Android Kotlin Fundamentals: Create a fragment

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

In this codelab, you learn about fragments, which represent a behavior or a portion of the user interface in an activity. You will create a fragment inside a fun starter app called AndroidTrivia. In the next codelab, you learn more about navigation and do further work on the AndroidTrivia app.

What you should already know

- The fundamentals of Kotlin
- How to create basic Android apps in Kotlin
- How to work with layouts

What you'll learn

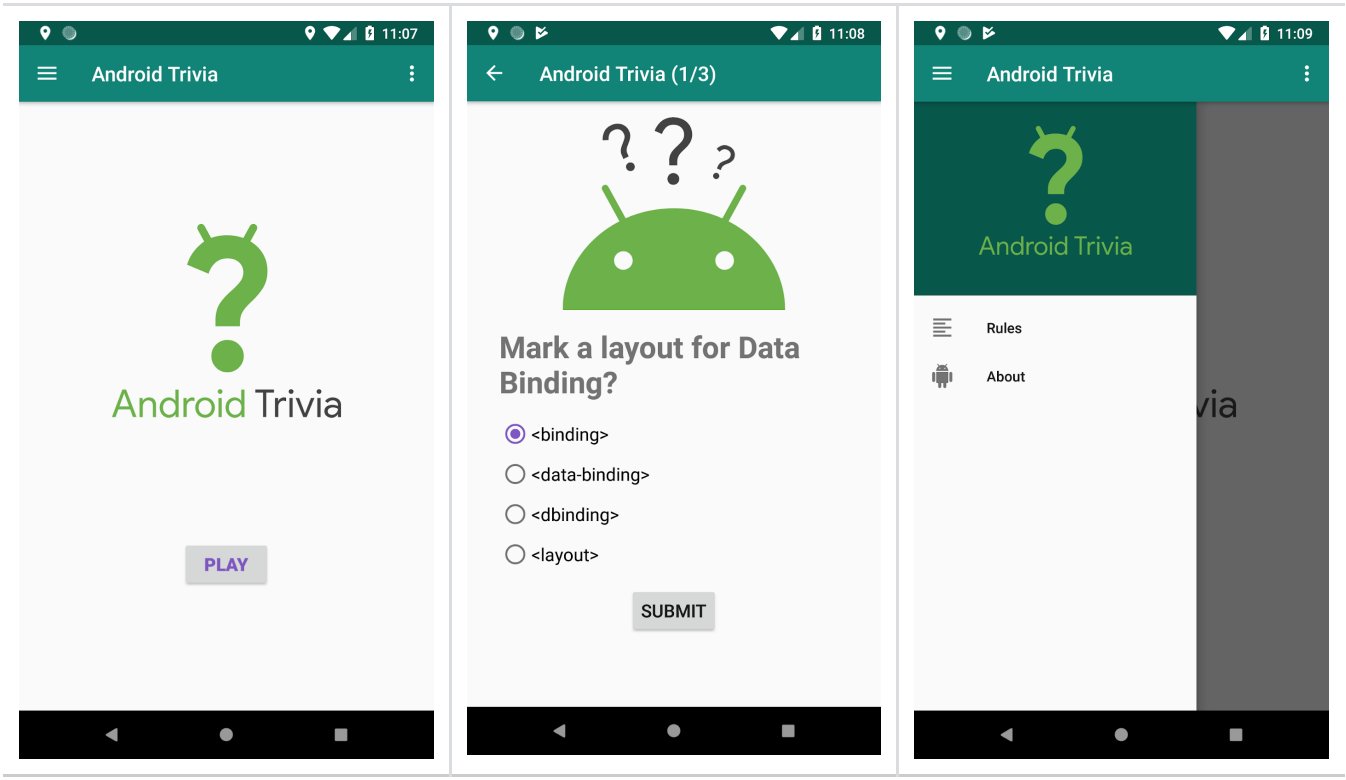
How to add a Fragment statically to your app

What you'll do

- Create a Fragment inside an activity.

2. App overview

In the three codelabs that make up this lesson, you work on an app called AndroidTrivia. The completed app is a game in which the user answers three trivia questions about Android coding. If the user answers all three questions correctly, they win the game and can share their results.



The AndroidTrivia app illustrates navigation patterns and controls. The app has several components:

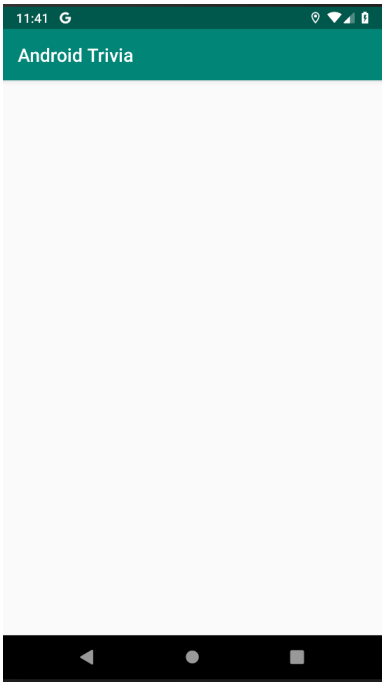
- In the title screen, shown on the left in the screenshot above, the user starts the game.
- In the game screen with questions, shown in the middle above, the user plays the game and submits their answers.
- The navigation drawer, shown on the right above, slides out from the side of the app and contains a menu with a header. The drawer icon  opens the navigation drawer. The navigation-drawer menu contains a link to the About page and a link to the rules of the game.

The top of the app displays a view called the *app bar (*or *action bar)* which shows the app's name.

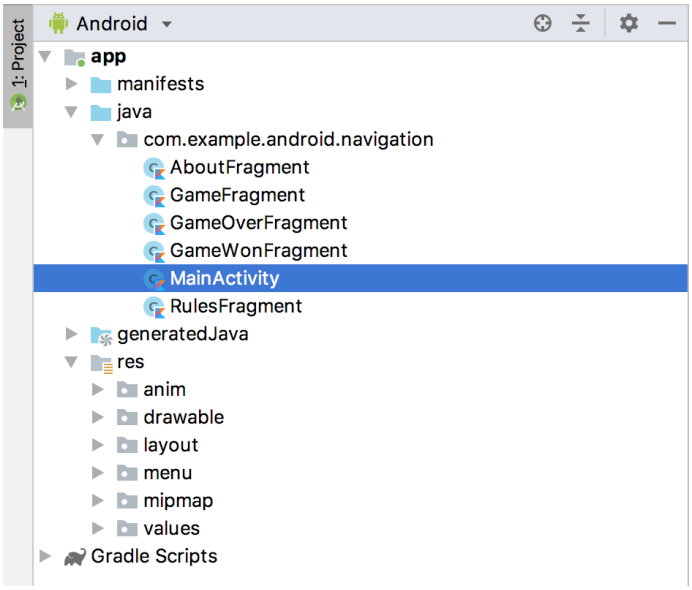
3. Task: Explore the starter app project

In this codelab, you work from a starter app that provides template code and Fragment classes that you need as you complete the Trivia app.

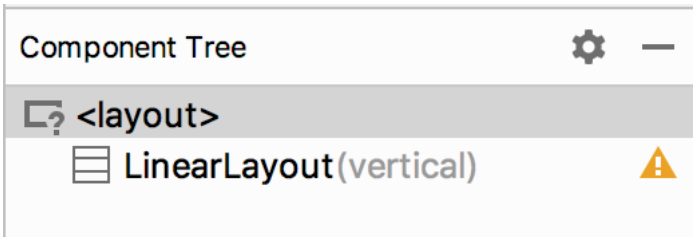
1. Download the AndroidTrivia-StarterAndroid Studio project. You may have to download the entire android-kotlin-fundamentals-starter-apps zip file.
2. Open the project in Android Studio and run the app. When the app opens, it doesn't do anything other than display the app name and a blank screen.



3. In the Android Studio Project pane, open the Project: Android view to explore the project files. Open the **app > java** folder to see the `MainActivity` class and Fragment classes.



4. Open the **res > layout** folder and double-click **activity_main.xml**. The `activity_main.xml` file appears in the Layout Editor. 5. Open the **Design** tab. The **Component Tree** for the `activity_main.xml` file shows the root layout as vertical `LinearLayout` .



In a vertical linear layout, all the child views in the layout are aligned vertically.

4. Task: Add a fragment

A `Fragment` represents a behavior or a portion of user interface (UI) in an `Activity`. You can combine multiple fragments in a single activity to build a multi-pane UI, and you can reuse a `Fragment` in multiple activities.

Think of a `Fragment` as a modular section of an activity, something like a "sub-activity" that you can also use in other activities:

- A `Fragment` has its own lifecycle and receives its own input events.
- You can add or remove a `Fragment` while the activity is running.
- A `Fragment` is defined in a Kotlin class.
- A Fragment's UI is defined in an XML layout file.

The AndroidTrivia app has a main activity and several fragments. Most of the fragments and their layout files have been defined for you. In this task, you create a fragment and add the fragment to the app's main activity.

Step 1: Add a Fragment class

In this step, you create a blank `TitleFragment` class. Start by creating a Kotlin class for a new Fragment:

1. In Android Studio, click anywhere inside the Project pane to bring the focus back to the project files. For example, click the **com.example.android.navigation** folder.
2. Select **File > New > Fragment > Fragment (Blank)**.
3. For the Fragment name, enter **TitleFragment**.
4. For the Fragment layout name, enter **placeholder_layout** (we will not use this layout for our app as it already has the layout designed for TitleFragment).
5. For source language, select **Kotlin**.
6. Click **Finish**.
7. Open the `TitleFragment.kt` fragment file, if it is not already open. It contains the `onCreateView()` method, which is one of the methods that's called during a Fragment's lifecycle.
8. Delete the code inside `onCreateView()`. The `onCreateView()` function is left with only the following code:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
}
```

9. In `TitleFragment` class, delete the `onCreate()` method, the fragment initialization parameters and companion object. Make sure your `TitleFragment` class looks like the following:

```
class TitleFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstan

    }
}
```

Create a binding object

The Fragment won't compile now. To make the Fragment compile, you need to create a binding object and inflate the Fragment's view (which is equivalent to using `setContentView()` for an Activity).

1. In the `onCreateView()` method in `TitleFragment.kt`, create a `binding` variable (`val binding`).

2. To inflate the Fragment's view, call the `DataBindingUtil.inflate()` method on the Fragment's `Binding` object, which is `FragmentTitleBinding`.

Pass four arguments into the `DataBindingUtil.inflate` method:

- `inflater`, which is the `LayoutInflater` used to inflate the binding layout.
- The XML layout resource of the layout to inflate. Use one of the layouts that is already defined for you, `R.layout.fragment_title`.
- `container` for the parent `ViewGroup`. (This parameter is optional.)
- `false` for the `attachToParent` value.

3. Assign the binding that `DataBindingUtil.inflate` returns to the `binding` variable.
4. Return `binding.root` from the method, which contains the inflated view. Your `onCreateView()` method now looks like the following code:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val binding = DataBindingUtil.inflate<FragmentTitleBinding>(inflater,
            R.layout.fragment_title,container,false)
    return binding.root
    }
```

5. Open **res>layout** and delete `placeholder_layout.xml`.

Step 2: Add the new fragment to the main layout file

In this step, you add the `TitleFragment` to the app's `activity_main.xml` layout file.

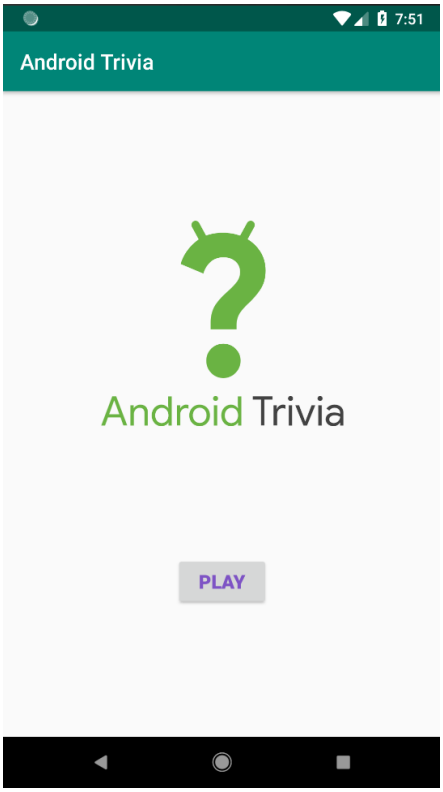
1. Open **res > layout > activity_main.xml** and select the **Code** tab to view the layout XML code.
2. Inside the existing `LinearLayout` element, add a `fragment` element.
3. Set the fragment's ID to `titleFragment`.
4. Set the fragment's name to the full path of the Fragment class, which in this case is `com.example.android.navigation.TitleFragment`.
5. Set the layout width and height to `match_parent`.

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

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical">
            <fragment
                android:id="@+id/titleFragment"
                android:name="com.example.android.navigation.TitleFragment"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                />
        </LinearLayout>

</layout>
```

6. Run the app. The Fragment has been added to your main screen.


7:51
Android Trivia
Android Trivia
PLAY

Android Kotlin Fundamentals: 03.2 Define navigation paths

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

In the previous codelab, you modified the AndroidTrivia app to add a Fragment to an existing activity. In this codelab, you add navigation to that app.

Introduction to navigation

Structuring the user's experience of navigating through an app has always been an interesting topic for developers. For Android apps, the Navigation Architecture Component makes it easier to implement navigation.

A *destination* is any place inside the app to which a user can navigate. A *navigation graph* for an app consists of a set of destinations within the app. Navigation graphs allow you to visually define and customize how users navigate among destinations in your app.

What you should already know

- The fundamentals of Kotlin
- How to create basic Android apps in Kotlin
- How to work with layouts

What you'll learn

How to use navigation graphs

How to define navigation paths in your app

What an Up button is, and how to add one

How to create an options menu

How to create a navigation drawer

What you'll do

- Create a navigation graph for your fragments using the navigation library and the Navigation Editor.
- Create navigation paths in your app.
- Add navigation using the options menu.
- Implement an Up button so that the user can navigate back to the title screen from anywhere in the app.
- Add a navigation drawer menu.

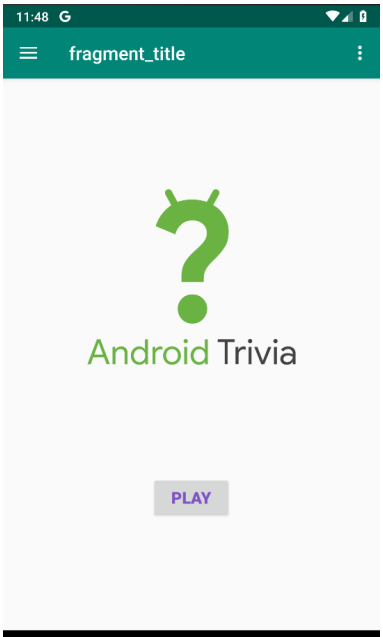
2. App overview

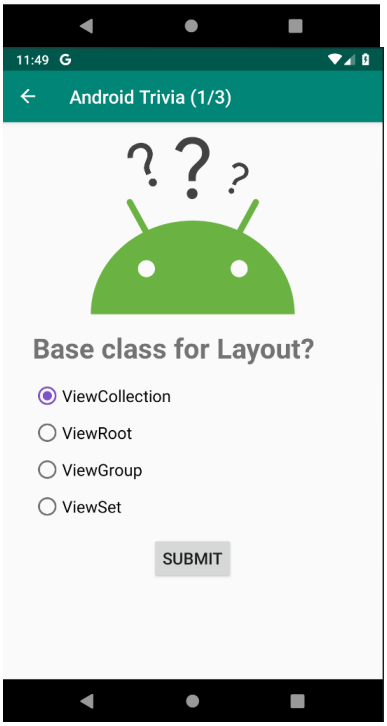
The AndroidTrivia app, which you started working on in the previous codelab, is a game in which users answer questions about Android development. If the user answers three questions correctly, they win the game.

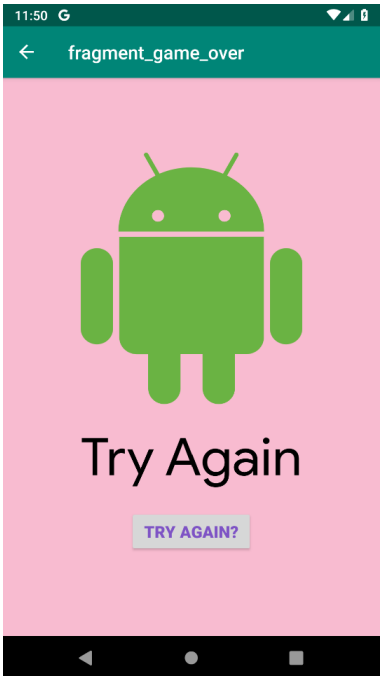
If you completed the previous codelab, use that code as the starter code for this codelab. Otherwise, download the AndroidTriviaFragmentapp from GitHub to get the starter code.

In this codelab, you update the AndroidTrivia app in the following ways:

- You create a navigation graph for the app.
- You add navigation for a title screen and a game screen.
- You connect the screens with an action, and you give the user a way to navigate to the game screen by tapping **Play**.
- You add an Up button, which is shown as the left-arrow at the top of some screens.






11:50
fragment_game_over
Try Again
TRY AGAIN?

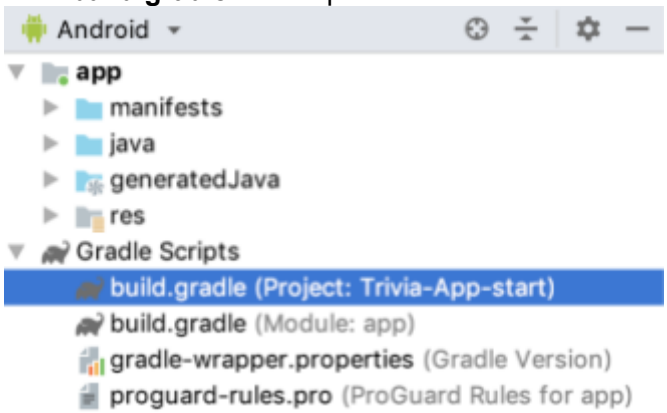
3. Task: Add navigation components to the project

Step 1: Add navigation dependencies

The Navigation component is a library that can manage complex navigation, transition animation, deep linking, and compile-time checked argument passing between the screens in your app.

To use the navigation library, you need to add the navigation dependencies to your Gradle files.

1. To get started, download the AndroidTriviaFragmentstarter app or use your own copy of the AndroidTrivia app from the previous codelab. Open the AndroidTrivia app in Android Studio.
2. In the Project: Android pane, open the **Gradle Scripts** folder. Double-click the project-level **build.gradle** file to open the file.



3. At the top of the project-level `build.gradle` file, along with the other `ext` variables, add a variable for the `navigationVersion`. To find the latest navigation version number, see Declaring dependenciesin the Android developer documentation.

```
ext {
        ...
        navigationVersion = "2.3.0"
        ...
    }
```

4. In the **Gradle Scripts** folder, open the module-level **build.gradle** file. Add the dependencies for `navigation-fragment-ktx` and `navigation-ui-ktx`, as shown below:

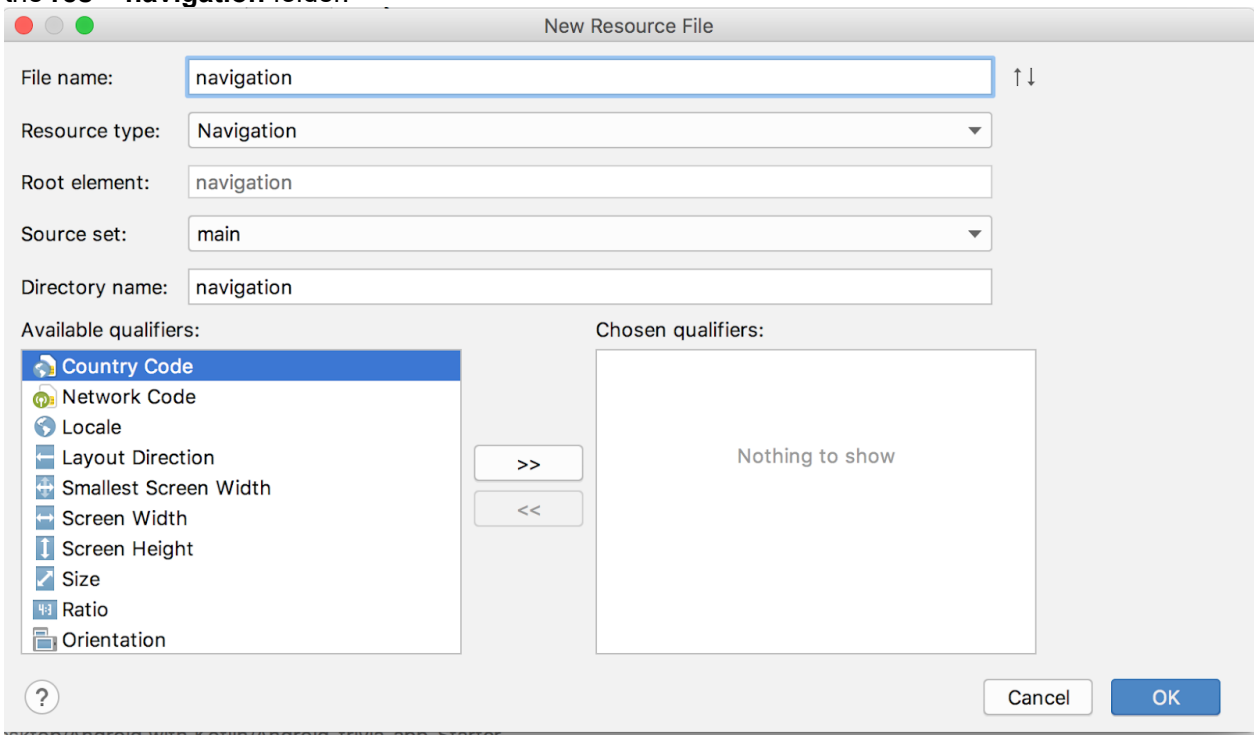
```
dependencies {
  ...
  implementation "androidx.navigation:navigation-fragment-ktx:$navigationVersion"
  implementation "androidx.navigation:navigation-ui-ktx:$navigationVersion"
  ...
}
```

5. Rebuild the project.

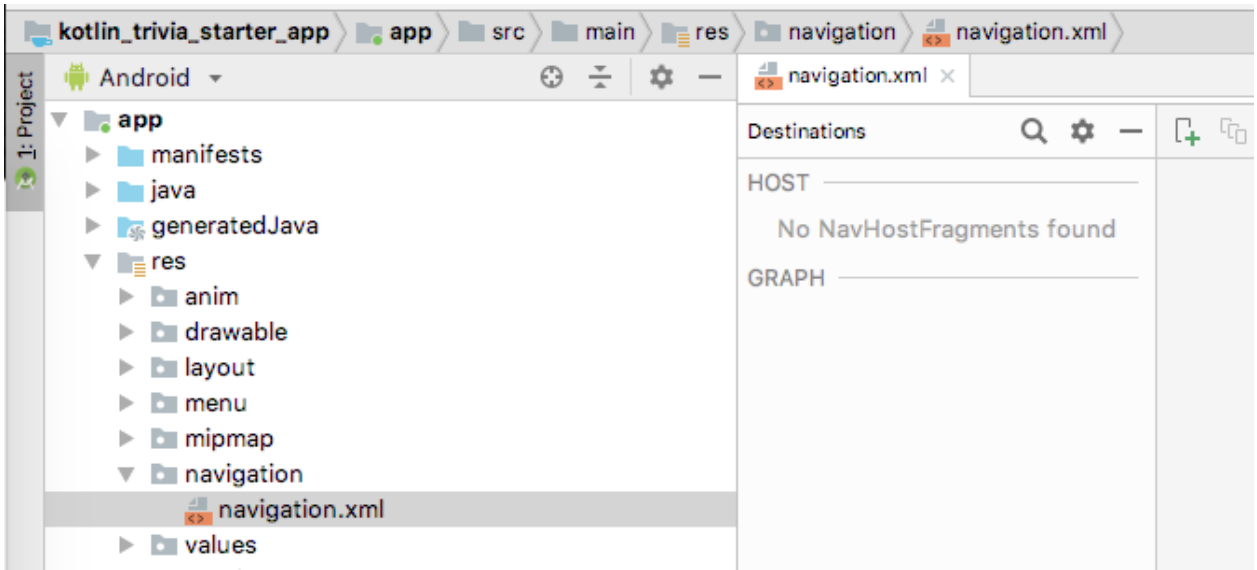
> **Note**: Android Studio may alert you to new versions of your dependencies. You may have to update the version-number variables using the **Project Structure** dialog, ie, **File->Project Structure**and update the number to obtain the latest version.

Step 2: Add a navigation graph to the project

1. In the Project: Android pane, right-click the **res** folder and select **New > Android Resource File**.
2. In the **New Resource File** dialog, select **Navigation** as the **Resource type**.
3. In the **File name** field, name the file `navigation`.
4. Make sure the **Chosen qualifiers** box is empty, and click **OK**. A new file, **navigation.xml**, appears in the **res > navigation** folder.



5. Open the **res > navigation > navigation.xml** file and click the **Design** tab to open the Navigation Editor. Notice the **No NavHostFragments found** message in the layout editor. You fix this problem in the next task.



4. Task: Create the NavHostFragment

A *navigation host fragment* acts as a host for the fragments in a navigation graph. The navigation host Fragment is usually named `NavHostFragment`.

As the user moves between destinations defined in the navigation graph, the navigation host Fragment swaps fragments in and out as necessary. The Fragment also creates and manages the appropriate Fragment back stack.

In this task, you modify your code to replace the `TitleFragment` with the `NavHostFragment`.

1. Open **res > layout > activity_main.xml** and open the **Code** tab.
2. In the `activity_main.xml` file, change the name of the existing title Fragment to `androidx.navigation.fragment.NavHostFragment`.
3. Change the ID to `myNavHostFragment`.
4. The navigation host Fragment needs to know which navigation graph resource to use. Add the `app:navGraph` attribute and set it to the navigation graph resource, which is `@navigation/navigation`.
5. Add the `app:defaultNavHost` attribute and set it to `"true"`. Now this navigation host is the default host and will intercept the system Back button.

Inside the `activity_main.xml` layout file, your `fragment` now looks like the following:

```
<!-- The NavHostFragment within the activity_main layout -->
            <fragment
                android:id="@+id/myNavHostFragment"
                android:name="androidx.navigation.fragment.NavHostFragment"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                app:navGraph="@navigation/navigation"
                app:defaultNavHost="true" />
```

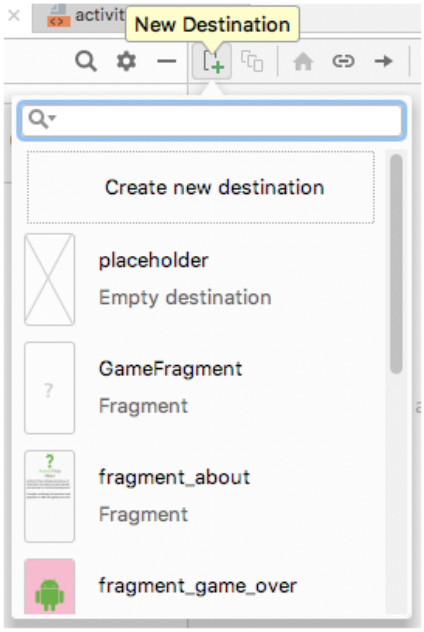
5. Task: Add fragments to the navigation graph

In this task, you add the title Fragment and the game Fragment to the app's navigation graph. You connect the fragments to each other. Then you add a click handler to the **Play** button so that the user can navigate from the title screen to the game screen.

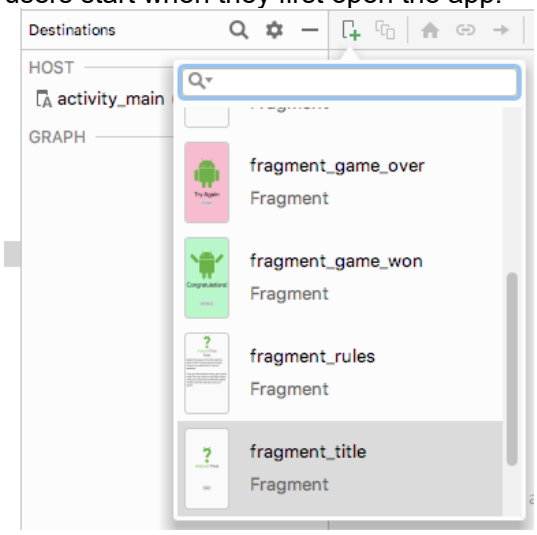
Step 1: Add two fragments to the navigation graph and connect them with an action

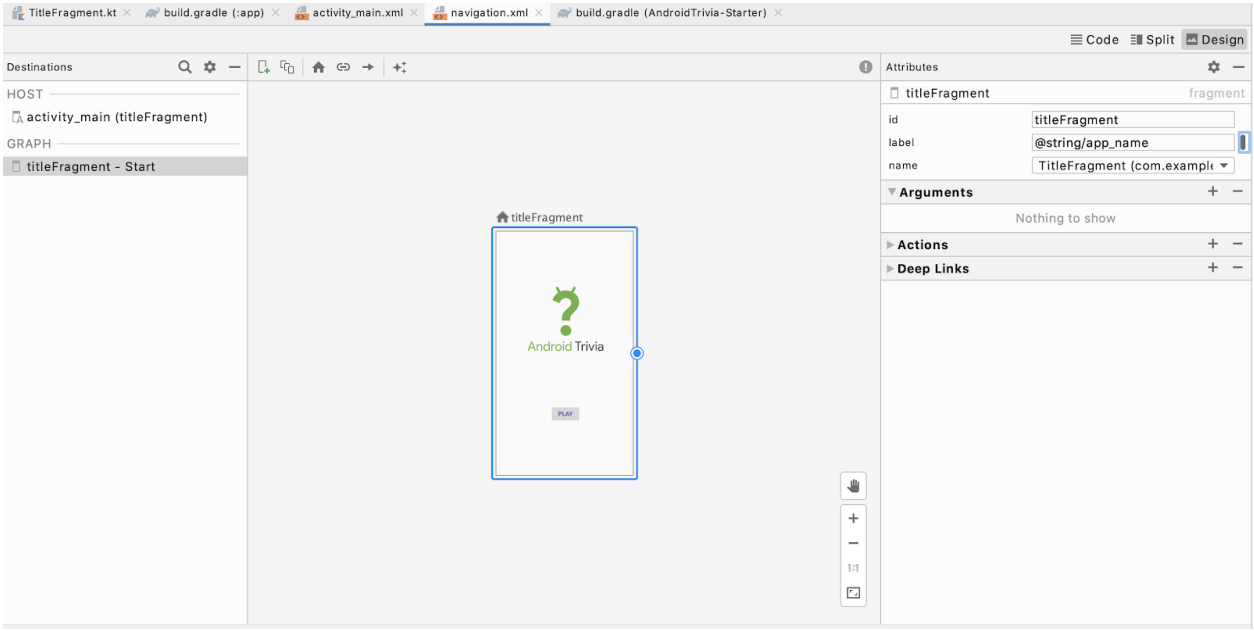
1. Open **navigation.xml** from the **navigation** resource folder. In the Navigation Editor, click the **New Destination** button

. A list of fragments and activities appears.



2. Select **fragment_title**. You add **fragment_title** first because the `TitleFragment` Fragment is where app users start when they first open the app.



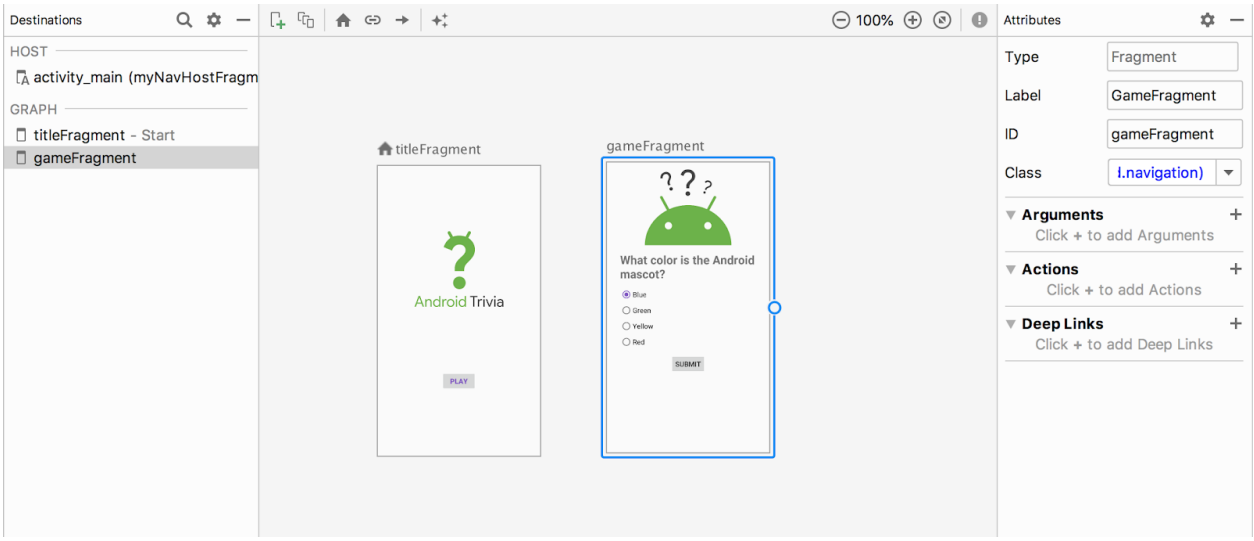


3. Use the **New Destination** button to add the **GameFragment**.

If the preview shows a "Preview Unavailable" message, click the **Code** tab to open the navigation XML. Make sure that the `fragment` element for the `gameFragment` includes `tools:layout="@layout/fragment_game"`, as shown below.

```
<!-- The game fragment within the navigation XML, complete with tools:layout. -->
<fragment
    android:id="@+id/gameFragment"
    android:name="com.example.android.navigation.GameFragment"
    android:label="GameFragment"
    tools:layout="@layout/fragment_game" />
```

4. In the layout editor (using the **Design** view), drag the game Fragment to the right so it doesn't overlap with the title Fragment.



5. In the preview, hold the pointer over the title Fragment. A circular connection point appears on the right side of the Fragment view. Click the connection point and drag it to the `gameFragment` or drag it to anywhere in the `gameFragment` preview. An *Action* is created that connects the two fragments.
6. To see the Action's attributes, click the arrow that connects the two fragments. In the **Attributes**pane, check that the action's ID is set to `action_titleFragment_to_gameFragment`.

Step 2: Add a click handler to the Play button

The title Fragment is connected to the game Fragment by an action. Now you want the **Play** button on the title screen to navigate the user to the game screen.

1. In Android Studio, open the `TitleFragment.kt` file. Inside the `onCreateView()` method, add the following code before the `return` statement:

```
binding.playButton.setOnClickListener{}
```

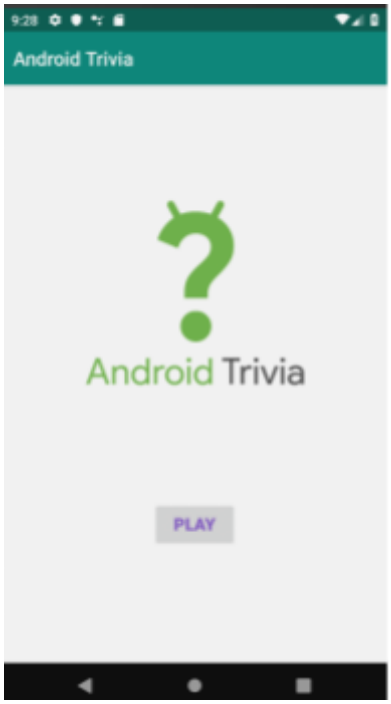
2. Inside `setOnClickListener`, add code to access the **Play** button through the binding class and navigate to the game fragment:

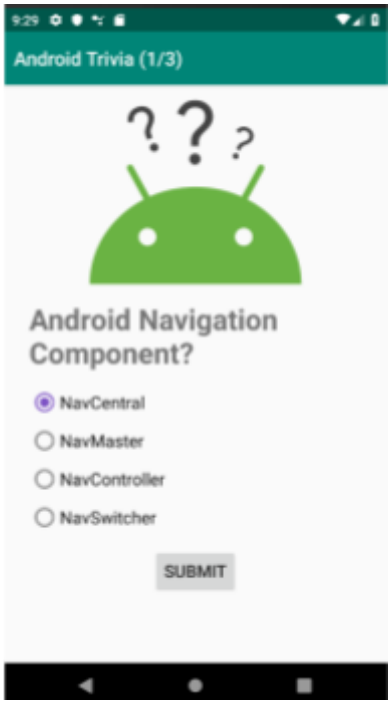
```
//The complete onClickListener with Navigation
binding.playButton.setOnClickListener { view : View ->
        view.findNavController().navigate(R.id.action_titleFragment_to_gameFragment)
}
```

3. Build the app and make sure that it has all the imports it needs. For example, you might need to add the following line to the `TitleFragment.kt` file:

```
import androidx.navigation.findNavController
```

4. Run the app and tap the **Play** button on the title screen. The game screen opens.


Android Trivia
Android Trivia
PLAY


Android Trivia (1/3)
Android Navigation Component?
NavCentral
NavMaster
NavController
NavSwitcher
SUBMIT

6. Task: Add conditional navigation

In this step, you add *conditional navigation*, which is navigation that's only available to the user in certain contexts. A common use case for conditional navigation is when an app has a different flow, depending on whether the user is logged in.

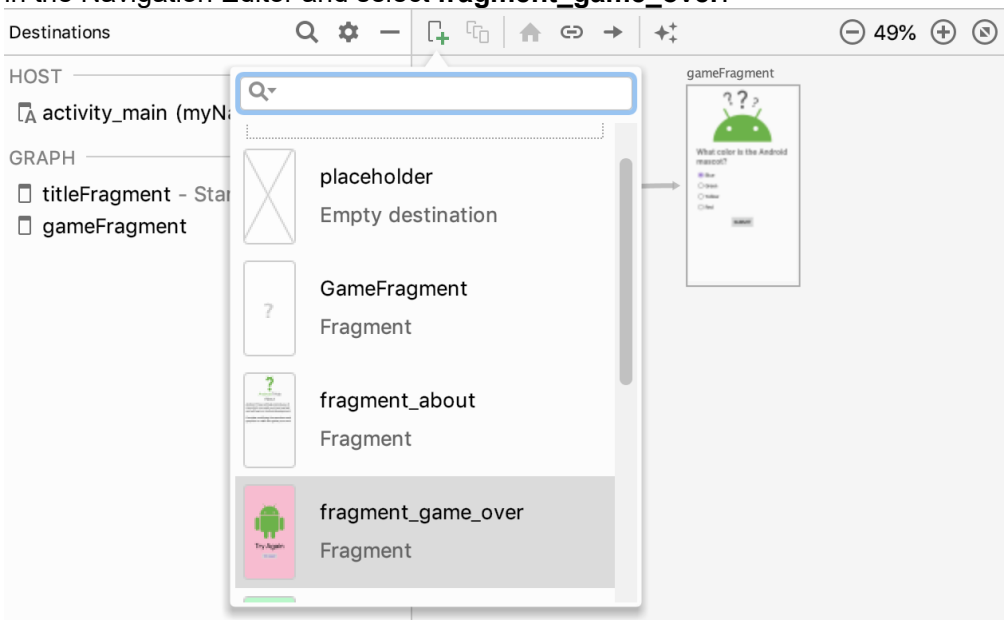
Your app is a different case: Your app will navigate to a different Fragment, based on whether the user answers all the questions correctly.

The starter code contains two fragments for you to use in your conditional navigation:

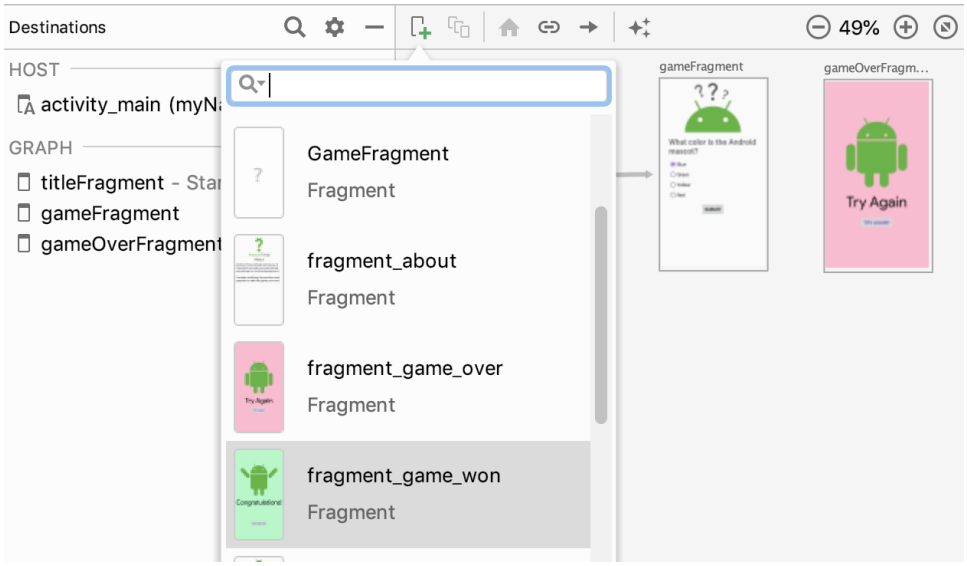
- The `GameWonFragment` takes the user to a screen that shows a "Congratulations!" message.
- The `GameOverFragment` takes the user to a screen that shows a "Try Again" message.

Step 1: Add GameWonFragment and GameOverFragment to the navigation graph

1. Open the `navigation.xml` file, which is in the `navigation` folder.
2. To add the game-over Fragment to the navigation graph, click the **New Destination** button  in the Navigation Editor and select **fragment_game_over**.

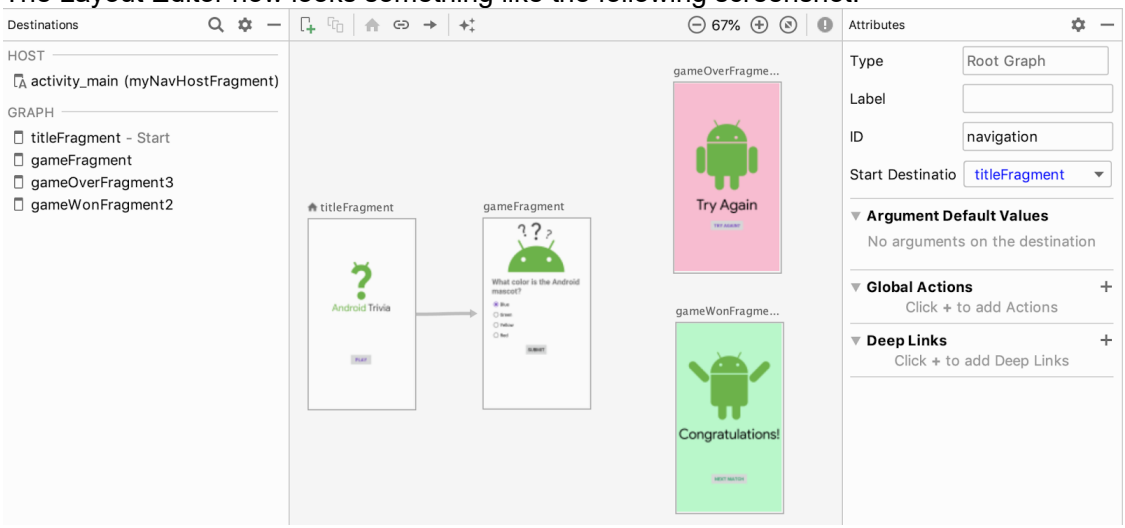


3. In the preview area of the layout editor, drag the game-over Fragment to the right of the game Fragment so the two don't overlap. Make sure to change the **ID** attribute of the game-over Fragment, to `gameOverFragment`.
4. To add the game-won Fragment to the navigation graph, click the **New Destination** button  and select **fragment_game_won**.



5. Drag the game-won Fragment below the game-over Fragment so the two don't overlap. Make sure to name the **ID** attribute of the game-won Fragment as `gameWonFragment`.

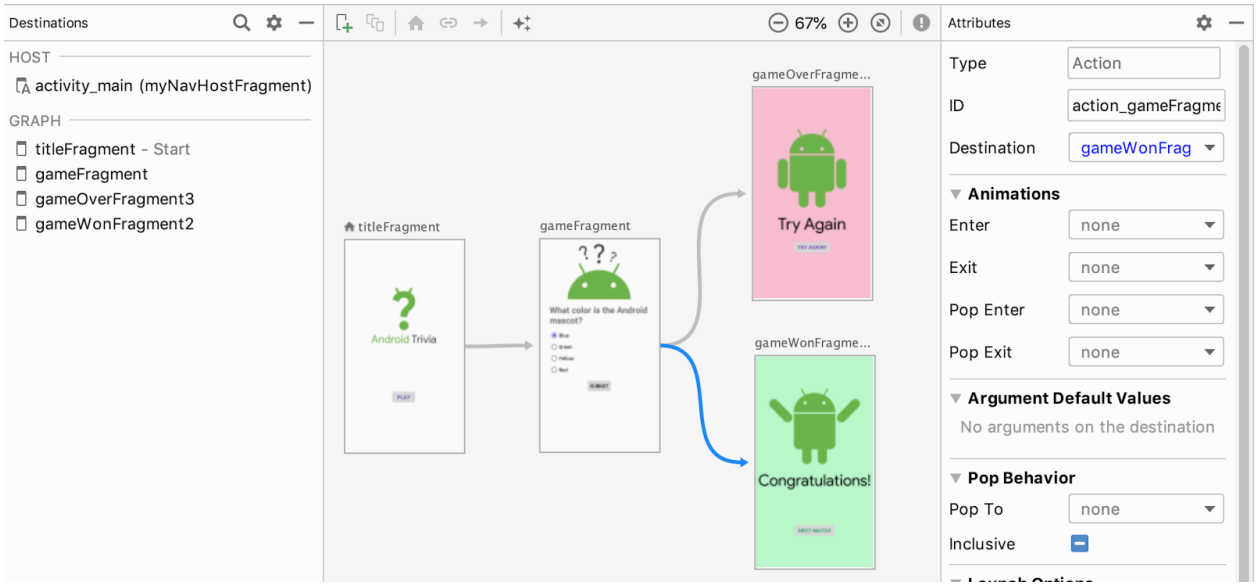
The Layout Editor now looks something like the following screenshot:



Step 2: Connect the game Fragment to the game-result Fragment

In this step, you connect the game Fragment to both the game-won Fragment and the game-over Fragment.

1. In the preview area of the Layout Editor, hold the pointer over the game Fragment until the circular connection point appears.
2. Click the connection point and drag it to the game-over Fragment. A blue connection arrow appears, representing an Action that now connects the game Fragment to the game-over Fragment.
3. In the same way, create an action that connects the game Fragment to the game-won Fragment. The Layout Editor now looks something like the following screenshot:



4. In the preview, hold the pointer over the line that connects the game Fragment to the game-won Fragment. Notice that the ID for the Action has been assigned automatically.

Step 3: Add code to navigate from one Fragment to the next

`GameFragment` is a Fragment class that contains questions and answers for the game. The class also includes logic that determines whether the user wins or loses the game. You need to add conditional navigation in the `GameFragment` class, depending on whether the player wins or loses.

1. Open the `GameFragment.kt` file. The `onCreateView()` method defines an `if` / `else` condition that determines whether the player has won or lost:

```
binding.submitButton.setOnClickListener @Suppress("UNUSED_ANONYMOUS_PARAMETER")
        {
              ...
                // answer matches, we have the correct answer.
                if (answers[answerIndex] == currentQuestion.answers[0]) {
                    questionIndex++
                    // Advance to the next question
                    if (questionIndex < numQuestions) {
                        currentQuestion = questions[questionIndex]
                        setQuestion()
                        binding.invalidateAll()
                    } else {
                        // We've won!  Navigate to the gameWonFragment.
                    }
                } else {
                    // Game over! A wrong answer sends us to the gameOverFragment.
                }
            }
        }
```

2. Inside the `else` condition for winning the game, add the following code, which navigates to the `gameWonFragment`. Make sure that the Action name (`action_gameFragment_to_gameWonFragment` in this example) exactly matches what's set in the `navigation.xml` file.

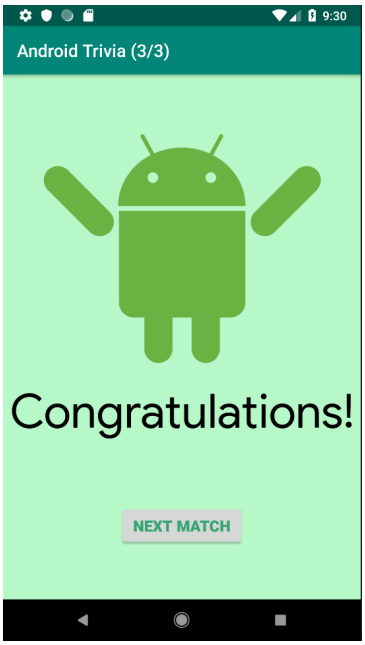
```
// We've won!  Navigate to the gameWonFragment.
view.findNavController()
```

```
    .navigate(R.id.action_gameFragment_to_gameWonFragment)
```

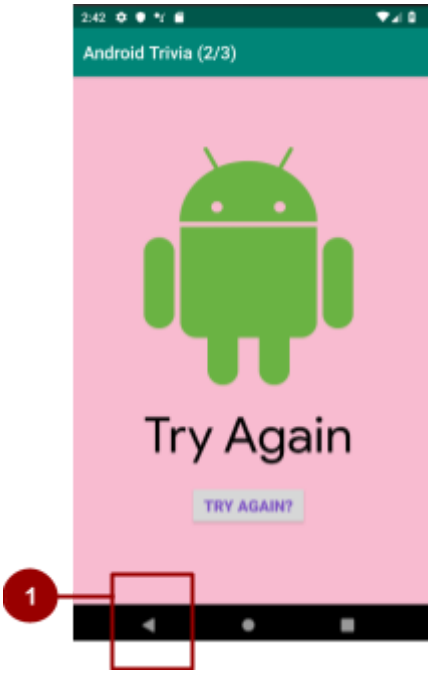
3. Inside the `else` condition for losing the game, add the following code, which navigates to the `gameOverFragment`:

```
// Game over! A wrong answer sends us to the gameOverFragment.
view.findNavController().
    navigate(R.id.action_gameFragment_to_gameOverFragment)
```

4. Run the app and play the game by answering the questions. If you answer all three questions correctly, the app navigates to the `GameWonFragment`.



If you get any question wrong, the app immediately navigates to the `GameOverFragment`.

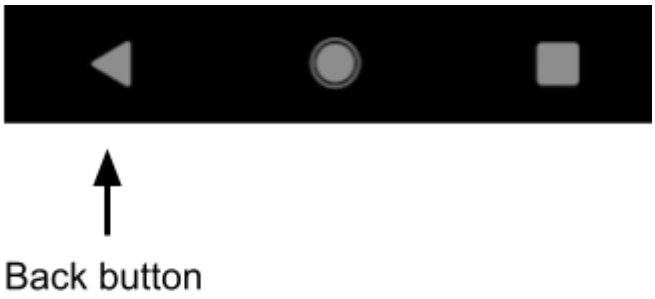


The Android system's Back button is shown as 1 in the screenshot above. If the user presses the Back button in the game-won fragment or the game-over Fragment, the app navigates to the question screen. Ideally, the Back button should navigate back to the app's title screen. You change the destination for the Back button in the next task.

7. Task: Change the Back button's destination

The Android system keeps track of where users navigate on an Android-powered device. Each time the user goes to a new destination on the device, Android adds that destination to the *back stack*.

When the user presses the Back button, the app goes to the destination that's at the top of the back stack. By default, the top of the back stack is the screen that the user last viewed. The Back button is typically the left-most button at the bottom of the screen, as shown below. (The Back button's exact appearance is different on different devices.)



Until now, you've let the navigation controller handle the back stack for you. When the user navigates to a destination in your app, Android adds this destination to the back stack.

In the AndroidTrivia app, when the user presses the Back button from the `GameOverFragment` or `GameWonFragment` screen, they end up back in the `GameFragment`. But you don't want to send the user to the `GameFragment`, because the game is over. The user could restart the game, but a better experience would be to find themselves back at the title screen.

A navigation action can modify the back stack. In this task, you change the action that navigates from the game fragment so that the action removes the `GameFragment` from the back stack. When the user wins or loses the game, if they press the Back button, the app skips the `GameFragment` and goes back to the `TitleFragment`.

Step 1: Set the pop behavior for the navigation actions

In this step, you manage the back stack so that when the user is at the `GameWon` or `GameOver` screen, pressing the Back button returns them to the title screen. You manage the back stack by setting the "pop" behavior for the actions that connect the fragments:

- The `popUpTo` attribute of an action "pops up" the back stack to a given destination before navigating. (Destinations are removed from the back stack.)
- If the `popUpToInclusive` attribute is `false` or is not set, `popUpTo` removes destinations up to the specified destination, but leaves the specified destination in the back stack.
- If `popUpToInclusive` is set to `true`, the `popUpTo` attribute removes all destinations up to and *including* the given destination from the back stack.
- If `popUpToInclusive` is `true` and `popUpTo` is set to the app's starting location, the action removes *all* app destinations from the back stack. The Back button takes the user all the way out of the app.

In this step, you set the `popUpTo` attribute for the two actions that you created in the previous task. You do this using the **Pop To** field in the **Attributes** pane of the Layout Editor.

1. Open `navigation.xml` in the **res > navigation** folder. If the navigation graph does not appear in the layout editor, click the **Design** tab.
2. Select the action for navigating from the `gameFragment` to the `gameOverFragment`. (In the preview area, the action is represented by a blue line that connects the two fragments.)
3. In the **Attributes** pane, set **popUpTo** to `gameFragment`. Select the **popUpToInclusive** checkbox.

This sets the `popUpTo` and `popUpToInclusive` attributes in the XML. The attributes tell the navigation component to remove fragments from the back stack up to and including `GameFragment`. (This has the same effect as setting the **popUpTo** field to `titleFragment` and clearing the **popUpToInclusive**checkbox.)

4. Select the action for navigating from the `gameFragment` to the `gameWonFragment`. Again, set **popUpTo** to `gameFragment` in the **Attributes** pane and select the **popUpToInclusive** checkbox.

5. Run the app and play the game, then press the Back button. Whether you win or lose, the Back button takes you back to the `TitleFragment`.

Step 2: Add more navigation actions and add onClick handlers

Your app currently has the following user flow:

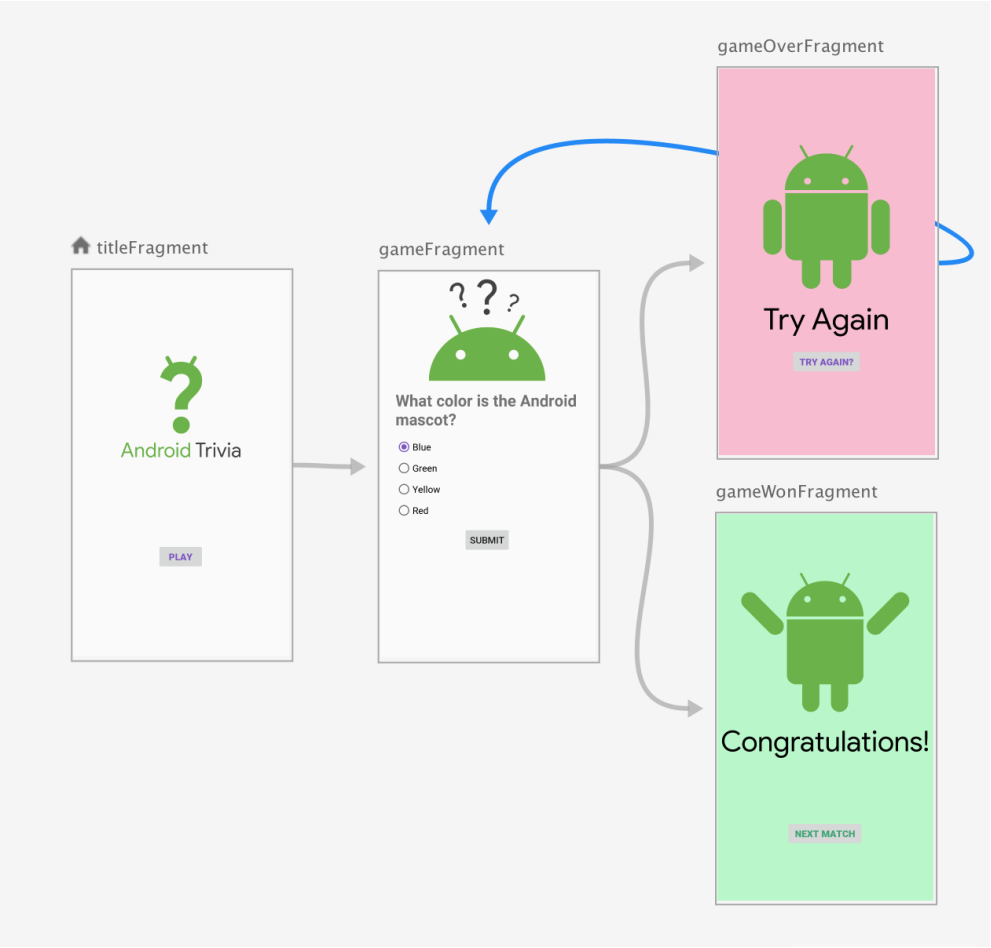
- The user plays the game and wins or loses, and the app navigates to the `GameWon` or `GameOver` screen.
- If the user presses the system Back button at this point, the app navigates to the `TitleFragment`. (You implemented this behavior in Step 1 of this task, above.)

In this step you implement two more steps of user flow:

- If the user taps the **Next Match** or **Try Again** button, the app navigates to the `GameFragment` screen.
- If the user presses the system Back button at this point, the app navigates to the `TitleFragment` screen (instead of back to the `GameWon` or `GameOver` screen).

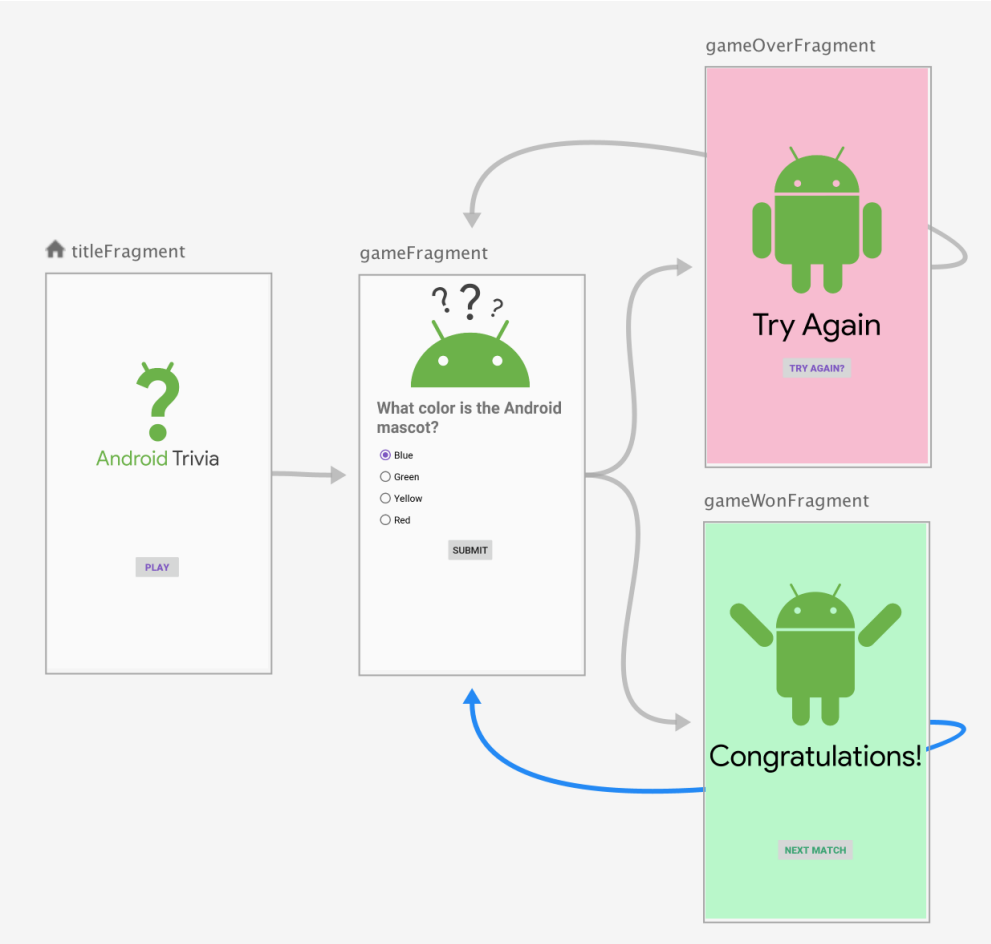
To create this user flow, use the `popUpTo` attribute to manage the back stack:

1. Inside the `navigation.xml` file, add a navigation action connecting `gameOverFragment` to `gameFragment`. Make sure that the Fragment names in the action's ID match the Fragment names that are in the XML. For example, the action ID might be `action_gameOverFragment_to_gameFragment` **.**



2. In the **Attributes** pane, set the action's **popUpTo** attribute to `titleFragment`. 3. Clear the **popUpToInclusive** checkbox, because you do not want the `titleFragment` to be included in the destinations that are removed from the back stack. Instead, you want everything up to the `TitleFragment` (but not including it) to be removed from the back stack.

4. Inside the `navigation.xml` file, add a navigation action connecting `gameWonFragment` to `gameFragment`.



5. For the action you just created, set the **popUpTo** attribute to `titleFragment` and clear the **popUpToInclusive** checkbox.

Now add functionality to the **Try Again** and **Next Match** buttons. When the user taps either button, you want the app to navigate to the `GameFragment` screen so that the user can try the game again.

1. Open the `GameOverFragment.kt` Kotlin file. At the end of the `onCreateView()` method, before the `return` statement, add the following code. The code adds a click listener to the **Try Again** button. When the user taps the button, the app navigates to the game Fragment.

```
// Add OnClick Handler for Try Again button
        binding.tryAgainButton.setOnClickListener{view: View->
        view.findNavController()
                .navigate(R.id.action_gameOverFragment_to_gameFragment)}
```

2. Open the `GameWonFragment.kt` Kotlin file. At the end of the `onCreateView()` method, before the `return` statement, add the following code:

```
// Add OnClick Handler for Next Match button
        binding.nextMatchButton.setOnClickListener{view: View->
            view.findNavController()
                    .navigate(R.id.action_gameWonFragment_to_gameFragment)}
```

3. Run your app, play the game, and test the **Next Match** and **Try Again** buttons. Both buttons should take you back to the game screen so that you can try the game again.

4. After you win or lose the game, tap **Next Match** or **Try Again**. Now press the system Back button. The app should navigate to the title screen, instead of going back to the screen that you just came from.

8. Task: Add an Up button in the app bar

The app bar

The *app bar*, sometimes called the *action bar*, is a dedicated space for app branding and identity. For example, you can set the app bar's color. The app bar gives the user access to familiar navigation features such as an options menu. To access the options menu from the app bar, the user taps the icon with the three vertical dots

.

The app bar displays a title string that can change with each screen. For the title screen of the AndroidTrivia app, the app bar displays "Android Trivia." On the question screen, the title string also shows which question the user is on ("1/3," "2/3," or "3/3.")

The Up button

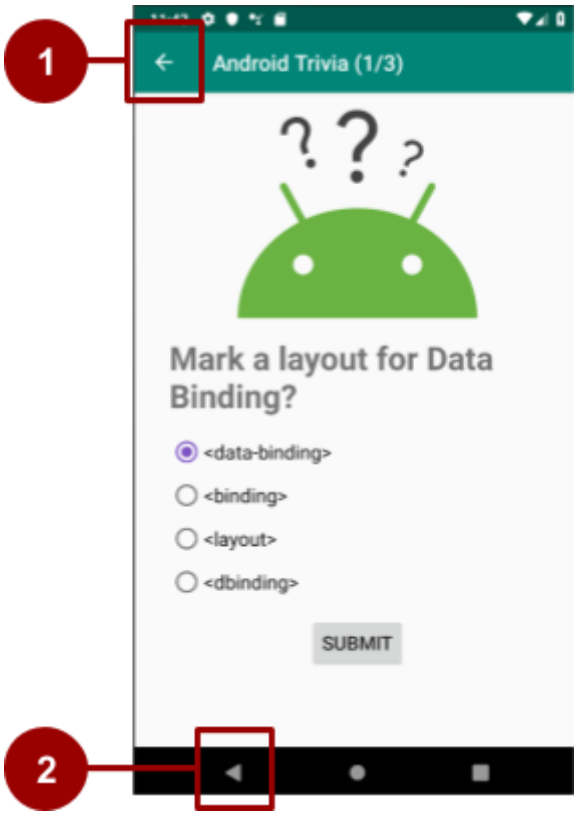
Currently in your app, the user uses the system Back button to navigate to previous screens. However, Android apps can also have an on-screen Up button that appears at the top left of the app bar.

In the AndroidTrivia app, you want the Up button to appear on every screen except the title screen. The Up button should disappear when the user reaches the title screen, because the title screen is at the top of the app's screen hierarchy.

Up button versus Back button:

- The Up button, shown as 1 in the screenshot below, appears in the app bar.
- The Up button navigates within the app, based on the hierarchical relationships between screens. The Up button never navigates the user out of the app.
- The Back button, shown as 2 in the screenshot below, appears in the system navigation bar or as a mechanical button on the device itself, no matter what app is open.
- The Back button navigates backward through screens that the user has recently worked with (the back stack).

For more details, see Designing Back and Up navigation.



Add support for an Up button

The navigation components include a UI library called `NavigationUI`. The Navigation component includes a NavigationUI class. This class contains static methods that manage navigation with the top app bar, the navigation drawer, and bottom navigation. The navigation controller integrates with the app bar to implement the behavior of the Up button, so you don't have to do it yourself.

In the following steps, you use the navigation controller to add an Up button to your app:

1. Open the `MainActivity.kt` kotlin file. Inside the `onCreate()` method, add code to find the navigation controller object:

```
val navController = this.findNavController(R.id.myNavHostFragment)
```

2. Also inside the `onCreate()` method, add code to link the navigation controller to the app bar:

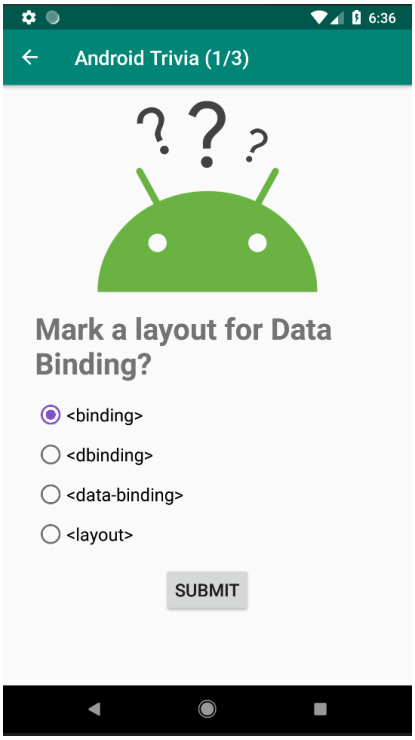
```
NavigationUI.setupActionBarWithNavController(this,navController)
```

3. After the `onCreate()` method, override the `onSupportNavigateUp()` method to call `navigateUp()` in the navigation controller:

```
override fun onSupportNavigateUp(): Boolean {
        val navController = this.findNavController(R.id.myNavHostFragment)
        return navController.navigateUp()
    }
```

4. Run the app. The Up button appears in the app bar on every screen except the title screen. No matter where you are in the app, tapping the Up button takes you to the title screen.

You may see "fragment_title" in the upper left corner. Edit **res>navigation>navigation.xml** and change the label of `com.example.android.navigation.TitleFragment` from "fragment_title" to @string/app_name. And then define this resource in **res>values>strings.xml** as "Android Trivia" and re-run the app.



9. Task: Add an options menu

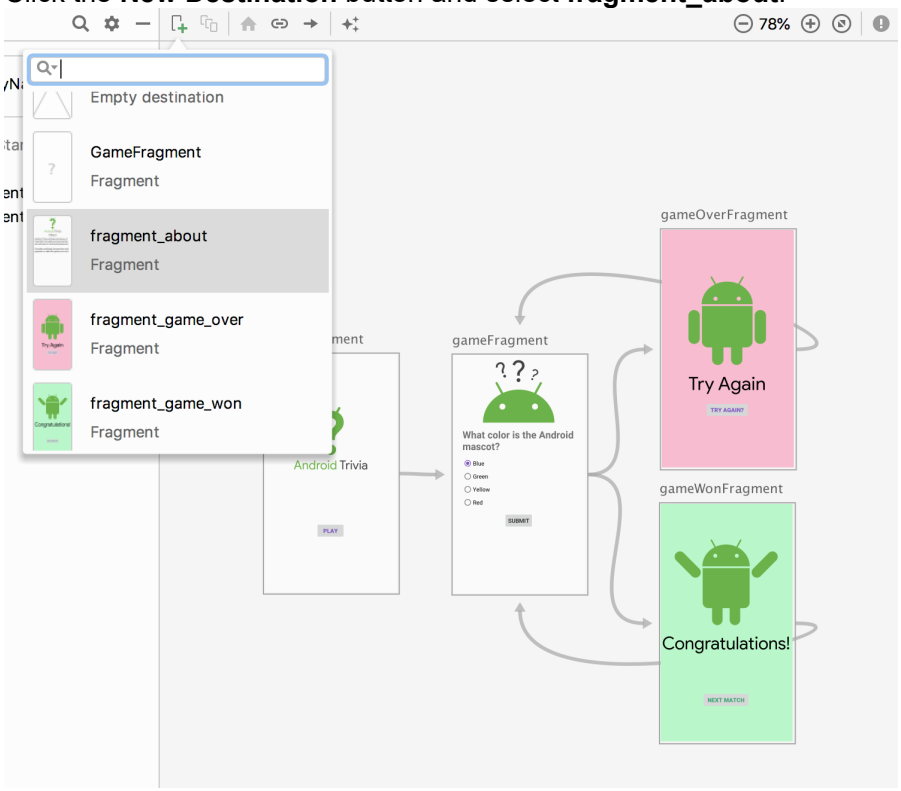
Android has different types of menus, including the *options menu*. On modern Android devices, the user accesses the options menu by tapping three vertical dots

that appear in the app bar.

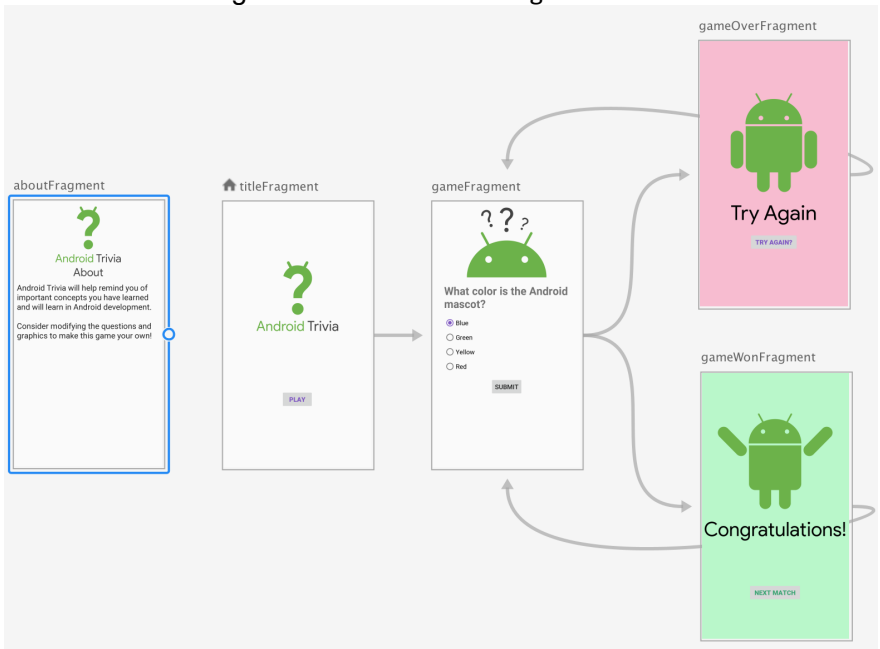
In this task, you add an **About** menu item to the options menu. When the user taps the **About** menu item, the app navigates to the `AboutFragment`, and the user sees information about how to use the app.

Step 1: Add the AboutFragment to the navigation graph

1. Open the `navigation.xml` file and click the **Design** tab to see the navigation graph.
2. Click the **New Destination** button and select **fragment_about**.

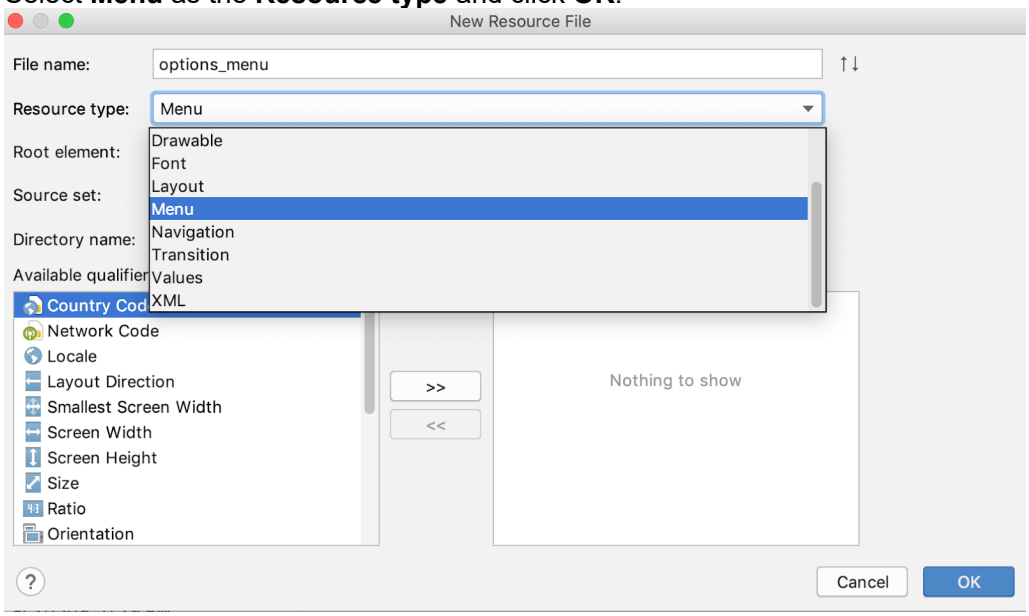


3. In the layout editor, drag the "about" Fragment to the left so it doesn't overlap with the other fragments. Make sure the Fragment's `ID` is `aboutFragment`**.**

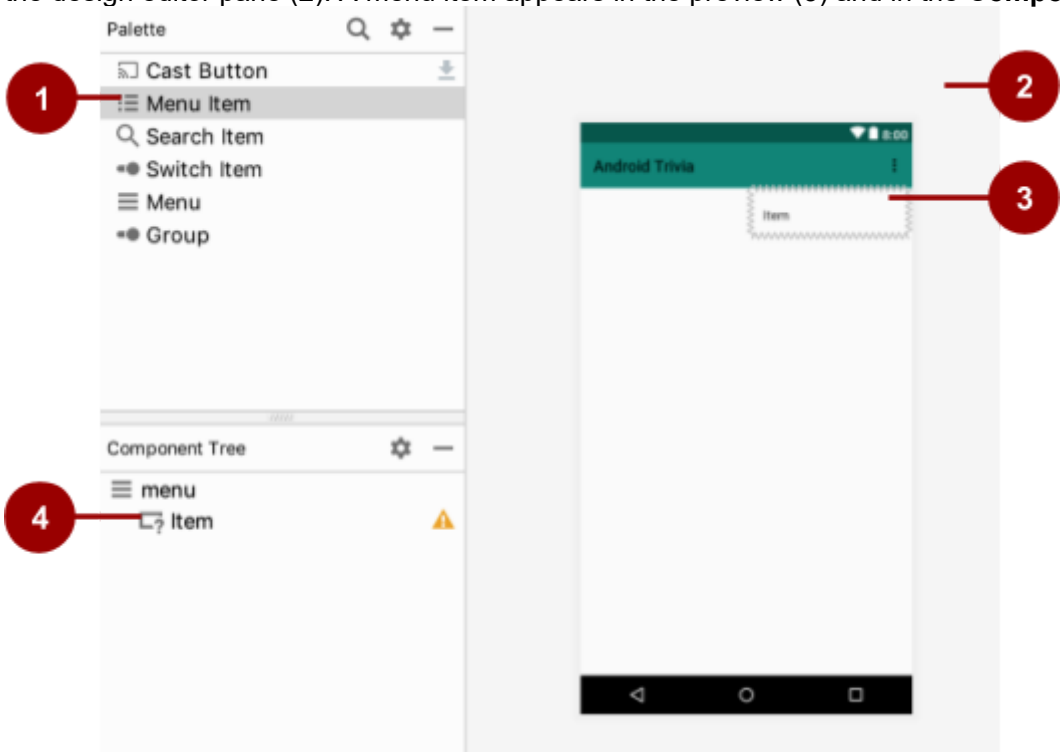


Step 2: Add the options-menu resource

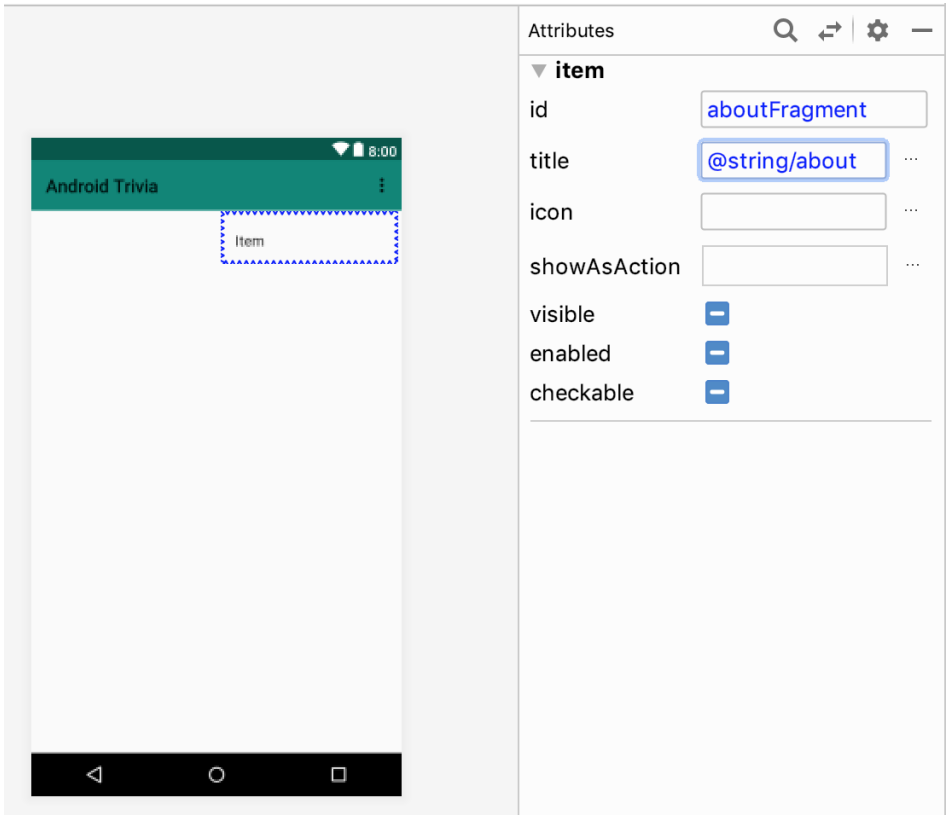
1. In the Android Studio Project pane, right-click the **res** folder and select **New > Android Resource File**.
2. In the **New Resource File** dialog, name the file `options_menu` .
3. Select **Menu** as the **Resource type** and click **OK**.



4. Open the `options_menu.xml` file from the **res > menu** folder and click the **Design** tab to see the Layout Editor.
5. From the **Palette** pane, drag a **Menu Item** (shown as 1 in the screenshot below) and drop it anywhere in the design editor pane (2). A menu item appears in the preview (3) and in the **Component Tree** (4).



6. In the preview or in the **Component Tree**, click the menu item to show its attributes in the **Attributes** pane.
7. Set the menu item's ID to **aboutFragment**. Set the title to **@string/about**.



Tip: Make sure that the `ID` of the menu item that you just added is exactly the same as the `ID` of the `AboutFragment` that you added in the navigation graph. This will make the code for the `onClick` handler much simpler.

Step 3: Add an onClick handler

In this step, you add code to implement behavior when the user taps the **About** menu item.

1. Open the `TitleFragment.kt` Kotlin file. Inside the `onCreateView()` method, before the `return`, call the `setHasOptionsMenu()` method and pass in `true`.

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    ...
    setHasOptionsMenu(true)
    return binding.root
}
```

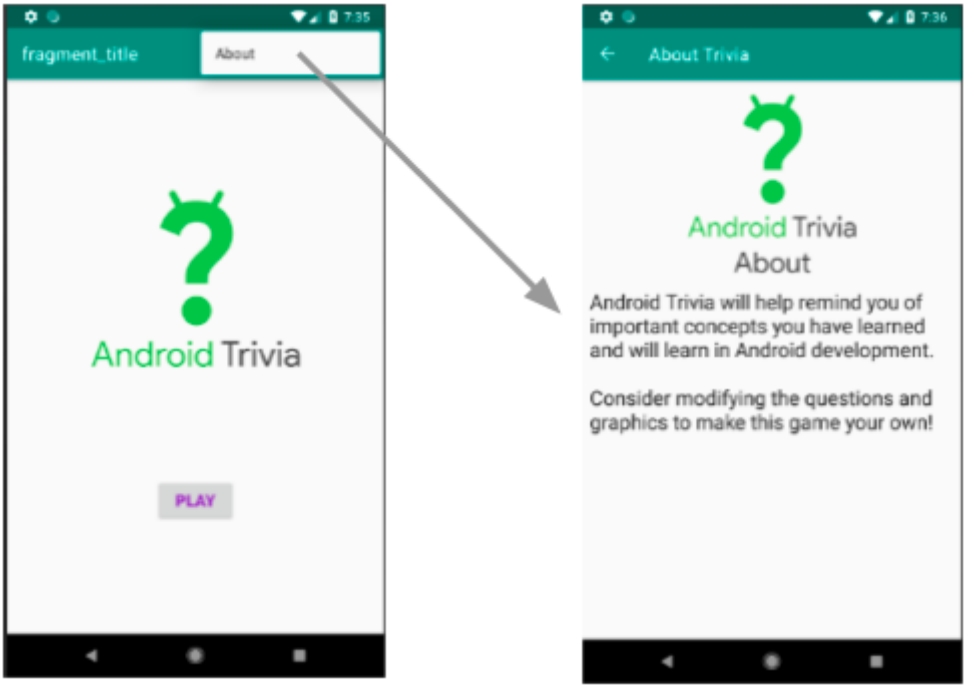
2. After the `onCreateView()` method, override the `onCreateOptionsMenu()` method. In the method, add the options menu and inflate the menu resource file.

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

3. Override the `onOptionsItemSelected()` method to take the appropriate action when the menu item is tapped. In this case, the action is to navigate to the Fragment that has the same `id` as the selected menu item.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
     return NavigationUI.
            onNavDestinationSelected(item,requireView().findNavController())
            || super.onOptionsItemSelected(item)
}
```

4. If the app doesn't build, check to see whether you need to import packages to fix unresolved references in the code. For example, you can add `import android.view.*` to resolve several references (and replace more specific imports such as `import android.view.ViewGroup`).
5. Run the app and test the **About** menu item that's in the options menu. When you tap the menu item, the app should navigate to the "about" screen.



10. Task: Add the navigation drawer

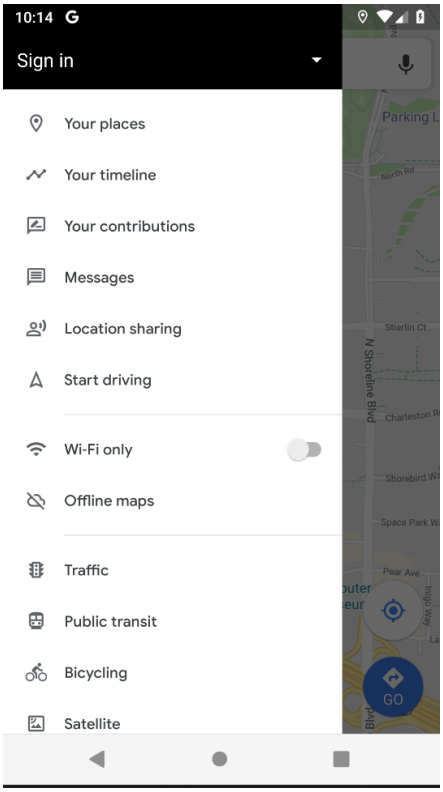
In this task, you add a *navigation drawer* to the AndroidTrivia app. The navigation drawer is a panel that slides out from the edge of the screen. The drawer typically contains a header and a menu.

On phone-sized devices, the navigation drawer is hidden when not in use. Two types of user actions can make the navigation drawer appear:

- The drawer appears when the user swipes from the starting edge of the screen toward the ending edge of the screen. In the AndroidTrivia app, the navigation drawer appears when the user swipes from left to right.
- The drawer appears when the user is at the start destination of the app and taps the *drawer icon* in the app bar. (The drawer icon is sometimes called the *nav drawer button* or *hamburger icon*

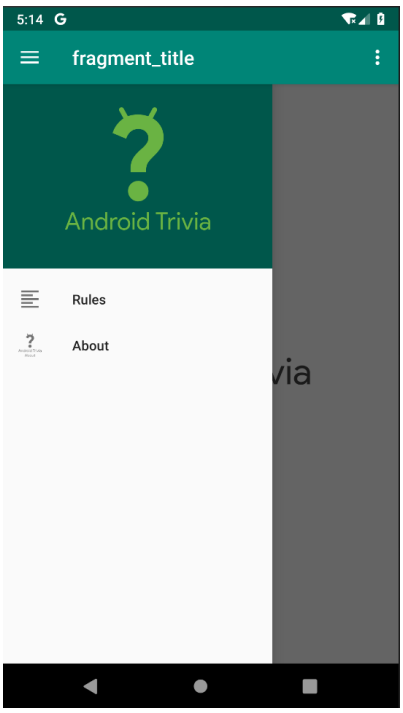
.)

The screenshot below shows an open navigation drawer.



The navigation drawer is part of the Material Components for Android library, or Material library. You use the Material library to implement patterns that are part of Google's Material Design guidelines.

In your AndroidTrivia app, the navigation drawer will contain two menu items. The first item points to the existing "about" Fragment, and the second item will point to a new "rules" Fragment.



Step 1: Add the Material library to your project

1. In the app-level Gradle build file, add the dependency for the Material library:

```
dependencies {
    ...
    implementation "com.google.android.material:material:$version"
    ...
}
```

2. Sync your project.

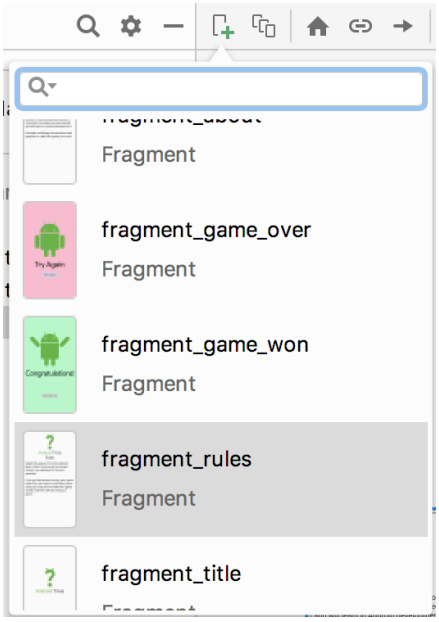
> **Note:** You may have to create or update the variable **$version** in your project's **File>ProjectStructure>Variables** settings to match the latest version.

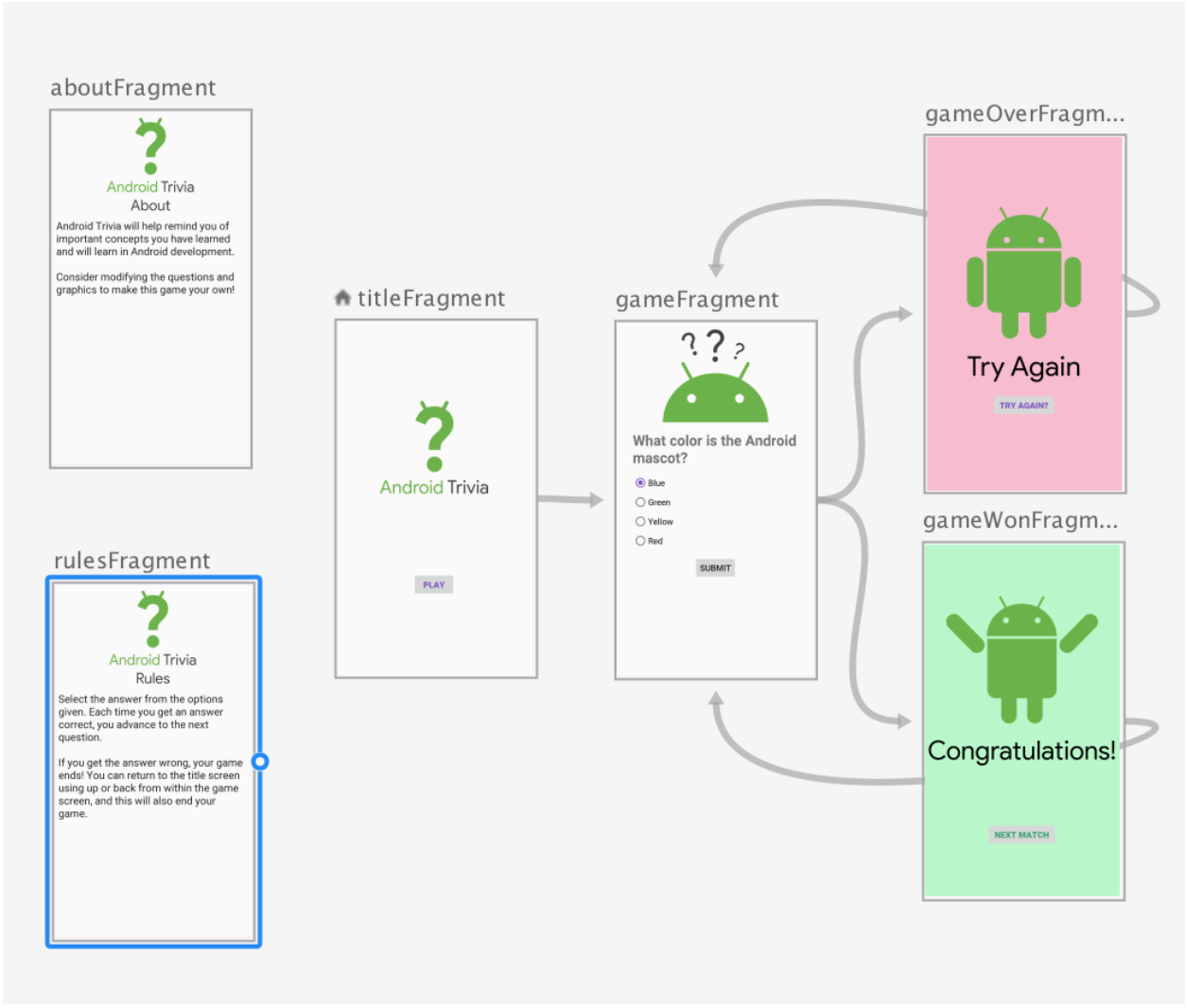
Step 2: Make sure the destination fragments have IDs

The navigation drawer will have two menu items, each representing a Fragment that can be reached from the navigation drawer. Both destinations must have an ID in the navigation graph.

The `AboutFragment` already has an `ID` in the navigation graph, but the `RulesFragment` does not, so add it now:

1. Open the `fragment_rules.xml` layout file to see what it looks like. Click the **Design** tab to look at the preview in the design editor.
2. Open the `navigation.xml` file in the Navigation Editor. Click the **New Destination** button and add the rules Fragment. Set its **ID** to **rulesFragment .**

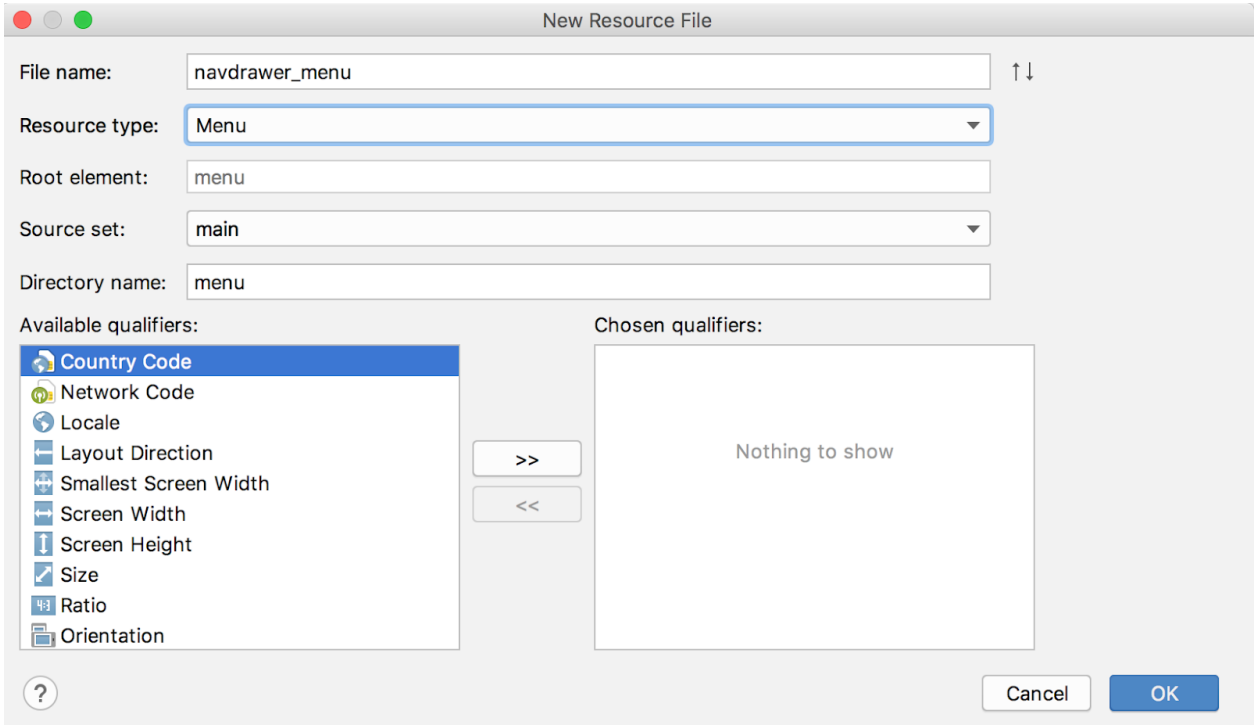




Step 3: Create the drawer menu and the drawer layout

To create a navigation drawer, you create the navigation menu. You also need to put your views inside a `DrawerLayout` in the layout file.

1. Create the menu for the drawer. In the Project pane, right-click the **res** folder and select **New Resource File**. Name the file `navdrawer_menu`, set the resource type to **Menu**, and click **OK**.



2. Open `navdrawer_menu.xml` from the **res > menu** folder, then click the **Design** tab. Add two menu items by dragging them from the **Palette** pane into the **Component Tree** pane.
3. For the first menu item, set the **id** to **rulesFragment.** (The ID for a menu item should be the same as the ID for the Fragment.) Set the **title** to **@string/rules** and the **icon** to **@drawable/rules**.

4. For the second menu item, set the **id** to **aboutFragment**, the **title** string to **@string/about**, and the icon to **@drawable/about_android_trivia**.

> **Note:** If you use the same `ID` for the menu item as for the destination Fragment, you don't need to write any code at all to implement the `onClick` listener!

5. Open the `activity_main.xml` layout file. To get all the drawer functionality for free, put your views inside a `DrawerLayout`. Wrap the entire `<LinearLayout>` inside a `<DrawerLayout>`. (In other words, add a `DrawerLayout` as the root view.)

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <androidx.drawerlayout.widget.DrawerLayout
        android:id="@+id/drawerLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <LinearLayout
        . . .
```

```
        </LinearLayout>
    </androidx.drawerlayout.widget.DrawerLayout>
</layout>
```

6. Now add the drawer, which is a `NavigationView` that uses the `navdrawer_menu` that you just defined. Add the following code in the `DrawerLayout`, after the `</LinearLayout>` element:

```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/navView"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:headerLayout="@layout/nav_header"
    app:menu="@menu/navdrawer_menu" />
```

Step 4: Display the navigation drawer

You created the menu items for the navigation drawer and the navigation drawer layout. Now you need to connect the navigation drawer to the navigation controller so that when users select items in the navigation drawer, the app navigates to the appropriate Fragment.

1. Open the `Mainactivity.kt` Kotlin file. In `onCreate()`, add the code that allows the user to display the navigation drawer. Do this by calling `setupWithNavController()`. Add the following code at the bottom of `onCreate()`:

```
NavigationUI.setupWithNavController(binding.navView, navController)
```

2. Run your app. Swipe from the left edge to display the navigation drawer, and make sure each of the menu items in the drawer goes to the right place.

Although the navigation drawer works, you need to fix one more thing. Typically apps also allow users to display the navigation drawer by tapping the drawer button (three lines)

in the app bar on the home screen. Your app does not yet display the drawer button on the home screen.

Step 5: Display the navigation drawer from the drawer button

The final step is to enable the user to access the navigation drawer from the drawer button at the top left of the app bar.

1. In the `Mainactivity.kt` Kotlin file, add the `lateinit drawerLayout` member variable to represent the drawer layout:

```
private lateinit var drawerLayout: DrawerLayout
```

> **Note:** Kotlin is a "null safety" language. One of the ways it offers null safety is through the **`lateinit`** modifier, which lets you delay the initialization of the variable without any danger of returning a null reference.
>
> In this case, **`drawerLayout`** is declared with **`lateinit`** to avoid the need to make it nullable. It will be initialized in **`onCreate()`**.
>
> To learn more, see Late-Initialized Properties and Variablesin the Kotlin documentation.

2. Inside the `onCreate()` method, initialize `drawerLayout` after the `binding` variable has been initialized.

```
val binding = DataBindingUtil.setContentView<ActivityMainBinding>(this,
                R.layout.activity_main)

drawerLayout = binding.drawerLayout
```

3. Add the `drawerLayout` as the third parameter to the `setupActionBarWithNavController()` method:

```
NavigationUI.setupActionBarWithNavController(this, navController, drawerLayout)
```

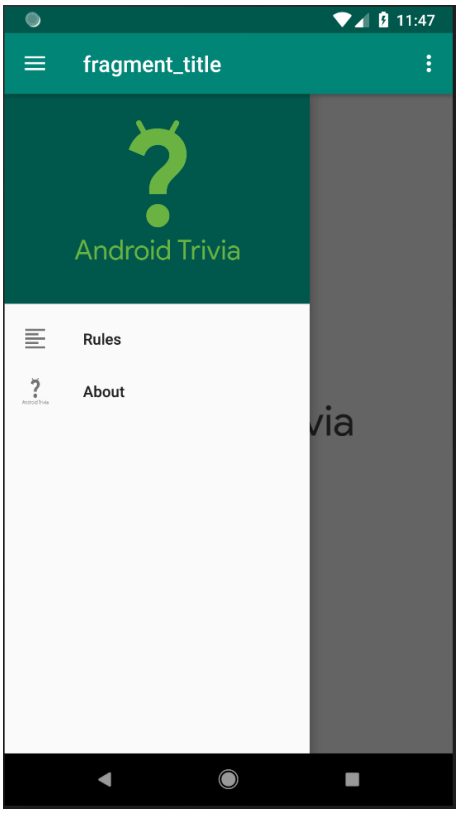
4. Edit the `onSupportNavigateUp()` method to return `NavigationUI .navigateUp` instead of returning `navController .navigateUp` . Pass the navigation controller and the drawer layout to `navigateUp()` . The method will look like as follows:

```
override fun onSupportNavigateUp(): Boolean {
    val navController = this.findNavController(R.id.myNavHostFragment)
    return NavigationUI.navigateUp(navController, drawerLayout)
}
```

5. You might need to add another import to the file so all the references resolve, for example:

```
import androidx.drawerlayout.widget.DrawerLayout
```

6. Run your app. Swipe from the left edge to display the navigation drawer, and make sure each of the menu items in the drawer goes to the right place.
7. Go to the home screen and tap the nav drawer button
   
   to make sure the navigation drawer appears. Make sure that clicking the **Rules** or **About** options in the navigation drawer takes you to the right place.



Congratulations!

You have now added several different navigation options to your app.

The user can now progress through the app by playing the game. They can get back to the home screen at any time by using the Up button. They can get to the About screen either from the Options menu or from the navigation drawer. Pressing the Back button takes them back through previous screens in a way that makes sense for the app. The user can open the navigation drawer by swiping in from the left on any screen, or by tapping the drawer button in the app bar on the home screen.

Your app includes robust, logical navigation paths that are intuitive for your user to use. Congratulations!

Android Kotlin Fundamentals: 03.3 Start an external Activity

About this codelab

subject Last updated Feb 19, 2021

account_circle Written by Google Developers Training team

1. Welcome

This codelab is part of the Android Kotlin Fundamentals course. You'll get the most value out of this course if you work through the codelabs in sequence. All the course codelabs are listed on the Android Kotlin Fundamentals codelabs landing page.

In the previous codelab, you modified the AndroidTrivia app to add navigation to the app. In this codelab, you modify the app so that the user can share their game-play results. The user can initiate an email or text, or they can copy their game-play results to the clipboard.

What you should already know

- The fundamentals of Kotlin
- How to create basic Android apps in Kotlin

What you'll learn

How to use the `Bundle` class to pass arguments from one Fragment to another

How to use the Safe Args Gradle plugin for type safety

How to add a "share" menu item to an app

What an implicit intent is and how to create one

What you'll do

- Modify the AndroidTrivia code to use the Safe Args plugin, which generates `NavDirection` classes.
- Use one of the generated `NavDirection` classes to pass type-safe arguments between the `GameFragment` and the game-state fragments.
- Add a "share" menu item to the app.

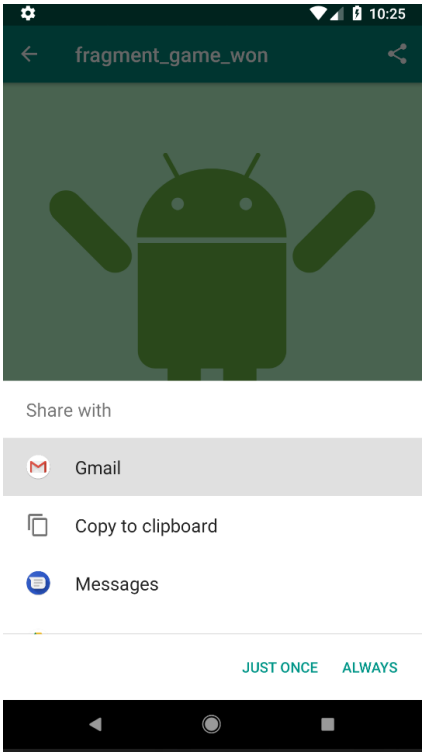
- Create an implicit intent that launches a chooser that the user can use to share a message about their game results.

2. App overview

The AndroidTrivia app, which you worked on in the previous two codelabs, is a game in which users answer questions about Android development. If the user answers three questions correctly, they win the game.

Download the AndroidTriviaNavigationstarter code, or if you successfully completed the previous codelab, use that code as the starter code for this codelab.

In this codelab, you update the AndroidTrivia app so that users can send their game results to other apps and share their results with friends.



3. Task: Set up and use the Safe Args plugin

Before users can share their game results from within the AndroidTrivia app, your code needs to pass parameters from one Fragment to another. To prevent bugs in these transactions and make them type-safe, you use a Gradle plugin called *Safe Args*. The plugin generates `NavDirection` classes, and you add these classes to your code.

In later tasks in this codelab, you use the generated `NavDirection` classes to pass arguments between fragments.

Why you need the Safe Args plugin

Often your app will need to pass data between fragments. One way to pass data from one Fragment to another is to use an instance of the `Bundle` class. An Android `Bundle` is a key-value store.

A *key-value store*, also known as a *dictionary* or *associative array*, is a data structure where you use a unique key (a string) to fetch the value associated with that key. For example:

Key	**Value**
"name"	"Anika"
"favorite_weather"	"sunny"
"favorite_color"	"blue"

Your app could use a `Bundle` to pass data from Fragment A to Fragment B. For example, Fragment A creates a bundle and saves the information as key-value pairs, then passes the `Bundle` to Fragment B. Then Fragment B uses a key to fetch a key-value pair from the `Bundle`. This technique works, but it can result in code that compiles, but then has the potential to cause errors when the app runs.

The kinds of errors that can occur are:

- **Type mismatch errors**. For example, if Fragment A sends a string but Fragment B requests an integer from the bundle, the request returns the default value of zero. Since zero is a valid value, this kind of type mismatch problem does not throw an error when the app is compiled. However, when the user runs the app, the error might make the app misbehave or crash.
- **Missing key errors**. If Fragment B requests an argument that isn't set in the bundle, the operation returns `null`. Again, this doesn't throw an error when the app is compiled but could cause severe problems when the user runs the app.

You want to catch these errors when you compile the app in Android Studio, so that you catch these errors before deploying the app into production. In other words, you want to catch the errors during app development so that your users don't encounter them.

To help with these problems, Android's Navigation Architecture Component includes a feature called *Safe Args*. Safe Args is a Gradle plugin that generates code and classes that help detect errors at compile-time that might not otherwise be surfaced until the app runs.

Step 1: Open the starter app and run it

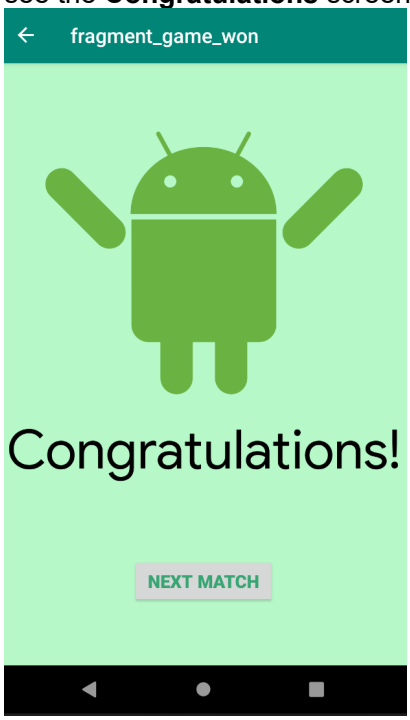
1. Download the AndroidTriviaNavigationstarter app for this codelab:

If you completed the previous codelab on adding navigation to your app, use your solution code from that codelab.

Run the app from Android Studio:

1. Open the app in Android Studio.

2. Run the app on an Android-powered device or on an emulator. The app is a trivia game with a navigation drawer, an options menu on the title screen, and an Up button at the top of most of the screens.
3. Explore the app and play the game. When you win the game by answering three questions correctly, you see the **Congratulations** screen.



In this codelab, you add a **share** icon to the top of the **Congratulations** screen. The **share** icon lets the user share their results in an email or text message.

Step 2: Add Safe Args to the project

1. In Android Studio, open the project-level `build.gradle` file.
2. Add the `navigation-safe-args-gradle-plugin` dependency, as shown below:

```
// Adding the safe-args dependency to the project Gradle file
dependencies {
   ...
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$navigationVersion"

}
```

3. Open the app-level `build.gradle` file.
4. At the top of the file, after all the other plugins, add the `apply plugin` statement with the `androidx.navigation.safeargs` plugin:

```
// Adding the apply plugin statement for safeargs
apply plugin: 'androidx.navigation.safeargs'
```

5. Re-build the project. If you are prompted to install additional build tools, install them.

The app project now includes generated `NavDirection` classes.

The Safe Args plugin generates a `NavDirection` class for each Fragment. These classes represent navigation from all the app's actions.

For example, `GameFragment` now has a generated `GameFragmentDirections` class. You use the `GameFragmentDirections` class to pass type-safe arguments between the game Fragment and other fragments in the app.

To see the generated files, explore the **generatedJava** folder in the **Project > Android** pane.

> **Caution:** Do not edit the NavDirection classes. These classes are regenerated whenever the project is compiled, and your edits will be lost.

Step 3: Add a NavDirection class to the game Fragment

In this step, you add the `GameFragmentDirections` class to the game Fragment. You'll use this code later to pass arguments between the `GameFragment` and the game-state fragments (`GameWonFragment` and `GameOverFragment`).

1. Open the `GameFragment.kt` Kotlin file that's in the **java** folder.
2. Inside the `onCreateView()` method, locate the game-won conditional statement ("We've won!"). Change the parameter that's passed into the `NavController.navigate()` method: Replace the action ID for the game-won state with an ID that uses the `actionGameFragmentToGameWonFragment()` method from the `GameFragmentDirections` class.

The conditional statement now looks like the following code. You'll add parameters to the `actionGameFragmentToGameWonFragment()` method in the next task.

```
// Using directions to navigate to the GameWonFragment
view.findNavController()
        .navigate(GameFragmentDirections.actionGameFragmentToGameWonFragment())
```

3. Likewise, locate the game-over conditional statement ("Game over!"). Replace the action ID for the game-over state with an ID that uses the game-over method from the `GameFragmentDirections` class:

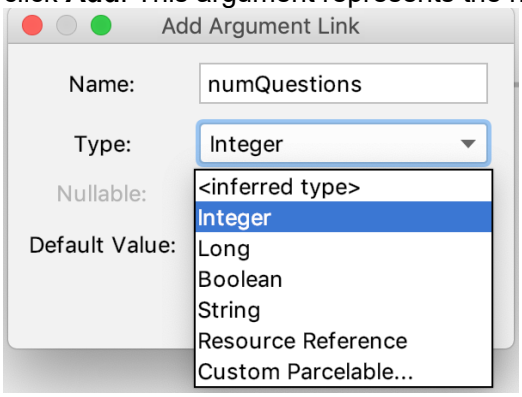
```
// Using directions to navigate to the GameOverFragment
view.findNavController()
        .navigate(GameFragmentDirections.actionGameFragmentToGameOverFragment())
```

4. Task: Add and pass arguments

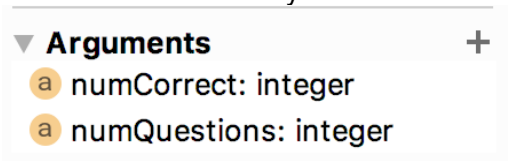
In this task, you add type-safe arguments to the `gameWonFragment` and pass the arguments safely into a `GameFragmentDirections` method. Similarly you then will replace the other Fragment classes with their equivalent `NavDirection` classes.

Step 1: Add arguments to the game-won Fragment

1. Open the `navigation.xml` file, which is in the **res > navigation** folder. Click the **Design** tab to open the navigation graph, which is where you'll set the arguments in the fragments.
2. In the preview, select the **gameWonFragment**.
3. In the **Attributes** pane, expand the **Arguments** section.
4. Click the + icon to add an argument. Name the argument `numQuestions` and set the type to **Integer**, then click **Add**. This argument represents the number of questions the user answered.



5. Still with the **gameWonFragment** selected, add a second argument. Name this argument **numCorrect** and set its type to **Integer**. This argument represents the number of questions the user answered correctly.



If you try to build the app now, you will likely get two compile errors.

```
No value passed for parameter 'numQuestions'
No value passed for parameter 'numCorrect'
```

You fix this error in the coming steps.

> **Note:** If you're using Android Studio 3.2 or lower, you might have to change `app:type = "integer"` to `app:argType = "integer"` in the `navigation.xml` file.

Step 2: Pass the arguments

In this step, you pass the `numQuestions` and `questionIndex` arguments into the `actionGameFragmentToGameWonFragment()` method from the `GameFragmentDirections` class.

1. Open the `GameFragment.kt` Kotlin file and locate the game-won conditional statement:

```
else {
 // We've won!  Navigate to the gameWonFragment.
 view.findNavController()
```

```
        .navigate(GameFragmentDirections
                .actionGameFragmentToGameWonFragment())
}
```

2. Pass the `numQuestions` and `questionIndex` parameters to the `actionGameFragmentToGameWonFragment()` method:

```
// Adding the parameters to the Action
view.findNavController()
        .navigate(GameFragmentDirections
                .actionGameFragmentToGameWonFragment(numQuestions, questionIndex))
```

You pass the total number of questions as `numQuestions` and the current question being attempted as `questionIndex`. The app is designed in such a way that the user can only share their data if they answer all the questions correctly—the number of correct questions always equals the number of questions answered. (You can change this game logic later, if you want.)

3. In `GameWonFragment.kt`, extract the arguments from the bundle, then use a `Toast` to display the arguments. Put the following code in the `onCreateView()` method, before the `return` statement:

```
val args = GameWonFragmentArgs.fromBundle(requireArguments())
Toast.makeText(context, "NumCorrect: ${args.numCorrect}, NumQuestions: ${args.numQuestions
```

4. Run the app and play the game to make sure that the arguments are passed successfully to the `GameWonFragment`. The toast message appears on the **Congratulations** screen, saying "NumCorrect: 3, NumQuestions: 3".

You do have to win the trivia game first, though. To make the game easier, you can change it to a single-question game by setting the value of `numQuestions` to `1` in the `GameFragment.kt` Kotlin file.

Step 3: Replace Fragment classes with NavDirection classes

When you use "safe arguments," you can replace Fragment classes that are used in navigation code with `NavDirection` classes. You do this so that you can use type-safe arguments with other fragments in the app.

In `TitleFragment`, `GameOverFragment`, and `GameWonFragment`, change the action ID that's passed into the `navigate()` method. Replace the action ID with the equivalent method from the appropriate `NavDirection` class:

1. Open the `TitleFragment.kt` Kotlin file. In `onCreateView()`, locate the `navigate()` method in the **Play** button's click handler.
   Pass `TitleFragmentDirections.actionTitleFragmentToGameFragment()` as the method's argument:

```
binding.playButton.setOnClickListener { view: View ->
    view.findNavController()
            .navigate(TitleFragmentDirections.actionTitleFragmentToGameFragment())
}
```

2. In the `GameOverFragment.kt` file, in the **Try Again** button's click handler, pass `GameOverFragmentDirections.actionGameOverFragmentToGameFragment()` as the `navigate()` method's argument:

```
binding.tryAgainButton.setOnClickListener { view: View ->
    view.findNavController()
        .navigate(GameOverFragmentDirections.actionGameOverFragmentToGameFragment())
}
```

3. In the `GameWonFragment.kt` file, in the **Next Match** button's click handler, pass `GameWonFragmentDirections.actionGameWonFragmentToGameFragment()` as the `navigate()` method's argument:

```
binding.nextMatchButton.setOnClickListener { view: View ->
    view.findNavController()
          .navigate(GameWonFragmentDirections.actionGameWonFragmentToGameFragment())
}
```

4. Run the app.

You won't find any changes to the app's output, but now the app is set up so that you can easily pass arguments using `NavDirection` classes whenever needed.

5. Task: Sharing game results

In this task, you add a sharing feature to the app so that the user can share their game results. This is implemented by using an Android Intent, specifically an implicit intent. The sharing feature will be accessible via an options menu inside the `GameWonFragment` class. In the app's UI, the menu item will appear as a **share** icon at the top of the **Congratulations** screen.

Implicit intents

Up until now, you've used navigation components to navigate among fragments within your Activity. Android also allows you to use *intents* to navigate to activities that other apps provide. You use this functionality in the AndroidTrivia app to let the user share their game-play results with friends.

An `Intent` is a simple message object that's used to communicate between Android components. There are two types of intents: explicit and implicit. You can send a message to a specific target using an *explicit intent*. With an *implicit intent*, you initiate an Activity without knowing which app or Activity will handle the task. For example, if you want your app to take a photo, you typically don't care which app or Activity performs the task. When multiple Android apps can handle the same implicit intent, Android shows the user a chooser, so that the user can select an app to handle the request.

Each implicit intent must have an `ACTION` that describes the type of thing that is to be done. Common actions, such as `ACTION_VIEW`, `ACTION_EDIT`, and `ACTION_DIAL`, are defined in the `Intent` class.

> **Terminology alert!** `Intent` actions are unrelated to actions shown in the app's navigation graph.

For more about implicit intents, see Sending the User to Another App.

Step 1: Add an options menu to the Congratulations screen

1. Open the `GameWonFragment.kt` Kotlin file.
2. Inside the `onCreateView()` method, before the `return`, call the `setHasOptionsMenu()` method and pass in `true`:

```
setHasOptionsMenu(true)
```

Step 2: Build and call an implicit intent

Modify your code to build and call an `Intent` that sends the message about the user's game data. Because several different apps can handle an `ACTION_SEND` intent, the user will see a chooser that lets them select how they want to send their information.

1. Inside the `GameWonFragment` class, after the `onCreateView()` method, create a private method called `getShareIntent()`, as shown below. The line of code that sets a value for `args` is identical to the line of code used in the class's `onCreateView()`.

In the rest of the method's code, you build an `ACTION_SEND` intent to deliver the message that the user wants to share. The data's MIME type is specified by the `setType()` method. The actual data to be delivered is specified in the `EXTRA_TEXT`. (The `share_success_text` string is defined in the `strings.xml` resource file.)

```
// Creating our Share Intent
private fun getShareIntent() : Intent {
    val args = GameWonFragmentArgs.fromBundle(requireArguments())
    val shareIntent = Intent(Intent.ACTION_SEND)
    shareIntent.setType("text/plain")
            .putExtra(Intent.EXTRA_TEXT, getString(R.string.share_success_text, args.numCo
```

```
    return shareIntent
}
```

2. Below the `getShareIntent()` method, create a `shareSuccess()` method. This method gets the `Intent` from `getShareIntent()` and calls `startActivity()` to begin sharing.

```
// Starting an Activity with our new Intent
private fun shareSuccess() {
    startActivity(getShareIntent())
}
```

3. The starter code already contains a `winner_menu.xml` menu file. Override `onCreateOptionsMenu()` in the `GameWonFragment` class to inflate `winner_menu`.

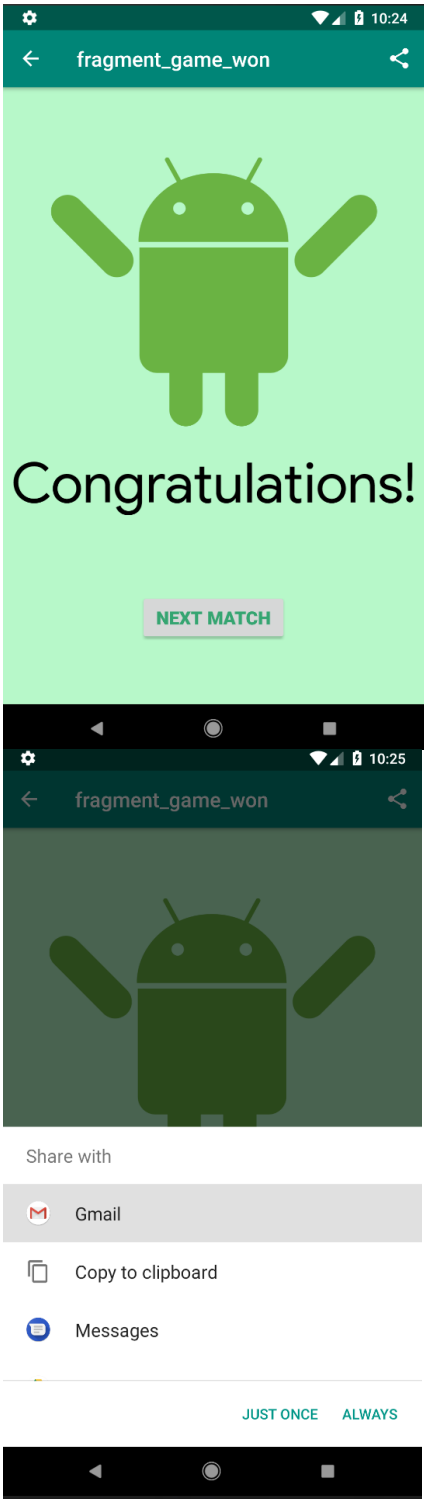
Use `getShareIntent()` to get the `shareIntent`. To make sure the `shareIntent` resolves to an `Activity`, check with the Android package manager (`PackageManager`), which keeps track of the apps and activities installed on the device. Use the Activity's `packageManager` property to gain access to the package manager, and call `resolveActivity()`. If the result equals `null`, which means that the `shareIntent` doesn't resolve, find the sharing menu item from the inflated menu and make the menu item invisible.

```
// Showing the Share Menu Item Dynamically
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.winner_menu, menu)
        if(getShareIntent().resolveActivity(requireActivity().packageManager)==null){
            menu.findItem(R.id.share).isVisible = false
        }
}
```

4. To handle the menu item, override `onOptionsItemSelected()` in the `GameWonFragment` **class**. Call the `shareSuccess()` method when the menu item is clicked:

```
// Sharing from the Menu
override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when(item.itemId){
            R.id.share -> shareSuccess()
        }
        return super.onOptionsItemSelected(item)
}
```

5. Now run your app. (You might need to import some packages into `GameWonFragment.kt` before the code will run.) After you win the game, notice the **share** icon that appears at the top right of the app bar. Click the share icon to share a message about your victory.


10:24
fragment_game_won
Congratulations!
NEXT MATCH
10:25
fragment_game_won
Share with
Gmail
Copy to clipboard
Messages
JUST ONCE
ALWAYS