Declarative UI for Android™ reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Declarative UI for Android™

This extension is complementary to Dataflow for Android™ and adds various features that will allow to create your user interface in a more elegant and performant way.

Essentialy it unifies the expressiveness of XML with the performance of natively constructing your views.

Kotlin™ only

The goal of this library is to make creating view as simple and clear as possible, removing as much boilerplate as possible. Sadly, conciseness is not one of the strong points of Java™, and simply too many kotlin features are needed to make the use of this library worth it. For this reason the use of Kotlin™ is basically mandatory to use this extension. (Yes, in theory you *could* still use Java, as there are no technical limits, but if you do: be warned)

Getting started

Setup

To add this extension all you have to do is add the following dependency:

Gradle

```
implementation 'com.femastudios:dataflow-android-declarativeui:1.0.0'
```

Further instructions on how to configure you Android™ project can be found here.

First look

Here's a quick example on how to create views in your activity:

Kotlin

```kotlin
package com.femastudios.dataflow.android.declarativeui.testapp

import android.os.Bundle
import android.view.Gravity
import androidx.appcompat.app.AppCompatActivity
import com.femastudios.dataflow.android.declarativeui.*
import com.femastudios.dataflow.util.mutableFieldOf

class YourActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent.scrollView.linearLayout(vertical = true) {
            val text = mutableFieldOf("Hi")
            add.text(text) {
                gravity = Gravity.CENTER
                textSize = 40f
            }
            add.linearLayout {
                add {
                    horizontalWeight(1f)
                }.button("Hello") {
                    text.value = "Hello"
                }
                add {
                    horizontalWeight(1f)
                }.button("World") {
                    text.value = "World"
                }
            }

            repeat(100) {
                add.text(it.toString())
            }
        }
    }
}
```

This will create a `ScrollView` that contains a single vertical `LinearLayout`. This layout will then contain a text view that displays the content of the field `text` and another `LinearLayout`, this time horizontal, that contains two buttons that change the value of the field. After that 100 `TextView`s are added, each with a progressive number starting from `0`.

Views prefetching

Creating views is thing done frequently on Android™ apps, that could potentially decrease performance when done too rapidly. For this reason this library uses a technique that we call "View prefetching", that allows us to create the most required views when there is less work to do, between frames. This is achieved using Android's Choreographer.

The class that handles this is called `ViewPrefetcher`. To construct we need a max size, and a function that creates a view, given the context. To obtain a view we can call the `getIntance()` method.

Example:

Kotlin
```kotlin
val myPrefetcher = ViewPrefetcher(5, { context -> MyView(context) })
val myView = myPrefetcher.getInstance()
```

The `ViewPrefetcher` is lazy, and it activates the first time the `getIntance()` method is called.

Predefined prefetchers

A series of prefetchers is already declared in the object `DefaultPrefetchers`. By default they will return the androidx implementation when possible (e.g. `AppCompatTextView`), otherwise the native Android™ implementation. They have a max size relative to the usage commonness.

You should not use those constants directly, but rather use the ones declared in the class `Prefetchers`: the difference is that in this class they are declared as `var`, which means that you can change them if you want to change the default max size or if you want to change the returned implementation. This task is ideally done on app start, before the creation of any view. A good place is your `Application.onCreate()` function.

Example using the default implementation:

Kotlin
```kotlin
Prefetchers.TEXT_VIEW.getInstance() //by default will return an AppCompatTextView
```

Example changing the default prefetcher:

Kotlin

```kotlin
class MyTextView(context : Context) : TextView(context) { /* ... */ }

//In your Application's onCreate:
override fun onCreate() {
    super.onCreate();
    //Change the view prefetcher with our custom max size and implementation
    //Be aware that since the var is declared as ViewPrefetcher<TextView>
    //we won't be aware from outside that it's a MyTextView
    Prefetchers.TEXT_VIEW = ViewPrefetcher(20) { MyTextView(it) }
}

//In the code:
Prefetchers.TEXT_VIEW.getInstance() //will actually return a MyTextView, but the type will
only be TextView
```

Creating views

ViewCreator

To understand how this library works, we must take a look at the `ViewCreator` class, that acts as a proxy to create views. When we construct it we must pass two parameters: the `Context` that will be used to create views, and an `action` that accepts the newly created `View`.

This class contains various properties of a type that extends `ViewHandler`, each one named like a type of view (e.g. `linearLayout`). This `ViewHandler` classes usually implement the invoke operator, so we can use it like a function. As a last parameter it's virtually always present an optional `block` parameter, that is a function that accepts the view as its receiver; it is used to further change some view parameters. Sometimes the invoke operator cannot be used because there are multiple conflicting variants, so it makes more sense to create normal functions.

Each method has the following general structure:

1. Creates the required `View` with the given parameters
2. Calls the passed `block` with the view as its receiver
3. Calls the `action` function that was passed to the constructor
4. Returns the newly created `View`

You can find a list of the views supported by default in the `ViewCreator` documentation. All these view handlers use the the prefetchers declared in `Prefetchers` to create the views.

Creating views

To just create a `View` we can use a the extension property `Context.new`, that returns a `ViewCreator` whose action is empty.

Example:

Kotlin

```kotlin
val f = mutableField("asd")
//This will create a new text view binded to the field f, with a red background color
context.new.text(f) {
    setBackroundColor(Color.RED)
}
```

Adding views

Another default implementation can be found on the extension property ViewGroup.add: this time the action will add each created view to the `ViewGroup` with the default `LayoutParams`.

Example:

Kotlin

```
context.new.linearLayout {
    add.text("Hello world") //This will add a new AppCompatTextView with the text "Hello world" to the linear layout
}
```

Specifying the `LayoutParams`

Since each `ViewGroup` has its own layout parameters, a specialized extension function has been added to every common `ViewGroup`. This functions, still called `add`, accepts a function whose receiver is a specialized `LayoutParams` instance. Also, there are a few extension functions of `LayoutParams` that help changing its values.

Example:

Kotlin

```
context.new.linearLayout(vertical = true) {
    add { //Receiver is LinearLayout.LayoutParams
        matchHeight()
    }.text("Hello world") { //Receiver is AppCompatTextView
        gravity = Gravity.CENTER
    }
    add { //Receiver is LinearLayout.LayoutParams
        verticalWeight(1F)
    }.button("Hi!")
}
```

Base views

Here we'll explore some of the functions to create the most basic views and see how they work.

In these example we'll always omit the `block` parameter since it's the same for all the views. Also, for all views is present a function that can be called with only the `block` function.

Texts

The most basic view that displays some text. The `text` function creates a `TextView`, and accepts either a `CharSequence` or a `Field<CharSequence>`.

Kotlin

```
context.new.text("Hello!")
context.new.text(fieldOf("Hello"))
```

Buttons

There are quite a few functions that create a `Button`. You can either use `button` or `borderlessButton`, that will create a button with the style attribute

`android.R.attr.borderlessButtonStyle`.

The two parameters accepted by this function are:

- `text`: a `CharSequence` or `Field<CharSequence>` that specifies the text of the button
- `onClickListener`: either a kotlin function `(View) -> Unit`, a `View.OnClickListener` or a `Field` of one of them, that specifies the action to perform when the button is clicked

There is also a special version that doesn't accept the `block` parameter, making the click listener able to move outside the parenthesis.

Kotlin

```
context.new.button("Hello!") { //This is the special version with no `block` parameter
    //Do something on click...
}
context.new.borderlessButton(fieldOf("Hello!"), {
    //Do something on click...
}) { /* block */ }
```

Progress bars

To create an indeterminate circle `ProgressBar` we can use `circleProgressBar`.

If we want to create an horizontal progress bar we can use `progressBar`. The parameters are:

- `progress`: a `Field<Int?>` that is bound to the progress of the view; when `null` the progress bar becomes indeterminate
- `max`: an `Int` that represents the max value of the progress bar. Defaults to `100`
- `animate`: whether to animate the changes between progresses (the first change is always not animated). Defaults to `false`

Kotlin

```
val f = mutableFieldOf(50)

context.new.circleProgressBar()
context.new.progressBar(f, 75, animate = true)
context.new.progressBar() //horizontal, indeterminate
```

Images

To create an `ImageView` we can use `imageView`. The parameters are:

- `drawable` or `drawableRes`: a `Drawable`, drawable resource id (`Int`) or `Field` of one of them that represents the image
- `scaleType`: optionally, an `ImageView.ScaleType`. Defaults to `null`.

Kotlin

```
val f = mutableFieldOf(R.drawable.xyz)

context.new.imageView(R.drawable.abc, ImageView.ScaleType.CENTER_CROP)
context.new.imageView(f)
```

Check boxes and switches

Until now we saw views that could only display values, but not change it. For `CheckBox` and `Switch` the user must be able to tap them and change their status: for this reason both `checkBox` and `switch` accept a `MutableField<Boolean>`. They also accept a `text`, similarly to `TextView`.

Kotlin
```
val f = mutableFieldOf(false)

context.new.checkBox("Check me", f)
context.new.switch(fieldOf("Hello"), f)
```

There are also the counterparts function `disabled` that will accept a simple `Field`, but that will create the views disabled, so that the user cannot change them.

Kotlin
```
val f = fieldOf(true)

context.new.checkBox.disabled("You cannot uncheck me!", f)
context.new.switch.disabled(fieldOf("Hi"), f)
```

Edit texts

To create a `EditText` we can use `editText`. Its only parameter is a `MutableField<String>` that will be bound to the content of the edit text.

Kotlin
```
val f = mutableFieldOf("Hello")

context.new.editText(f)
```

Seek bars

`seekBar` is used to create a `SeekBar`. It accepts, like for the progress bar, a `progress` (which this time must be a `MutableField<Int>`) and a `max` value. Like for check boxes and switches, there is a disabled version that can be called using `seekBar.disabled`.

Kotlin
```
val f = mutableFieldOf(50)

context.new.seekBar(f, 75)
context.new.seekBar.disabled(fieldOf(99))
```

Layouts

Linear layouts

`linearLayout` accepts an optional parameter `vertical` that tells it whether to be horizontal or

vertical.

Kotlin
```
context.new.linearLayout(vertical = true) {
    //...
}
```

Relative layouts

`relativeLayout` doesn't accept any additional parameters.

Kotlin
```
context.new.relativeLayout {
    //...
}
```

Scroll views and frame layouts

`scrollView` and `frameLayout` can accept a `View` that will automatically be added as their only child.

Kotlin
```
context.new.scrollView(context.new.linearLayout {
    //...
})
context.new.frameView(context.new.text("Hi"))
```

Radio buttons

Radio buttons are a bit trickier for two reasons:

- The value of a field is associated with the state of multiple views
- We have a dynamic number of states, depending on how many options we need

Before starting, we shall define the following terms:

- Radio group: a collection of radio buttons that refer to the same logical value
- Clearable radio group: a radio group where there can be no item selected (represented as `null` by the field value)

All functions actually create a `RadioGroup`, rather than a single `RadioButton`; they all accept a parameter `T : Any`, which is the type of the value in the field and a parameter `items`, which is a `List<T>` of all the possible values.

The function names reflect what they accept:

- `clearable`: functions support that the radio group can have no item selected, and accept a `MutableField<T?>` as selected value, where `null` represents the state "no item selected". If a function is not clearable, it can only accept a `MutableField<T>`
- `custom`: normally the radio buttons will automatically be added using the items list as a source, one after the other. If we want to better control this behavior, we can use the `custom` variant: it

passes to the `block` function a parameter `createItem`, which is also a function that, given an item, returns a `RadioButton`, which you can then add wherever you want inside the radio group.
- `enum`: in this case the type parameter must be an `Enum` and the items list is not needed, since it can be obtained automatically from the enum class.

Examples:

Kotlin

```
enum class Priority { LOW, NORMAL, HIGH, URGENT }

//Simplest way when you have an enum
val item1 = mutableFieldOf(Priority.NORMAL)
context.new.radioGroup.enum<Priority>( //items list is automatically taken from enum
    item1, //field containing current value
    { fieldOf(it.ordinal + " ) " + it.name) } //function that returns a field representing
the name for the given item
)

//If you have a list of possible values
val item2 = mutableFieldOf("hello")
context.new.radioGroup.list(
    item, //field containing current value
    listOf("hello", "ciao", "hola"), //items
    { fieldOf(it) } //function that returns a field representing the name for the given
item
)

//If you want to add the radio buttons manually
val item2 = mutableFieldOf(1)
context.new.radioGroup.listCustom(
    item, //field containing current value
    listOf(1, 5, 10, 20), //items
    { fieldOf(it.toString()) } //function that returns a field representing the name for
the given item
) { createItem -> //Block that is also responsible to create the items
    createItem(5) //Each createItem call creates a RadioButton for the given item and adds
it to the RadioGroup
    createItem(1)
    //Maybe do something in the middle...
    createItem(20)
    createItem(10)
}
```

Gotchas

- If you create a non-clearable radio group and then subsequently call clearCheck() it will result in a `IllegalStateException`
- When creating a custom radio group:
    - Calling the `createItem` function with the same parameter twice will result in an `IllegalStateException`
    - Failing to add one or more radio buttons will result in a cleared state if the value of the field becomes on of the not added ones
    - Adding a `RadioButton` not created with the `createItem` function will result in a cleared state when that item is selected, meaning that this item is virtually unselectable by the user.

Spinners

Logically spinners are akin to radio buttons: we can select a single value from many items. The differences are:

- The spinner **cannot** have a cleared state (there must always be a selected item)
- Since it uses an adapter, the items can change

Like for radio groups we have a specialized version for enums.

A few examples:

Kotlin

```
enum class Priority { LOW, NORMAL, HIGH, URGENT }

//With enums
val item1 = mutableField(Priority.NORMAL)
context.new.spinner<Priority>(
    item1,
    { it.ordinal + " ) " + it.name }
)

//With mutable items
val items = mutableListFieldOf("hello", "ciao", "hola")
val item2 = mutbaleFieldOf("ciao")
context.new.spinner(item2, items, { it })

item2.value = "hello" //Change selected value
items.remove("ciao") //Remove an item from the list
```

Gotchas

- Removing the currently selected item from the items list will result in an `IllegalStateException`
- Setting the field value to something that is not contained in the items list will result in an `IllegalStateException`

Recycler views

When creating a `RecyclerView` we have a few options. If we need to multiple view types then we should use the function that accepts a `Field<List<ItemInfo<T, VT>>`, where `T` is the type of the items and `VT` is a class representing a view type (usually an enum), that must implement the `ViewType` interface. Each `ItemInfo` contains the item, its id and the view type. Other than that we will need to pass a function that creates a view and one that, given a view and an item binds the necessary values. We can also optionally pass a `RecyclerView.LayoutManager` to use that defaults to `LinearLayoutManager`.

If we need only a single view type we can simply pass a `Field<List<T>>`, but we must also pass an additional parameter `idProvider` which is a function that, given an item, returns its id.

Examples:

Kotlin

```kotlin
val items = mutableListFieldOf("a", "b", "c", "d")

//Single view type
context.new.recyclerView(
    items,
    idProvider = { it.hashCode() },
    viewCreator = { context.new.text() }
    viewBinder = { view, item -> view.setText(item) }
)
```

See the FieldAdapter documentation for more info.

Adding your own

In order to keep the whole experience uniform, we highly suggest to take the time to create extensions in order to support your custom views in the `ViewCreator` class.

There are two ways to do so, each with its own pros and cons.

In the example we'll assume to have a custom view named `MovieView`:

Kotlin

```kotlin
class MovieView(context : Context) : View(context) { /* ... */}
```

Method 1: using the `ViewHandler` class

This is the most complete and flexible way of adding support for your custom view.

It boils down to:

- Creating a class that extends `ViewHandler` and overload the invoke operator and/or add named functions
- Optionally creating a `ViewPrefetcher` for your view in order to speed up the view creation
- Creating an extension property of `ViewCreator` that returns a new instance of your `ViewHandler`

Example:

Kotlin

```
//We leave the class "open" so if we want to extend MovieView and have its own handler with
the same base methods it's easy to do so
//We also parametrize the type of the view for the same reason
open class MovieViewViewCreator<out MV : MovieView> : BaseViewHandler<MV> {

    //These two constructors simply call the super implementation
    constructor(viewCreator: ViewCreator, supplier: (Context) -> PB) : super(viewCreator,
supplier)
    constructor(viewCreator: ViewCreator, viewPrefetcher: ViewPrefetcher<PB>) :
super(viewCreator, viewPrefetcher)

    //This is our invoke operator that should create the view given our custom parameters
    //In this case we'll accept a Field containing a Movie instance
    //It's good practice to always add an optional last block parameter
    //It's also good practice to add the inline keyword whenever we accept a function
    inline operator fun invoke(
        movie: Field<Movie>,
        block: MV.() -> Unit = {}
    ): MV {
        return this { //Here we invoke the invoke operator with the block only, defined in
BaseViewCreator
            //If you need to change the behavior of the empty invoke operator, extend
ViewCreator instead of BaseViewCreator

            setMovie(movie) //Assuming we have this method inside the MovieView
            block() //Call the block
        }
    }

    //Here you can add additional functions depending on your needs
    //For instance, we could add another invoke operator that accepts a simple Movie
instead of a Field<Movie>
}

//Optionally create a ViewPrefetcher
val MOVIE_VIEW_PREFETCHER = ViewPrefetcher(5) { MovieView(it) }

//Finally add the extension property to ViewCreator
val ViewCreator.movieView
    get() = MovieViewViewCreator(this, MOVIE_VIEW_PREFETCHER)

//Now we can call
val movie : Field<Movie>
context.new.movieView(movie) {
    //this is MovieView
}
```

- **Pros**: flexible, can respect view hierarchy without copy-pasting the same methods
- **Cons**: more code

Method 2: direct extension function on `ViewCreator`

If you only need to add support to a small view that doesn't need that much flexibility you can always add an extension function directly on `ViewCreator` that skips the `ViewHandler` altogether.

Example:

Kotlin

```
fun ViewCreator.movieView(movie: Field<Movie>, block: MovieView.() -> Unit = {}) :
MovieView {
    return view(MovieView(context)) {
        setMovie(movie) //Assuming we have this method inside the MovieView
        block() //Call the block
    }
}

//Now we can call
val movie : Field<Movie>
context.new.movieView(movie) {
    //this is MovieView
}
```

- **Pros**: less code
- **Cons**: not flexible, requires copy pasting the same method for a sub-type of the view