Programming Paradigms Tutorials
Reactive Programming

Reactive system

If we go back in time to the year 1999, the internet was being used by merely 280 million people. Fast forward to today, even one single website handles more traffic than that figure. Take the example of an online retail giant, Amazon.com, which handles monthly traffic of 2.4 billion users. The way how we handle data change because of that. We need systems which are fast, and message or event driven, that why reactive programing is that popular.
A reactive system is characterized by four principles:

- **Responsive** a reactive system needs to handle requests in a reasonable time (I let you define reasonable).
- **Resilient** a reactive system must stay responsive in the face of failures (crash, timeout, error), so it must be designed for failures and deal with them appropriately.
- **Elastic** a reactive system must stay responsive under various loads. Consequently, it must scale up and down, and be able to handle the load with minimal resources.
- **Message driven** components from a reactive system interacts using asynchronous message passing.

Reactive Programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. ReactiveX or Rx is the most popular API for reactive programming. It’s built on the ideologies of the Observable Pattern, Iterator Pattern, and Functional

The build blocks for Rx code are the following:

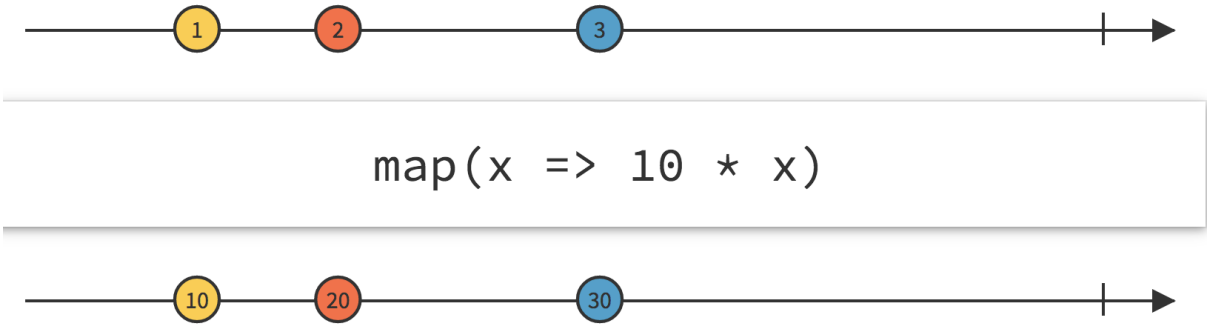
- **observables** representing sources of data
- **subscribers (or observers)** listening to the observables
- a set of methods for modifying and composing the data

The Observable emits streams of data, which the Observer listens and reacts to, setting in motion a chain of operations on the data stream.

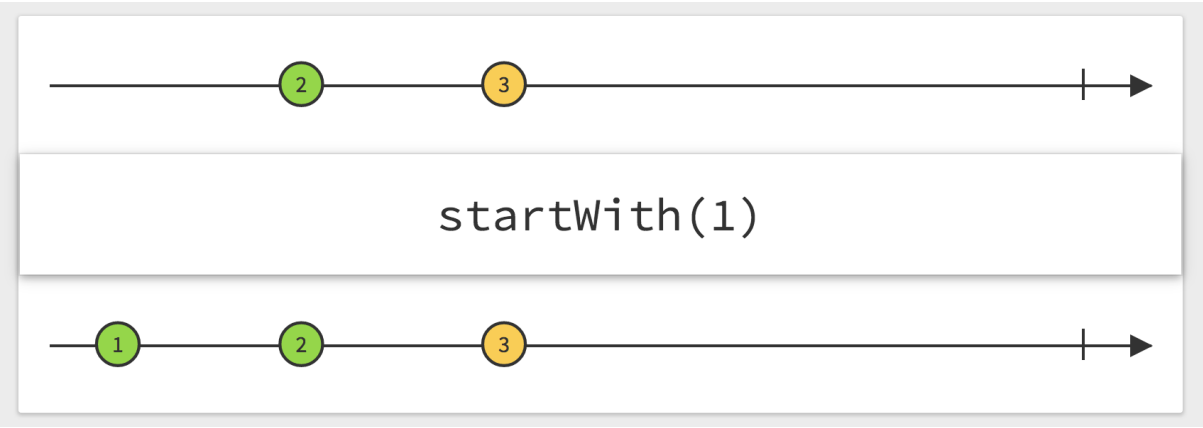
Operators allow you to transform, combine, manipulate, and work with the sequences of items emitted by Observables.

```
Observable<Integer> flow = Observable.range(1, 5)
.map(v -> v * v)
.filter(v -> v % 3 == 0)
.subscribe(System.out::println);
```

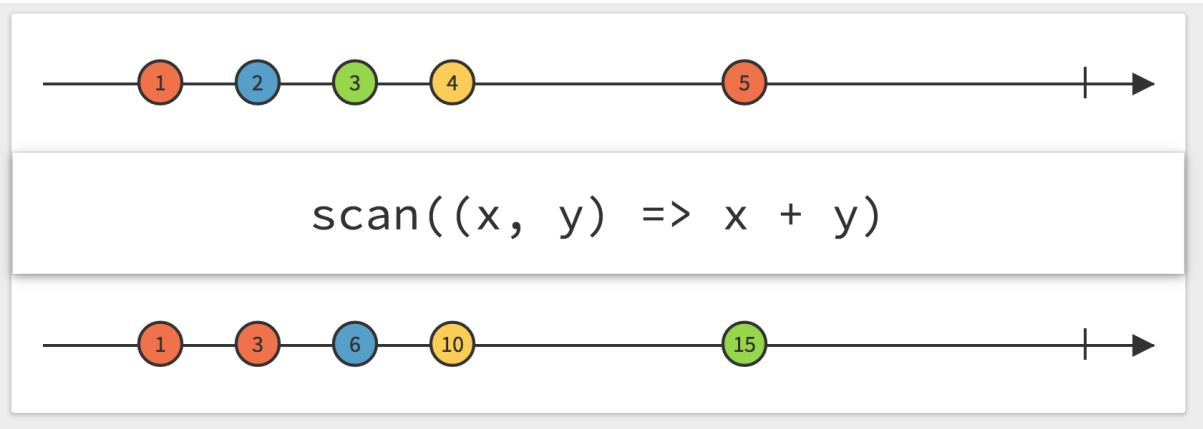
The most popular operators:
Map – transform the items emitted by an Observable by applying a function to each item.



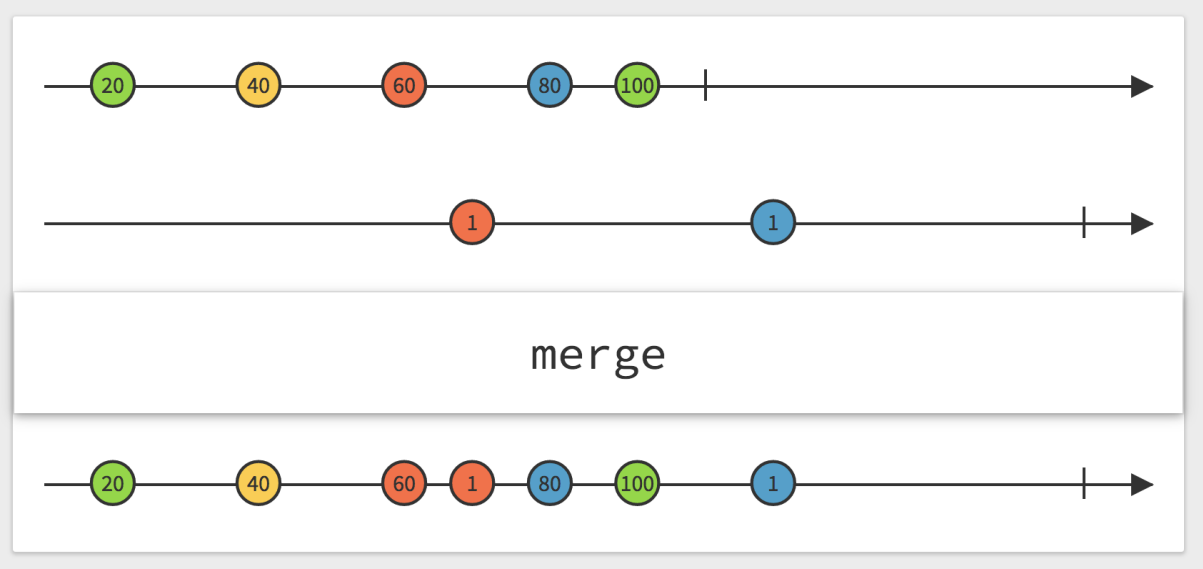
StartWith – emit a specified sequence of items before beginning to emit the items from the source Observable.



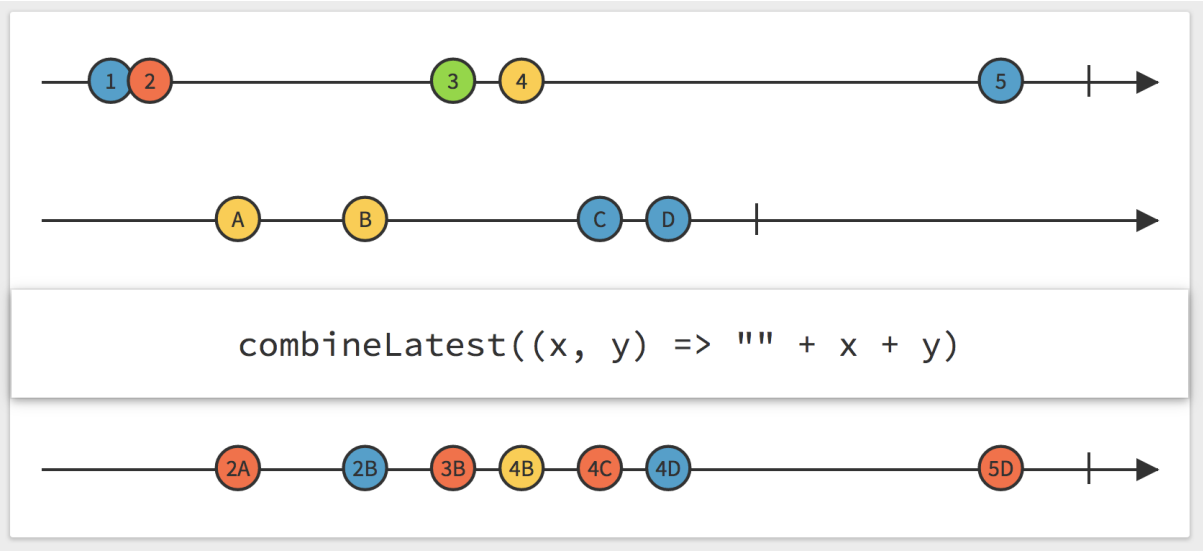
Scan – apply a function to each item emitted by an Observable, sequentially, and emit each successive value.



Merge – combine multiple Observables into one by merging their emissions



Combine latest – when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function.



Exercises:

1. There are two bank accounts. Each is represented by stream of changes (+$100, -$200 etc.). Using reactive programming write a code which provide sum of both accounts on each change. Remember initial value of money on both accounts is $0.
2. There are two bank accounts. This time reactive streams represent current value of money on the account ($1000, $200, $1600). Using reactive programming write a code which provide sum of both accounts on each change.
3. Write observable which emit next natural number every 20 seconds (start from 1) For this observable write a method which return factorial for current value. Try to optimize this code as much as you can (hint: you can use operators).