Dataflow reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Dataflow

Dataflow is a Kotlin™/JVM library that combines reactive programing, functional programming and the observer pattern to simplify the presentation of data in your program. Let the programmers focus on the logic of the application while we do the hard work.

The main advantages of using this library are:

- **Always consistent** state between UI and data
- **Single point of truth**: think only about the main source of data and let the library manage the rest
- Express in a **few lines of code** data transformations that will usually require hundred

It aims to be as simple as possible, while being fully featured and highly efficient. In particular:

- Thread safe
- Automatic lifecycle handling, to avoid memory leaks
- Lazy by default
- Automatically detects and avoids useless computations
- Atomic changes

Dataflow has several extensions that provide more features. For example, the async extension, provides the following features:

- Automatically moves most of the work on background threads
- Debouncing
- Associate a state with data (loading, error, etc.)
- Performance optimizations on collections

Why Kotlin™?

The main reasons for the choice of Kotlin™ as the main language for this library are:

- **Becoming the new standard**: Kotlin™ is quickly eroding market-share from Java™. Even Google™ in its latest I/O 2019 said that it's going to adopt a kotlin-first mentality for the Android™ platform.
- **Open-source**: the language is open-source and supported by the Kotlin™ foundation, in which Google™ and JetBrains™ actively take part.
- **Conciseness**: one of the main advantages of using Kotlin™ is the fact that the language is a lot more concise then Java™, making writing code a more pleasant and elegant task. You'll see for yourself in the examples how much redundant code is needed in Java™ and how much compact Kotlin is, while maintaining a high degree of legibility.
- **More advanced features**: when it comes to new cool features, Java™ really falls behind. Operators overload, infix notation, extension functions, generics variance, data classes, internal visibility modifier, distinction between read-only and mutable collections, sealed classes and singleton support are only some of the features that make life easier and allow to construct more powerful apps and libraries.
- **Less baggage**: being a newer language, Kotlin™ doesn't suffer from old decisions that have been proven being problematic. Two striking examples are null-safe types and checked exception: the absence of the first and the presence of the latter complicate things in Java™.
- **Based on JVM**: while Java™ is becoming older and older, the JVM has always shined. It allows

programs to run on all different kinds of hardware, can perform a lot run-time optimization that simply couldn't be done by compiled languages and offers robust and well-tested APIs.
- **Java™ retrocompatibility**: by being completely retrocompatible with Java™, you can start developing in Kotlin™ right away without ever changing a single line of old Java™ code. Also, you can take advantage of the immense number of Java™ libraries written all over the world.

Get started now!

Go to the setup page to start learning how to use Dataflow!

Getting started

Setup

In this page we will explain the necessary steps in order to prepare your environment to use Dataflow.

Getting an account

Dataflow is a commercial library. If you already have your credentials you can proceed to the next step, otherwise you'll have to first contact sales in order to get a license.

Adding the library to your project

Adding Dataflow to your project is very easy. First, you must add our servers to the repositories list:

Gradle

```
repositories {
    maven {
        credentials {
            username "your-username"
            password "your-password"
        }
        url "https://artifactory.femastudios.com/artifactory/libs-release"
        authentication {
            basic(BasicAuthentication)
        }
    }
}
```

Maven

```
<servers>
    <server>
        <username>your-username</username>
        <password>your-password</password>
        <id>fema-artifactory</id>
    </server>
</servers>
<repositories>
    <repository>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <id>fema-artifactory</id>
        <url>https://artifactory.femastudios.com/artifactory/libs-release</url>
    </repository>
</repositories>
```

Then, simply include the library as following:

Gradle

```
implementation 'com.femastudios:dataflow:1.0.2'
```

Maven

```
<dependency>
    <groupId>com.femastudios</groupId>
    <artifactId>dataflow</artifactId>
    <version>1.0.2</version>
</dependency>
```

It is also necessary to add the following configuration for the Kotlin™ compiler:

Gradle

```
compileKotlin {
    kotlinOptions {
        jvmTarget = "1.8"
        freeCompilerArgs += "-Xjvm-default=enable"
    }
}
```

Maven

```
<build>
<plugins>
    <plugin>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-plugin</artifactId>
        <version>${kotlin.version}</version>
        ...
        <configuration>
            <jvmTarget>1.8</jvmTarget>
            <args>
                <arg>-Xjvm-default=enable</arg>
            </args>
        </configuration>
    </plugin>
</plugins>
</build>
```

For more information about configuring your environment with Kotlin™ please visit the official Kotlin™ doc for Maven and for Gradle.

Gradle is the suggested choice.

First look

Fields

The main concept of this library are fields: a `Field<T>` is an interface that holds a value of type `T` and allows to register (and cancel) callbacks for data changes.

`MutableField<T>` is an interface that builds on top of `Field`, adding the possibility to change the value.

Here's a first example that shows how to construct fields:

Kotlin

```kotlin
val field = mutableFieldOf(5) //Creates a mutable field of Int
    println(field.value) //Outputs 5
    field.value = 10 //Sets the value contained by the field to 10
    println(field.value) //Outputs 10
```

Java

```java
MutableField<Integer> field = MutableField.of(5); //Creates a mutable field of Integer
System.out.println(field.getValue()); //Outputs 5
field.setValue(10); //Sets the value contained by the field to 10
System.out.println(field.getValue()) //Outputs 10
```

For more information see Creating fields.

Registering listeners

There are three main ways to listen for changes in the field value:

- **Strongly**: a strong reference to the listener is kept until the `remove()` function is called.
- **Weakly**: only a weak reference to the listener will be kept, allowing the garbage collector to reclaim it if all other references are lost. The caller is responsible for holding a strong reference to the callback.
- **With lifecycle**: the listener is active until the lifecycle owner is alive.

In the examples we'll use the strong version for simplicity's sake. Here's our first example:

Kotlin

```kotlin
val field = mutableFieldOf(5)
    field.listeners.addStrongly { newVal ->
        print(newVal)
    }
    for(i in 1..5) {
        field.value = i
    }
    //Output: 12345
```

Java

```java
MutableField<Integer> field = MutableField.of(5);
field.listeners().addStrongly((newData) -> {
    System.out.print(newData);
};
for(int i = 1; i <= 5; i++) {
    field.setValue(i);
}
//Output: 12345
```

For more information see Listening fields.

Field manipulation

The real power of fields is unleashed when performing data manipulation. The main philosophy is that any transformation on fields will yield another `Field` instance. In this way we can express all the transformations we want and keep fields around, allowing us to listen to changes in the original data that will propagate all the way down.

The following example demonstrates a basic usage of field manipulation:

Kotlin

```kotlin
val number = mutableFieldOf(5)
    val negation = number.transform { -it }
    negation.listeners.addStrongly { newVal ->
        print("$newVal ")
    }
    number.value = 7
    number.value = -4
    number.value = 1
    //Output: -7 4 -1
```

Java

```java
MutableField<Integer> number = MutableField.of(5);
Field<Integer> negation = field.transform(v -> -v);
negation.listeners().addStrongly((newData) -> {
    System.out.print(newData + " ");
});
number.setValue(7);
number.setValue(-4);
number.setValue(1);
//Output: -7 4 -1
```

The `transform()` function returns a `Field`: notice that it is **not** a mutable field and we **cannot** directly set a value.

For more information see Manipulating fields.

Basics

Creating fields

The first thing to consider when creating a field is whether we want to be able to change its value. If the answer is yes, then we'll need a `MutableField`.

Mutable fields

Most of the time, we will want to be able to set values in our fields manually. For this purpose we need a `MutableField` instance.

Kotlin

```kotlin
val f /* : MutableField<String> */ = mutableFieldOf("hello world")
    f.value = "foo bar" //Allowed
```

Java

```java
MutableField<String> f = MutableField.of("hello world");
f.setValue("foo bar"); //Allowed
```

If we want to change the value based on the previous one, we can do something like this:

Kotlin

```kotlin
val f = mutableFieldOf(5)
    f.setValue { it * 10 } //Will multiply 5 by 10, setting the value to 50
    println(f.value)
```

Java

```java
MutableField<Integer> f = MutableField.of(5);
f.setValue(old -> old * 10); //Will multiply 5 by 10, setting the value to 50
```

The given lambda will **atomically** change the value of the field: the value won't be changed until the computation is complete, and multiple calls to the function will be enqueued and be executed one at a time.

Constant fields

Sometimes the need may arise to create a field with a constant value (i.e. we cannot set the value).

Kotlin

```kotlin
val f /* : Field<String> */ = fieldOf("hello world")
    println(f.value) //OK
    //Cannot call f.value = "foo bar"
```

Java
```java
Field<String> f = Field.of("hello world");
//Cannot call f.setValue("foo bar");
```

Restrictions on field values

Please note that there are a couple restrictions on things that should be put as field values:

- **Only immutable data**: fields cannot know when the internal state of the value they hold changes, thus being unable to call the necessary callbacks in that case. For this reason, you should **never** put mutable data in fields; this also makes the fields thread-safe.
- `equals()` **implemented correctly**: two equal objects should **never** have some accessible property whose value is different. This because when a value is trying to be set in a field, it is first checked for equality with the old one; if `true` is returned the value is **not** set and listeners are **not** called. This is necessary to improve efficiency as the number of fields in the program explodes.

Obliviously fields cannot check if the values satisfy this restrictions, neither at compile-time nor at run-time, so it is the programmer's duty to avoid this situations.

Utility functions

In order to make field creation easier and more idiomatic, several utility functions are available to create both constant and mutable fields.

Fields of constants

These functions allow to create a `Field` whose value is a well-known constant (either `null`, `true` or `false`).

Kotlin
```kotlin
fieldOfNull() //yields Field<Nothing> containing null
    fieldOfTrue() //yields Field<Boolean> containing true
    fieldOfFalse() //yields Field<Boolean> containing false
```

Java
```java
FieldUtils.fieldOfNull(); //yields Field containing null
Field.ofNull(); //same as above

FieldUtils.fieldOfTrue(); //yields Field<Boolean> containing true
FieldUtils.fieldOfFalse(); //yields Field<Boolean> containing false
```

Collection fields

We can also put **immutable** collections in fields, and Dataflow provides a few utility functions for this purpose. More information about collections in fields can be found here.

List fields

A way to create fields or mutable fields that contain an **immutable** `List`.

Kotlin

```kotlin
//yields Field<List<String>> containing the immutable list ["a", "b", "c"]
    listFieldOf("a", "b", "c")

    //yields Field<List<Int?>> containing an immutable empty list
    listFieldOf<Int?>()

    //yields MutableField<List<Float>> containing the immutable list [1, 2, 3]
    mutableListFieldOf(1f, 2f, 3f)
```

Java

```java
//yields Field<List<String>> containing the immutable list ["a", "b", "c"]
FieldUtils.listFieldOf("a", "b", "c")

//yields Field<List<Integer>> containing an immutable empty list
FieldUtils.<Integer>listFieldOf()

//yields MutableField<List<Float>> containing the immutable list [1, 2, 3]
FieldUtils.mutableListFieldOf(1f, 2f, 3f)
```

Set fields

Similarly to lists, these functions create fields or mutable fields that contain an **immutable** `Set`.

Kotlin

```kotlin
//yields Field<Set<String>> containing the immutable set {"a", "b", "c"}
    setFieldOf("a", "b", "c")

    //yields Field<Set<Int?>> containing an immutable empty set
    setFieldOf<Int?>()

    //yields MutableField<Set<Float>> containing the immutable set {1, 2, 3}
    mutableSetFieldOf(1f, 2f, 3f)
```

Java

```java
//yields Field<Set<String>> containing the immutable set {"a", "b", "c"}
FieldUtils.setFieldOf("a", "b", "c")

//yields Field<Set<Integer>> containing an immutable empty list
FieldUtils.<Integer>setFieldOf()

//yields MutableField<Set<Float>> containing the immutable set {1, 2, 3}
FieldUtils.mutableSetFieldOf(1f, 2f, 3f)
```

Map fields

Lastly, we can also create fields or mutable fields that contain an **immutable** `Map`.

Kotlin

```kotlin
//yields Field<Map<Int, Char>> containing the immutable map {1: 'a', 2: 'b'}
    mapFieldOf(1 to 'a', 2 to 'b')

    //yields Field<Map<Int, Boolean?>> containing an immutable empty map
    mapFieldOf<Int, Boolean?>()

    //yields MutableField<Map<Char, Double>> containing the immutable map {'a': 1.5, 'b':
5.4}
    mutableSetFieldOf('a' to 1.5, 'b' to 5.4)
```

Java

```java
//yields Field<Map<Int, Character>> containing the immutable map {1: 'a', 2: 'b'}
FieldUtils.mapFieldOf(new Pair(1, 'a'), new Pair(2, 'b'))

//yields Field<Map<Int, Boolean>> containing an immutable empty map
FieldUtils.<Int, Boolean>mapFieldOf()

//yields MutableField<Map<Character, Double>> containing the immutable map {'a': 1.5, 'b':
5.4}
FieldUtils.mutableSetFieldOf(new Pair('a', 1.5), new Pair('b', 5.4))
```

Manipulating fields

The core aspect that makes this library awesome are transformations. It is very common to change in some way the data before displaying it or passing it to another function. To achieve this but keep the reactive nature of fields around, we must apply a transformation.

Note: all transformations are lazy by default. This means that they won't get computed until they are needed in some way (either someone tries to obtain the value or a listener is registered).

One-way transformations

In most cases, we'll only need a one-way transformation: this means that we get back a simple non-mutable `Field`, and that we only need to supply a single function that takes as input the value of our field and transforms it to another value.

Using `transform()`

This function can be called on any `Field` instance and accepts as single argument a function that converts a value of the original field to a new value, either of the same or different type. (Given a `Field<T>`, the function is defined as `T -> O` where `O` can be any type). The new field will then recalculate its value calling the provided function whenever it detects a change in the original field. For this reason transformation functions should always be pure (have no side effects).

Here's our first example:

Kotlin

```
val number = mutableFieldOf(10)
    val isPositive /* : Field<Boolean> */ = number.transform { it > 0 }
    println(isPositive.value) //Prints true
    number.value = -5
    println(isPositive.value) //Prints false
```

Java

```
MutableField<Integer> number = MutableField.of(10);
Field<Boolean> isPositive = number.transform(n -> n > 0);
System.out.println(isPositive.getValue()) //Prints true
number.setValue(-5);
System.out.println(isPositive.getValue()) //Prints false
```

Using `then()`

A more advanced way of running a field transformation is through the `then()` function: it acts very similarly to `transform()`, but allows to return a `Field` on the transformation function. The returned field will recalculate its value both when the original field changes and when the returned field changes.

In the following example, suppose we have a class `User` that contains a property `username : MutableField<String>`:

Kotlin

```
val currentUser = mutableFieldOf(User(6, mutableFieldOf("Tom")))
    val currentUsername /* : Field<String> */ = currentUser.then { it.username }

    println(currentUsername.value) //Prints "Tom"

    //Changing current user
    val seven = User(7, mutableFieldOf("Dick"))
    currentUser.value = seven
    println(currentUsername.value) //Prints "Dick"

    //Changing username of current user
    seven.username.value = "Harry"
    println(currentUsername.value) //Prints "Harry"
```

Java

```java
class User {
    public final int id;
    public final MutableField<String> username;

    public User(int id, MutableField<String> username) {
        this.id = id;
        this.username = username;
    }
}
MutableField<User> currentUser = MutableField.of(new User(6, MutableField.of("Tom")));
Field<String> currentUsername = currentUser.then(u -> u.username);

System.out.println(currentUsername.getValue()); //Prints "Tom"

//Changing current user
User seven = new User(7, MutableField.of("Dick"));
currentUser.setValue(seven);
System.out.println(currentUsername.getValue()); //Prints "Dick"

//Changing username of current user
seven.username.setValue("Harry");
System.out.println(currentUsername.getValue()); //Prints "Harry"
```

In this example we have transformed a field that contains a user instance into a field that represents the current user's username.

Transforming more than one field at a time

Sometimes we want to create a new field that is calculated by using more than one field as its input. In this case we can call a `transform` or `then` function that accept a number of parameters and pass the respective values to the lambda.

Kotlin

```kotlin
val n1 = mutableFieldOf(1)
    val n2 = mutableFieldOf(2)

    val sum = transform(n1, n2) { a, b ->
        a + b
    }
```

Java

```java
MutableField<Integer> n1 = MutableField.of(1);
MutableField<Integer> n2 = MutableField.of(2);

Field<Integer> sum = FieldUtils.transform(n1, n2, a, b -> a + b);
```

Two-way transformations

While most of the time we'll need to transform data only in one way, sometimes it could be useful to set a "transformed" value, that will work its way back to change the original data. This is obviously doable only in a `MutableField<T>` instance, providing two functions: a `T -> O` to transform the data, and an additional one `O -> T` to convert it backwards. The function name is `twoWayTransform()`.

Here's an example:

Kotlin

```kotlin
val number = mutableFieldOf(5)
    val numberTwoTimes /* : MutableField<Int> */ = number.twoWayTransform({ it * 2 }, { it
/ 2 })
    println(numberTwoTimes.value) //Prints 10

    number.value = 10
    println(numberTwoTimes.value) //Prints 20

    numberTwoTimes.value = 50
    println(number.value) //Prints 25
```

Java

```java
MutableField<Integer> number = MutableField.of(5);
MutableField<Integer> numberTwoTimes = number.twoWayTransform(n -> n * 2, n -> n / 2);
System.out.println(numberTwoTimes.getValue()); //Prints 10

number.setValue(10);
System.out.println(numberTwoTimes.getValue()); //Prints 20

numberTwoTimes.setValue(50);
System.out.println(number.getValue()); //Prints 25
```

Problems with two-way transformations

Two-way transformations have a few gotchas since not all transformations have a perfect opposite one.

For instance, in the above example, the given functions are **not** symmetric (think of odd numbers). What happens if we set `49` to `numberTwoTimes`?

When we change the value of `numberTwoTimes` to `49` the following steps occur:

1. The second transformation function is evaluated with the parameter `49`, which gives `24`;
2. `24` is set to the field `number`;
3. The first transformation is computed with the new value of `24`, which gives `48`;
4. `48` is set to the field `numberTwoTimes`;
5. The second transformation function is once again evaluated with the parameter `48`, which gives, again, `24`;
6. `24` is set to the field `number`, but since it already contained that value, the process stops.

So we effectively have two main problems with uneven transformation functions:

- **Value stability**: after setting a value, a series of events is generated that effectively changes the value we just set.
- **Infinite ping-ponging**: this more serious problem occurs if the transformation functions never reach an agreement on what the global state should be. For instance, imagine if both the transformation functions incremented the value by one: an endless loop between the two fields would ensue, consuming lots of CPU cycles and potentially freezing the program.

In order to avoid these problems we must either accurately choose our transformation functions to be perfectly symmetric, or accept the explained value stability problem. In any case, we should **always** provide functions that ultimately reach an agreement on the field values.

Exceptions in transformation functions

If a transformation function throws an exception that you want to handle, you must do it inside the transformation function itself. For example:

Kotlin

```
val str = mutableFieldOf("10");
    val num = str.transform {
        try {
            it.toInt()
        } catch(nfe: NumberFormatException) {
            null //Default value
        }
    }
```

Java

```
MutableField<Stirng> str = MutableField.of("10");
Field<Integer> str = FieldUtils.transform(str, n -> {
    try {
        return Integer.parseInt(n);
    } catch(NumberFormatException nfe) {
        return null; //Default value
    }
});
```

In the example we've captured the exception in the transformation function and provided a default value, so when the string in `str` is not parsable `num` will become `null`. If however you don't want to handle the exception, you can avoid the `try...catch` block, but it won't be caught anywhere.

What happens if an exception is thrown?

If an exception manages to "escape" the transformation function it will naturally be propagated up the stack to the caller that triggered the field reevaluation. Notice that since fields are lazy by nature the exception could bubble up in different places, and for this reason it shouldn't be caught. The only place to handle exceptions is inside the transformation functions themselves, as shown in the example above.

Utility functions

In order to ease the programmer of writing many redundant code, the library provides a lot of utility functions to deal with fields in the smoothest way possible. Most of theses functions are extension functions on fields of a specific type. While they are mainly a kotlin feature, it's still possible to call them in Java™ code through a static method, but it's a bit cumbersome. For this reason, it is suggested to use kotlin as much possible.

Most functions divide in two categories:

- **Transformations**: they provide default implementations for either one-way or two-way transformation of fields.
- **Setters**: available only on mutable fields, they easily allow to change the content of the field.

A thing to keep in mind is that the `Field` class is declared as `Field<out T>`, so any extension function of `Field<A>` will also be present in `Field<B : A>`, so for instance all extension functions of a `Field<Collection>` can also be called in a `Field<List>`, `Field<Set>` and so on. The same thing does **not** hold for mutable fields.

Furthermore, some functions have been implemented with operator overloading or with infix notation so, additionally to the standard way, they can also be called with their specific syntax.

Java™ usage

All these functions can be called in java code statically on their respective utils class. For extension functions the first parameter must always be the field you would call the function on in kotlin followed by the rest of the parameters. The utils class name is always `FieldUtils` prefixed with the name of the type contained in the field (e.g. for booleans it's `BooleanFieldUtils`). Example:

Kotlin

```
f1.and(f2)
```

Java

```
BooleanFieldUtils.and(f1, f2);
```

Non-complete list

Here you can find a **non**-exhaustive list of extension functions. For the complete list as well as a complete description of what they do, please visit the specific page of the documentation.

In the following examples all fields starting with `f` (`f`, `f1`, `f2`, etc.) are simple `Field`, while all fields starting with `mf` (`mf`, `mf1`, `mf2`, etc.) are `MutableField`.

On all fields

Kotlin

```
//Transformations
val f1 : Field<T>? = ...
f1.orValue(v) //Yields f1 if f1 is not null, otherwise a constant field containing v
f1.orNull()  //Yields f1 if f1 is not null, otherwise a constant field containing null

val f2 : Field<Field<T>> = ...
f2.reduce() //Yields a Field<T> representing the value of the innermost field

val f3 : Field<T?> = ...
f3.notNull(v) //Yields a Field<T> cotaining the value of f3 if this value is not null,
otherwise v

val f4 : Field<T> = ...
f3 eq f4 //Yields a Field<Boolean> that is true if the values of f3 and f4 are equals()
f4 eq v //Yields a Field<Boolean> that is true if the value of f3 and v are equals()
f3 neq f4 //Yields a Field<Boolean> that is true if the values of f3 and f4 are NOT
equals()
```

Java
```
//Transformations
Field<T> f1  = ...; //f1 can be null
FieldUtils.orValue(f, v); //Yields f1 if f1 is not null, otherwise a constant field
containing v
FieldUtils.orNull(f); //Yields f1 if f1 is not null, otherwise a constant field containing
null

Field<Field<T>> f2 = ...;
FieldUtils.reduce(f2) //Yields a Field<T> representing the value of the innermost field

Field<T> f3 = ...; //the value contained by f3 can be null
f3.notNull(v) //Yields a Field<T> cotaining the value of f3 if this value is not null,
otherwise v

Field<T> f4 = ...;
FieldUtils.eq(f3, f4); //Yields a Field<Boolean> that is true if the values of f3 and f4
are equals()
FieldUtils.eq(f4, v); //Yields a Field<Boolean> that is true if the value of f4 and v are
equals()
FieldUtils.neq(f3, f4); //Yields a Field<Boolean> that is true if the values of f3 and f4
are NOT equals()
```

On Boolean fields

Kotlin
```
//Transformations
f.not()
f1 and f2 //Also or and xor

//Setters
mf.toggle()
```

Java
```
//Transformations
BooleanFieldUtils.not(f);
BooleanFieldUtils.and(f1, f2); //Also or and xor

//Setters
BooleanFieldUtils.toggle(mf);
```

On Int fields

Kotlin
```
//Transformations
(((f1 + 5) / 4) % -f2) * 17 //All operations between two Int fields or a field and an Int
f.abs() //Absolute value
min(f, 5) //min and max between two Int fields or a field and an Int
f.toLong() //Convert to long, float and double

//Setters
mf.increment() //Also decrement()
mf.negate()
```

Java

```java
//Transformations
IntFieldUtils.times(IntFieldUtils.rem(IntFieldUtils.div(IntFieldUtils.plus(f1, 5), 4),
IntFieldUtils.unaryMinus(f2)), 17); //All operations between two Integer fields or a field
and an Integer. Since in java it's particularly verbose, in this case it is probably better
using the transform() function manually
IntFieldUtils.abs(f) //Absolute value
IntFieldUtils.min(f, 5) //min and max between two Integer fields or a field and an Integer
IntFieldUtils.toLong(f) //Convert to long, float and double

//Setters
IntFieldUtils.increment(mf); //Also decrement()
IntFieldUtils.negate(mf);
```

On Long fields

The same functions that are available on `Int` fields are also available on `Long` fields with the appropriate modifications.

On `Float` fields

Kotlin

```kotlin
//Transformations
(((f1 + 5f) / 4f) % -f2) * 17f //All operations between two Float fields or a field and a
Float
f1.pow(f2) //f1 elevated to the power of f2
sin(f3) //Sin of f3
f4.roundToLong() //f4 rounded to Long

//Setters
mf.increment(1.5f) //Also decrement()
mf.negate()
```

Java

```java
//Transformations
FloatFieldUtils.times(FloatFieldUtils.rem(FloatFieldUtils.div(FloatFieldUtils.plus(f1, 5f),
4f), FloatFieldUtils.unaryMinus(f2)), 17f); //All operations between two Float fields or a
field and an Float. Since in java it's particularly verbose, in this case it is probably
better using the transform() function manually
FloatFieldUtils.pow(f1, f2) //f1 elevated to the power of f2
FloatFieldUtils.sin(f3) //Sin of f3
FloatFieldUtils.roundToLong(f4) //f4 rounded to Long

//Setters
FloatFieldUtils.increment(mf, 1.5f); //Also decrement()
FloatFieldUtils.negate(mf);
```

On `Double` fields

The same functions that are available on `Float` fields are also available on `Double` fields with the appropriate modifications.

On `String` fields

Kotlin

```
//Transformations
f1 + " world" //Concatenates the strings
f1 + f2 //Concatenates the strings
```

Java

```
//Transformations
StringFieldUtils.concatenate(f1, " world"); //Concatenates the strings
StringFieldUtils.concatenate(f1, f2); //Concatenates the strings
```

On `CharSequence` fields

Kotlin

```
//Transformations
f.length() //Yields a Field<Int> containing the length of the CharSequence
```

Java

```
//Transformations
CharSequenceFieldUtils.length(f) //Yields a Field<Integer> containing the length of the
CharSequence
```

On `Pair` fields

Kotlin

```
//Transformations
f.first() //Yields a Field<T> containing the first element of the pair in f
f.second() //Yields a Field<T> containing the second element of the pair in f
```

Java

```
//Transformations
PairFieldUtils.first(f); //Yields a Field<T> containing the first element of the pair in f
PairFieldUtils.second(f); //Yields a Field<T> containing the second element of the pair in
f
```

On `Triple` fields

Kotlin

```
//Transformations
f.first() //Yields a Field<T> containing the first element of the triple in f
f.second() //Yields a Field<T> containing the second element of the triple in f
f.third() //Yields a Field<T> containing the third element of the triple in f
```

Java

```
//Transformations
TripleFieldUtils.first(f); //Yields a Field<T> containing the first element of the triple
in f
TripleFieldUtils.second(f); //Yields a Field<T> containing the second element of the triple
in f
TripleFieldUtils.third(f); //Yields a Field<T> containing the third element of the triple
in f
```

On `Comparable` fields

Kotlin

```
//Transformations
f1 gt f2 //Yields a Field<Boolean> that is true when f1 > f2
f2 lte 5 //Yields a Field<Boolean> that is true when f2 <= 5
4 gte f3 //Yields a Field<Boolean> that is true when 4 >= f3
```

Java

```
//Transformations
ComparableFieldUtils.gt(f1, f2); //Yields a Field<Boolean> that is true when f1 > f2
ComparableFieldUtils.lte(f2, 5); //Yields a Field<Boolean> that is true when f2 <= 5
ComparableFieldUtils.get(4, f3); //Yields a Field<Boolean> that is true when 4 >= f3
```

On `Collection` fields

Kotlin

```
//Transformations
f.isEmpty() //Yields a Field<Boolean> that is true when the content of f is empty
f.size() //Yields a Field<Int> that contains the size of the collection contained in f

//Setters
mf.add(x) //Creates a new immutable collection consisting of the collection in f appended
with x, then sets it to the field
mf.remove(y) //Creates a new immutable collection consisting of the collection in f without
y, then sets it to the field
```

Java

```
//Transformations
CollectionFieldUtils.isEmpty(f); //Yields a Field<Boolean> that is true when the content of
f is empty
CollectionFieldUtils.size(f); //Yields a Field<Integer> that contains the size of the
collection contained in f

//Setters
CollectionFieldUtils.add(f, x); /Creates a new immutable collection consisting of the
collection in f appended with x, then sets it to the field
CollectionFieldUtils.remove(f, y); //Creates a new immutable collection consisting of the
collection in f without y, then sets it to the field
```

Nearly all the extension functions of collections of Kotlin™ are available as an utility transformation function in Dataflow. A complete list can be found in the Kotlin documentation.

Most common ways of changing a collection are available but be careful! Each change you perform will create a new immutable list and discard the previous one, so you should consider every operation `Ω(n)` with respect to the collection size. The reason for this behavior is explained in detail in the section Restrictions on field values.

On `Iterable` fields

Kotlin

```
//Transformations
f.contains(x) //Yields a Field<Boolean> that is true when the x is contained in the field
f.filter(/* predicate */) //Yields a Field<Iterable<T>> that contains the values in f that
match the predicate
f.min() //Available on field of Iterable of numbers and Comparable, yields a Field<T>
containing the min value (Also max())
```

Java

```
//Transformations
IterableFieldUtils.contains(f, x) //Yields a Field<Boolean> that is true when the x is
contained in the field
IterableFieldUtils.filter(f, /* predicate */) //Yields a Field<Iterable<T>> that contains
the values in f that match the predicate
IterableFieldUtils.min(f); //Available on numbers and Comparable, yields a Field<T>
containing the min value (Also max())
```

Nearly all the extension functions of iterables of Kotlin™ are available as an utility transformation function in Dataflow. A complete list can be found in the Kotlin documentation.

On `List` fields

Kotlin

```
//Transformations
f.firstOrNull(x) //Yields a Field<T> that contains the first element of the list in f, or
null if it doens't exist

//Setters
mf.add(x) //Creates a new immutable list consisting of the list in f appended with x, then
sets this list to the field
mf.shuffle() //Creates a new immutable list with the same elements of the old one shuffled,
then sets this list to the field
```

Java

```
//Transformations
ListFieldUtils.firstOrNull(f, x) //Yields a Field<T> that contains the first element of the
list in f, or null if it doens't exist

//Setters
ListFieldUtils.add(mf, x); //Creates a new immutable list consisting of the list in f
appended with x, then sets this list to the field
ListFieldUtils.shuffle(mf); //Creates a new immutable list with the same elements of the
old one shuffled, then sets this list to the field
```

Nearly all the extension functions of lists of Kotlin™ are available as an utility transformation function in Dataflow. A complete list can be found in the Kotlin documentation.

Most common ways of changing a list are available but be careful! Each change you perform will create a new immutable list and discard the previous one, so you should consider every operation `Ω(n)` with respect to the list size. The reason for this behavior is explained in detail in the section Restrictions on field values.

On `Set` fields

Kotlin

```
//Setters
mf.add(x) //Creates a new immutable set consisting of the set in f appended with x, then
sets it to the field
mf.clear() //Creates a new immutable empty set and sets to the field
```

Java

```
//Setters
SetFieldUtils..add(mf, x); //Creates a new immutable set consisting of the set in f
appended with x, then sets it to the field
SetFieldUtils.clear(mf); //Creates a new immutable empty set and sets to the field
```

On Map fields

Kotlin

```
//Transformations
f.keys() //Yields a Field<Set<K>> that contains the keys of the map in f
f.count { /* predicate */ } //Yields a Field<Int> that contains the number of entries that
satisfy the predicate

//Setters
mf.put(v1, v2) //Creates a new immutable map consisting of the map in f with the mapping
(v1)=>(v2) added, then sets it to the field
```

Java

```
//Transformations
MapFieldUtils.keys(f); //Yields a Field<Set<K>> that contains the keys of the map in f
MapFieldUtils.count(f, /* predicate */); //Yields a Field<Integer> that contains the number
of entries that satisfy the predicate

//Setters
MapFieldUtils.put(mf, v1, v2) //Creates a new immutable map consisting of the map in f with
the mapping (v1)=>(v2) added, then sets it to the field
```

Nearly all the extension functions of maps of Kotlin™ are available as an utility transformation function in Dataflow. A complete list can be found in the Kotlin documentation.

Most common ways of changing a map are available but be careful! Each change you perform will create a new immutable map and discard the previous one, so you should consider every operation `Ω(n)` with respect to the map size. The reason for this behavior is explained in detail in the section Restrictions on field values.

Listening fields

The last main step to learn Dataflow is to listen for changes in fields. There are two main ways of doing that, described in their own page.

The suggested method to use is the **second** one.

Using ListenersManager

This is the more low-level way of listening fields: you pass callback which gets added to a list and is called whenever the field changes. You can also remove them when they're not necessary anymore. All of this is handled by the `ListenersManager` class: each field has one associated instance that is exposed through the `listeners` property.

Note: the lifecycle method for listening field changes should always be your first choice. You should read this page to understand how listeners work internally, but you should always use the lifecycles when possible.

Weak vs strong

When we set a listener we must do it in one of two ways:

- **Strongly**: the `ListenersManager` holds a reference of the callback until it is manually removed. This means that if the callback implicitly holds some references, they'll be kept as well. When using this mode you must make sure to cancel the callback as soon as it's not needed anymore to prevent memory leaks.
- **Weakly**: in this mode the `ListenersManager` hold a `WeakReference` to the callback. This means that when all other references are lost it internally becomes `null` and the callback is then removed when the manager notices. While this eases the programmer from the duty of removing the callback, it also gives them the responsibility to store a reference until it's needed: usually this is done by keeping it inside an object with a *lifecycle*, for example an Android™ `Activity`.

Usually it's preferable to use the weak mode, while the strong mode should be used only when you know the exact moment you need to remove the listener.

Registering a listener

The `ListenersManager` in fields accepts an instance of `FieldListener`. It has a single function called `onFieldChanged()` that passes you the field that changed and its new data. The callback is called on the same thread that triggered it (i.e. the thread that changed the value).

Strong listeners

Adding a strong listener is pretty straightforward. See this example:

Kotlin

```kotlin
val f = mutableFieldOf("hello")
    f.listeners.addStrongly { newData -> //Can also have a `source` parameter
        println(newData)
    }
```

Java

```java
f.listeners().addStrongly((source, newData) -> {
    System.out.println(newData);
});
```

Since kotlin's support of SAM conversions works only on Java™ interfaces, the two `addStrongly` functions you see used in kotlin are extension functions.

Weak listeners

The process to add a weak listener is a bit more complicated as we must keep the reference to the `FieldListener` instance somewhere.

Kotlin

```kotlin
class ObjectWithLifecycle {
        //`fieldListener()` requires also have the source as first paramter
        val listener = simpleFieldListener<String> {
            println(it)
        }
    }

    val f = mutableFieldOf("hello")
    val obj = ObjectWithLifecycle()
    f.listeners.addWeakly(obj.listener)
```

Java

```java
class ObjectWithLifecycle {
    public final FieldListener<Integer> listener = (source, newData) -> {
        System.out.println(newData);
    };
}

ObjectWithLifecycle obj = new ObjectWithLifecycle();
f.listeners().addWeakly(obj.listener);
```

Doing this the only strong reference is held by the `ObjectWithLifecycle` instance, so when it gets garbage collected, the listener dies with him.

A more compact way of doing it is directly implementing the `FieldListener` interface in the object:

Kotlin

```kotlin
class ObjectWithLifecycle : FieldListener<String> {

        override fun onFieldChanged(source: Field<String>, newData: String) {
            println(newData)
        }
    }

    val f = mutableFieldOf("hello")
    val obj = ObjectWithLifecycle()
    f.listeners.addWeakly(obj)
```

Java

```java
class ObjectWithLifecycle implements FieldListener<Integer> {

    public void onFieldChanged(Field<Int> source, Integer newData) {
        System.out.println(newData);
    }
}

ObjectWithLifecycle obj = new ObjectWithLifecycle();
f.listeners().addWeakly(obj);
```

With this technique we can avoid the creation of one object (the listener instance) by implementing it directly on an already existing object. Obviously we can do this only for one field since we can implement only on time the same interface; this leaves us with two possibilities: either we listen only one field or we register multiple fields and check with the source object which one triggered the callback before performing the appropriate action. If both seem unacceptable, we can instantiate one listener for each field and keep it in the object, like in the first example.

Canceling a listener

If we want to manually cancel a listener we can do so by calling the `remove()` function passing the listener instance as its only parameter, like so:

Kotlin

```kotlin
f.listeners.remove(listener)
```

Java

```java
f.listeners().remove(listener);
```

This works for both strongly and weakly set listeners.

Using lifecycles

This way of listening fields has a very similar concept to the weak way of the previous page: we still register weak listeners inside the `ListenersManager`, but our reference are automatically kept inside the object's lifecycle. This is possible by implementing the `LifecycleOwner` interface. The callback is still called by the thread that triggered it.

The `LifecycleOwner` interface

This interface has two main functions to implement:

- `attachToLifecycle()`: accepts a `ListenerHolder` instance that we must attach in some way to the object
- `detachFromLifecycle()`: accepts a `ListenerHolder` instance that we must remove from the object

`ListenerHolder` is a particular class that holds the callback and some associated info; you cannot directly instantiate it.

This interface also has multiple versions of a `listen()` function: given its parameters it creates a `ListenerHolder` and calls `attachToLifecycle()`. After that it returns the created `ListenerHolder`.

Whenever you want to listen to a field and attach the callback to a `LifecycleOwner` all you have to do is call `listen()` on it passing the callback: everything else will automatically be taken care of.

The `BaseLifecycleOwner` class

This is the basic implementation of `LifecycleOwner` that already provides all the needed functions to work.

We can now see a brief example on an object extending `BaseLifecycleOwner`:

Kotlin

```kotlin
class ObjectWithLifecycle : BaseLifecycleOwner() {
    }

    val f = mutableFieldOf(10)
    val obj = ObjectWithLifecycle()
    obj.listen(f) {
        println(it)
    }
    f.value = 42
```

Java

```java
class ObjectWithLifecycle extends BaseLifecycleOwner {
}
MutableField<Integer> f = MutableField.of(10)
ObjectWithLifecycle obj = new ObjectWithLifecycle();
obj.listen(f, newData -> {
    System.out.println(newData);
});
f.setValue(42);
```

A thing we should keep in mind is that calling `listen()` will also execute the callback the first time. If you do not want this behavior you can set the parameter `callImmediately` to `false`.

Please note that the callbacks listeners are registered and listening to changes until the garbage collector destroys the `BaseLifecycleOwner`. If you don't like this behavior, you can manually call `stop()` and `destroy()` methods.

Global lifecycle

If you need to register a listener for the whole duration of the program you can use a special lazily-instantiated `LifecycleOwner` instance, called global lifecycle:

Kotlin

```kotlin
val f = mutableFieldOf("hello")
    globalLifecycle.listen(f) {
        println(it)
    }
    println("world")
```

Java

```java
MutableField<String> f = MutableField.of("hello");
LifecycleOwner.globalLifecycle.listen(f, (v) -> {
    System.out.println(v);
});
f.setValue("world");
```

You could technically remove listeners from the global lifecycle, but doing so is discouraged since the listener's actual lifecycle wouldn't actually be global.

Block lifecycle

Last but not least, we can call the `lifecycle()` function providing a lambda that is immediately evaluated and that accepts as its receiver the newly created `LifecycleOwner`; as soon as it terminates the `LifecycleOwner` and all its listeners are stopped.

This method is meant for short-lived listeners whose life-span isn't necessarily attached to another already existing object.

Example:

Kotlin

```kotlin
val f = mutableFieldOf(1)
    f.value = 2
    f.value = 3
    lifecycle {
        listen(f) {
            print("$it ")
        }
        f.value = 4
        f.value = 5
    }
    f.value = 6
    f.value = 7

    //Outputs 3 4 5
```

Java
```java
MutableField<Integer> f = MutableField.of(1);
f.setValue(2);
f.setValue(3);
LifecycleOwner.lifecycle(lc -> {
    lc.listen(f, v -> {
        System.out.print(v + " ");
    });
    f.setValue(4);
    f.setValue(5);
});
f.setValue(6);
f.setValue(7);

//Outputs 3 4 5
```

As soon as the lambda exits, all listeners are canceled.

The `LifecycleOwner.deferred()` function

You can pass an `Executor` to this function and obtain a new `LifecycleOwner` whose listeners will be called on the given executor.

Advanced

Managing collections

Up until this point we've seen examples where fields where mainly containing very basic values such as `Int` or `String`. We'll now discuss how to manage collections inside fields.

First of all, recall that we can set only immutable values inside fields. For this reason Kotlin™ collections already provide some help since they already have the distinction between mutable and read-only collections.

Be careful: read-only doesn't necessarily mean immutable! We could have a read-only collection that changes because its backed by some other mutable collection. Also, we could easily cast a mutable collection to a read-only one. For this reasons you should try to user the idiomatic functions to create fields containing collections whenever possible.

Creating fields

The library provides multiple ways of idiomatically create fields of **immutable** collections:

Kotlin

```kotlin
listFieldOf(1, 2, 3); //Yields Field<List<Int>>
mutableListFieldOf(1, 2, 3); //Yields MutableField<List<Int>>
```

Java

```java
FieldUtils.listFieldOf(1, 2, 3); //Yields Field<List<Integer>>
FieldUtils.mutableListFieldOf(1, 2, 3); //Yields MutableField<List<Integer>>
```

Notice that in both cases the value is an **immutable** `List`, even if the field is itself mutable. To change the value we must create a new list.

Similar methods exist for `Set`s and `Map`s. Examples can be found here.

Transformations

Here we'll explain how to run transformations when our data starts getting a little bit more complicated. In all the following examples we'll obtain the same thing in different ways. Suppose we have this data structure:

Kotlin

```kotlin
class Movie(val name : MutableField<String>, val year : MutableField<Int>)

val movies : Field<List<Movie>>
```

Java
```java
class Movie {
    public final MutableField<String> name;
    public final MutableField<Integer> year;

    //Constructor omitted
}

Field<List<Movie>> movies;
```

And we want to obtain a `Field<List<String>>` whose value will looks like this: `["Avatar 2009", "Titanic 1999", ...]`, and want this `Field` to automatically change as soon as the list of movie changes, or one of the properties of a movie changes.

In each section we'll use a more advanced and high-level function improving each time readability, ease-of-use and performance.

Using plain-old `then()`

Kotlin
```kotlin
val f = movies.then { list ->
        val allFields /* : List<Field<String>> */ = list.map { movie ->
            //The operator + is overridden for Field<String>
            movie.name + " (" + movie.year + ")"
        }
        transform(allFields) { it } //Yields Field<List<String>>
    }
    println(f.value)
```

Java
```java
Field<List<String>> f = movies.then(list -> {
    List<Field<String>> allFields = list.stream().map(movie ->
        transform(movie.name, movie.year, (name, year) -> name + " (" + year + ")"  )
    ).collect(Collectors.toList());
    return transform(allFields, l -> l); //Yields Field<List<String>>
});
System.out.println(f.getValue());
```

- **Pros**: uses already learnt paradigms.
- **Cons**: difficult to read and write; recalculates all values for each change.

Snapshots

We'll now introduce a new concept: snapshots! When the value of a field is a collection we can call the `snapshot()` function: we'll pass a lambda that will be called once for each item in the collection. In this lambda you must return a field whose content will be captured by the snapshot.

The example will clarify:

Kotlin

```kotlin
val f = movies.snapshot { movie ->
        //Returning a Field<Pair<String, Int>>
        movie.name pair movie.year
    }.transform { snapshot -> //Transforming the Field<IterableSnapshot>
        snapshot.originalIterable.map { movie ->
            //Obtaining from the snapshot the value of the fields
            val (name, year) = snapshot[movie]
            "$name ($year)"
        }
    }
    println(f.value)
```

Java

```java
Field<List<String>> f = IterableFieldUtils.snapshot(movies, movie -> {
    //Returning a Field<Pair<String, Int>>
    return movie.name.pair(movie.year);
}).transform( snapshot -> { //Transforming the Field<IterableSnapshot>
    return snapshot.originalIterable().stream().map(movie -> {
        //Obtaining from the snapshot the value of the fields
        Pair<String, Int> pair = snapshot.get(movie);
        return pair.getFirst() + " (" + pair.getSecond() + ")";
    }).collect(Collectors.toList());
});
System.out.println(f.getValue());
```

- **Pros**: most flexible; recalculates only needed values.
- **Cons**: difficult to read and write; lot of code.

For more info about snapshot see the `IterableSnapshot` class doc.

Transform for snapshots

We'll now learn to use a version of the transform function that builds on top of `snapshot()` but is easier to use.

The transform function will accept a number of lambdas where each one accepts an element of the collection and returns a field. Then, as last parameter, a lambda must be provided that accepts the collection contained in the field and one function for each lambda with the following signature `T.() -> I` where `T` is an element of the collection and `I` is the type of the fields returned by the corresponding lambda.

In other words, all lambdas but the last one are responsible to creating different views of the data contained in the iterable, while the last lambda is responsible to combining all the views in a single object, that will be contained in the returning field.

Example:

Kotlin

```kotlin
val f = movies.transform(
        { it.name },
        { it.year },
        { collection /* : List<Movie> */, name, year ->
            //name and year are functions that accept a movie and return a string (the name) and an int (the year).
            //Kotlin allows also the syntax movie.function() instead of function(movie) in this case
            collection.map { m ->
                m.name() + " (" + m.year() + ")"
                //Same as calling name(m) + " (" + year(m) + ")"
            }
        }
    )
    println(f.value)
```

Java

```java
Field<List<String>> f = IterableFieldUtils.transform(movies,
    m -> m.name,
    m -> m.year,
    (/* List<Movie> */ collection, name, year) ->
        collection.stream().map(m ->
            name.invoke(m) + " (" + year.invoke(m) + ")"
        ).collect(Collectors.toList())
);
System.out.println(f.getValue());
```

- **Pros**: cleaner and easy to write, so-so to read; recalculates only needed values.
- **Cons**: slightly less flexible.

Utility functions

We finally arrived at the last iteration of our example, using already provided utility functions that use internally the previous described functions.

For this example we'll need the `mapF()` function:

Kotlin

```kotlin
val f = movies.mapF { m ->
        m.name + " (" + m.year  + ")"
    }
    println(f.value)
```

Java

```java
Field<List<String>> f = IterableFieldUtils.mapF(movies, m ->
    FieldUtils.transform(m.name, m.year, (name, year) -> name + " (" + year + ")")
);
System.out.println(f.getValue());
```

- **Pros**: super-easy to write and read; recalculates only needed values; immediately know what's going on; less code.
- **Cons**: specialized function.

There are a ton of utility extension functions that can run this kind of transformations. A complete list can be found here.

Changing value

Changing the value of collection fields is a delicate process since a new one must be created for each modification. There are several helper methods will automatically do this for you, for example:

Kotlin

```kotlin
val f = mutableListFieldOf("hello")
    val listBefore = f.value
    println(listBefore) //["hello"]

    f.add("world") //Will create a new list ["hello", "world"] and set it to the field,
atomically

    val listAfter = f.value
    println(listAfter) //["hello", "world"]
    println(listBefore === listAfter) //false, different object
```

Java

```java
MutableField<List<String>> f = FieldUtils.listFieldOf(1, 2, 3);
List<String> listBefore = f.getValue();
System.out.println(listBefore); //["hello"]

FieldListUtils.add(f, "world"); //Will create a new list ["hello", "world"] and set it to
the field, atomically

List<String> listAfter = f.getValue();
System.out.println(listAfter); //["hello", "world"]
System.out.println(listBefore === listAfter) //false, different object
```

Virtually all methods of `List`, `Set` and `Map` are available with similar behavior. It goes without saying that doing this repeatedly is not very efficient, for this reason subsequent changes should be batched together.

Wrapping fields

A `FieldWrapper<T>` is a specialization of `MutableField<T>` that allows the method `setField()`. When this method is called passing a field instance, what it does is basically mirror its behavior, so when its value changes our `FieldWrapper` value will also change.

This pattern is pretty useful when we want a class to accept both a field and an actual value. For instance, imagine we're developing a class similar to Android™'s `TextView`. We could do something like this:

Kotlin

```kotlin
fun main() {
    class TextView : BaseLifecycleOwner() {
        //We implement BaseLifecycleOwner so we can attach listeners to our TextView

        //This is the publicly available property to set the text in this TextView
        val text = fieldWrapperOf("<no movie>") //Initial text

        init {
            //We listen to the changes in the text and draw it
            listen(text) {
                draw(it)
            }
        }

        //Method to draw a string
        private fun draw(str: String) {
            println(str)
        }
    }

    val avatar = Movie(mutableFieldOf("Avatar"), mutableFieldOf(2009))
    val tv = TextView()

    //All the following calls are valid
    tv.text.value = "Hello world"
    tv.text.setField(avatar.name) //avatar.name is a MutableField<String>
    avatar.name.value = "Blue"
    tv.text.value = "foo bar"
    avatar.name.value = "Avatar"

    //Outputs:
    //<no movie>
    //Hello world
    //Avatar
    //Blue
    //foo bar
```

Java

```java
final class TextView extends BaseLifecycleOwner {
    //We implement BaseLifecycleOwner so we can attach listeners to our TextView

    @NotNull
    @Override
    public Map<Object, ListenerHolder<?>> getAttachedItems() {
        return attachedItems;
    }

    //This is the publicly available property to set the text in this TextView
    public final FieldWrapper<String> text = FieldWrapper.of("<no movie>"); //Initial text

    public TextView() {
        //We listen to the changes in the text and draw it
        listen(text, this::draw);
    }

    //Method to draw a string
    private void draw(String str) {
        System.out.println(str);
    }
}

Movie avatar = new Movie(MutableField.of("Avatar"), MutableField.of(2009));
TextView tv = new TextView();

//All the following calls are valid
tv.text.setValue("Hello world");
tv.text.setField(movie.name); //movie.name is a MutableField<String>
avatar.name.setValue("Blue");
tv.text.setValue("foo bar");
avatar.name.setValue("Avatar");

//Outputs:
//<no movie>
//Hello world
//Avatar
//Blue
//foo bar
```

When we set a value (e.g. `"foo bar"`) the previous field is detached and the wrapper will not receive any more updates from it.

It is also possible to manually detach the field calling the method `detachField()`: in this case the value will stay the same, but the wrapper will not receive any more updates.

Consistency problems

As a final warning we must analyze what happens when a field depends on another field multiple times, either directly or indirectly.

Take this example:

Kotlin

```kotlin
val num = mutableFieldOf(10)
    val isEven = num % 2 eq 0
    val isOdd = num % 2 eq 1

    val evenOrOdd = isEven or isOdd
    lifecycle {
        listen(num) {
            println("num = $it")
        }
        listen(evenOrOdd) {
            println("evenOrOdd = $it")
        }
        println()
        num.value = 20
        num.value = 41
        num.value = 56
    }
```

Java

```java
MutableField<Integer> num = MutableField.of(10);
Field<Boolean> isEven = num.transform(n -> n % 2 == 0);
Field<Boolean> isOdd = num.transform(n -> n % 2 == 1);

Field<Boolean> evenOrOdd = FieldUtils.transform(isEven, isOdd, (even, odd) -> even || odd);
LifecycleUtils.lifecycle(lc -> {
    lc.listen(num, n -> {
        System.out.println("num = " + n);
    });
    lc.listen(evenOrOdd, eoo -> {
        System.out.println("evenOrOdd = " + eoo);
    });
    System.out.println();
    num.setValue(20);
    num.setValue(41);
    num.setValue(56);
});
```

Logically the value of `evenOrOdd` should always be true, since the number contained in `num` must be either even or odd. But since `evenOrOdd` depends on `num` indirectly **two times**, this cannot be guaranteed, since the propagation order of fields changes is not well defined.

If you run the example you'll see that, even if for a brief moment, `evenOrOdd` assumes the value of `false`.

Note that it **is** guaranteed that the field will eventually reach the correct state. The only thing that is **not** guaranteed when a field depends on another one multiple times is the fact that it won't never assume an "impossible" value.

Resolution

To solve this problem we can simply remove the multiple dependency by merging the transformations.

For instance, the above example is fixed like so:

Kotlin

```kotlin
val num = mutableFieldOf(10)
    val evenOrOdd = num.transform {
        it % 2 == 0 || it % 2 == 1
    }
    lifecycle {
        listen(num) {
            println("num = $it")
        }
        listen(evenOrOdd) {
            println("evenOrOdd = $it")
        }
        println()
        num.value = 20
        num.value = 41
        num.value = 56
    }
```

Java

```java
MutableField<Integer> num = MutableField.of(10);
Field<Boolean> evenOrOdd = num.transform(n -> n % 2 == 0 || n % 2 == 1);

LifecycleUtils.lifecycle(lc -> {
    lc.listen(num, n -> {
        System.out.println("num = " + n);
    });
    lc.listen(evenOrOdd, eoo -> {
        System.out.println("evenOrOdd = " + eoo);
    });
    System.out.println();
    num.setValue(20);
    num.setValue(41);
    num.setValue(56);
});
```

As you can see in this example the value of `evenOrOdd` will never assume `false`.