# Diffing! (In Java) A Refactoring To Functional Case Study

Manuel Paccagnella
Lambda Land, manuel.paccagnella@gmail.com

Version 1, 27 December 2015

## Contents

# 1 Abstract

This document describes an incremental refactoring of a piece of Java 8 code from imperative to functional style. The main goal of this article is to give to the reader a glipse on the principles and techniques behind *functional programming* (FP), not to teach Java or a particular FP library for this language. So, the reader is expected to read this article using API references and other supporting material to fill-in the details.

# 2 Context

The code presented in this article is made-up, although I've seen code like this in the wild. You can find it on GitHub and follow along.

Suppose we have a social platform for book reviews a-la GoodReads. We want to add a feature that tracks changes on a review, field by field. Our data model is this:

```
data Review = Review String Username LocalDateTime Rating String
data Difference = Difference String String String
```

> This notation allows to concisely define new *data types*. In particular with:
>
> ```
> data Difference = Difference String String String
> ```
>
> We define a new type called `Difference`, whose values can be created/expressed using the *data constructor* with the same name which requires three arguments, all of them of type `String`. In other words, we have defined a data type called `Difference` which contains three values of type `String`.
>
> In this domain they are:
>
> 1. Description of the field
> 2. Human-readable `String` representation of the old value
> 3. Human-readable `String` representation of the new value
>
> An alternative formulation would be:

```
data Difference = Difference {description :: String,
                              oldValue :: String,
                              newValue :: String}
```

Here is our starting point, the original Java implementation:

```
public class Diff {
  public static List<Difference> diff(Review x, Review y) {
    if (x == null || y == null) throw new IllegalArgumentException(
        "Reviews shouldn't be null!");

    List<Difference> changes = new ArrayList<>();
    if (!Objects.equals(x.getTitle(), y.getTitle())) changes.add(
        difference("Title", Utils.formatValue(x.getTitle()),
                   Utils.formatValue(y.getTitle())));
    if (!Objects.equals(x.getTitle(), y.getTitle())) changes.add(
        difference("Username", Utils.formatValue(x.getUsername()),
                   Utils.formatValue(y.getUsername())));
    if (!Objects.equals(x.getTitle(), y.getTitle())) changes.add(
        difference("Updated on", Utils.formatValue(x.getUpdated()),
                   Utils.formatValue(y.getUpdated())));
    if (!Objects.equals(x.getTitle(), y.getTitle())) changes.add(
        difference("Rating", Utils.formatValue(x.getRating()),
                   Utils.formatValue(y.getRating())));
    if (!Objects.equals(x.getTitle(), y.getTitle())) changes.add(
        difference("Text", Utils.formatValue(x.getText()),
                   Utils.formatValue(y.getText())));
    return changes;
  }
}
```

# 3 First refactoring: DRY!

The first thing we notice is that there are *a lot* of repetitions there, plus several copy&paste-induced errors (we always compare the review title for every field, which is incorrect). This code is extremely repetitive, there are basically no abstractions, and is very imperative.

The very first thing we should do is *removing duplications by creating an abstraction*. In other words, we need to apply the DRY principle: when we found ourselves to write the same code structure more than once, it's time to create an abstraction and use it to avoid repetitions. Often, this means **creating a method instead of copy-pasting**.

Let's extract a method that encapsulates the `Difference` values creation logic:

```
private static <T> void compareReviewField(String description, T oldValue,
                                           T newValue,
```

```
                                          List<Difference> changes) {
  if (!Objects.equals(oldValue, newValue)) changes.add(
      difference(description, Utils.formatValue(oldValue),
                 Utils.formatValue(newValue)));
}
```

This way the main method becomes more concise, readable and maintainable:

```
public static List<Difference> diff(Review x, Review y) {
  if (x == null || y == null) throw new IllegalArgumentException(
      "Reviews shouldn't be null!");

  List<Difference> changes = new ArrayList<>();
  compareReviewField("Title", x.getTitle(), y.getTitle(), changes);
  compareReviewField("Username", x.getUsername(), y.getUsername(), changes);
  compareReviewField("Updated on", x.getUpdated(), y.getUpdated(), changes);
  compareReviewField("Rating", x.getRating(), y.getRating(), changes);
  compareReviewField("Text", x.getText(), y.getText(), changes);
  return changes;
}
```

Notice that we also *removed a bug*: by expressing the common logic in a generic way, we can't make copy&paste errors.

### 3.1 Summary

We have created a new *abstraction* that we have reused several times in order to:

- Write less code.
- Make it more readable (and maintainable).
- Avoid repetitions (which as we've seen are error-prone).

## 4 Second refactoring: Purity!

Let's begin to use a more functional approach making our `#diff()` method a *pure function*.

### 4.1 Pure functions

A `y = f(x)` function is said to be *pure* when, quoting Wikipedia:

> [...] both below statements about the function hold:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices [...].
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices [...] .

The result value need not depend on all (or any) of the argument values. However, it must depend on nothing other than the argument values. The function may return multiple result values and these conditions must apply to all returned values for the function to be considered pure. If an argument is call by reference, any parameter mutation will alter the value of the argument outside the function, which will render the function impure.

Summarizing, a function is pure when:

- It does not do *any* side-effect (printing things, talking over the network, reading/writing files or a DB, etc).
- It produces a result, and the computation depends *only* on the input parameters. For the same input, it *always* produces the same result.
- The input parameters are not modified (they are *immutable*).

Working with pure functions is easier because:

- They are easier to reason about: you don't have to take into account external dependencies, temporal couplings, global state, etc. For the same reason, they are also *much easier to test*. We can have a very useful property called **referential transparency**: "every *expression* `e` can be substituted everywhere it occurs in the program `p` with the result of its evaluation, without altering the meaning of `p`"[1]. This means that you can reason *algebrically* about your program and do mechanical refactorings with much more confidence[2].
- We have *composability* at our disposal: if their types are compatible, pure functions can be freely combined in a lot of ways. This is not possibile with *impure* functions since there may be side-effects that must occur (or not) in a certain sequence, or if some things must be done (or not) together.

Throughout this article we'll use a simple notation[3] to express function signatures and types declarations, which is much more concise (and expressive) than Java code. For example:

```
add1 :: Int -> Int
```

[1] Thanks to Rúnar Bjarnason for this definition!
[2] Of course this is especially true with *statically-typed* languages.
[3] This notation is valid Haskell code, surprise!

This example denotes the *signature* of a function called `add1`, which takes a value of type `Int` and returns a value of type `Int`. This is just the signature, we're not interested in the implementation right now.

A function with two parameters can be expressed like this:

```
add :: Int -> Int -> Int
```

This signature reads like: "here we define a function called `add` that takes an `Int` value, then another `Int` value, and finally computes another `Int` value". There is no ambiguity in what's a parameter type and where is the return type since *every function has always only one return value, the last one.*

> Intuitively, that's how to read this signatures. The correct way however is this:
>
> ```
> add :: Int -> (Int -> Int)
> ```
>
> `add` is a one parameter function that returns another one parameter function, that in this case will compute the result. In general, an $n$ parameters function is equivalent to a "chain" of $n$ one parameter functions, and you can go from one formulation to the other through a technique called currying (and viceversa with the dual: *uncurrying*). "Curried" functions are very powerful and useful, but you'll see it more and more as you explore functional programming and apply it.

Another example is the signature of our method `#diff`:

```
diff :: Review -> Review -> [Difference]
```

In this article we're going to use a supporting library called Functional Java (FJ). This library defines several very useful functional constructs, and in particular `F` types to model pure[4] functions, that can be used to implement the examples we've just seen:

```
add1 :: F<Integer, Integer>
add :: F2<Integer, Integer, Integer>
```

### 4.1.1 Predicates

Predicates are a family of functions that have a specific signature:

```
a -> Boolean
```

This function is *polymorphic* since it's defined on an input value of type `a`, which is a *type parameter*. In other words, we can use that function with any type we want: `String`s, `Int`s, etc.

So, a predicate is a function that given a value of a certain type, returns true or false. In FJ a predicate is represented by the type `F<T, Boolean>`.

[4] As pure as possible, with Java you can't have strong guarantees about purity.

## 4.2 Higher-order functions

So called *higher-order functions* (HOF) are ordinary functions that can take other functions as parameters and/or return a function as a result. This concept gives us a great expressive power because this way **behaviour can be treated as data**: passed around, constructed and modified (not only *applied* to some values).

> Code is data, data code,—that is all Ye know on earth, and all ye need to know.

HOFs that requires other functions as parameters, permit to express *function application patterns* which enables an higher abstraction level. This is hardly a new concept: it's a generalization of the Command Pattern (just easier and more powerful) and it has been used in AWT since Java 1.

This is so useful and foundational that Java 8 introduced one of the most far-reaching changes since its inception: lambda expressions. Conceptually, a function can be seen as a class with a single method:

```
public interface F<A, B> {
    B f(A a);
}
```

In fact, this is basically how Functional Java models them. With this library[5] we can adopt a functional style even without first-class support of functions (lambda expressions), but be warned that without them things tends to become quite verbose and not very readable if overused. The TL;DR is: *just use Java 8 if you can!*

### 4.2.1 filter

As a first example of HOF we're going to look at `filter`. Its signature is:

```
filter :: (a -> Bool) -> [a] -> [a]
```

In short it is a function that applies a predicate `(a -> Bool)` *to every element* of the sequence `[a]` (a list of values of generic type `a`), discarding all values that does NOT satisfy it.

For example:

```
filter even [1, 2, 3, 4, 5, 6]
-- [2, 4, 6]
```

[5] There are others out there. For example: JavaSlang, TotallyLazy and fun4j. You could also roll your own if you want to.

Here we have a *function application* that can be read like this: "apply to `filter` the arguments `even` and `[1, 2, 3, 4, 5, 6]`".

- `filter` is our HOF.
- `even` is a predicate that requires an `Int` and returns true if it's... even.
- `[1, 2, 3, 4, 5, 6]` is a literal representation of a *list*, in this case of `Int` numbers.

The last line is the result, expressed as a comment.

### 4.2.2 map

`map` is another HOF with this signature:

```
map :: (a -> b) -> [a] -> [b]
```

In practice it applies a function `(a -> b)` *to every element* of the list `[a]`, producing a list of `b` values. Notice that `a` and `b` are *type parameters*[6] and can be the same actual type.

For example:

```
map add1 [1, 2, 3, 4, 5, 6]
-- [2, 3, 4, 5, 6, 7]
```

## 4.3 Maybe/Option

`Maybe` (which Functional Java calls `Option<T>` and Java 8 `Optional<T>`), is a generic type that denotes that a value of a certain type can be absent. Another way to see it is that `Option<T>` is a *collection* of values of type `T`, with an additional constraint on its *size* `s` such as: $0 <= s <= 1$. In other words, an `Option<T>` is a "box" that can contain (or not) a value of type `T`.

Using `Option` instead of `null`[7] makes explicit *statically and at the type system level* the fact that a value is optional: I can have it, or not. This way:

- It's explicit: no more sifting through the API documentation to check if it's the case.
- No more forgotten null checks and resulting `NullPointerException`s *at runtime*. The compiler can check them for you and doesn't compile until you manage both cases.

[6] Here, `a` is a "variable" that means "some type that we call `a`, we don't care what it is but only that in this signature when I say `a` I mean this type".

[7] Which Tony Hoare considers his "billion dollar mistake", rightfully so in my opinion.

## 4.4 Unit

`Unit` can be thought as a synonym of `void` and means "nothing". Pure functions in which signature appear `Unit` are generally quite useless:

```
foo :: Int -> Unit
bar :: Unit -> Int
```

Remember that the evaluation of a pure function computes a value that can be the result of some transformation or calculation based *only* on the values it's applied to. There is no other option: no DBs, no remote services, no IPC, no files, nothing. Knowing this, we can say important things about those functions even without looking at their implementations[8]:

- `foo` does nothing. It doesn't compute any value, and being pure it cannot do anything else.
- `bar` will always return the same, constant value. Again, being pure it can't take this value from anywhere and it can't "come up" with one in any other way (based on time or randomness for example). So it must return a constant value.

## 4.5 In practice

To make our method `#diff` more "pure" means, first of all, that *it shouldn't modify one of its arguments*. Instead, it should only compute a value and directly return it. Adding it to a list (or printing it to a log, or writing it into a database table, etc. etc.) has to be done in a different place, at a different abstraction level. Doing it in this way we gain a stronger separation of concerns and additional flexibility: we can apply it to a wider spectrum of use-cases.

More specifically, it means moving from:

```
diff :: String -> a -> a -> [Difference] -> Unit
```

To:

```
diff :: String -> a -> a -> Maybe Difference
```

That is:

```
private static <T> Option<Difference> compareReviewField(String description,
                                                         T oldValue,
                                                         T newValue) {
  if (!Objects.equals(oldValue, newValue)) return Option.some(
      difference(description, Utils.formatValue(oldValue),
                 Utils.formatValue(newValue)));
  else return Option.none();
}
```

[8] This ability to deduce the *behaviour* or the possible behaviours of a function looking only at it's signature is distinctive of statically-typed *pure* functions and it's called Parametricity.

After doing this, we must also change the structure of our main method by building a *data transformation pipeline* like this:

1. We create a list[9] of `Option<Difference>`.
2. Then filter the `Optional` values that actually contains a `Difference` value.
3. Finally, we `map` to every `Option` the function to extract the value it contains.

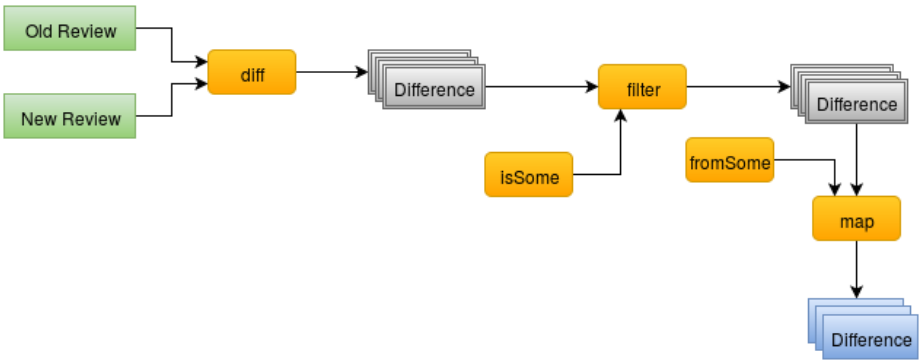


Figure 1: 2rd refactoring, first functional pipeline

In code:

```
public static List<Difference> diff(Review x, Review y) {
  if (x == null || y == null) throw new IllegalArgumentException(
      "Reviews shouldn't be null!");

  fj.data.List<Option<Difference>> changes = list(
      compareReviewField("Title", x.getTitle(), y.getTitle()),
      compareReviewField("Username", x.getUsername(), y.getUsername()),
      compareReviewField("Updated on", x.getUpdated(), y.getUpdated()),
      compareReviewField("Rating", x.getRating(), y.getRating()),
      compareReviewField("Text", x.getText(), y.getText()));

  return changes
         .filter(Option.isSome_())
         .map(Option.fromSome())
         .toJavaList();
}
```

## 4.6 Summary

At this point we have changed `#diff` by making it more functional, and as a consequence more flexible and reusable. We have also built a first data-transformation pipeline which in the rest of this document we're going to simplify and make more clear and flexible.

[9]We're talking about FJ's List type.

# 5 Third refactoring: SRP!

Let's take a step back and try to understand what we are really trying to accomplish with all this code: we have two reviews and we want to get back a human-readable representation of all the *data fields* that are different between those two. The typical use-case is obtaining a description of the differences between two different versions of the same review at different points in time.

In practice, we want something like this:

```
diff :: Review -> Review -> [Difference]
```

To write this function we use another one, `compareReviewField`:

```
compareReviewField :: String -> a -> a -> Maybe Difference
```

Which has multiple responsibilities because it has to do very different things:

1. Check if the values of a specific field are different.
2. *If* they are, create a value of type `Difference`.

To follow the Single Responsibility Principle (SRP) and write more simple, flexible and maintainable code, let's separate this two responsibilities in two separated functions and refactor the data transformation pipeline to work with them. Conceptually we want:

1. *Filtering*: we want to filter the value pairs right away, before even talking about the `Difference` type.
2. *Creating `Difference` values*: now we have only pairs with different values, and we can just create `Difference` values from them without further elaborations. It's just a data transformation.

To talk more precisely about those "value pairs" `(String, a, a)` that we want to filter and than transform in `Difference` values, let's create a new type!

```
data Field a = Field String a a
```

> This *data declaration*, as you may have been guessed, is *parmetric*: this means that we can have `Field` values that can contain values of any type (but both must be the same). For example: `Field Int`, `Field String`, etc.

Therefore, we have to define this new type in Java. Since immutable *values* are better:

```java
public class Field<T> {
  private final String description;
  private final T      oldValue, newValue;

  private Field(String description, T oldValue, T newValue) {
    this.description = description;
    this.oldValue = oldValue;
    this.newValue = newValue;
  }

  public static <T> Field<T> field(String name, T oldValue, T newValue) {
    Checks.notNull(name, oldValue, newValue);
    return new Field<>(name, oldValue, newValue);
  }

  public String getDescription() {
    return description;
  }

  public T getOldValue() {
    return oldValue;
  }

  public T getNewValue() {
    return newValue;
  }

  // Omissis: #equals(), #hashCode() & #toString()
}
```

Now we can modify `#compareReviewField` by making this changes:

- Simplify: we can remove the equality check between the two field values, extracting it in a different function.
- Change its name in `#toDifference` to better communicate its new (and single) intent.
- Make it return a *function* that we can easily apply in our data transformation pipeline.

```haskell
isFieldChanged :: Field a -> Boolean
toDifference :: Field a -> Difference
```

In Java:

```java
private static F<Field<?>, Difference> toDifference() {
  return f -> difference(f.getDescription(),
                         Utils.formatValue(f.getOldValue()),
                         Utils.formatValue(f.getNewValue()));
}
```

We could express this method signature like this:

```
toDifference :: Unit -> (Field t -> Difference)
```

The equality check can be moved in a predicate, using the same strategy:

```
private static F<Field<?>, Boolean> isFieldChanged() {
  return f -> !Objects.equals(f.getOldValue(), f.getNewValue());
}
```

Finally, we can modify the main method:

```
public static List<Difference> diff(Review x, Review y) {
  if (x == null || y == null) throw new IllegalArgumentException(
      "Reviews shouldn't be null!");

  fj.data.List<Field<?>> changes = list(
      field("Title", x.getTitle(), y.getTitle()),
      field("Username", x.getUsername(), y.getUsername()),
      field("Updated on", x.getUpdated(), y.getUpdated()),
      field("Rating", x.getRating(), y.getRating()),
      field("Text", x.getText(), y.getText()));

  return changes
         .filter(isFieldChanged())
         .map(toDifference())
         .toJavaList();
}
```

### 5.1 Summary

With this refactoring we have:

- Simplified the pipeline.
- Separated different responsibilities in several fine-grained and lower-level functions, which are easier to read, test and compose.
- Created a new concept, `Field`, to represent the value of the same field in two different reviews.

## 6 Fourth Refactoring: Type classes!

Let's take another look at `#toDifference`:

```
private static F<Field<?>, Difference> toDifference() {
  return f -> difference(f.getDescription(),
                         Utils.formatValue(f.getOldValue()),
                         Utils.formatValue(f.getNewValue()));
}
```

Its signature is:

```
toDifference :: Unit -> (Field a -> Difference)
```

Do you see something wrong in there? Let's see input and output one next the other to make it clearer:

```
data Field a = Field String a a
data Difference = Difference String String String
```

Basically, to transform a `Field a` value into a `Difference` one, the function returned from `#toDifference` must be able to produce a `String` representation of the values contained in that `Field a` *without knowing what type* `a` *is*. In other words, it must be able to *render* into the form we need (a human-readable `String`) arbitrary values.

However, how such a "rendering" is done right now? In our current implementation we use a *static utility method* [sic], that using the `instanceof` operator tries to test for some known types. So, this utility method *must know all possible types that will be placed into* `Field` *values*:

```
public class Utils {
  public static String formatValue(Object value) {
    if (value == null) return "none";
    else if (value instanceof String) return (String) value;
    else if (value instanceof Username) return ((Username) value).getName();
    else if (value instanceof LocalDateTime)
      return ((LocalDateTime) value).format(DateTimeFormatter.ISO_DATE_TIME);
    else return value.toString();
  }
}
```

This is... not the best way to do it. The rendering, done this way, is:

- *Implicit*
- Done *dynamically* through reflection
- Performed in a separated place that must be maintained in sync with the code that creates `Field` values upstream.

Since we can place anything into a `Field a` value, if we create a `Field` with a type not explicitly checked by an `if` [sic] in that static method (implicitly tied to the diffing process), in the best case we'll get unsuitable renderings (if `#toString()` has been overridden):

```
SomeObject{prop1=...,prop2=..., prop3=..., ...)
```

And in the worst case absolutely useless ones:

```
com.example.SomeObject@1b7ebdf8
```

In the end, what we want to use the values contained into a `Field` for?

- If they are `null`, we want the string "none".
- If they are NOT `null`, we want a human-readable `String` representation of them.

Instead of *ignoring* the actual type of the values contained in a `Field a`, and invoking an "utility method" that uses `instanceof` to try to understand how to produce a `String` from them, can we do better? Yes! We can define a *subset of types* that offer a way to produce a `String` representation semantically correct and useful for our purposes:

```
public interface Renderable {
  String render();
}
```

We also need a "null" implementation, so let's apply the Null Object pattern:

```
public class RenderableNone implements Renderable {
  public static RenderableNone renderableNone() {
    return new RenderableNone();
  }

  @Override
  public String render() {
    return "none";
  }
}
```

Now we can modify `#toDifference` to make it work only on `Field<? extends Renderable>` values, so that we can take advantage from the `#render()` method that they expose to produce the values we need:

```
private static F<Field<? extends Renderable>, Difference> toDifference() {
  return (Field<? extends Renderable> f) -> {
    Renderable oldValue = Option.fromNull(f.getOldValue())
                                .orSome(renderableNone());
    Renderable newValue = Option.fromNull(f.getNewValue())
                                .orSome(renderableNone());
    return difference(f.getDescription(), oldValue.render(),
                      newValue.render());
  };
}
```

Finally, we need to change `#diff` to make this constraint explicit:

```
public static List<Difference> diff(Review x, Review y) {
  if (x == null || y == null) throw new IllegalArgumentException(
      "Reviews shouldn't be null!");

  fj.data.List<Field<? extends Renderable>> changes = list(
      field("Title", x.getTitle(), y.getTitle()),
      field("Username", x.getUsername(), y.getUsername()),
      field("Updated on", x.getUpdated(), y.getUpdated()),
      field("Rating", x.getRating(), y.getRating()),
      field("Text", x.getText(), y.getText()));

  return changes
         .filter(isFieldChanged())
         .map(toDifference())
         .toJavaList();
}
```

But now the code doesn't compile since we try to create `Field` values with standard Java types like `String` ("Title" for example) and `LocalDateTime` ("Updated on" in this case) that don't implement our new `Renderable` interface (and in Java we can't make them implement it).

The thing we are trying to accomplish in programming languages with a more advanced type system can be done with type classes. They have many advantages, among the others: we can make an arbitrary type (even the ones that are out of our control) a member of a type class by writing *type class instances* (not to be confused with class instances). For example we could make `LocalDateTime` a member of `Renderable`:

```
class Renderable a where
  render :: a -> String

instance Renderable LocalDateTime where
  render d = format ISO_DATE_TIME d
```

In a simplistic way, you could think about type classes as interfaces and type classes instances as interface implementations for arbitrary types.

However, in Java this is not possible so we are forced to create some adapters:

```
public class RenderableString implements Renderable {
  private final String value;

  private RenderableString(String value) {this.value = value;}

  public static RenderableString renderableString(String value) {
    return value != null ? new RenderableString(value) : null;
  }

  @Override
```

```
  public String render() {
    return value;
  }

  // Omissis: #equals() & #hashCode() & #toString()
}

public class RenderableTime implements Renderable {
  private final LocalDateTime value;

  private RenderableTime(LocalDateTime value) {this.value = value;}

  public static RenderableTime renderableTime(LocalDateTime value) {
    return value != null ? new RenderableTime(value) : null;
  }

  @Override
  public String render() {
    return value.format(ISO_DATE_TIME);
  }

  // Omissis: #equals() & #hashCode() & #toString()
}
```

> We need to create adapters only for types out of our control. Since we own `Username` and `Rating`, we can make them directly implement `Renderable`.

In this way, we can fix `#diff` and `#isFieldChanged` to obtain:

```
public static List<Difference> diff(Review x, Review y) {
  if (x == null || y == null) throw new IllegalArgumentException(
      "Reviews shouldn't be null!");

  fj.data.List<Field<? extends Renderable>> changes = list(
      field("Title", renderableString(x.getTitle()),
            renderableString(y.getTitle())),
      field("Username", x.getUsername(), y.getUsername()),
      field("Updated on", renderableTime(x.getUpdated()),
            renderableTime(y.getUpdated())),
      field("Rating", x.getRating(), y.getRating()),
      field("Text", renderableString(x.getText()),
            renderableString(y.getText())));

  return changes
         .filter(isFieldChanged())
         .map(toDifference())
         .toJavaList();
}
```

### 6.1 Summary

With this refactoring we have further generalized the mechanism, moving the *responsibility* of rendering a human-readable representation of field values near the definition of the values themselves. And all of this by also **making explicit this constraint statically**, which now is verifiable by the compiler (you don't need to *remember* to modify two unrelated places anymore).

## 7 Conclusions

We started with code:

- Extremely procedural
- Full of repetitions
- Difficult to read and maintain
- With multiple responsibilities mixed together
- With implicit dependencies

And we refactored it to code which is (IMHO):

- Concise (or as concise as possible)
- Simple
- With a clear intent
- Flexible
- *Composable*
- With *explicit and statically verifiable* dependencies and constraints

All this through a series of refactorings that took us toward a *functional* design, by building a data transformation pipeline by *composing pure functions* that works on immutable values. Every function is as simple as possible (following the SRP), which makes comprehension, testing and reuse easier.

This is just an example, I don't claim it is the best possible design[10]. But if you have come this far, I hope it has been a useful and interesting journey and that I've been able to give you taste of what FP is all about. If you are interested and want to know more, here are some resources:

- My blog
- Book: Functional Programming in Java by Pierre-Yves Saumont
- Functional Thinking by Neal Ford
- Book: Java 8 Lambdas by Richard Warburton
- Book: Haskell Programming by Christopher Allen and Julie Moronuki
- How to learn Haskell: free resources to learn Haskell

[10] We could have used `Show` for example

# 8 Changelog

## 8.1 Version 1 (27 December 2015)

First version.

# 9 License