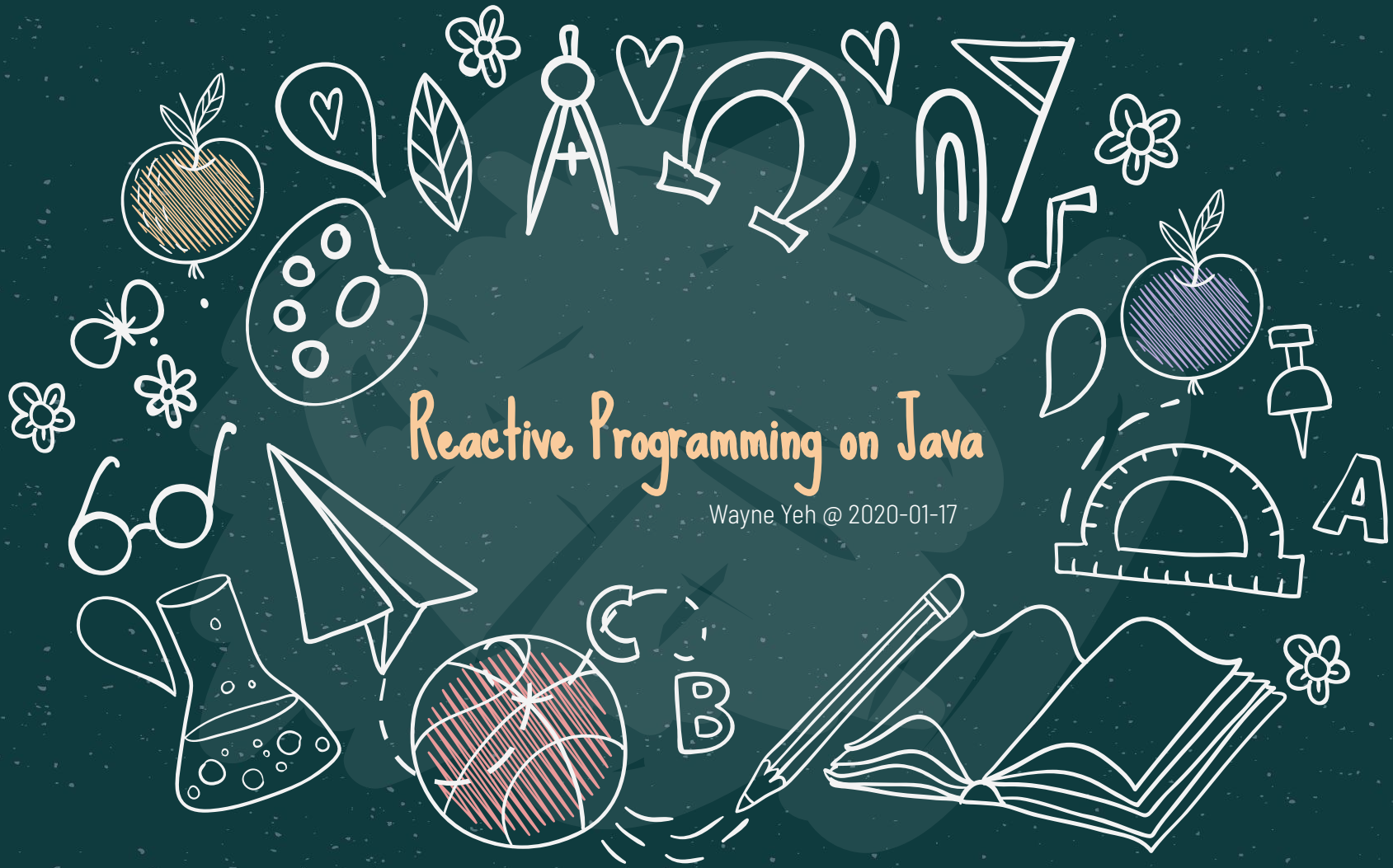

Reactive Programming on Java
Wayne Yeh @ 2020-01-17
A
B
C

01

Evolution of Java

How did we improve utilization?

02

Blocking Can Be Wasteful

But are we saved by asynchronous?

03

Reactive

Reactive system and reactive programming

Modern Java in Action https://www.oreilly.com/library/view/modern-java-in/9781617293566/
Project Reactor https://projectreactor.io/

1. Evolution of Java

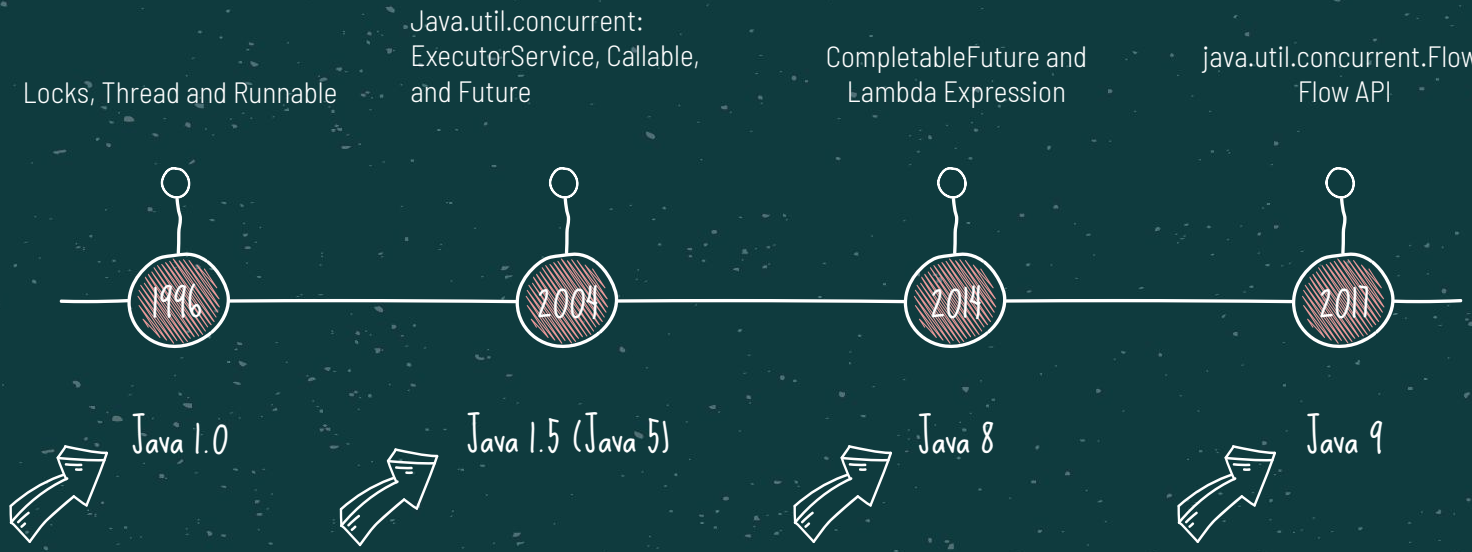
How did we improve utilization?

Java Over the Past 20 Years

Java has evolved considerably in its support for concurrent programming, largely reflecting the changes in hardware, software systems, and programming concepts over the past 20 years.

- Locks、Thread
- ExecutorService
- Lambda Expression (functional programming)
- Reactive programming

Timeline of the Evolution



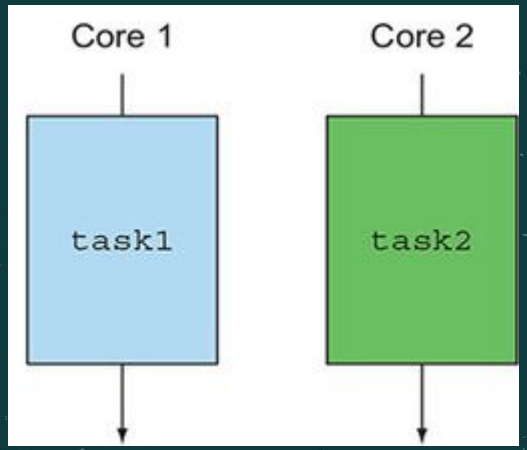
2. Blocking Can be Wasteful

But are we saved by asynchronous?

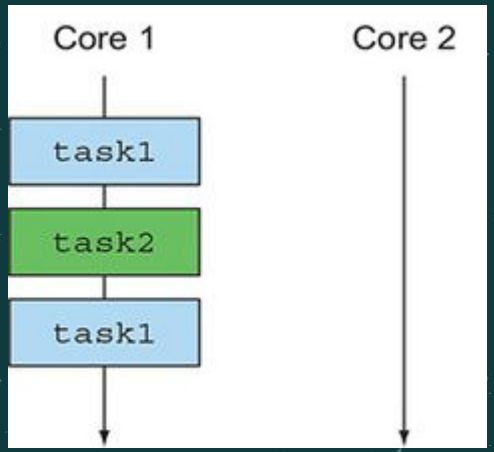
Parallelism and Concurrency

There are, broadly, two ways one can improve a program's performance:

Parallelism



Concurrency



Modern Java in Action https://learning.oreilly.com/library/view/modern-java-in/9781617293566/kindle_split_029.html

Blocking Can Be Wasteful

- Usually, Java developers write programs by using blocking code.
- This practice is fine until there is a performance bottleneck. Then it is time to introduce additional threads, running similar blocking code.
    - Introduce contention and concurrency problems.
    - blocking wastes resources.
    - The parallelization approach is not a silver bullet.

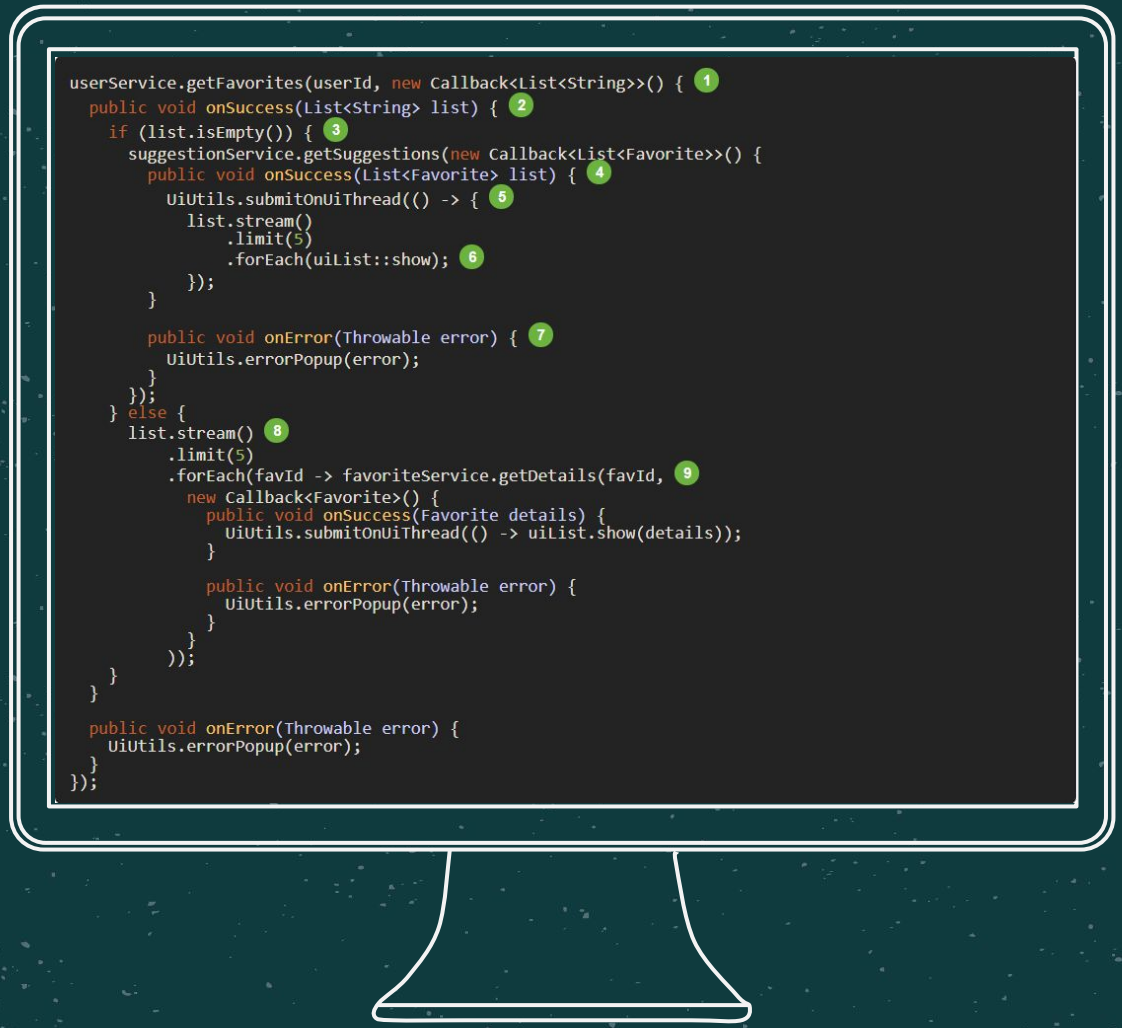
Asynchronicity to the Rescue?

- By writing asynchronous, non-blocking code, you let the execution switch to another active task that uses the same underlying resources and later comes back to the current process when the asynchronous processing has finished.
  - Callbacks
  - Future

Callbacks are hard to compose together, quickly leading to code that is difficult to read and maintain (known as “Callback Hell”).

Example

Showing the top five favorites from a user on the UI or suggestions if she does not have a favorite.



```
userService.getFavorites(userId, new Callback<List<String>>() { 1
  public void onSuccess(List<String> list) { 2
    if (list.isEmpty()) { 3
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { 4
          UiUtils.submitOnUiThread(() -> { 5
            list.stream()
                .limit(5)
                .forEach(uiList::show); 6
            });
        }

        public void onError(Throwable error) { 7
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() 8
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, 9
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

EVENT LOOP

- An **asynchronous** and **nonblocking** way is essential.
- The reactive frameworks and libraries share threads (relatively expensive and scarce resources) among lighter constructs.
- These techniques not only have the benefit of being cheaper than threads, but also have a major advantage from developers' point of view: they raise the level of abstraction of implementing concurrent and asynchronous applications.
- Most reactive frameworks (such as RxJava and Akka) allow blocking operations to be executed by means of a separate dedicated thread pool. All the threads in the main pool are free to run uninterruptedly, keeping all the cores of the CPU at the highest possible use rate.

3. Reactive

Reactive system and reactive programming

Reactive Manifesto

- Responsive
- Resilient
- Elastic
- Event-driven

Reactive Manifesto

04

The system responds in a timely manner if at all possible.

The system stays responsive under varying workload.

02

03

The system stays responsive in the face of failure.

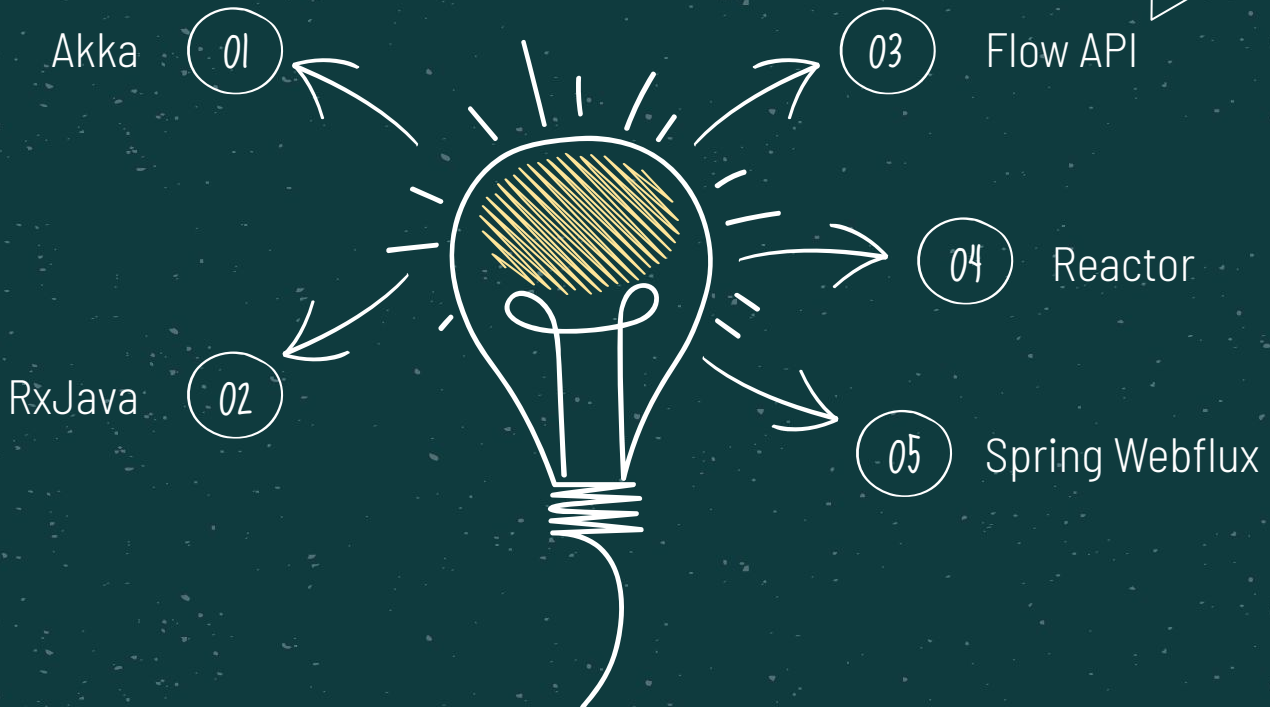
Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

01


Reactive Libraries
~~AWS~~ JAVA
Lambda?
Functional Programming?

Reactive Libraries

Standard in Java 9



Comparison

	RxJava	Reactor
API		
Type-safety		
Checked exceptions		
Testing		
Debugging		
 Spring support		
Android development		
Maturity		

https://www.nurkiewicz.com/2019/02/rxjava-vs-reactor.html

Publisher in Reactor

Flux

A Flux<T> is a standard Publisher<T> that represents an asynchronous sequence of **0 to N** emitted items, optionally terminated by either a completion signal or an error.

Mono

A Mono<T> is a specialized Publisher<T> that emits at most **one** item and then (optionally) terminates with an onComplete signal or an onError signal.

BASIC EXAMPLES

```
String[] colors = {"red", "blue", "green"};

List<String> newColors =
    Flux.fromArray(colors)
        .map(color -> color.toUpperCase())
        .collectList()
        .block();
log.info("Colors: {}", newColors);
```

Create a Flux from a string array

Collect as list (Flux to Mono)

Wait for result (to synchronous call)

Mono<List<String>> reactor.core.publisher.Flux.collectList()

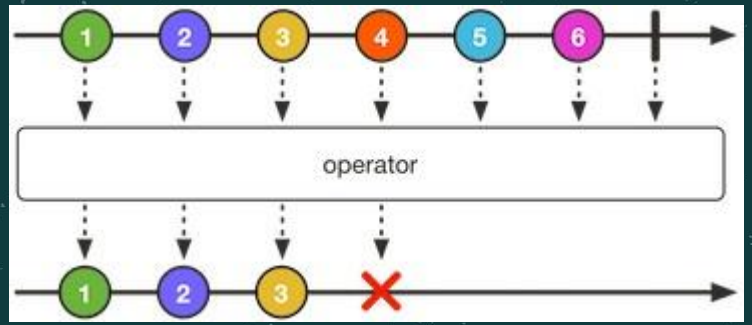
Collect all elements emitted by this `Flux` into a `List` that is emitted by the resulting `Mono` when this sequence completes.

Colors: [RED, BLUE, GREEN]

Publisher in Reactor

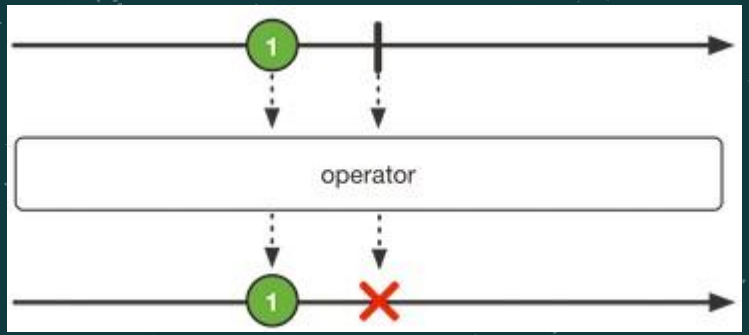
1. onNext()
2. onNext()
3. ....
4. onComplete() or onError()

Flux



1. onNext()
2. onComplete() or onError()

Mono

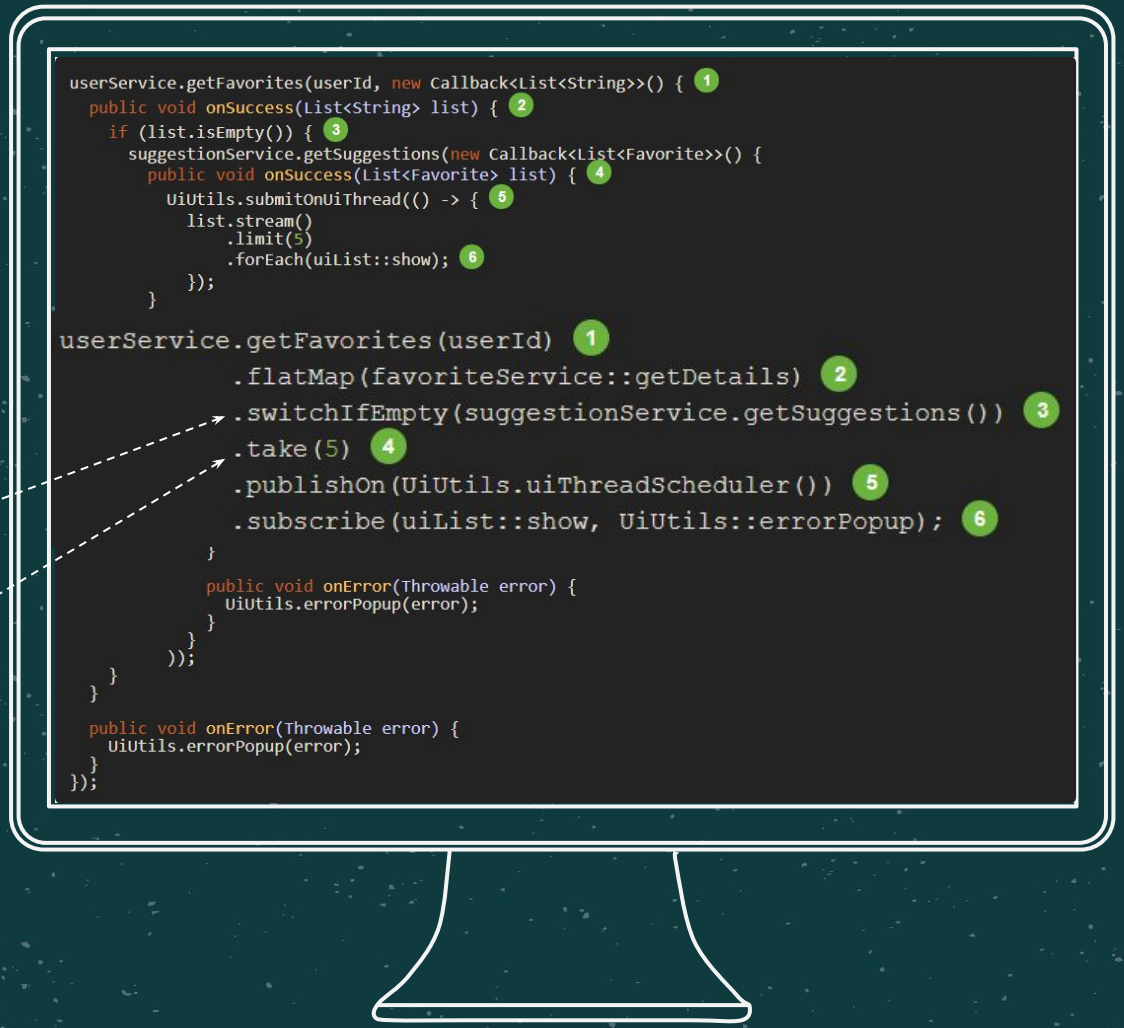


Example

Showing the top five favorites from a user on the UI or suggestions if she does not have a favorite.

Suggestion if she does not have a favorite.

Showing the top five.



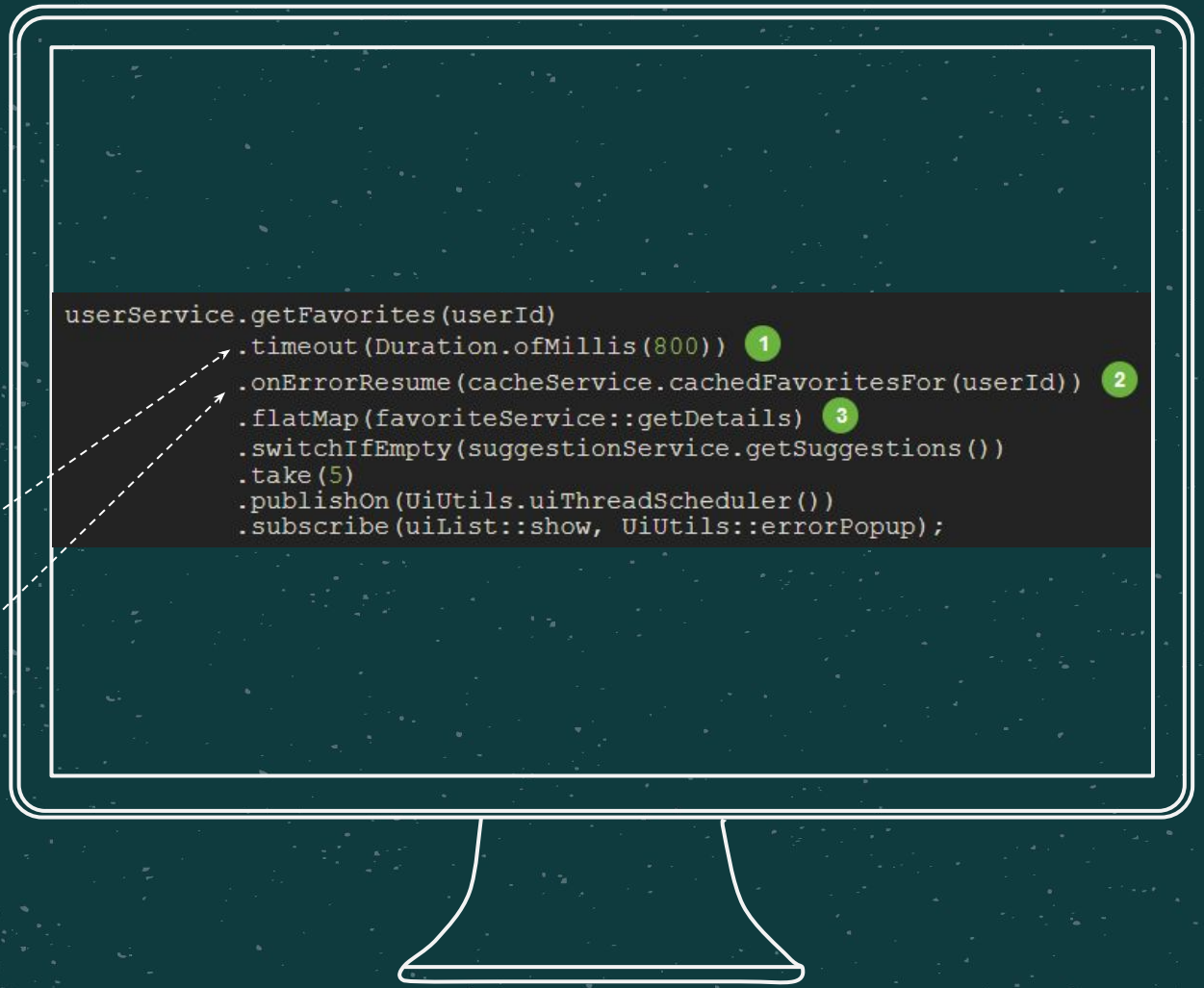
```
userService.getFavorites(userId, new Callback<List<String>>() { 1
  public void onSuccess(List<String> list) { 2
    if (list.isEmpty()) { 3
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { 4
          UiUtils.submitOnUiThread(() -> { 5
            list.stream()
                .limit(5)
                .forEach(uiList::show); 6
          });
        }

userService.getFavorites(userId) 1
            .flatMap(favoriteService::getDetails) 2
            .switchIfEmpty(suggestionService.getSuggestions()) 3
            .take(5) 4
            .publishOn(UiUtils.uiThreadScheduler()) 5
            .subscribe(uiList::show, UiUtils::errorPopup); 6
        }

        public void onError(Throwable error) {
          UiUtils.errorPopup(error);
        }
      }
    ));
  }
}

public void onError(Throwable error) {
  UiUtils.errorPopup(error);
}
});
```

Example

What if you want to ensure the favorite IDs are retrieved in less than 800ms or, if it takes longer, get them from a cache?

If the part above emits nothing for more than 800ms, propagate an error.

In case of an error, fall back to the cacheService.



Subscriber in Reactor

1. Subscribe and trigger the sequence.
2. Do something with each produced value.
3. Deal with values but also react to an error.
4. Deal with values and errors but also run some code when the sequence successfully completes.
5. Deal with values and errors and successful completion but also do something with the Subscription produced by this subscribe call.

HOW CAN I .... CREATE A NEW SEQUENCE?

- that emits a T, and I already have: just
  - ...from an Optional<T>: Mono#justOrEmpty(Optional<T>)
  - ...from a potentially null T: Mono#justOrEmpty(T)
- that emits a T returned by a method: just as well
  - ...but lazily captured: use Mono#fromSupplier or wrap just inside defer
- that emits several T I can explicitly enumerate: Flux#just(T...)
- that iterates over:
  - an array: Flux#fromArray
  - a collection or iterable: Flux#fromIterable
  - a range of integers: Flux#range
  - a Stream supplied for each Subscription: Flux#fromStream(Supplier<Stream>)
- that emits from various single-valued sources such as:
  - a Supplier<T>: Mono#fromSupplier
  - a task: Mono#fromCallable, Mono#fromRunnable
  - a CompletableFuture<T>: Mono#fromFuture

https://projectreactor.io/docs/core/release/reference/#which.create

HOW CAN I .... TRANSFORMING AN EXISTING SEQUENCE?

- I want to transform existing data:
  - on a 1-to-1 basis (eg. strings to their length): map
    i. ...by just casting it: cast
    ii. ...in order to materialize each source value's index: Flux#index
  - on a 1-to-n basis (eg. strings to their characters): flatMap + use a factory method
- I want to add pre-set elements to an existing sequence:
  - at the start: Flux#startWith(T...)
  - at the end: Flux#concatWith(T...)
- I want to aggregate a Flux: (the Flux# prefix is assumed below)
  - into a List: collectList, collectSortedList
  - into a Map: collectMap, collectMultiMap
  - into an arbitrary container: collect
  - into the size of the sequence: count
- I want to combine publishers...
  - in sequential order: Flux#concat or .concatWith(other)
    - ...but delaying any error until remaining publishers have been emitted: Flux#concatDelayError
    - ...but eagerly subscribing to subsequent publishers: Flux#mergeSequential

https://projectreactor.io/docs/core/release/reference/#which.values

HOW CAN I .... PEEKING INTO A SEQUENCE?

- Without modifying the final sequence, I want to:
  - get notified of / execute additional behavior (sometimes referred to as "side-effects") on:
    - emissions: doOnNext
    - completion: Flux#doOnComplete, Mono#doOnSuccess (includes the result, if any)
    - error termination: doOnError
    - cancellation: doOnCancel
    - "start" of the sequence: doFirst
      - this is tied to Publisher#subscribe(Subscriber)
    - post-subscription : doOnSubscribe
      - as in Subscription acknowledgment after subscribe
      - this is tied to Subscriber#onSubscribe(Subscription)
    - request: doOnRequest
    - completion or error: doOnTerminate (Mono version includes the result, if any)
      - but after it has been propagated downstream: doAfterTerminate
    - any type of signal, represented as a Signal: Flux#doOnEach
    - any terminating condition (complete, error, cancel): doFinally

https://projectreactor.io/docs/core/release/reference/#which.peeking

Nothing happens until you subscribe

```
userService.getFavorites(userId) 1
           .flatMap(favoriteService::getDetails) 2
           .switchIfEmpty(suggestionService.getSuggestions()) 3
           .take(5) 4
           .publishOn(UiUtils.uiThreadScheduler()) 5
           .subscribe(uiList::show, UiUtils::errorPopup); 6
```

THREADING

What will be the result?

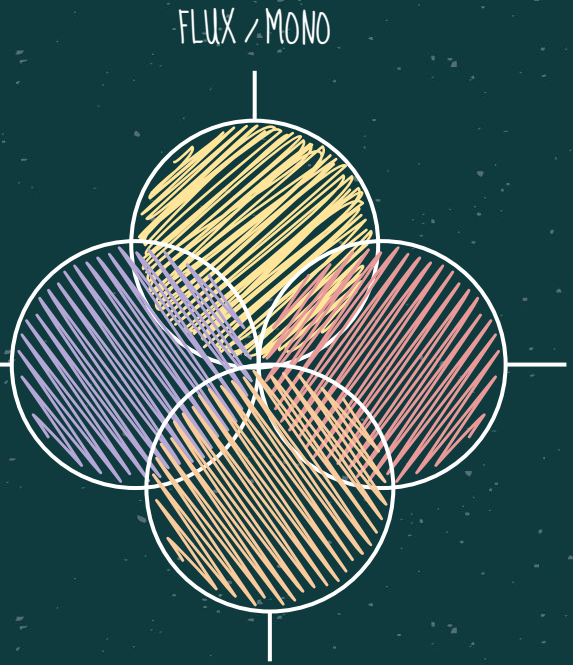
```java
@Slf4j
@Component
public class BasicMonoRunner implements CommandLineRunner {
  @Override
  public void run(String... args) throws Exception {
    String[] colors = {"red", "blue", "green"};

    Mono<List<String>> mono = Flux.fromArray(colors)
      .map(color -> color.toUpperCase())
      .collectList();

    mono.subscribe(bColors -> log.info("Colors #1: {}", bColors));
    mono.subscribe(bColors -> log.info("Colors #2: {}", bColors));
    mono.subscribe(bColors -> log.info("Colors #3: {}", bColors));
    mono.subscribe(bColors -> log.info("Colors #4: {}", bColors));
    mono.subscribe(bColors -> log.info("Colors #5: {}", bColors));
  }
}
```

```
01:50:10.904 [main] INFO  t.j.r.BasicMonoRunner - Colors #1: [RED, BLUE, GREEN]
01:50:10.905 [main] INFO  t.j.r.BasicMonoRunner - Colors #2: [RED, BLUE, GREEN]
01:50:10.906 [main] INFO  t.j.r.BasicMonoRunner - Colors #3: [RED, BLUE, GREEN]
01:50:10.906 [main] INFO  t.j.r.BasicMonoRunner - Colors #4: [RED, BLUE, GREEN]
01:50:10.906 [main] INFO  t.j.r.BasicMonoRunner - Colors #5: [RED, BLUE, GREEN]
```

Reactor, like RxJava, can be considered to be **concurrency-agnostic**.

- Most operators continue working in the Thread on which the previous operator executed.
- Unless specified, the topmost operator (the source) itself runs on the Thread in which the subscribe() call was made.
  - Scheduler
  - publishOn()
  - subscribeOn()

WHAT'S NEXT?



https://projectreactor.io/docs/core/release/reference/#error.handling

https://projectreactor.io/docs/core/release/reference/#debugging

https://projectreactor.io/docs/core/release/reference/#testing

IMMUTABILITY MATTERS

AVOID NULL

TRY IT YOURSELF

√123

STUDY HARD!

+ × ÷

THANK YOU!

https://projectreactor.io/docs/core/release/reference