Dataflow for Android™ reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Dataflow for Android™

This extension allows to seamlessly integrate fields within the Android™ environment.

The main features are:

- **Preferences integration**: read and write preferences through fields.
- **Set fields in views**: bind a field to a view in order to display its content.

This library also makes use of a library that adds some basic utilities to Android™ classes.

Getting started

Setup

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

Gradle

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

Also you'll need to add the following compile and packaging options:

Gradle

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
        freeCompilerArgs += '-Xjvm-default=enable'
    }
    packagingOptions {
        exclude 'META-INF/library_release.kotlin_module'
    }
}
```

Remember also to add our server to the repositories, as explained here

Proguard and Multidexing

Since our libraries contain a lot of methods it is suggested to use Proguard to automatically remove the ones you don't use.

To enable proguard:

Gradle

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-
rules.pro'
    }
    debug {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules-
debug.pro'
    }
}
```

Also, add these lines to your `proguard-rules-debug.pro` if you want to keep seeing methods and

class names during debug:

```
-keepattributes SourceFile,LineNumberTable
-keepattributes LocalVariableTable,LocalVariableTypeTable
-keepnames class ** { *; }
```

If even after applying Proguard you still exceed the Android™'s limited 64K number of methods you should enable multidexing.

First look

There are two main features in this extension: preferences integration and fields in views.

Preferences integration

With the integration of shared preferences we can create fields that are bound to a specific key, like so:

Kotlin

```kotlin
val counter = context.defaultPreferences.getIntField("favorite_number", 0)

counter.rawValue.value = 73
```

Java

```java
SharedPreferences sp = PreferenceManager.getDefaultSharedPreferences(getContext());
MutableField<Integer> counter = SharedPreferencesUtils.getIntField(sp, "favorite_number",
0);

counter.getRawValue().setValue(73);
```

In this example we created a field that is bound to the `"favorite_number"` key in the default preferences. This means that we can change it but also listen for changes.

For more information see Shared preferences.

Fields in views

The most important feature is the ability to directly set a `Field` in a `View`. This works by attaching a `LifecyleOwner` instance to the view through tags. Most common functions are available:

Kotlin

```kotlin
val tv = TextView(context)
tv.setText(movie.name) //movie.name is a Field<String>
tv.setVisible(movie.name.isNotBlank())
```

Java

```
TextView tv = new TextView(getContext());
ViewUtils.setText(tv, movie.name);
ViewUtils.setVisible(tv, CharSequenceFieldUtils.isNotBlank(movie.getName()));
```

For more information see View binding.

Shared preferences

RawPreferenceField

The first logical step for Android™ support is to integrate with shared preferences because they change somewhat frequently, listening for changes is rarely done and not so straightforward, but can help make the app feel more reactive.

The most low-level way to bind a field to a preference is using a `RawPreferenceField`. The value is always nullable because the preference could be not present. It offers no option for a default value because it needs to mirror **exactly** the internal value of the preferences.

Creating a `RawPreferenceField`

The functions to create it are extensions of the `SharedPreferences` class. As usual, for java they can be called statically in a special utility class. A complete list can be found here.

Kotlin

```kotlin
val sp = context.getSharedPreferences("test_prefs") //Obtain SharedPreferences instance
val f /* : RawPreferenceField<String?> */ = sp.getRawStringField("pref_key")
f.value = "hello" //Sets the preference to "hello"
f.value = null //Removes the preference
```

Java

```java
SharedPreferences sp = getContext().getSharedPreferences("test_prefs"); //Obtain
SharedPreferences instance
RawPreferenceField<String> f = SharedPreferencesUtils.getRawStringField(sp, "pref_key");
f.setValue("hello"); //Sets the preference to "hello"
f.setValue(null); //Removes the preference
```

The returned field is a special implementation of `MutableField` that will allow you to read and change the value of the shared preference. Setting it to `null` will remove the preference. It also extends the `BasePreferenceField` class that exposes these three properties:

- `key`: the `String` containing the preference key associated with the field
- `preferences`: the `SharedPreferences` object associated with the key
- `defaultValue`: the default value (in this case this value will always be `null`)

Internally a listener is registered to the shared preferences so we can have multiple instances pointing at the same key without any problem. When the value is updated it is immediately applied.

Notice that the value mirrors **exactly** what is stored in the preferences: in this way we do not incur in problems related to the fields' laziness (i.e. the value doesn't change if it's the same).

Abstracting values

Preference fields can be created for all types supported by Android™'s shared preferences (`Boolean`, `Int`, `Long`, `Float`, `String`, `Set<String>`), but what if we want to store something a little more complex? Most of the times we store in the shared preferences a value that represents something but is not the whole object (like an ID). In this cases we can simply apply a two-way transformation, either explicitly by calling twoWayTransform() on the returned field or by passing the two functions directly to the get field function, like so:

Kotlin

```
val f /* : RawPreferenceField<Movie?> */ = sp.getRawLongField("favorite_movie", {
Movie.byIdOrNull(it) }, { it.id })
f.value = avatar //Sets Avatar's ID to the preference
f.value = null //Removes the preferece
```

Java

```
SharedPreferences sp = getContext().getSharedPreferences("test_prefs"); //Obtain
SharedPreferences instance
RawPreferenceField<Movie> f = SharedPreferencesUtils.getRawLongField(sp, "favorite_movie",
id -> Movie.byIdOrNull(id), movie -> movie.id);
f.setValue(avatar); //Sets Avatar's ID to the preference
f.setValue(null); //Removes the preference
```

The first function maps an low-level value to a high-level one, the second one is the opposite. In the example the first retrieves a movie from a saved `Long` ID, while the second one maps a movie to its ID that will be saved in the preferences.

Since `null` is used to represent the absence of the preference we cannot map it to a low-level value, however we are still able to return `null` in the low to high level one because we want the possibility to handle incorrect values. In the example above a movie with the saved ID could not be found for a lot of reasons. When this happens, the function yields `null` which then "pongs" back and removes the preference.

PreferenceField

This implementation is able to provide a bit more abstraction by requiring a default value.

Creating a `PreferenceField`

Creating it is very similar to the creation of the raw one:

Kotlin

```
val pref /* : PreferenceField<String> */ = sp.getStringField("pref_key", "default_value")
```

Java

```
PreferenceField<String> pref = SharedPreferencesUtils.getStringField(sp, "pref_key",
"default_value");
```

Like its raw counterpart it extends `BasePreferenceField`, exposing the three already seen properties; the difference is that this time it only implements `Field`, making it read-only. The value will be the passed default value when the preference is not present.

But why is it read only? It is because having a default value creates some problems due to the value of the field not being exactly the same as the one shared preferences provide.
For example, if the default value is `10` and someone calls `pref.value = 10`, what is the correct behavior? Remove the preference? Set it to `10`? Moreover, given the the lazy nature of fields (i.e. the value doesn't change if it's the same) the value simply wouldn't get saved if the value was already `10`.

Changing the value of a `PreferenceField`

In order to modify the value associated with this preference we have two options, exposed through two `MutableField` properties:

- `rawValue`: this field has the same value and behaves exactly has a `RawPreferenceField`, i.e. it's `null` when there is no preference set. It is the safest way to change the value, but since its value can be `null` it might be a bit of an hassle to manage (for instance, calling `increment()` wouldn't be possible).
- `unsafeMutable`: this field adds the default value into the equation, making it easier to manage, but with more problems: in fact setting a value equal to the default one does **not** guarantee whether the preference will be saved with that value or if it will remain unset. Please notice that is field is a lazy property, so it's initialized upon the first call. Doing this creates a new mutable field that is a two-way transformation of the raw one, so every time this field changes the value will be saved, regardless if it's a default value or not. So once its created, it will affect your preference.

Problems of `unsafeMutable`

You can read more about `unsafeMutable` behavior in the doc.

Problem 1: unable to remove preference

Kotlin

```kotlin
//Initially we suppose there is no value in the preferences
val num = sp.getIntField("favorite_number", 73)
num.rawValue.value = 10 //Set favorite number to 10, all OK until here
num.unsafeMutable.increment() //Increments the favorite number to 11, creating the
unsafeMutable in the process
//This call now *tries* to remove the preference. The effect is that the preference is
removed, the unsafeMutable detects it,
//puts as its value the default one (73) and triggers the change of rawValue to 73, which
then sets the preference to 73
num.removePreference()

val num2 = sp.getIntField("favorite_number", 88) //Same preference, different default value
num2.value //Should yield 88, will yield 73
```

Java

```
//Initially we suppose there is no value in the preferences
PreferenceField<Int> num = SharedPreferencesUtils.getIntField(sp, "favorite_number", 73);
num.getRawValue().setValue(10); //Set favorite number to 10, all OK until here
IntFieldUtils.increment(num.getUnsafeMutable()); //Increments the favorite number to 11,
creating the unsafeMutable in the process
//This call now *tries* to remove the preference. The effect is that the preference is
removed, the unsafeMutable detects it,
//puts as its value the default one (73) and triggers the change of rawValue to 73, which
then sets the preference to 73
num.removePreference();

PreferenceField<Int> num2 = SharedPreferencesUtils.getIntField(sp, "favorite_number", 88)
//Same preference, different default value
num2.getValue() //Should yield 88, will yield 73
```

Problem 2: unable to save preference with the default value

Kotlin

```
//Initially we suppose there is no value in the preferences
val num = sp.getIntField("favorite_number", 73)
num.unsafeMutable.value = 73 //The value was already 73 so the change isn't detected and
the value is not actually saved

val num2 = sp.getIntField("favorite_number", 88) //Same preference, different default value
num2.value //Should yield 73, will yield 88
```

Java

```
//Initially we suppose there is no value in the preferences
PreferenceField<Int> num = SharedPreferencesUtils.getIntField(sp, "favorite_number", 73);
num.getUnsafeMutable().setValue(73); //The value was already 73 so the change isn't
detected and the value is not actually saved

PreferenceField<Int> num2 = SharedPreferencesUtils.getIntField(sp, "favorite_number", 88);
num2.getValue(); //Should yield 73, will yield 88
```

Abstracting more complex values

As on the raw counterparts, we can pass two map transformation functions. The `null` rules are the same and the behavior is the same. For more info see the correspondent section.

View binding

Listening fields on views

Once we're in the Android™ environment it is obvious that we'll want, sooner or later, show some values contained in fields to the user. If you're an Android developer you'll already know that the UI is made of `Views`.

Views as lifecycle owners

Every `View` instance has an extension function `asLifecycleOwner()` that will return a special `LifecycleOwner` that will be attached to the view. This `LifecycleOwner` will be aware of the attached status of the view, so listeners will automatically be stopped when the view detaches and restarted when it reattaches, in order to avoid useless computations.

Since most of the times we'll want to change the UI from the listeners, there is also the `asUIThreadLifecycleOwner()` that will automatically call the listeners on the UI thread.

Recalling the page about lifecycles we can attach to this instance listeners of fields, like so:

Kotlin

```
val tv = TextView(context)
tv.asUIThreadLifecycleOwner().listen(movie.name) { //Will be called on the UI thread
    tv.setText(it)
}
```

Java

```
TextView tv = new TextView(getContext());
ViewListenUtils.asUIThreadLifecycleOwner(tv).listen(movie.name, name -> { //Will be called
on the UI thread
    tv.setText(name);
});
```

This method of binding fields to views is the most low-level one: if you **only** need to change the state of the view the suggested method is to use view states.

Utility functions

Simple utility functions exist in order to avoid calling the `asUIThreadLifecycleOwner()` method each time:

Kotlin

```
val tv = TextView(context)
tv.listen(movie.name) { //Will be called on the UI thread
    tv.setText(it)
}
```

Java

```
TextView tv = new TextView(getContext());
ViewListenUtils.listen(tv, movie.name, name -> { //Will be called on the UI thread
    tv.setText(name);
});
```

The `listen()` extension functions in views are semantically equivalent to the `listen()` functions in `LifecycleOwner`.

View state

We will now introduce view states: a `ViewState<V>` is special class that represents one property with its respective value for a view of type `V`. A view state could be for example the text of a `TextView` or its background color.

A view state can be set or unset to a view; an important property is that **they do not leave side effects**, i.e. adding a view sate and then removing it afterwards will leave the view in the same state as if it was never applied.

Creating a view state

As usual we'll start from the most low-level functions. To create a view state for a single property we'll need to provide the following information:

- A key object that must be the same for all states that change the same property
- A field that contains our value
- Two functions: one that reads the value contained in the view (for restoring it later), and one that applies the value of the field
- Optionally we can also pass two functions that will be called when the view state is added and removed from the view.

In the following example we'll create a view state that changes the text of a `TextView`:

Kotlin

```
createViewState<TextView, String>(KEY, movie.name
    { getText() }, //Reads the old value for later restoring. Receiver is TextView
    { setText(it) } //Given a movie name, sets it to the TextView. Receiver is TextView
)
```

Java

```
ViewState.createViewState(KEY, movie.name,
    tv -> tv.getText(), //Reads the old value for later restoring. Receiver is TextView
    (tv, name) -> tv.setText(name) //Given a movie name, sets it to the TextView. Receiver
is TextView
);
```

Summing view states

We can easily sum more view states and obtain a single bigger one. This is how it's done:

Kotlin

```
val big = vs1 + vs2 + vs3 + null + vs4
```

Java

```
ViewState big = ViewStateUtils.sum(vs1, vs2, vs3, null, vs4);
```

As you can see from the example, summing `null` is tolerated: it is simply interpreted as a view state that does nothing. After the sum `big` is now a view state that encapsulates all the behaviors of the summed view states.

Default view states

There are tens of utility functions for the most common view states. A few examples:

Kotlin

```
text("hello") //Creates a view state with the text "hello"
text(movie.name) //Creates a view state with the text binded to the field movie.name
paddingLeft(16)
visibility(View.GONE)
```

Java

```
State.text("hello"); //Creates a view state with the text "hello"
State.text(movie.name); //Creates a view state with the text binded to the field movie.name
State.paddingLeft(16)
State.visibility(View.GONE)
```

A complete list of default view states can be found in the doc.

Adding a view state permanently

In order to add a view state to a view we can do the following:

Kotlin

```
view.states += text(movie.name) //the state can be a ViewState or also a Field<ViewState>
```

Java

```
ViewStateUtils.states(view).add(State.text(movie.name)); //the state can be a ViewState or
also a Field<ViewState>
```

If you do this you cannot later remove the state you've added, so be sure to call this function only when the state you set is permanent.

Setting and removing a view state

If you want to be able to add and remove view states dynamically you can do so with the function `set()`. Once more you'll need to provide a key **logically different from the key of a single view state**. This concept is important because it's easy to view the two keys as the same logical value, but it's not the case.

When we create a view state we manage a single property and the key is bound to that property. Recall that a view state can represent the sum of multiple states, therefore managing multiple properties. The key that is required now is associated to the view state as a whole, possibly composed of multiple states.

Let's see this example to clarify:

Kotlin

```
class Info(
    val text: String,
    val hasErrors: Boolean
)

val MY_KEY = Any()

val f = mutableFieldOf(Info("Hello", false))

//When we set this state the view will display the text in white on a red background if it
is in error
view.states.set(MY_KEY, f) {
    text(it.text) + if (it.hasErrors) {
        backgroundColor(Color.RED) + textColor(Color.WHITE)
    } else null
}

//When we set this state the view will display the text on a green backround if it's not in
error
//Since the key is the same as the previous one, all previous effects are removed
view.states.set(MY_KEY, f) {
    text(it.text) + if (!it.hasErrors) {
        backgroundColor(Color.GREEN)
    } else null
}
```

Java

```
class Info {
    public final String text;
    public final boolean hasErrors;

    //Constructor omitted for brevity
}

Object MY_KEY = new Object();

MutableField<Info> f = MutableField.of(new Info("hello", false));

//When we set this state the view will display the text in white on a red background if it
is in error
ViewStateUtils.states(view).set(MY_KEY, f, info -> {
    return ViewStateUtils.sum(
        State.text(info.text),
        info.hasErrors ? ViewStateUtils.sum(State.backgroundColor(Color.RED),
State.textColor(Color.WHITE)) : null
    );
});

//When we set this state the view will display the text on a green backround if it's not in
error
//Since the key is the same as the previous one, all previous effects are removed
ViewStateUtils.states(view).set(MY_KEY, f, info -> {
    return ViewStateUtils.sum(
        State.text(info.text),
        !info.hasErrors ? State.backgroundColor(Color.GREEN) : null
    );
});
```

To later remove a view state you can use the `remove()` function:

Kotlin

```
view.states.remove(MY_KEY)
```

Java

```
ViewStateUtils.states(view).remove(MY_KEY);
```

Utility functions

Most of the times we'll just want to set a field to a view without thinking too much about it. For this reason there are several utility function to do just that, usually three per property: one that accepts a simple field that contains that value and two with the suffix `ViewState` that accept a view state or a field of view state.

A few examples:

Kotlin

```
textView.setText(movie.name)
progressBar.setProgress(movie.progress)
```

Java

```
ViewUtils.setText(textView, movie.name);
ViewUtils.setProgress(progressBar, movie.progress);
```

A complete list of these utility functions can be found in the doc.

Other bindings

Activity and Fragment binding

Like with Android's `Views`, we may want to listen for `Field`'s changes on an `Activity` or on a `Fragment`.

To do so, the paradigm is the same we used on `Views`: the extension functions `asLifecycleOwner()` or `asUIThreadLifecycleOwner()` return a `LifecycleOwner` that will call callbacks respectively on the thread that made the change or on the UI thread. Moreover, this `LifecycleOwner` respects the lifecycle of the `Activity` or `Fragment`: in particular, the callbacks will stop when the `Activity` or `Fragment` enter the stopped state, and will resume when they enter the started state.

Like with `Views`, the library provides extension functions for both `Activity` and `Fragment` for calling `listen()` directly without using `asUIThreadLifecycleOwner`.

Example

Suppose that we have an object called `theme` that contains some `Fields` that specify the aspect of the app, and one of those `Fields` is the background color the app should have.

To react to changes to that `Field`, we can do the following:

Kotlin

```
activity.asUIThreadLifecycleOwner().listen(theme.backgroundColor) { //Will be called on the
UI thread
    activity.window.decorView.setBackgroundColor(it)
}
```

Java

```
ActivityListenUtils.asUIThreadLifecycleOwner(activity).listen(theme.backgroundColor, bg ->
{ //Will be called on the UI thread
    activity.getWindow().getDecorView().setBackgroundColor(bg)
});
```

Equivalently:

Kotlin

```
activity.listen(theme.backgroundColor) { //Will be called on the UI thread
    activity.window.decorView.setBackgroundColor(it)
}
```

Java

```
ActivityListenUtils.listen(theme.backgroundColor, bg -> { //Will be called on the UI thread
    activity.getWindow().getDecorView().setBackgroundColor(bg)
});
```

The simplicity of this approach allows to react to changes that usually are not handled dynamically, but

are only handled when the `Activity` starts.

Please note that although the example uses `Activity`, the same code can be used on `Fragment`s as well.

AlertDialog binding

Sometimes we want to show an `AlertDialog`, but the title or the message of that dialog depends on a `Field`.

In this case we can use a `FieldAlertDialogBuilder`. This class extends `AlertDialog.Builder` from the `AndroidX` library, but provides the methods `setTitle` and `setMessage` that accept a `Field` of `Charsequence` and will respectively change the title or the message of the dialog as soon as the `Field` changes.

For example:

Kotlin

```
FieldAlertDialogBuilder()
    .setTitle(movie.name)
    .setMessage(movie.name.transform { name->
        "Set $name as watched?"
    })
    .setPositiveButton(...)
    .setNeutralButton(...)
    .show()
```

Java

```
new FieldAlertDialogBuilder()
    .setTitle(movie.getName())
    .setMessage(movie.getName().transform(name->
        "Set " + name + " as watched?"
    ))
    .setPositiveButton(...)
    .setNeutralButton(...)
    .show();
```

Adapters

FieldListViewAdapter

`FieldListViewAdapter` is an abstract class that extends Android's `BaseAdapter` that allows to easily have an adapter that accepts a `Field<List<T>>`. It effortlessly can be created through the `FieldListViewAdapter.of` function. It accepts the following parameters:

- `items`: the `Field<List<T>>` containing the items we want to display
- `viewCreator`: a function that creates a view (of type `V`)
- `viewBinder`: a function that, given a view of type `V` and an item of type `T` binds the item to the view
- `idProvider`: an optional function that provides unique IDs for the items. Defaults to calling `hashCode()`

An example:

Kotlin

```kotlin
//Assuming we have a class Movie and a MovieView with the method setMovie()
val movies : Field<List<Movie>> = getMovies()

listView.setAdapter(FieldListViewAdapter.of(
    items = movies,
    viewCreator = { MovieView(context) },
    viewBinder = { view, movie -> view.setMovie(movie) },
    idProvider = { movie -> movie.id }
))
```

Java

```java
//Assuming we have a class Movie and a MovieView with the method setMovie()
Field<List<Movie>> movies = getMovies();

listView.setAdapter(FieldListViewAdapter.of(
    movies,                                 //items
    () -> new MovieView(getContext()),      //viewCreator
    (view, movie) -> view.setMovie(movie),  //viewBinder
    movie -> movie.getId()                  //idProvider
));
```

FieldListViewArrayAdapter

This class is another possible choice when we have a `Field<List<T>>` that we want to display in a `ListView`. It mimics (and extends) the Android's `ArrayAdapter` class.

We have two ways of constructing it: via the constructor or via the `FieldListViewArrayAdapter.of` function. Both these version require the following parameters:

- `items`: the `Field<List<T>>` containing the items to display.

- `context`: a `Context`.
- `resource`: the layout resource to inflate for each view.
- `textViewResourceId`: a `TextView` ID contained in the passed layout that is the primary text. If `0` is passed, it takes the whole resource as a `TextView`.

The constructor than accepts a lambda that, given the inflated `View`, the `TextView`, the item `T` and the `isDropdownView` boolean binds the item to the view.

The `of()` method simply accepts a lambda that, for a given item `T` returns a `Field<CharSequence?>` containing the text to put in the main `TextView`.

There is also a special `of()` version that directly accepts a `Field<List<CharSequence?>>` as items.

Examples:

Kotlin

```
//Assuming we have a class Movie
val movies : Field<List<Movie>> = getMovies();

//Example 1
listView.setAdapter(FieldListViewArrayAdapter.of(
    items = movies,
    context = context,
    resource = android.R.layout.list_view_items
    textViewResourceId = android.R.id.textView,
    textGetter = { it.name }
))

//Example 2
listView.setAdapter(FieldListViewArrayAdapter.of(
    items = movies.transform { it.name }, //We are passing directly a
Field<List<CharSequence?>>
    context = context,
    resource = android.R.layout.list_view_items
    textViewResourceId = android.R.id.textView
))

//Example 3, using constructor
listView.setAdapter(FieldListViewArrayAdapter(
    items = movies,
    context = context,
    resource = android.R.layout.list_view_items
    textViewResourceId = android.R.id.textView,
    viewBinder = { view, textView, item, isDropdownView ->
        textView.setText(movie.name)
    }
))
```

Java

```
//Assuming we have a class Movie
Field<List<Movie> movies = getMovies();

//Example 1
listView.setAdapter(FieldListViewArrayAdapter.of(
    movies,                                     //items
    context,                                    //context
    android.R.layout.list_view_items            //resource
    android.R.id.textView,                      //textViewResourceId
    movie -> movie.getName()                    //textGetter
));

//Example 2
listView.setAdapter(FieldListViewArrayAdapter.of(
    movies.transform(movie -> movie.getName()), //items: We are passing directly a
Field<List<CharSequence?>>
    context,                                    //context
    android.R.layout.list_view_items            //resource
    android.R.id.textView                       //textViewResourceId
));

//Example 3, using constructor
listView.setAdapter(new FieldListViewArrayAdapter(
    movies,                                     //items
    context,                                    //context
    android.R.layout.list_view_items            //resource
    android.R.id.textView,                      //textViewResourceId
    (view, textView, item, isDropdownView) -> { //viewBinder
        textView.setText(movie.name)
    }
));
```

RecyclerView FieldAdapter

While we still provide adapters for `ListView`, as you may already know, you should build your app using `RecyclerViews`, as they provide more flexibility.

Our class `FieldAdapter` extends the `RecyclerView.Adapter` class. Before seeing how to create a it we must introduce a few classes.

The `ViewType` class

Since a `RecyclerView` can display views of different types, we created this class to represent it. To create it we can simply call the constructor that accepts two parameters:

- `viewCreator`: a function that, given a `Context`, creates a view of type `V`
- `viewBinder`: a function that, given a view `V` and an item `T`, binds the item to the view

Example:

Kotlin

```
//Assuming we have a class Movie and a MovieView with the method setMovie() that accepts a
Movie
val movies : Field<List<Movie>> = getMovies()

val movieViewType = ViewType(
    viewCreator = { context -> MovieView(context) },
    viewBinder = { view, movie : Movie -> view.setMovie(movie) }
)
```

Java

```
//Assuming we have a class Movie and a MovieView with the method setMovie()
Field<List<Movie>> movies = getMovies();

ViewType<Movie, MovieView> movieViewType = new ViewType<>(
    context -> new MovieView(context),          //viewCreator
    (view, Movie movie) -> view.setMovie(movie) //viewBinder
);
```

We can also pass an optional ID to represent this view type. If we omit it, an unique one will automatically be generated.

The `ViewTypeField` class

This class extends `ViewType` and, instead of the `viewCreator`/`viewBinder` couple, requires just a `viewCreator`, that this time accepts a view `V` and a `Field<T?>` for the item.

Example:

Kotlin

```
//Assuming we have a class Movie and a MovieView with the method setMovie() that accepts a
Field<Movie?>

val movieViewType = ViewTypeField(
    viewCreator = { context, movie : Field<Movie?> ->
        val view = MovieView(context)
        view.setMovie(movie);
        view
    }
)
```

Java

```
//Assuming we have a class Movie and a MovieView with the method setMovie()

ViewTypeField<Movie, MovieView> movieViewType = new ViewTypeField<>(
    (context, Field<Movie> movie) -> {  //viewCreator
        MovieView view = new MovieView(context);
        view.setMovie(movie);
        return movie;
    }
);
```

As you can see the field has type `T?` representing the item it needs to display. The value is `null` the first time the view is instantiated, because it happens before it's known which item it needs to represent.

The `ItemInfo` class

This class holds the information about a single item. It contains an item of type `T`, a view type and an item ID of type `Long`. If the item ID is not provided it defaults to `RecyclerView.NO_ID`. The knowledge of this class will be needed when created adapters that can display multiple view types.

Creation of a `FieldAdapter`

The creation of this class passes through the `FieldAdapter.of` function, which has a few variants.

For a single view type

To create an adapter that handles a single view type all we need to pass is:

- `items`: a `Field<List<T>>` containing the items to display
- `idProvider`: a function that, given an item `T`, returns a unique ID for that item
- `viewType`: a `ViewType<T, *>` that represents the single view type

There are also two functions that, instead of the view type, accept directly the `viewCreator`/`viewBinder` or the single `viewCreator`, as seen for the `ViewType` and `ViewTypeField` constructors.

Example:

Kotlin

```kotlin
val movies : Field<List<Movie>> = getMovies()

//Assuming we have the movieViewType created before
FieldAdapter.of(
    items = movies,
    idProvider = { it.id },
    viewType = movieViewType
)
```

Java

```java
Field<List<Movie>> movies = getMovies();

//Assuming we have the movieViewType created before
FieldAdapter.of(
    movies,                 //items
    movie -> movie.getId(), //idProvider
    movieViewType           //viewType
);
```

For multiple view types

In order to create a `FieldAdapter` that is able to display multiple view types, the only needed parameter is a `Field<List<ItemInfo<T>>>`, since the `ItemInfo` class already contains all the necessary information.

Example:

Kotlin

```kotlin
//Assuming we have the following classes:
// -Movie: that represents a film
// -MovieView: with the method setMovie() that accepts a Movie
// -Show: that represents a TV series
// -ShowView: with the method setShow() that accepts a Show
// -Media: class extended by both Movie and Show

//We create the view type for a Movie...
val movieViewType = ViewType(
    viewCreator = { context -> MovieView(context) },
    viewBinder = { view, movie : Movie -> view.setMovie(movie) }
)
//...and for a show
val showViewType = ViewType(
    viewCreator = { context -> ShowView(context) },
    viewBinder = { view, show : Show -> view.setShow(show) }
)

val media : Field<List<Media>> = getMedia()

FieldAdapter.of(
    items = media.map {
        //Here we transform each item in its corresponding ItemInfo
        when(it) {
            is Movie -> ItemInfo(it, it.id, movieViewType)
            is Show -> ItemInfo(it, it.id, showViewType)
            else -> throw IllegalStateException()
        }
    }
)
```

Java

```java
//Assuming we have the following classes:
// -Movie: that represents a film
// -MovieView: with the method setMovie() that accepts a Movie
// -Show: that represents a TV series
// -ShowView: with the method setShow() that accepts a Show
// -Media: class extended by both Movie and Show

//We create the view type for a Movie...
ViewType<Movie, MovieView> movieViewType = new ViewType<>(
    context -> new MovieView(context),          //viewCreator
    (view, Movie movie) -> view.setMovie(movie) //viewBinder
);
//...and for a show
ViewType<Show, ShowView> showViewType = new ViewType<>(
    context -> new ShowView(context),           //viewCreator
    (view, Show show) -> view.setShow(show)     //viewBinder
);

Field<List<Media>> media = getMedia();

FieldAdapter.of(
    IterableFieldUtils.map(media, item -> {
        //Here we transform each item in its corresponding ItemInfo
        if(it instanceof Movie) new ItemInfo<>(it, it.id, movieViewType)
        else if(it instanceof Show) new ItemInfo<>(it, it.id, showViewType)
        else throw new IllegalStateException();
    })
);
```