Preflight Checks (Before You Start) **2**
Grab the project off github 2
Look at the file structure 3
Try to build the project (fix errors) 4
Debug the app 5
Logging 5
Breakpoints 6
Setting a debugger 7
Diving Into The App **8**
Fixing our App (TODO: #PERMISSION) 8
Fixing our API (TODO: #FIX-API) 8
Setting up our Activity (TODO: #SET-UP-ACTIVITY) 10
Picking our dogs (TODO: #PICK-DOGS) 11
Pick the correct dog (TODO: #PICK-CORRECT-DOG) 12
Checking answers (TODO: #CHECK-ANSWER) 14
Start the second round (TODO: #ROUND-COMPLETE) 15
Add a sixth image (Optional) (TODO: #ADD-IMAGE) 16
BONUS (Optional) 22
Implement a launch screen 22
Keep score and implement a results page 22

Don’t copy-paste from this document. It’s better to write things down yourself.

The format of the sections starts vague and then gets very specific. If you’re comfortable, try it yourself! Lean on the code written as much as you need to, or ask for more help if the document isn’t as clear as it tries to be.

At the start of each section, you should be able to run and build the project. If you are unable to do so, raise your hand and ask for help.

Because of this, the order in which we write things may be slightly nonsensical. Keep this in mind while coding. We’re not going to be able to test what we’re doing as we’re doing it, but in the real world we’re not as strapped for time.

One last thing: When calling upon functions that have already been written for you, go explore them. Find out what they do, why they do it, and how they do it.

A Quick Note: THIS APP WILL CRASH when you try running it at first. There’s nothing wrong! (well there is obviously) The first thing we’ll do is fix this problem once we start digging into the code.

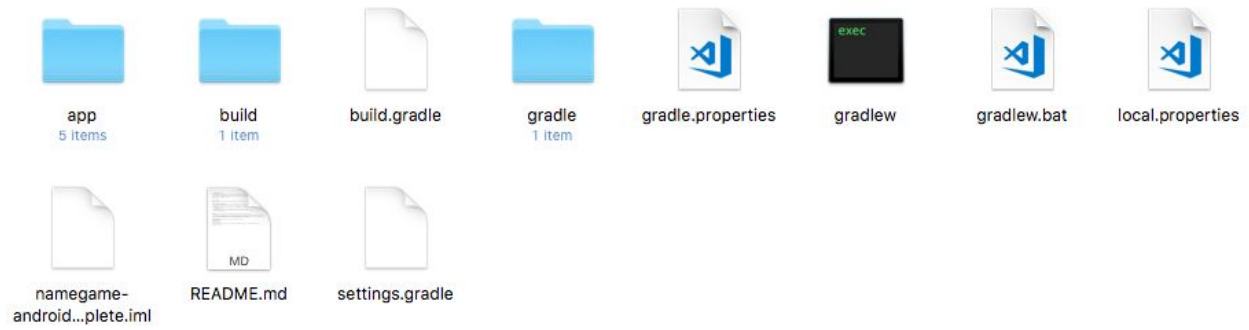
Preflight Checks (Before You Start)

Grab the project off github

Use the following URL to grab the incomplete project from github.

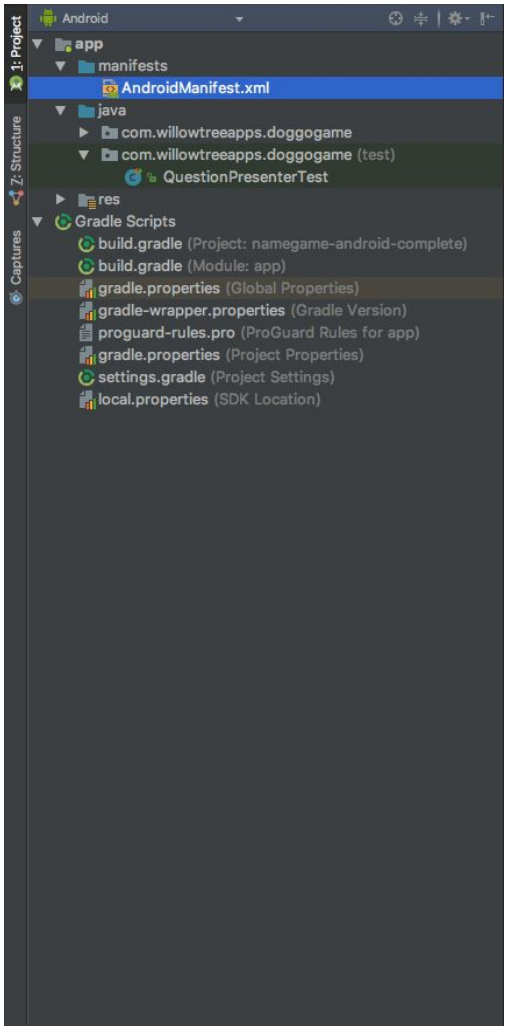
Clone or download this GitHub repository and extract it to your computer.

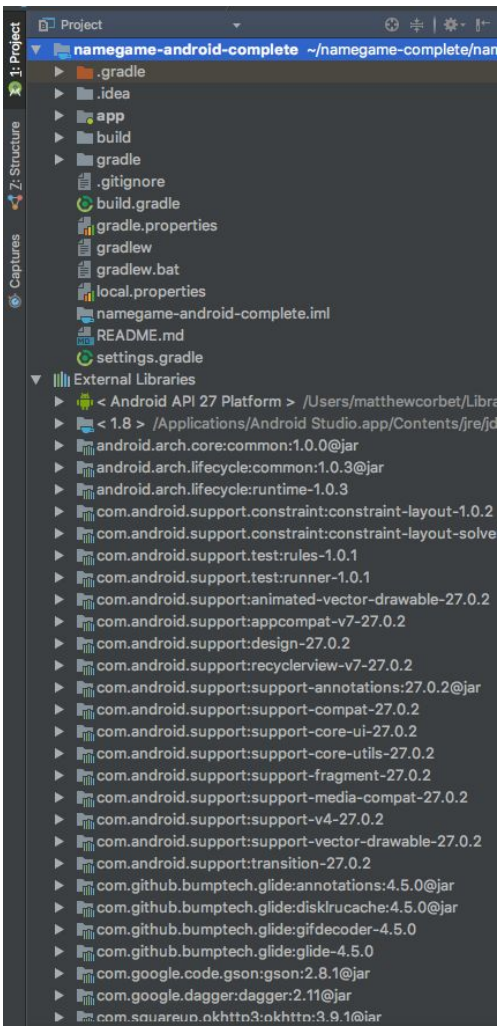
Then open Android Studio to import the project.
Click on **File > New > Import Project** and choose the project. To import successfully, choose to import at the highest level folder of the application. Your view should be something like this (depending on your OS)



Look at the file structure

On the left side of the Android Studio window, you'll see a few different vertically-aligned tabs. One of those is the **Project** tab (also accessed by the shortcut **Control (Command on Mac) + 1**.





Depending on your view, you'll see different things. The left image iis the **Android** view (and this is the view you should stick with for this workshop), while the right image is the **Project** view. The Project view is a bit closer to the actual file structure, while the Android view groups together certain things.

To switch views, click the icon in the top left of this menu (visible in the above screenshots as **Android** and **Project**). Switch to the Android view.

In the Android view, there are two main folders - **app** and **Gradle Scripts**. The Gradle scripts help us import libraries, set up builds, and a lot more. We're not worried about Gradle for right now, so you can ignore anything there.

Under **app** we have three more folders: **manifests**, **java**, and **res.**

Manifests holds our AndroidManifest, used to declare all our activities, fragments...basically our screens. It also handles declarations for things like permissions and other metadata.

Java holds all our code. It's where we implement everything the app needs - our views, our data models - everything. It also holds all our tests.

Res holds all of our resources. These can be things like colors, strings/text, and dimensions that we want to reference all throughout the code. This folder also holds all our layout resource files for defining what our screens look like.

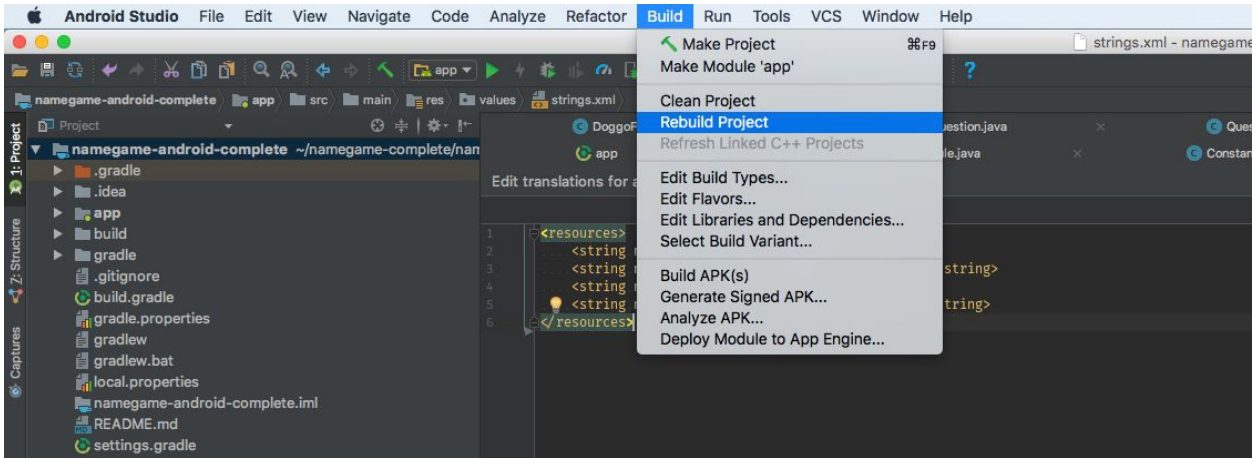
In the **java** folder you'll see two folders. Both are named **com.willowtreeapps.doggogame**, but one has (test) beside it. The one without the (test) is all our source code. Expanding that will show all the packages we've set up. Looking around there you'll find all the classes used in the app.

Try to build the project (fix errors)

When you first import the project, it should run automatically and try to build. There might be times where you need to rebuild the project (maybe you changed something that generates code and now need it to regenerate the code with your changes).

To rebuild your project, open the **Build** menu and select **Rebuild Project.** If there are any errors, they should show up in the **Message** pane in the bottom of the IDE.

To rebuild:



Often, Android Studio will be kind enough to tell you exactly what went wrong and point you to the exact spot where this error occurs. Other times you might have to hunt for it (your instructor and/or Google and StackOverflow are your friends here).

Debug the app

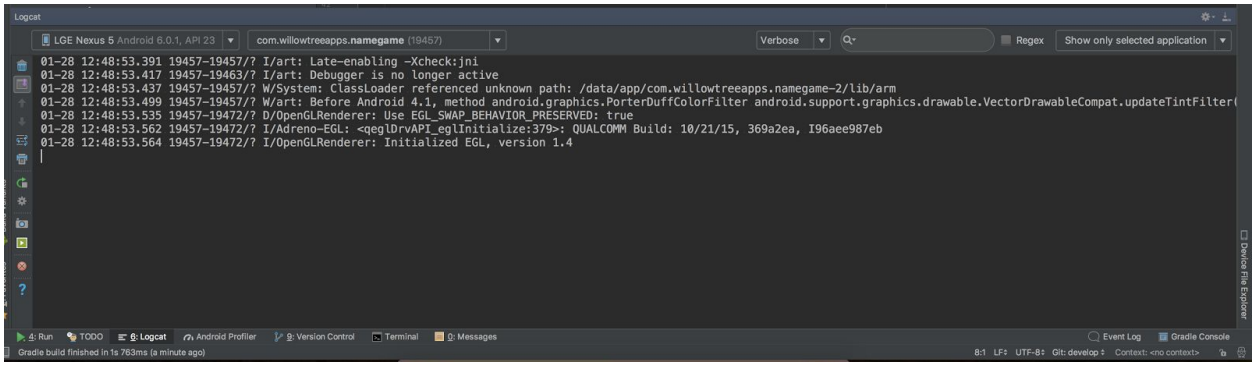
Debugging is incredibly important. We can step through the app, putting breakpoints at certain spots to ensure that our logic is sound and that we're doing things in the order we want. There's extensive documentation on debugging on the official Android documentation, but three quick things to be aware of when debugging: **Logging**, **Breakpoints**, and **Running the app with a debugger.**

Logging

To Log things, we can use the Android Log class. Let's say we have a `Person` object with fields `private String firstName` and `private String lastName`. We want to verify that firstName is "Rick", so we might use this statement to Log it:

```
Log.d(TAG, person.getFirstName());
```

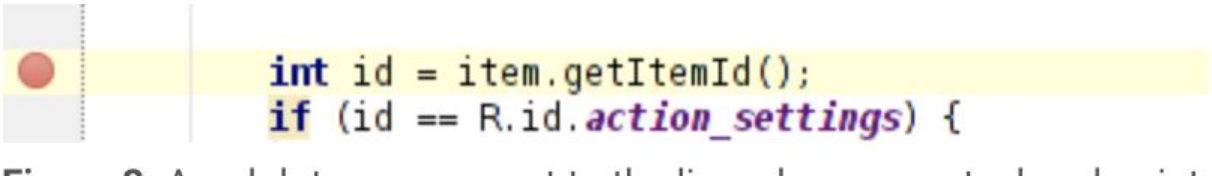
TAG is a custom String we define so that we can easily find the things we want when we're debugging.
We can see the output of this in our bottom panel:



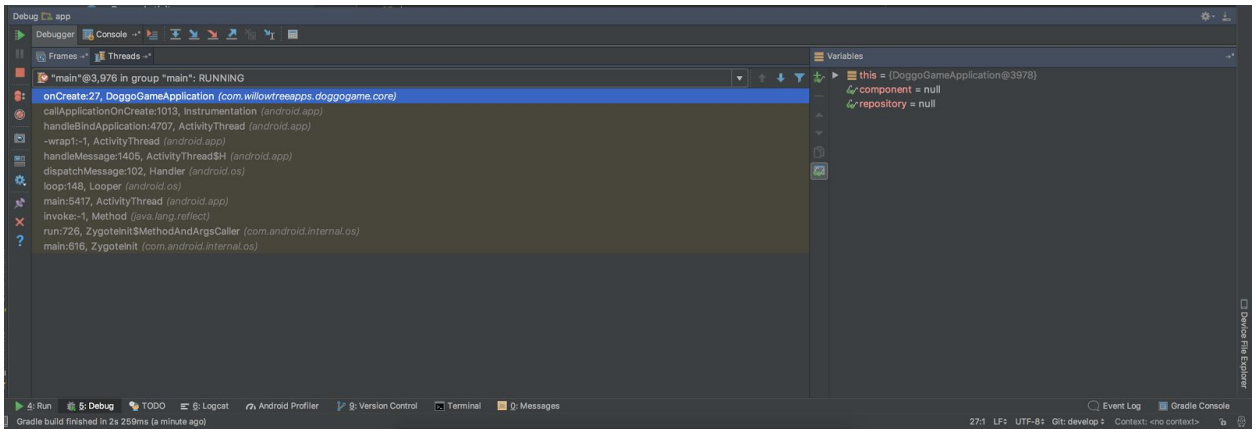
We're on the **Logcat** tab at the bottom of the IDE, and all this output in the text box contains various logging messages. Using the text field in the top right of this area, we can narrow things down to only the tags we want.

Breakpoints

To set a **Breakpoint,** either click on the left gutter (just to the left of the code) or bring your cursor to the line you want to breakpoint and hit Control (Command on Mac) + F8. The end result should be a line that looks like `int id = item.getItemId();`:



When you're in your app and you hit a breakpoint, all of a sudden the **Debug** panel will pop up from the bottom and let you know all this information. Let's see what that looks like and how to interpret everything.



On the far left here are a bunch of buttons (Play, Stop, etc). The play button will continue the app from where the breakpoint left off, while the stop button will stop the app entirely.

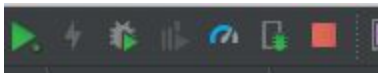
Just to the right of that we can see a bunch of text with a brown-ish background. This is your stack trace to get to where your breakpoint is currently. We can see all the methods (both system methods and our app's methods) that were called so far. We can even jump back in the chain and make sure that methods were getting called in the right places from other classes.

On the far right are all the variables in our current class (relative to the class the breakpoint is in). We can expand these variables to see their values, so we can see the value of each field in a model, or each item in a list. This can be really useful to verify that variables are populated correctly, updated correctly, etc.

Setting a debugger

When you want to debug your app, you have two options: start the app with the debugger or attach a debugging process later.

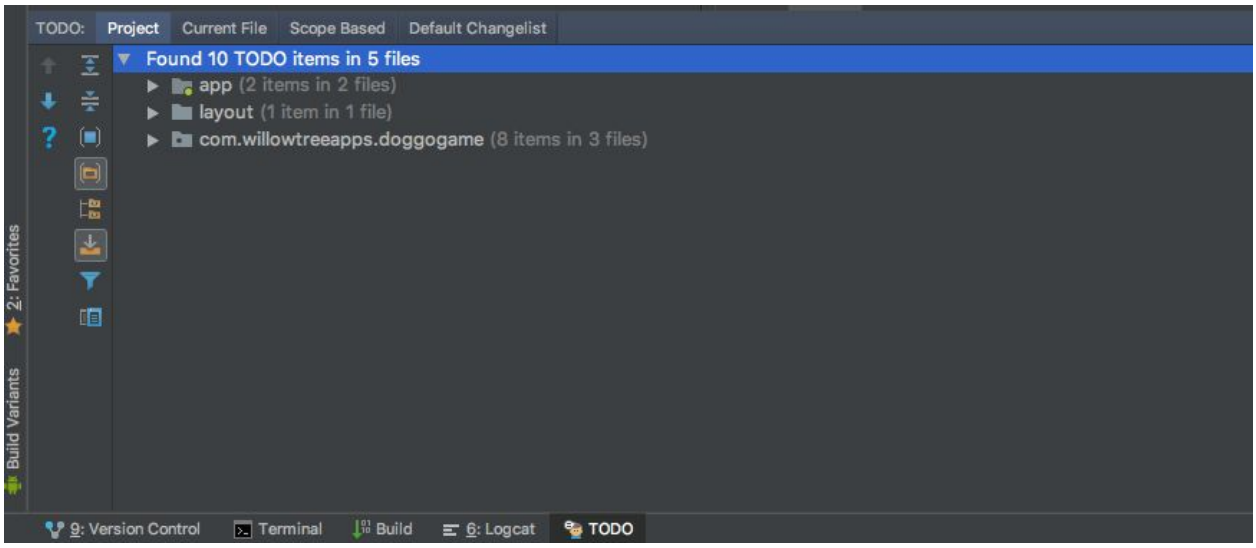
The easiest thing to do here is to start your app with the debugger attached. To do this, hit **Ctrl + D** or click the small bug icon with the play button in the top bar



This will launch the **Deployment Target** window where you can choose which device/emulator you want to run the app on. Pick one, click **OK**, and watch as your app starts up.

Diving Into The App

A quick note: Each section has a TODO followed by a # and descriptor. These TODOs are in the code as well, and correspond with where the work gets done. You can also pull up the TODO tab at the bottom of Android Studio and see where all these TODOs are in the entire codebase.



Fixing our App (TODO: #PERMISSION)

The first thing you might notice is that our app crashes immediately on launch. On Android, when you want to make use of an internet connection in an app, we have to explicitly declare that we need the Internet permission.

To do this, let's open up AndroidManifest.xml.

Let's add a line in there above the `<application>` tag.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Now we should be able to launch the app without a crash! It's not doing anything, but at least we're not crashing anymore.

Fixing our API (TODO: #FIX-API)

Right now our API doesn't really do anything.

This app uses a commonly used library called Retrofit that lets us turn our API calls into a Java interface.

We have two models created for handling these API calls: DoggoBreedResponse and DoggoImage. The Breed Response will be a list of Strings.

Our `fetchDoggos()` method will make the API call and create a Callback for our DoggoBreedResponse. This callback handles two instances: `onResponse` and `onFailure()`. We've already handled onFailure, so let's focus on handling onResponse.

First, we need to check if our response was actually successful. To determine this, let's check the **response** object.

```
if (response.isSuccessful()) {
 //Do some stuff with our successful response
}
```

Let's add an else block in case our response worked. We'll use the getError method already defined to generate the Throwable.

```
if (response.isSuccessful()) {
 //Do some stuff with our successful response
} else {
 onFailure(call, getError(response, "Error getting breed"));
}
```

Next, let's get the body of our response. The way this body is defined, we'll get back a DoggoBreedResponse from the body.

```
if (response.isSuccessful()) {
 DoggoBreedResponse breedResponse = response.body();
} else {
 onFailure(call, getError(response, "Error getting breed"));
}
```

Because this is Java, we don't know if this response body is null. Let's null check it. While we're at it, let's go ahead and get the status from the response and check if that was a success. We'll need to null check it and see if it corresponds to "success".

```
if (response.isSuccessful()) {
 DoggoBreedResponse breedResponse = response.body();
 if (breedResponse != null) {
   String status = breedResponse.getStatus();
   if (status != null && status.equals(DoggoBreedResponse.SUCCESS)) {
   }
 }
} else {
 onFailure(call, getError(response, "Error getting breed"));
}
```

Last step! Since we have all our breeds, we need to make some follow up calls to get an image for each breed. Luckily, the `getImages()` method will do this for us, and all it needs is the list of breeds in our DoggoBreedResponse. Let's add a call to `getImages()`

```
if (response.isSuccessful()) {
 DoggoBreedResponse breedResponse = response.body();
 if (breedResponse != null) {
   String status = breedResponse.getStatus();
   if (status != null && status.equals(DoggoBreedResponse.SUCCESS)) {
      getImages(breedResponse.getBreeds());
   }
 }
} else {
 onFailure(call, getError(response, "Error getting breed"));
}
```

Now we're done with our API and have all we need to get started! If you try to run the app, you'll see that we just sit on a progress spinner and it never goes away. To fix this we'll need to dive into our activity to kick things off.

Setting up our Activity (TODO: #SET-UP-ACTIVITY)

Next stop is our DoggoActivity. In our AndroidManifest, we defined this activity as the launch point in our app with this line:

```
<action android:name="android.intent.action.MAIN" />
```

That's why we see a progress spinner first thing - our DoggoActivity has it visible by default.

We need to do two things to make sure we start our game.

First, let's flesh out initDoggos() - right now it's not doing anything. We have a DoggoRepository available to us, so let's check if we have any dogs available through that.

```
private void initDoggos() {
   if (!doggoRepository.hasDoggos()) {
   }
}
```

So if we **do not** have any dogs available yet, that means that our dogs haven't finished loading yet. Our DoggoActivity implements the **DoggoCallback** methods of onSuccess and onFailure, but right now the repository doesn't have any callbacks registered. Let's register the DoggoActivity as a callback.

```
private void initDoggos() {
   if (!doggoRepository.hasDoggos()) {
       doggoRepository.setCallback(this);
   }
}
```

Now our onSuccess method in DoggoActivity will be called and the spinner goes away! Our game still isn't starting though. We need to tell the activity to start the game. Let's implement that in our onSuccess method.

Our startRound method will start our game for us, so let's add that to onSuccess

```
public void onSuccess(List<Doggo> doggos) {
   progress.setVisibility(INVISIBLE);
   container.setVisibility(VISIBLE);
   startRound();
}
```

Picking our dogs (TODO: #PICK-DOGS)

Even though we're getting all our data and starting the game, we still aren't seeing anything show up. Our first problem is that we currently aren't picking any dogs to show. Let's fix that.

Open up the **QuestionPresenter** class and look at the **pickDoggos** and **pickCorrectDoggo** methods. At this point we have a list of all dogs and our list randomizer. Let's make use of those to pick our dogs.

Set our picked dogs in pickDoggos:

```
   public void pickDoggos() {
        pickedDoggos = randomizer.pickN(doggos, choiceNumber);
   }
```

Now we have a list of dogs to use for our questions. Each dog includes a breed name and a URL for the image to use. Let's make sure we're loading those images.

This involves two steps:

First, let's add the loadImages call to our pickDoggos method:

```
    public void pickDoggos() {
        pickedDoggos = randomizer.pickN(doggos, choiceNumber);
        loadImages();
    }
```

Right now, loadImages is empty. Let's make sure each dog's image is being sent to the view. We have a reference to the view with our presenter's view field. That view has a method called setImage that takes both an integer (the index) and a String (the image URL). Put a call to setImage inside the loop.

```
private void loadImages() {
   for (int i = 0; i < pickedDoggos.size(); i++) {
       view.setImage(i, pickedDoggos.get(i).getImageUrl());
   }
}
```

Pick the correct dog (TODO: #PICK-CORRECT-DOG)

Progress! If we run the app now, we're getting pictures of dogs showing up. Nothing is happening though. We don't have any title, and all we've done is pick random dogs. Let's make sure the app knows which dog is the *correct dog*.

In our Presenter, we have the pickCorrectDoggo method. You can see that this method is being called from the DoggoFragment with this thing as the integer:

```
savedInstanceState == null ? -1 : savedInstanceState.getInt(KEY_DOGGO)
```

The savedInstanceState is something we get if the app reconstructs itself after a destructive event (you rotate from portrait to landscape, or maybe the app was destroyed because the phone needed the memory your app was taking up). If something like that happened, the **onSaveInstanceState** method saves the correct dog index for us.

That was just a long explanation to let you know that you don't really need to worry about the parameter coming in from this method.

What we need to do now is to make sure that we're setting a correct index if we're not coming back from one of those destructive states. To do this, let's add an else clause to the **pickDoggo** method. We'll use the randomizer to get a random index.

```
public void pickCorrectDoggo(int savedIndex) {
  if (savedIndex >= 0) {
     correctIndex = savedIndex;
  } else {
     correctIndex = randomizer.pickOneIndex(pickedDoggos);
  }
}
```

Or use a ternary operator for brevity

```
public void pickCorrectDoggo(int savedIndex) {
   correctIndex = savedIndex >= 0 ? savedIndex : randomizer.pickOneIndex(pickedDoggos);
}
```

Alright, we've got the correct index now. Let's update our title with the correct breed name so we know what breed we're looking for. The view's **setTitle** method takes a String for the breed name.

```
public void pickCorrectDoggo(int savedIndex) {
   if (savedIndex >= 0) {
```

```
        correctIndex = savedIndex;
    } else {
        correctIndex = randomizer.pickOneIndex(pickedDoggos);
    }
    view.setTitle(pickedDoggos.get(correctIndex).getBreedName());
}
```

Now the title updates with one of the breed names in our list!

Checking answers (TODO: #CHECK-ANSWER)

We still aren't getting any feedback whenever we tap an image. Looking at our code in DoggoFragment, we've set a click listener on each face in our **onViewCreated** method:

```
for (int i = 0; i < container.getChildCount(); i++) {
    final int index = i;
    ImageView face = (ImageView) container.getChildAt(index);
    faces.add(face);
    face.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            presenter.checkAnswer(index);
        }
    });
}
```

Looks like we're making a call to the presenter's **checkAnswer** method whenever a face is clicked.

Right now **checkAnswer** is just incrementing our **tries** variable. Let's implement some logic to check the correct answer and update whether that was the correct answer or not.

We have a parameter in this method - the index that was clicked. Let's check that against our **correctIndex**

```
public void checkAnswer(int index) {
    tries++;
```

```
    if (index == correctIndex) {
    } else {
    }
}
```

If the index isn't the correct index, then we should update the view and let it know that the wrong answer was chosen. Update the view by calling the **setWrongAnswer** method with the index clicked

```
public void checkAnswer(int index) {
    tries++;
    if (index == correctIndex) {
    } else {
        view.setWrongAnswer(index);
    }
}
```

We also have the **setCorrectAnswer** method, so let's use that for a correct answer

```
public void checkAnswer(int index) {
    tries++;
    if (index == correctIndex) {
        view.setCorrectAnswer(index);
    } else {
        view.setWrongAnswer(index);
    }
}
```

Try running the app now. Our screen will load up and clicking on our images will now show us whether we were right or not. The images and the title will fade out after the correct answer...but then nothing.

Let's do one last thing to get the rounds fully functional.

Start the second round (TODO: #ROUND-COMPLETE)

In the DoggoFragment, you might have seen an interface called **Listener**. It's a pretty simple interface:

```
public interface Listener {
   void onRoundComplete();
}
```

We register a listener in the fragment's **onAttach** method and unregister it in **onDetach**. That listener ends up being the **DoggoActivity**.

Right now, **onRoundComplete** in **DoggoActivity** does...nothing. Let's make it go to the next round with our **goToNextRound** method.

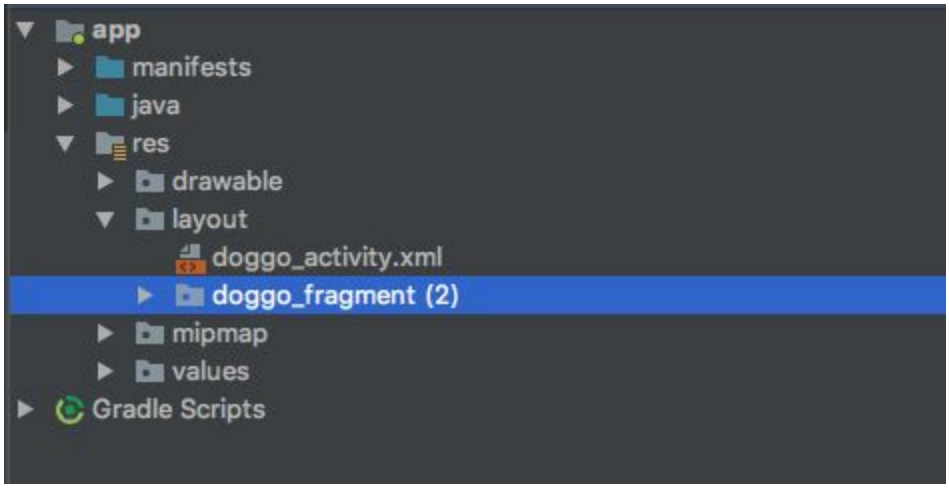
```
public void onRoundComplete() {
   goToNextRound();
}
```

Let's try running our app now. We start our initial round, answer the question, and...it starts a new round! Our app is functional at this point, but there's still a lot of things you could do to improve it.

Add a sixth image (Optional) (TODO: #ADD-IMAGE)

We have five faces in the app already, but what if we wanted to add a sixth? It involves a change to our layout file, but it'll also involve changes to our logic so that we always populate that sixth doggo.

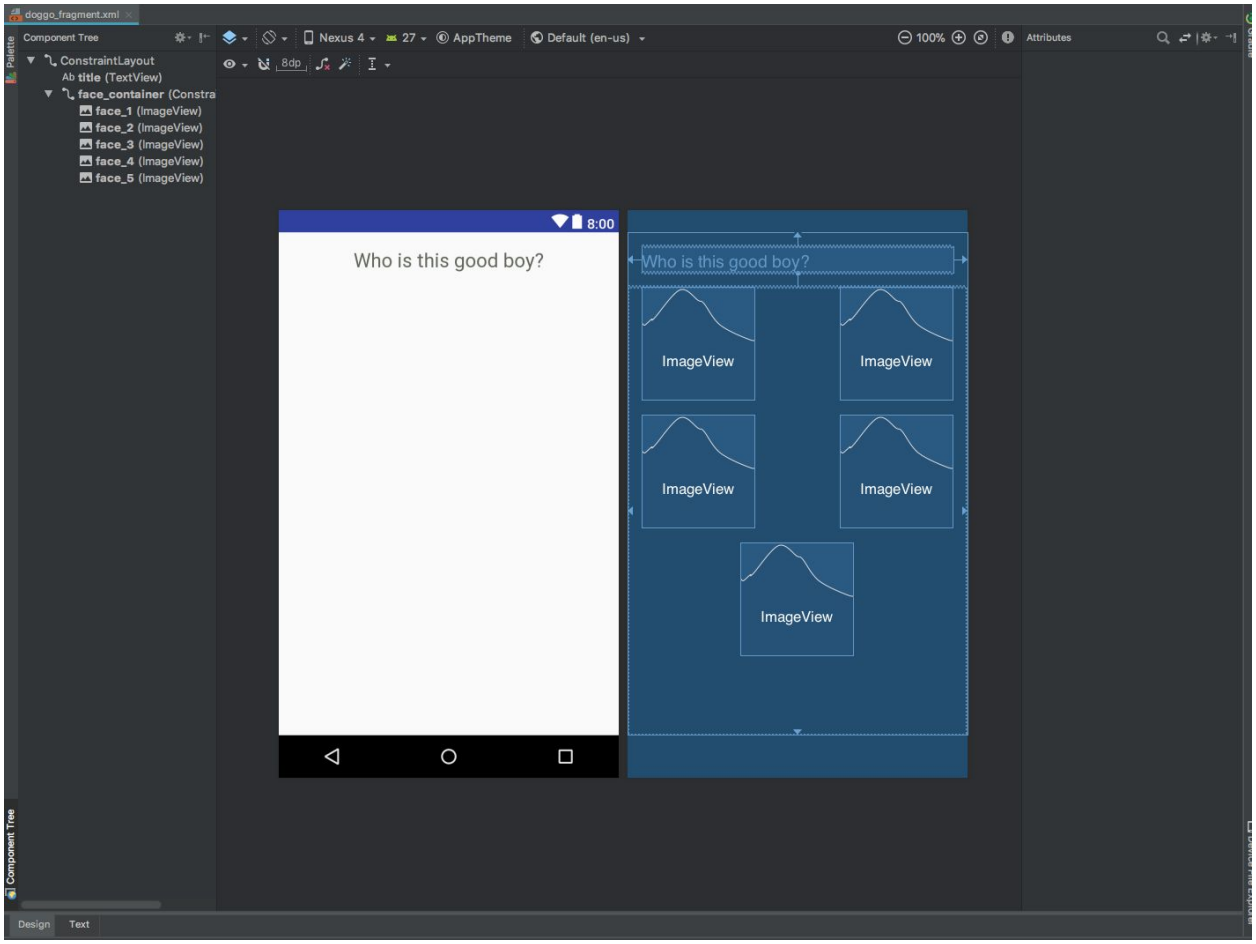
Our first step is to open up our layout file. These are found in **app > res > layout**. All our layout files are written in XML. You'll see two of them to start with - one is a .xml file named **doggo_activity.xml** and the other is a folder called **doggo_fragment**.



If you open the doggo_fragment folder, you'll see two files that look to be named the exact same except that one of them has **(land**) after its name. In Android we have multiple buckets that layouts can fit under. If you only specify one layout file, then that layout file is used everywhere. If you put a layout file in the **land/** folder though, then you'll have two separate layout files - one for portrait and one for **land**scape. There are also other buckets like **w600dp** which would be for something 600 DP (device independent pixels) wide, such as a tablet. These can cross as well, so we can have a bucket that's **both** landscape and 600 DP wide.

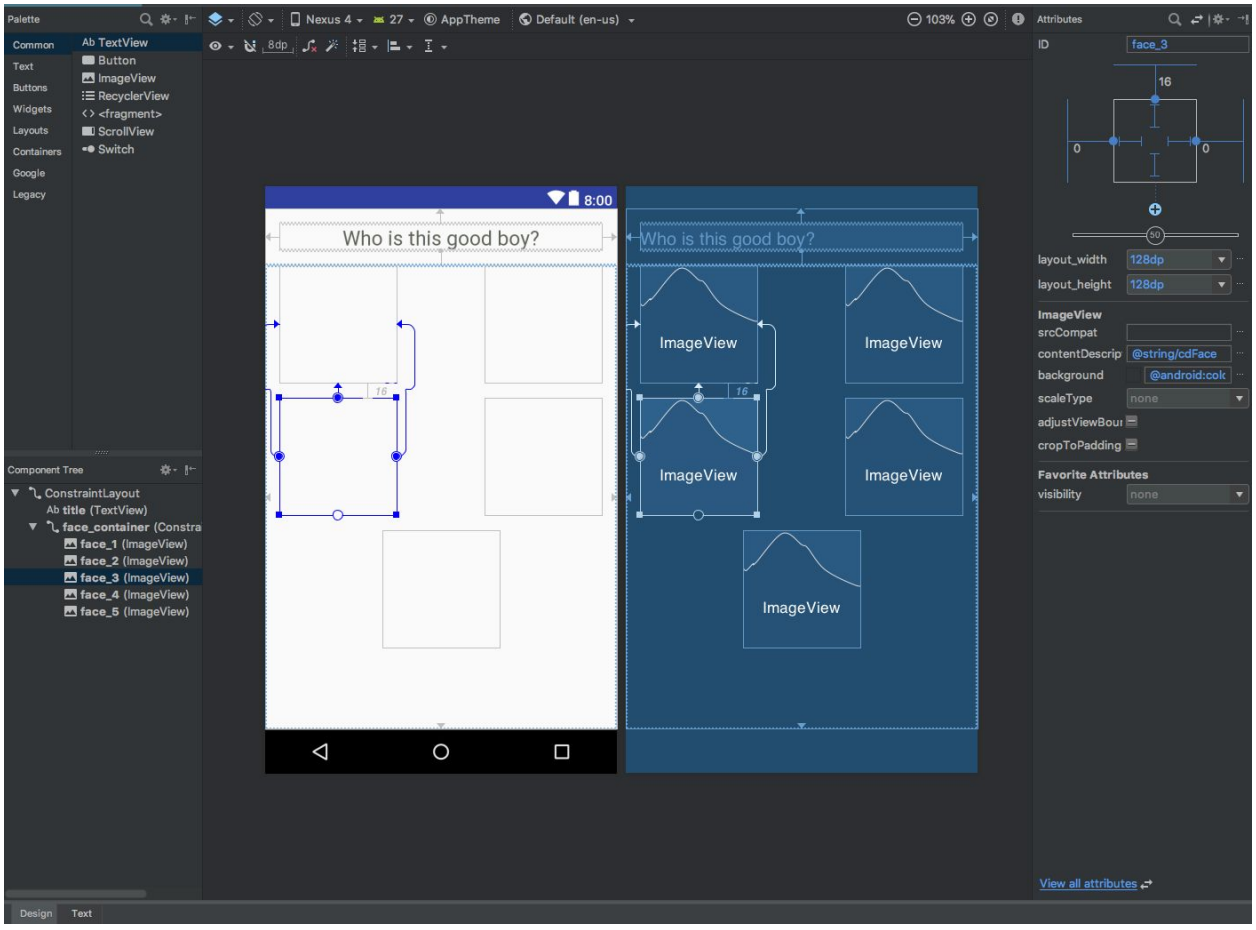
For now, concentrate on the default file, **doggo_fragment.xml** and leave the landscape version alone. If you're feeling adventurous, supporting landscape is a fun road to go down.

On opening our XML file, we see the layout editor - a tool to give us an idea of what our screen looks like before we try it out.



The **Component Tree** on the left shows us all the widgets in our view, and clicking on a widget will highlight its name in the component tree (and vice versa).

In the bottom left of this view, we can see two tabs: **Design** and **Text**. The Design view is what we're looking at right now. If we click the **Palette** button, we'll also see a list of common widgets



TextViews, Buttons, ImageViews...all common Android widgets for things like displaying text or images. We can click and hold on one of the widgets and drag it on to the layout and it'll plop right down. If you place one on the mock up and want to delete it, just click on the widget and hit Delete and it'll go away.

For now, let's go into our XML by hitting the **Text** tab in the bottom left. All Android layouts boil down to a bunch of XML. Each widget has different attributes - some (like **layout_width** and **layout_height**) are required, while some aren't. Some are unique to certain widget types (like text in a TextView, or an image source in an ImageView).

This layout file uses a **ConstraintLayout**. Basically, it positions widgets on the screen relative to something else. References to **parent** refer to whatever the widget is inside, so when the **face1** ImageView refers to the parent, it's referring to the **face_container** ConstraintLayout.

The documents have a lot of resources on getting started with layouts, like

Declaring Layouts,
Building a UI with the Layout Editor, and, specifically for this exercise,
ConstraintLayouts
For now, to add a sixth face, let's shift things around. Faces 1-4 are positioned fine as is. Face 5 is positioned below face 3, and halfway between both sides of the screen.

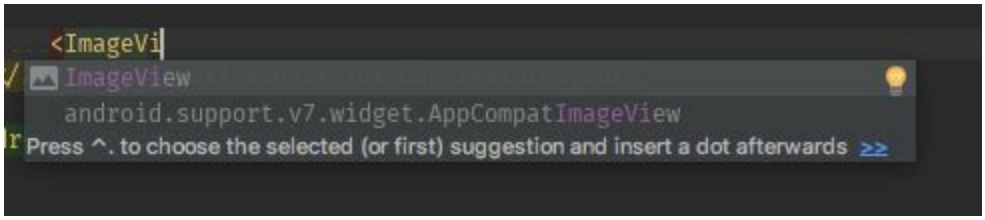
If we want to add a sixth face, we'll want to put face 5 below face 3. Luckily it's already positioned that way thanks to the `app:layout_constraintTop_toBottomOf="@id/face_3"` attribute. Let's make sure it lines up with the start of face 3 as well. Face 3 is already aligned to the start of face 1 (due to the attribute `app:layout_constraintStart_toStartOf="@id/face_1"` being set), so we just need to line up face 5 with face 1. Add the following line inside the XML for face_5:

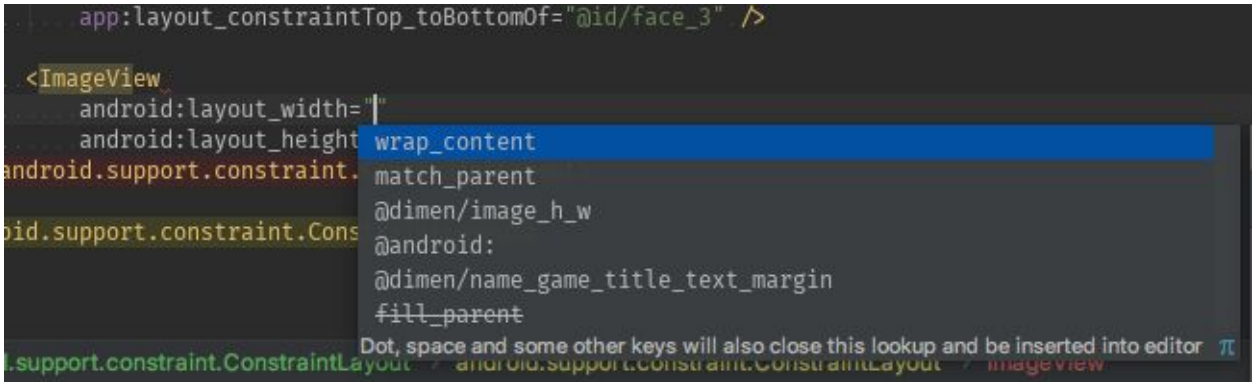
```
app:layout_constraintStart_toStartOf="@id/face_1"
```

Now your widget for face_5 should look like this:

```
<ImageView
    android:id="@+id/face_5"
    android:layout_width="128dp"
    android:layout_height="128dp"
    android:layout_marginTop="16dp"
    android:background="@android:color/transparent"
    android:contentDescription="A face"
    app:layout_constraintStart_toStartOf="@id/face_1"
    app:layout_constraintTop_toBottomOf="@id/face_3" />
```

Let's add our sixth face now. Below face_5, start typing **<ImageView** and the IDE should prompt you to autocomplete. Hit enter and you'll see a couple things that look like this:





For the layout_width and layout_height attributes, put **@dimen/image_h_w**. This refers to another folder found in **app > res > values > dimens** (for now, ignore the w820dp bucket). This is where we specify dimensions that we can reuse throughout the app. The **image_h_w** dimension is noted as 128dp - 128 device independent pixels. DP are useful in having a uniform measurement, to where 1 DP will be the same relative amount of space regardless of device.

Your ImageView should look like this now:

```
<ImageView
    android:layout_width="@dimen/image_h_w"
    android:layout_height="@dimen/image_h_w" />
```

Next, let's add an ID. IDs are useful in creating references to our widgets. It's how we can manipulate our TextViews, or enable our buttons in code.

Insert the line

```
android:id="@+id/face_6"
```

Into the ImageView. You'll see some ids marked with **@id** and others marked with **@+id**. The plus sign is just a shortcut to add the ID to a list that Android keeps of all the IDs we have so far. All these Ids get turned into integers in something called the **R file**.

Next, let's add a margin so that the top of our new face isn't squished right up against whatever's above it. Add the line

```
android:layout_marginTop="@dimen/name_game_title_text_margin"
```

In our face_6 ImageView. This is yet another dimension we defined in our dimens.xml file. Now your new ImageView should look something like this:

```
<ImageView
    android:id="@+id/face_6"
    android:layout_width="@dimen/image_h_w"
    android:layout_height="@dimen/image_h_w"
    android:layout_marginTop="@dimen/name_game_title_text_margin" />
```

Now let's give it a background and a description. The background defines what the background looks like - it could be another image or it could just be a color. In this case, we'll define it to be an Android property for a transparent background.
Add

```
android:background="@android:color/transparent"
```

to your ImageView. Additionally, add a contentDescription. This is an accessibility feature that will let people with impaired vision have their phone read to them what an image is. Add

```
android:contentDescription="@string/cdFace"
```

to the ImageView.

Finally, let's add our constraints so that we can position this face in the right spot. Add these lines to the ImageView:

```
app:layout_constraintStart_toStartOf="@id/face_2"
app:layout_constraintEnd_toEndOf="@id/face_2"
app:layout_constraintTop_toBottomOf="@id/face_4"
```

These lines will position our last face below face 4 and aligned with face 2. If we look in the Design editor, our face_6 widget is just where it should be.

Now we need to update our code. This should be fairly quick, since our DoggoFragment will adjust automatically, as long as all the ImageViews are inside the face_container. In the **QuestionPresenter**, let's update our **choiceNumber** variable to be 6 rather than 5.

Now if we run the app, we should see six faces show up instead of the previous 5.

BONUS (Optional)

Implement a launch screen

Instead of launching straight into the game, create a launch page. This would involve creating a new activity and navigating to another activity.

Keep score and implement a results page

The only thing the app really keeps track of right now is how many tries a user takes per question, and that doesn't persist from question to question. Try to keep score of how many tries it took, or even completely switch things around and only allow the user to guess once per question.