


Creating a Custom View in Android

 blogs.oregonstate.edu/workla/2021/11/04/creating-a-custom-view-in-android/

Let's say you are making a project that has a certain set of views that are often repeated. For example, maybe you are placing a name and address combo in many locations in your project. Currently, if just using the stock views within Android, you may end up repeating the same set of views in your XML files to account for this – a textview for name, a textview for street address, a text view for city etc. What if, instead, you wanted to simply enter a view for address info and then provide it with the information and it automatically creates and fills in all of the pieces? This is a perfect example where custom views can help keep your project D.R.Y.

What are Custom Views?

A custom view is a view that you design in any way you want by extending another view and then overwriting everything you feel necessary. This custom view can then be used in code, and everything defined in it will be inserted wherever it is used. Maybe a stock dialog isn't the layout you want for all the dialogs in your project? Create a custom dialog you can then call instead.

Let's walk through the steps of creating a custom view with an example. I am going to create a custom view that contains a user info. This view could then be used anywhere I want to display my user. In my example, the info I need to display includes a username, user level and a logo.

Step 1: Decide what to extend

You can extend from any view/layout, but think of picking a view that provides some functionality you may want [1]. For example, if you are going to want a card, extend card to start and then customize the corners, elevation, content etc so you don't have to start from scratch. For my example, I am going to extend a linear layout to hold my stuff, as that functionality is all the functionality I need from my parent class.

Step 2: Create layout file

In your layout file, you specify all the details just like you would if you weren't going to have a custom view. The benefit to this custom view is that now this will only be in this one place instead of repeated everywhere you want to use it. Here is my very simple example layout file:

```

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

    <ImageView
        android:id="@+id/logo"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_margin="24dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        android:contentDescription="@string/contentDescripLogo"
        tools:src="@drawable/placeholder_image"/>

    <TextView
        android:id="@+id/username"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_margin="24dp"
        android:textStyle="bold"
        android:textSize="20sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/logo"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="testUser" />

    <TextView
        android:id="@+id/levelTitle"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:layout_marginTop="24dp"
        android:layout_marginStart="24dp"
        android:layout_marginBottom="24dp"
        app:layout_constraintStart_toEndOf="@+id/logo"
        app:layout_constraintTop_toBottomOf="@+id/username"
        app:layout_constraintBottom_toBottomOf="parent"
        android:text="@string/levelTitleText" />

    <TextView
        android:id="@+id/level"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:layout_marginTop="24dp"
        android:layout_marginStart="8dp"
        android:layout_marginBottom="24dp"
        android:layout_marginEnd="24dp"
        app:layout_constraintStart_toEndOf="@+id/levelTitle"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="@+id/levelTitle"
        app:layout_constraintBottom_toBottomOf="@+id/levelTitle"
        tools:text="Grand Master"/>

</androidx.constraintlayout.widget.ConstraintLayout>

```

Step 3: Create a child class extending some view

In my case, I will create the UserInfoView class and extend linear layout.

```

package com.example.customviewexample

import android.content.Context
import android.graphics.drawable.Drawable
import android.util.AttributeSet
import android.view.LayoutInflater
import android.widget.ImageView
import android.widget.LinearLayout
import android.widget.TextView
import androidx.appcompat.content.res.AppCompatResources

class UserInfoView @JvmOverloads constructor(context: Context, attrs: AttributeSet? = null,
    defStyleAttr: Int = 0, defStyleRes: Int = 0): LinearLayout(context, attrs, defStyleAttr, defStyleRes) {

    private var logoView: ImageView
    private var usernameView: TextView
    private var levelView: TextView

    var username: String = ""
        set(value) {
            field = value
            usernameView.text = value
        }

    var level: String = ""
        set(value) {
            field = value
            levelView.text = value
        }

    var logo: Drawable? = null
        set(value) {
            field = value
            value?.let { it: Drawable
                logoView.setImageDrawable(it)
            }
        }

    init {
        LayoutInflater.from(context).inflate(R.layout.custom_view, root: this, attachToRoot: true)
        orientation = VERTICAL

        //get views we will add data for
        logoView = findViewById(R.id.logo)
        usernameView = findViewById(R.id.username)
        levelView = findViewById(R.id.level)

        //fill views with custom attributes
        attrs?.let { it: AttributeSet
            val typedArray = context.obtainStyledAttributes(it, R.styleable.UserInfoView)

            username = resources.getText(typedArray.getResourceId(R.styleable.UserInfoView_username, R.string.defaultString)).toString()
            level = resources.getText(typedArray.getResourceId(R.styleable.UserInfoView_level, R.string.defaultString)).toString()
            val logoRes = typedArray.getResourceId(R.styleable.UserInfoView_logoRef, defValue: -1)
            if (logoRes != -1) {
                logo = AppCompatResources.getDrawable(context, logoRes)
            }

            typedArray.recycle()
        }
    }
}

```

Within this class I have basically inflated my custom view's XML file, and then set up the ability to set the imageview's image and textviews' texts in both code or via custom XML attributes. Let's talk about both of these:

Step 4a: Create Custom attributes

To create custom attributes, we declare a styleable (customarily this would live in a res/values/attrs.xml). Place each attribute name in here and then declare what type you are passing in. For example, the username will be a string and the logoRef will be a reference to a drawable.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="UserInfoView">
        <attr name="username" format="string" />
        <attr name="level" format="string" />
        <attr name="logoRef" format="reference" />
    </declare-styleable>
</resources>
```

Next, look in the above picture of my UserInfoView class and note the init function. To access these attributes in our class, we obtain the styled attributes [2]. If attributes are not null, then we can get text and resource by referencing the given name from the styleable and assigning those to wherever we are going to use them in our custom view. One last note, when done with your styled attribute access, remember to call .recycle()

Step 4b: Create variables to dynamically update view in code

The UserInfoView class shows three variables that can be used to dynamically set data for the custom view: username, logo, and level. Custom setters were set up so that when somebody assigns a value to these variables that the resulting textviews and imageview will be updated accordingly.

Step 5: Add custom view to our project

To add this custom view, we can simply use it like any other view in our resource file. In this example program, we use it twice.

```

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

    <TextView
        android:id="@+id/meTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:textSize="30sp"
        android:textStyle="bold"
        android:textColor="@color/colorPrimary"
        android:text="@string/thisIsMeText"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <com.example.customviewexample.UserInfoView
        android:id="@+id/meInfo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/meTitle" />

    <TextView
        android:id="@+id/friendTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:textSize="30sp"
        android:textStyle="bold"
        android:textColor="@color/colorPrimary"
        android:text="@string/thisIsMyFriendText"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/meInfo" />

    <com.example.customviewexample.UserInfoView
        android:id="@+id/friendInfo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:username="@string/secondUser"
        app:level="@string/secondLevel"
        app:logoRef="@drawable/person2"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/friendTitle"
        app:layout_constraintBottom_toBottomOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

The first use just shows the basic implementation, while the second shows the use of the custom XML attributes we set up earlier. Notice that to set the username attribute, all we needed to do was add a line in our custom view that said `app:username="{string resource here}"`. The code above from our `UserInfoView` class access and assigns that value.

Now let's look at the MainActivity class and see how we can dynamically update the code.

```
package com.example.customviewexample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.appcompat.content.res.AppCompatResources

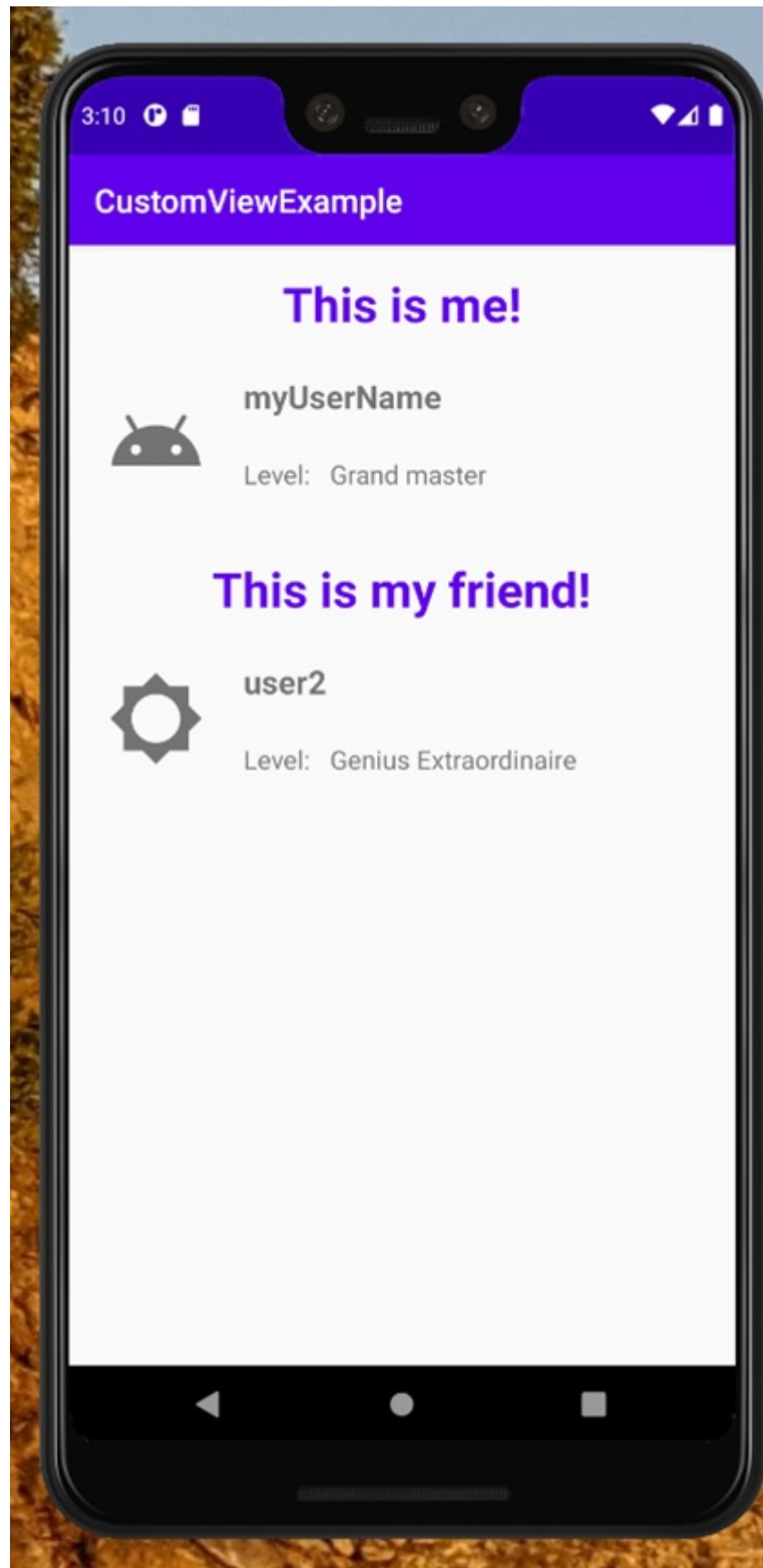
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val meInfo = findViewById<UserInfoView>(R.id.meInfo)
        meInfo.username = "myUserName"
        meInfo.level = "Grand master"
        meInfo.logo = AppCompatResources.getDrawable(context: this, R.drawable.person1)
    }
}
```

Notice for meInfo, we simply find the view, and then have access to the view's .username, .level, and .logo properties to assign a value.

Result

The result of this code is here:



Notice that we got to repeat the set of views with a logo, username and level multiple times without having to copy and paste that set of views in every spot in our main activity layout file. For groups of views that you know you will use a lot, making a custom view for the design can save repeating code and make your code less error prone. And the benefit doesn't stop there. This also allows you to create your own edittext or textview

where you override its normal functionality, create repeated logic-heavy views in one spot, or even a completely unique view when no pre-built view meets your needs [3]. Custom views allow you to create a fully custom app.

Sources:

[1] "Creating a View Class", Oct 27, 2021. Accessed on: Nov 4, 2021. [Online]. Available: <https://developer.android.com/training/custom-views/create-view>

[2] Elye, *Building Custom Components with Kotlin*, Medium, May 27, 2017. Accessed on: Nov 4, 2021. [Online] Available: <https://medium.com/mobile-app-development-publication/building-custom-component-with-kotlin-fc082678b080>

[3] "Custom View Components", Dec 27, 2019. Accessed on: Nov 4, 2021. [Online]. Available: <https://developer.android.com/guide/topics/ui/custom-components>