Dataflow async reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Dataflow async

This extension builds directly on top of Dataflow and provides all the necessary structure to perform async operations with fields.

The main features of this library are:

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

Getting started

Setup

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

Gradle

```
implementation 'com.femastudios:dataflow-async:1.0.1'
```

Maven

```
<dependency>
    <groupId>com.femastudios</groupId>
    <artifactId>dataflow</artifactId>
    <version>1.0.1</version>
</dependency>
```

Remember also to properly configure your environment as explained as explained here.

First look

In this library we associate a state with the data. So a data can be *loaded* or *not loaded*, where the latter further decomposes in *error* and *loading*. The `Attribute` class is the equivalent of fields for this new paradigm.

Creating attributes

The functions to create attribute are similar to the ones that exist for creating fields.

Kotlin

```
attributeOf("hello") //Yields an Attribute<String> with the LOADED value of "hello"
attributeOfNull() //Yields an Attribute<Nothing?> with the LOADED value of null

//Yields an attribute with the LOADING status that, when queried (lazy=true), will begin
computation asynchronously
attributeOf(lazy = true) { //receiver is a WorkContext instance (more on this later)
    "super" + "expensive" + "computation"
}
```

Transforming attributes

Transformations always computed asynchronously if not differently specified. The available extension functions are the same you find available for fields.

Example:

Kotlin

```kotlin
val attr = attributeOf(123);
val pow2 = attr.transform {
    it * it
}
```

Java

```java
Attribute<Integer> attr = Attribute.of(213);
Attribute<Integer> pow2 = AttributeUtils.transform(attr, num -> num * num);
```

Attributes

Introduction

As already mentioned an `Attribute` is like a field, but the data has an associated state. Another difference is that all transformations are made asynchronously if not otherwise specified.

For this reasons attributes are the obvious choice when we need to handle some data that could be not immediately available (or not available at all) and in general if we have data that may take a while to calculate. An example could be having on an attribute some remote data that needs to be downloaded.

Last but not least, it doesn't exist a mutable attribute, like in fields, so we cannot directly set a new value inside them.

Under attributes' hood

In order to represent this pair of data/state we use the class `AttributeData<T>`. Internally usually an attribute simply contains a `Field<AttributeData<T>>`.

For this reason we can call in all attributes the method `asField()` that will return a `Field<AttributeData<T>>`, like so:

Kotlin

```
val field /* : Field<AttributeData<Int>> */ = attributeOf(456).asField()
    println(field.value)
```

Java

```
Field<AttributeData<Integer>> field = Attribute.of(456).asField();
```

For more information see AttributeData.

Creating attributes

To create a field we must call the `attributeOf()` function, which has multiple versions that accept different inputs:

- **Value**: the most simple one simply accepts a value, in which case the returned attribute will have the *loaded* state.
- **AttributeData**: this version accepts an `AttributeData` instance that reflects the internal state we want to give it.
- **Task**: this version accepts a lambda that calculates the value. For more info see Tasks. This version will also return a subclass of attribute: `RecomputableAttribute`. This class adds the possibility to call the method `recompute()` to retrigger the task. It also supports two optional parameters:
    - *lazy*: a boolean value that tells if the attribute should start calculating right away (`false`) or if

it should start when it is requested for the first time (`true`).
- *flowStrategy*: a strategy that defines the behavior of the loading task. See the flow strategy section for more info.

Here's an example `lazy = false`:

Kotlin

```
//When using lazy=false the attribute starts computing the value right away on a background thread
    val a = attributeOf(lazy = false) {
        println("Computing...")
        10
    }
    Thread.sleep(100)
    println(a.value)
```

Java

```
//When using lazy=false the attribute starts computing the value right away on a background thread
Attribute<Integer> a = Attribute.of(false, workContext -> {
    System.out.println("Computing...");
    return 10;
});
Thread.sleep(10);
System.out.println(a.getValue());
```

And here with `lazy = true`:

Kotlin

```
//When using lazy=true the attribute is in LOADING until is queried for the first time
    val a = attributeOf(lazy = true) {
        println("Computing...")
        10
    }
    Thread.sleep(1000); //Wait a bit
    println("Waited 1s")
    println("attribute is: " + a.value) //Attribute is still LOADING
    //Calling `a.value` triggered the computation. Attaching a listener with `listen()` does the same.
    //Computation starts and the attribute goes to LOADED when computation is finished
    Thread.sleep(10); //Wait for computation
    println("attribute is: " + a.value) //Attribute is now LOADED
```

Java

```
//When using lazy=true the attribute is in LOADING until is queried for the first time
Attribute<Integer> a = Attribute.of(true, workContext -> {
    System.out.println("Computing...");
    return 10;
});
Thread.sleep(1000); //Wait a bit
System.out.println("attribute is: " + a.getValue()); //Attribute is still LOADING
//Calling `a.getValue()` triggered the computation. Attaching a listener with `listen()` does the same.
//Computation starts and the attribute goes to LOADED when computation is finished
Thread.sleep(10); //Wait for computation
System.out.println("attribute is: " + a.getValue()); //Attribute is now LOADED
```

From a field

If we have a simple field that we want to transform into an `Attribute` we can call the `async()` function, like so:

Kotlin

```kotlin
val field = mutableFieldOf(10)
    println(field.value)

    //attr will have the LOADED status with the same value as the field
    val attr = field.async()
    println(attr.value)

    //When the field changes, the attribute also changes
    field.value = 44
    println(attr.value)
```

Java

```java
Field<Integer> field = Field.of(10);
System.out.println(field.getValue());

//attr will have the LOADED status with the same value as the field
Attribute<Integer> attr = AttributeUtils.async(field);
System.out.println(attr.getValue());

//When the field changes, the attribute also changes
field.setValue(44);
System.out.println(attr.getValue());
```

Whenever the backing `Field` changes the attribute will also change.

AttributeData

`AttributeData<T>` is an abstract sealed class that represents the value of an attribute.

The class hierarchy defines the status, and has this scheme:

- `Loaded`: represents and contains a value whose status is *loaded*.
- `NotLoaded`: represents a value that is not yet loaded. It is a sealed class as well, that divides in:
    - `Loading`: represents a value that is currently loading.
    - `Error`: represents a value that couldn't be loaded due to an error. It contains a message and a description of the error.

Creating `AttributeDatas`

To create `AttributeDatas` we can either use their constructor or call this convenience methods:

Kotlin

```
AttributeData.errorInstance("Task failed successfuly", "A descriptive description")
//Creates an ERROR AttributeData
AttributeData.loadedInstance(1234) //Creates a LOADED AttributeData with the value 1234
AttributeData.loadingInstance() //Creates a LOADING AttributeData
```

Java

```
AttributeData.errorInstance("Task failed successfuly", "A descriptive description");
//Creates an ERROR AttributeData
AttributeData.loadedInstance(1234); //Creates a LOADED AttributeData with the value 1234
AttributeData.loadingInstance(); //Creates a LOADING AttributeData
```

Extending statuses

The nice things about this class is that is also extendable: for instance, imagine we want to create an attribute that downloads something from some APIs, but we want to also retain the information of what kind of error happened. We can easily extend the `Error` class to accommodate our needs:

Kotlin

```
enum class ApiErrorType {
    API_LIMIT_REACHED, SERVER_BUSY, BAD_REQUEST, SSL_ERROR, OTHER
}

class ApiError(val type: ApiErrorType) : Error("Error calling APIs: $type")
```

Java

```
enum ApiErrorType {
    API_LIMIT_REACHED, SERVER_BUSY, BAD_REQUEST, SSL_ERROR, OTHER
}

class ApiError extends Error<Nothing> {
    public final ApiErrorType type;

    public ApiError(ApiErrorType type) {
        super("Error calling APIs: " + type);
        this.type = type;
    }
}
```

Now we can return this an instance of this error from our task and check for it when needed.

Tasks

A task is defined to be something that calculates the value of an attribute. Examples of tasks include the lambda passed when creating an attribute and the transformation function between two attributes.

What distinguishes tasks from a simple function is the fact that they must ultimately produce an `AttributeData` instance. The problem is that it would be highly inconvenient for you, the end user of this library, to always create an instance to return since most of the time you'll need to return a `Loaded` instance in transformation functions. For this reason the return value of tasks is `T`, **not** `AttributeData<T>`: it is implied that when the task returns a value, the status is *loaded*.

How to return something different then? For this purpose all tasks accept as their receiver a `WorkContext` instance, from which we can create and throw a particular exception that conveys this meaning. For more information about receiver of functions see the Kotlin™ documentation.

This examples of transformations of an attribute will clarify:

Kotlin

```
val a = attributeOf(lazy = false) { //The receiver of this lambda is a WorkContext
instance, so `this` is now a WorkContext
        println("this = " + (this is WorkContext)) //Verify that `this` is a `WorkContext`
instance
        val ret = Random().nextInt(100)
        if(ret < 50) {
            throwError("Nasty error!") //<-- called on the WorkContext receiver
        } else {
            ret
        }
    }
    Thread.sleep(100) //Allow time for the attribute to compute
    println(a.value) //Will be either LOADED or ERROR, depending on random result
```

Java

```
Attribute<Integer> a = Attribute.of(workContext -> {
    //Since Java doesn't support the change of `this` in lambdas the WorkContext is passed
as its first parameter
    final int ret = new Random().nextInt(100);
    if(ret < 50) {
        //Since Java cannot understand that a method can always throw an exception
        //we must use a different function that creates the exception and throw it
ourselves
        throw workContext.errorException("Nasty error!");
    } else {
        return ret;
    }
});
Thread.sleep(1000); //Allow time for the attribute to compute
System.out.println(a.getValue()); //Will be either LOADED or ERROR, depending on random
result
```

In this example we declared a new attribute with a task: this task generates a random number and returns it if it is less than `50`, otherwise throws an exception generated with the passed `WorkContext`. There are also more functions that allow you to pass an instance of `NotLoaded`.

Note that you must explicitly throw errors, and that any uncaught exception in tasks will propagate its way up to caller, similarly to what happens on fields.

FlowStrategy

Each attribute we create (either directly or with a transformation functions) has its own flow strategy that defines how the inner `AttributeData` is calculated and is expressed through the `FlowStrategy` sealed class.

Currently there are three possible choices:

- **Consistent**: the task will be executed in a background thread and while doing so the internal status will be set to *loading*.
- **Eventually consistent**: the task will be executed in a background thread but the old value will be kept until the new one is produced. The status will be set to *loading* only when the first time the value is being calculated or the previous state was *error*.
- **Synchronous**: the task will be executed in the thread the triggered the calculation. The old value will be kept until the new one is set. The status will never be set to *loading*. The behavior is similar to a standard lazy field, with the difference that the status could be error.

If not explicitly specified, the default flow strategy for the `attributeOf()` function is `FlowStrategy.CONSISTENT`, while for transformations it's `FlowStrategy.EVENUTALLY_CONSISTENT`, in order to avoid spamming *loading* statues though the transformation chain every time something changes.

When a task needs to be executed in a background thread a `ThreadPoolExecutor` is internally used.

Example with `FlowStrategy.CONSISTENT`

Kotlin

```
val attr = attributeOf(lazy=false, flowStrategy=FlowStrategy.CONSISTENT) {
    expensiveComputation("1 + 1")
}
//Since not lazy it immediately starts the computation on a background thread, status is
LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED

attr.recompute() //Trigger recomputation
//Status goes immediately to LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED
```

Java

```
Attribute<Integer> = Attribute.of(false, FlowStrategy.CONSISTENT, workContext -> {
    return expensiveComputation("1 + 1");
});
//Since not lazy it immediately starts the computation on a background thread, status is
LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED

attr.recompute(); //Trigger recomputation
//Status goes immediately to LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED
```

Example with `FlowStrategy.EVENTUALLY_CONSISTENT`

Kotlin

```kotlin
val attr = attributeOf(lazy=false, flowStrategy=FlowStrategy.EVENTUALLY_CONSISTENT) {
    expensiveComputation("1 + 1")
}
//Since not lazy it immediately starts the computation on a background thread, status is
LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED

attr.recompute() //Trigger recomputation
//Status remains to LOADED with the old value
//□ Some time passes...
//Computation ends, status becomes LOADED with the new value
```

Java

```java
Attribute<Integer> = Attribute.of(false, FlowStrategy.EVENTUALLY_CONSISTENT, workContext ->
{
    return expensiveComputation("1 + 1");
});
//Since not lazy it immediately starts the computation on a background thread, status is
LOADING
//□ Some time passes...
//Computation ends, status becomes LOADED

attr.recompute(); //Trigger recomputation
//Status remains to LOADED with the old value
//□ Some time passes...
//Computation ends, status becomes LOADED with the new value
```

Example with `FlowStrategy.SYNCHRONOUS`

Kotlin

```kotlin
//Since not lazy it immediately starts the computation on the current thread
val attr = attributeOf(lazy=false, flowStrategy=FlowStrategy.SYNCHRONOUS) {
    expensiveComputation("1 + 1")
}
//When attr is available status will be LOADED with the calculated value

attr.recompute() //Trigger recomputation on the current thread
//When recompute() returns the status will be LOADED with the newly calculated value
```

Java

```java
//Since not lazy it immediately starts the computation on the current thread
Attribute<Integer> = Attribute.of(false, FlowStrategy.SYNCHRONOUS, workContext -> {
    return expensiveComputation("1 + 1");
});
//When attr is available status will be LOADED with the calculated value

attr.recompute(); //Trigger recomputation on the current thread
//When recompute() returns the status will be LOADED with the newly calculated value
```

Listening attributes

Recall that an attribute is internally a `Field<AttributeData<T>>`. There isn't a direct way to listen to

attributes, but you use the same infrastructure that is already present for fields. For more information see the dataflow reference for listening fields.

How to listen

Using `asField()`

Attributes expose the underlying fields through the `asField()` function, that returns a `Field<AttributeData<T>>`. You can then listen for changes in this field.

Using `listen()`

There is also a convenience extension function that directly accepts an attribute and that listens for the underlying field.

Example:

Kotlin

```
val attr = attributeOf { 10 }
    lifecycle {
        listen(attr) {
            when(it) {
                is Loaded -> println("LOADED! Value is: ${it.value}")
                is Loading -> println("Loading...")
                is Error -> println("Error loading data! ${it.displayableMessage}")
            }
        }
        Thread.sleep(100) //Wait to allow first computation
        println("recompute()")
        attr.recompute()
        Thread.sleep(100) //Wait to allow second computation
    }
```

Java

```
Attribute<Integer> attr = Attribute.of(wc => 10);
LifecycleOwner.lifecycle(lc -> {
    //In java it's just easier to call `asField()` then it is to call the extension
function
    lc.listen(attr.asField(), v -> {
        if(v instanceof Loaded) {
            System.out.println("LOADED! Value is: " + v.getValue());
        } else if(v instanceof Loading) {
            System.out.println("Loading...");
        } else if(v instanceof Error) {
            System.out.println("Error loading data! " + v.getDisplayableMessage());
        } else {
            throw new IllegalStateException(); //can never reach here because AttributeData
and NotLoaded are sealed classes
        }
    });
    Thread.sleep(100); //Wait to allow first computation
    System.out.println("recompute()");
    attr.recompute();
    Thread.sleep(100); //Wait to allow second computation
});
```

As you can see using this method you can verify the status of the attribute, and you may as check for custom statuses.

Using `valueOrNull()`

If you don't need the distinction between the not loaded statuses, you can simply call the `valueOrNull()` function that will return a `Field<T?>` whose value will be `null` when the status is not loaded.

Example:

Kotlin

```
lifecycle {
    listen(attr.valueOrNull()) {
        if(it == null) {
            println("Attribute value not available")
        } else {
            println(it)
        }
    }
}
```

Java

```
LifecycleOwner.lifecycle(lc -> {
    lc.listen(attr.valueOrNull(), v -> {
        if(v == null) {
            System.out.println("Attribute value not available");
        } else {
            System.out.println(v);
        }
    });
});
```

Listening on another thread

If we need to do a lot of work on our listener we may consider doing it on another thread. Luckily it's very easy to do it in Dataflow! All you need to do is call the `async()` function on the wanted lifecycle owner and then proceed to listen for fields or attributes as usual.

Example:

Kotlin

```
//With FlowStrategy.SYNCHRONOUS we ensure that when calling `recompute()` the calculation
is done on the main thread
    val attr = attributeOf(flowStrategy = FlowStrategy.SYNCHRONOUS) {
        println("Task thread: " + Thread.currentThread().id)
        10
    }

    globalLifecycle.async().listen(attr) {
        //Some expensive computation
        println("Listener thread: " + Thread.currentThread().id) //Verify we are on a
different thread
    }

    println("Main thread: " + Thread.currentThread().id)
    attr.recompute()
    Thread.sleep(100) //Allow for recompute() to run before terminating program
```

Java

```
//With FlowStrategy.SYNCHRONOUS we ensure that when calling `recompute()` the calculation
is done on the main thread
Attribute<Integer> attr = Attribute.of(FlowStrategy.SYNCHRONOUS, wc => {
    System.out.println("Task thread: " + Thread.currentThread().id);
    return 10;
});

LifecycleOwnerUtils.async(globalLifecycle).listen(attr) {
    //Some expensive computation
    System.out.println("Listener thread: " + Thread.currentThread().id); //Verify we are on
a different thread
}

System.out.println("Main thread: " + Thread.currentThread().id);
attr.recompute();
Thread.sleep(100); //Allow for recompute() to run before terminating program
```

Other features

Debouncing and delayed

When using this library it's possible to obtain a field or attribute whose value is the same, but in some way changed temporally. The initial value of this field is always the same as the original.

Debouncing

The first one is created through the `debounced()` function: when the value of the original field/attribute changes it waits for a grace period before applying the change to itself. If the value changes during this grace period, the waits starts again. It is also possible to specify a maximum amount of time between updates, to avoid a situation where the original data constantly changes thus never allowing the debounced field to update.

Kotlin

```
var i = 0
    val attr = attributeOf {
        i++
    }
    lifecycle {
        listen(attr.debounced(delay = 100, maxDelay = 200)) {
            println("debounced value $it")
        }

        for(x in 0 until 100) {
            attr.recompute()
            Thread.sleep(10)
        }

        Thread.sleep(220) //Allow max delay to elapse
    }
```

Java

```
int i = 0
Attribute<Integer> attr = Attribute.of(wc -> i++);
LifecycleOwner.lifecycle(l -> {
    l.listen(attr.debounced(100, 200), (newValue) => {
        System.out.println("debounced value " + newValue);
    });

    for(int x = 0; x < 100; x++) {
        attr.recompute();
        Thread.sleep(10);
    }

    Thread.sleep(220); //Allow max delay to elapse
});
```

As you can see the debounced attribute changes value every once in a while, depending on the values

of `delay` and `maxDelay`. If you try and remove the `maxDelay` from the debounced you'll see that the listener is called only at the end, when the "spamming" of `recompute()` finishes. Try to remove completely the debounced attribute, listening only for `attr`, and you'll see how many calls this technique saves.

Note: you can also pass a lambda to calculate the delay. It's useful if you want different delays for different values (for instance, you can avoid debouncing errors). The lambda will pass as the first parameter the current **debounced** value and the new value from the original field.

Delayed

You can also obtained a `delayed()` attribute or field: this simply means that all changes will be delayed by a given amount of time, effectively having a past live snapshot of your data.

Kotlin

```
var i = 0
    val attr = attributeOf {
        Thread.sleep(10)
        i++
    }
    lifecycle {
        val start = System.currentTimeMillis()
        listen(attr) {
            println("attr=$it @ " + (System.currentTimeMillis() - start) + "ms")
        }
        listen(attr.delayed(1000)) {
            println("delayed=$it @ " + (System.currentTimeMillis() - start) + "ms")
        }

        println()
        attr.recompute()
        Thread.sleep(250)
        attr.recompute()
        Thread.sleep(100)
        attr.recompute()

        Thread.sleep(1100) //Allow delayed to finish
    }
```

Java

```
int i = 0
Attribute<Integer> attr = Attribute.of(wc -> {
    Thread.sleep(10);
    return i++;
});
LifecycleOwner.lifecycle(l -> {
    final long start = System.currentTimeMillis()
    l.listen(attr, (newValue) => {
        System.out.println("attr=" + it + " @ " + (System.currentTimeMillis() - start) +
"ms");
    });
    l.listen(attr.delayed(1000), (newValue) => {
        System.out.println("delayed=" + it + " @ " + (System.currentTimeMillis() - start) +
"ms");
    });

    System.out.println();
    attr.recompute();
    Thread.sleep(250);
    attr.recompute();
    Thread.sleep(100);
    attr.recompute();

    Thread.sleep(1100) //Allow delayed to finish
});
```

As you can see by running the example all values assumed by `attr` are assumed by its delayed counterpart one second later. Note that due to scheduling uncertainty some intermediate states may be lost if they are too temporally close; you can see this by removing the sleep in the attribute calculation. Don't worry: the most recent state is always chosen.

Note: as for debouncing, you can also pass a lambda to calculate the delay. The lambda will always pass as the first parameter the current **delayed** value and the new value from the original field.

Wrapping attributes

Wrapping attributes is done in the same exact way it's done for fields. You can construct a wrapper using the function `attributeWrapperOf()` as with fields, and you can set custom `AttributeData`s, as well as other attributes. There are also some convenience methods. To a more thorough explanation please refer to the field wrapper documentation.

Example:

Kotlin

```
val i = AtomicInteger()
    val attr = attributeOf { i.incrementAndGet() }
    val attrWrap = attributeWrapperOf<Int>() //If nothing given initially, status is
LOADING
    lifecycle {
        listen(attrWrap) {
            println(it)
        }

        attrWrap.setAttribute(attr)
        attr.recompute()
        Thread.sleep(10)
        attr.recompute()
        Thread.sleep(10)
        attrWrap.setError("Error!")
        attrWrap.setLoadedValue(42)
    }
```

Java

```
AtomicInteger i = new AtomicInteger();
Attribute<Integer> attr = Attribute.of(wc -> { i.incrementAndGet(); });
AttributeWrapper<Integer> attrWrap = AttributeWrapper.of() //If nothing given initially,
status is LOADING
LifecycleOwner.lifecycle(lc -> {
    lc.listen(attrWrap, it -> {
        System.out.println(it);
    });

    attrWrap.setAttribute(attr);
    attr.recompute();
    Thread.sleep(10);
    attr.recompute();
    Thread.sleep(10);
    attrWrap.setError("Error!");
    attrWrap.setLoadedValue(42);
});
```

Wait while loading

Sometimes it may be necessary to synchronously wait for an attribute to finish loading, albeit not frequently. We can easily do this by using the function `waitWhileLoading()`. The function has this behavior:

- Returns the first `Loaded` or `Error` instance that the attribute assumes since this call (current value included)
- Returns a `Loading` instance if the `maxWaitTime` parameter expires. If the `maxWaitTime` is `null` then this function can *never* return a `Loading` instance.
- Throws `InterruptedException` if the caller thread is interrupted while waiting.

Sample usage:

Kotlin

```kotlin
val attr = attributeOf {
        Thread.sleep(1000)
        "hello"
    }
    val start = System.currentTimeMillis()
    println("Starting waiting...")
    val data = attr.waitWhileLoading()
    println("Wait over after " + (System.currentTimeMillis() - start) + "ms")
    println("Data is $data")
```

Java

```java
val attr = Attribute.of(wc -> {
    Thread.sleep(1000)
    return "hello"
});
long start = System.currentTimeMillis();
System.out.println("Starting waiting...");
AttributeData<String> data = attr.waitWhileLoading();
System.out.println("Wait over after " + (System.currentTimeMillis() - start) + "ms");
System.out.println("Data is " + data);
```

You'll see that if you add a small `maxTime` parameter the returned data will be loading.

WeakWorstStatusWrapper

`WeakWorstStatusWrapper` is an interface that allows to monitor the status of multiple attributes. It's designed to deal with thousands of attributes, and for this reason it spawns two threads. The references to the handled attributes are kept weakly.

The interface extends `Field<AttributeData<Nothing?>>`, and will contain the worst status of all the attributes:

1. if it's empty (no attributes have been added, or all have been removed), the status will be *loading*;
2. if all attributes have status *loaded*, the status will be *loaded*;
3. if at least one attribute has status *error*, the status will be *error*;
4. if at least one attribute has status *loading*, the status will be *loading*.

If the `preferLoading` value is set to `true`, step 3 and 4 are reversed.

If more than one attributes have status *error*, the errors will be merged using the method `merge()`.

You can use the `WeakWorstStatusWrapper.create()` function to create an empty one, than add attributes to watch with the method `addAttribute` and remove them with `removeAttribute()`. When you don't need it anymore, you have to call the `close()` function, in order to interrupt the spawned threads.

Example:

Kotlin

```kotlin
val attr1 = attributeWrapperOf(1)
    val attr2 = attributeWrapperOf("2")
    val attr3 = attributeWrapperOf(3.0)

    WeakWorstStatusWrapper.create().use { wwsw ->
        wwsw.addAttribute(attr1)
        wwsw.addAttribute(attr2)
        wwsw.addAttribute(attr3)
        lifecycle {
                listen(wwsw) {
                    println("WWSW: $it")
                }
                println("\nChainging attr1 to loading...")
                attr1.setLoading() //Since there is at least one loading, WWSW will be
loading
                Thread.sleep(50)

                println("\nChainging attr2 to error...")
                attr2.setError("Error!") //Notice that since we have only one error, this
instance will be the same in the WWSW
                Thread.sleep(50)

                println("\nChainging attr3 to error...")
                attr3.setError("Ops!") //Now that we have introduced a second error with no
merge policy, the error in the WWSW will be a generic one (no message)
                Thread.sleep(50)

                println("\nChainging preferLoading = true...")
                wwsw.preferLoading = true //Changing pereferLoading=true, the WWSW will now
have Loading status since there is at least one loading
                Thread.sleep(50)
            }
    }
```

Java

```
AttributeWrapper<Integer> attr1 = AttributeWrapper.of(1);
AttributeWrapper<String> attr2 = AttributeWrapper.of("2");
AttributeWrapper<Double> attr3 = AttributeWrapper.of(3.0);

try(WeakWorstStatusWrapper wwsw = WeakWorstStatusWrapper.create()) {
    wwsw.addAttribute(attr1)
    wwsw.addAttribute(attr2)
    wwsw.addAttribute(attr3)
    LifecycleOwner.lifecycle(lc -> {
            lc.listen(wwsw, it -> {
                System.out.println("WWSW: " + it);
            });
            System.out.println("\nChainging attr1 to loading...");
            attr1.setLoading(); //Since there is at least one loading, WWSW will be loading
            Thread.sleep(50);

            System.out.println("\nChainging attr2 to error...");
            attr2.setError("Error!"); //Notice that since we have only one error, this
instance will be the same in the WWSW
            Thread.sleep(50);

            System.out.println("\nChainging attr3 to error...");
            attr3.setError("Ops!"); //Now that we have introduced a second error with no
merge policy, the error in the WWSW will be a generic one (no message)
            Thread.sleep(50);

            System.out.println("\nChainging preferLoading = true...");
            wwsw.setPreferLoading(true); //Changing pereferLoading=true, the WWSW will now
have Loading status since there is at least one loading
            Thread.sleep(50);
        });
}
```