07: Reactive Programming (RP) in Java Interview Q&A

Posted on March 18, 2016 by Arulkumaran Kumaraswamipillai

Q1. Explain "**pull**" vs "**push**" paradigms (aka imperative/interactive vs reactive) with respect to processing data in programming?

A1.

Interactive/imperative Programming (**Pull**): is all about asking for something and getting it in return. One common pattern in this world is the **iterator pattern**, which loops through and pull data out of a collection or stream. Prints 1 to 10 by **pulling** data out of the stream.

```
1
```

```
import java.util.stream.IntStream;

public class Imperative {

    public static void main(String[] args) {
        IntStream.range(1, 11)
                 .forEach(number -> System.out.p
    }
}

```

"forEach" iterates over the data and pulls data.

In "imperative Programming" a consumer is in control by pulling data.

Reactive Programming (**Push**): is all about registering interest (i.e. subscribing) and then we have items **pushed** to us asynchronously as they become available. This is basically the **observer pattern**. The "listening" to the stream is called subscribing. The functions we are defining are observers. The stream is the subject (or "observable") being observed.

Same code using "**CompletableFuture**" introduced in Java 8.

```

import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Reactive {

    public static void main(String[] args) {

        //asynchronously generate numbers.
        CompletableFuture<List<Integer>> numbers
            try {
                TimeUnit.SECONDS.sleep(5); //jus
            } catch (Exception e) {
                e.printStackTrace();
            }
             List<Integer> collectNumbers = IntS
             return collectNumbers;
        });

        System.out.println("Waiting for the numb

        //This reacts to completion of the numbe
        //numbersStage pushes data onto printing
        CompletableFuture<Void> printingStage =

```


16 Technical Key Areas

```
29          System.out.println("We get here because
30
31          printingStage.join();  //wait till print
32
33          System.out.println("Completed printing n
34      }
35  }
36
```

Output

```
1
2  Waiting for the numbers to be generated......
3  We get here because processing is asynchronous...
4  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5  Completed printing numbers......
6
```

In “Reactive Programming” a producer is in control by pushing data.

Key Points:

1) The CompletableFuture object “numbersStage” returns immediately regardless of numbers being generated or not. The number generation happens asynchronously on a separate thread and the code moves to next line on the main thread by printing “Waiting for the numbers to be generated……”.

2) The “printingStage” registers its interest (aka subscribes) to the “numberStage” for the numbers by waiting asynchronously. The code then moves on and prints “We get here because processing is asynchronous……”.

3) The “printingStage.**join**();” is the first **blocking call** in the program waiting for the printing to complete. Otherwise the execution will complete.

4) When the “numbersStage” is completed, it **pushes** the results on to the “printingStage”.

5) The “printingStage” **reacts** to the data being available, and prints the numbers. Because the “printingStage” reacts to

⊞ Java Debugging (21)
⊞ Judging Experience I
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

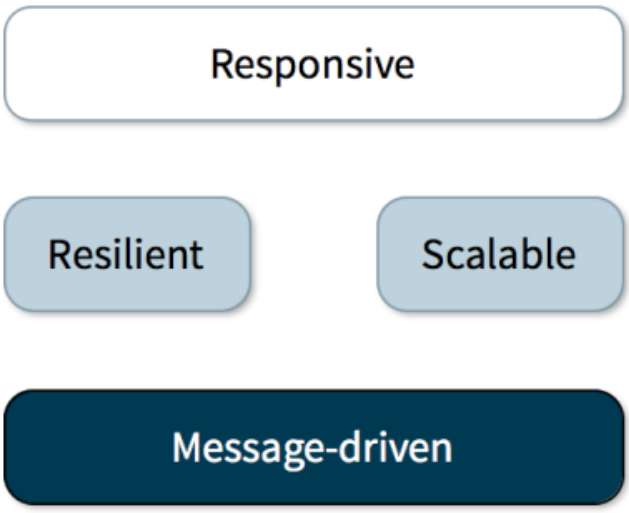
the event of data being available asynchronously, it is called "reactive programming".

Q2. What is reactive programming?
A2. It is a bit complicated to get a unified definition, depending on who you talk to. Reactive programming respond to a stimuli.

1) Reactive programming is programming with asynchronous data streams.

2) Reactive programming is all about reacting to events (i.e **event driven**), reacting to load conditions (i.e. **scalable**), reacting to error conditions (i.e. **resilient**), and react quickly to user or system requests (i.e. being more **responsive**). Events are nothing but messages. So, it is also known as the **message driven programming**.

3) Reactive programming is a programming paradigm oriented around data flows and the propagation of change.



4 Principles of reactive programming

Q3. Why adopt reactive programming?
A3.

Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

How good are your .....?

open all | close all
Career Making Know-
Job Hunting & Resum

1. More concise code: Reactive Programming raises the level of abstraction of your code. You can just focus on events and then chain events. The events define the business logic. You don't have to constantly fiddle with a large amount of implementation details, hence code in RP will likely be more concise.

2. Multi-threading code is hard to write & test: To take advantage of today's multi-core architecture, our applications should be able to manage multiple concurrent tasks and divide large tasks into smaller segments that can run in parallel. Writing multi-threaded code that works as expected and actually improves performance is not a simple task. The code is also hard to test.

So, compared to the traditional approach of multiple-threading, which communicates through shared, synchronized state, a reactive application is composed of **loosely coupled event handlers**.

3. Better scalability as the reactive components are loosely coupled, location independent, and communicate via passing messages. This makes it much easier to **scale up** (adding more instances of a component on a multi-core node) and **scale out** (adding more nodes to a cluster).

4. More resilient compared to traditional multi-thread applications as the loosely coupled components, with strongly encapsulated state, are managed by observers/supervisors which prevent cascading failures. In the above code example, exception can be handles with line shown below

```
1
2 //.....
3
4 numbersStage.exceptionally((ex) -> {System.out.pr
5
6 System.out.println("Waiting for the numbers to be
7
8 //....
9
```

5. Better performance & maintainability as a reactive programming paradigm is oriented around data flows and the propagation of change. There is a lesser need for storing & reading data.

Q4. What is an actor based concurrency?
A4. Actor-based applications are about passing asynchronous messages among multiple actors. Actors can pass messages back and forth, or even pass messages to themselves. An actor can pass a message to itself in order to finish processing a long-running request after it services other messages in its queue first. An actor is a construct with the following properties:

1) A **mailbox** for receiving messages.
2) The **actor's logic**, which relies on pattern matching to determine how to handle each type of message it receives.
3) Isolated state—rather than shared state—for storing context between requests.

This actor based architecture is loosely coupled & non blocking. The invoked routine is encapsulated by an actor, and this opens up many possibilities, like distributing routines across a cluster of machines. So, a huge benefit of actor-based concurrency is that its event driven architecture enables it to be very very scalable across network boundaries.

Akka is an actor-based toolkit and runtime for building highly concurrent, distributed & resilient message-driven applications on the JVM. You can build industrial strength reactive applications with Akka.

RxJava is another framework like Akka for developing reactive applications. It is composed of asynchronous and event-based programs using observable sequences for the Java VM.

Q5. How does RxJava work in a nutshell?
A5. Using RxJava, you can represent multiple asynchronous data streams (i.e. data sources) like stock price feeds, web

service requests for orders, etc., and subscribe to the event stream using the Observer. A producer stream produces data at different points in time. An observer (i.e a consumer) is notified whenever data in that stream and does something with it. You can then apply functions like map, filter, and reduce. A reactive programming paradigm is oriented around data flows and the propagation of change.

Popular Posts

♦ 11 Spring boot interview questions & answers

825 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

766 views

18 Java scenarios based interview Questions and Answers

400 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

388 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

295 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

285 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

279 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

239 views

001B: ♦ Java architecture & design concepts interview questions & answers

201 views

Bio Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ 11: Threads performing tasks by talking to each other

12: Complete the trades matching logic ›

Posted in Low Latency, member-paid, Reactive Programming

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from "Blah blah" to "Wow"? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

 ↑