**Conferences by Oleg Bunin (Ontico)**

Professional conferences for IT developers

hedgehog_on_rainbow March 9, 2021 at 12:30 p.m

# Reactive programming in Java: how, why and is it worth it? Part II



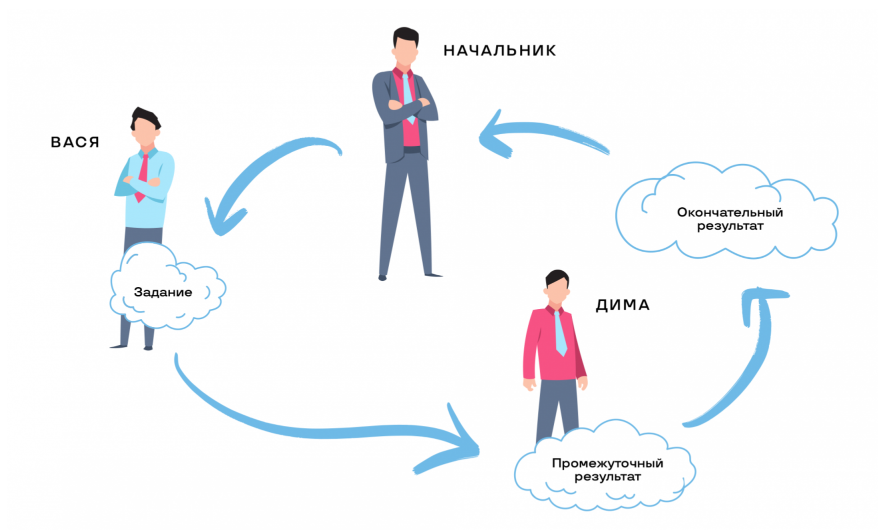
Blog of Oleg Bunin Conference Company (Ontiko), High performance*, Programming*, Java*, Parallel programming*

Reactive programming is one of the hottest trends of our time. Teaching it is a difficult process, especially if there are no suitable materials. This article can act as a kind of digest. At the RIT ++ 2020 conference, Luxoft Training expert and trainer Vladimir Sonkin spoke about the tricks of managing asynchronous data flows and approaches to them, and also showed examples in which situations reactivity is needed and what it can give.

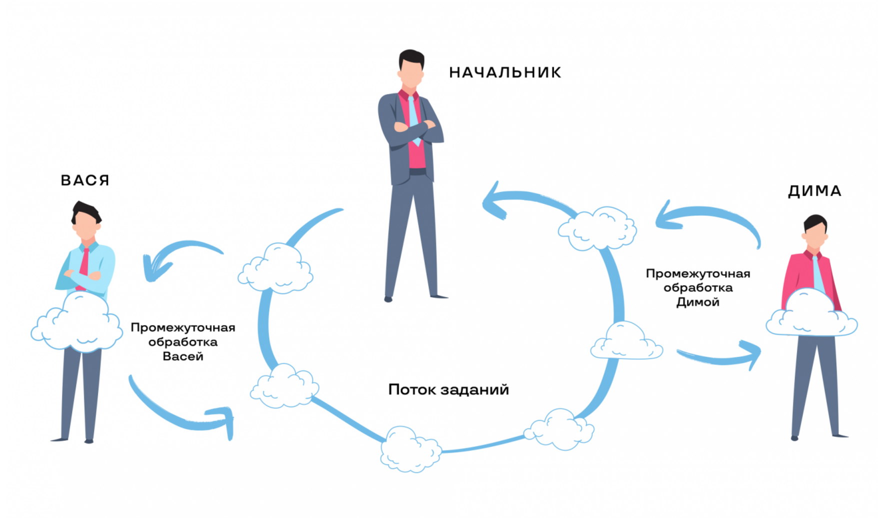
The first part of the article talked about what led to the emergence of reactive programming, where it is used, and what asynchrony can give us. It's time to talk about the next step to get the most out of asynchrony, and that's reactive programming.

## Reactivity

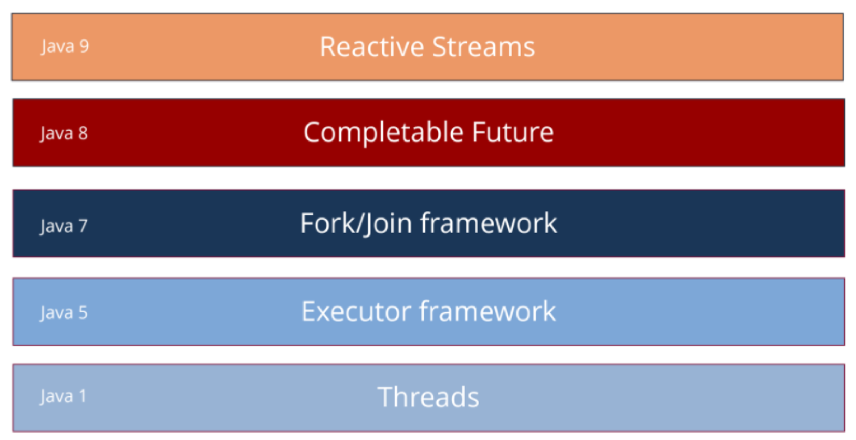
Reactive programming is asynchrony coupled with streaming data processing. That is, if there are no thread locks in asynchronous processing, but data is still processed in chunks, then reactivity adds the ability to process data by a thread. Remember that example when the boss entrusts a task to Vasya, he must pass the result to Dima, and return Dima to the boss? But our task is a certain portion, and until it is done, it cannot be passed on further. This approach really unloads the boss, but Dima and Vasya are periodically idle, because Dima needs to wait for the results of Vasya's work, and Vasya has to wait for a new task.




Now imagine that the task is divided into many subtasks. And now they are floating in a continuous stream:

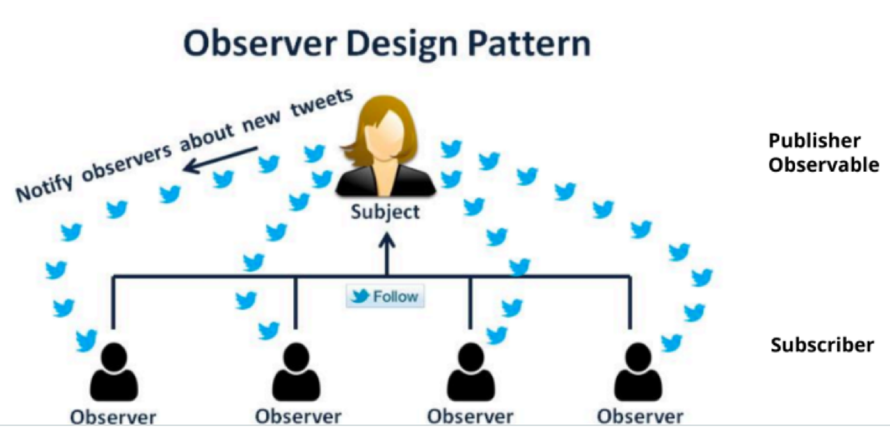



It is said that when Henry Ford came up with his assembly line, he quadrupled labor productivity, which helped him make automobiles affordable. Here we see the same thing: we have small portions of data, and a pipeline with a data stream, and each handler passes this data through itself, somehow transforming it. As Vasya and Dima, we have threads of execution (threads), thus providing multi-threaded data processing.




This diagram shows the different parallelization technologies that have been added to Java in different versions. As we can see, the Reactive Streams specification is at the top - it does not replace everything that came before it, but it adds the highest level of abstraction, which means its use is simple and effective. Let's try to figure this out.

The idea of reactivity is built on the Observer design pattern.



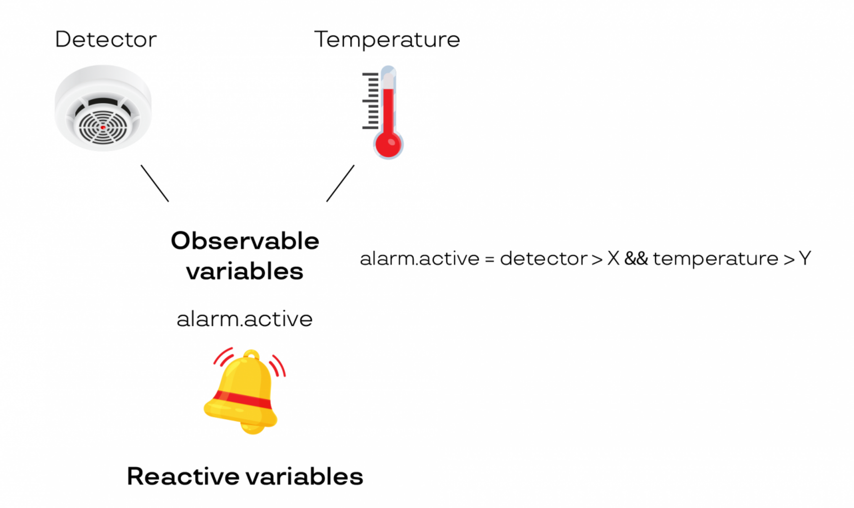
Let's remember what this pattern is. We have subscribers and what we subscribe to. Twitter is considered as an example here, but you can subscribe to a community or person, and then receive updates on any social network. After subscribing, as soon as a new message appears, all subscribers receive a notify, that is, a notification. This is the base pattern.

This scheme has:

- Publisher - the one who publishes new messages;
- Observer - the one who subscribes to them. In reactive streams, the subscriber is usually called the Subscriber. The terms are different, but in essence they are the same thing. In most communities, the terms Publisher/Subscriber are more familiar.

This is the basic idea on which everything is built.

One of the life examples of reactivity is the fire alarm system. Suppose we need to make a system that includes an alarm in case of excess smoke and temperature.




We have a smoke detector and a thermometer. When there is a lot of smoke and / or the temperature rises, the value on the corresponding sensors increases. When the value and temperature on the smoke sensor are above the threshold, the bell turns on and notifies of the alarm.

If we had a traditional rather than a reactive approach, we would write code that polls the smoke detector and temperature sensor every five minutes and turns the bell on or off. However, in the reactive approach, the reactive framework does this for us, and we only prescribe the conditions: the bell is active when the detector is greater than X, and the temperature is greater than Y. This happens every time a new event arrives.

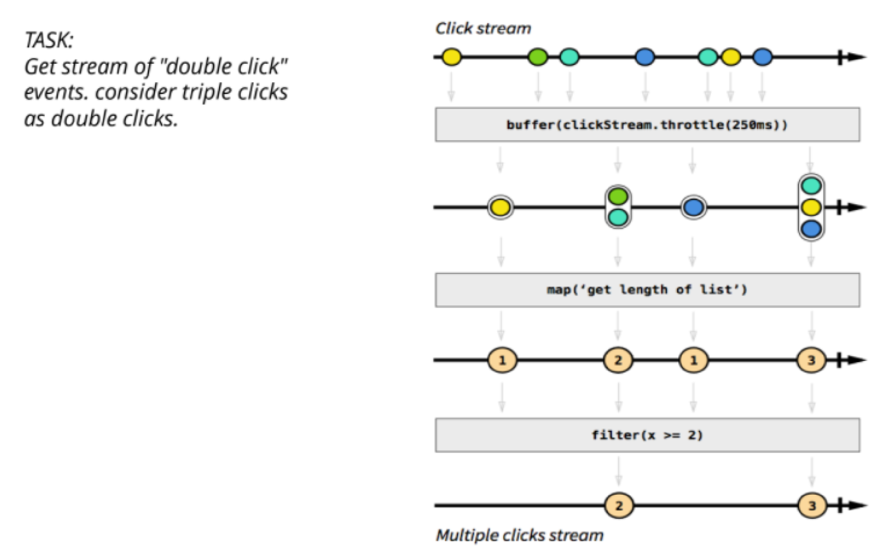
There is a stream of data coming from the smoke detector: for example, the value is 10, then 12, and so on. The temperature also changes, this is a different stream of data - 20, 25, 15. Each time a new value appears, the result is recalculated, which leads to turning the alert system on or off. It is enough for us to formulate the condition under which the bell should turn on.

If we return to the Observer pattern, we have a smoke detector and a thermometer - these are message publishers, that is, data sources (Publisher), and the bell is subscribed to them, that is, it is a Subscriber, or an observer (Observer).


Having dealt with the idea of reactivity a bit, let's delve into the reactive approach. We'll talk about reactive programming operators. Operators allow you to somehow transform data streams, changing data and creating new streams. For example, consider the **distinctUntilChanged** operator . It removes the same values that follow each other. Indeed, if the value on the smoke detector has not changed, why should we react to it and recalculate something there:


**Reactive approach**

Let's consider one more example: let's say we are developing a UI, and we need to track double clicks with the mouse. A triple click will be treated as a double click.



Clicks here are a stream of mouse clicks (in the diagram 1, 2, 1, 3). We need to group them. For this we use the throttle operator. We say that if two events (two clicks) occurred within 250 ms, they need to be grouped. The second diagram shows grouped values (1, 2, 1, 3). This is a stream of data, but already processed - in this case, grouped.

Thus, the initial stream was transformed into another one. Next, you need to get the length of the list ( 1, 2, 1, 3). We filter, leaving only those values that are greater than or equal to 2. Only two elements (2, 3) are left on the bottom diagram - these were double clicks. Thus, we have converted the initial stream into a double click stream.

This is reactive programming: there are input streams, somehow we pass them through the handlers, and we get an output stream. In this case, all processing occurs asynchronously, that is, no one is waiting for anyone.

Another good metaphor is the plumbing system: there are pipes, one connected to another, there are some valves, maybe there are purifiers, heaters or coolers (these are operators), pipes are divided or combined. The system works, water flows. It's the same in reactive programming, only water flows in the plumbing, and we have data.

You can think of stream cooking soup. For example, there is a task to cook a lot of soup as efficiently as possible. Usually a saucepan is taken, a portion of water is poured into it, vegetables are cut, etc. This is not streaming, but the traditional approach, when we cook soup in batches. We cooked this pan, then you need to put the next one, and after that - another one. Accordingly, it is necessary to wait until water boils again in a new pan, salt, spices, etc. dissolve. All this takes time.

Imagine this option: in a pipe of the desired diameter (enough to fill the pan), the water is immediately heated to the desired temperature, there are chopped beets and other vegetables. They come in whole and come out shredded. At some point, everything mixes up, the water is salted, etc. This is the most efficient cooking, supoconveyor. And that's the idea behind the reactive approach.

## Observable example

Now let's look at the code in which we publish events:

```
Observable<String> locations =
      Observable.just(“Bucharest",
           “Krakow", “Moscow",
           “Kiev", “Sofia"); //declaration

locations.subscribe(s -> System.out.println(s));
```

Observable.just allows you to put several values into the stream, and if ordinary reactive streams contain values that are stretched in time, then here we put them all at once - that is, synchronously. In this case, these are the names of cities that you can subscribe to in the future (here, for example, cities are taken that have a Luxoft training center).

The girl (Publisher) published these values, and Observers subscribe to them and print the values from the stream.

This is similar to Java 8 Streams. Both are synchronous streams. Both here and in Java 8, we know the list of values right away. But if a normal Java 8 stream were used, we would not be able to report something there. Nothing can be added to the stream: it is synchronous. In our example, the streams are asynchronous, that is, new events can appear in them at any time - say, if a training center opens in a new location in a year - it can be added to the stream, and reactive operators will correctly handle this situation. We added events and immediately subscribed to them:

```
locations.subscribe(s -> System.out.println(s)))
```

We can add a value at any time, which is displayed after some time. When a new value appears, we ask it to be printed, and the output is a list of values:

*Result*

```
Bucharest
Krakow
Moscow
Kiev
Sofia
```

In this case, it is possible not only to specify what should happen when new values appear, but also to additionally work out scenarios such as the occurrence of errors in the data flow or the completion of the data flow. Yes, although often data streams do not end (for example, readings of a thermometer or a smoke sensor), many streams can end: for example, a data stream from a server or from another microservice. At some point, the server closes the connection, and there is a need to somehow respond to this.

## Implementing and subscribing to an observer

Java 9 does not have an implementation of reactive streams, only a specification. But there are several libraries - implementations of the reactive approach. This example uses the RxJava library. We subscribe to the data stream and define several handlers, that is, methods that will be launched at the beginning of the stream processing (onSubscribe), upon receipt of each next message (onNext), upon an error occurring (onError) and upon completion of the stream (onComplete):

```
Observable<String> locations = Observable.just("Minsk", "Krakow", "Moscow", "Kiev", "Sofia");
Observer<Integer> observer = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
    }
    @Override
    public void onNext(Integer value) {
        System.out.println("Length: " + value);
    }
    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("Done.");
    }
};

locations.map(String::length).filter(l -> l >= 5).subscribe(observer);
```

Let's look at the last line.

```
locations.map(String::length).filter(l -> l >= 5).subscribe(observer);
```

We use the map and filter operators. If you've worked with Java 8 streams, you're certainly familiar with map and filter. Here they work exactly the same. The difference is that in reactive programming these values can appear gradually. Every time a new value comes in, it goes through all the transformations. So, String::length will replace the strings with the length in each of the strings.

In this case, you get 5 (Minsk), 6 (Krakow), 6 (Moscow), 4 (Kiev), 5 (Sofia). We filter, leaving only those that are greater than 5. We will get a list of string lengths that are greater than 5 (Kyiv will be eliminated). We subscribe to the final stream, after that the Observer is called and reacts to the values in this final stream. For each next value, it will output the length:

```
public void onNext(Integer value) {
System.out.println("Length: " + value);
```

That is, Length 5 will appear first, then Length 6. When our stream is completed, onComplete will be called, and at the end "Done." will appear:

```
public void onComplete() {
System.out.println("Done.");
```

Not all threads can terminate. But some are capable of it. For example, if we were reading something from a file, the stream will end when the file ends.

If an error occurs somewhere, we can react to it:

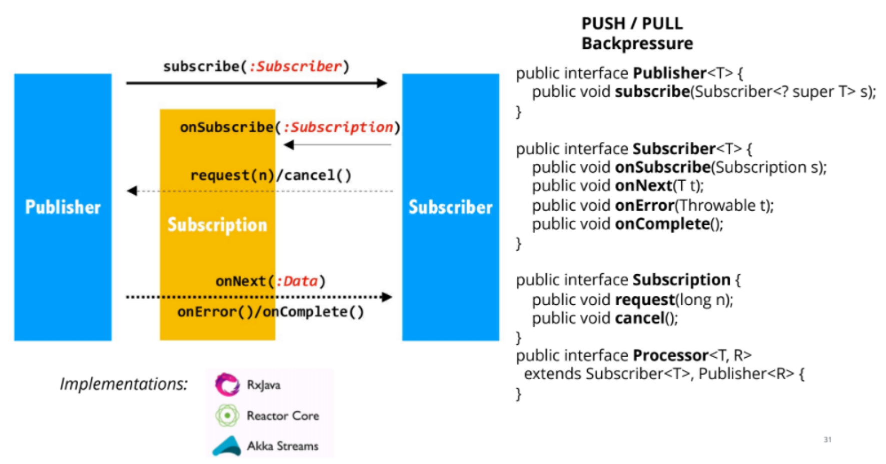
```
public void onError(Throwable e) {
e.printStackTrace();
```

Thus, we can react in different ways to different events: to the next value, to the completion of the thread, and to an error situation.

## Reactive Streams spec

Reactive streams entered Java 9 as a specification.

If the previous technologies (Completable Future, Fork/Join framework) have received their implementation in the JDK, then reactive streams have no implementation. There is only a very short specification. There are only 4 interfaces:




Considering our example from the Twitter picture, we can say that:

Publisher - a girl who posts tweets;

Subscriber - subscriber. It determines what to do if:

- We started listening to the stream (onSubscribe). When we have successfully subscribed, this function will be called;
- The next value appeared in the stream (onNext);
- There was an erroneous value (onError);
- The thread has ended (onComplete).

Subscription - we have a subscription that can be canceled (cancel) or request a certain number of values (request (long n)). We can define the behavior for each next value, or we can take the values manually.

Processor - the handler is two in one: it is both a Subscriber and a Publisher. It takes some values and puts them somewhere.

If we want to subscribe to something, we call Subscribe, we subscribe, and then we will receive updates every time. You can request them manually using request. And you can define behavior when a new message arrives (onNext): what to do if a new message appears, what to do if an error occurs, and what to do if the Publisher ends the stream. We can define these callbacks, or unsubscribe (cancel).

## PUSH / PULL models

There are two flow models:

- Push model - when there is a “pushing” of values.

For example, you subscribed to someone on Telegram or Instagram and receive notifications (they are called push messages, you do not request them, they come by themselves). It can be, for example, a pop-up message. You can define how to respond to each new message.

- Pull model - when we ourselves make a request.

For example, we don't want to subscribe because there is already too much information, but we want to go to the site ourselves and find out the news.

For the Push model, we define callbacks, that is, functions that will be called when the next message arrives, and for the Pull model, we can use the request method when we want to know what's new.

The pull model is very important for backpressure. What is it?

You may just be spammed with your subscriptions. In this case, it is unrealistic to read all of them, and there is a chance to lose really important data - they will simply drown in this message flow. When the subscriber, due to the large flow of information, cannot cope with everything that the Publisher publishes, Backpressure is obtained.

In this case, you can use the Pull model and request one message at a time, primarily from those data streams that are most important to you.

## Implementations

Let's take a look at existing implementations of reactive streams:

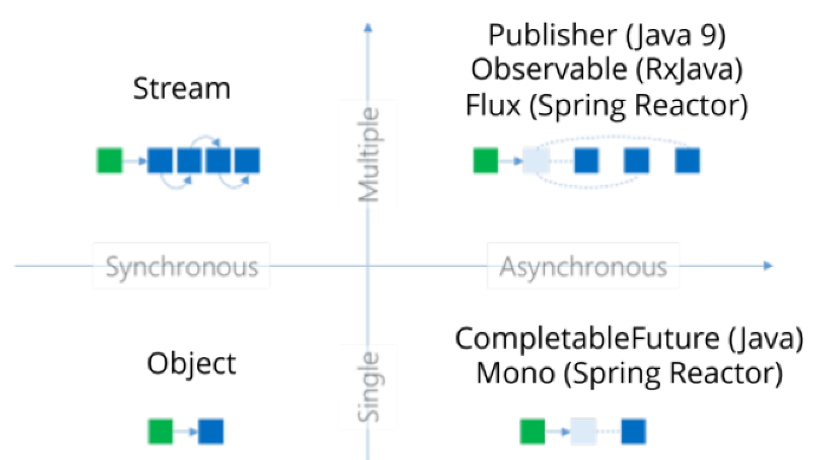
- RxJava. This library is implemented for different languages. In addition to RxJava, there is Rx for C#, JS, Kotlin, Scala, etc.
- reactor core. It was created under the auspices of Spring, and was included in Spring 5.
- Akka streams from Scala creator Martin Odersky. They created the Akka framework (the Actor approach), and Akka streams are the implementation of reactive streams that are friends with this framework.

In many ways, these implementations are similar, and they all implement the reactive streams specification from Java 9.

Let's take a closer look at Spring's Reactor.

## Function may return...

Let's summarize what a function can return:




- Single/Synchronous;

A normal function returns a single value, and does so synchronously.

A normal function returns a single value, and does so synchronously.

- Multipple/Synchronous;

If we are using Java 8, we can return a Stream from a function. When many values are returned, they can be sent for processing. But we cannot send data for processing before all of it is received - after all, Streams work only synchronously.

- Single/Asynchronous;

The asynchronous approach is already used here, but the function returns only one value:

- or CompletableFuture (Java), and after some time an asynchronous response comes;
- or Mono, which returns a single value in the Spring Reactor library.

- Multiple/Asynchronous.

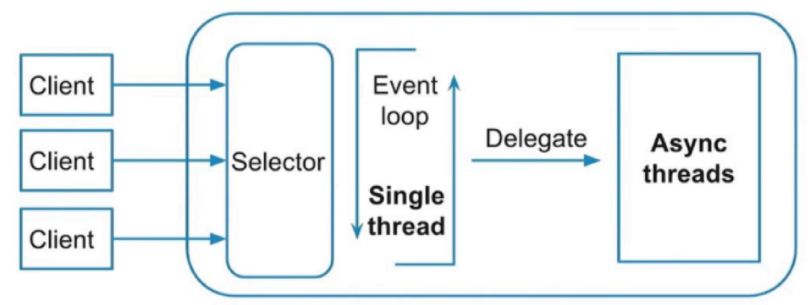
And here is just - jet streams. They are asynchronous, that is, they return a value not immediately, but after some time. And it is in this variant that you can get a stream of values, and these values will be stretched in time. Thus, we combine the advantages of Stream streams, which allow you to return a chain of values, and asynchrony, which allows you to postpone the return of a value.

For example, you are reading a file and it changes. In the case of Single/Asynchronous, after some time you get the entire file. In the Multiple/Asynchronous case, you get a stream of data from a file that you can start processing right away. That is, you can simultaneously read data, process it, and, possibly, write it somewhere. . Reactive asynchronous threads are called:

- Publisher (in the Java 9 specification);
- Observable (в RxJava);
- Flux (в Spring Reactor).

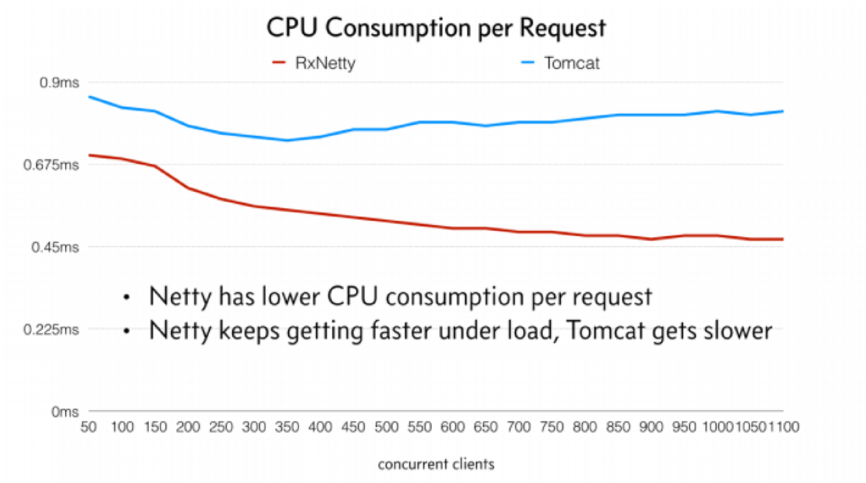
## Netty as a non-blocking server

Let's look at an example of using Flux reactive streams along with Spring Reactor. Reactor is based on the Netty server. Spring Reactor is the core of the technology we will be using. And the technology itself is called WebFlux. For WebFlux to work, you need an asynchronous, non-blocking server.

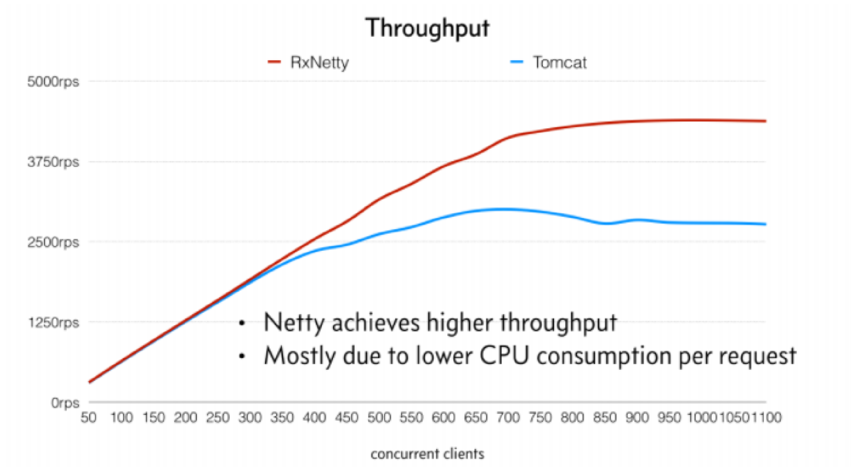



The way the Netty server works is similar to how Node.js works. There is a Selector - an input stream that receives requests from clients and sends them for execution to freed threads. If Tomcat is used as a synchronous server (Servlet container), then Netty is used as an asynchronous one.

Let's see how much computational resources Netty and Tomcat use to execute a single request:



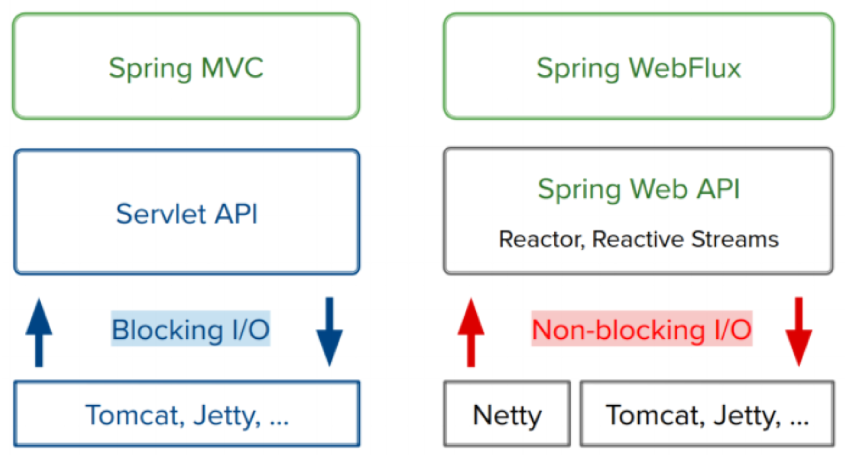

Throughput is the total amount of data processed. With a small load, up to the first 300 users, RxNetty and Tomcat have the same, and after Netty it goes into a decent lead - almost 2 phrases.




## Blocking vs Reactive

We have two request processing stacks:

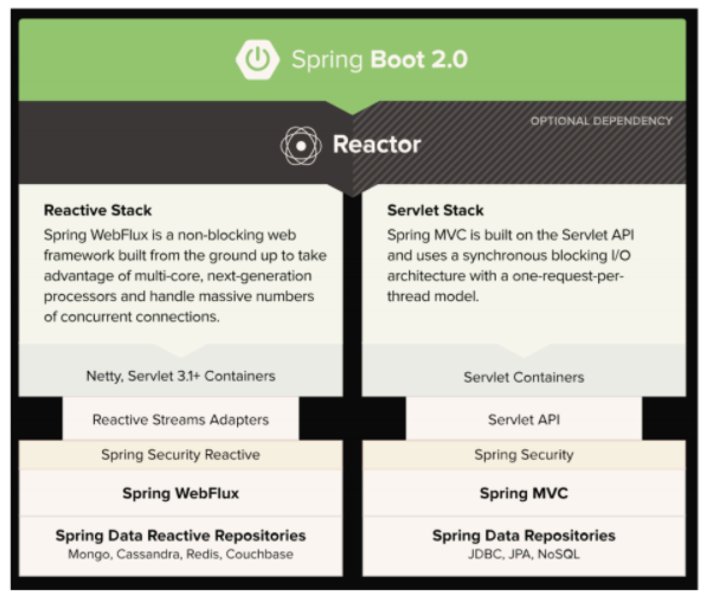
- Traditional blocking stack.
- Non-blocking stack - everything happens asynchronously and reactively in it.




In the blocking stack, everything is built on the Servlet API, in the reactive non-blocking stack, everything is built on Netty.

Let's compare the reactive stack and the Servlet stack.

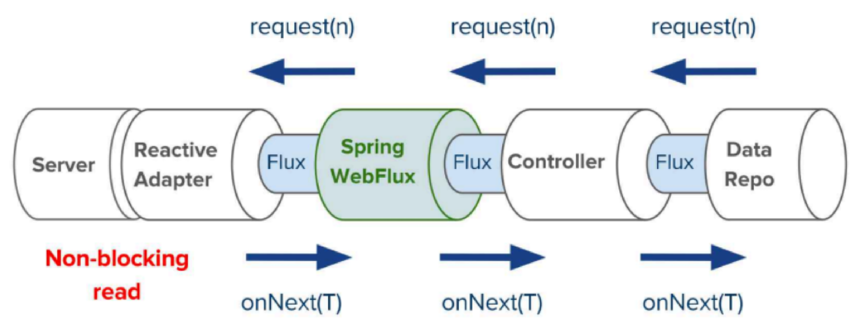
Reactive Stack uses Spring WebFlux technology. For example, reactive streams are used instead of the Servlet API.




For us to get a tangible performance advantage, the entire stack must be reactive. Therefore, reading data must also come from a reactive source.

For example, if we have standard JDBC, it is a non-reactive blocking source because JDBC does not support non-blocking I/O. When we send a query to the database, we have to wait until the result of this query arrives. Accordingly, it is not possible to gain an advantage.

In the Reactive Stack, we take advantage of reactivity. Netty works with the user, Reactive Streams Adapters with Spring WebFlux, and at the end there is a reactive base: that is, the entire stack is reactive. Let's look at it in a diagram:




Data Repo is a repository where data is stored. If there are requests, for example, from a client or an external server, they enter the controller through Flux, are processed, added to the repository, and then the response goes in the opposite direction.

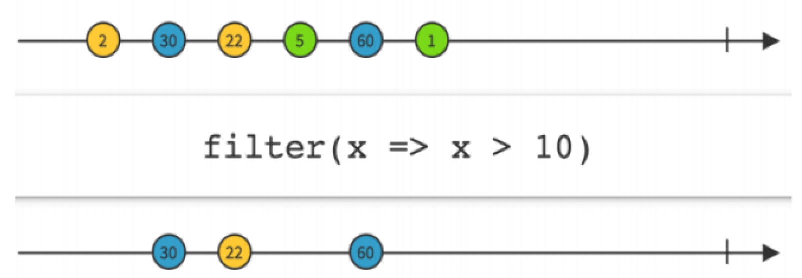
At the same time, all this is done in a non-blocking way: we can use either the Push approach, when we determine what to do with each next operation, or the Pull approach, if there is a possibility of Backpressure, and we want to control the data processing speed ourselves, and not receive everything data at once.

## Operator

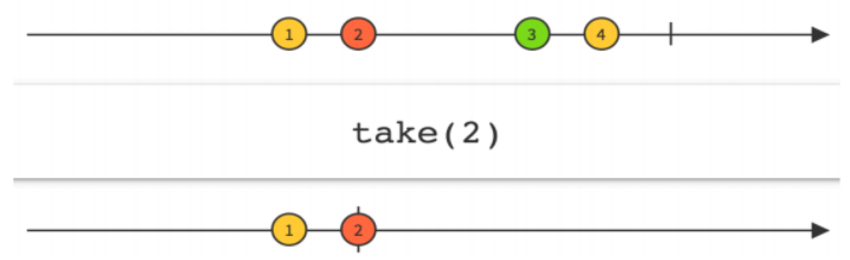
There are a huge number of operators in reactive streams. Many of them are similar to those found in regular Java streams. We will cover only a few of the most common operators that we will need for a practical example of using reactivity.

### Filter operator

You are probably already familiar with filters from the Stream interface.



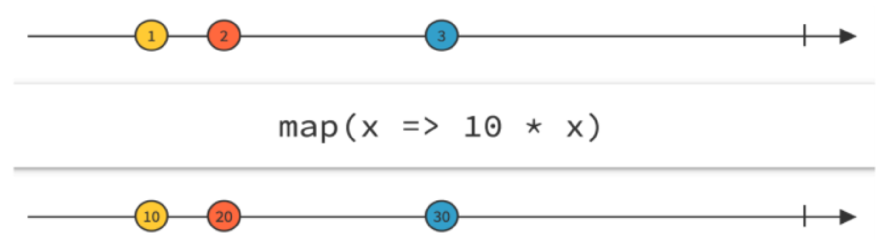

The syntax of this filter is exactly the same as the regular one. But if the Java 8 stream has all the data at once, here it can appear gradually. The arrows to the right are the timeline, and the circles are the emerging data. We can see that the filter leaves only values greater than 10 in the final stream.




Take 2 means that only the first two values need to be taken.

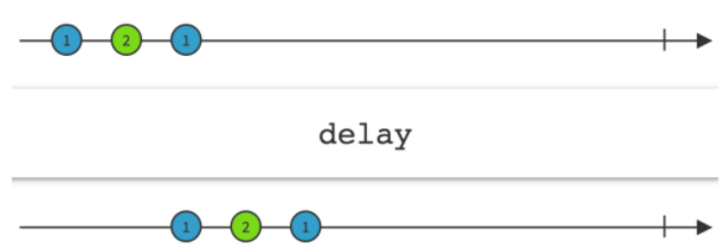
### Map operator

The Map operator is also familiar:




This is an action that happens with each value. Here - multiply by ten: it was 3, it became 30; there were 2, now there are 20, etc.

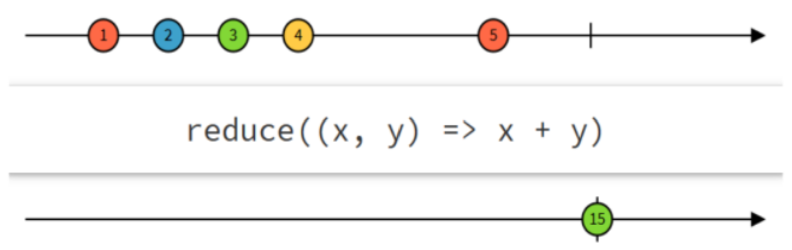
### Delay operator




Delay: All operations are shifted. This statement may be needed when the values are already being generated, but the preparatory processes are still in progress, so you have to postpone processing the generated, but the preparatory processes are still in progress, so you have to postpone processing the data from the stream.

**Reduce operator**

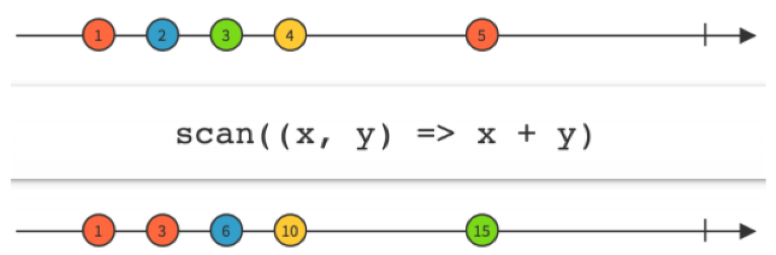
Another well-known operator:




It waits for the end of the thread (onComplete) - in the diagram it is represented by a vertical bar. After that, we get the result - here it is the number 15. The reduce operator added up all the values that were in the stream.

**Scan operator**

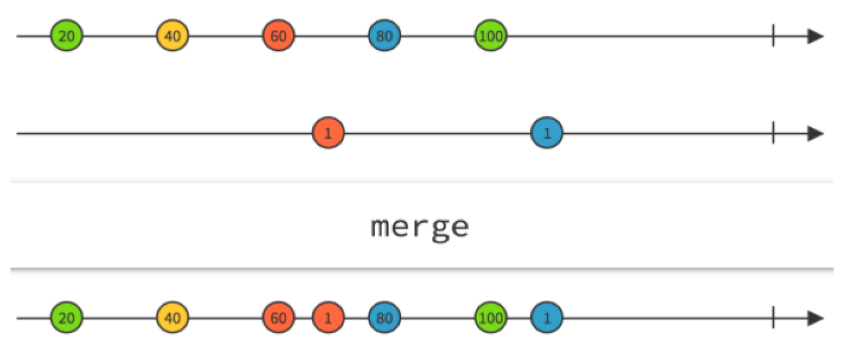
This operator differs from the previous one in that it does not wait for the end of the thread.




The scan operator calculates the current value as a running total: first it was 1, then it added 2 to the previous value, it became 3, then it added 3, it became 6, another 4, it became 10, etc. We got 15 at the output. Then we see a vertical line - onComplete. But maybe it will never happen: some threads don't terminate. For example, a thermometer or a smoke detector has no termination, but scan will help calculate the current total value, and with some combination of operators, the current average value of all data in the stream.

**Merge operator**

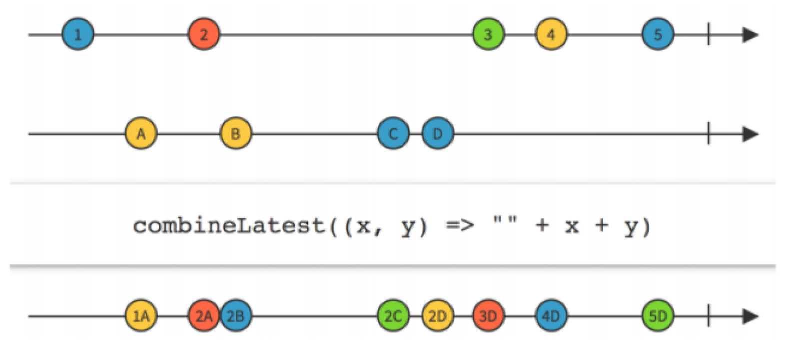
Combines the values of two streams.




For example, there are two temperature sensors in different places, and we need to process them uniformly, in a common thread.

**Combine latest**

Given a new value, combines it with the last value from the previous stream.



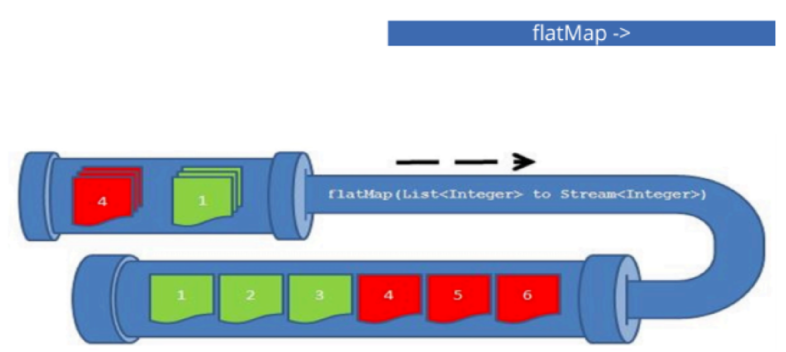

If a new event occurs in a thread, we combine it with the last received value from another thread. Let's say that in this way we can combine the values from the smoke sensor and the thermometer: when a new temperature value appears in the temperatureStream, it will be combined with the last smoke value received from the smokeStream. And we will get a couple of values. And already for this pair, you can perform the final calculation:

```
temperatureStream.combineLatest(smokeStream).map((x, y) -> x > X && y > Y)
```

As a result, at the output we get a stream of true or false values - turn the bell on or off. It will be recalculated each time a new value appears in temperatureStream or smokeStream.

**FlatMap operator**

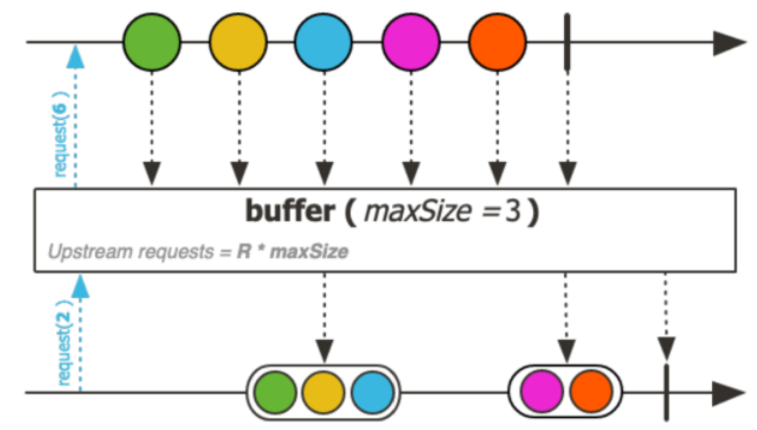
This operator is probably familiar to you from Java 8 streams. The elements of a stream in this case are other streams. It turns out a stream of streams. They are inconvenient to work with, and in these cases we may need to “flatten” the flow.




You can imagine such a flow as a conveyor, on which boxes of spare parts are placed. Before we start using them, the parts must be taken out of the boxes. This is exactly what the flatMap operator does.

Flatmap is often used when processing a stream of data received from a server. Because the server returns a stream so that we can process individual data, this stream must first be “deployed”. This is what flatMap does.

**Buffer operator**




It is an operator that helps to group data. The output of Buffer is a stream whose elements are lists (List in Java). It can come in handy when we want to send data not one at a time, but in chunks.

We said from the very beginning that reactive streams allow you to split a task into subtasks, and process them in small portions. But sometimes it's better to do the opposite, to collect many small parts into blocks. Let's say, continuing the example with the conveyor and spare parts, we may need to send spare parts to another factory (another server). But sending each individual spare part is inefficient. It is better to collect them in boxes, say 100 pieces, and send them in larger batches.

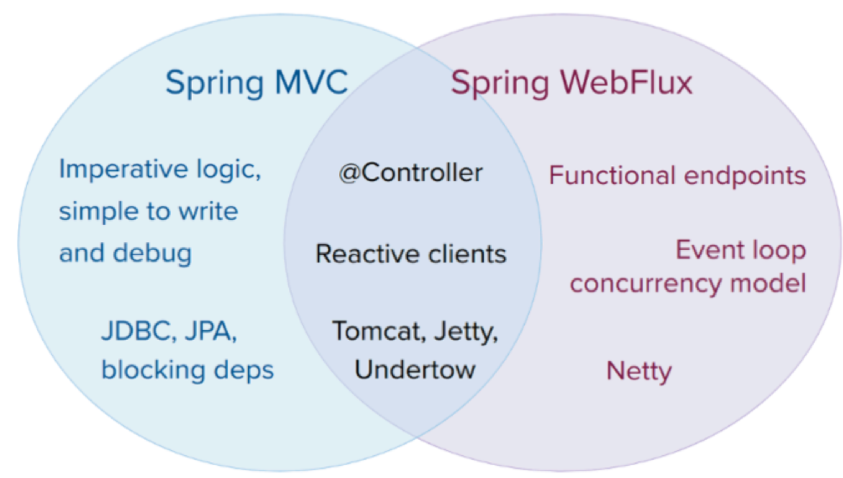
In the diagram above, we group individual values into three elements (since there were five in total, we got a "box" of three, and then of two values). That is, if flatMap unpacks data from boxes, buffer, on the contrary, packs them.

There are more than a hundred reactive programming operators in total. Only a small part has been disassembled here.

## Total

There are two approaches:

- Spring MVC is a traditional model that uses JDBC, imperative logic, etc.
- Spring WebFlux which uses a reactive approach and a Netty server.




There is something that unites them. Tomcat, Jetty, Undertow can work with both Spring MVC and Spring WebFlux. However, the default server in Spring for working with a reactive approach is Netty.

> The HighLoad++ Spring 2021 conference will take place on May 20 and 21, 2021. You can buy tickets now.
>
> And very soon another interesting event will take place, this time online: on March 18 at 17:00 Moscow time there will be a meetup "How the most modern payment system in the WORLD works: architecture and security".
>
> Together with the developers **of Mir Plat.Form** , we will figure out how to ensure the stability of all services already at the design stage and how to make sure that the system can develop without affecting business processes. Meetap will be of interest to developers, architects and security specialists.
>
> Do you want to receive the Saint HighLoad++ 2020 mini-conference conference materials for free? Subscribe to our newsletter.

**Tags:** reactive programming, java, conference, upload

**Hub:** Blog of Oleg Bunin Conference Company (Ontiko), High performance, Programming, Java, Parallel programming



Conferences by Oleg Bunin (Ontico)
Professional conferences for IT developers
Twitter Telegram In contact with

22 Karma 0 Rating
**Alena Mironova** @hedgehog_on_rainbow
Content editor

**Comments** 5

**vba** 03/09/2021 at 13:50

For me, well, Spring should be punished and not mentioned about this company, every time when it comes to reactive programming in Java, despite Reactor or WebFlux. For ten years, this company has done everything to bury reactive programming in Java. They were more interested in releasing crafts like Spring Roo, and not, for example, integrating SpringMVC with Mina, and then Netty. Shame, Shame, Shame...

-3 Answer

**AntonioXXX** 03/14/2021 at 00:08

In general, a very good article, everything is clear and detailed. And judging by just one comment before me, everyone understood everything too.

+1 Answer

**traps** 03/29/2021 at 15:58

Thank you for the article! Good introduction to the problem.

0 Answer

when 03/29/2021 at 15:58

The article says "We will cover only a few of the most common operators that we will need for a practical example of using reactivity.", but I could not find this practical example. Will he be in some other article? The operators themselves are described just fine, thank you very much. And the whole article as a whole is also just wonderful.

0 Answer

hedgehog_on_rainbow 03/29/2021 at 15:58

Thanks for the feedback! We are thinking about making a separate article with practical examples.

0 Answer

Only authorized users can post comments. Sign in please.

## Publications

BEST OF THE DAY SIMILAR

amazing_mike 13 hours ago

### Robot-not-vacuum cleaner with a knife or how we made smart scissors on wheels

7 min 3.6K

case

+49 16 30 +30

oldadmin 10 hours ago

### How to use GitLab under sanctions?

Average 5 min 6.1K

+31 34 19 +19

T1_Consulting 12 hours ago

### How Taras became Senior+ due to CS 1.6 and grandfather with CHP

Simple 5 min 6.7K

case

+24 13 11 +11

Kudesnick33 11 hours ago

### Registers vs libraries on the example of hearts

15 min 2.1K

case

+22 26 20 +20

monreve 17 hours ago

### Tiny11: thinner Windows 11. Can it really run on older PCs? Part 1

4 min 24K

+18 42 77 +77

show more