Reactive Programming in Java: How, Why, and Is It Worth Doing? Part II. Reactivity

Reactive programming is one of the most popular trends nowadays. Learning this approach is a cumbersome process, especially if you do not have relevant materials. This article may serve as a kind of digest.

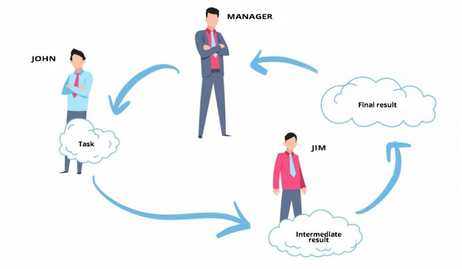
19 Jul 2021 4918

At the RIT++ 2020 conference, our Luxoft Training expert and trainer Vladimir Sonkin talked about various tricks to manage asynchronous data flows and approaches to them and showed some examples where you might need reactivity and what it could provide.

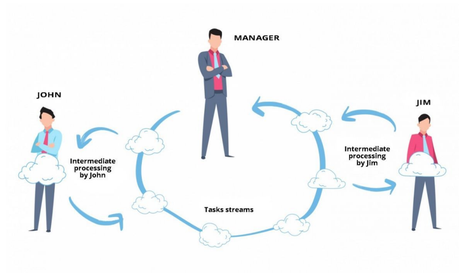
Our first series of articles focused on what led to reactive programming, where it is applied, and what asynchronicity can bring. Not it is time to talk about the next step that will enable you to get maximum benefits from asynchronicity - reactive programming.

Reactivity

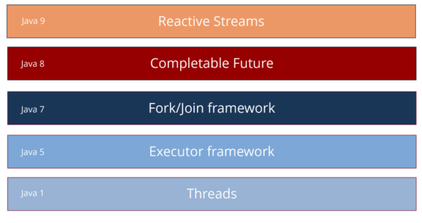
Reactive programming is asynchronicity combined with streaming data. In other words, there is no thread blocking in asynchronous processing, yet data is processed in portions. Reactivity adds a capability of data processing in a flow. Remember the example where the manager assigns a task to John, who should pass the result to Jim, and Jim should submit it to the manager? But our task is a specific portion, and you cannot pass it on until it is done. Such an approach can help removes some tasks from the manager, but Jim and John stay idle from time to time because Jim has to wait for John's results and John has to wait for a new task.



And now, imagine that a task has been split into multiple subtasks, and they are flowing in a continuous stream.

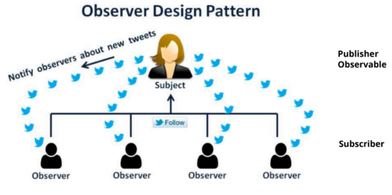


When Henry Ford invented the conveyor, he managed to increase performance four times, making cars affordable. Here we can see the same thing: we have small portions of data, and the conveyor with data thread and every handler passes the data through them, transforming it in a certain way. Execution threads act as John and Jim, providing multi-thread data processing.



This scheme shows various streaming technologies that have been added to different Java versions. As we can see, the Reactive Streams specification is on top: it does not replace everything that was there before but adds the highest level of abstraction, which makes it easy and efficient in use. Let's try to understand that.

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



Remember this pattern. We have subscribers and something for which they subscribe. Twitter is an example here, but you can subscribe to any community or person and then get updates on any social network. Once subscribed, all subscribers receive notifications when a new message appears. This is a basic pattern.

In this scheme, there are:

- Publishers — those who publish new messages;

Other articles

How to implement Slowly Changing Dimensions(SCD) Type 2 in Spark?

How to incrementally migrate the data from RDBMS to Hadoop using Sqoop Incremental Append technique?

Why MongoDB don't fetch all the matching documents for the query fired

How to solve the issue of full disk utilization in HDFS Namenode

Can We Use HDFS as Back-up Storage?

How to do Indexing in MongoDB with Elastic Search? Part 1

How to do Indexing in MongoDB with Elastic Search? Part 2

How to store data on browser using NoSQL IndexedDB?

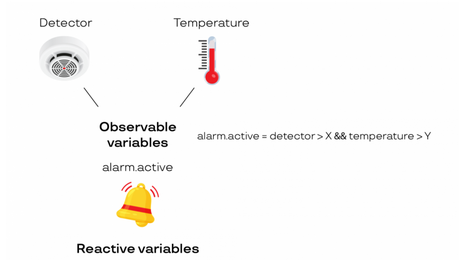
How to Apply MBTI in HR: Motivation for every day. Groups of People & their Motivations

Check out our new courses

- Observers — those who are subscribed to them. In the case of reactive streams, observers are usually called Subscribers. These are two different terms for essentially the same thing. Most communities use the terms Publisher/Subscriber.

It's a basic idea on which everything is built.

A real-life example of reactivity is a fire alarm system. Suppose we need to build a system that turns on the alarm in case of excessive smoke and high temperature.

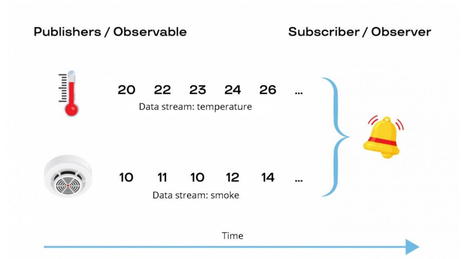


We have a smoke detector and a thermometer. Once there is too much smoke and/or the temperature is too high, the respective sensors' values increase. When temperature and smoke values exceed a certain threshold, the bell rings, and the alarm signals turn on.

If we had used a traditional (non reactive) approach, we would write code that interrogates the smoke detector and temperature sensor every five minutes and turns the alarm bell on or off. In the case of the reactive approach, however, this is done by a reactive framework, and we just set conditions: alarm is on when the smoke detector value is above X, and the temperature is above Y. It works every time a new event arrives.

A data thread goes from the smoke detector: for example, value 10, then 12, etc. The temperature could change too, it is another data thread — 20, 25, 15. Every time a new value appears, the result is recalculated, which leads to turning the alarm on or off. We should just set a condition under which the alarm must turn on.

In terms of the Observer pattern, the smoke detector and temperature sensor are Publishers (data sources), and the alarm bell subscribes to them. It is a Subscriber (or Observer).



Now that we got a basic idea of reactivity let's go into some details of the reactive approach. We discuss reactive programming operators. Operators enable us to transform data threads in a certain way, changing data and creating new threads. For example, consider the operator **distinctUntilChanged**. It removes the same values that follow one another. Indeed, if the value of the smoke detector remains the same, we should react to it and recalculate something:

The original article can be found here.

Interested in learning how to program with Java or in upgrading your Java programming skills? Check out our trainings.

Share the knowledge

Back to the list