

[Live Webinar] Gain a Competitive Edge with Graph Analytics

[Sign Up Now](#)

DZone > Java Zone > Introduction to Reactive Programming

# Introduction to Reactive Programming

by Tiago Albuquerque · Apr. 08, 20 · Java Zone · Tutorial

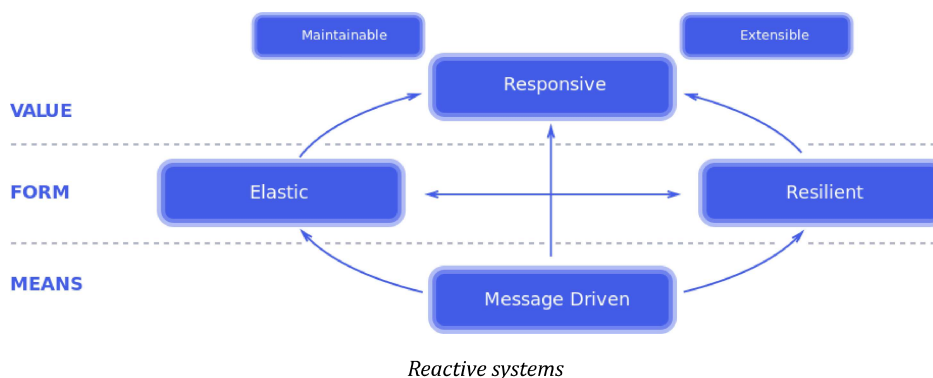
Apache Druid was invented to address the lack of a data store optimized for real-time analytics. [Download this whitepaper](#) for an introduction to Apache Druid, including its evolution, architecture and features, and common use cases.

Presented by Impley Data

Reactive programming is about dealing with data streams and the propagation of change.

Reactive systems are applications whose architectural approach make them responsive, resilient, elastic and message-driven.

- **Responsive:** Systems should respond in a timely manner.
- **Message Driven:** Systems should use asynchronous message communication between components to ensure loose coupling.
- **Elastic:** Systems should stay responsive under high load.
- **Resilient:** Systems should stay responsive when some components fail.



According to the “Reactive Manifesto”:

**“Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.”**

There are Reactive libraries available for many programming languages that enable this programming paradigm.

Such libraries from the “ReactiveX” family are:

**“..used for composing asynchronous and event-based programs by using observable sequences. It extends the observer pattern to support sequences of data or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.”**

So, it's possible to avoid the “*callback hell*” problem and abstract other issues concerning threads and low-level asynchronous computations. That makes our code more readable and focused in business logic.

## The Observable x Observer Model

Simply put, an observable is any object that emits (stream of) events, that the observer reacts to. The Observer Object subscribes to an Observable to listen whatever items the observable emits, so it gets notified when the observable state changes. The observer is also called subscriber or reactor, depending on the library used.



*Observable and observer*

The Observer stands ready to react appropriately when the Observable emits items in any point in time. This pattern facilitates concurrent operations because it doesn't need to block while waiting for the Observable to emit items.

The Observer contract expects the implementation of some subset of the following methods:

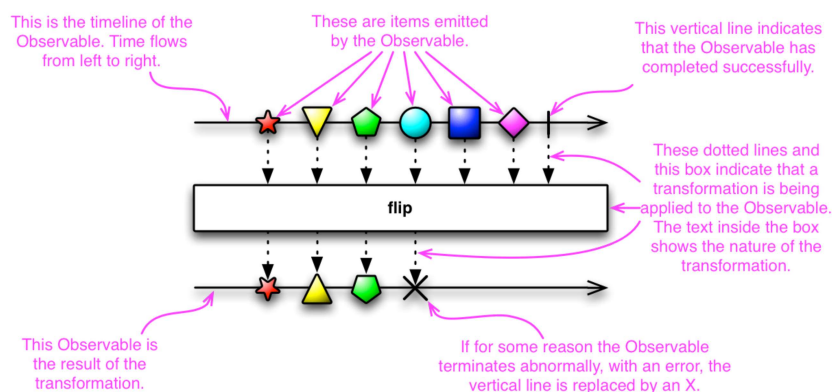
- **OnNext** : Whenever the Observable emits an event, this method is called on our Observer, which takes as parameter the object emitted so we can perform some action on it.
- **OnCompleted** : This method is called after the last call of the onNext method, indicating that the sequence of events associated with an Observable is complete and it has not encountered any errors.
- **OnError** : This method is called when it has encountered some error to generate the expected data, like an unhandled exception.

## Operators

Operator is a function that, for every element the source Observable emits, it applies that function to that item, and then emit the resulting element in another Observable.

So, operators operate on an Observable and return another Observable. This way, operators can be combined one after other in a chain to create data flows operations on the events.

The behavior of each operator is usually illustrated in marble diagrams like this (Rx Marbles):



*Behavior of components*

Reactive operators have many similarities to those of functional programming, bringing better (and faster) understanding of them.

Some of the most used core operators in ReactiveX libraries are:

- `map` : transforms items emitted by an Observable by applying a function to each them.
- `flatMap` : transforms the objects emitted by an Observable into Observables (“nested Observables”), then flatten the emissions from those into a single Observable.
- `filter` : emits only items from an Observable that pass a predicate test.
- `just` : converts objects into an Observable that emits those objects.
- `takeWhile` : discards items emitted by an Observable after a specified condition becomes false.
- `distinct` : suppresses duplicate objects emitted by an Observable.

There is also an important concept of ***backpressure***, which provides solutions when an `Observable` is emitting items more quickly than a `Observer` can consume them.

---

## “Show Me the Code!”

After some background theory, let's get to the fun part!

Below let's go through a hands-on approach, to provide an understanding by seeing the magic in motion!

The examples use the *RxJava (version 1.3.8)* library:

Java

```
1 <!-- maven dependency -->
2 <dependency>
3   <groupId>io.reactivex</groupId>
4   <artifactId>rxjava</artifactId>
5   <version>1.3.8</version>
6 </dependency>
```

Here it is a simple inline “Hello World” code using an observable and immediate subscription:

Java

```
1 // simplest way (in-line)
2 public static void helloWorld1() {
3   Observable.just("Hello Reactive World 1!").subscribe(System.out::println);
4 }
```

It's possible to do implicit or more explicit calls to observer functions/methods:

Java

```
1 // explicit onNext function call
2 public static void helloWorld2() {
3   Observable<String> myObservable = Observable.just("Hello World 2!"); // not emitted yet
4   Action1<String> onNextFunction = msg -> System.out.println(msg);
5   myObservable.subscribe(onNextFunction); // item emitted at subscription time (cold observable)!
6 }
7
8 // explicit onNext and onError functions call
9 public static void helloWorld3() {
10  Observable<String> myObservable = Observable.just("Hello World 3!"); // not emitted yet
11  Action1<String> onNextFunction = System.out::println;
12  Action1<Throwable> onErrorFunction = RuntimeException::new;
13  myObservable.subscribe(onNextFunction, onErrorFunction); // item emitted at subscription time (cold observable)!
14 }
```

Segregating Observable and Observer objects:

```
Java

1
2 public static void helloWorld4() {
3
4     Observable<String> myObservable = Observable.just("Hello World 4!");
5
6     Observer<String> myObserver = new Observer<String>() {
7         @Override
8         public void onCompleted() { System.out.println("onCompleted called!"); }
9         @Override
0         public void onError(Throwable e) { System.out.println("onError called!"); }
1         @Override
2         public void onNext(String msg) { System.out.println("onNext => Message received: " + msg); }
3     };
4
5     myObservable.subscribe(myObserver);
6 }
```

The code above prints:

```
Java

1 onNext => Message received: Hello World 4!
2 onCompleted called!
```

Since it is emitted just one item, it can be a single object:

```
Java

1 // since it is emitted just one item, it can be a Single object
2 public static void helloWorld5() {
3     Single<String> mySingle = Single.just("Hello World 5 from Single!");
4     mySingle.subscribe(System.out::println, RuntimeException::new);
5 }
```

Operators examples:

```
Java

1 public static void operatorsExamples() {
2     // filter = apply predicate, filtering numbers that are not even
3     Func1<Integer, Boolean> evenNumberFunc = x -> x%2 == 0;
4     // map = transform each elements emitted, double them in this case
5     Func1<Integer, Integer> doubleNumberFunc = x -> 2*x;
6
7     Observable<Integer> myObservable = Observable.range(1, 10) // emits int values from the range
8         .filter(evenNumberFunc)
9         .map(doubleNumberFunc);
0
1     myObservable.subscribe(System.out::println); // prints 4 8 12 16 20
2 }
```

The output of the code above is:

```
Plain Text

1 4
2 8
3 12
4 16
5 20
```

Interval operator:

```
Java
```

```

1 // Interval operator
2 private static void intervalExample() {
3     Observable.interval(2, TimeUnit.SECONDS) // emits a sequential number every 2 seconds
4         .take(5) // limit to first 5 elements
5         .toBlocking() // converts to a blocking observable
6         .subscribe(System.out::println); // prints 0 to 4 in 2 seconds interval
7 }

```

It's also possible to get an Observable from a List , a Callable or a Future instance:

Java

```

1 // Creating Observables from a Collection/List
2 private static void observableFromListExample() {
3     List<Integer> intList = IntStream.rangeClosed(1, 10).mapToObj(Integer::new).collect(Collectors.toList());
4     Observable.from(intList).subscribe(System.out::println); // prints from 1 to 10:
5 }
6
7 // Creating Observables from Callable function
8 private static void observableFromCallableExample() {
9     Callable<String> callable = () -> "From Callable";
10    // defers the callable execution until subscription time
11    Observable.fromCallable(callable).subscribe(System.out::println);
12 }
13
14 // Creating Observables from Future instances
15 private static void observableFromFutureExample() {
16     CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "From Future");
17     // converts a Future into Observable
18     Observable.from(future).subscribe(System.out::println);
19 }

```

Of course, we can set `<nerd mode on>` and implement a Star Wars battle using Reactive Programming (source code here):

Java

```

1 // StarWar battle: Let's get nerdy...
2 private static void starWarsBattle() {
3     Random random = new Random();
4
5     // Stormtrooper class
6     class Stormtrooper {
7         private int imperialNr;
8
9         public Stormtrooper(int imperialNumber) {
10             this.imperialNr = imperialNumber;
11         }
12         public String getName() {
13             return "#" + imperialNr;
14         }
15     }
16
17     // callable func that creates a Stormtrooper after 3 seconds delay
18     Callable<Stormtrooper> trooperGenerator = () -> {
19         Thread.sleep(3 * 1000);
20         return new Stormtrooper(random.nextInt());
21     };
22
23     // Creating Observables of Stormtrooper creation
24     List<Observable<Stormtrooper>> observables = IntStream.rangeClosed(1, 4)
25         .mapToObj(n -> Observable.fromCallable(trooperGenerator)).collect(Collectors.toList());
26
27     // Jedi observer to fight every trooper created in time
28     Observer<Stormtrooper> jedi = new Observer<Stormtrooper>() {
29         @Override
30         public void onCompleted() {
31             System.out.println("May the force be with you!");
32         }
33         @Override
34         public void onError(Throwable e) {
35             throw new RuntimeException(e);
36         }
37         @Override
38         public void onNext(Stormtrooper t) {
39             System.out.println("Jedi defeated Stormtrooper " + t.getName());
40         }
41     };
42 }

```

```
1 // Stormtrooper creation, every defeated Stormtrooper will trigger this event
2 }
3 };
4
5 // Jedi subscribe to listen to every Stormtrooper creation event
6 observables.forEach(o -> o.subscribe(jedi)); // Battle inits at subscription time
7 }
```

The output of the code above may be (troopers ID numbers are random):

Plain Text

```
1 Jedi defeated Stormtrooper #246
2 May the force be with you!
3 Jedi defeated Stormtrooper #642
4 May the force be with you!
5 Jedi defeated Stormtrooper #800
6 May the force be with you!
7 Jedi defeated Stormtrooper #205
8 May the force be with you!
```

## Resources and Further Readings

- The Reactive Manifesto
- ReactiveX
- RxJava

Why MariaDB? Row-based for transactions, columnar  
Plus built-in query routing, streaming CDC, JSON func  
standard SQL syntax. [View Deployment Guide](#)

Presented by MariaDB

## Like This Article? Read More From DZone



**Go Reactive With RxJava**



**Creating an Observable in RxJava**




**Handling RxJava Observables in a Web UI**



**Free DZone Refcard  
Getting Started With Headless CMS**

Topics: JAVA , REACTIVE PROGRAMMING , RXJAVA , OBSERVER , OBSERVABLE , REACT

Published at DZone with permission of Tiago Albuquerque . [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.