Select wordListFragment and set the label to word_list_fragment_label



wordListFragment — fragment

id	wordListFragment
label	@string/word_list_fragment_label
name	

Congratulations on making it this far! Run your app one more time and you should see everything just as it was at the start of the codelab, only now, all your navigation is hosted in a single activity with a separate fragment for each screen.

12. Summary

- Fragments are reusable pieces of UI that can be embedded in activities.
- The lifecycle of a fragment differs from the lifecycle of an activity, with view setup occurring in onViewCreated(), rather than onCreateView().
- A FragmentContainerView is used to embed fragments in other activities and can manage navigation between fragments.

Using the Navigation Component

- Setting the navGraph attribute of a FragmentContainerView allows you to navigate between fragments within an activity.
- The NavGraph editor allows you to add navigation actions and specify arguments between different destinations.
- While navigating using intents requires you to pass in extras, the Navigation component uses SafeArgs to auto-generate classes and methods for your navigation actions, ensuring type safety with arguments.

Use cases for fragments.

- Using the Navigation component, many apps can manage their entire layout within a single activity, with all navigation occurring between fragments.
- Fragments make common layout patterns possible, such as master-detail layouts on tablets, or multiple tabs within the same activity.