Exploring View Binding on Android

joebirch.co/2019/09/18/exploring-view-binding-on-android

hitherejoebirch

18th Sep 2019

When it comes to manipulating our user interfaces within Android applications, there are a couple of approaches that we can take. In these cases, we need to obtain a reference these views in-order to manipulate them in some way. For this, we'll either use findViewById(), followed by casting the view to the corresponding type. Or if we're using kotlin, then we'll use the kotlin android extensions to perform synthetic access to the views from our layout files. Whilst these two are perfectly fine approaches to take, they do come with some possible pitfalls:

> When using findViewById, the casting of view types is not type safe. This means that whilst we may think that we are accessing a specific type of view that we have casted to, the view we are referencing from our layout could actually be of a different type. This is more common to happen than you may think – maybe it's a simple error you've made in a file you're familiar with or an incorrect reference to the wrong view component from your XML file, with casting in place this will cause a class cast exception.
>
> Whether we are referencing views using findViewId or kotlin synthetics, these approaches are not null safe. This means that if a view is not available yet within a layout, or we are using an incorrect ID that does not exist within the layout of our activity/fragment, then a null pointer exception will be thrown as the referenced view is not available for use at the time of access.
>
> In some cases, we may be using multiple layout files for a single screen, this may be to handle different layout configurations such as a landscape and portrait layouts. In these cases we may have different view components displayed, meaning that a component that is available within the landscape layout may not be available in the portrait layout. So for example, if we are accessing a view in our portrait layout, and then switch to our landscape layout where that view is not available and try to access it, we will see a null pointer exception as at compile time and runtime we may not be aware that the view in question is nullable.

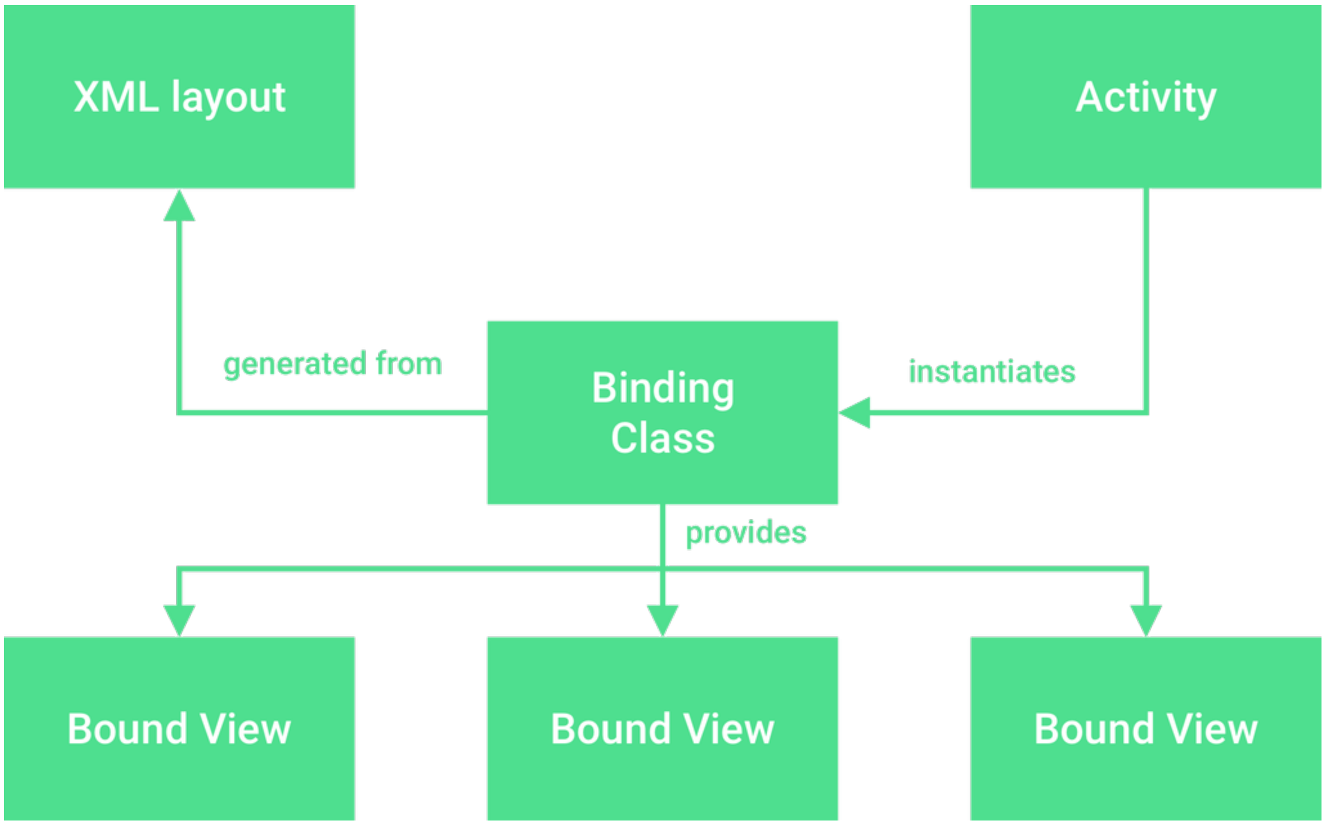
To alleviate the above issues, View Binding introduces us to Binding classes that will bind the views defined in our XML layout to a generated class which can be used within our activities / fragments. Using this generated class we can then access the views available from the binding, giving us some of the following advantages. **Note:** It's important to be aware that this View Binding is different from Data Binding – we do not use ViewBinding to bind layouts with data in XML.

> The bound views within our binding class are generated with their corresponding type, meaning that any references made to views through this class are completely type safe. This removes any chance of incorrect casting being carried out, removing the risk of running into class cast exceptions.
>
> Because the binding class is generated based off of the attached layout file, all views within this class are null safe and available to access from our activity / fragment. This means that we are unable to reference incorrect IDs from other layouts or views that may not be available, meaning that we can access views without the worry of causing null pointer exceptions.

> The View Binding classes have the ability to mark view references as @Nullable, in the case that a view may be not be available in all layout configurations. This means that we can access views with the awareness that the view may be null, allowing us to take the appropriate precautions when accessing the specified view.

With the above in mind, we can see that View Binding provides us with an approach that makes the accessing of views less error prone – allowing us to reduce the chances of crashes occuring for our users.

When it comes to View Binding, there are a couple of concepts which come together in-order to provide this functionality. To begin with we'll be working the same within our XML layouts, the main difference is that we need to declare a class inside of our activity that will create the binding that create the relationship between



Note: You'll need to be using at least Android Studio 3.6 Canary 11 to use View Binding

To get started with adding View Binding to our android application, we need to go ahead and add the following to the corresponding build.gradle file:

```
android {
  ...
  viewBinding {
      enabled = true
  }
}
```

View Binding is module specific, so you will need to add this to each module that you wish to provide view binding for. If you are going to be using this in every module, then you can place this inside of the project build.gradle file in a manner that allows you to reuse it throughout your project.

When it comes to our XML files, we don't actually need to do anything differently to be able to access views from the generated binding classes – most of the changes will come from within the classes that are accessing those bindings. Let's say we have a layout file for an activity called add_profile.xml, which looks a little something like this:

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

  <TextView
     android:id="@+id/text_title"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     app:layout_constraintBottom_toTopOf=
       "@+id/button_authenticate"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />

  <Button
     android:id="@+id/button_add_profile"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:padding="16dp"
     android:text="@string/label_authenticate"
     android:layout_marginBottom="24dp"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent" />

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

We're going to want to access these views from within our activity so that we can assign the corresponding content to them. To do this we're going to need to declare a binding class for that layout file. When we want to declare this binding class we need to adhere to the naming convention defined by the View Binding API – so for example, for add_profile.xml this will look like so:

```
 private lateinit var binding:
AddProfileBinding
```

To begin with, the API will take the name of our layout file, remove any underscores and then sentance-case the words that were seperated by these underscores. Finally, you can see here that the word `Binding` has been added to end of our file name – this is required and will be added for every view binding class that is generated. Now that we have this binding class declared, we need to assign a reference to it. For this, we're going to use the LayoutInflator reference within our activity and call the static inflate() method on our Binding class. Doing so will inflate our layout into our binding class so that our bound views are accessible for use.

```
@Override
fun onCreate(savedInstanceState: Bundle) {
  super.onCreate(savedInstanceState)
  binding = AddProfileBinding.inflate(layoutInflater)
  setContentView(binding.root)
}
```

You'll also notice here that we are setting the content view of our screen to this root property from the binding. Every binding that is created has a root – this root represents the root component within the layout file that we inflated into the binding class. This is a convenient way for us to be able to finalise the display of our screen.

At this point we have our binding class and can finally use it to setup our screen. Looking at the previous add_profile.xml file from above, you may recall seeing a textview and button. We'll now access these from our binding class and configure them for display:

```
binding.textTitle.text = getString(R.string.some_string)
  binding.buttonAddProfile.setOnClickListener {
  // do something
}
```

You can see here that we're accessing the components that are defined within our layout file (with the underscores removed and camelcase applied). Regardless of the views in our XML, the way we access them here will apply for all view types. Here you can see no casting in place – when we access the textTitle field, that is referenced as a TextView as that's what the binding class has allocated it. The same applies for the addProfileButton – without any casting in place we avoid incorrect castings being made, as well resulting in code that is more readable. As well as this, we get the advantages mentioned previously when it comes to view binding. Another point being nullable views, allowing us to handle these cases appropriately:

```
binding.buttonAddProfile?.setOnClickListener {
// do something
}
```

In some cases there may be components that you do not wish to create bindings for – maybe you do not want them available for manipulation from with the corresponding binding class. Here, you can make use of the viewBindingIgnore attribute from within our XML layout to exclude this view from being added to the generated binding class.

```
<TextView
  ...
  tools:viewBindingIgnore="true"
/>
```

Whilst this has been a quick dive into View Binding on Android, I hope that it’s been enough to demonstrate the advantages that this approach to view manipulation gives us in our applications. Whether it’s type safety, null safety or an overall cleaner approach to view references in our code, View Binding offers functionality that all applications will be able to take advantage of.

Have you tried View Binding yet? Or have any thoughts / questions on the content outlined here? If so, I would love to hear from you!

Follow @hitherejoe