IT5100B

Industry Readiness
Stream Processing

LECTURE 3

Reactive Programming

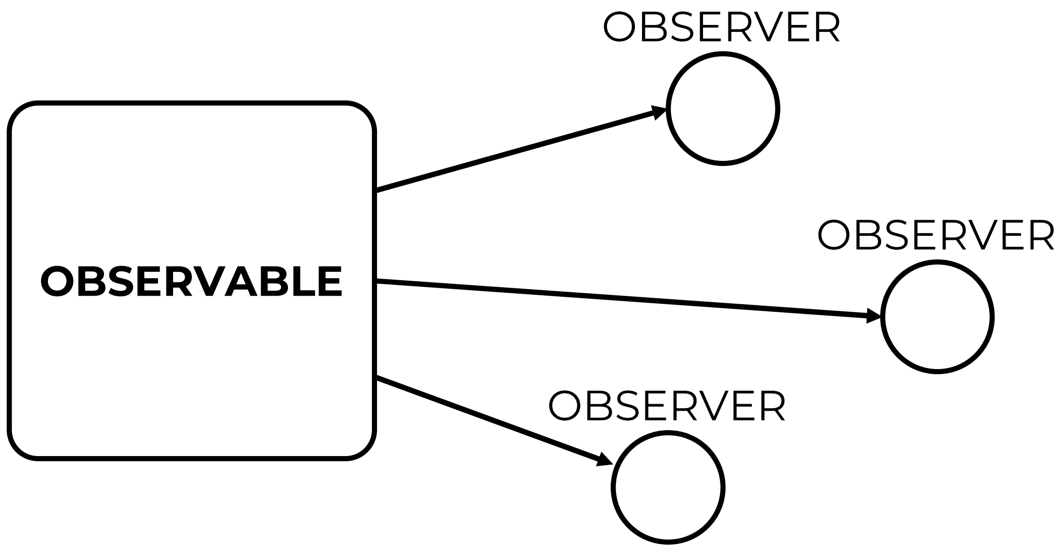
FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

- What and Why Reactive Programming
- Project Reactor
- Flux/Mono Basics
- Additional Reactive Topics
- Reactive Programming Tips

WHAT/WHY REACTIVE PROGRAMMING

REACTIVE PROGRAMMING



Declarative programming paradigm concerned with **data streams** and **propagation of change**

SMALL RESTAURANT

```
def chicken_chop():
    chicken = grill_chicken()
    fries = fry_fries()
    return chicken + fries
```

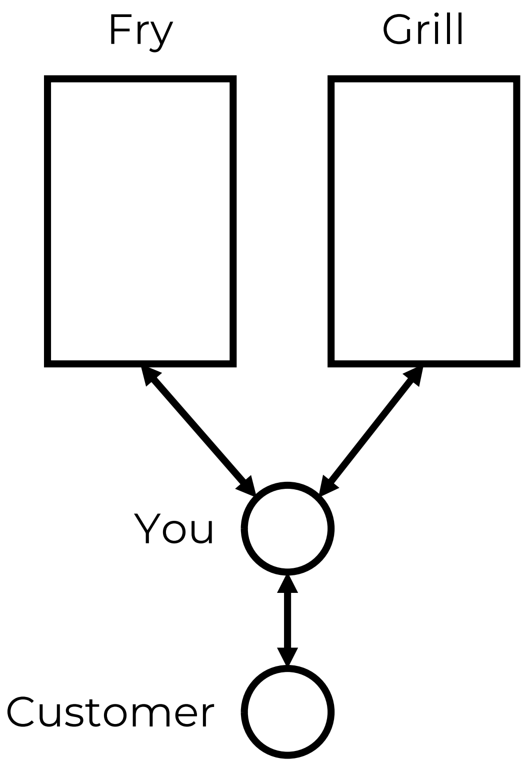
Image source: Shutterstock

You are the chef taking and making orders

Suppose there is only one customer:

1. Receive the order
2. Grill the chicken
3. Fry the fries
4. Put chicken and fries together
5. Deliver order

MEDIUM RESTAURANT

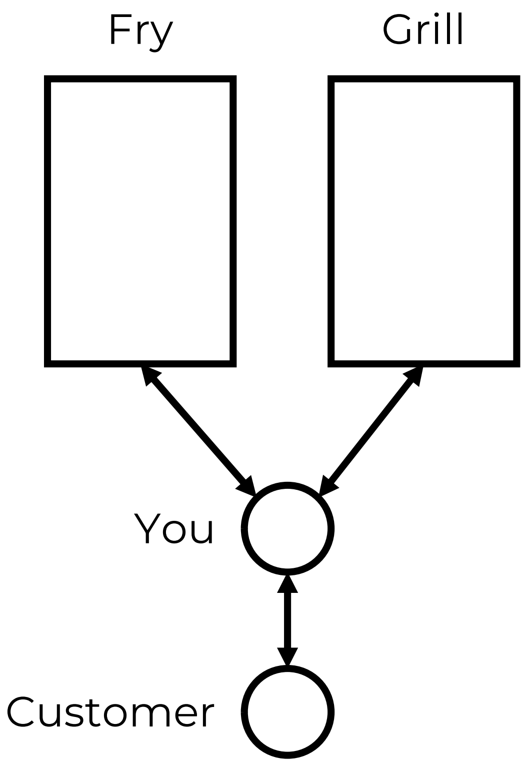


Suppose now you're the **head chef** and you have many chicken grillers and fries fryers

Suppose there is only one customer:

1. Receive the order
2. Tell the griller to grill chicken
3. Once you have the chicken, tell fryer to fry fries
4. Once you have the fries, put them together and deliver

MEDIUM RESTAURANT

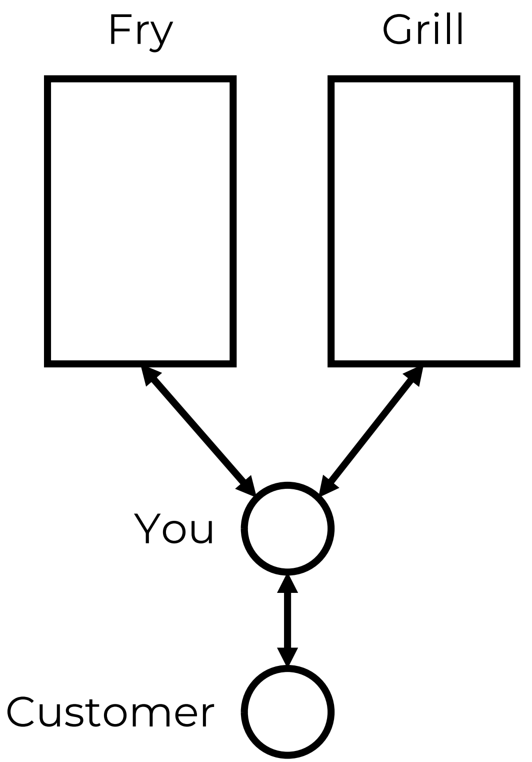


Suppose now you're the **head chef** and you have many chicken grillers and fries fryers

Suppose there is only one customer:

1. Receive the order
2. Tell the griller to grill chicken
3. Wait for chicken to come
4. Tell fryer to fry fries
5. Wait for fries to come
6. Put them together and deliver

MEDIUM RESTAURANT

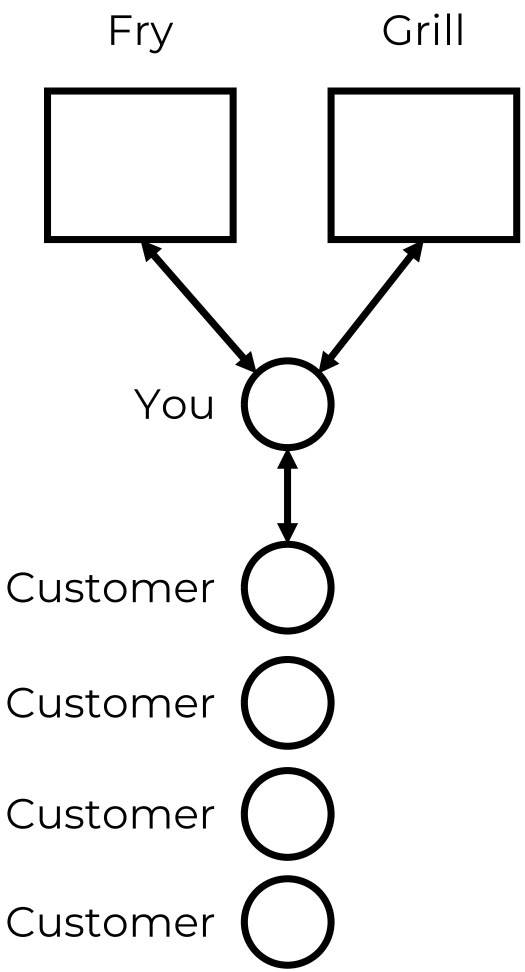


Suppose now you're the **head chef** and you have many chicken grillers and fries fryers

Suppose there is only one customer:

1. Receive the order
2. Tell the griller to grill chicken
3. Tell fryer to fry fries
4. Wait for both to come
5. Put them together and deliver

BIG RESTAURANT

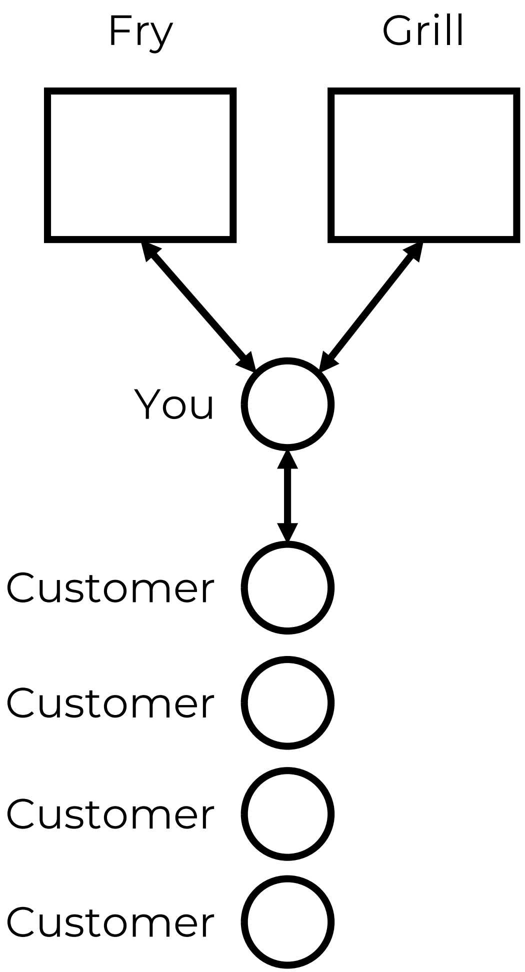


Suppose now business is good and you have many customers:

For each customer:

1. Receive the order
2. Tell the griller to grill chicken
3. Tell fryer to fry fries
4. Wait for both to come
5. Put them together and deliver

BIG RESTAURANT



Suppose now business is good and you have many customers:

For each customer:

1. Receive the order
2. Tell the griller to grill chicken
3. Tell fryer to fry fries
4. When both comes:
    1. Put them together and deliver

NON-BLOCKING

Life is all about **waiting**:

- Waiting for messages to come in
- Waiting for HTTP request to send response
- Waiting for the girl you’re seeing to text you back

Instead of doing nothing while waiting, carry on with your life

When things come in, **react** to them

NON-BLOCKING BEHAVIOUR

How do we introduce non-blocking behaviour into programs?

1. Create new classes that have a `whenSomethingComes(doSomething)` method
2. Re-write programs using this behavior
3. Write other features like schedulers to ensure performance is optimized, error handling, etc

Can we expect developers to do this?

CALLBACKS

How to give your chef a chicken:

1. Look in refrigerator
2. If there are chickens, give them one
3. Otherwise, get one from the store and give it to them

Callback hell!

```java
refrigerator.getChickens(new Callback<List<Chicken>>() {
  public void onSuccess(List<Chicken> ls) {
    if (ls.isEmpty()) {
      store.buyChicken(new Callback<Chicken>() {
        public void onSuccess(Chicken c) {
          chef.give(c);
        }
        public void onError(Throwable t) {
          chef.noMoreChickens();
        }
      });
    } else {
      chef.give(ls.get(0));
    }
  }
  public void onError(Throwable t) {
    store.buyChicken(new Callback<Chicken>() {
      public void onSuccess(Chicken c) {
        chef.give(c);
      }
      public void onError(Throwable t) {
        chef.noMoreChickens();
      }
    });
  }
});
```

COMPLETABLEFUTURES

How to give your chef a chicken:

1. Look in refrigerator
2. If there are chickens, give them one
3. Otherwise, get one from the store and give it to them

- Slightly better with lambda expressions, still hard to read!
- Clunky error handling!
- What about non-blocking streams?

```
refrigerator.getChickens().handle((ls, exp) -> {
  if (ls != null && ls.isEmpty()) {
    store.buyChicken().handle((chicken, exp) -> {
      if (chicken != null) {
        chef.give(chicken);
      } else {
        chef.noMoreChickens();
      }
    });
  } else if (ls != null) {
    chef.give(ls.get(0));
  } else {
    store.buyChicken().handle((chicken, exp) -> {
      if (chicken != null) {
        chef.give(chicken);
      } else {
        chef.noMoreChickens();
      }
    });
  }
});
```

ITERATOR VS OBSERVER

A PARADIGM SHIFT

Iterable

Iterator

I will **pull** each element from you and then I will do xyz

Observable

Observer

You **will** push each element to me and then I will do xyz

Aside from underlying implementation, the way you will interact with iterables (like lists) and observables (reactive streams) is the same!

ITERATOR VS OBSERVER

A PARADIGM SHIFT

Iterator code

```
keyPressList.filter(x -> x != null)
  .map(Event::withId)
  .forEach(System.out::println);
```

Observer code

```
keyPressObservable.filter(x -> x != null)
  .map(Event::withId)
  .forEach(System.out::println);
```

The code we write for two completely different models should look virtually the same!

NON-BLOCKING REACTIVE PROGRAMMING

```
refrigerator.getChickens()               // get list of chickens
  .next()                                // just take one
  .doOnError(chef::somethingHappened)    // tell chef if error occurred
  .onErrorComplete()                     // recover from error if occurred
  .switchIfEmpty(store.getChicken())     // if nothing from fridge, get from store
  .singleOptional()                      // potentially got nothing from whole process
  .subscribe(oc ->
      oc.ifPresentOrElse(chef::give,     // if chicken received, give to chef
        chef::noMoreChickens),           // if no chickens received, tell chef
    chef::somethingHappened);            // if error occurred at the end, tell chef
```

Much cleaner way to express non-blocking code in familiar style with robust error handling

REACTIVE PROGRAMMING IN THE WILD

CASE STUDY: TOMCAT VS NETTY

	Apache Tomcat	**Netty (Reactive)**
Throughput	2310 req/s	3421 req/s
Memory	521MB	62MB
CPU Usage	~70%	~50%
Threads	220	19

https://medium.com/@skhatri.dev/springboot-performance-testing-various-embedded-web-servers-7d460bbfdb1b

“[Netty can] service **50,000+** messages per second from approximately **30,000 connected clients** on a **commodity Intel server** costing approximately $850”

- Matt Rhodes, RhoMoSoft, February **2009**

Maximizing thread usage by non-blocking improves performance and allows applications to handle more concurrent requests

KEY POINT #1

Reactive programming frameworks allow us to write **non-blocking** programs in a familiar **declarative** style

PROJECT REACTOR

REACTIVE PROGRAMMING LIBRARIES

PROJECT **REACTOR**

Fully non-blocking I/O with **functional API** over **typed [0|1|N] sequences**

Works nicely with popular frameworks, drivers and protocols:

- Spring Boot and WebFlux
- CloudFoundry Java Client
- Reactive Relational Database Connectivity (R2DBC)

Project Reactor is an alternative to RxJava

PROJECT REACTOR

MAIN CLASSES

`Flux<T>`

0 or N elements

`Mono<T>`

0 or 1 element

Non-reactive equivalents:

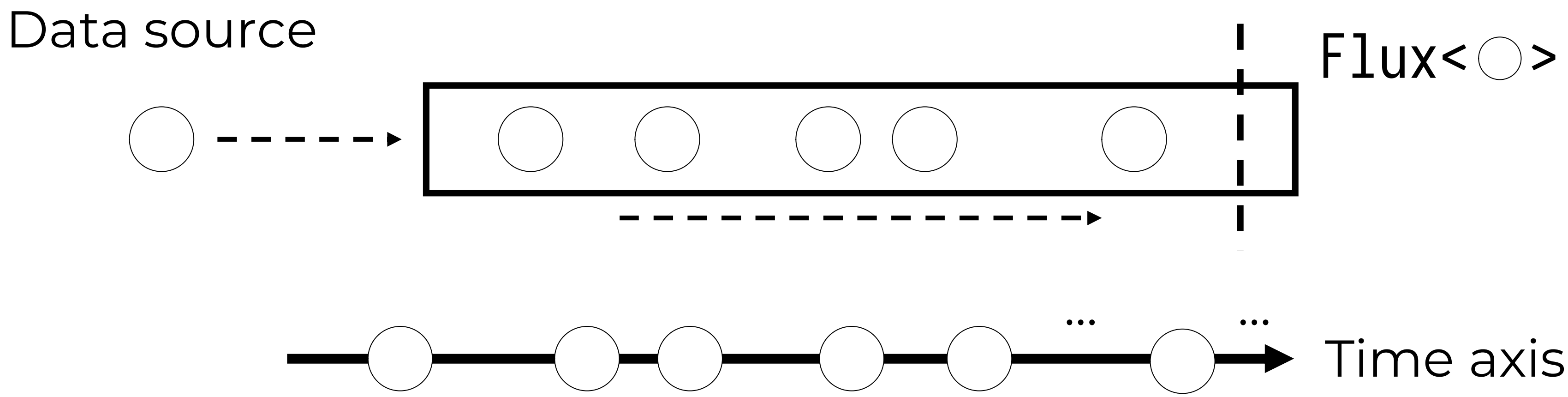
`Stream<T>`

`Optional<T>`

See the Flux and Mono API documentations

ASSEMBLY LINE

MENTAL MODEL OF FLUX AND MONO

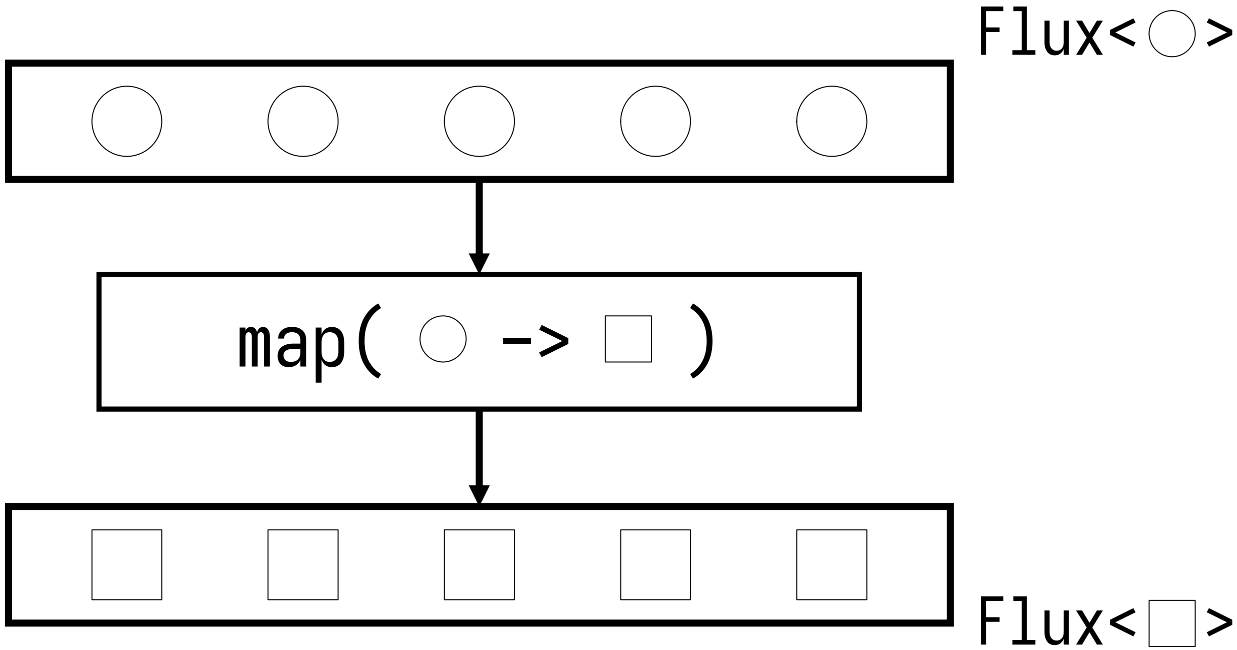


Each Flux or Mono object is a Publisher that publishes new data

Think of this as a conveyor belt carrying items

ASSEMBLY LINE

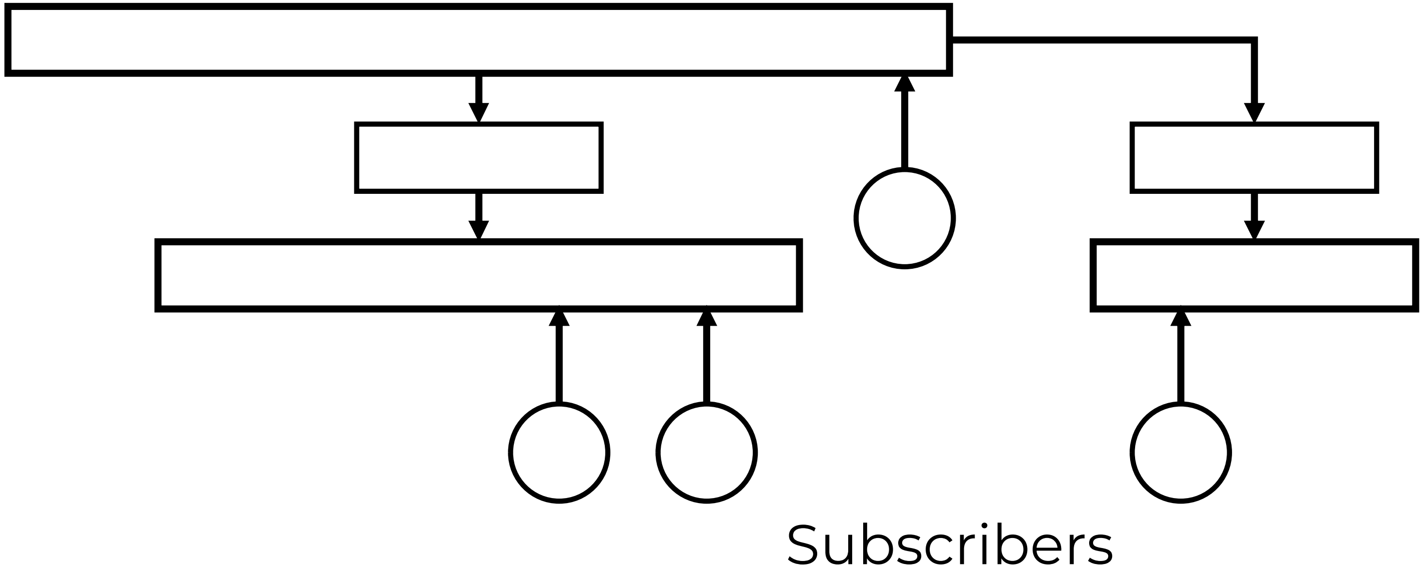
MENTAL MODEL OF FLUX AND MONO



Each operation on a Flux or Mono is like a workstation that creates a new Flux or Mono

ASSEMBLY LINE

MENTAL MODEL OF FLUX AND MONO

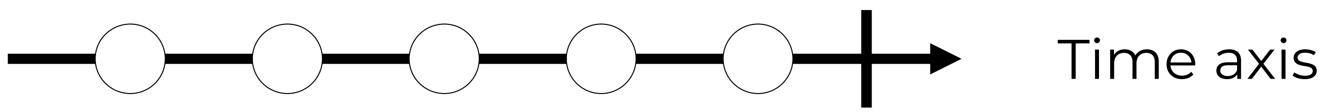


Each Flux or Mono can be **subscribed** to, which produces subscriptions known as Disposables

Generally, nothing happens until a subscription occurs (lazy!)

MARBLE DIAGRAMS

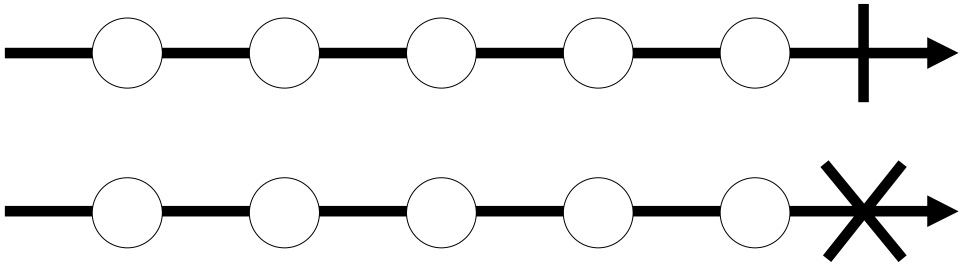
MENTAL MODEL OF FLUX AND MONO



Visual representations of Flux and Mono and signals produced through the flow of time

MARBLE DIAGRAMS

SIGNALS



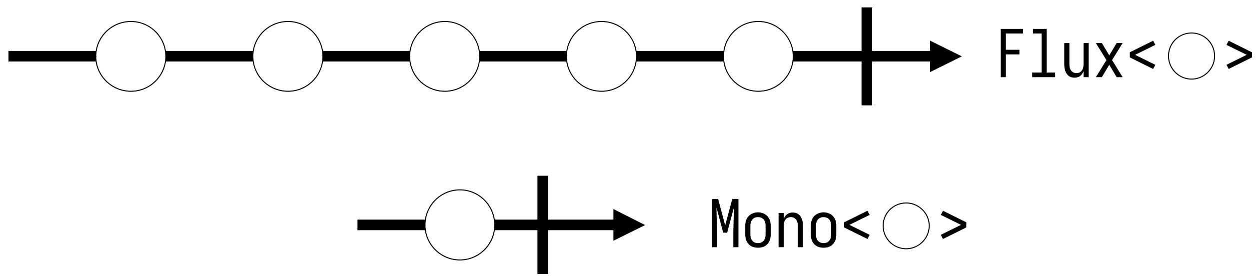
One of three signals can be emitted from a Flux or Mono:

1. An object (next signal)
2. A complete signal (no more elements after)
3. An error signal (no more elements after-ish*)

---

*`Flux#onErrorContinue` can cause upstream publishers to recover from errors but this should be rarely used

FLUX VS MONO

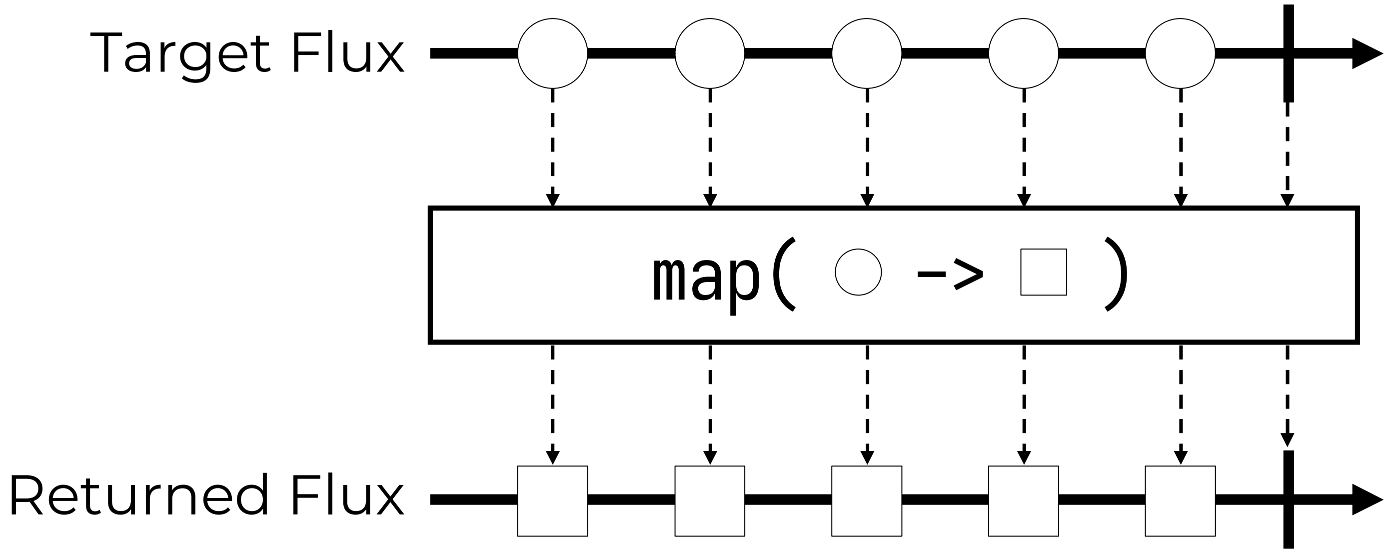


Flux produces 0 or more elements, Mono produces at most one

Both Flux and Mono are Publishers in the Reactive Streams specification

MARBLE DIAGRAMS

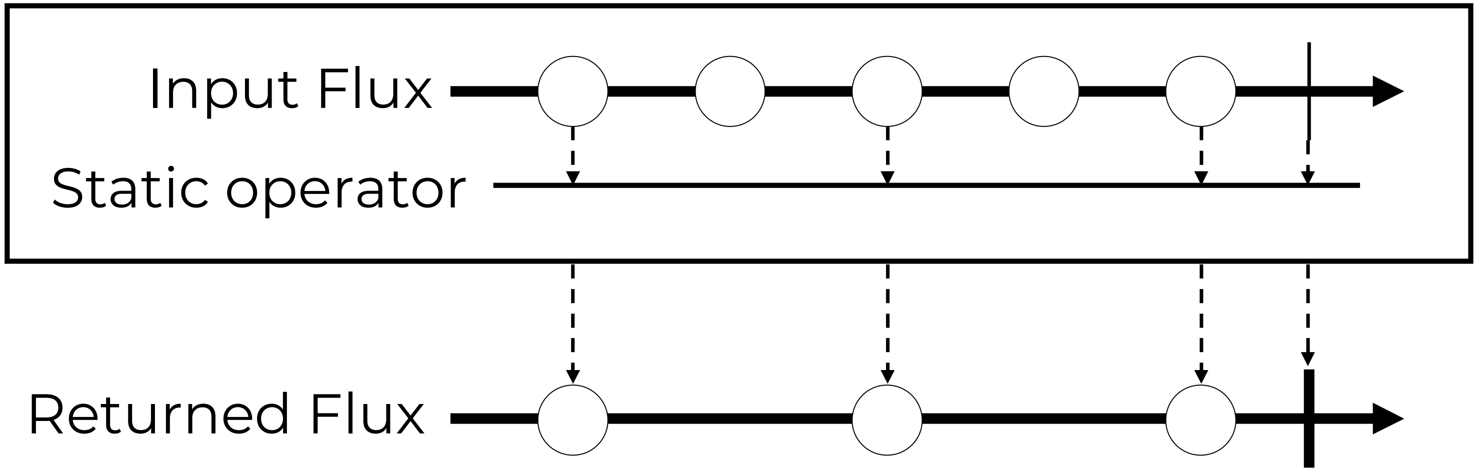
OPERATORS



Instance operators operate on Flux/Mono objects and produce a new Flux/Mono

MARBLE DIAGRAMS

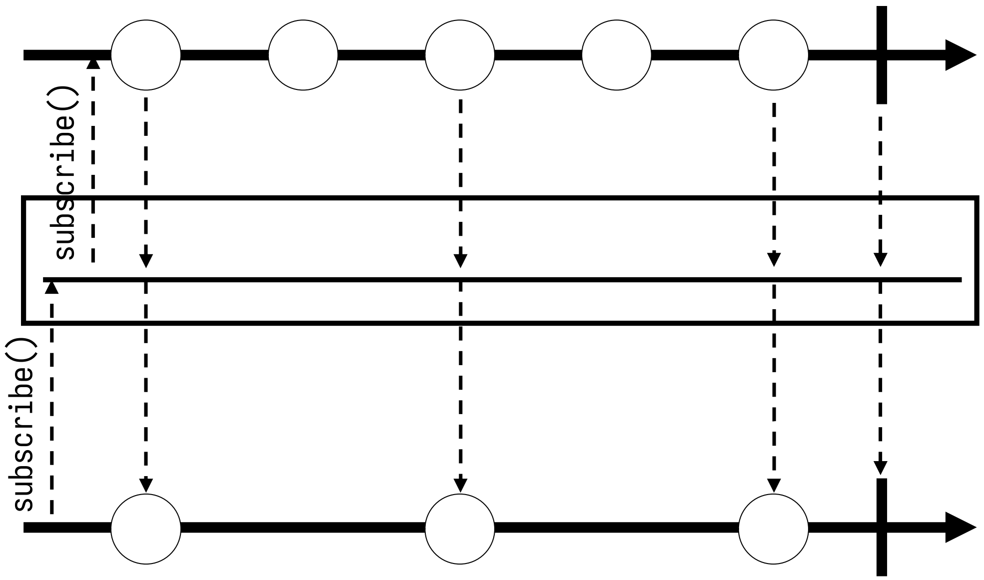
STATIC OPERATORS



Static operators return new Flux/Mono may also receive publishers

MARBLE DIAGRAMS

UPSTREAM SIGNALS

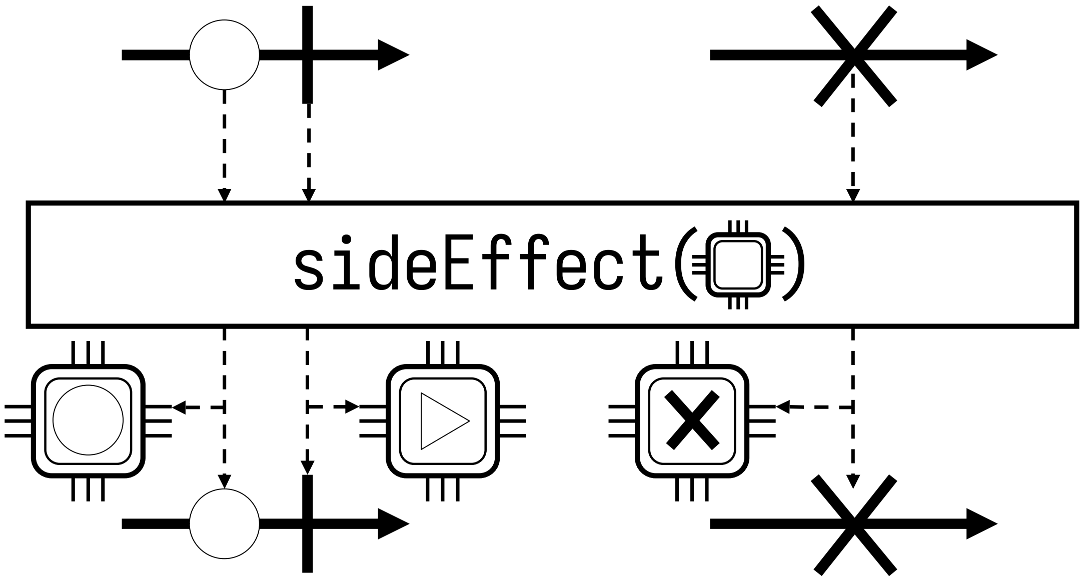


Signals can be sent upstream to earlier publishers

`subscribe` signals to earlier publishers to beginning emitting events

MARBLE DIAGRAMS

SIDE EFFECTS



Side effects can be executed on different signals; some happen before (left) or after (right) the signal is sent

FLUX/MONO BASICS

PUBLISHER CREATION

JUST/FROM

```
Mono<Integer> m1 = Mono.just(1);
Flux<String> f2 = Flux.just("hello", "world!");
Mono<Object> m3 = Mono.justOrEmpty(null);
```

The `just` series of static methods wrap element(s) in a Flux/Mono

```
Mono<Integer> m4 = Mono.fromSupplier(() -> 1);
Flux<Double> f5 = Flux.fromStream(Stream.of(1, 1.5, 2));
```

The `from` series of static methods create Flux/Mono from different data structures

PUBLISHER CREATION

FROM EXTERNAL SERVICES

```
Flux<User> m = db.sql("SELECT * FROM users")
        .map(User::fromDb)
        .all();
```

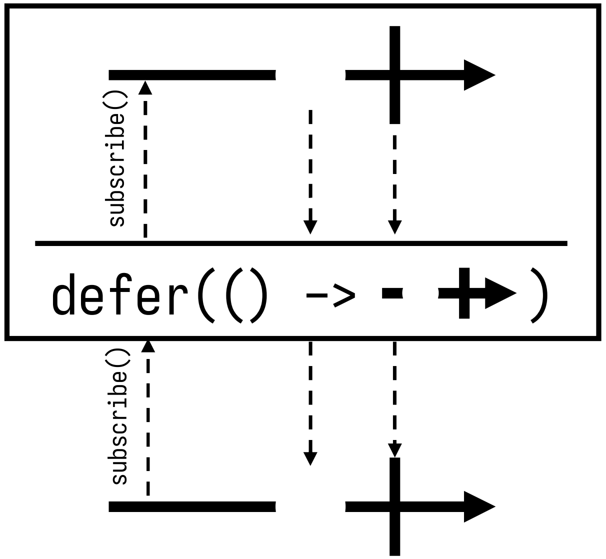
Generally, reactive data sources are not created programmatically:

- R2DBC SQL query to a database gives Flux/Mono
- ReactorKafka gives Flux/Mono on event production to Kafka message broker
- Etc.

PUBLISHER CREATION

DEFER

```
AtomicInteger i = new AtomicInteger(1);
Mono<Integer> m = Mono.defer(() -> {
    int x = i.getAndIncrement();
    System.out.println("Mono #" + x);
    return Mono.just(x);
});
```



Lazily supply a Publisher every time a subscription is made; each subscriber will get a subscriber-specific instance

PUBLISHER CREATION

GENERATE

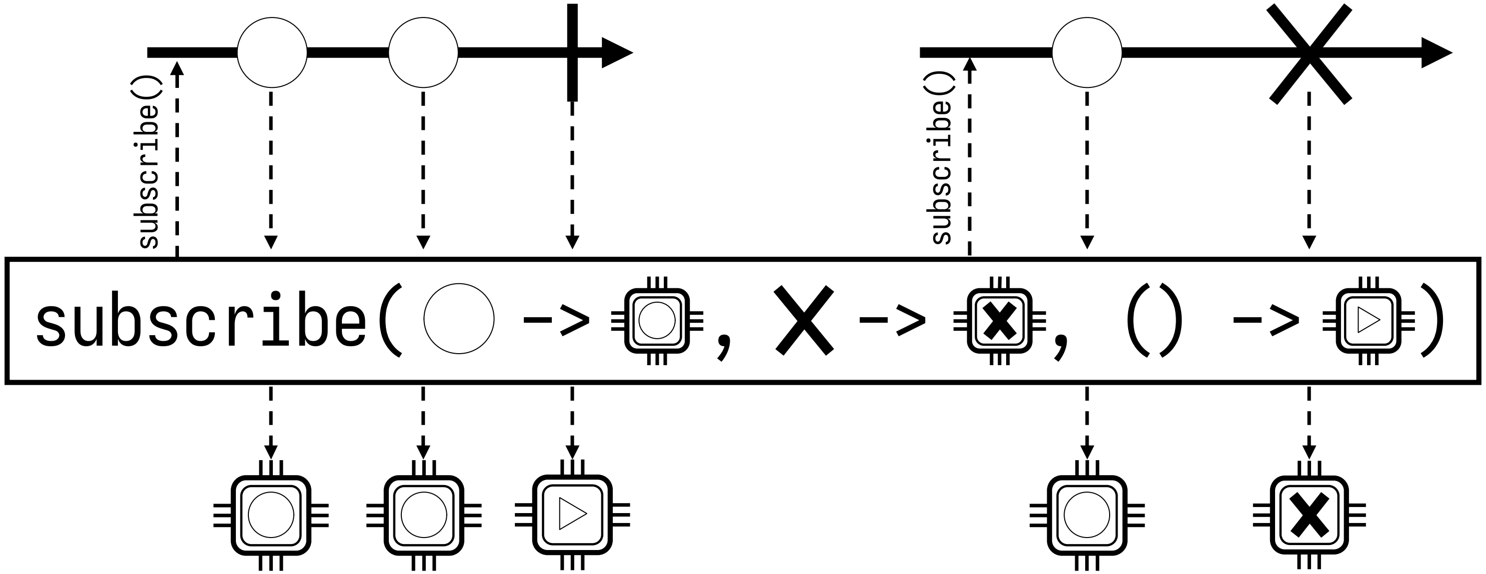
```java
Flux<Integer> f = Flux.generate(() -> 1, (i, s) -> {
    s.next(i); // push next element into sink
    if (i == 10) s.complete(); // send complete signal
    return i + 1; // return next state
});
```

Stateful synchronous generation of elements (provides a lot of fine-grain control)

Similar to `Stream#iterate`

SUBSCRIPTION

SUBSCRIBE VARIANTS



Triggers the publisher to start emitting signals; each signal can be handled by a different consumer; `subscribe` returns a `Disposable` which can be `disposed` (cancels the subscription)

PUBLISHER TRANSFORMS

MAP AND FILTER

```
Flux.range(0, 10)
        .map(x -> x * 2)
        .filter(x -> x % 3 == 0)
        .subscribe(System.out::println);
```

Simple transforms can be done with `map` and `filter` as per usual

EXERCISE

FLUX CREATION

Write a static method `cycle` that receives a `List` and produces a `Flux` that cycles over the elements repeatedly (potentially unbounded). Use `generate` (and `map` too).

```
Flux<String> f = cycle(List.of("a", "b", "c"));
f.take(5).subscribe(System.out::println); // a, b, c, a, b
```

SOLUTION

CYCLE

```
public static <T> Flux<T> cycle(List<? extends T> ls) {
    if (ls.isEmpty()) return Flux.empty();
    return Flux.<Integer, Integer>generate(
                () -> 0, // index of first item
                (i, s) -> {
                    s.next(i); // push index
                    return (i + 1) % ls.size(); // next index
                })
            .map(ls::get); // map indices to elements
}
```

PUBLISHER TRANSFORMS

FLATMAP

`flatMap` works as you would expect, except with (mainly) three variants:

	`flatMap`	`flatMapSequential`	`concatMap`
Subscription of inner publishers	Eager	Eager	Only after previous inner fluxes have completed
Ordering of signals	Next signal is first emission from any inner flux	Ordered like concatenation	Ordered (concatenation)
Interleaving	Yes	No	No (concatenation)

PUBLISHER TRANSFORMS

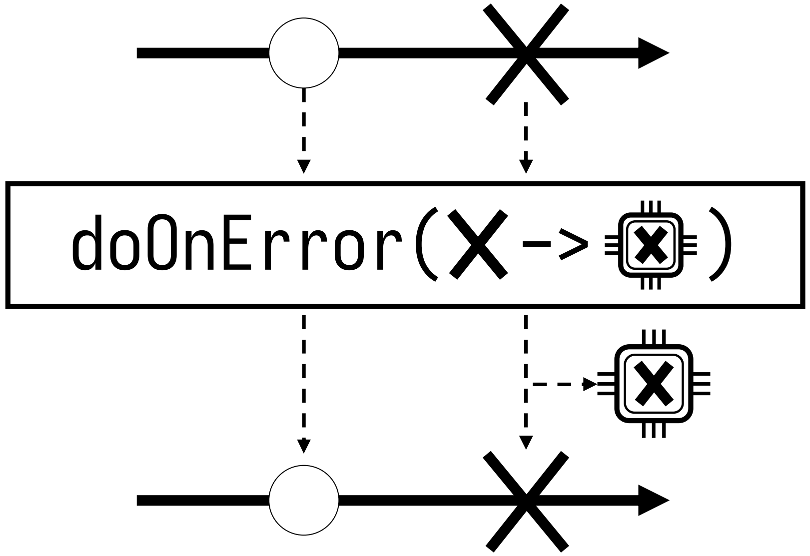
GENERAL TRANSFORMS

```
Function<Flux<Integer>, Flux<String>> f = s -> s.map(x -> x * 2)
        .filter(x -> x % 3 == 0)
        .map(x -> x + " haha");

Flux.range(1, 10)
        .transform(f)
        .subscribe(System.out::println);
}
```

For convenience and readability, you may use the general `transform` method that performs a bunch of `transforms` on a publisher; there are `transform` variants you may find useful too

HOOKS



The `do` series of methods add side-effects to signals

EXERCISE

TRANSFORMS AS “AND THEN” OPERATIONS

```
public static Mono<Integer> getFromDb(int i) {
    return Mono.just(i)
            .delayElement(Duration.ofSeconds(1))
            .doOnNext(x -> System.out.println("retrieved " + x));
}
public static Mono<Integer> putInDb(int i) {
    return Mono.just(i)
            .delayElement(Duration.ofSeconds(1))
            .doOnNext(x -> System.out.println("put "+ x));
}
```

Write a static method `increment` that receives an integer i, then gets it from the database, increases it by one, and puts it back in the database

SOLUTION

TRANSFORMS AS “AND THEN” OPERATIONS

```
public static Mono<Integer> increment(int i) {
    return getFromDb(i).map(x -> x + 1)
            .flatMap(Main::putInDb);
}
```

MERGES

There are many ways to merge publishers together:

- `concat`: concatenate publishers
- `merge`
- `zip`: combine n-wise to n-tuples
- `and/when`: coordinate termination
- `firstWith...`: Choose first emitter
- `...ifEmpty`: Reactive if-else
- `then`: Chain publishers

ERROR HANDLING

There are many error handling operations:

- `onError...`: perform some transform when error signal received
- `doOnError`: perform side-effect on error signal
- `retry`: re-subscribe to the publisher if given error signal
- `subscribe`: subscribe with custom error handler

EXERCISE

MERGES AND ERROR HANDLING

```
public static Mono<Integer> getRandInt() {
    return Mono.just(rng.nextInt(10))
            .doOnNext(x -> System.out.println("got " + x));
}
```

Define a method divide that gets two integers a and b reactively and returns a / b reactively. If there is a ArithmeticException, print an error message and complete with no elements

SOLUTION

MERGES AND ERROR HANDLING

```
public static Mono<Integer> divide() {
    return getRandInt().zipWith(getRandInt())
            .map(x -> x.getT1() / x.getT2())
            .doOnError(ArithmeticException.class, System.out::println)
            .onErrorComplete(ArithmeticException.class);
}
```

ORIGINAL EXAMPLE WALKTHROUGH

```
refrigerator.getChickens()                  // get list of chickens
  .next()                                   // just take one
  .doOnError(chef::somethingHappened)       // tell chef if error occurred
  .onErrorComplete()                        // recover from error if occurred
  .switchIfEmpty(store.getChicken())        // if nothing from fridge, get from store
  .singleOptional()                         // potentially got nothing from whole process
  .subscribe(oc ->
      oc.ifPresentOrElse(chef::give,        // if chicken received, give to chef
        chef::noMoreChickens),              // if no chickens received, tell chef
    chef::somethingHappened);               // if error occurred at the end, tell chef
```

ADDITIONAL REACTIVE TOPICS

CONTROLLING BACKPRESSURE

PUSH/PULL MODEL

```
myFlux().subscribe(new BaseSubscriber<>() {
    protected void hookOnNext(Integer value) {
        System.out.println(value);
        request(1);
    }
    protected void hookOnComplete() {
        System.out.println("no more!");
    }
});
```

Create custom subscribers that pull available data on demand, controlling backpressure

HOT/COLD PUBLISHERS

So far most publishers we have seen are cold:

Cold publisher—nothing happens until subscribe, each subscribe triggers publication (most assembly operations)

Hot publisher—publish data eagerly, when new subscribers only see signals after subscription (unless cached) (`just`)

THREADING AND SCHEDULERS

HANDLING CONCURRENCY AND SCHEDULERS

```
return Mono.fromSupplier(() -> {
    try { Thread.sleep(1000); } catch (Exception e) { }
    return 1;
}).publishOn(Schedulers.boundedElastic());
```

Reactor is **concurrency agnostic**, you get to choose how threading is done

MANY MORE!

Many more topics on reactive programming with Reactor can be found on their amazing reference documentation

REACTIVE PROGRAMMING TIPS

REACTIVE ALL THE WAY!

In `Streams`/`Optionals`, eventually we will `reduce`/`collect`/`get`.
When do we `block` to retrieve elements from publishers?

Ideally, **never**!
Let the framework decide when to `block`; if you `block`, you wait!

DON'T BREAK THE CHAIN

```java
public static Mono<Mono<Void>> something() {
    return Mono.just(1)
            .map(x -> doStuff());
}
public static Mono<Void> doStuff() {
    return Mono.just(1)
            .doOnNext(x -> System.out.println("stuff"))
            .then();
}
```

```java
public static Mono<Void> something() {
    return Mono.just(1)
            .flatMap(x -> doStuff());
}
public static Mono<Void> doStuff() {
    return Mono.just(1)
            .doOnNext(x -> System.out.println("stuff"))
            .then();
}
```

Cold publishers are never triggered until subscribed; subscribing to a publisher will not trigger subscription unconnected ones

KEY POINT #2

Stay reactive!

CONTENTS

- What and Why Reactive Programming
- Project Reactor
- Flux/Mono Basics
- Additional Reactive Topics
- Reactive Programming Tips

KEY POINTS

- **Reactive programming** frameworks allow us to write **non-blocking** programs in a familiar **declarative** style
- Stay reactive!