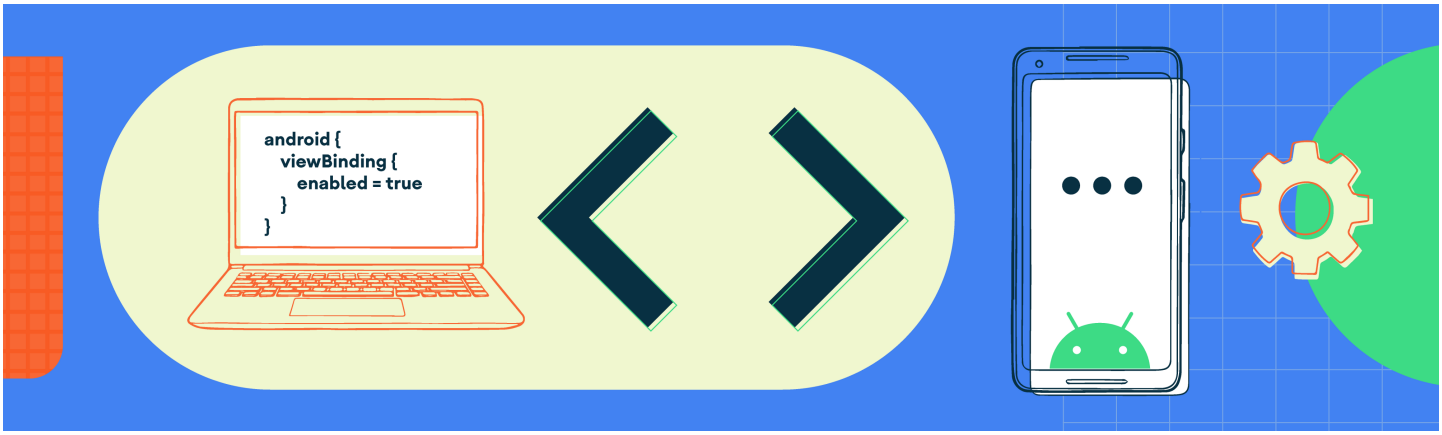



# Use a vinculação de visualização para substituir findViewById

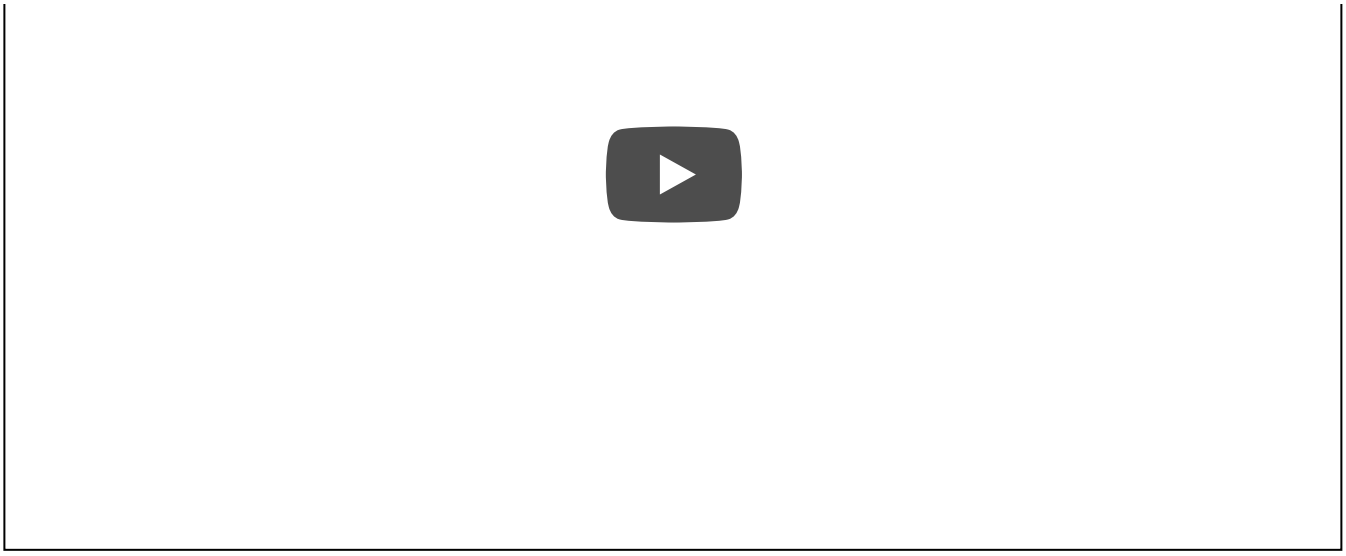
Sean McQuillan
13 de fevereiro de 2020 · 6 min de leitura

Novo no Android Studio 3.6, a vinculação de visualização oferece a capacidade de substituir `findViewById` por objetos de vinculação gerados para simplificar o código, remover bugs e evitar todos os clichês de `findViewById`.

## TL; DR

- Ative a vinculação de visualização em `build.gradle` (sem dependências de bibliotecas)
- A vinculação de visão gera um objeto de vinculação para cada layout em seu módulo ( `activity_awesome.xml` → `ActivityAwesomeBinding.java` )
- O objeto Binding contém uma propriedade para cada visualização com um id no layout - com o tipo correto e segurança nula
- Suporte total para a **linguagem de programação Java** e **Kotlin**

> Android Jetpack: Replace findViewById with view binding



## Atualize `build.gradle` para ativar a vinculação de visualização

Você não precisa incluir nenhuma biblioteca extra para habilitar a associação da visão. Ele está integrado ao plug-in do Android Gradle, começando com as versões enviadas no Android Studio 3.6. Para ativar a vinculação de visualização, configure `viewBinding` em seu `build.gradle` arquivo de nível de módulo .

```
// Disponível no Android Gradle Plugin 3.6.0
android {
    viewBinding {
        enabled = true
    }
}
```

No Android Studio 4.0, `viewBinding` foi movido para `buildFeatures` [ notas de versão ] e você deve usar:

```
// Android Studio 4.0
android {
    buildFeatures {
        viewBinding = true
    }
}
```

Once enabled for a project, view binding will generate a binding class for *all* of your layouts automatically. You don't have to make changes to your XML — it'll automatically work with your existing layouts.

> View binding works with your existing XML, and will generate a binding object for each layout in a module.

You can use the binding class whenever you inflate layouts such as `Fragment`, `Activity`, or even a RecyclerView `Adapter` (or `ViewHolder`).

### Use view binding in an Activity

If you have a layout called `activity_awesome.xml`, which contains a button and two text views, view binding generates a small class called `ActivityAwesomeBinding` that contains a property for every view with an ID in the layout.

You don't have to call `findViewById` when using view binding — instead just use the properties provided to reference any view in the layout with an id.

The root element of the layout is always stored in a property called `root` which is generated automatically for you. In an `Activity`'s `onCreate` method you pass `root` to `setContentView` to tell the Activity to use the layout from the binding object.

> ***Easy Mistake:*** *Calling `setContentView(…)` with the layout resource id instead of the inflated binding object is an easy mistake to make. This causes the layout to be inflated twice and listeners to be installed on the wrong layout object.*
>
> ***Solution:*** *When using view binding in an `Activity`, you should always pass the layout from the binding object with `setContentView(binding.root)`.*

## Safe code using binding objects

`findViewById` is the source of many user-facing bugs in Android. It's easy to pass an id that's not in the current layout — producing `null` and a crash. And, since it doesn't have any type-safety built in it's easy to ship code that calls `findViewById<TextView>(R.id.image)`. View binding replaces `findViewById` with a concise, safe alternative.

View bindings are…

- **Type-safe** because properties are *always* correctly typed based on the views in the layout. So if you put a `TextView` in the layout, view binding will expose a `TextView` property.
- **Null-safe** for layouts defined in multiple configurations. View binding will detect if a view is only present in some configurations and create a `@Nullable` property.

And since the generated binding classes are regular Java classes with Kotlin-friendly annotations, you can use view binding from both the Java programming language and Kotlin.

## What code does it generate?

View binding generates a Java class that replaces the need for `findViewById` in your code. It will generate one binding object for every XML layout in your module while mapping names so `activity_awesome.xml` maps to `ActivityAwesomeBinding.java`.

When editing an XML layout in Android Studio, code generation will be optimized to *only* update the binding object related to that XML file, and it will do so in memory to make things fast. This means that changes to the binding object are available immediately in the editor and you don't have to wait for a full rebuild.

> Android Studio is optimized to update the binding objects immediately when editing XML layouts.

Let's step through the generated code for the example XML layout from earlier in this post to learn what view binding generates.

Properties generated by view binding are correctly typed and null-safe.

View binding will generate one correctly-typed property for each view with a specified `id`. It will also generate a property called `rootView` that's exposed via a getter `getRoot`. View binding doesn't do any logic– it just exposes your views in a binding object so you can wire them up without error-prone calls to `findViewById`. This keeps the generated file simple (and avoids slowing down builds).

If you're using Kotlin, this class is optimized for interoperability. Since all properties are annotated with `@Nullable` or `@NonNull` Kotlin knows how to expose them as null-safe types. To learn more about interop between the languages, check out the documentation for calling Java from Kotlin.

View binding generates an inflate method, which is the primary way to make new binding objects.

In `ActivityAwesomeBinding.java`, view binding generates a public `inflate` method. The one argument version passes `null` as the parent view and doesn't attach to parent. View binding also exposes a three argument version of `inflate` that lets you pass the `parent` and `attachToParent` parameters when needed.

The call to `bind` is where the magic happens. It will take the inflated layout and bind all of the properties, with some error checking added to generate readable error messages.

Simplified version of the generated public bind method.

The `bind` method is the most complex code in the generated binding object, with a call to `findViewById` for each view to bind. And here you can see the magic happen – since the compiler can check the types and potential nullability of each property directly from the XML layouts it can safely call `findViewById`.

Note, the actual generated code for the `bind` method is longer and uses a labeled break to optimize bytecode. Check out this post by Jake Wharton to learn more about the optimizations applied.

On each binding class, view binding exposes three public static functions to create a binding an object, here's a quick guide for when to use each:

- **`inflate(inflater)`** – Use this in an `Activity onCreate` where there is no parent view to pass to the binding object.
- **`inflate(inflater, parent, attachToParent)`** – Use this in a `Fragment` or a RecyclerView `Adapter` (or `ViewHolder`) where you need to pass the `parent ViewGroup` to the binding object.
- **`bind(rootView)`** – Use this when you've already inflated the view and you just want to use view binding to avoid `findViewById`. This is useful for fitting view binding into your existing infrastructure and when refactoring code to use `ViewBinding`.

## What about included layouts

One binding object will be generated for each `layout.xml` in a module. This is true even when another layout `<include>`s this this layout.

Example of include tag in view binding. Note that the <include> tag has an id!

In the case of included layouts, view binding will create a reference to the included layout's binding object.

Note that the `<include>` tag has an id: `android:id="@+id/includes"` . This is required for view binding to generate a property (just like a normal view).

> Include tags must have an id to generate a binding property.

View binding will generate a reference to the `IncludedButtonsBinding` object in `ActivityAwesomeBinding`.

### Using view binding and data binding

View binding is only a replacement for `findViewById`. If you also want to automatically bind views in XML you can use the data binding library. Both libraries can be applied to the same module and they'll work together.

When both are enabled, layouts that use a `<layout>` tag will use data binding to generate binding objects. All other layouts will use view binding to generate binding objects.

> You can use data binding and view binding in the same module.

We developed view binding in addition to data binding because many developers provided feedback that they wanted a lighter weight solution to replace `findViewById` without the rest of the data binding library – and view binding provides that solution.

### View binding and Kotlin synthetics or ButterKnife

One of the most common questions asked about view binding is, “Should I use view binding instead of Kotlin synthetics or ButterKnife?” Both of these libraries are used successfully by many apps and solve the same problem.

For most apps we recommend trying out view binding instead of these libraries because view binding provides safer, more concise view lookup.



Comparing features of View binding, ButterKnife, and Kotlin synthetics.

> *Embora ButterKnife valide anulável / não nulo em tempo de execução, o compilador não verifica se você combinou corretamente o que está em seus layouts*

## Recomendamos experimentar a vinculação de visualização para uma consulta de visualização segura e concisa.

### Saber mais

Para saber mais sobre a vinculação de visualizações, verifique a documentação oficial .

E adoraríamos ouvir suas experiências com a biblioteca #ViewBinding no Twitter!

Agradecimentos a Jake Wharton e Jose Alcérreca .

Android Ver encadernação Ligação de dados UI Findviewbyid