

Java & Spring Reactive Programming

Rony Keren

Introduction

Reactive Programming

“is a declarative programming paradigm concerned with data streams and the propagation of change” (wiki)

Introduction

What is the problem with current programming?

- In order to understand it clearly – let's take a look at how traditional web servers works today:

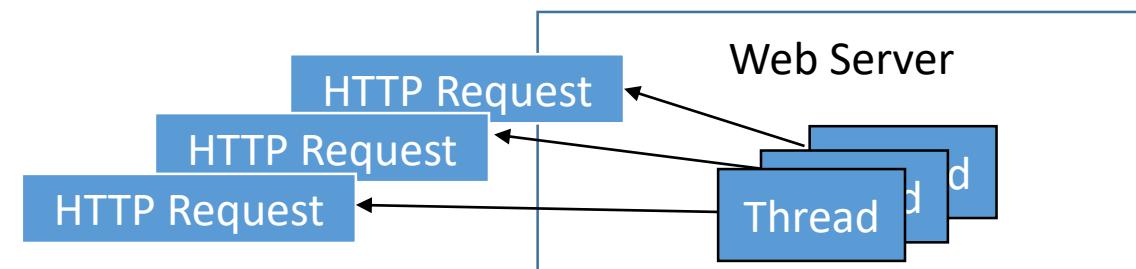
- Based on request-response mechanism
 - Responses are packed at-once
 - Server architecture is ‘Request-based’



Introduction

Request-Based causes back pressure

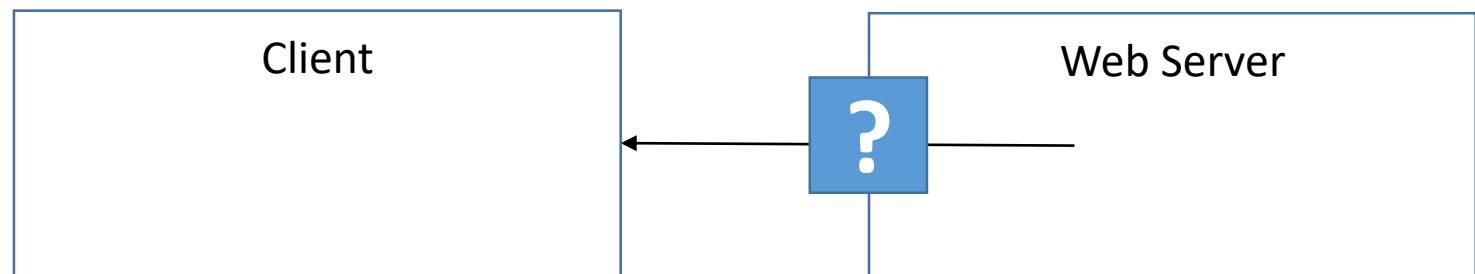
- Threads are consumed according to requests – ‘Request Oriented’ mechanism
- Each thread is usually dedicated to the request until completed
- This can cause lot of requests wait for hosting threads
- In high and intense concurrencies clients end up with timeouts
- Client capability of processing responses in not counted



Introduction

Response at-once makes data streaming hard

- Clients may have to handle lots of data
 - No one asks the client how much ‘response’ it can handle....
 - No way for clients to specify it in a built-in manner
- HTTP 1 tricks for streaming
 - Client refresh (intensive requests, repeatable data)
 - Response flushing (suffers from timeouts)
 - Web-sockets (performs badly)



Introduction

The Reactor Design Pattern

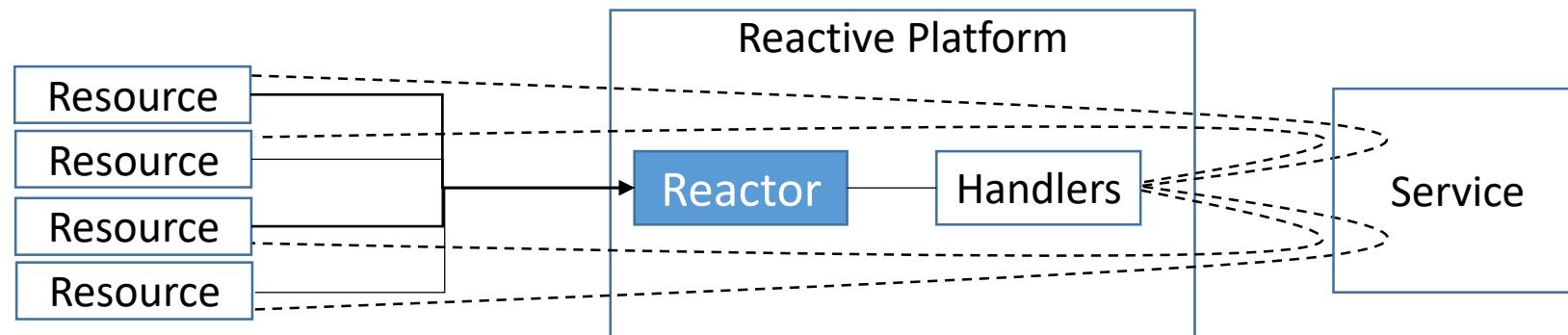
An implementation technique of event driven model

- Mostly, uses single dispatcher thread event-loop blocking on resources I/O requests (clients)
- Resources emits events which are sequentially dispatched by the thread to handlers
- Events might be:
 - new connection
 - ready for read
 - ready for write
 - timeout
 - disconnecting
- Handlers (or callbacks) performs the actual work in a non-blocking manner
 - doesn't block the dispatcher thread

Introduction

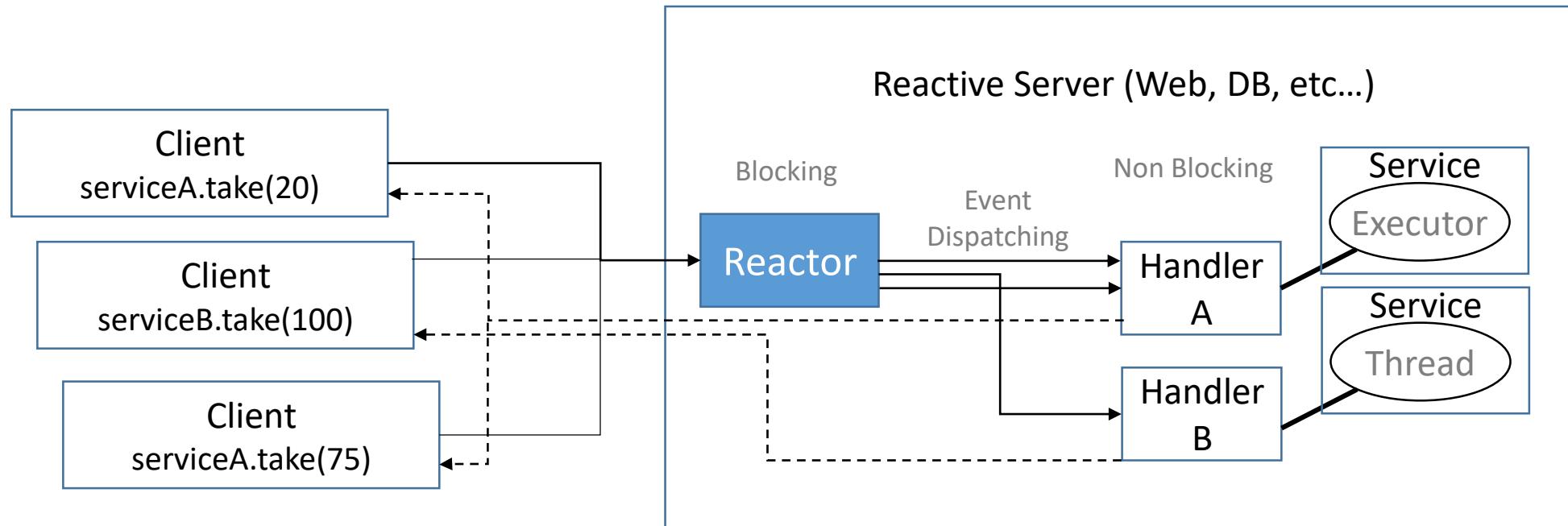
The Reactor Design Pattern

- Reactive system has one input and multiple outputs
 - Decouples application-level code from reusable reactor
 - Reactor is like an operator that accepts client calls and forwards it to the appropriate extension
- De-multiplexed services
 - Clients can split requests in order to receive responses in appropriate timing & sizes
 - Work is not done in a “Thread per request” manner – work is distributed to subsystems via handlers



Introduction

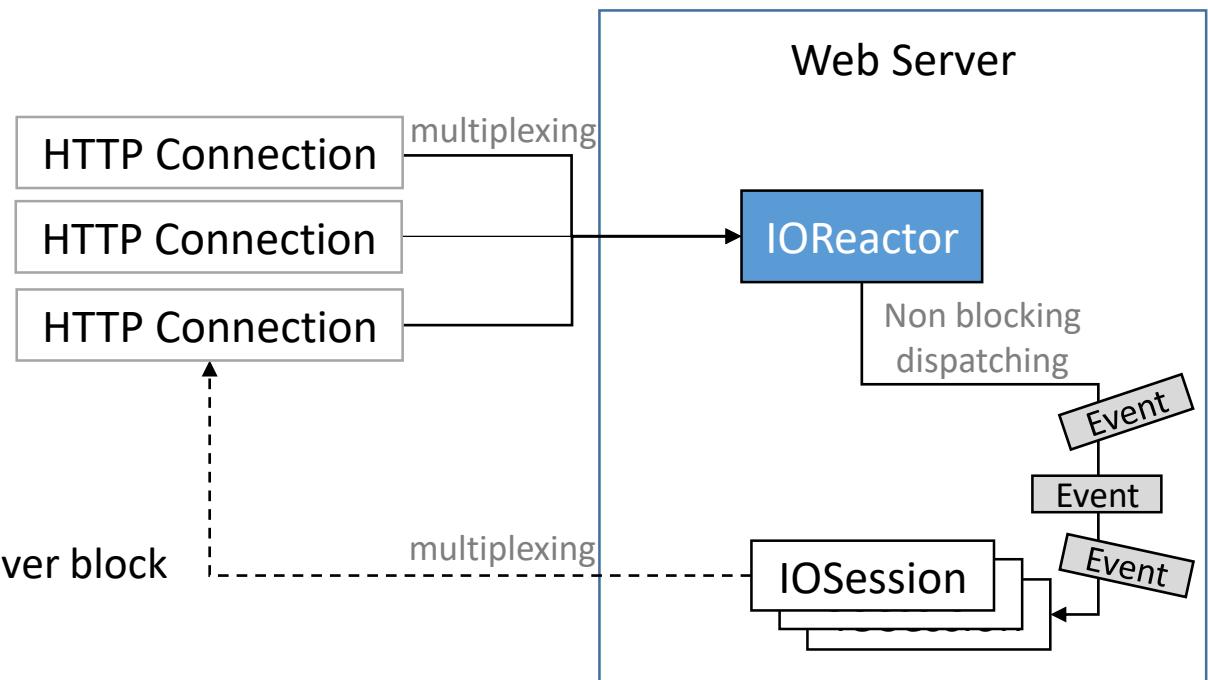
The Reactor Design Pattern



Introduction

Java HttpCore NIO Reactor Platform

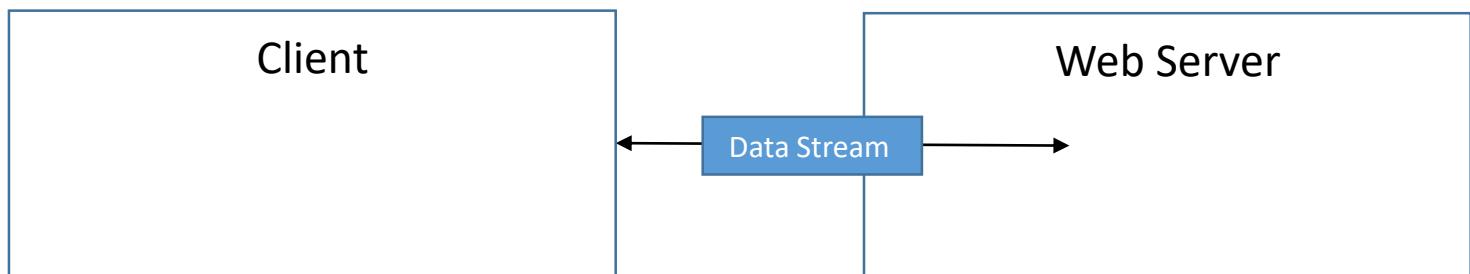
- **IOReactor**
 - Is the dispatcher
 - Uses several threads
 - Uses multiplexing (selectable channel)
 - Recommended: one IOReactor per CPU
 - May listen to several ports
 - Can be shutdown
- **IOEventDispatch**
 - Is the event delegated to registered handlers
 - Triggered by IOReactor thread(s) – so should never block
- **IOSession**
 - Is an handler capable of reading and writing from and to clients
 - Sessions stores continuous data internally – NOT in any thread context (!)
 - Uses non-blocking I/O as well (selectable channels)
 - Can be shutdown



Introduction

Reactive programming benefits

- Preventing back pressure
- Data streaming over HTTP2
- Using few threads effectively (no blocks)
- Simple asynchronous modelling
- Handling client request in a pipeline manner



Introduction

Reactive Programming Evolution in Java

- Java 7 – ForkJoin
 - Executor which supports recursive tasks
- Java 8 – CompletableFuture & Functional programming
 - Future result can be used for further manipulation in the same or alternative thread pool
- Java 9 – Flow API - RxJava



Java Flow API

Java Flow API

Java 9 adds the core infrastructure for reactive programming

- Flow – unit that processes events and encapsulates concurrency
- Subscriber – event endpoint
- Publisher – generates events and publishes to registered subscribers
- Processors – subscribing interceptors (for creating subscription chain)

Java Flow API

Subscriber

- Subscriber supports data subscription over reactive platforms
- Defines methods for reactive subscription
 - onComplete()
 - onError(Throwable t)
 - onNext()
 - onSubscribe()

Java Flow API

Publisher

- Publisher supports registering subscribers for consuming data over reactive platforms
 - `Subscribe(Subscriber<T> sub)`
 - Extends `Subscriber`. Publishers are also Subscribers
- Subscriber events
 - `onSubscribe()` – after calling `Publisher.subscribe(Subscriber s)`
 - `onNext()` – notifies `Subscription.request(long)`

Java Flow API

Steps in creating reactive application:

- Create a Flow.Publisher
- Register Flow.Subscribers via Publisher.subscribe()
- Implement Subscriber to handle events:
 - onSubscribe()
 - onNext()
 - onError()
 - onComplete()
- Use Publisher to generate events
- Flow acts like a Pipe here – passing events from publisher to the ‘Sink’ side – the Consumer
- BUT –unlike pipes - it uses Executor, the daemon common pool (ForkJoin)

Java Flow API

- When creating Publisher
 - Default constructor uses common pool (Fork-Join daemon pool)
 - Alternative executors may be used instead
 - This is how all thread complexity remains hidden
- Multiple subscribers may be registered to a single Publisher
 - Use `publisher.subscribe()`

```
//Create Publisher (works with common-pool)
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

// Create Publisher with dedicated pool
Executor e=Executors.newFixedThreadPool(3);
SubmissionPublisher<String> publisher = new SubmissionPublisher<>(e);
```

Java Flow API

Subscriber

```
public class MySubscriber<T> implements Subscriber<T> {
    private Subscription subscription;
    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
    }
    @Override
    public void onNext(T item) {
        subscription.request(1); //Long.MAX_VALUE may be considered as unbounded
    }
    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("Done");
    }
}
```

Java Flow API

Publishing messages

- Use `publisher.submit(T)`
- Each registered Subscriber get its own instance of Subscription

```
//Create Publisher
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

//Register Subscriber
MySubscriber<String> subscriber = new MySubscriber<>();
publisher.subscribe(subscriber);

//Publish messages
String[] items = {"msg1", "msg2", "msg3"};
Arrays.asList(items).stream().forEach(publisher::submit);
```

Java Flow API

Processors

- Enhanced subscribers
- While subscribers acts as endpoints, processors delegates messages
- Processor uses `Function<T,R>` to process messages
 - Incoming message is T
 - Outgoing message is R (which might be T as well..)

Java Flow API

Processors example

```
public class Processor1<T,R> extends SubmissionPublisher<R> implements Processor<T, R> {

    private Function<? super T, ? extends R> function;
    private Subscription subscription;

    public MyTransformProcessor(Function<? super T, ? extends R> function) {
        super();
        this.function = function;
    }
    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
    }
    @Override
    public void onNext(T item) {
        submit((R) function.apply(item));
        subscription.request(1);
    } ...
```

Java Flow API

Processors

- Now we can define a subscription chain
- This is how we split asynchronous tasks while using thread pools

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
MySubscriber<Integer> subscriber = new MySubscriber<>();

//Creating Midpoints Processors
Processor1<String, String> p1 =
    new Processor1<>(s -> {if(s.equals("msg1"))return "100"; return "200";});
Processor1<String, Integer> p2 =
    new Processor1<>(s -> Integer.parseInt(s));

//Configuring subscription chain
publisher.subscribe(p1);
p1.subscribe(p2);
p2.subscribe(subscriber);
```

Java Flow API

Signal

- Publishers supports Signal consumption
- Consumer<Signal> may check signal type and take action
- Signal is fired when publisher generates notification and supports:
 - isOnSubscribe()
 - isOnError()
 - isOnNext()
 - isOnComplete()

SignalType Enum

- AFTER_TERMINATE
- CANCEL
- ON_COMPLETE
- ON_ERROR
- ON_NEXT
- ON_SUBSCRIBE
- REQUEST

Spring Reactive Programming

Spring Reactive Programming

- Spring WebFlux
- Provides abstraction which supports

- RxJava
- ReactiveX



- Reactor



- Docs: *<https://projectreactor.io/docs>*



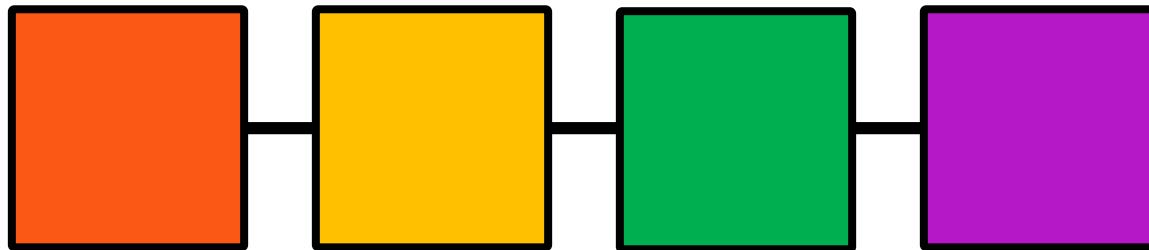
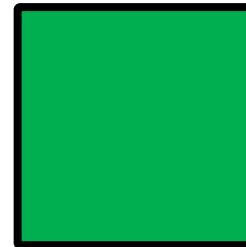
Spring Reactive Programming

- SpringBoot
 - Minimum major version 2.0.*
 - Select dependency: Reactive Web
 - Spring WebFlux
 -  Netty
- WebFlux Maven dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

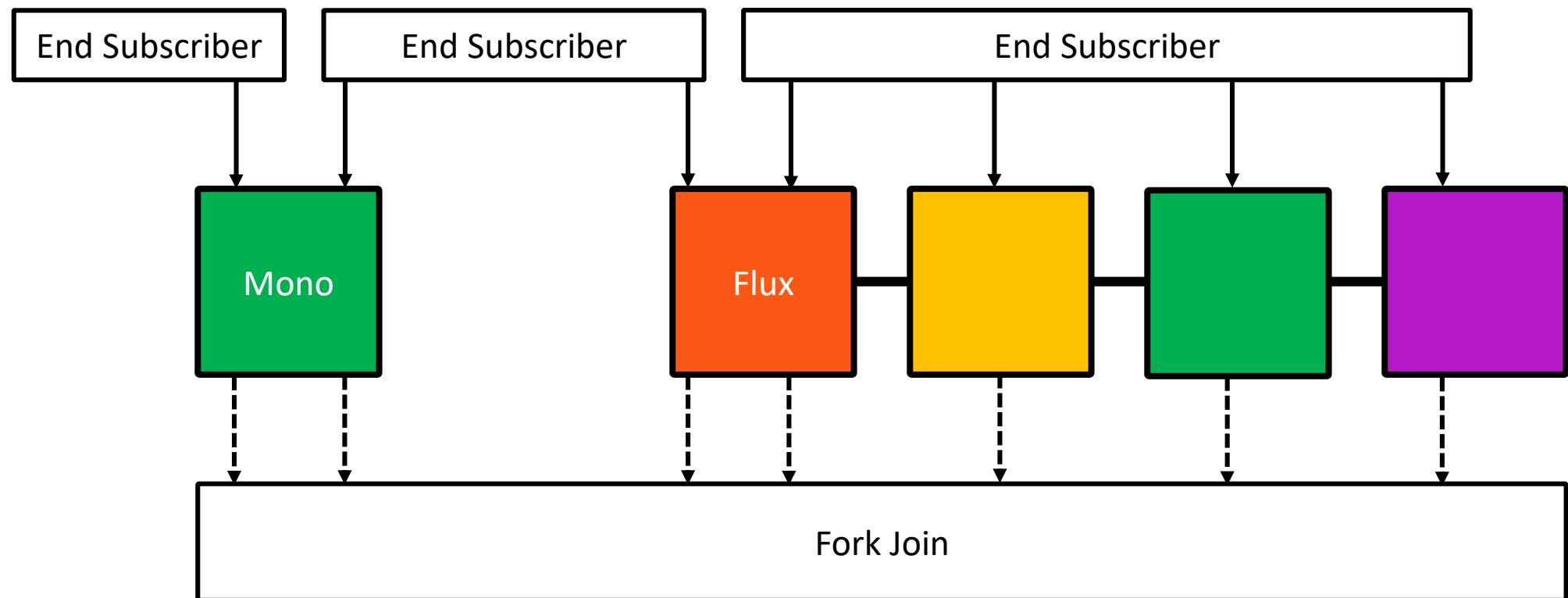
Publishers (& Subscribers)

- Spring publishers:
 - Mono – for single item subscription
 - Flux – for multiple items
 - Offers rich set of stream consumption functionality



Publishers (& Subscribers)

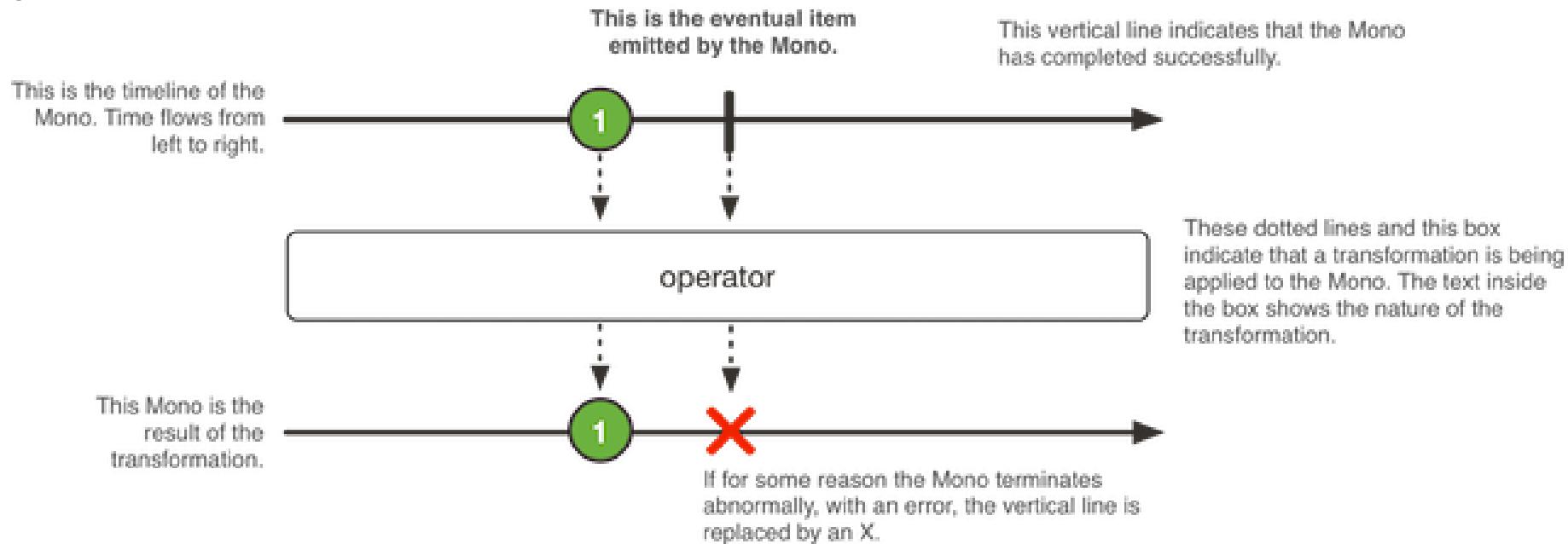
- Reactive Platform:



Mono

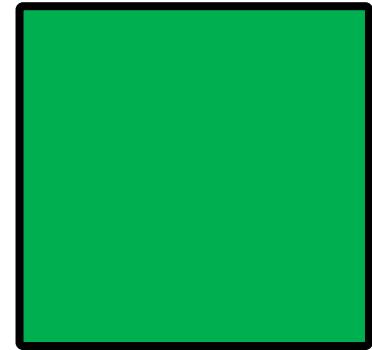
Mono

- For emitting:
 - Nothing
 - Single value
 - Error



Mono

- Creating Mono via static methods:
 - `create(Consumer<Sink<T>> callback)`
 - creates Mono<R> using Consumer<T>
 - Sink<T> is basically a kind of stream supplier
 - success() & error() constructs Mono<T>/Mono<Ex>
 - listeners may be attached



Mono

- Example:
 - `create(Consumer<Sink<T>> callback)`

```
public Mono<String> monoCreate(String value){  
    return Mono.create(sink->{  
        if(value.length()<10) {  
            String response="Hello "+value+" !";  
            sink.success(response);  
        }else  
            sink.error(new Exception("value ("+value+") is too long"));  
    });  
}
```

```
System.out.println("Mono Create: "+ex.monoCreate("David").block());
```

```
Mono Create: Hello David !
```

Mono

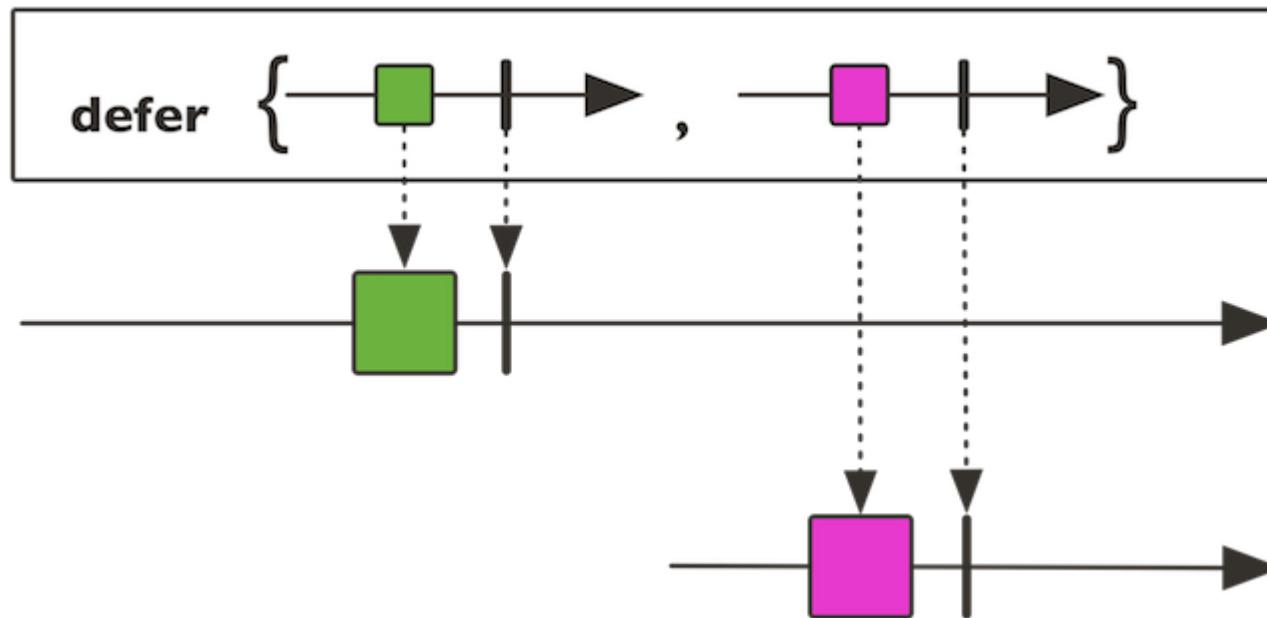
- Example:
 - `create(Consumer<Sink<T>> callback)`

```
// argument is too long and should lead to Mono.create error
System.out.println("Mono Create: "+ex.monoCreate("David12345").block());
```

```
Exception in thread "main" reactor.core.Exceptions$ReactiveException: java.lang.Exception: value (David12345) is too long
  at reactor.core.Exceptions.propagate(Exceptions.java:326)
  at reactor.core.publisher.BlockingSingleSubscriber.blockingGet(BlockingSingleSubscriber.java:91)
  at reactor.core.publisher.Mono.block(Mono.java:1175)
  at react.Application.mono(Application.java:50)
  at react.Application.main(Application.java:25)
Suppressed: java.lang.Exception: #block terminated with an error
```

Mono

- Creating Mono via static methods:
 - `defer(Supplier<Mono<T>> mono)`
 - creating `Mono<Mono<T>>` which supplies target `Mono<T>`



Mono

- Example:
 - `defer(Supplier<Mono<T>> mono)`

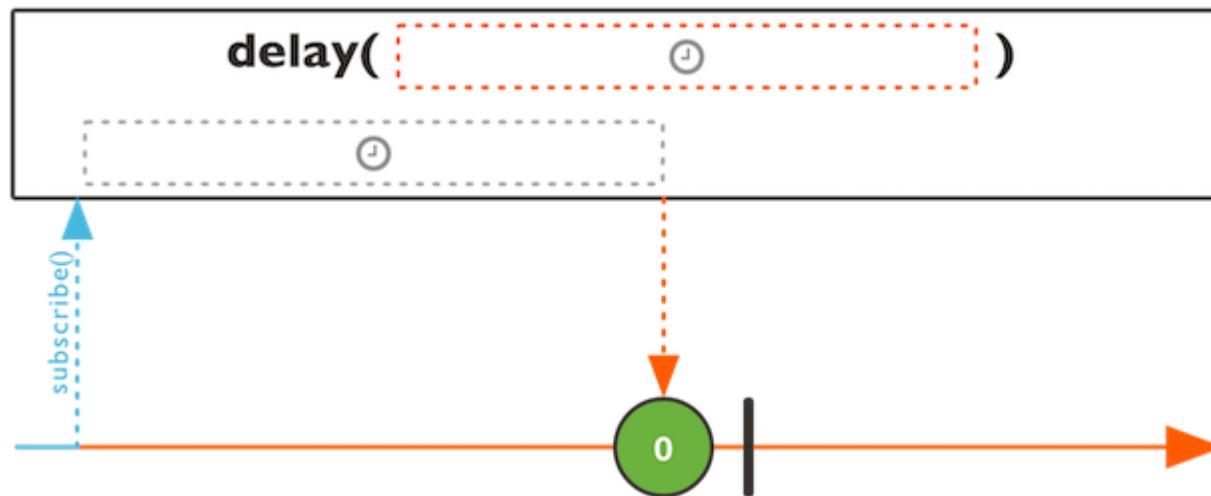
```
//here we use previous method as a Mono<String> Supplier
public Mono<String> monoDefer(String value){
    return Mono.defer(()->monoCreate("- "+value));
}
```

```
System.out.println("Mono Defer: "+ex.monoDefer("Eve").block());
```

```
Mono Create: Hello - Eve !
```

Mono

- Creating Mono via static methods:
 - `delay(Period period)`
 - `delay(Period period, Scheduler timer)`
 - creates Mono which completes after given delay and returns zero as `Mono<Long>`



Mono

- Example:
 - `delay(...)`

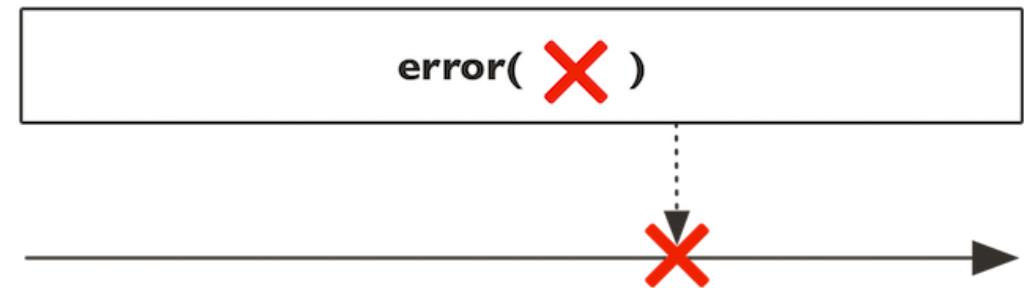
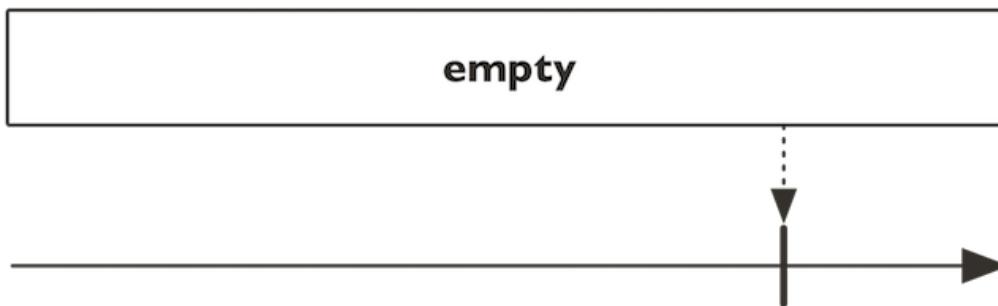
```
public Mono<Long> monoDelay(int sec){  
    return Mono.delay(Duration.ofSeconds(sec));  
}
```

```
System.out.println("Mono Delay: "+ex.monoDelay(0).block());
```

```
Mono Delay: 0
```

Mono

- Creating Mono via static methods:
 - `empty()`
 - creates empty Mono – returns `Mono<Void>`
 - `error(Throwable error)`
 - creates error Mono – returns `Mono<Throwable>`



Mono

- Example:
 - `empty(...)`

```
public Mono<?> monoEmpty(int sec){  
    return Mono.empty();  
}
```

```
System.out.println("Mono Empty: "+ex.monoEmpty().block());
```

```
Mono Empty: null
```

Mono

- Example:
 - `error(...)`

```
public Mono<Throwable> monoError(String errMsg){  
    return Mono.error(new Exception(errMsg));  
}
```

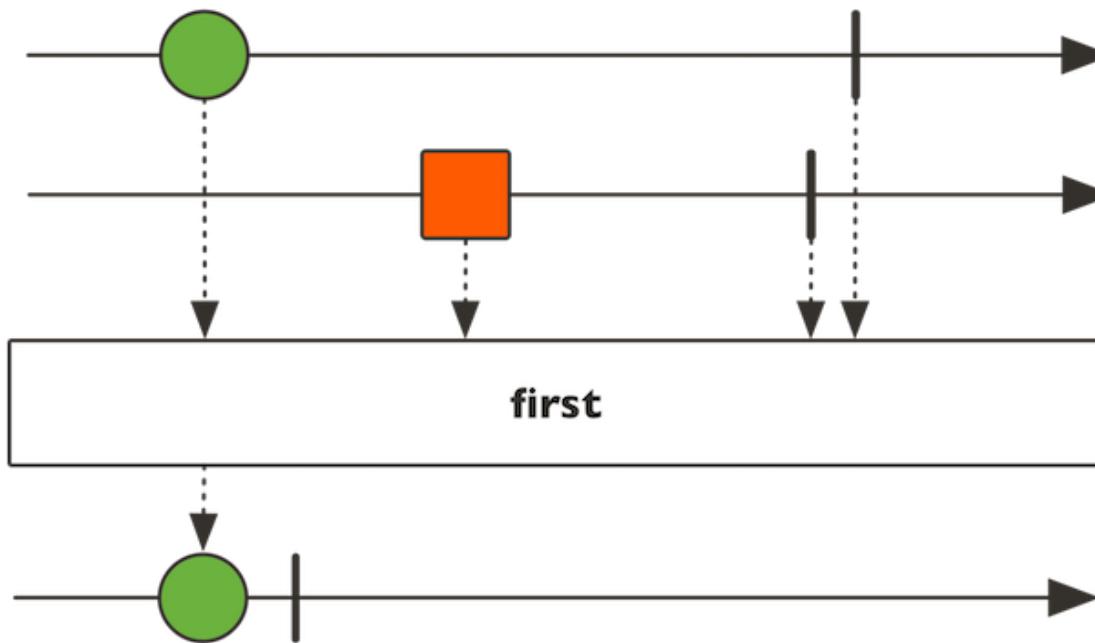
```
System.out.println("Mono Error: "+ex.monoError("Some Error").block());
```

```
Exception in thread "main" reactor.core.Exceptions$ReactiveException: java.lang.Exception: Some Error  
at reactor.core.Exceptions.propagate(Exceptions.java:326)  
at reactor.core.publisher.MonoError.block(MonoError.java:48)  
at react.Application.mono(Application.java:54)  
at react.Application.main(Application.java:25)  
Caused by: java.lang.Exception: Some Error  
at react.examples.MonoReactiveExamples.monoError(MonoReactiveExamples.java:39)  
... 2 more
```

Mono

- Creating Mono via static methods:

- `first(Iterable<Mono<T>> monos)`
- `first(Mono<T>... monos)`
 - returns first completed Mono<T>



Mono

- Example:
 - `first(...)`

```
public Mono<?> monoFirst(int sec){  
    Mono.first(monoDelay(1),monoCreate("Create"),monoJust("Just"));  
}
```

```
System.out.println("Mono First: "+ex.monoFirst().block());
```

```
Mono First: Hello Create !
```

Mono

- Creating Mono via static methods:
- Mono<T> factoring methods for supporting JSE, Flow, Concurrent APIs:
 - from(Publisher<T> publisher)
 - fromCallable(Callable<T> task)
 - fromCompletionStage(CompletionStage<T> stage)
 - fromFuture(Future<T> taskResult)
 - fromRunnable(Runnable task) – returns Mono<Void>
 - fromSupplier(Supplier<T> supplier)

Mono

- Creating Mono via static methods:

- `just(T data)`

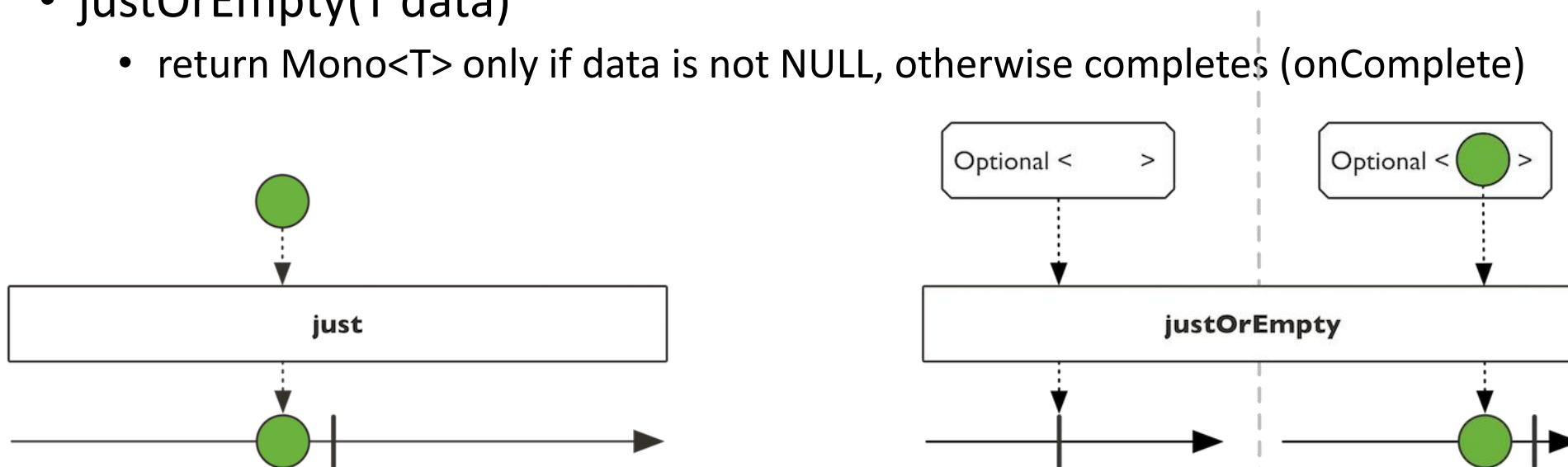
- returns `Mono<T>`, data might be null

- `justOrEmpty(Optional<T> optional)`

- return `Mono<T>` only if `optional.isPresent() == true`, otherwise completes (onComplete)

- `justOrEmpty(T data)`

- return `Mono<T>` only if data is not NULL, otherwise completes (onComplete)



Mono

- Example:
 - `just(...)`

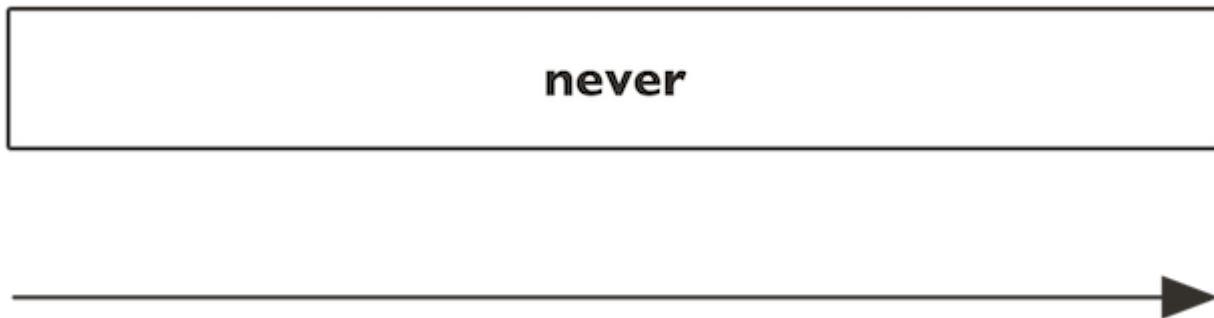
```
public Mono<String> monoJust(String data){  
    return Mono.just(data);  
}
```

```
System.out.println("\nMono Just: "+ex.monoJust("Just..").block());
```

```
Mono Just: Just..
```

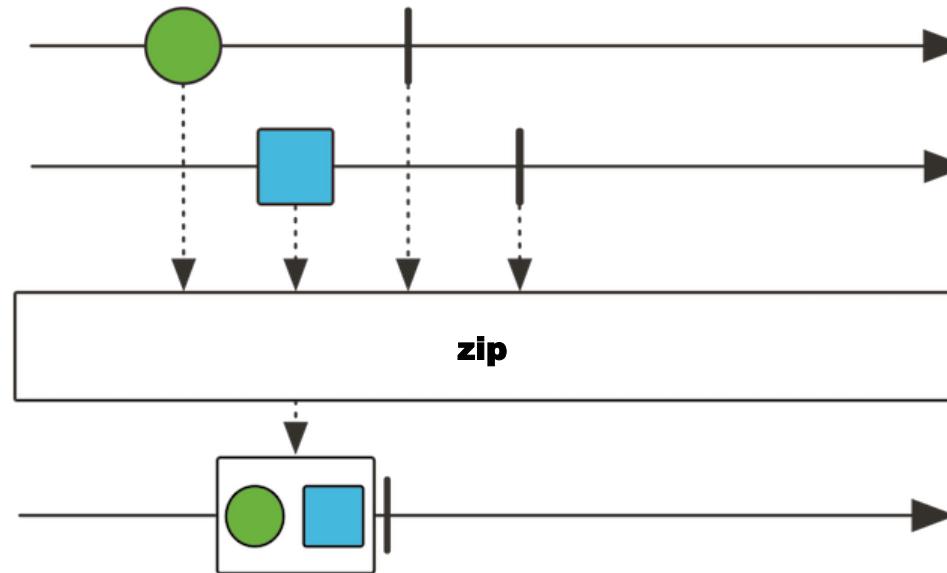
Mono

- Creating Mono via static methods:
 - `never()`
 - Mono<T> which never completes



Mono

- Creating Mono via static methods:
 - `zip(Iterable<Mono<T>> monos, Function<T,R> combiner)`
 - `zip(Function<T,R> combiner, Mono<T>... monos)`
 - combines all `Mono<T>` results as `Object[]` using combiner function



Mono

- Example:
 - `zip(...)`

```
//reduces to total words length
public Mono<Integer> monoZip(String ...data){
    List<Mono<String>> monos=new ArrayList<>();
    for(int x=0;x<data.length;x++) {
        monos.add(Mono.just(data[x]));
    }
    return Mono.zip(monos,(results)->{
        int total=0;
        for(Object r:results) {
            total+=((String)r).length();
        }
        return total;
    });
}
```

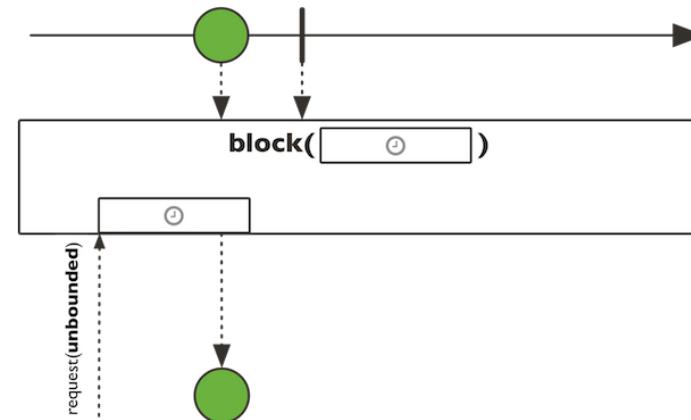
```
System.out.println("\nMono Zip: "+ex.monoZip("a","bb","ccc").block());
```

```
Mono Zip: 6
```

Mono

- Mono instance main methods:

- `as(Function<Mono<T>,R> transformer)`
 - transforms `Mono<T>` into `R`
- `block()`
- `block(Duration duration)`
 - waits until next signal and returns `T`
- `blockOptional()`
- `blockOptional(Duration duration)`
 - waits until next signal or `Mono` completes empty, returns `Optional<T>`

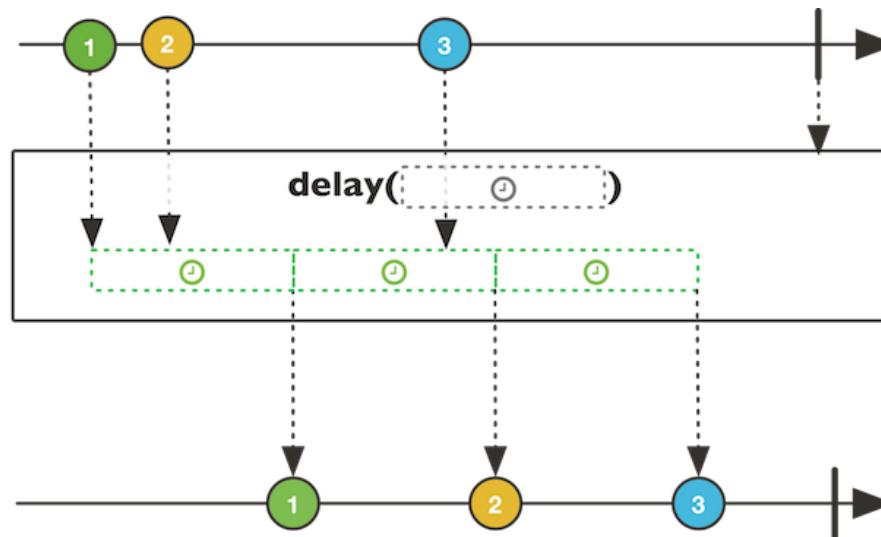


Mono

- Mono instance main methods:
 - `cancelOn(Scheduler expiration)`
 - sets subscriber to discard `Mono<T>` request on expiration
 - `checkpoint()`
 - activates assembly mono tracer
 - returns `Mono<T>` which will be the highest point up the stream or exception
 - should be used close to the end of the stream in order to reduce stack trace weight

Mono

- Mono instance main methods:
 - `delayElement(Duration duration)`
 - `delayElement(Duration duration, Scheduler scheduler)`
 - returns `Mono<T>` after specified duration
 - basically used for placing `sleep()` like actions on the stream



Mono

- Example:
 - `delayElement(...)`

```
public Mono<String> monoDelayElement(String data){  
    return Mono.just("1").doOnNext(System.out::print).doOnNext(s->System.out.println(" --("+s+")--"));  
}
```

```
System.out.println("\nMono Delay Element: ");  
ex.monoDelayElement().block();
```

```
Mono Delay Element:
```

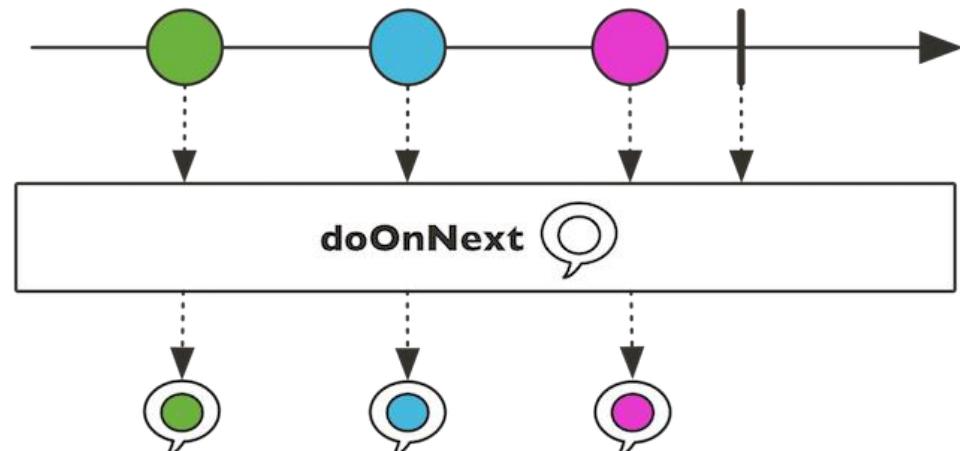
```
Hi ! Hi ! Hi !
```



Mono

- Mono instance main methods:

- `doOnNext(Consumer<T> consumer)`
- `doAfterSuccessOrError(BiConsumer<T, Throwable> consumer)`
 - uses given consumer to process T
 - BiConsumers assigned values:
 - null & null for empty mono
 - T & null for normal mono outcome
 - null & exception on mono error



Mono

- Example:
 - `doOnNext(...)`

```
public Mono<String> monoDoOnNext(String data){  
    return Mono.just("1").doOnNext(System.out::print).doOnNext(s->System.out.println(" --("+s+")--"));  
}
```

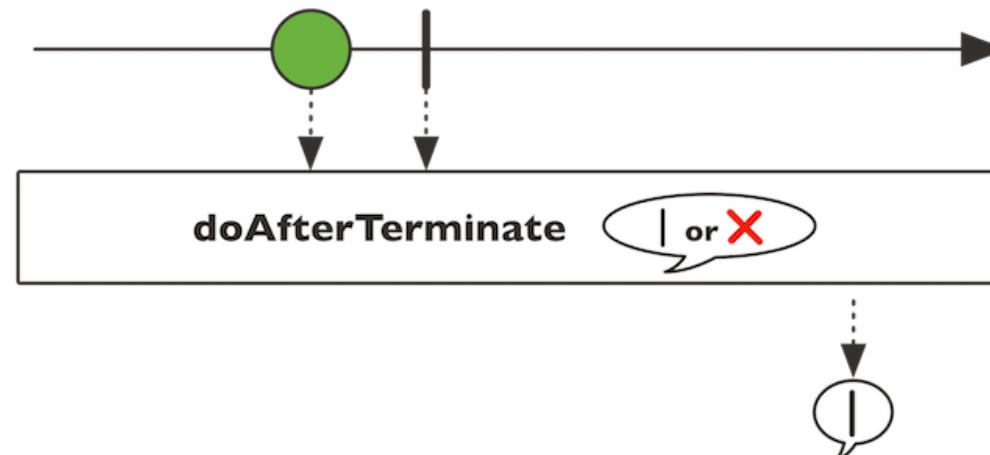
```
System.out.println("\nMono DoOnNext: ");  
ex.monoDoOnNext().block();
```

```
Mono DoOnNext:  
1 --(1)--
```

Mono

- Mono instance main methods:

- `doOnTerminate(Runnable task)`
 - executes task after `Mono<T>` termination
- `doFinally(Consumer<SignalType> consumer)`
 - Delegates to given consumer ‘completion type’ which might be:
 - `SignalType.ON_COMPLETE`
 - `SignalType.ON_ERROR`
 - `SignalType.ON_CANCEL`



Mono

- Example:
 - `doOnTerminate(...)`

```
public Mono<String> monoDoOnNext(String data){  
    return Mono.just("Done ! ").doOnTerminate(()->System.out.println("On Terminate"));  
}
```

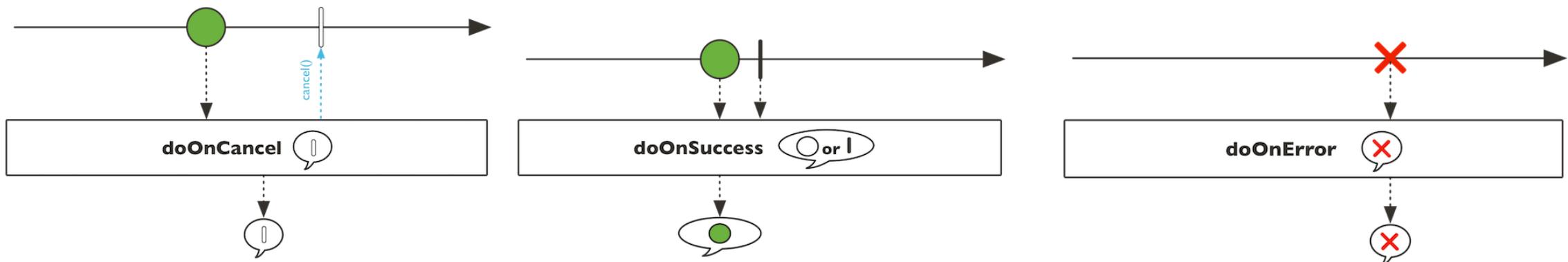
```
System.out.println("\nMono DoOnNext: ");  
ex.monoDoOnTerminate().block();
```

Mono On Terminate:
On Terminate

Done !

Mono

- Mono instance main methods:
 - `doOnCancel(Runnable task)`
 - executes given task only if & after `Mono<T>` cancellation
 - `doOnSuccess(Consumer<T> consumer)`
 - delegates T to given consumer only in case of success
 - `doOnError(Consumer<Throwable> consumer)`
 - `doOnError(Class<E> exceptionType, Consumer<E> consumer)`
 - delegates exception/specific-exception to given consumer only in case of success



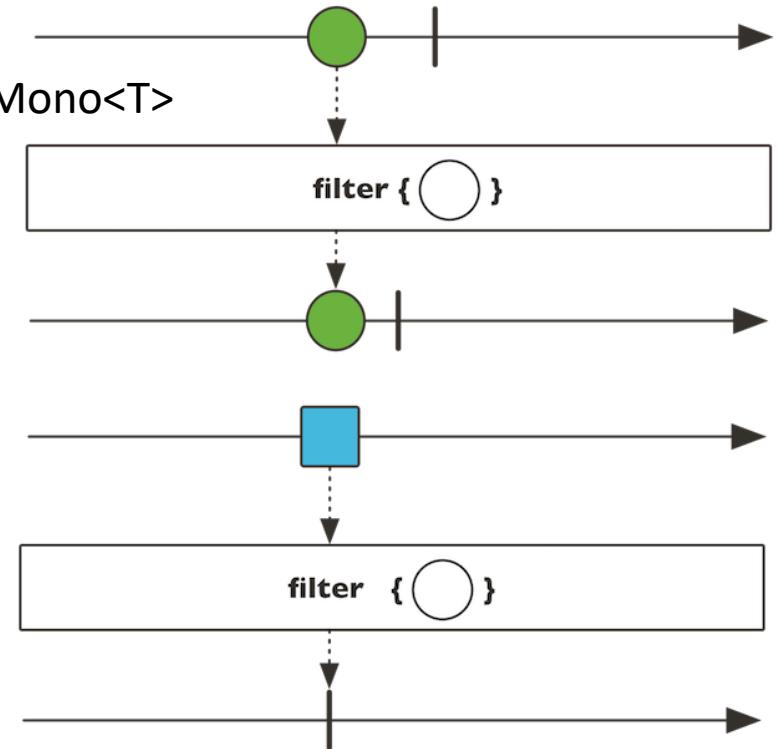
Mono

- Mono instance main methods:

- filter(Predicate<T> tester)

- tests Mono<T> value if exists:

- true - Mono<T> - if value passes the test (true) – replays Mono<T>
 - false or no-value – completes without any value



Mono

- Example:
 - filter(...)

```
public Mono<Integer> monoFilter(int value){  
    return Mono.just(value).filter(i->i%2==0);  
}
```

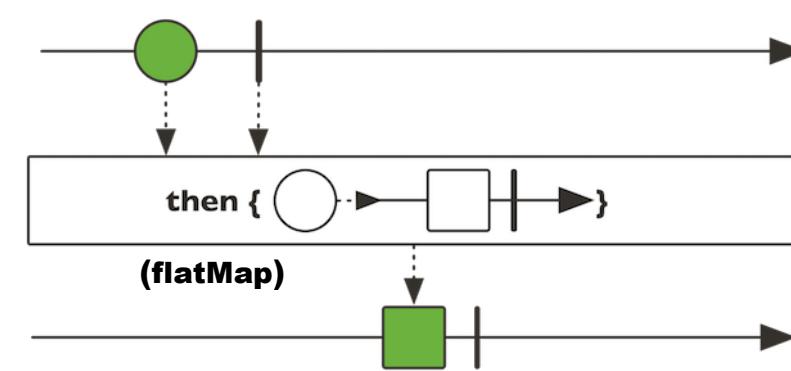
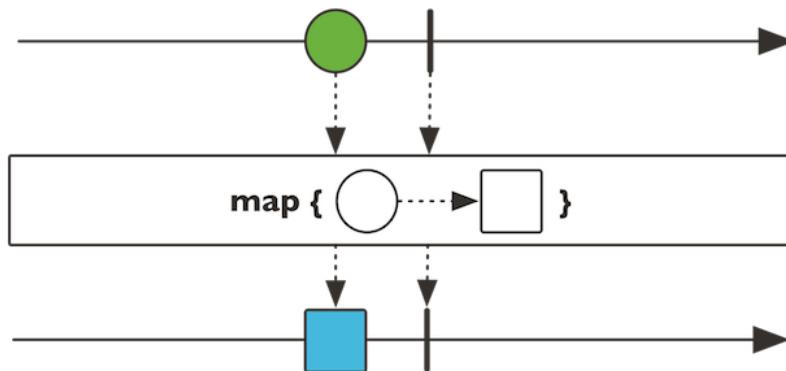
```
System.out.println("\nMono Filter(passes test): "+ex.monoFilter(20).block());  
System.out.println("\nMono Filter(fails test): "+ex.monoFilter(13).block());
```

```
Mono Filter(passes test): 20  
Mono Filter(fails test): null
```

Mono

- Mono instance main methods:

- `map(Function<T, Mono<R>) mapper)`
 - synchronously creates `Mono<R>` from `Mono<T>` by applying resulted `T` to a given mapper
- `flatMap(Function<T, Mono<R>) mapper)`
 - asynchronously creates `Mono<R>` from `Mono<T>` by applying resulted `T` to a given mapper



Mono

- Example:
 - flatMap(...)

```
public Mono<Integer> monoFlatMap(int value){  
    return Mono.just(value).flatMap(s->Mono.just(s.length()));  
}
```

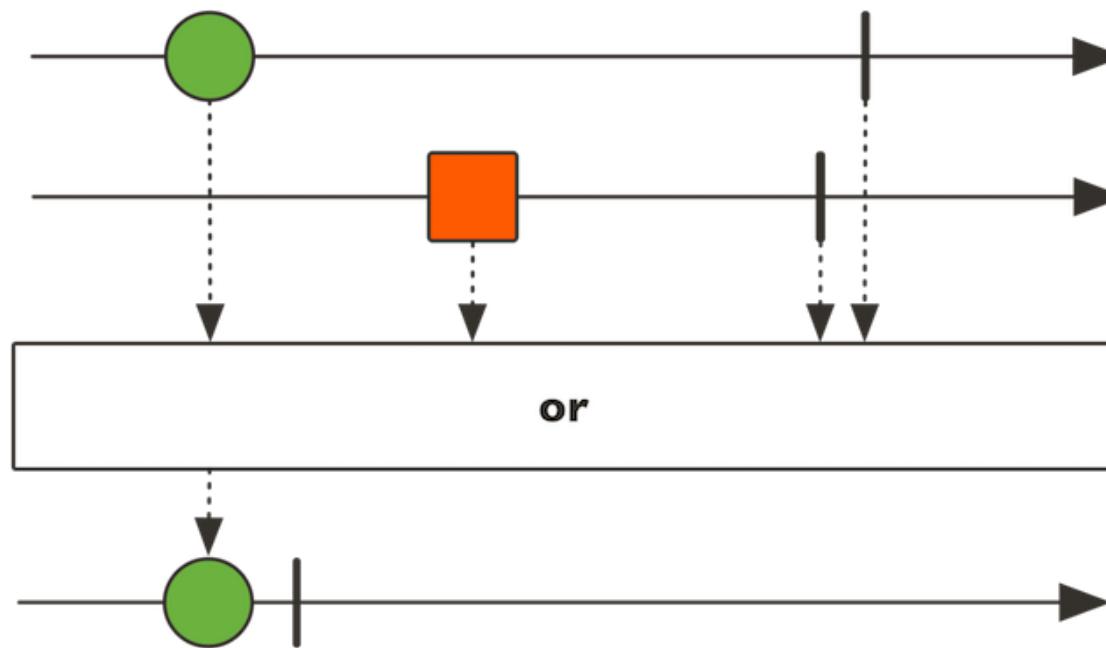
```
System.out.println("\nMono FlatMap: "+ex.monoFlatMap("Hello Reactive !").block());
```

```
Mono FlatMap: 16
```

Mono

- Mono instance main methods:

- or(Mono<T> alternative)
 - returns the first completed Mono<T> - this or alternative



Mono

- Example:
 - `or(...)`

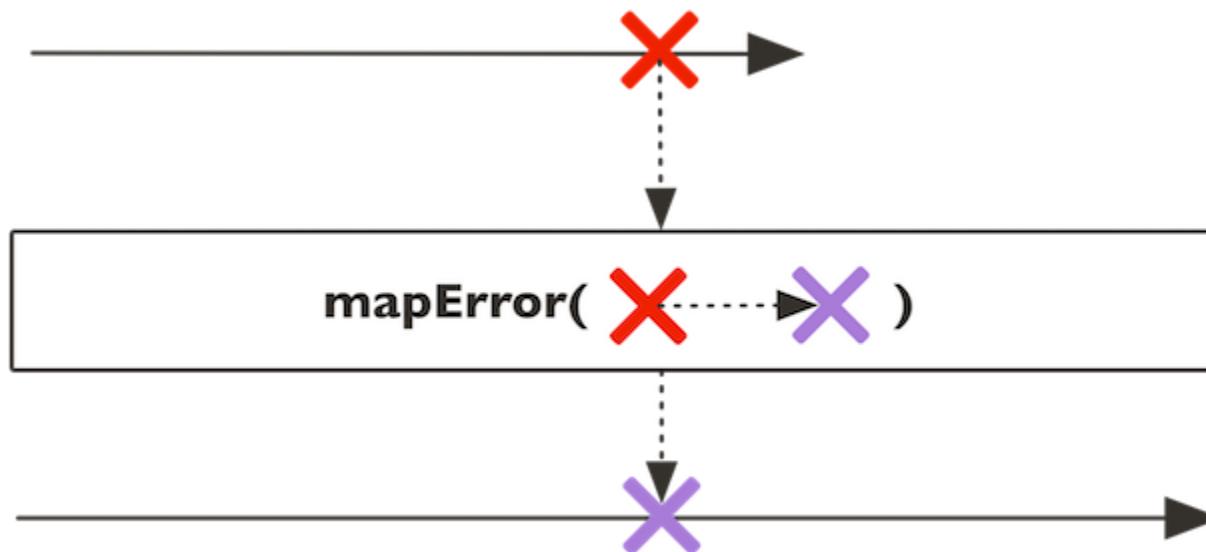
```
public Mono<String> monoOr(int delay1, int delay2){  
    Mono<String> first=Mono.just("First").delayElement(Duration.ofSeconds(delay1));  
    Mono<String> last=Mono.just("Second").delayElement(Duration.ofSeconds(delay2));  
    return first.or(last);  
}
```

```
System.out.println("\nMono Or: "+ex.monoOr(1,2).block());
```

```
Mono Or: First
```

Mono

- Mono instance main methods:
 - `onErrorMap(Function<T,T> mapper)`
 - maps Throwable into other Throwable, returns `Mono<T>`

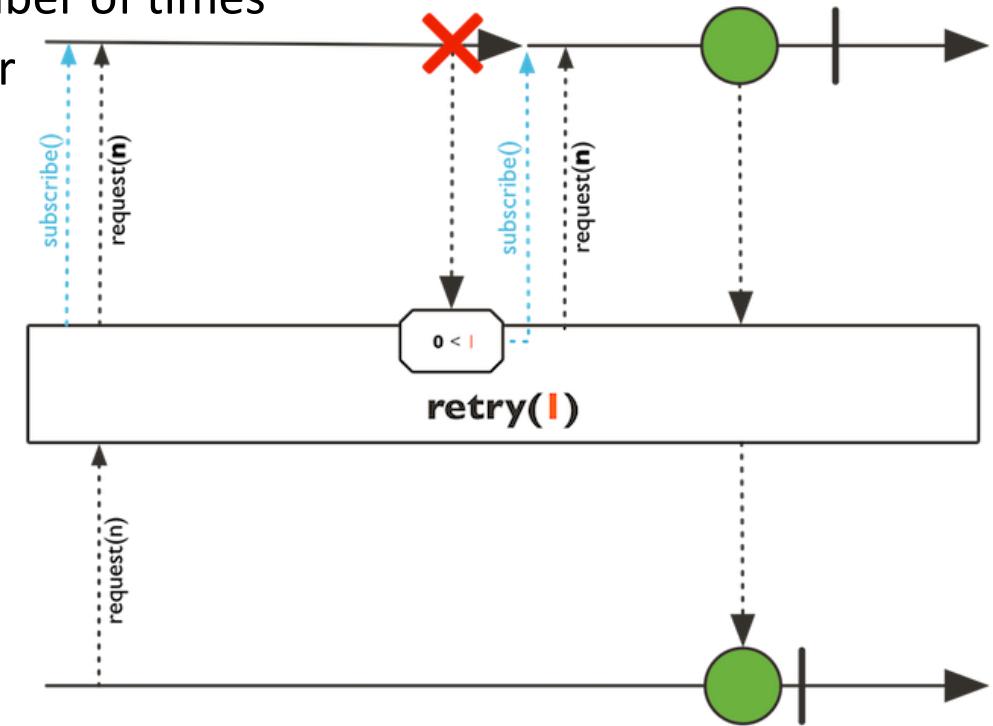


Mono

- Mono instance main methods:

- `retry(long numRetries)`

- re-subscribe to this Mono for fixed number of times
- retries only while terminating with error



Mono

- Example:
 - `retry(...)`

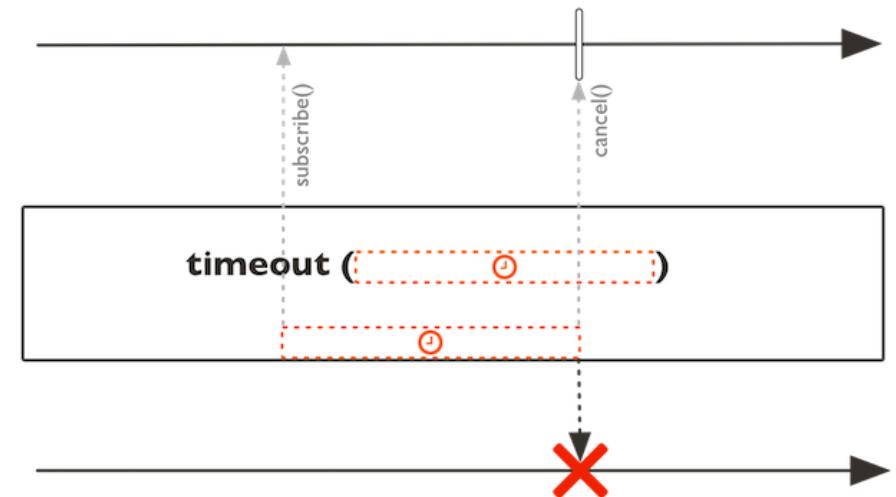
```
public Flux<Double> fluxRetry(int numOftries){  
    return Mono.error(new Exception()).doOnEach(e->System.out.print("-")).retry(numOftries);  
}
```

```
System.out.println("Mono Retry");  
System.out.println(ex.monoRetry(3).block());
```

```
Mono Retry  
null -null -null -null -Exception in thread "main" reactor.core.Exceptions$ReactiveException:  
java.lang.Exception
```

Mono

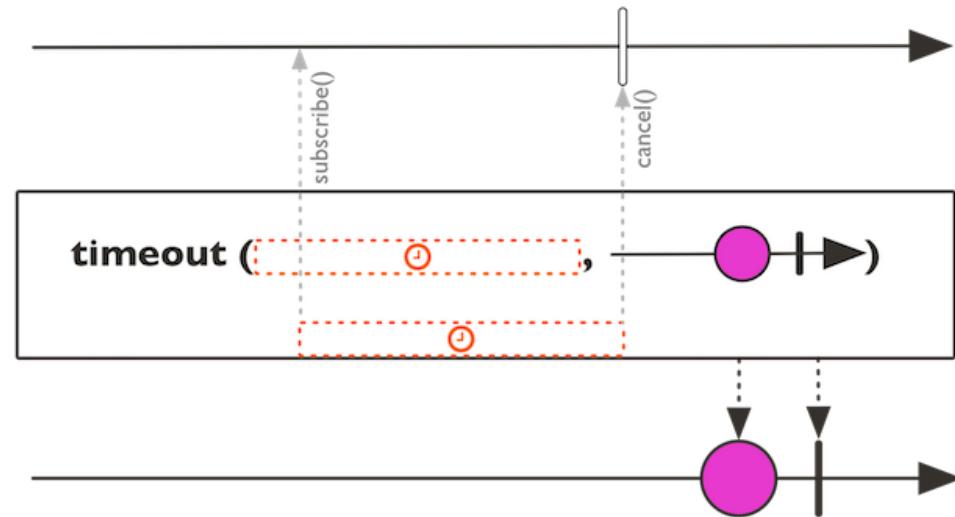
- Mono instance main methods:
 - `take(Duration duration)`
 - `take(Duration duration, Scheduler scheduler)`
 - gives Mono time to complete within specified timeframe



Mono

- Mono instance main methods:

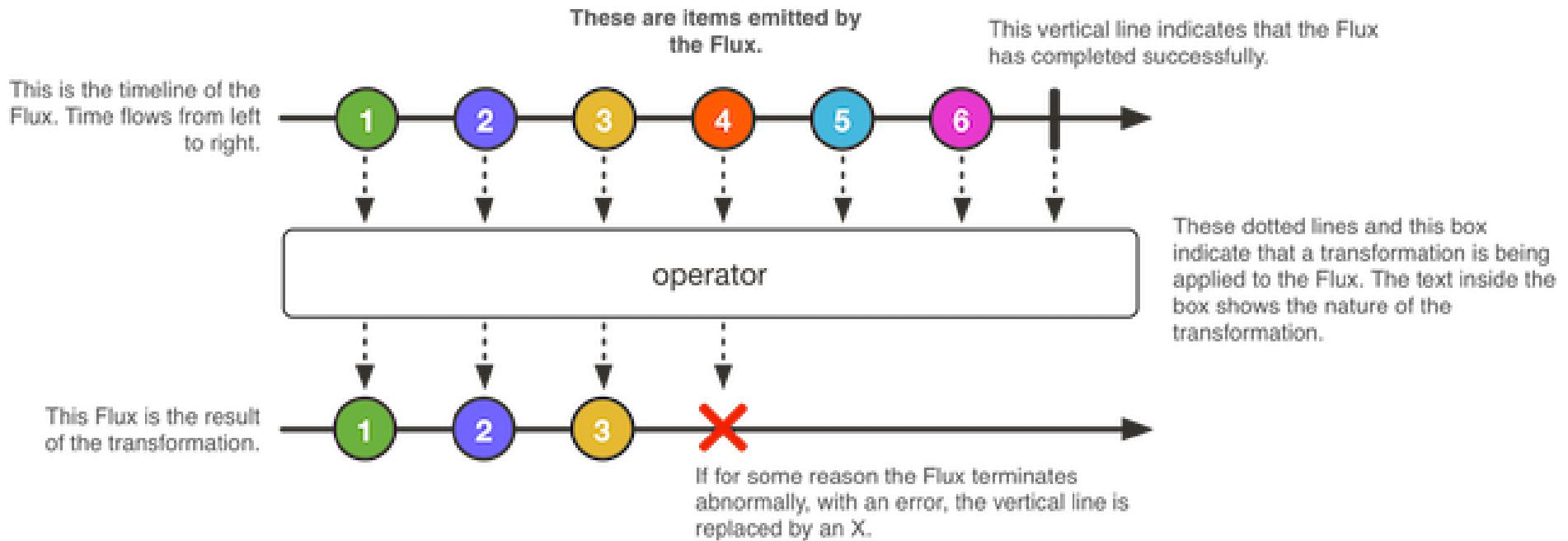
- `timeout(Duration duration, Mono<T> fallback)`
- `timeout(Duration duration, Mono<T> fallback, Scheduler timer)`
 - waits for completion
 - fallbacks to givenMono<T> when mono fails to complete within given timeframe



Flux

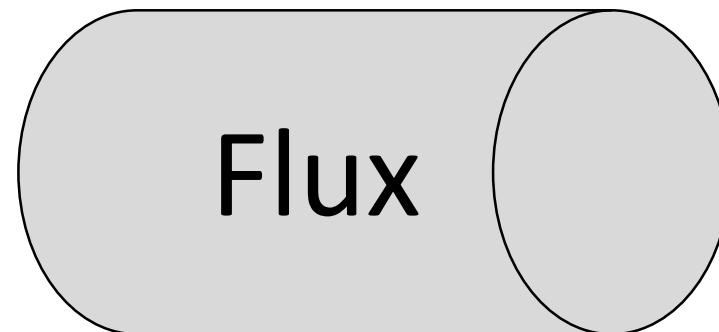
Flux

- For emitting:
 - 0-N elements
 - Error



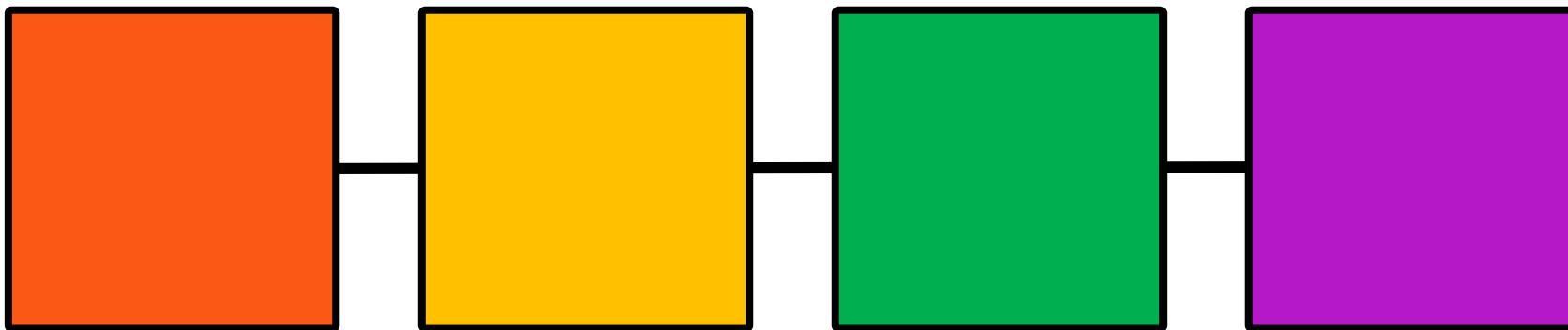
Flux

- Flux characteristics
 - Lazily executed (..are streams)
 - Immutable – every flux method factors another
 - Can be finite or infinite
 - Flux subscribers can
 - Decide when to start
 - Specify how much to take
 - Decide when to stop



Flux

- Creating Flux via static methods:
 - `create(Consumer<FluxSink> emitter)`
 - creates Flux<T> using Consumer<T>
 - `fromArray(T[] array)`
 - `fromIterable(Iterable<T> collection)`
 - `fromStream(Stream<T> stream)`



Flux

- Example:
 - `create(...)`

```
public Flux<Double> fluxCreate(){  
    return Flux.create(sink->sink.next(Math.random()));  
}
```

```
System.out.print("\nFlux Create: \n");  
Flux<Double> f=ex.fluxCreate();  
f.subscribe(System.out::println);  
f.subscribe(System.out::println);
```

```
Flux Create:  
0.1018257230717331  
0.05922774911639461
```

Flux

- Example:
 - `from(...)`

```
public Flux<String> fluxFrom(){  
    return Flux.fromStream(Stream.of("aaa","bbb","ccc"));  
}
```

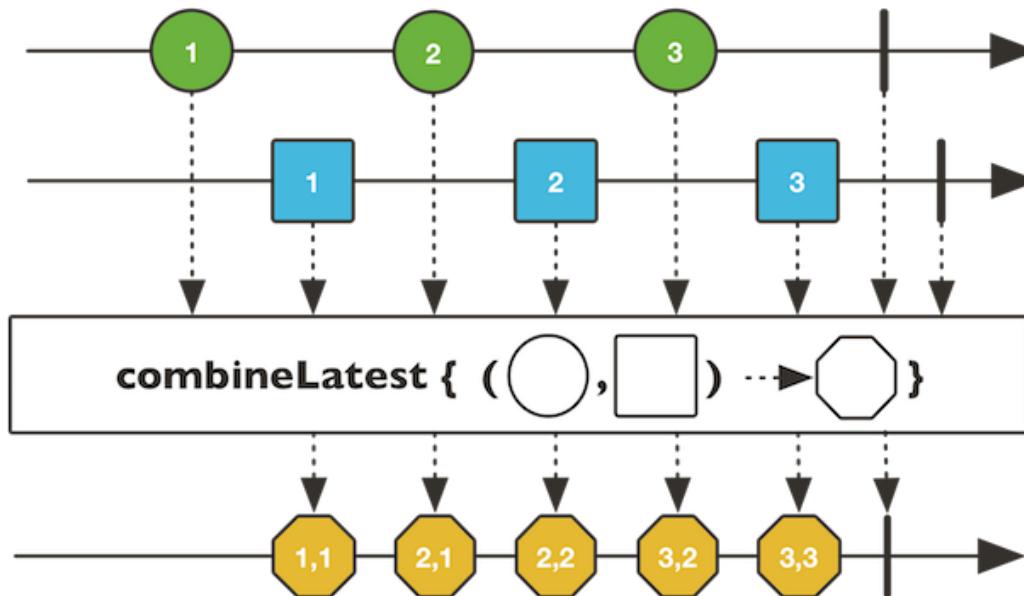
```
System.out.print("\nFlux From: ");  
ex.fluxFrom().subscribe(System.out::print);
```

```
Flux From: aaabbccc
```

Flux

- Creating Flux via static methods:

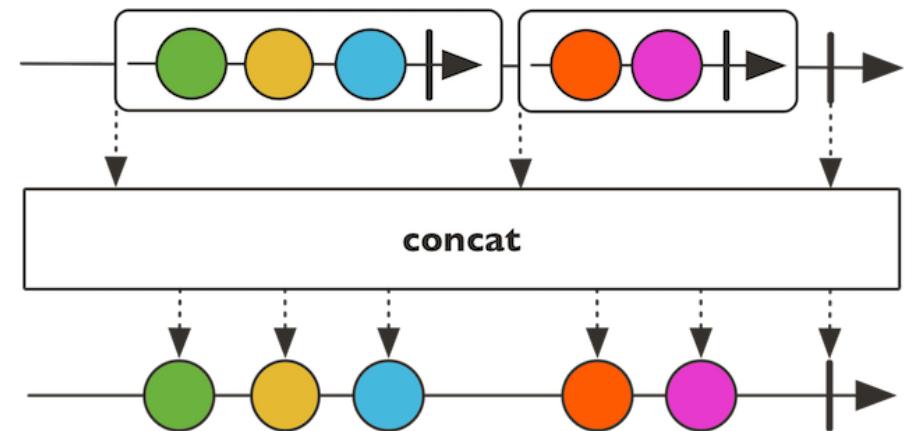
- `combineLatest(Function <Object[]> lastResults, V combiner, Publisher<T>...sources)`
- `combineLatest(Iterable<Publisher<T>> sources, Function <Object[]> lastResults, V combiner)`
 - all latest results (T sources) are combined into Flux<V>



Flux

- Creating Flux via static methods:
 - `concat(Publisher<T> sources)`
 - `concatWith(Iterable<Publisher<T>> sources)`
 - appends all value emitted by each publisher into one Flux<T>
 - `concatWithValues(T... values)`
 - fuses all value with current Flux<T> into new one

Note: Doesn't eagerly subscribe any elements



Flux

- Example:
 - concat(...)

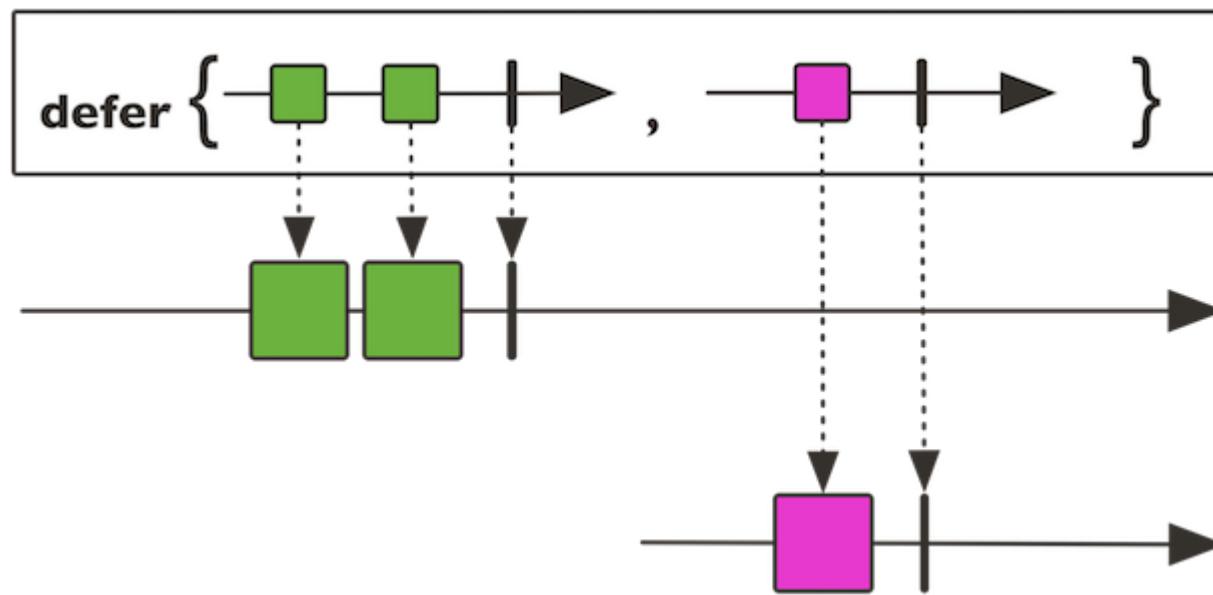
```
public Flux<String> fluxConcat(String...values){  
    return fluxFrom().concatWithValues(values);  
}
```

```
System.out.print("\n\nFlux Concat: ");  
ex.fluxConcat("1","2","3").subscribe(System.out::print);
```

```
Flux Concat: aaabbbccc123
```

Flux

- Creating Flux via static methods:
 - `defer(Supplier<Publisher<T>> publisher)`
 - creates new Flux<T> out of given Flux<T>



Flux

- Example:
 - `defer(...)`

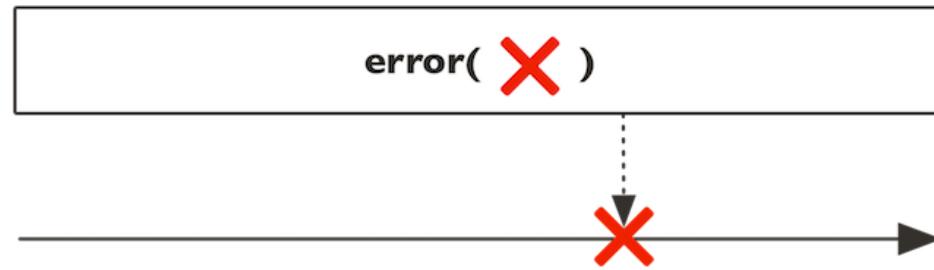
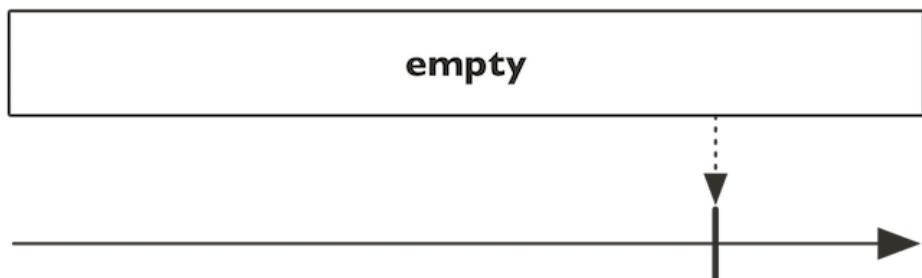
```
public Flux<Integer> fluxDefer(){  
    return Flux.defer(()->Flux.fromStream(IntStream.range(1,6).mapToObj(i->new Integer(i))));  
}
```

```
System.out.print("\n\nFlux Defer: ");  
ex.fluxDefer().subscribe(System.out::print);
```

```
Flux Defer: 12345
```

Flux

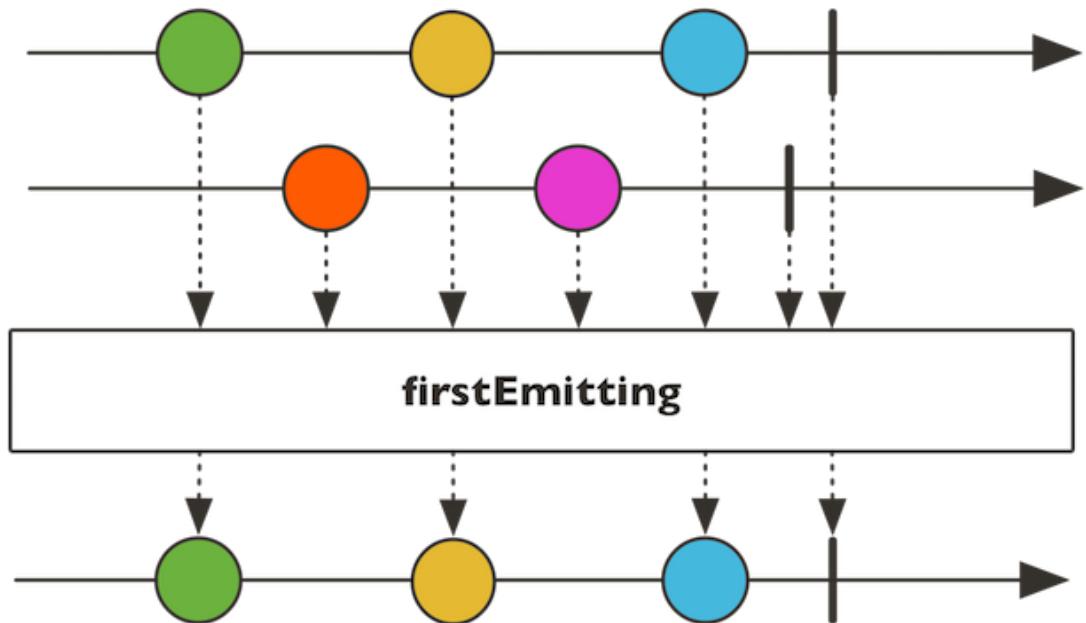
- Creating Flux via static methods:
 - `empty()`
 - creates new Flux<T> which complete without emitting any item
 - `error(Throwable error)`
 - creates new Flux<T> which complete with given error



Flux

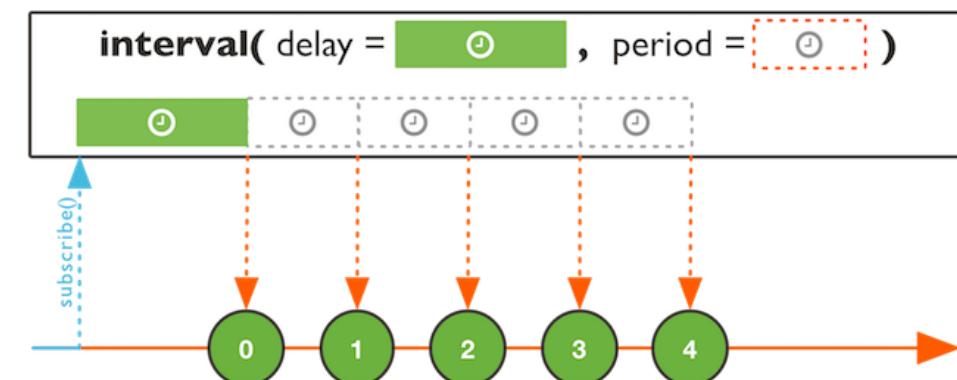
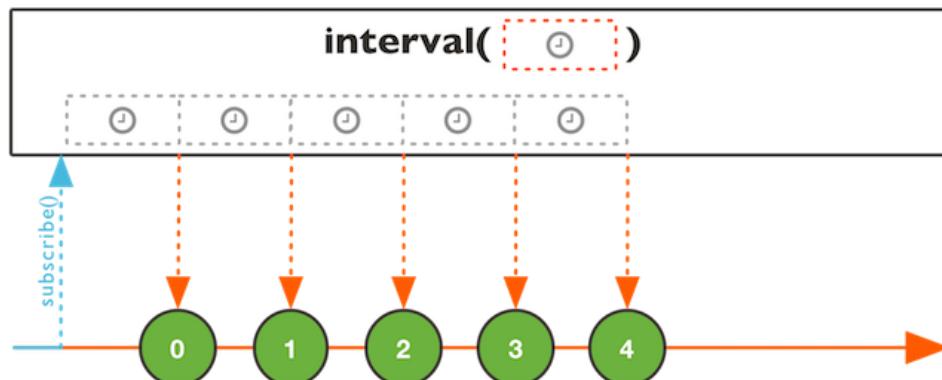
- Creating Flux via static methods:

- `first(Publisher<T>...sources)`
 - returns Flux<T> from the first source which signals onNext/onError onComplete
 - basically performs a Flux sources race



Flux

- Creating Flux via static methods:
 - `interval(Duration period)`
 - `interval(Duration delay, Duration period)`
 - `interval(Duration period, Scheduler timer)`
 - Returns Flux<Long>
 - Increments values every ‘period’



Flux

- Example:
 - `interval(...)`

```
public Flux<Long> fluxInterval(int millis, int count){  
    return Flux.interval(Duration.ofMillis(millis)).take(count);  
}
```

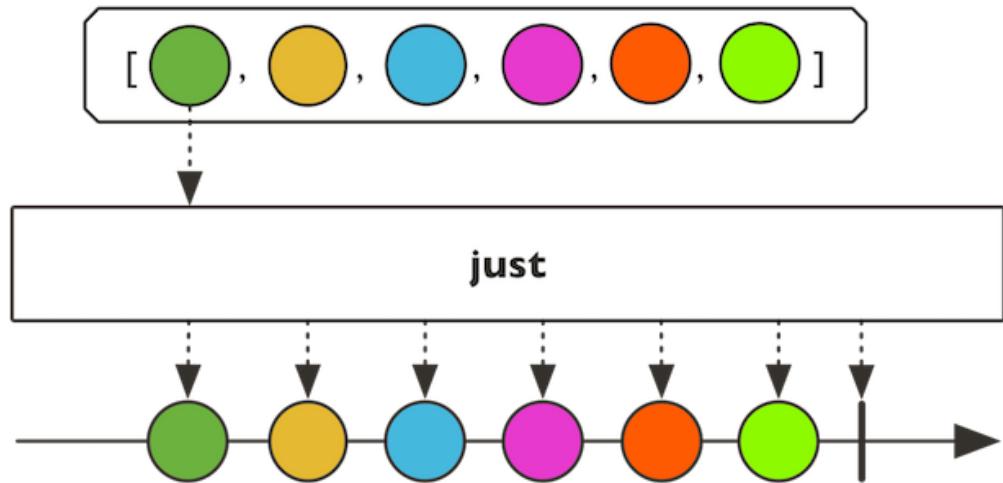
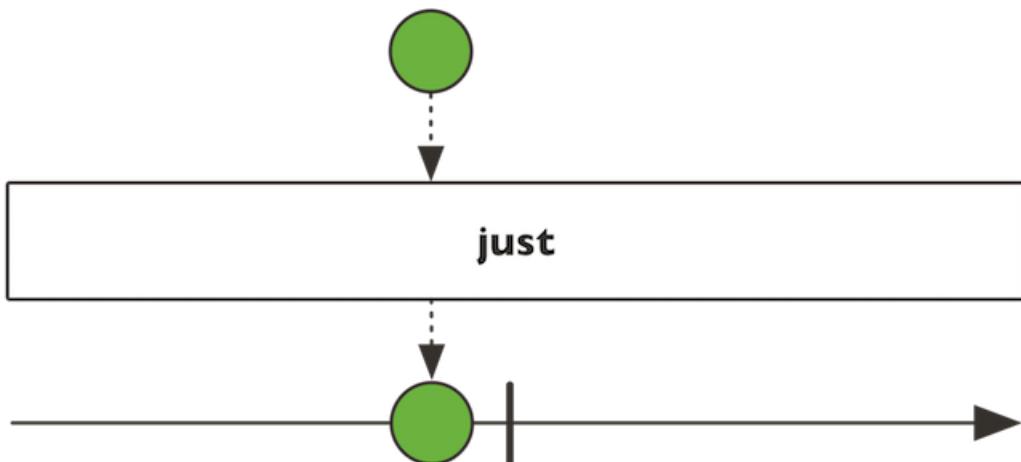
```
System.out.print("\n\nFlux Interval: ");  
ex.fluxInterval(500,3).subscribe(System.out::print);
```

Flux Interval: 012



Flux

- Creating Flux via static methods:
 - `just(T data)`
 - Creates Flux<T> which emits a single value
 - `just(T... data)`
 - Creates a Flux<T> with multiple values



Flux

- Creating Flux via static methods:

- just(T... data) important notification:

As expected,

- Flux.just(T) creates Flux<T>
- Flux.just(T1, T2, T3, ...) creates Flux<T>

BUT – assigning pre-defined array/varargs in one of the following ways:

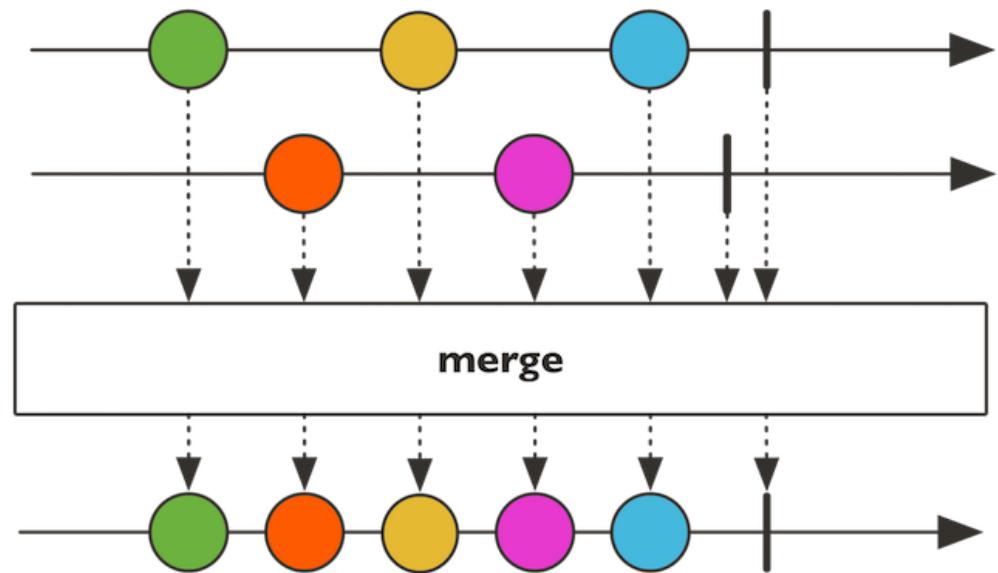
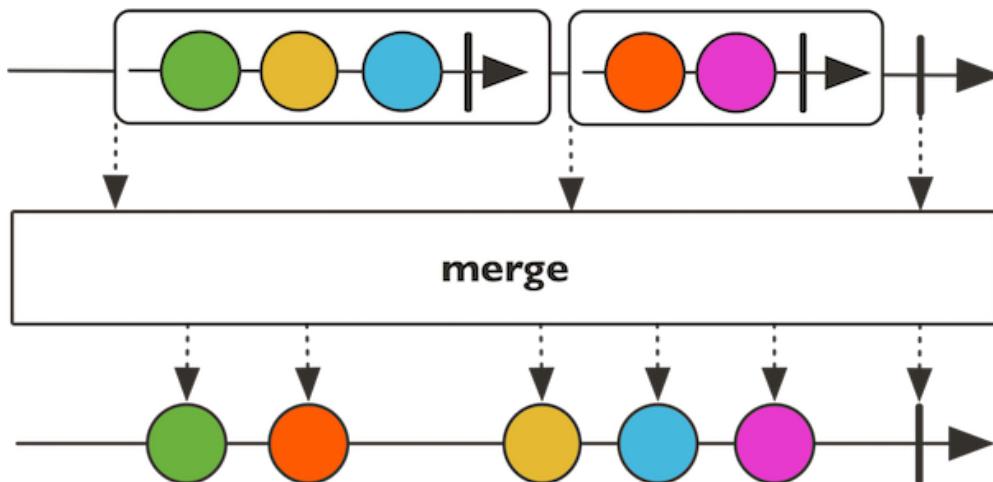
- T[] tArr = {T1, T2, T3} or
- method(T...tArr) or
- method(T[] tArr)

NOTE THAT: Flux.just(**tArr**) creates Flux<T[]>

- In order to create Flux<T> from a given array use **Flux.fromArray()** instead

Flux

- Creating Flux via static methods:
 - `merge(Publisher<T> toMerge)`
 - `merge(Iterable<Publisher<T>> toMerge)`
 - merges publisher(s) into Flux<T>
 - unlike concat() – eagerly subscribes elements
 - Fluxes should be finite



Flux

- Example:
 - `merge(...)`

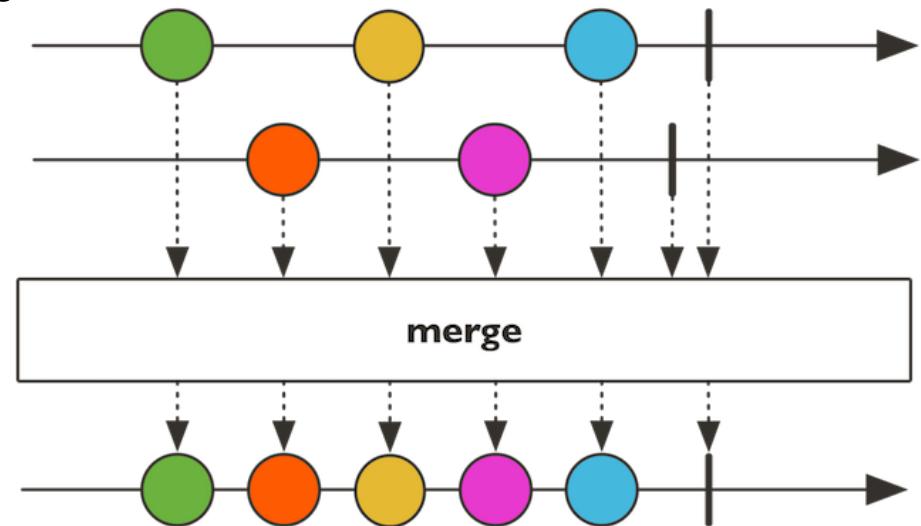
```
public Flux<Object> fluxMerge(){  
    return Flux.merge(fluxInterval(100, 10),fluxFrom().delayElements(Duration.ofMillis(100)));  
}
```

```
System.out.print("\n\nFlux Merge: ");  
ex.fluxMerge().subscribe(System.out::print);
```

```
Flux Merge: 0aaa1bbb2ccc3445678596789
```

Flux

- Creating Flux via static methods:
 - `mergeOrdered(Publisher<T>... toMerge)`
 - `mergeOrdered(Comparator<T>, Publisher<T>... toMerge)`
 - merges publisher(s) into Flux<T>
 - peeks the smallest values available on the stream first
 - unlike concat() – eagerly subscribes elements



Flux

- Example:
 - `mergeOrdered(...)`

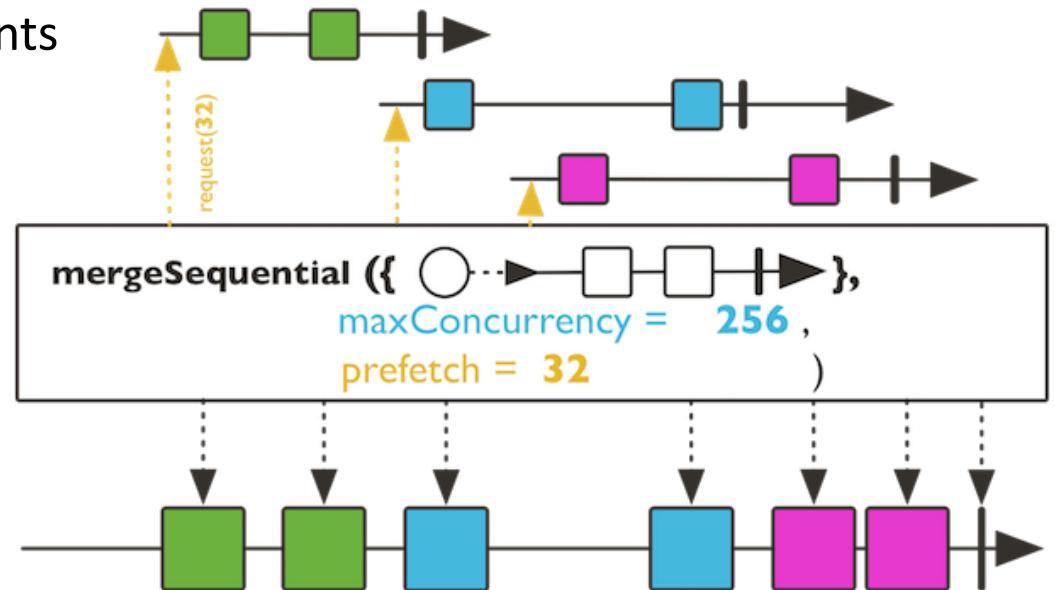
```
public Flux<Integer> fluxMergeOrdered(){  
    Integer [] random1=randomizeArray();  
    Integer [] random2=randomizeArray();  
    return Flux.mergeOrdered(Flux.just(random1).delayElements(Duration.ofMillis(200)),  
                            Flux.just(random2).delayElements(Duration.ofMillis(200)));  
}
```

```
System.out.print("\n\nFlux Merge Ordered: ");  
ex.fluxMergeOrdered().subscribe(System.out::print);
```

```
Flux Merge Ordered: 4061884466039962776130515163092163830011
```

Flux

- Creating Flux via static methods:
 - `mergeSequential(Publisher<T>... toMerge)`
 - merges publisher(s) into Flux<T>
 - emits sequentially from each Publisher according to its registration order
 - each publisher data is appended to the stream after previous publisher completes
 - unlike concat() – eagerly subscribes elements
 - Unlike merge() – doesn't mix elements



Flux

- Example:
 - `mergeSequential(...)`

```
public Flux<Object> fluxMergeSequential(){  
    return Flux.mergeSequential(fluxInterval(100, 10),fluxFrom(),fluxDefer());  
}
```

```
System.out.print("\n\nFlux Interval: ");  
ex.fluxMergeSequential(500,10).subscribe(System.out::print);
```

```
Flux Merge Sequential: 0123456789aaabbcc12345
```

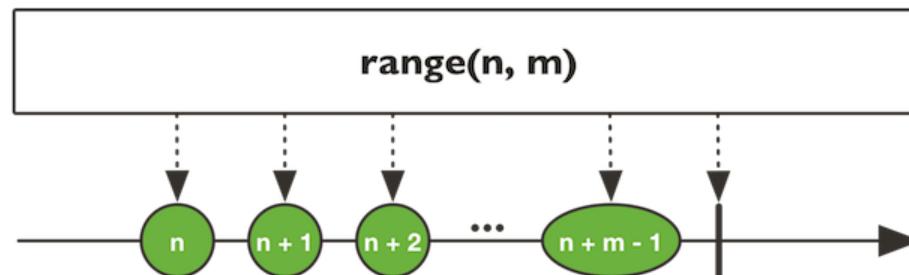
Flux

- Creating Flux via static methods:

- never()
 - never ending flux
 - signals nothing



- range(int start, int count)
 - generates Flux<Integer> with 'count' element starting from 'start' value



Flux

- Example:
 - `range(...)`

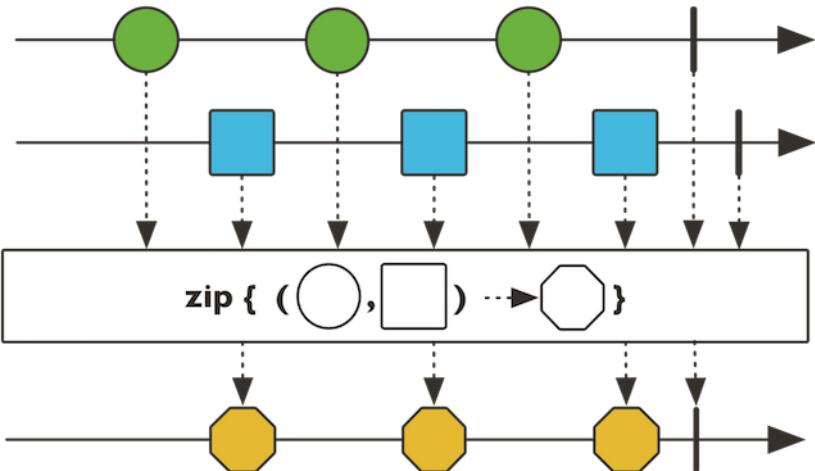
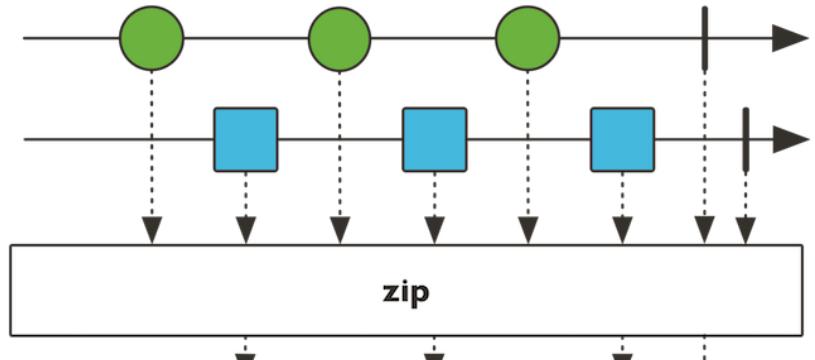
```
public Flux<Integer> fluxRange(){  
    return Flux.range(0,10);  
}
```

```
System.out.print("\n\nFlux Range: ");  
ex.fluxRange().subscribe(System.out::print);
```

```
Flux Range: 0123456789
```

Flux

- Creating Flux via static methods:
 - `zip(Publisher<T> p1, Publisher<E> p2)`
 - Fuses p1 & p2 into `Flux<Tuple2<T,E>>`
 - Overloaded for Tuple2-Tuple6
 - `zip(Function<Object[],O>, Publisher<T>...sources)`
 - Fuses Publishers<T> with combiner function into `Flux<O>`
 - `Object[]` is a set of values subscribed, coordinately, from all sources
 - `zip()` stops when shorter flux is completed and `Tuples<>` cannot be generated anymore



Flux

- Example:
 - `zip(...)`

```
public Flux<String> fluxZip(){  
    return Flux.zip((values)->{  
        String result="";  
        for(Object o:values) {  
            result+=" "+o.toString();  
        }  
        return result;  
    },Flux.range(1,3),fluxFrom());  
}
```

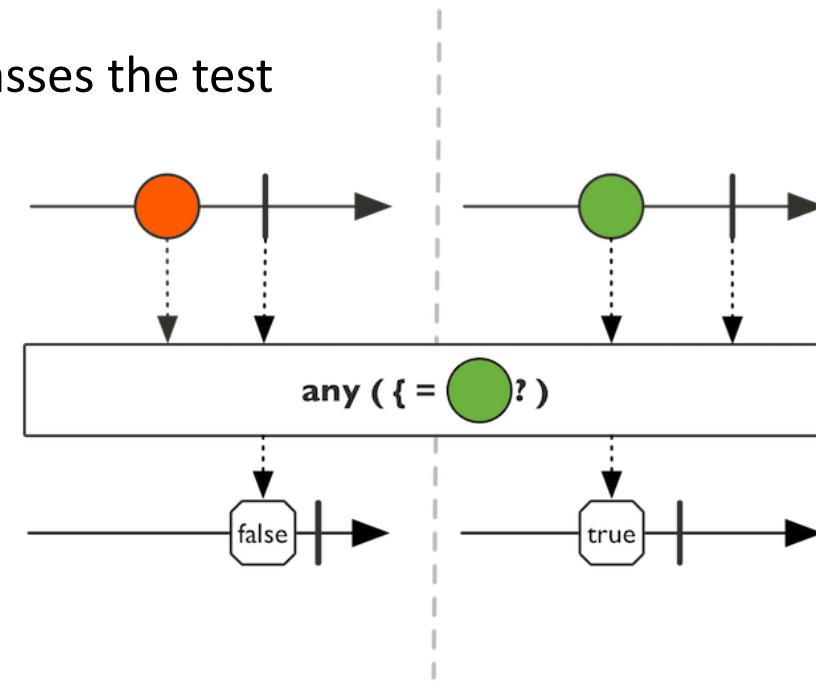
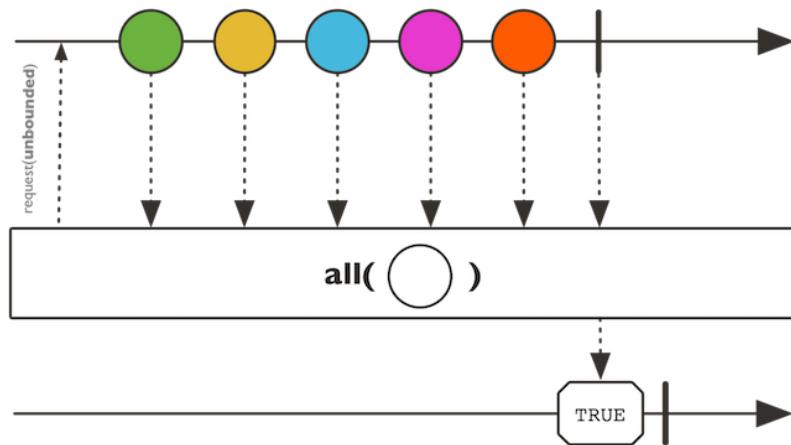
```
System.out.print("\n\nFlux Zip: \n");  
ex.fluxZip().subscribe(System.out::println);
```

```
Flux Zip:  
1 aaa  
2 bbb  
3 ccc
```

Flux

- Flux instance main methods:

- `all(Predicate<T> test)`
- `any(Predicate<T> test)`
 - results with `Mono<Boolean>`
 - true – if all/any of the items in the stream passes the test



Flux

- Flux instance main methods:
 - `as(Function<Flux<T>, P> transformer)`
 - results with P calculated from Flux<T>
 - `blockFirst()`
 - `blockFirst(Duration timeout)`
 - `blockLast()`
 - `blockLast(Duration timeout)`
 - blocks until signaling on first/last value
 - returns that value as T

Flux

- Example:
 - `as(...)`

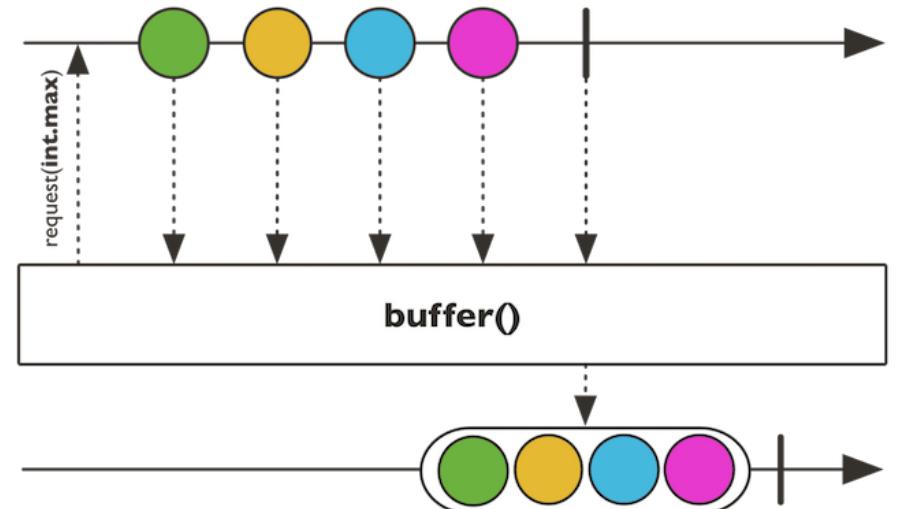
```
public Integer fluxAs(Integer...values){  
    AtomicInteger i=new AtomicInteger(0);  
    Flux.fromArray(values).as(flux->{  
        flux.doOnNext(v->i.addAndGet(v)).subscribe();  
        return 0;  
    });  
    return i.get();  
}
```

```
System.out.print("\n\nFlux As: "+ ex.fluxAs(1,2,3));
```

```
Flux As: 6
```

Flux

- Flux instance main methods:
 - `buffer(List<T>)`
 - collects items in given List
 - returns Flux<List<T>> when stream completes



Flux

- Example:
 - `buffer()`

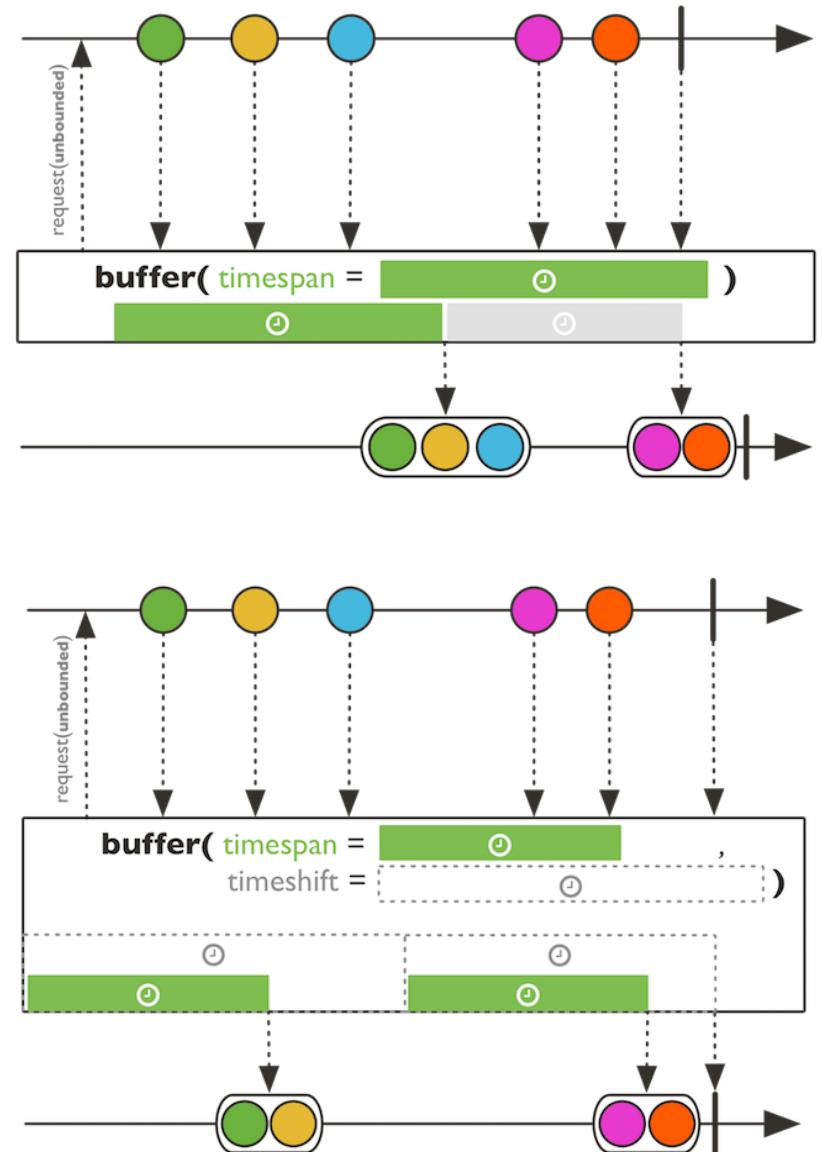
```
public Flux<List<Integer>> fluxBuffer(){  
    return Flux.range(1,9).buffer();  
}
```

```
System.out.print("\n\nFlux Buffer: \n");  
ex.fluxBuffer().subscribe(System.out::println);
```

```
Flux Buffer:  
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Flux

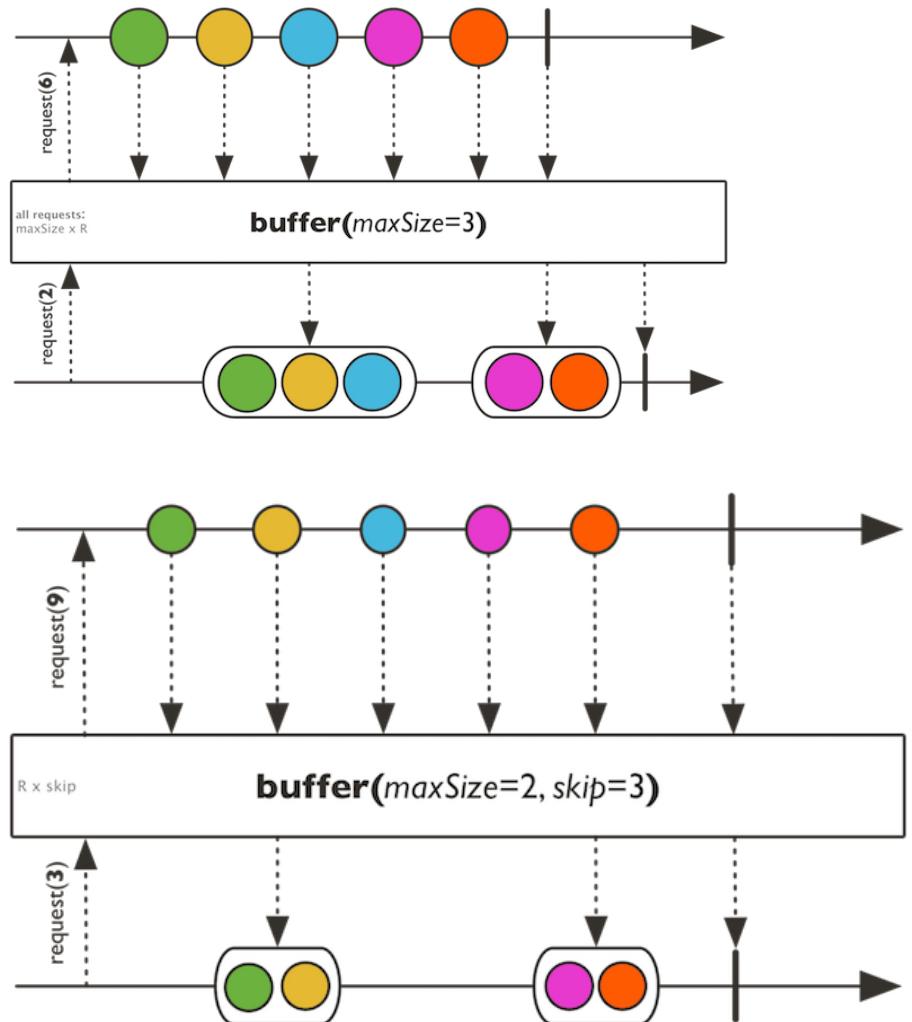
- Flux instance main methods:
 - **buffer(Duration timespan)**
 - collects items for given timespan
 - **buffer(Duration timespan, Duration timeshift)**
 - collects items for given timespan each time-shift
- returns Flux<List<T>>
- List<T> is signaled on expiration or when stream completes



Flux

- Flux instance main methods:

- `buffer(int maxSize)`
- `buffer(int maxSize, int skip)`
 - returns multiple `Flux<List<T>>` when ‘maxSize’ items collected or when stream completes
 - ‘skip’ specifies how many items to count before starting a new buffer



Flux

- Example:
 - `buffer(...)`

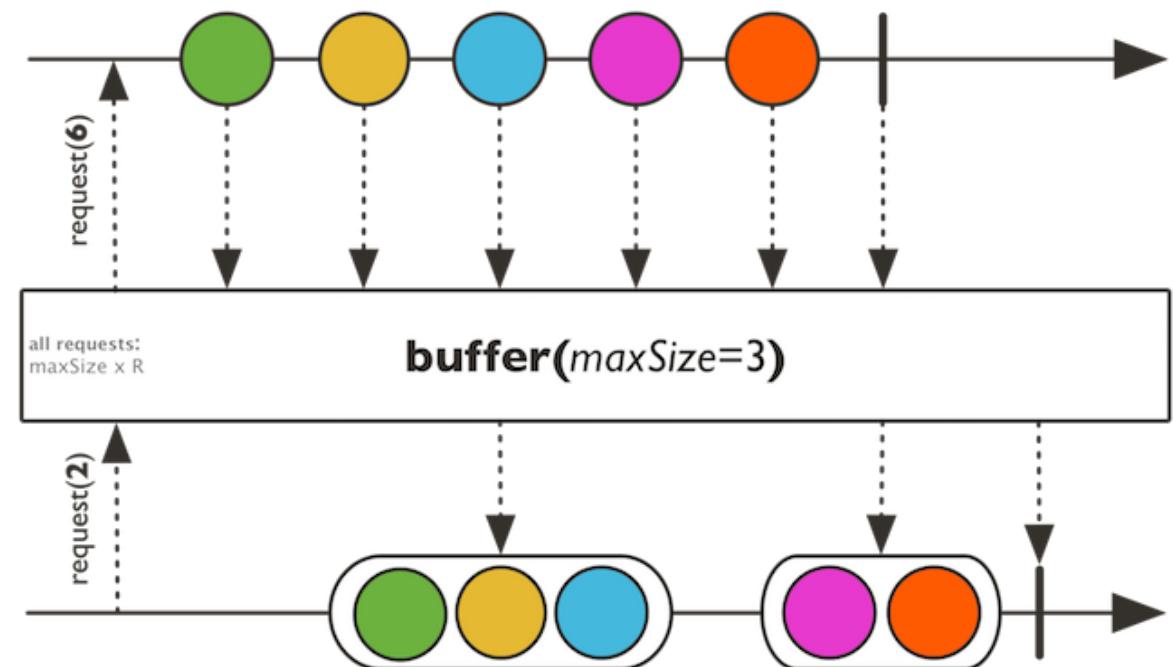
```
public Flux<List<Integer>> fluxBufferMaxSize(){  
    return Flux.range(1,14).buffer(5);  
}
```

```
System.out.print("\nFlux Buffer Max Size: \n");  
ex.fluxBufferMaxSize().subscribe(System.out::print);
```

```
Flux Buffer Max Size:  
[1, 2, 3, 4, 5][6, 7, 8, 9, 10][11, 12, 13, 14]
```

Flux

- Flux instance main methods:
 - `bufferUntil(Predicate<T> test)`
 - returns multiple `Flux<List<T>>`
 - each Flux is returned when item passes given test or when stream completes



Flux

- Example:
 - `bufferUntil(...)`

```
public Flux<List<Integer>> fluxBufferUntil(){  
    return Flux.fromArray(randomizeArray()).bufferUntil(n->n==1 || n==9);  
}
```

```
System.out.print("\n\nFlux Buffer Until: \n");  
Flux<List<Integer>> fl=ex.fluxBufferUntil();  
fl.subscribe(System.out::print);
```

```
Flux Buffer Until:  
[0, 2, 5, 1][9][8, 3, 2, 4, 5, 4, 9][8, 6, 6, 2, 6, 8, 6, 1]
```

Flux

- Flux instance main methods:
 - share()
 - Streams elements once
 - Starts on first subscriber
 - Later subscribers might miss elements

Flux

- Example:
 - share(...)

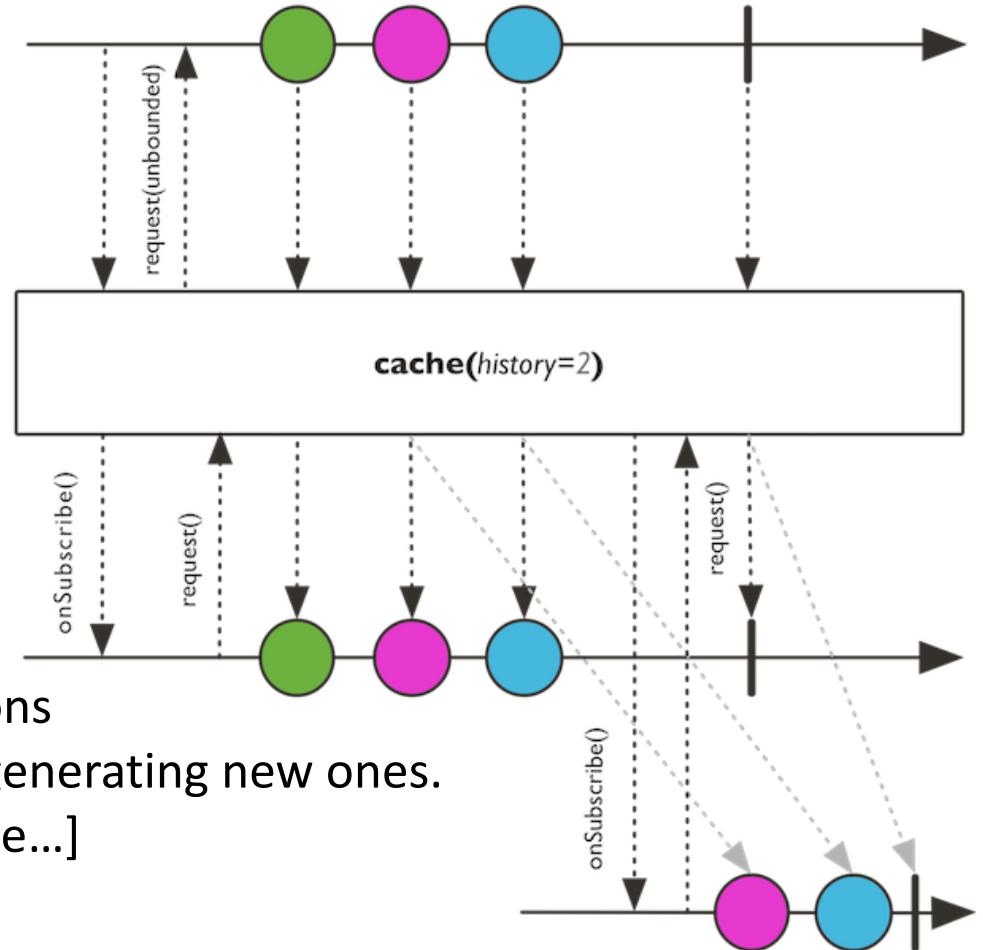
```
public void fluxShare() {  
    Flux<Long> startFlux = Flux.interval(Duration.ofSeconds(1)).take(10).share();  
    Flux firstFlux = Flux.from(startFlux);  
    firstFlux.subscribe(out -> System.out.println("firstFlux value: " + out));  
    try {Thread.sleep(5000);}catch (Exception e) {}  
    Flux secondFlux = Flux.from(startFlux);  
    secondFlux.subscribe(out -> System.out.println("secondFlux value: " + out));  
}
```

```
Flux Share  
firstFlux value: 0  
firstFlux value: 1  
firstFlux value: 2  
firstFlux value: 3  
firstFlux value: 4  
firstFlux value: 5  
secondFlux value: 5  
firstFlux value: 6  
secondFlux value: 6  
firstFlux value: 7  
secondFlux value: 7  
firstFlux value: 8  
secondFlux value: 8  
firstFlux value: 9  
secondFlux value: 9
```

Flux

- Flux instance main methods:

- cache()
- cache(int history)
- cache(Duration d)
 - returns Flux<T> which caches items (all/history/by duration) from origin Flux
 - Uses internal cache
 - Cached elements are not calculated for example – streaming data from connections may use the same connections instead of regenerating new ones.
[like: Stock-Quote fetcher, Shareable Resource...]



Flux

- Example:
 - cache()

```
public void fluxShare() {  
    Flux<Long> startFlux = Flux.interval(Duration.ofSeconds(1)).take(10).share().cache();  
    Flux firstFlux = Flux.from(startFlux);  
    firstFlux.subscribe(out -> System.out.println("firstFlux value: " + out));  
    try {Thread.sleep(5000);}catch (Exception e) {}  
    Flux secondFlux = Flux.from(startFlux);  
    secondFlux.subscribe(out -> System.out.println("secondFlux value: " + out));  
}
```

```
Flux Cache - All  
Elements  
firstFlux value: 0  
firstFlux value: 1  
firstFlux value: 2  
firstFlux value: 3  
firstFlux value: 4  
secondFlux value: 0  
secondFlux value: 1  
secondFlux value: 2  
secondFlux value: 3  
secondFlux value: 4  
firstFlux value: 5  
secondFlux value: 5  
firstFlux value: 6  
secondFlux value: 6  
firstFlux value: 7  
secondFlux value: 7  
firstFlux value: 8  
secondFlux value: 8  
firstFlux value: 9  
secondFlux value: 9
```

Flux

- Example:
 - `cache(...)`

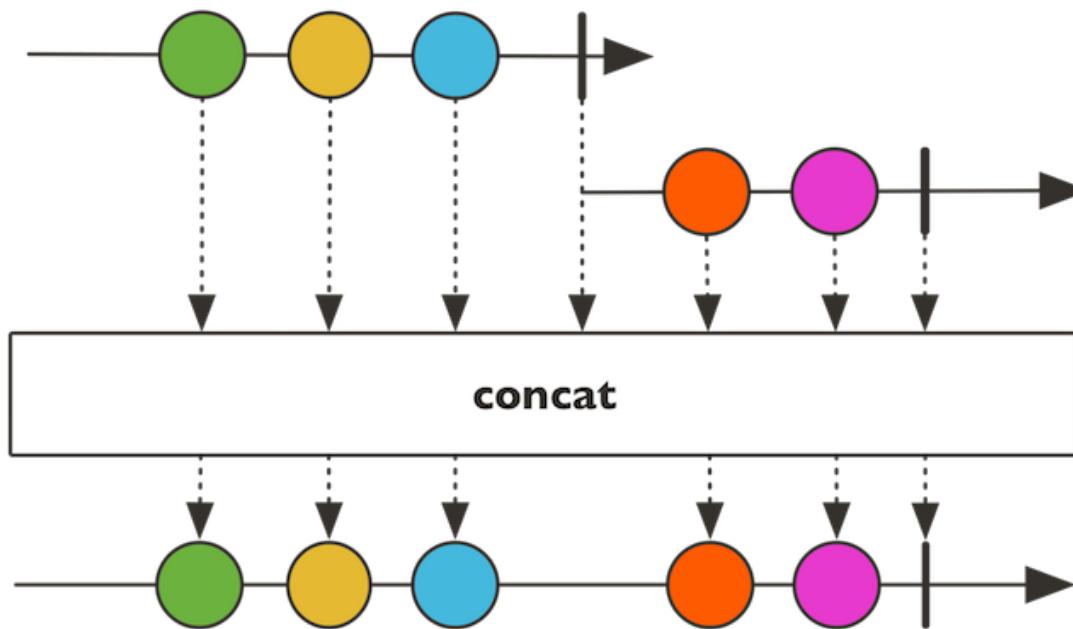
```
public void fluxShare() {  
    Flux<Long> startFlux = Flux.interval(Duration.ofSeconds(1)).take(10).share().cache(2);  
    Flux firstFlux = Flux.from(startFlux);  
    firstFlux.subscribe(out -> System.out.println("firstFlux value: " + out));  
    try {Thread.sleep(5000);}catch (Exception e) {}  
    Flux secondFlux = Flux.from(startFlux);  
    secondFlux.subscribe(out -> System.out.println("secondFlux value: " + out));  
}
```

Flux Cache - Last 2 Elements
firstFlux value: 0
firstFlux value: 1
firstFlux value: 2
firstFlux value: 3
firstFlux value: 4
secondFlux value: 3
secondFlux value: 4
firstFlux value: 5
secondFlux value: 5
firstFlux value: 6
secondFlux value: 6
firstFlux value: 7
secondFlux value: 7
firstFlux value: 8
secondFlux value: 8
firstFlux value: 9
secondFlux value: 9

Flux

- Flux instance main methods:

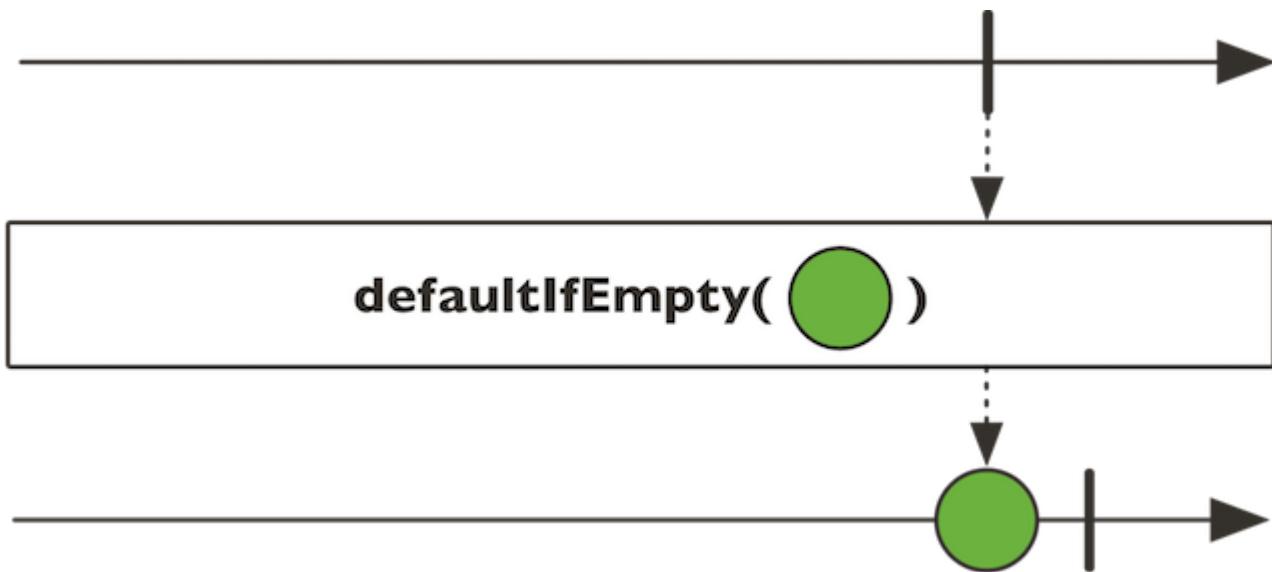
- concatWith(Publisher<T> source)
 - appends source to this Flux (after origin completes)
 - returns Flux<T>



Flux

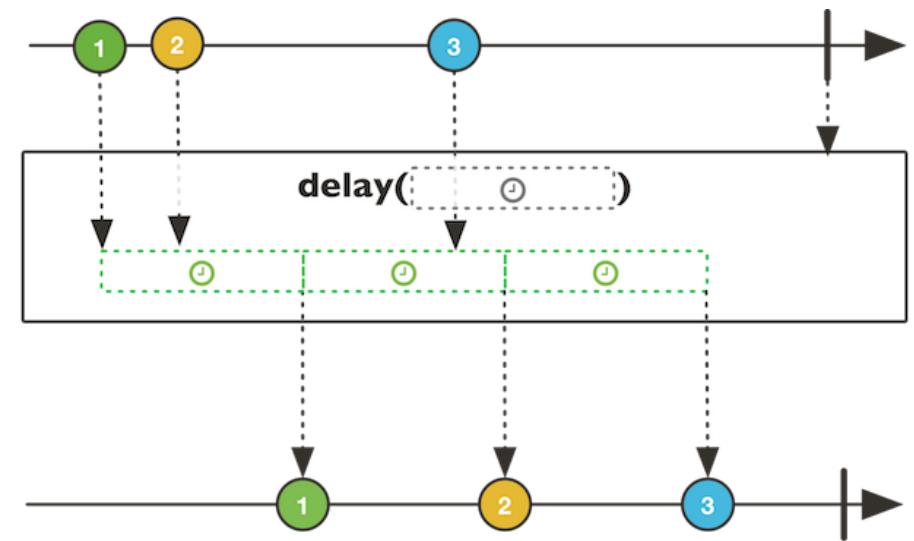
- Flux instance main methods:

- `defaultIfEmpty(T default)`
 - returns T if Flux is empty



Flux

- Flux instance main methods:
 - `delayElements(Duration delay)`
 - `delayElements(Duration delay, Scheduler timer)`
 - Delays items subscription according to given delay



Flux

- Example:
 - `delayElements(...)`

```
public Flux<Integer> fluxDelay(){  
    return Flux.range(1,10).delayElements(Duration.ofMillis(100));  
}
```

```
System.out.print("\n\nFlux Delay Elements: \n");  
ex.fluxDelay().subscribe(System.out::print);
```

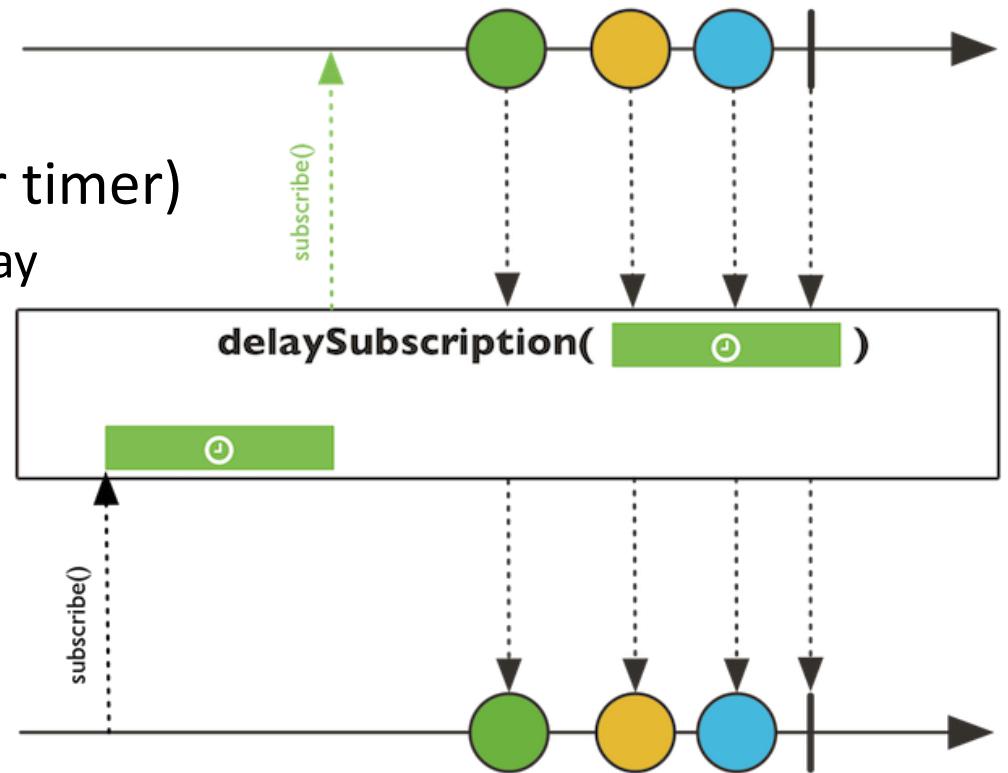
Flux Delay Elements:

12345678910



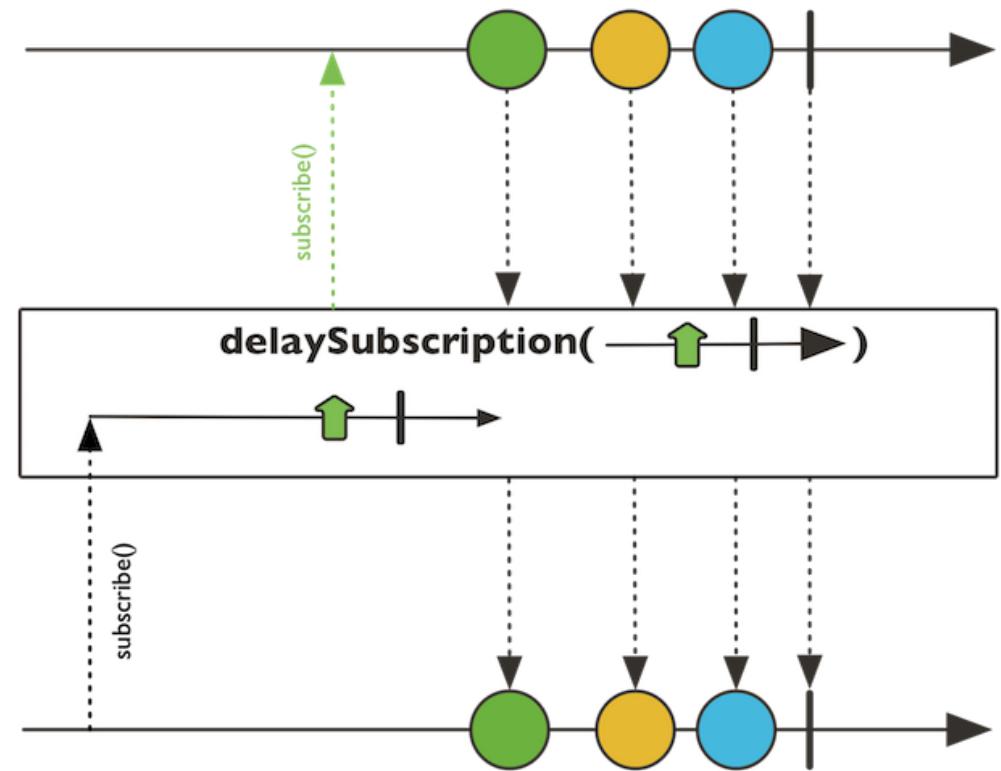
Flux

- Flux instance main methods:
 - `delaySubscription(Duration delay)`
 - `delaySubscription(Duration delay, Scheduler timer)`
 - blocks items subscription according to given delay



Flux

- Flux instance main methods:
 - `delaySubscription(Publisher<T> source)`
 - delays subscription until given publisher signals a value or completes

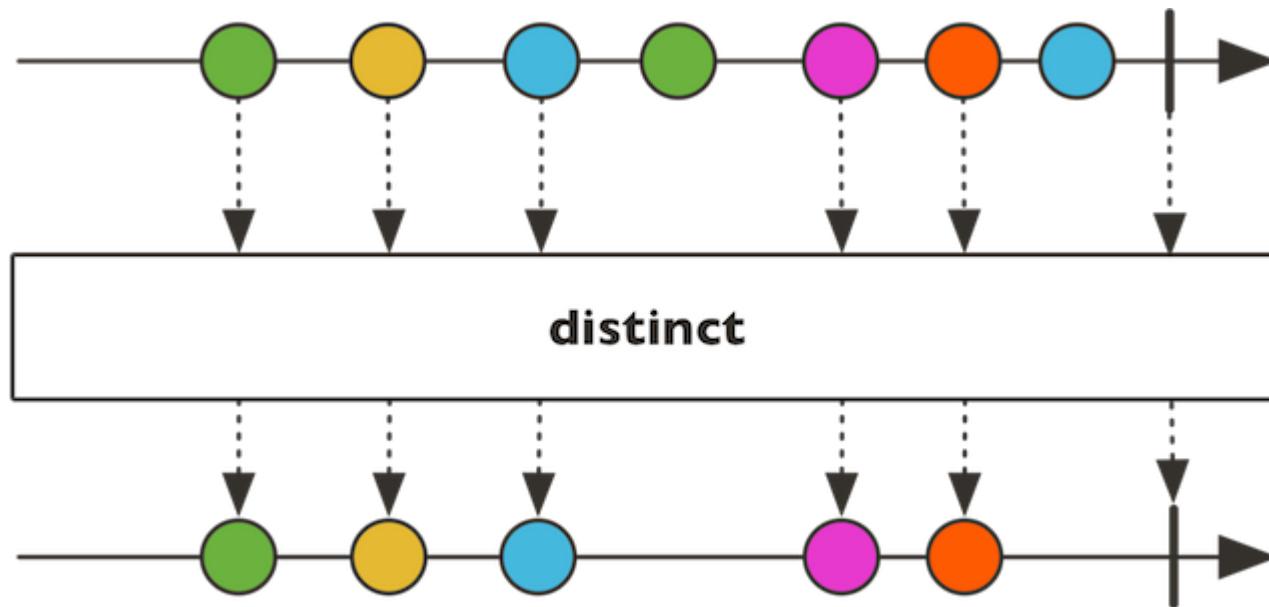


Flux

- Flux instance main methods:

- `distinct()`

- filters duplications for each subscriber



Flux

- Example:
 - `distinct(...)`

```
public Flux<Integer> fluxDistinct(){  
    Flux<Integer> origin=Flux.fromArray(randomizeArray());  
    System.out.print("Origin: ");  
    origin.subscribe(System.out::print);  
    System.out.println();  
    return origin.distinct();  
}
```

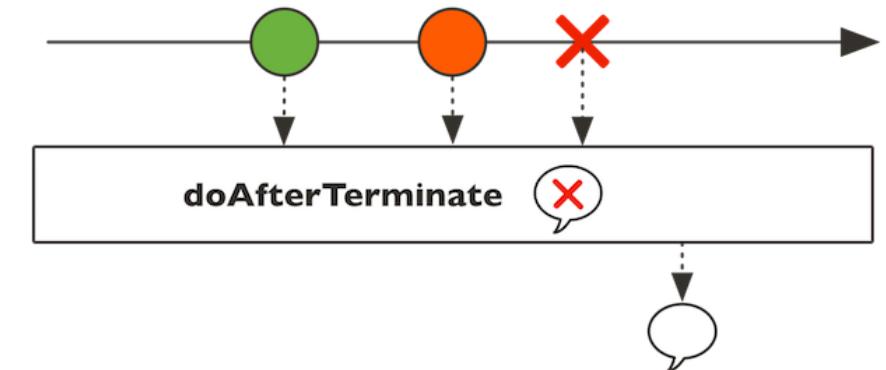
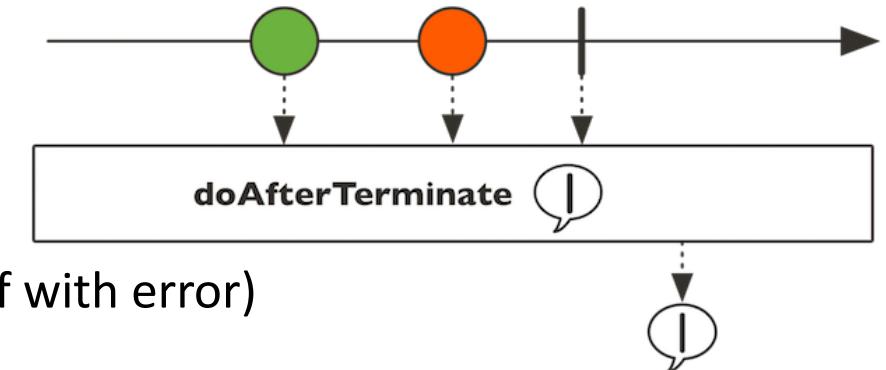
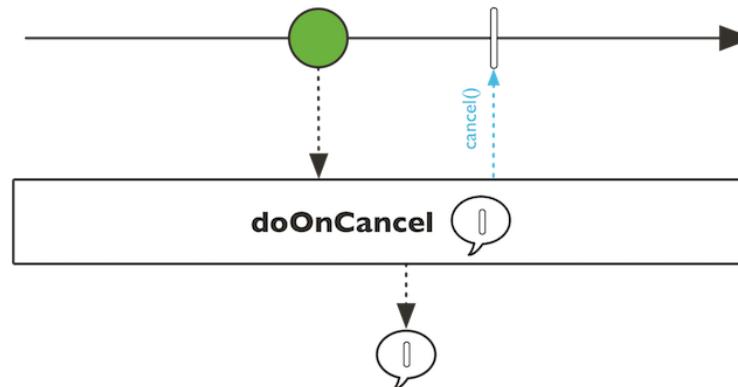
```
System.out.print("\n\nFlux Distinct: \n");  
ex.fluxDistinct().subscribe(System.out::print);
```

```
Flux Distinct:  
Origin: 10972391894345961675  
1097238456
```

Flux

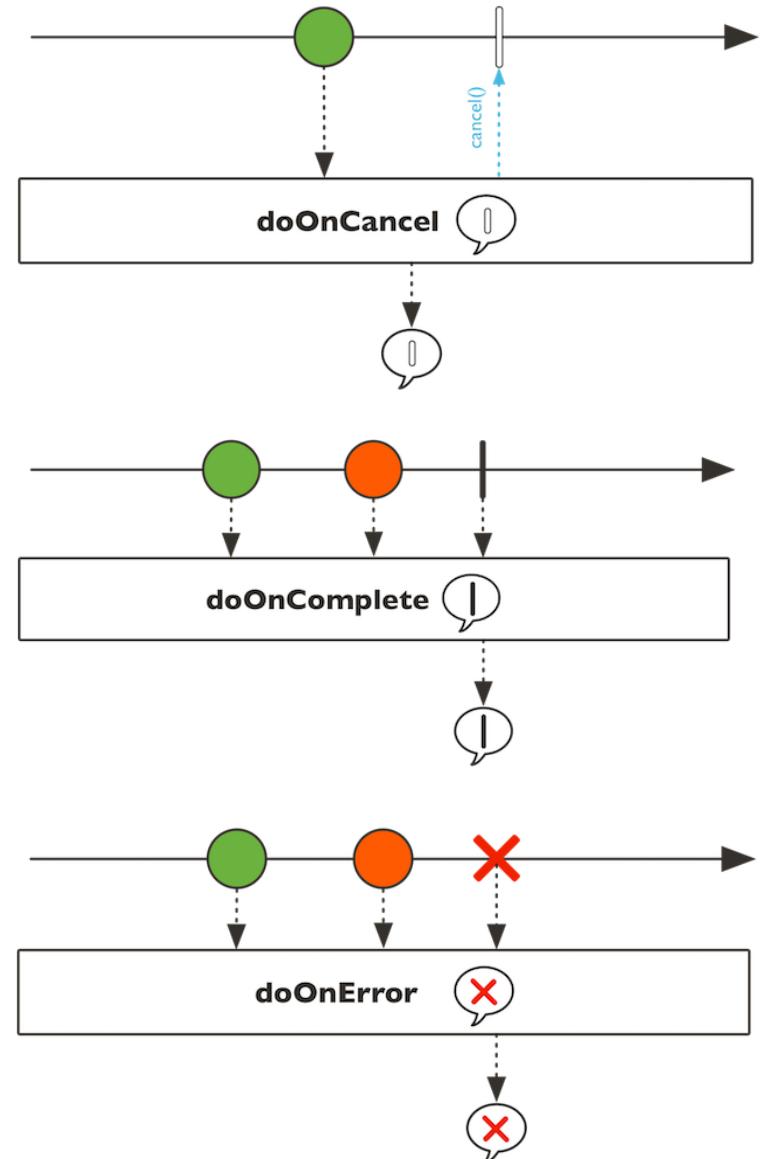
- Flux instance main methods:

- `doAfterTerminate(Runnable task)`
 - executes task after stream completes (successfully or with error)
- `doOnCancel(Runnable task)`
 - executes task when flux is canceled



Flux

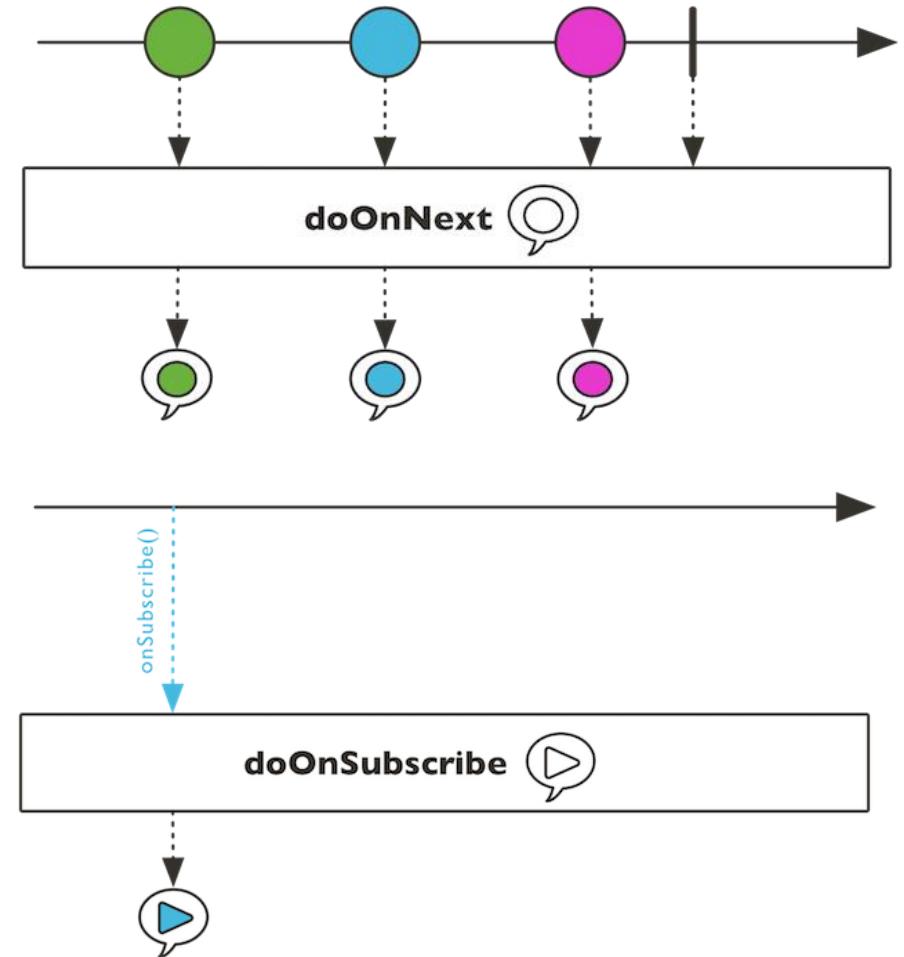
- Flux instance main methods:
 - **doOnCancel(Runnable task)**
 - executes task when flux is canceled
 - **doOnComplete(Runnable task)**
 - executes task when flux is completed successfully
 - **doOnError(Consumer<Throwable> error)**
 - passes error signal to given consumer



Flux

- Flux instance main methods:
 - `doOnNext(Consumer<Signal<T>> signal)`
 - passes onNext signals to given consumer
 - `doOnSubscribe(Consumer<Signal<T>> signal)`
 - passes onSubscribe signal to given consumer
 - `doOnEach(Consumer<Signal<T>> signal)`
 - passes any kind of signal to given consumer
 - signals: onNext, complete, error

T (item) can be recovered via `Signal.get()`



Flux

- Example:
 - doOnEach(...)

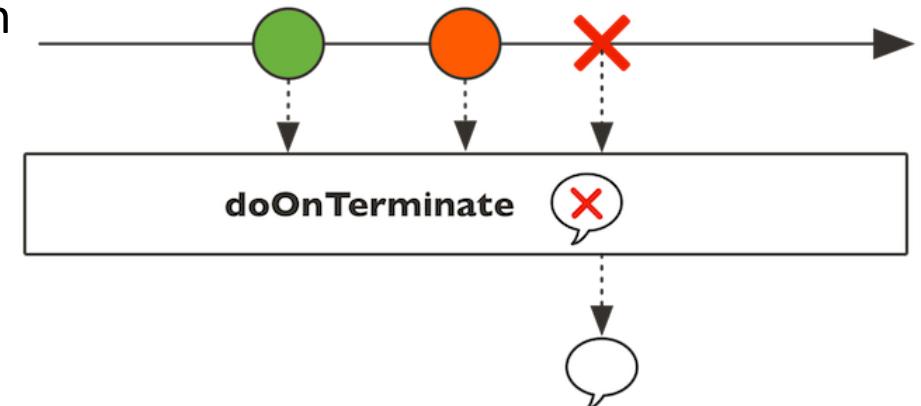
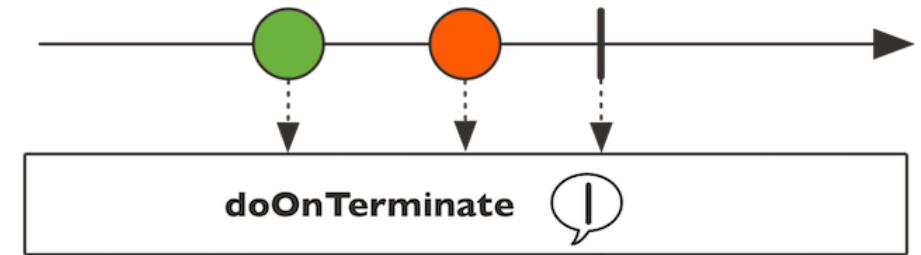
```
public Flux<Integer> fluxDoOnEach(){  
    return Flux.range(1,10).doOnEach(signal->  
    {  
        if(signal.isOnNext())  
            System.out.print("_");  
        if(signal.isOnComplete())  
            System.out.print(" Done!");  
    });  
}
```

```
System.out.print("\n\nFlux Do On Each: \n");  
ex.fluxDoOnEach().subscribe(System.out::print);
```

```
Flux Do On Each:  
_1_2_3_4_5_6_7_8_9_10 Done!
```

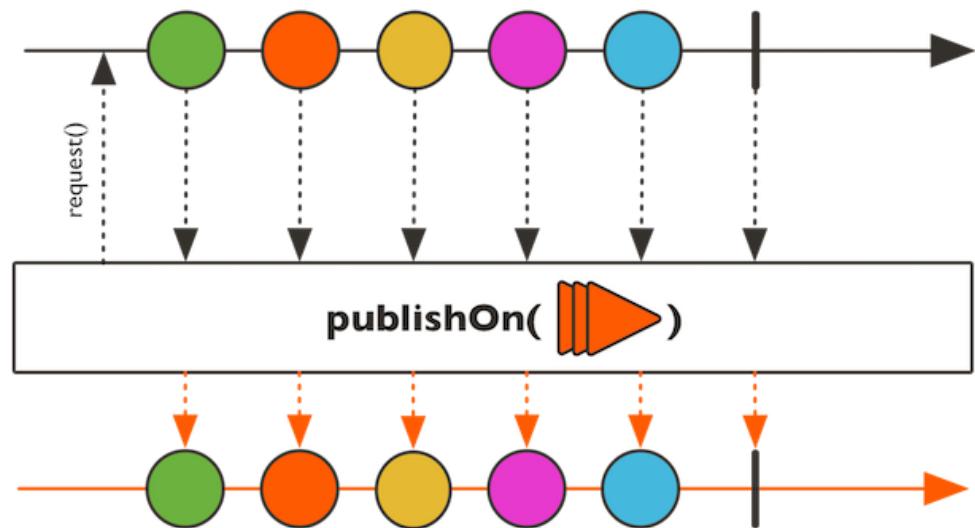
Flux

- Flux instance main methods:
 - `doOnTerminate(Runnable task)`
 - executes task on successful or error termination
 - `doFinally(Consumer<SignalType> signal)`
 - sends signal-type on successful or error termination



Flux

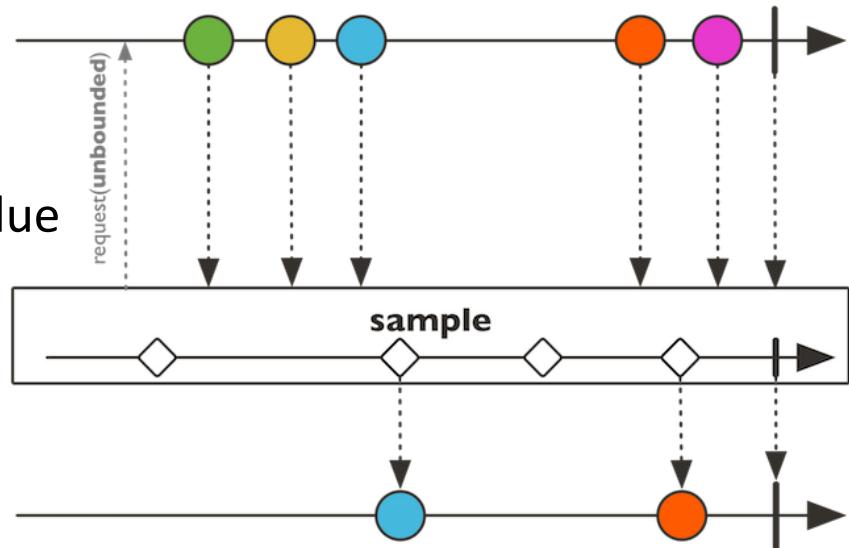
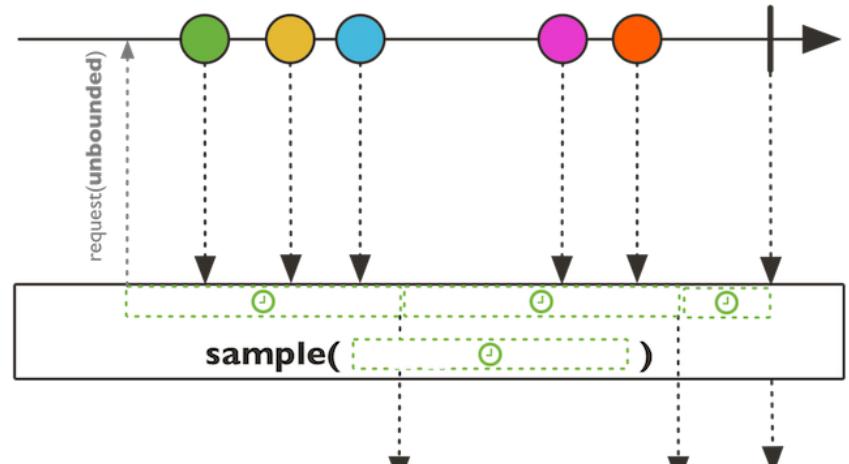
- Flux instance main methods:
 - `publishOn(Scheduler scheduler)`
 - `publishOn(Scheduler scheduler, int prefetch)`
 - Starts publishing according to scheduler



Flux

- Flux instance main methods:

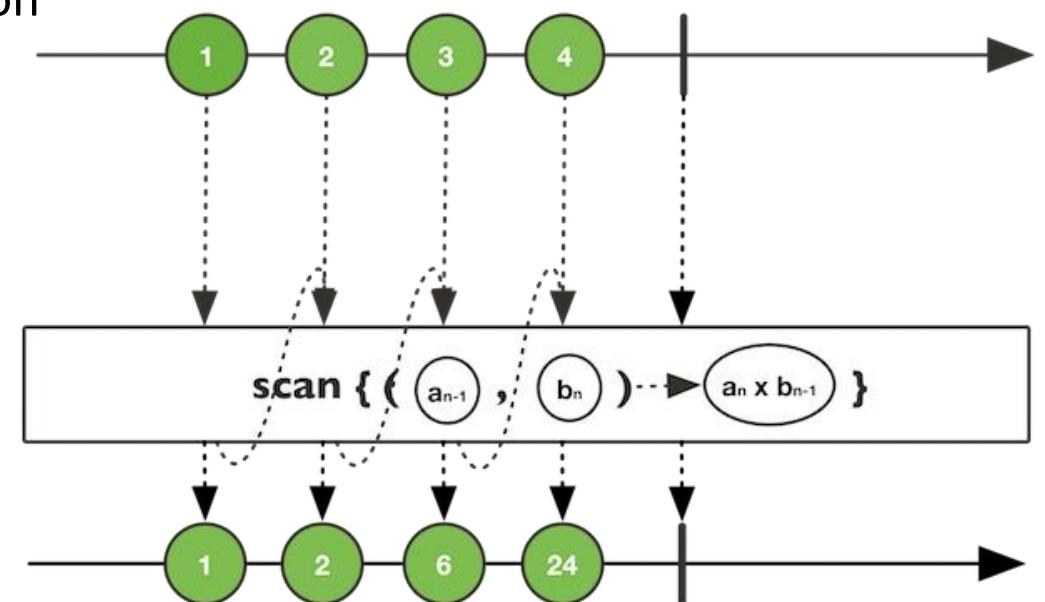
- **sample(Duration timespan)**
 - returns Flux<T> with latest emitted value within the periodical time window.
- **sample(publisher<U> sampler)**
 - returns Flux<T> with latest emitted value in the current Flux<T> since sampler (Flux<U>) signaled a value
 - when one terminates - so does the other



Flux

- Flux instance main methods:

- `scan(BiFunction<T,T,T> reducer)`
 - reduces every 2 last elements of current `Flux<T>` into new `Flux<T>` using given reducer function
 - function reduces `T & T` into `T`



Flux

- Example:
 - `scan(...)`

```
public Flux<Integer> fluxScan(){  
    return Flux.range(0,11).scan((n1,n2)->n1+n2);  
}
```

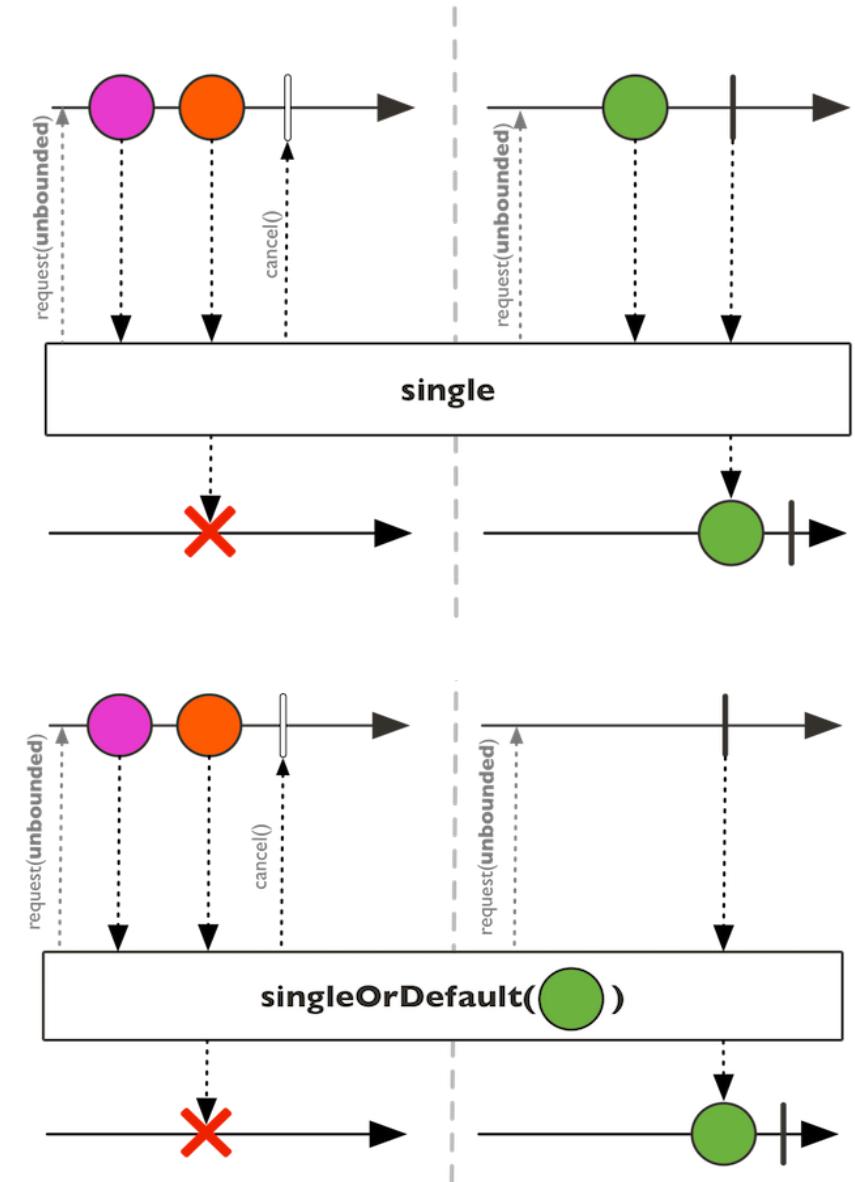
```
System.out.print("\n\nFlux Scan: \n");  
ex.fluxScan().subscribe(System.out::println);
```

Flux Scan:

0
1
3
6
10
15
21
28
36
45
55

Flux

- Flux instance main methods:
 - `single()`
 - returns `Mono<T>` when single value is emitted
 - `Mono<T>` contains emitted `T` value or null if empty
 - `single(T value)`
 - returns `Mono<T>` when single value is emitted
 - returns `T` value when `Flux<T>` ends empty
 - both result in `IndexOutOfBoundsException` if the stream has more than one element



Flux

- Example:
 - `single(...)`

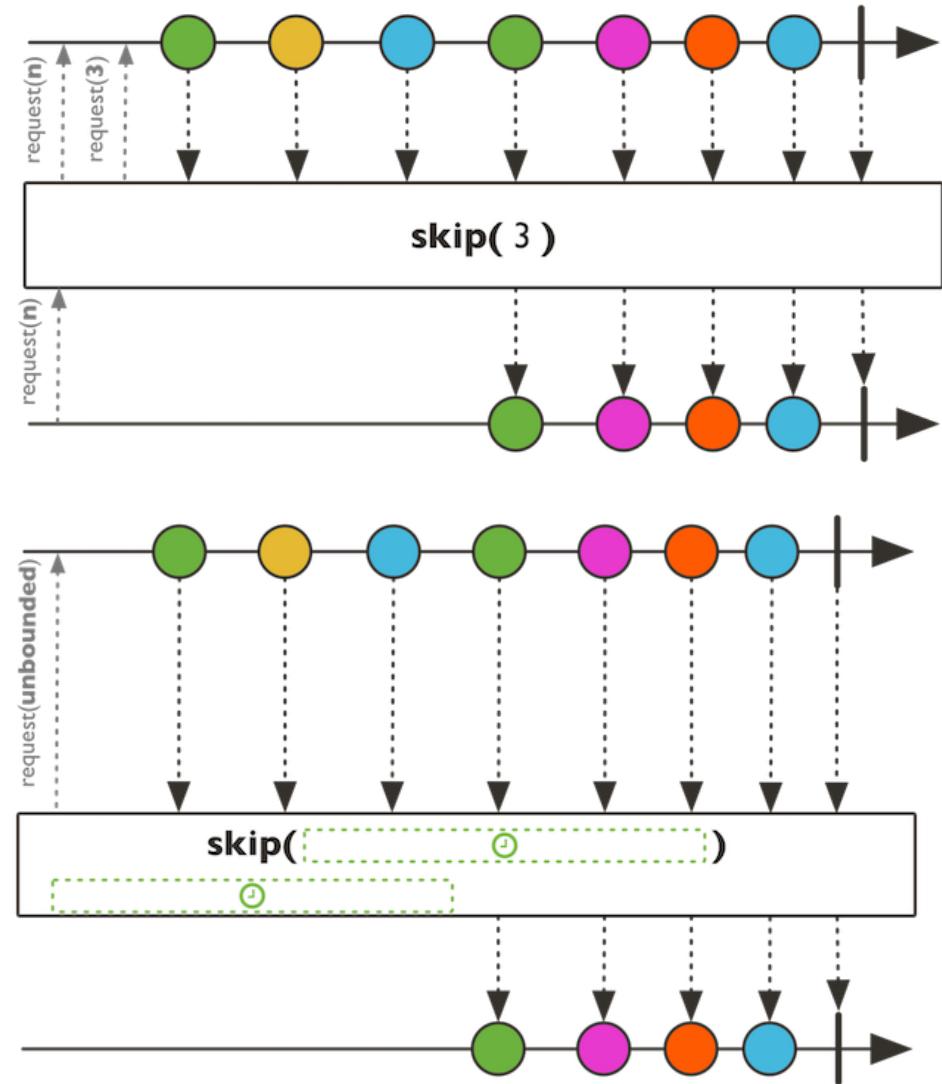
```
public Mono<Integer> fluxSingle(Integer...values){  
    return Flux.fromArray(values).single(100);  
}
```

```
System.out.print("\n\nFlux Single: \n");  
ex.fluxSingle().subscribe(System.out::println);  
ex.fluxSingle(50).subscribe(System.out::println);
```

```
Flux Single:  
100  
50
```

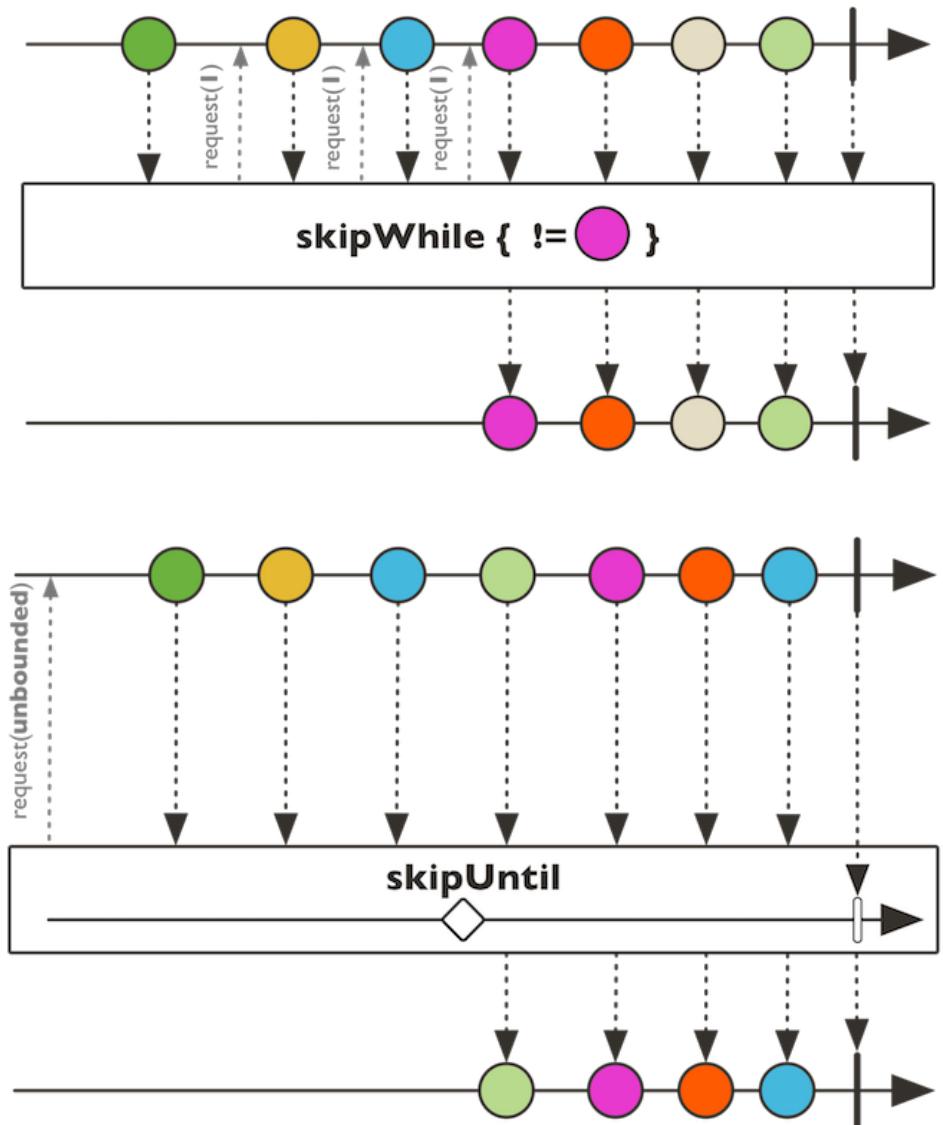
Flux

- Flux instance main methods:
 - **skip(long skipped)**
 - skips specified number of items from the beginning of the stream and emits the rest
 - **skip(Duration timespan)**
 - **skip(Duration timespan, Scheduler timer)**
 - skips items signaled during timespan



Flux

- Flux instance main methods:
 - `skipWhile(Predicate<T> test)`
 - skips items from this stream while test returns ‘true’ for a value
 - `skipUntil(Predicate<T> test)`
 - skips items from this stream until test returns ‘true’ for a value
 - `skipUntilOther(Publisher<?> other)`
 - skips items from this stream until ‘other’ signals `onNext` / `onComplete`



Flux

- Example:
 - `skip(...)`

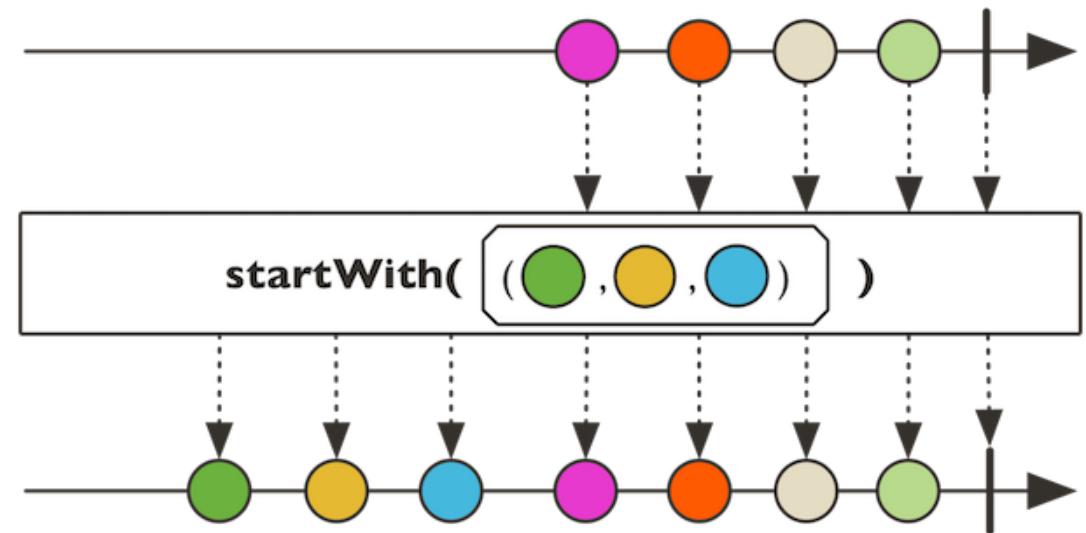
```
public Flux<Integer> fluxSkip(int startFrom){  
    return Flux.range(0,10).skip(startFrom);  
}
```

```
System.out.print("\n\nFlux Skip: \n");  
ex.fluxSkip(5).subscribe(System.out::print);
```

```
Flux Skip:  
56789
```

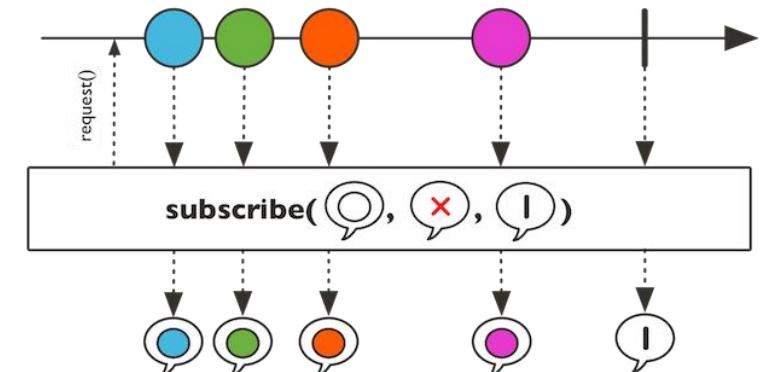
Flux

- Flux instance main methods:
 - `sort()`
 - `sort(Comparator<T> comparator)`
 - returns a sorted Flux<T>
 - `startWith(Iterable<T> startItems)`
 - `startWith(T... startItems)`
 - `startWith(Publisher<T> startItems)`
 - emits startItems before streaming values



Flux

- Flux instance main methods:
 - `subscribe()`
 - returns Disposable when no further processing is required (`Disposable.dispose()`)
 - `subscribe(Consumer<T> handler)`
 - `subscribe(Consumer<T> handler, Consumer<Throwable> errHandler)`
 - `subscribe(Consumer<T> handler, Consumer<Throwable> errHandler, Runnable task)`
 - handler – consumes T stream values
 - errHandler – consumes error termination
 - task – is executed on completion



Flux

- Example:
 - `subscribe(...)` & `Disposable.dispose()`

```
public void fluxDispose(){  
    Disposable d=Flux.interval(Duration.ofMillis(100)).subscribe(System.out::print);  
    try {Thread.sleep(1000);}catch (Exception e) {}  
    d.dispose();  
}
```

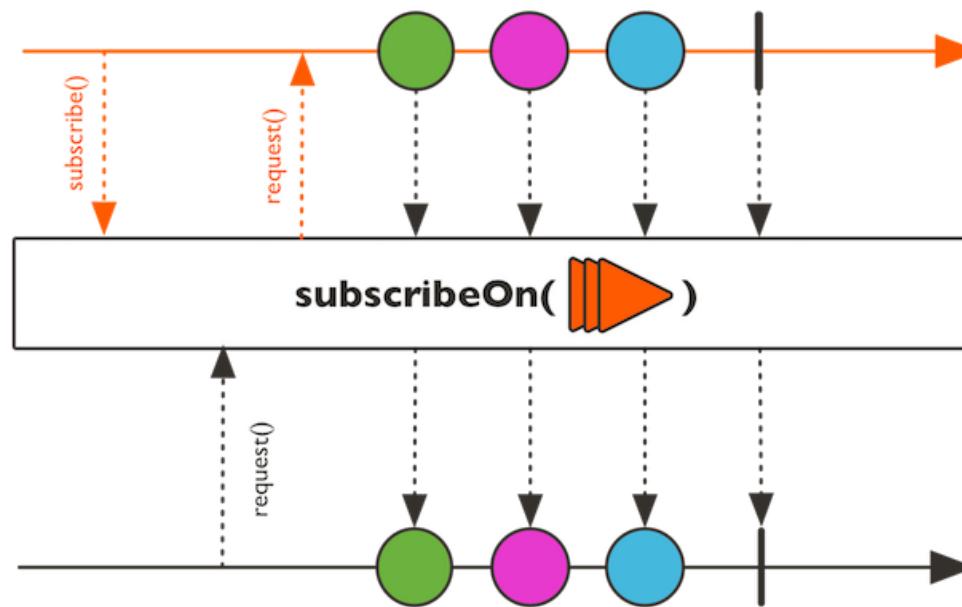
```
System.out.print("\n\nFlux Dispose Subscription: \n");  
ex.fluxDispose();
```

```
Flux Dispose Subscription:  
0123456789
```

Flux

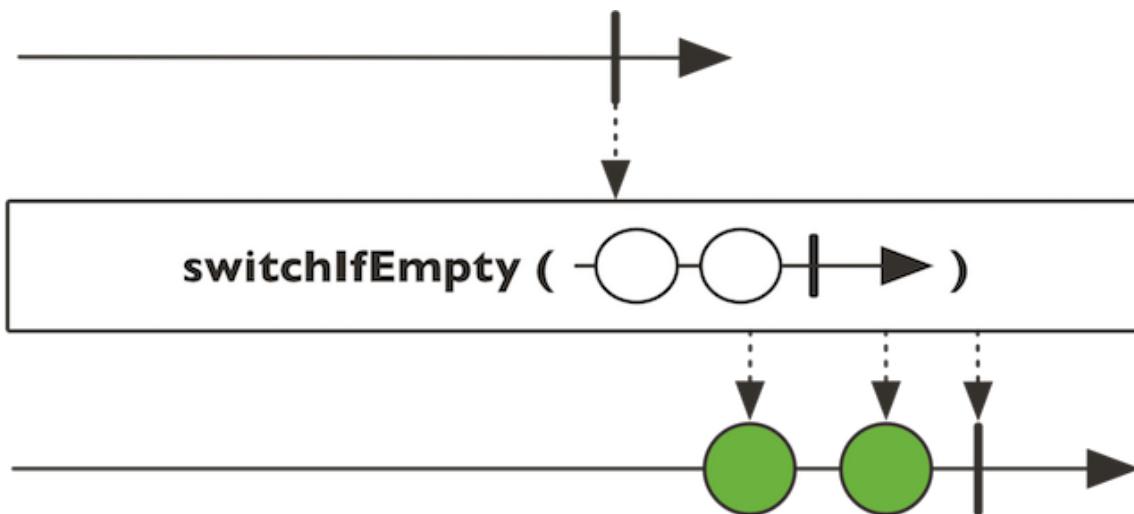
- Flux instance main methods:

- `subscribeOn(Scheduler scheduler)`
 - starts subscription according to scheduler



Flux

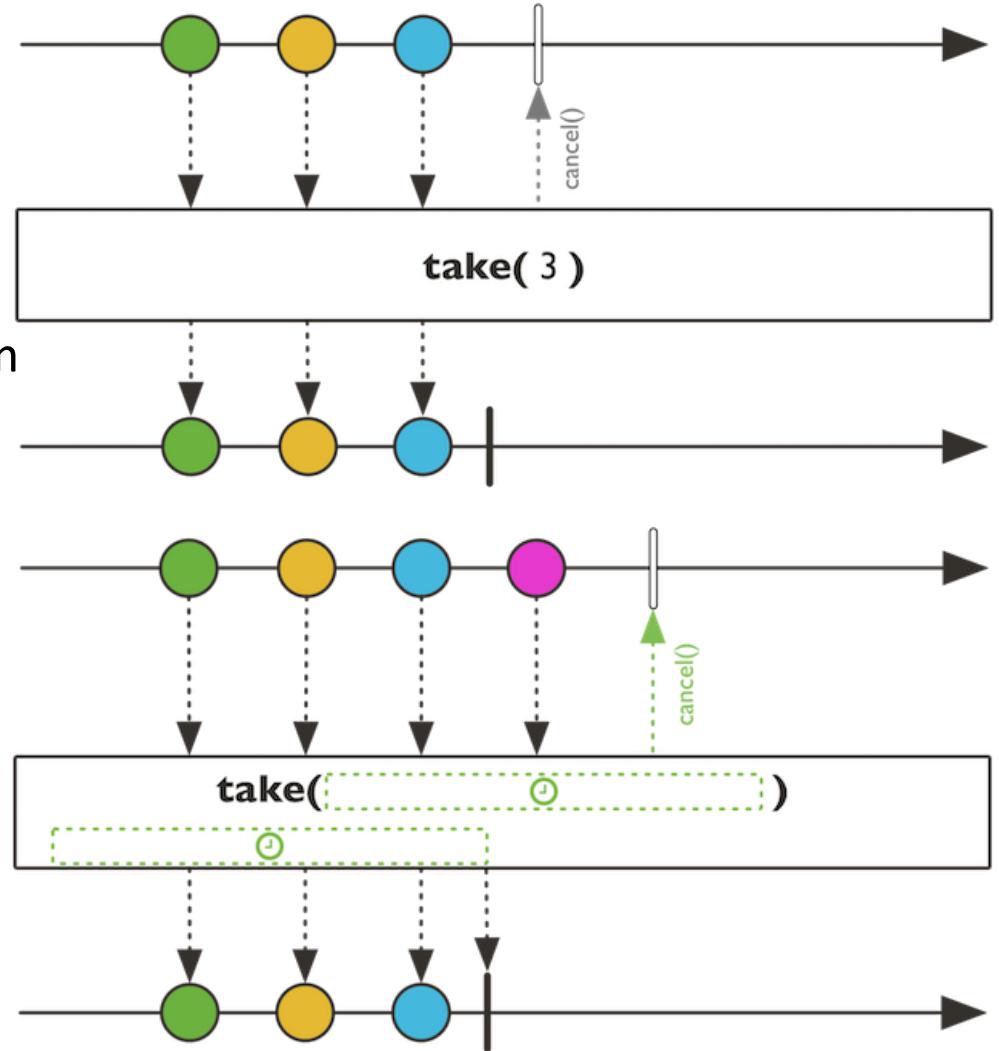
- Flux instance main methods:
 - `switchIfEmpty(Publisher<T> alternate)`
 - Switches to alternate stream if current stream terminates empty



Flux

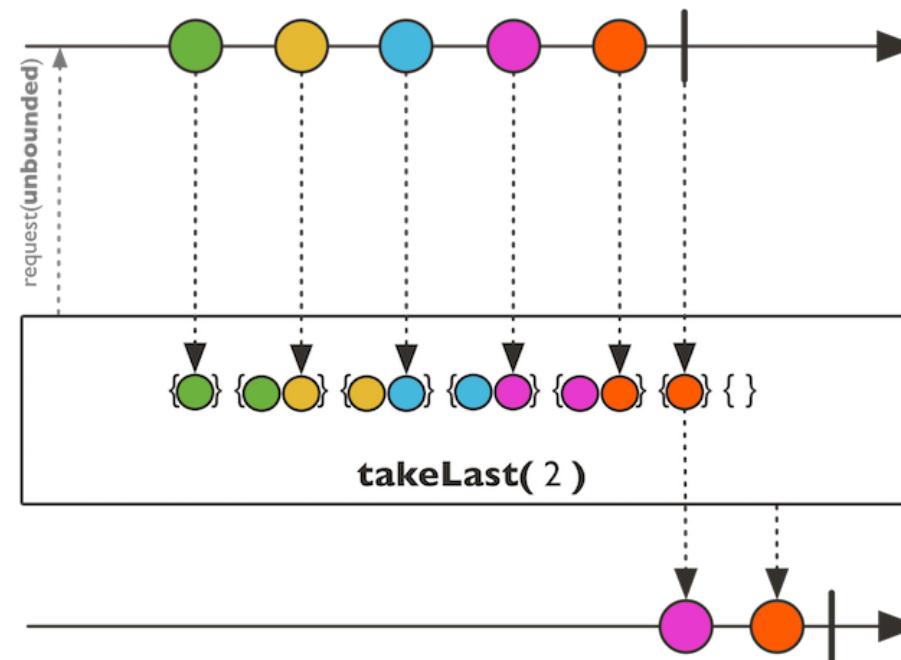
- Flux instance main methods:

- `take(long numItems)`
 - returns Flux<T> with only first ‘numItems’ from current stream
- `take(Duration delay)`
- `take(Duration delay, Scheduler timer)`
 - returns values until ‘delay’ expires



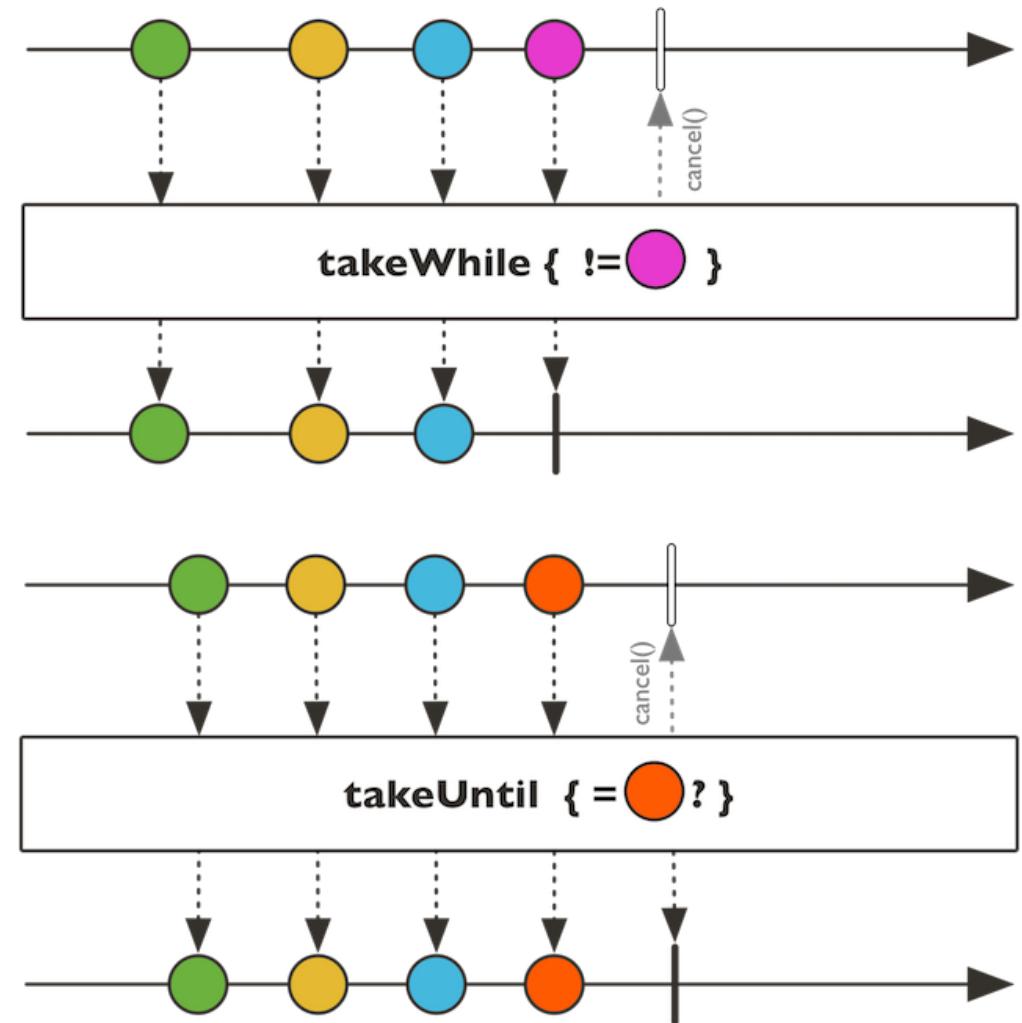
Flux

- Flux instance main methods:
 - `takeLast(long numItems)`
 - returns Flux<T> with only last ‘numItems’ from current stream



Flux

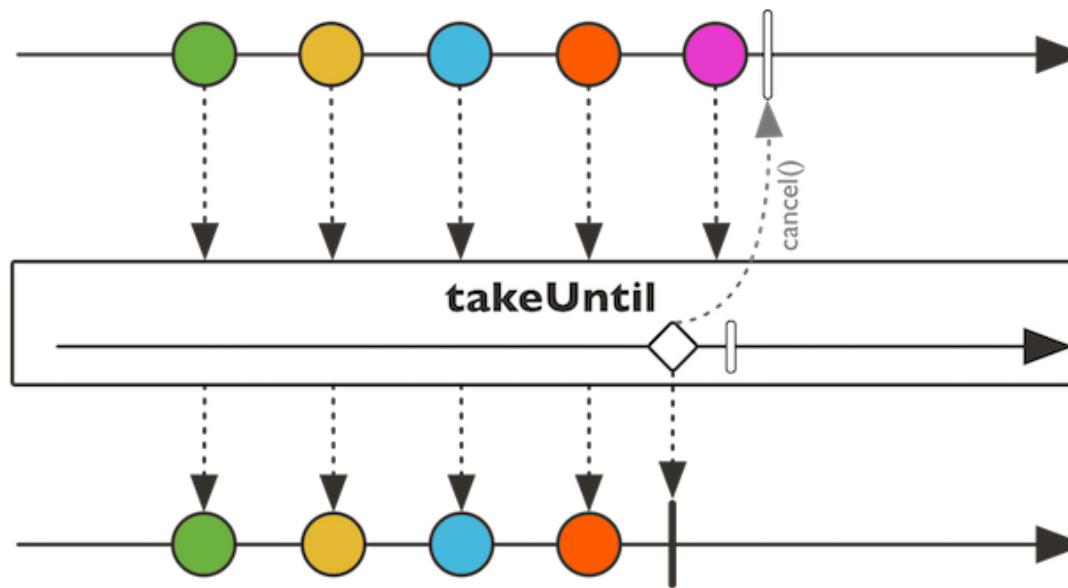
- Flux instance main methods:
 - `takeWhile (Predicate<T> test)`
 - returns values while T passes given test
 - `takeUntil(Predicate<T> test)`
 - returns values until T passes given test



Flux

- Flux instance main methods:

- `takeUntilOther(Publisher<T> other)`
 - returns values until other signals onNext or completes



Flux

- Example:
 - `take(...)`

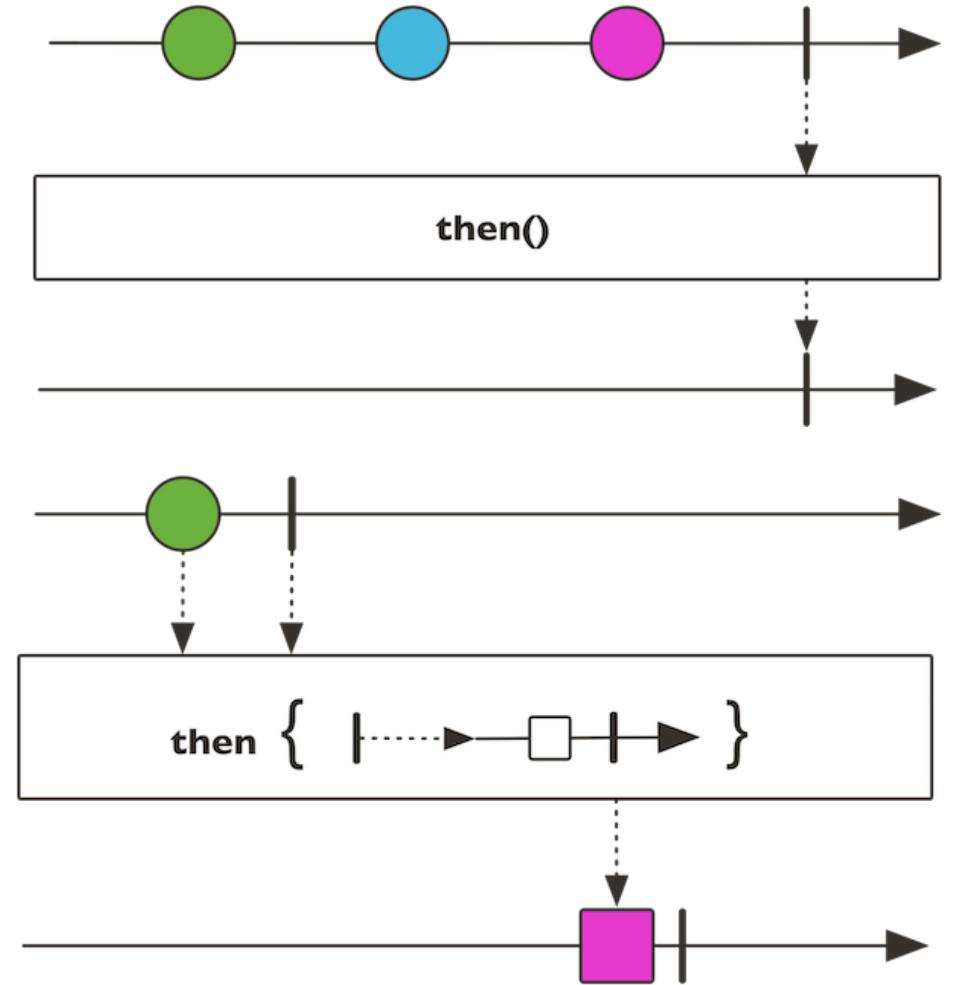
```
public Flux<Integer> fluxTake(int endAt){  
    return Flux.range(0,10).take(endAt);  
}
```

```
System.out.print("\n\nFlux Take: \n");  
ex.fluxTake(5).subscribe(System.out::print);
```

```
Flux Take:  
01234
```

Flux

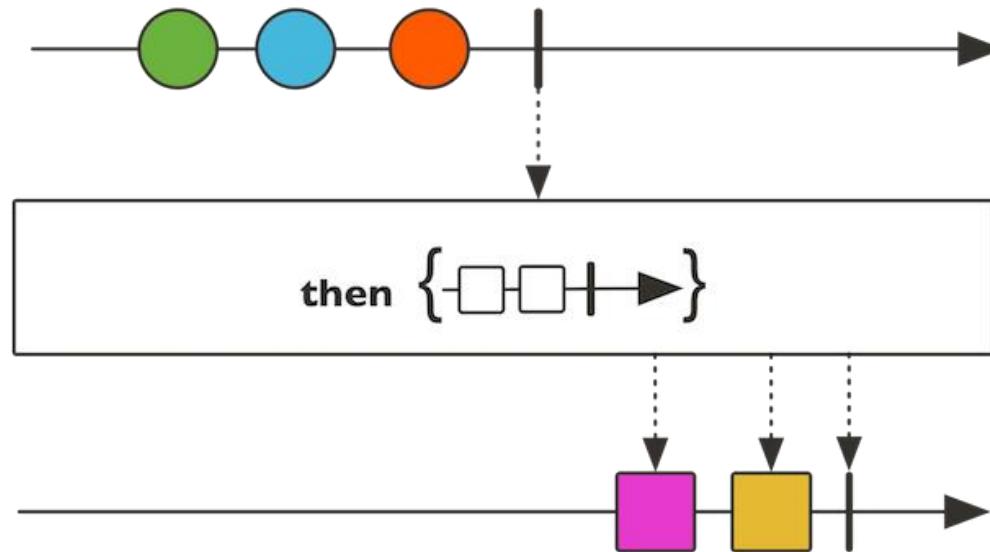
- Flux instance main methods:
 - `then()`
 - returns `Mono<Void>` on completion (successfully or with error)
 - `then(Mono<V>)`
 - returns `Mono<V>` on completion
 - errors are delegated to `Mono<V>`



Flux

- Flux instance main methods:

- `thenMany(Publisher<V> other)`
 - plays given stream when current `Flux<T>` completes
 - returns `Flux<V>`



Flux

- Example:
 - `than(...)`

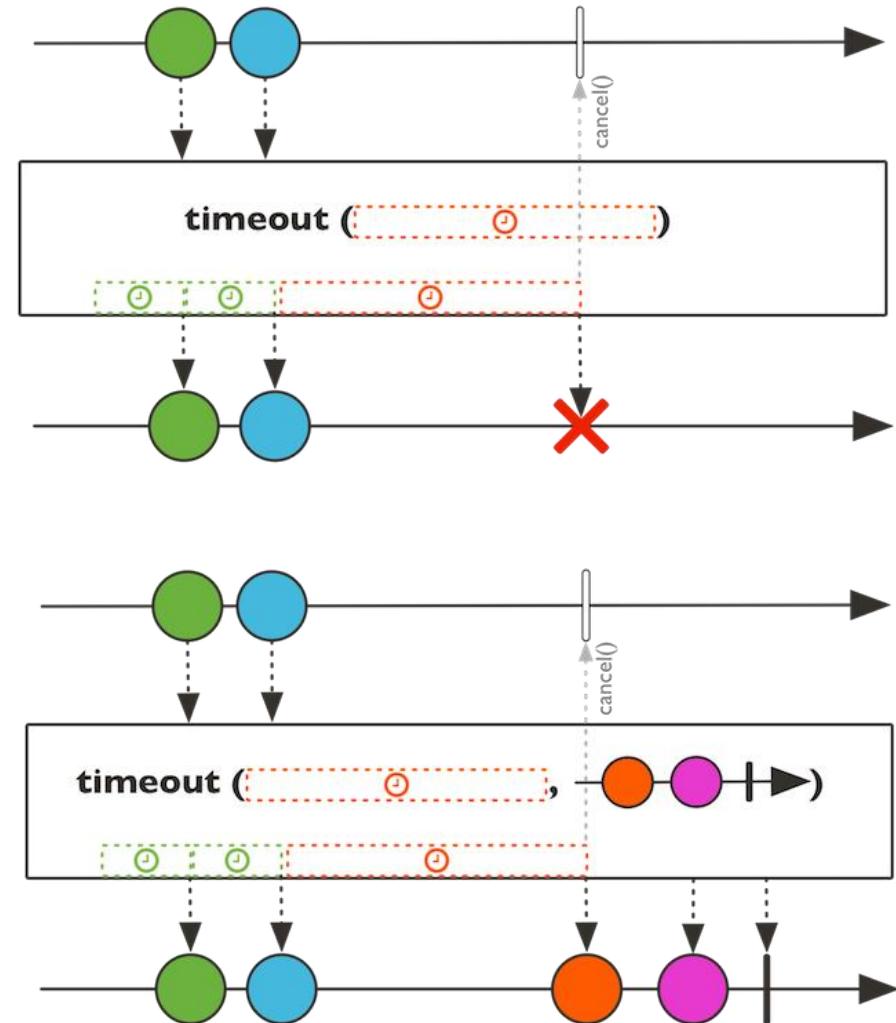
```
public Flux<?> fluxThenMany(){  
    return Flux.range(0,10).doOnNext(System.out::print).thenMany(fluxFrom());  
}
```

```
System.out.print("\n\nFlux Then Many: \n");  
ex.fluxThenMany().subscribe(System.out::print);
```

```
Flux Then Many:  
0123456789aaabbcc
```

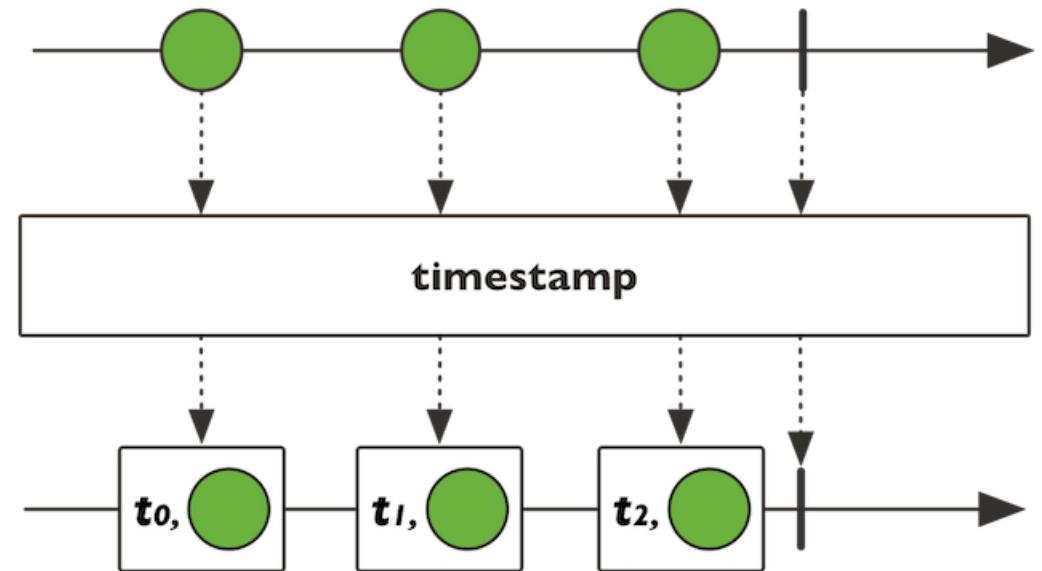
Flux

- Flux instance main methods:
 - `timeout(Duration timeout)`
 - propagates `TimeoutException` if no item emitted within given timeout
 - `timeout(Duration timeout, Publisher<T> fallback)`
 - fallbacks to given publisher if no item emitted within given timeout



Flux

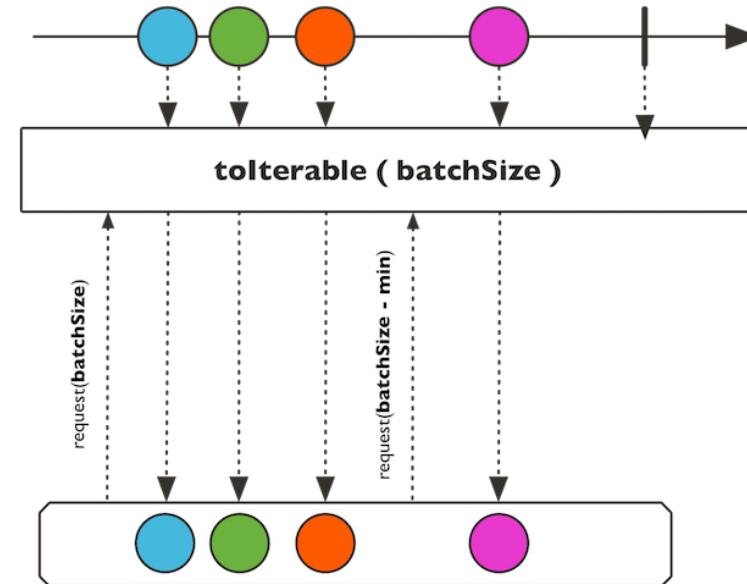
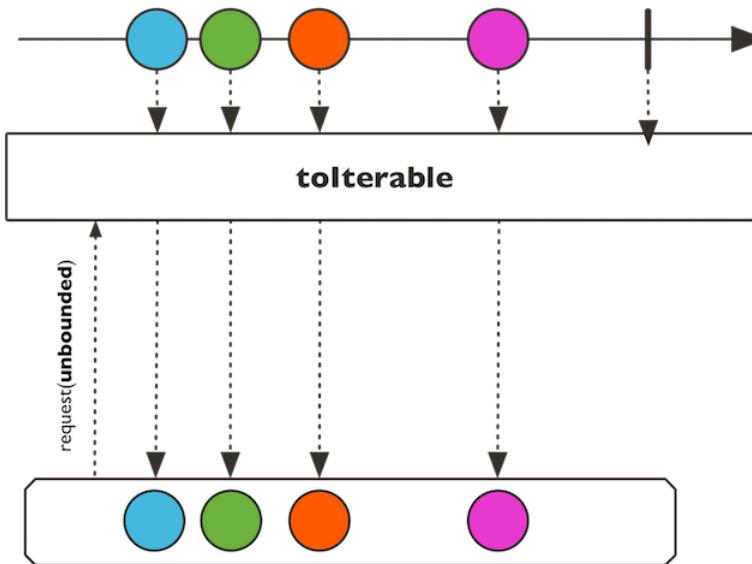
- Flux instance main methods:
 - `timestamp()`
 - returns `Flux<Tuple2<Long,T>>`
 - Long is current timestamp attached to item T
 - `timestamp(Scheduler scheduler)`
 - returns `Flux<Tuple2<Long,T>>`
 - Long is given scheduler current timestamp



Flux

- Flux instance main methods:

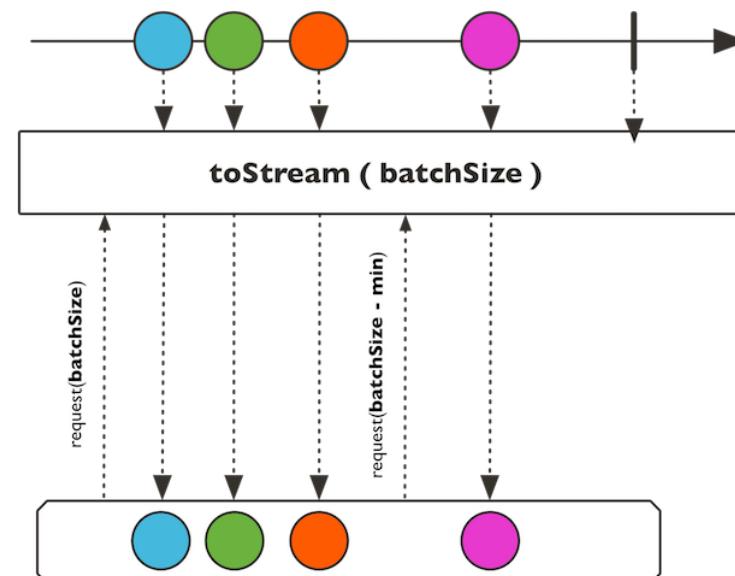
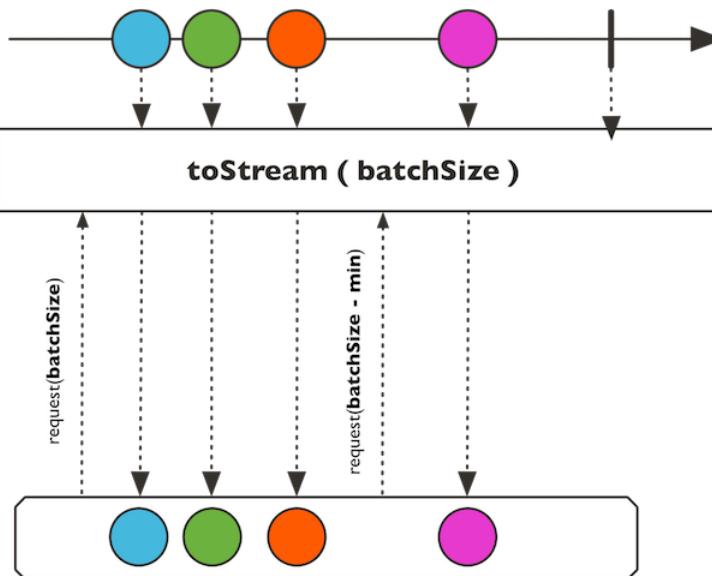
- `tolterable()`
- `tolterable(int batchSize)`
 - returns blocking Iterable<T>



Flux

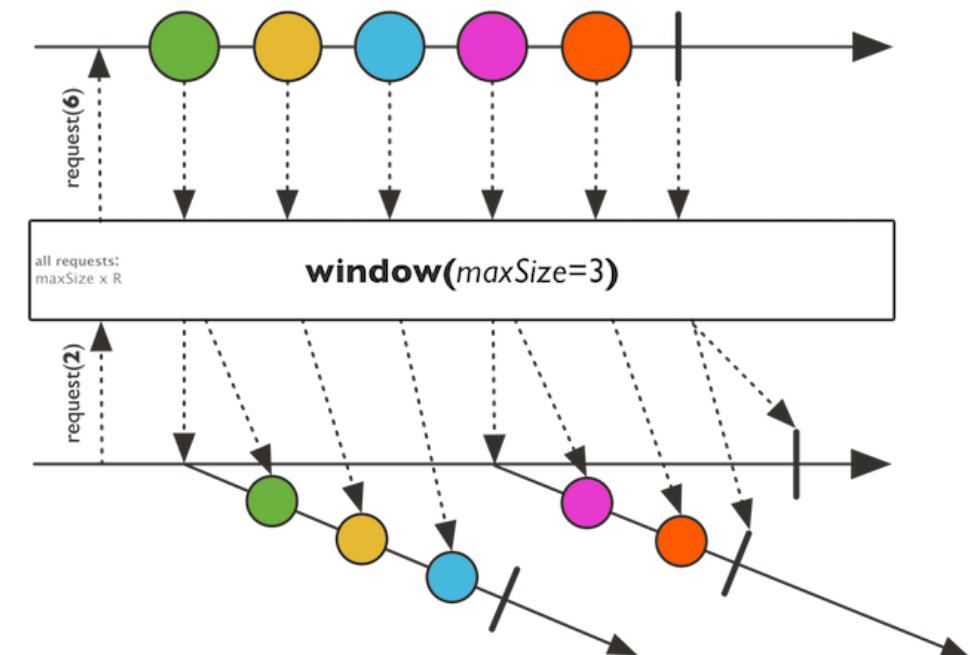
- Flux instance main methods:

- `toStream()`
- `toStream(int batchSize)`
 - returns `Stream<T>` which blocks for source `onNext` calls



Flux

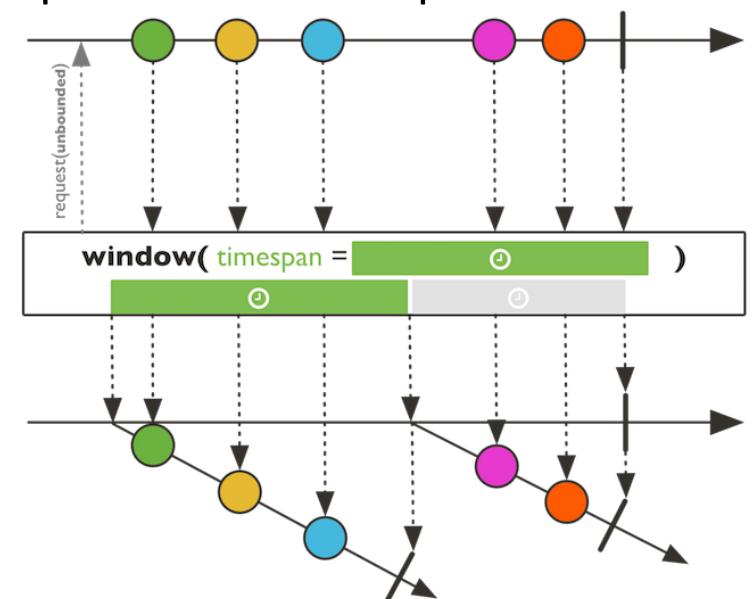
- Flux instance main methods:
 - `transform(Function<Flux<T>, Publisher<V>)`
 - transforms Flux<T> into Flux<V> with given mapper
 - `window(int maxSize)`
 - splits Flux<T> into multiple Flux<T> with given maxSize
 - returns Flux<Flux<T>>



Flux

- Flux instance main methods:

- `window(Duration timespan)`
- `window(Duration timespan , Duration timeshift)`
 - splits Flux<T> into multiple Flux<T> every period of time specified as ‘timespan’
 - timeshift specifies intervals
 - `window()` versions with Scheduler are also available
 - returns Flux<Flux<T>>



Flux

- Example:
 - `window(...)`

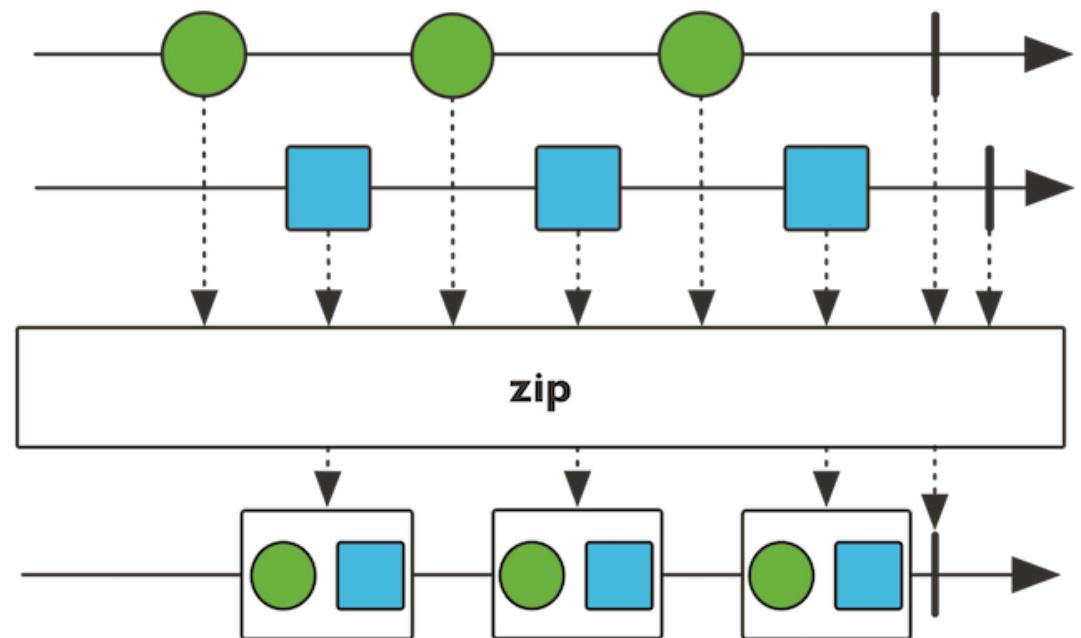
```
public Flux<Flux<Integer>> fluxWindow(){  
    return Flux.range(0,10).window(3).doOnNext(s->System.out.print(" window: "));  
}
```

```
System.out.print("\n\nFlux Window: \n");  
ex.fluxWindow().subscribe(flux->flux.subscribe(System.out::print));
```

```
Flux Window:  
window: 012 window: 345 window: 678 window: 9
```

Flux

- Flux instance main methods:
 - `zipWith(Publisher<V> other)`
 - merges current Flux<T> with Publisher<V>
 - `zipWithIterable(Iterable<V> other)`
 - merges current Flux<T> with Iterable<V>
 - results in Flux<Tuple2<T,V>>



Flux

- Example:
 - `zipWith(...)`

```
public Flux<Tuple2<Integer, String>> fluxZipWith(){  
    return Flux.range(1,3).zipWith(fluxFrom());  
}
```

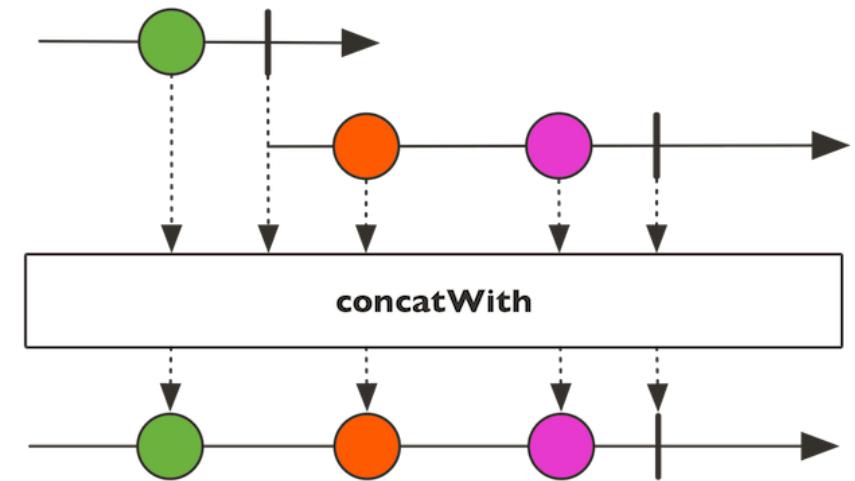
```
System.out.print("\n\nFlux Zip With: \n");  
ex.fluxZipWith().subscribe(tuple2->System.out.println(tuple2.getT1()+"-"+tuple2.getT2()));
```

```
Flux Zip With:  
1-aaa  
2-bbb  
3-ccc
```

Mono to Flux

Mono to Flux

- Turning Mono into Flux:
 - concatWith(Publisher<T> publisher)
 - Appends given Publisher to current Mono<T>
 - no interleave – all items become one Flux



Mono to Flux

- Example:
 - concatWith(...)

```
public Flux<String> monoConcat(){  
    return Mono.just("Hello").concatWith(Flux.just(" ", "Reactive", " ", "World", "!"));  
}
```

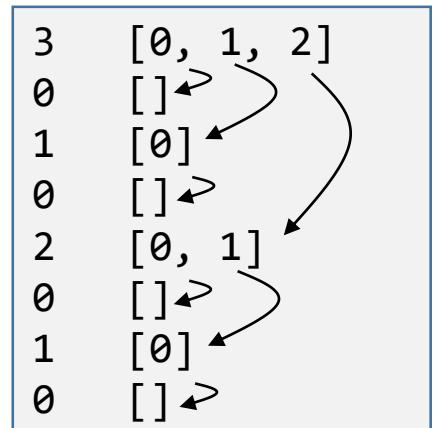
```
System.out.print("\n\nMono to Flux - Concat: \n");  
ex.monoConcat().subscribe(System.out::print);
```

```
Mono to Flux - Concat:  
Hello Reactive World!
```

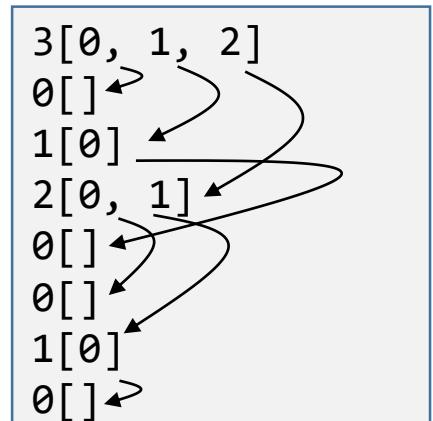
Mono to Flux

- Turning Mono into Flux:

- `expand(Function<T, Publisher<T> publisher)`
 - applies function to current `Mono<T>` value
 - recursively applies function to each of the generated values
 - uses breadth-first traversal strategy
 - for example, if function turns `Mono{3}` into `Flux{1,2,3}` the result will be:



- `expandDeep(Function<T, Publisher<T> publisher)`
 - does the same but uses deep-first strategy
 - for example, if function turns `Mono{3}` into `Flux{1,2,3}` the result will be:



Mono to Flux

- Example:
 - `expand(...)`

```
public Flux<Integer> monoExpand(int value){  
    return Mono.just(value).expand(val->{  
        List<Integer> data=new ArrayList<>();  
        for(int i=0;i<val;i++){  
            data.add(i);  
        }  
        System.out.println(data);  
        return Flux.fromIterable(data);  
    });  
}
```

```
System.out.print("\n\nMono to Flux - Expand: \n");  
ex.monoExpand(3).subscribe(System.out::print);
```

ono to Flux - Expand:
3[0, 1, 2]
0[]
1[0]
2[0, 1]
0[]
0[]
1[0]
0[]

Mono to Flux

- Example:
 - `expandDeep(...)`

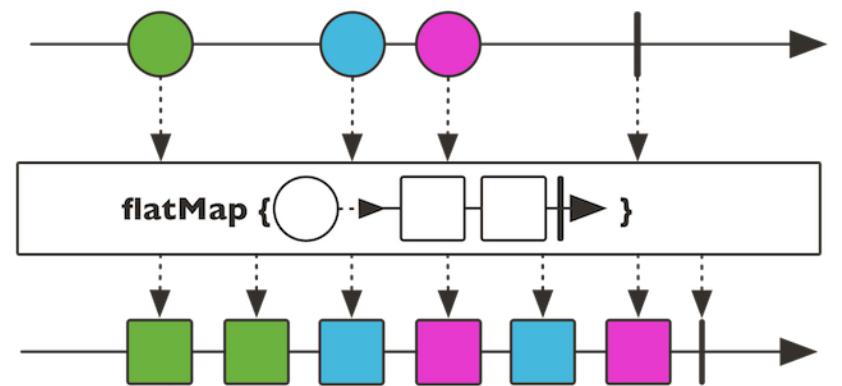
```
public Flux<Integer> monoExpand(int value){  
    return Mono.just(value).expandDeep(val->{  
        List<Integer> data=new ArrayList<>();  
        for(int i=0;i<val;i++){  
            data.add(i);  
        }  
        System.out.println(data);  
        return Flux.fromIterable(data);  
    });  
}
```

```
System.out.print("\n\nMono to Flux - Expand Deep: \n");  
ex.monoExpand(3).subscribe(System.out::print);
```

Mono to Flux - Expand Deep:
3[0, 1, 2]
0[]
1[0]
0[]
2[0, 1]
0[]
1[0]
0[]

Mono to Flux

- Turning Mono into Flux:
 - `flatMapIterable(Function<T, Iterable<R>> mapper)`
 - transforms `<T>` emitted by current Mono into `Flux<R>`
 - uses a mappers to map `T` into `Iterable<R>`
 - `flatMapMany(Function<T, Publisher<R>> mapper)`
 - transforms `<T>` emitted by current Mono into `Flux<R>`
 - uses a mappers to map `T` into `Publisher<R>`
- `flux()`
 - converts this `Mono<T>` to `Flux<T>`



Mono to Flux

- Example:
 - flatMapIterable(...)

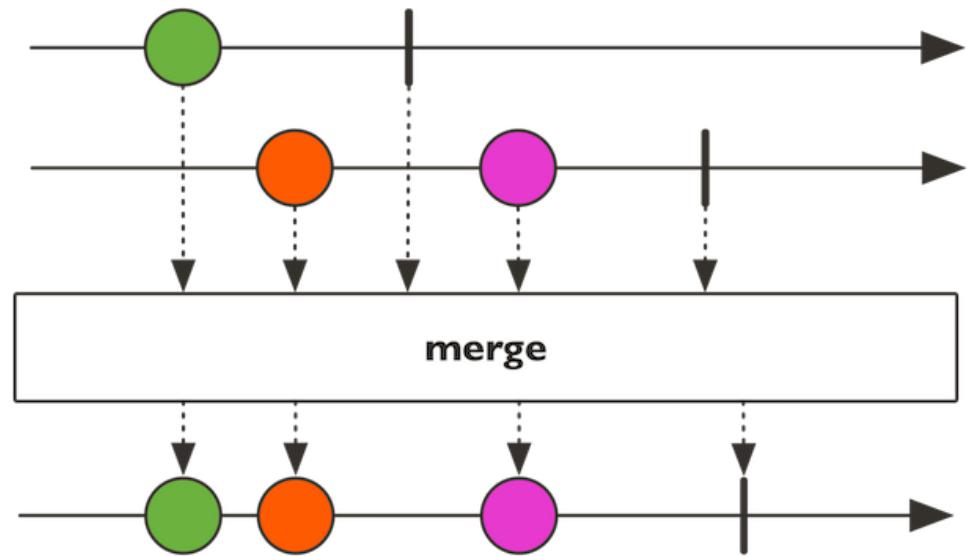
```
public Flux<Character> monoFlatMapIterable(int value){  
    return Mono.just(value).flatMapIterable(val->{  
        List<Character> data=new ArrayList<>();  
        for(int i=0;i<val;i++){  
            char curr=(char)('a'+i);  
            data.add(curr);  
        }  
        return data;  
    });  
}
```

```
System.out.print("\n\nMono to Flux - Flat Map Iterable: \n");  
ex.monoFlatMapIterable(5).subscribe(System.out::print);
```

```
Mono to Flux - Flat Map Iterable:  
abcde
```

Mono to Flux

- Turning Mono into Flux:
 - `mergeWith(Publisher<T> other)`
 - merges `<T>` with given publisher
 - appends other to emitted `<T>`
 - returns `Flux<T>`



Mono to Flux

- Example:
 - `mergeWith(...)`

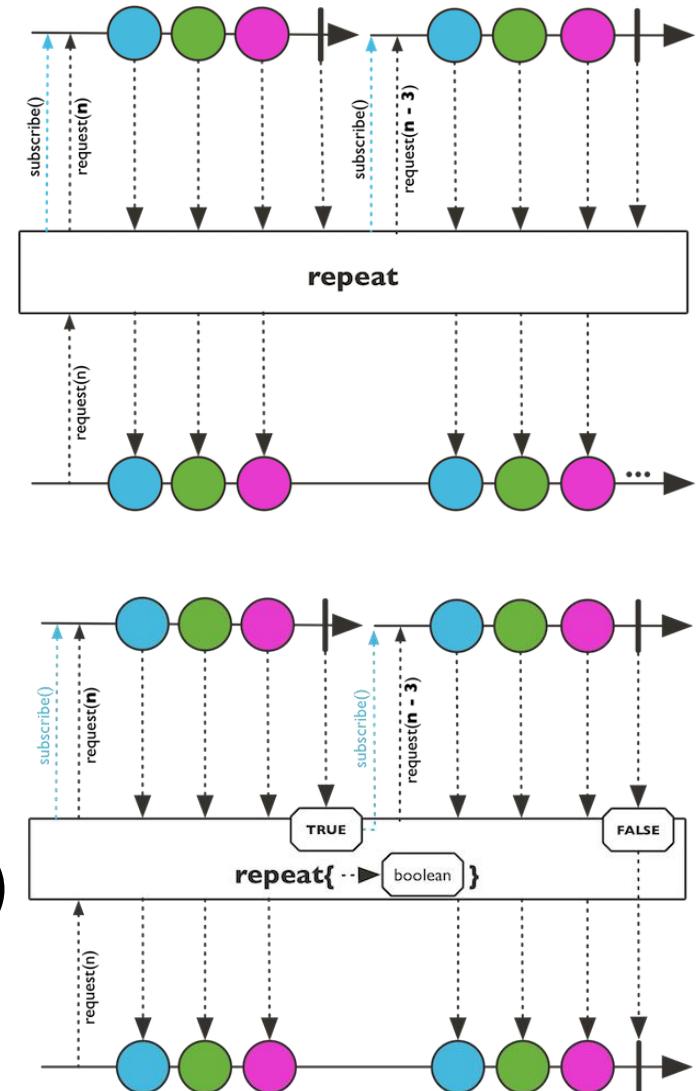
```
public Flux<String> monoMergeWith(){  
    return Mono.just("Hello").mergeWith(Flux.just(" Reactive"," !!!"));  
}
```

```
System.out.print("\n\nMono to Flux - Merge With: \n");  
ex.monoMergeWith().subscribe(System.out::print);
```

```
Mono to Flux - Merge With:  
Hello Reactive !!!
```

Mono to Flux

- Turning Mono into Flux:
 - `repeat()`
 - repeatedly & indefinitely subscribes current `Mono<T>`
 - keeps on subscribing endlessly on each completion
 - `repeat(BooleanSupplier predicate)`
 - does the same but stops when supplier returns ‘false’
 - `repeat(long numOfRepeats)`
 - does the same but stops after ‘numOfRepeats’ subscriptions
 - `repeat(long numOfRepeats, BooleanSupplier predicate)`



Mono to Flux

- Example:
 - `repeat(...)`

```
public Flux<String> monoRepeat(int numOfRepeats){  
    return Mono.just("repeat ").repeat(numOfRepeats);  
}
```

```
System.out.print("\n\nMono to Flux - Repeat: \n");  
ex.monoRepeat(3).subscribe(System.out::print);
```

```
Mono to Flux - Repeat:  
repeat repeat repeat
```

Mono to Flux

- Example:
 - `repeat(...Predicate<T>)`

```
public Flux<Integer> monoRepeatPredicate(int numOfRepeats){  
    int value=(int)(Math.random()*10);  
    System.out.print("(" + value + ") ");  
    return Mono.just(value).repeat(numOfRepeats, () -> value % 2 == 0);  
}
```

```
System.out.print("\n\nMono to Flux - Repeat Predicate: \n");  
ex.monoRepeatPredicate(3).subscribe(System.out::print);
```

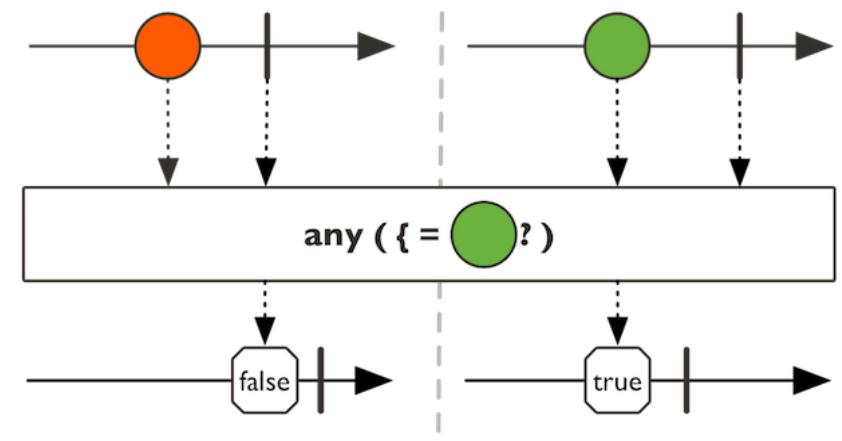
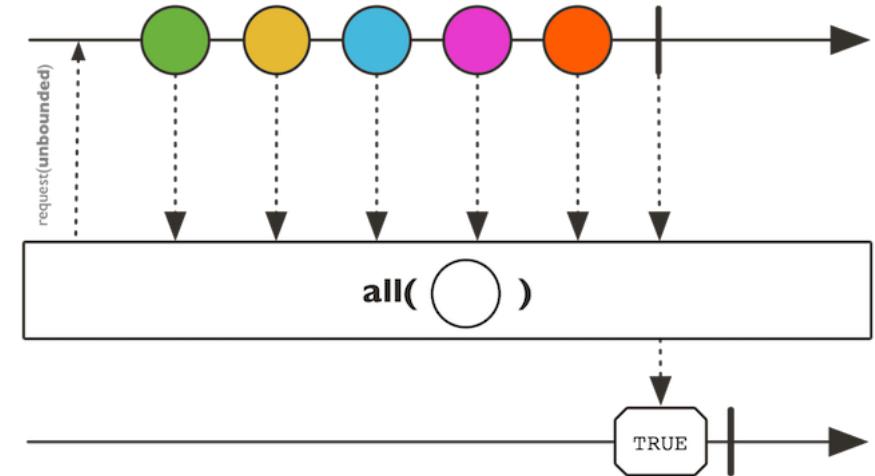
Mono to Flux - Repeat Predicate:
(3) 3

Mono to Flux - Repeat Predicate:
(8) 8888

Flux to Mono

Flux to Mono

- Turning Flux into Mono:
 - `all(Predicate<T> test)`
 - returns Mono<Boolean> with ‘true’ if all Flux values passes the test
 - `any(Predicate<T> test)`
 - returns Mono<Boolean> with ‘true’ if any of Flux values passes the test



Flux to Mono

- Example:
 - all(...)

```
public Mono<Boolean> fluxAll(Integer...numbers){  
    return Flux.fromArray(numbers).all(num->num%2!=0);  
}
```

```
System.out.print("\n\nFlux to Mono - All: \n");  
ex.fluxAll(1,5,6).subscribe(System.out::print);
```

```
Flux to Mono - All:  
false
```

Flux to Mono

- Example:

- any(...)

```
public Mono<Boolean> fluxAny(Integer...numbers){  
    return Flux.fromArray(numbers).any(num->num%2!=0);  
}
```

```
System.out.print("\n\nFlux to Mono - Any: \n");  
ex.fluxAny(1,4,5).subscribe(System.out::print);
```

```
Flux to Mono - Any:  
true
```

Flux to Mono

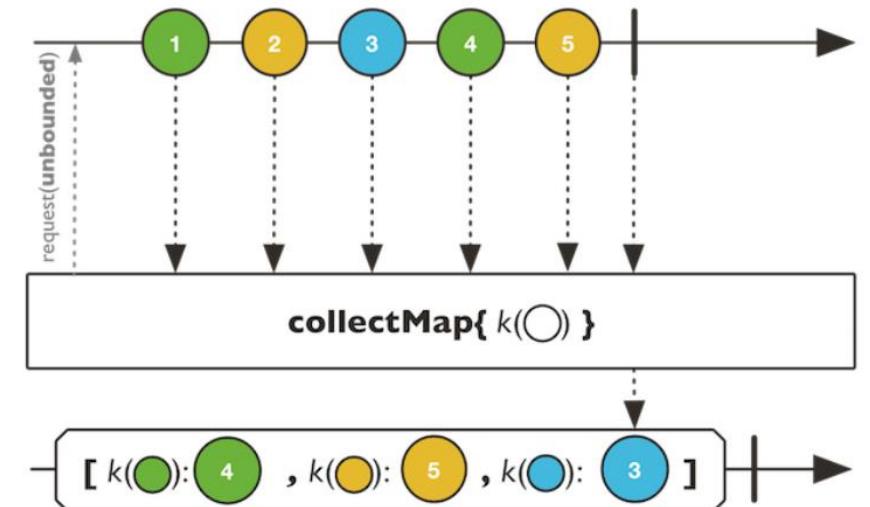
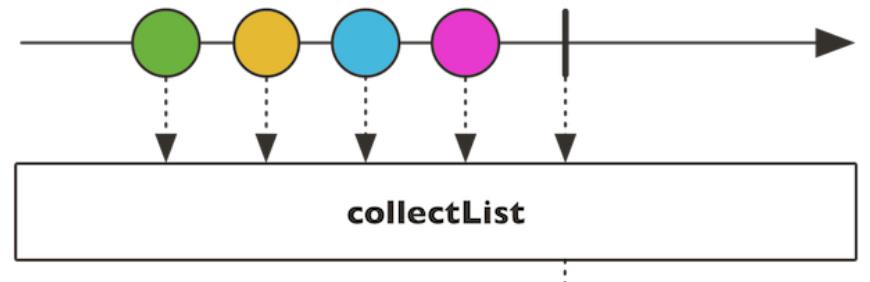
- Turning Flux into Mono:

- `collectList()`

- returns `Mono<List<T>>` containing all `Flux<T>` values when the sequence completes

- `collectMap(Function<T,K> extractor)`

- returns `Mono<Map<K,T>>` containing all `Flux<T>` values when the sequence completes
 - each `T` value is attached to a Key `<K>` extracted via given `Function<T,K>`



Flux to Mono

- Example:
 - `collectList(...)`

```
public Mono<List<Integer>> fluxCollectList(Integer...numbers){  
    return Flux.fromArray(numbers).collectList();  
}
```

```
System.out.print("\n\nFlux to Mono - Collect List: \n");  
ex.fluxCollectList(1,2,3).subscribe(System.out::print);
```

```
Flux to Mono - Collect List:  
[1, 2, 3]
```

Flux to Mono

- Example:
 - `collectList(...)`

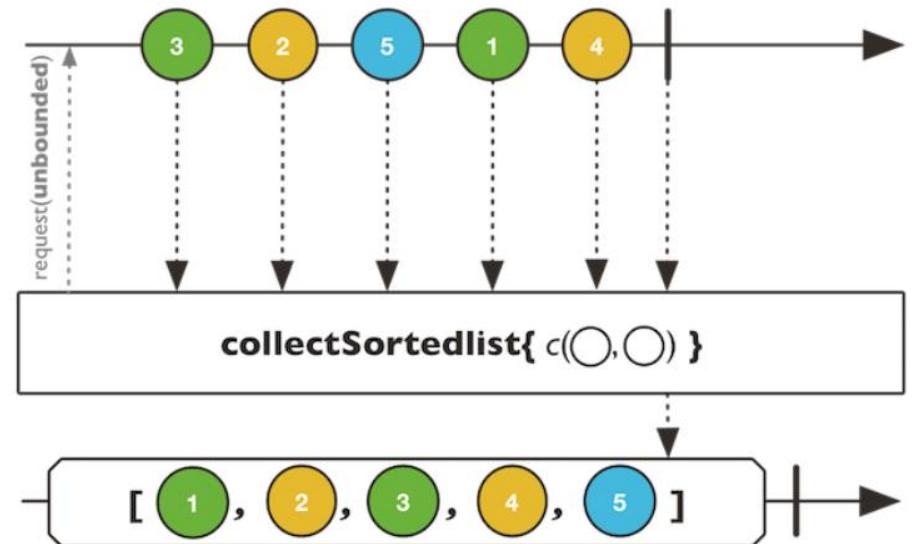
```
public Mono<Map<Integer, String>> fluxCollectMap(String ...values){  
    return Flux.fromArray(values).collectMap(str->str.length());  
}
```

```
System.out.print("\n\nFlux to Mono - Collect Map: \n");  
ex.fluxCollectMap("a", "bbb", "cc").subscribe(System.out::print);
```

```
Flux to Mono - Collect Map:  
{1=a, 3=bb, 2=ccc}
```

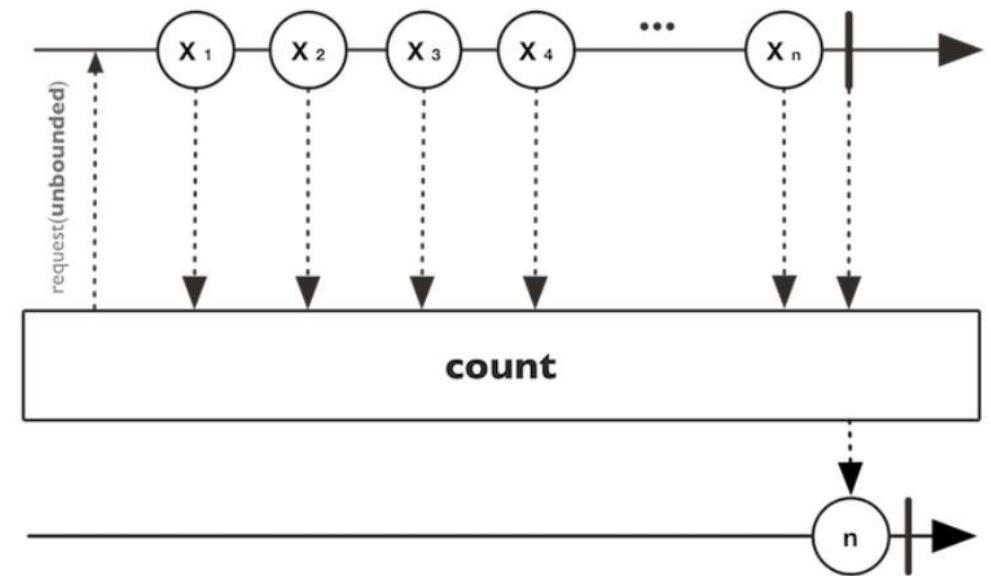
Flux to Mono

- Turning Flux into Mono:
 - `collectSortedList()`
 - `collectSortedList(Comparator<T> comparator)`
 - returns `Mono<List<T>>` containing all `Flux<T>` values when the sequence completes
 - `List<T>` is sorted according to natural ordering or given `Comparator<T>`



Flux to Mono

- Turning Flux into Mono:
 - `count()`
 - results in `Mono<Long>` which describes the number of values in current `Flux<T>`



Flux to Mono

- Example:
 - `count(...)`

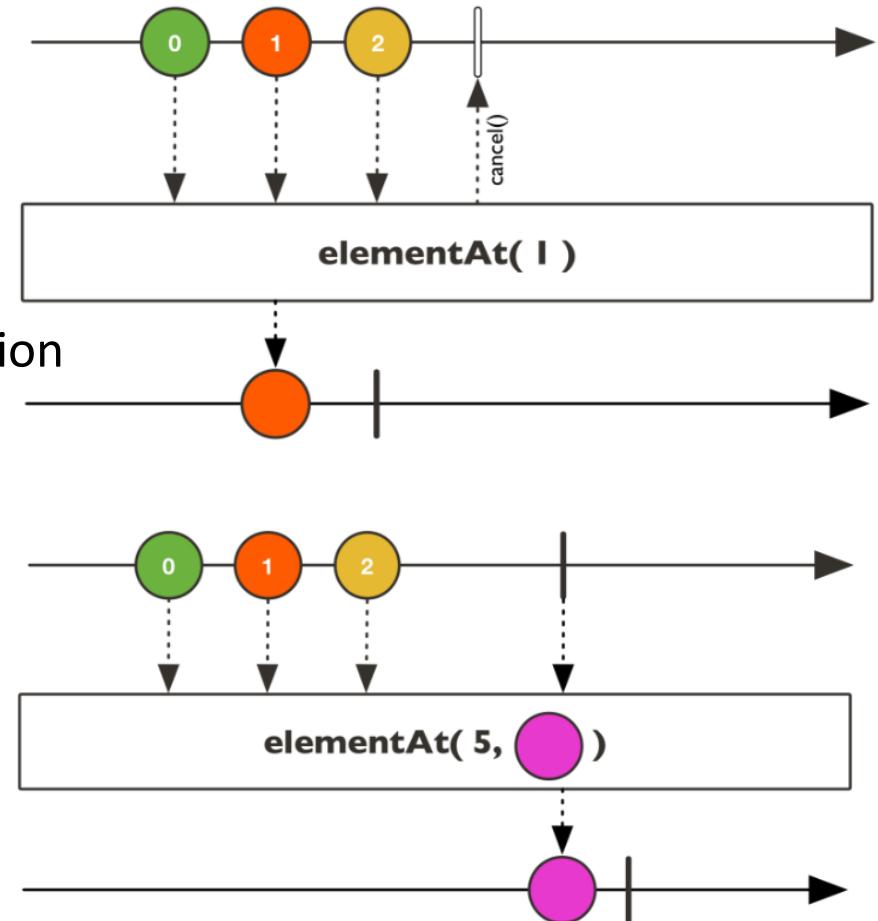
```
public Mono<Long> fluxCount(String...values){  
    return Flux.fromArray(values).count();  
}
```

```
System.out.print("\n\nFlux to Mono - Count: \n");  
ex.fluxCount("a","bb","ccc").subscribe(System.out::print);
```

```
Flux to Mono - Count:  
3
```

Flux to Mono

- Turning Flux into Mono:
 - `elementAt(int position)`
 - emits only the value in the given position
 - If no such position – throws `IndexOutOfBoundsException`
 - `elementAt(int position, T defaultValue)`
 - emits only the value in the given position
 - If no such position – returns ‘defaultValue’



Flux to Mono

- Example:
 - `elementAt(...)`

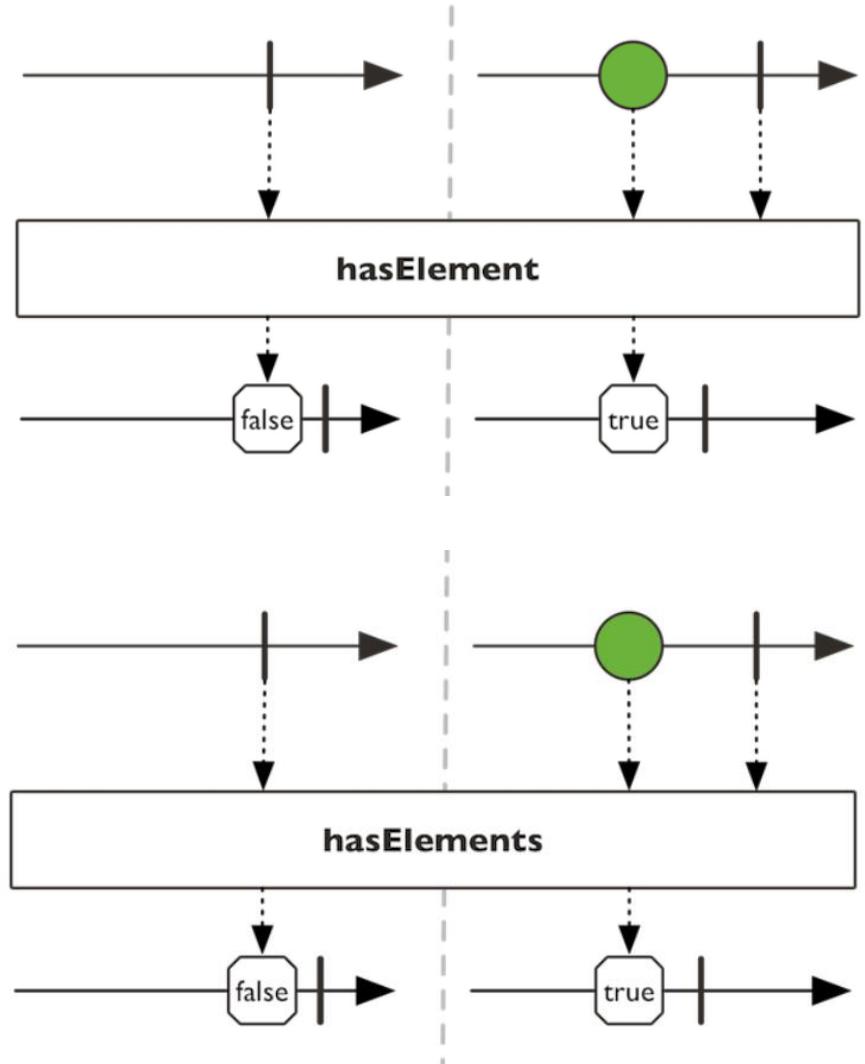
```
public Mono<String> fluxElementAt(int position){  
    return fluxFrom().elementAt(position, "Not Found");  
}
```

```
System.out.print("\n\nFlux to Mono - Element At: \n");  
ex.fluxElementAt(2).subscribe(System.out::print);
```

```
Flux to Mono - Element At:  
ccc
```

Flux to Mono

- Turning Flux into Mono:
 - `hasElement(T value)`
 - returns `Mono<Boolean>` with ‘true’ if any of the Flux values equals to ‘value’
 - `hasElements()`
 - returns `Mono<Boolean>` with ‘true’ if Flux is not empty



Flux to Mono

- Example:
 - `hasElement(...)`

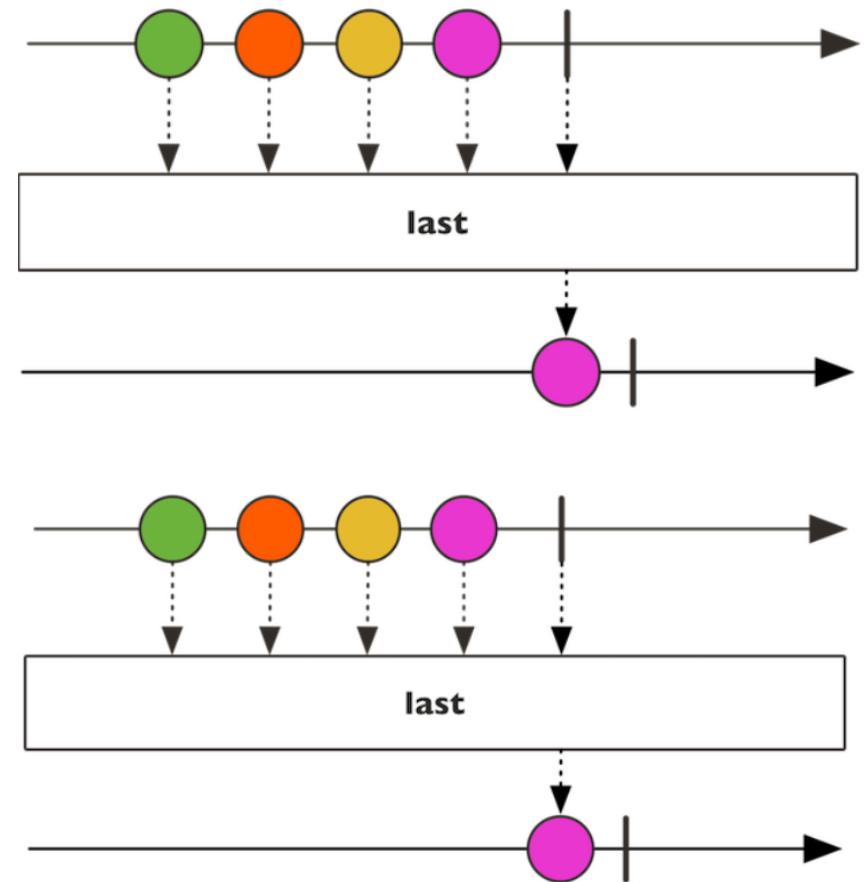
```
public Mono<Boolean> fluxHasElement(String value){  
    return fluxFrom().hasElement(value);  
}
```

```
System.out.print("\n\nFlux to Mono - Has Element: \n");  
ex.fluxHasElement("zzz").subscribe(System.out::print);
```

```
Flux to Mono - Has Element:  
false
```

Flux to Mono

- Turning Flux into Mono:
 - `last()`
 - emits last value of the current Flux<T>
 - if empty – throws NoSuchElementException
 - `last(T defaultValue)`
 - emits last value of the current Flux<T>
 - If empty – returns ‘defaultValue’



Flux to Mono

- Example:
 - `last(...)`

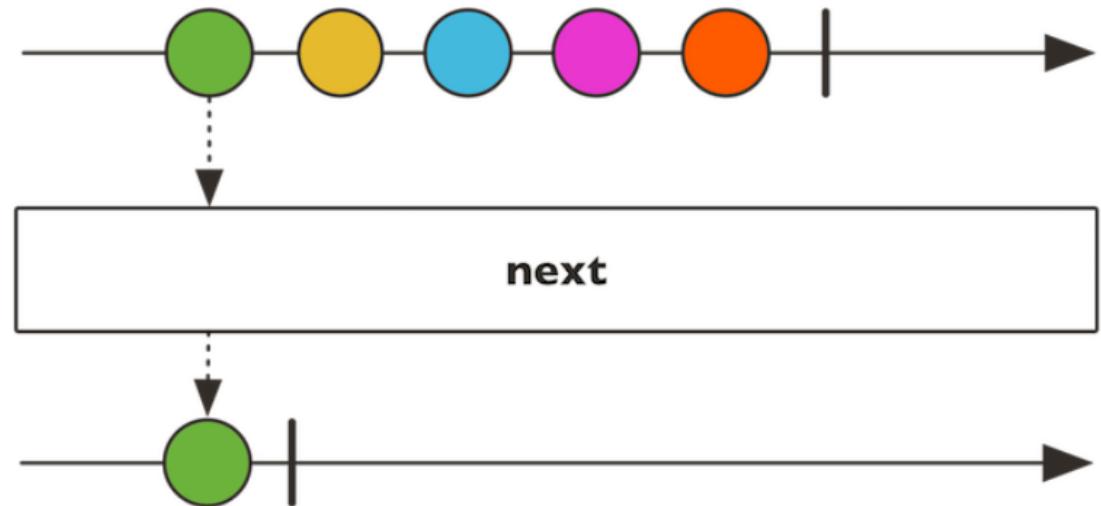
```
public Mono<String> fluxLast(){  
    return fluxFrom().last();  
}
```

```
System.out.print("\n\nFlux to Mono - Last: \n");  
ex.fluxLast().subscribe(System.out::print);
```

```
Flux to Mono - Last:  
ccc
```

Flux to Mono

- Turning Flux into Mono:
 - `next()`
 - emits the next value of the current Flux<T>
 - if Flux<T> is empty – returns empty Mono<?>



Flux to Mono

- Example:
 - `next(...)`

```
public Mono<String> fluxNext(){  
    return fluxFrom().next();  
}
```

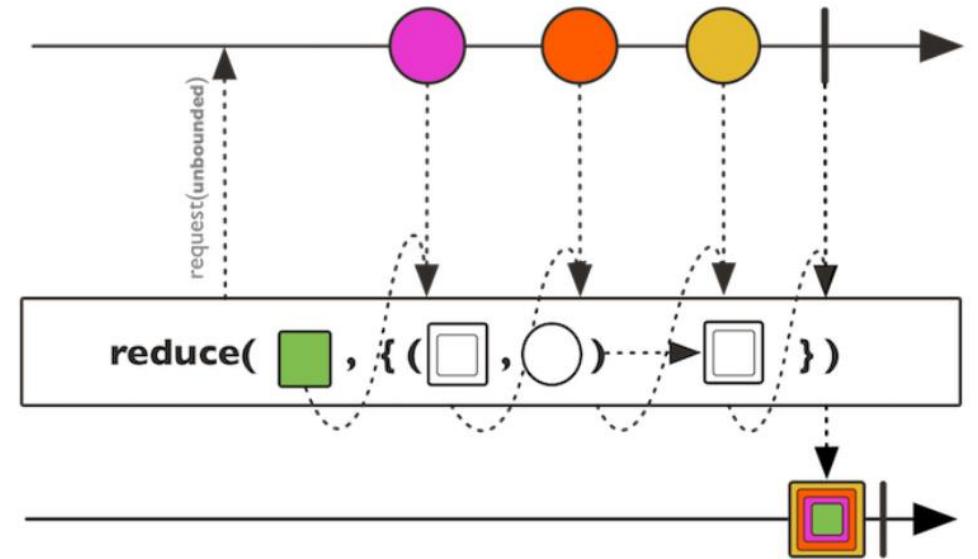
```
System.out.print("\n\nFlux to Mono - Next: \n");  
ex.fluxNext().subscribe(System.out::print);
```

```
Flux to Mono - Next:  
aaa
```

Flux to Mono

- Turning Flux into Mono:

- `reduce(A initial, BiFunction<A,T,A> reducer)`
- `reduceWith(Supplier<A> initial, BiFunction<A,T,A> reducer)`
 - reduces all $<T>$ values into $<A>$ single value
 - starts with given initial value / supplier
 - uses $\text{BiFunction}<\text{A}, \text{T}, \text{A}>$ which merges each T into given A and results with A value
- `reduce(BiFunction<T,T,T> aggregator)`
 - reduces all $<T>$ values into $<T>$ single value
 - uses $\text{BiFunction}<\text{T}, \text{T}, \text{T}>$ which merges each T into given T and results with T value
 - starts with the first value of current $\text{Flux}<\text{T}>$



Flux to Mono

- Example:
 - `reduce(...)`

```
public Mono<Integer> fluxReduce(String...values){  
    return Flux.fromArray(values).reduce(0,(result,value)->result+=value.length());  
}
```

```
System.out.print("\n\nFlux to Mono - Reduce: \n");  
ex.fluxReduce("a","bb","ccc").subscribe(System.out::print);
```

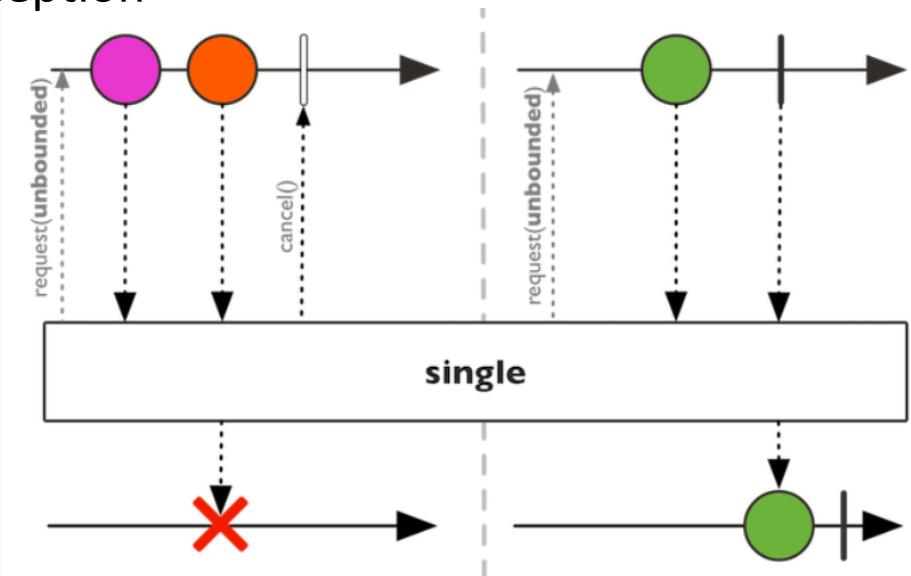
```
Flux to Mono - Reduce:  
6
```

Flux to Mono

- Turning Flux into Mono:

- `single()`

- emits the single value of the current Flux<T>
 - if Flux<T> is empty – throws IndexOutOfBoundsException
 - If Flux<T> has more than one element – throws NoSuchElementException



Flux to Mono

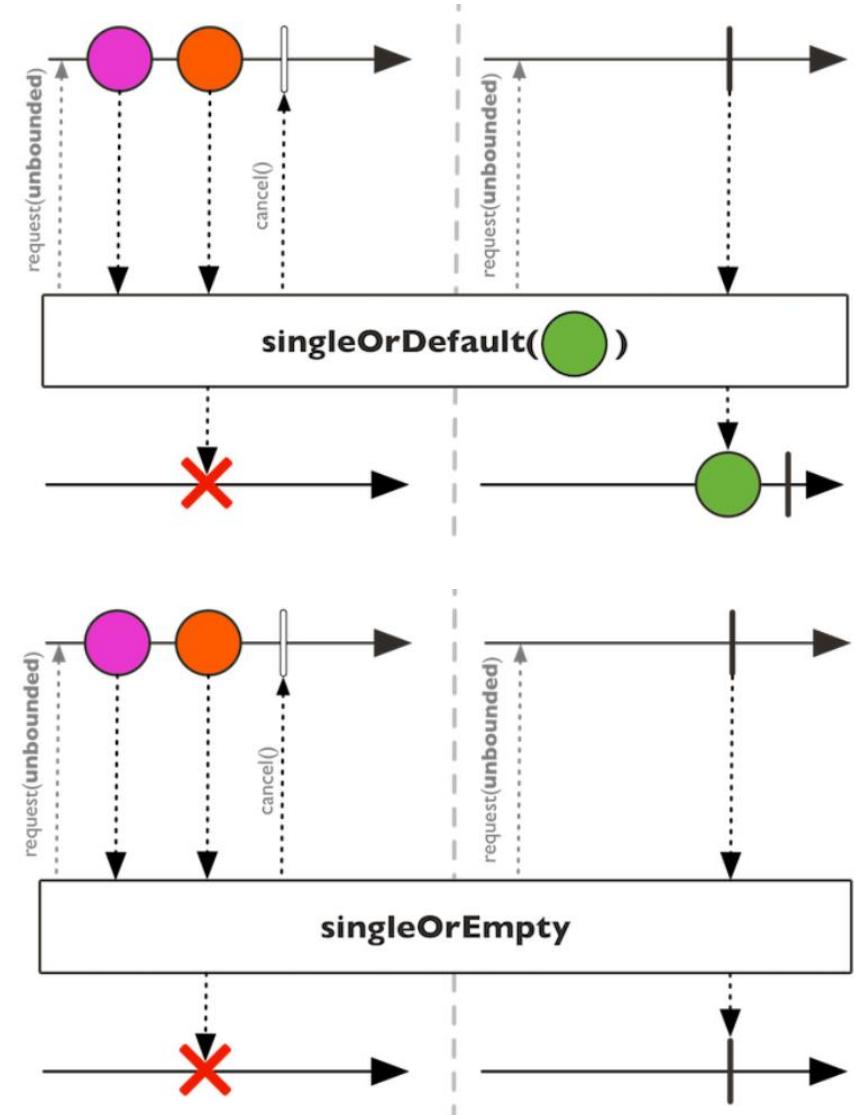
- Turning Flux into Mono:

- `single(T defaultValue)`

- emits the single value of the current `Flux<T>`
 - if `Flux<T>` is empty – returns ‘`defaultValue`’
 - If `Flux<T>` has more than one element – throws `NoSuchElementException`

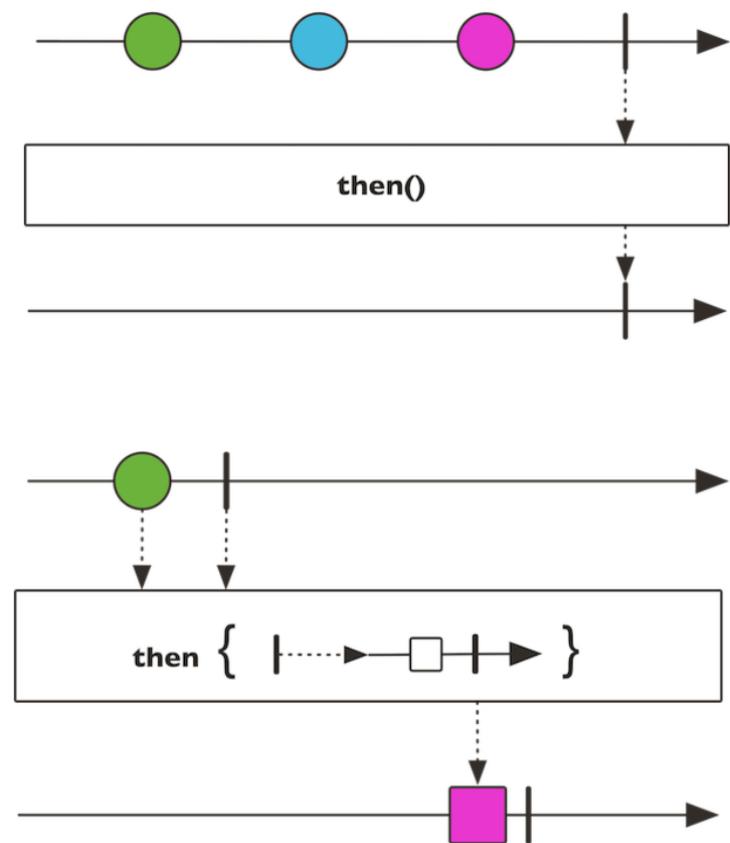
- `singleOrEmpty()`

- emits the single value of the current `Flux<T>`
 - if `Flux<T>` is empty – returns empty `Mono<?>`
 - If `Flux<T>` has more than one element – throws `NoSuchElementException`



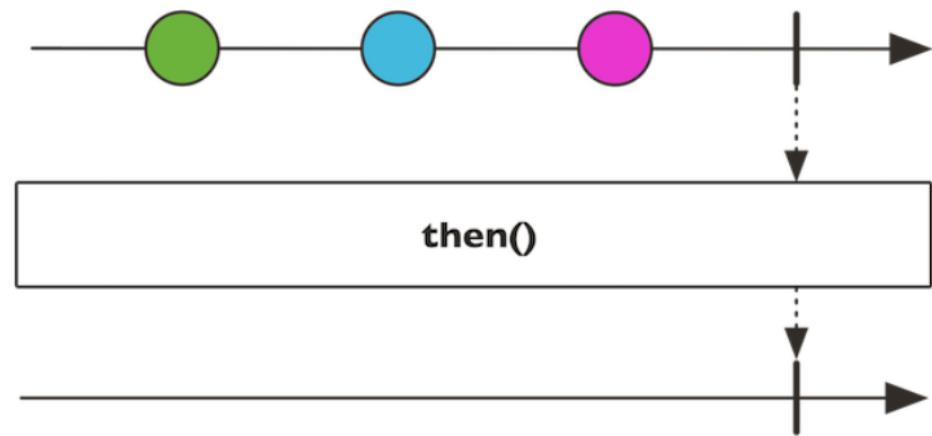
Flux to Mono

- Turning Flux into Mono:
 - `then()`
 - returns `Mono<Void>` which signals when current `Flux<T>` completes
 - `then(Mono<V> other)`
 - plays signals from given `Mono<V>` when current `Flux<T>` completes



Flux to Mono

- Turning Flux into Mono:
 - `thenEmpty(Publisher<Void> other)`
 - Returns `Mono<Void>` which signals after both current `Flux<T>` and given `Publisher<Void>` ‘other’ completes



Flux to Mono

- Example:
 - `then(...)`

```
public Mono<String> fluxThen(){  
    return Flux.just(1,2,3,4,5).doOnNext(System.out::print).then(Mono.just(" Done"));  
}
```

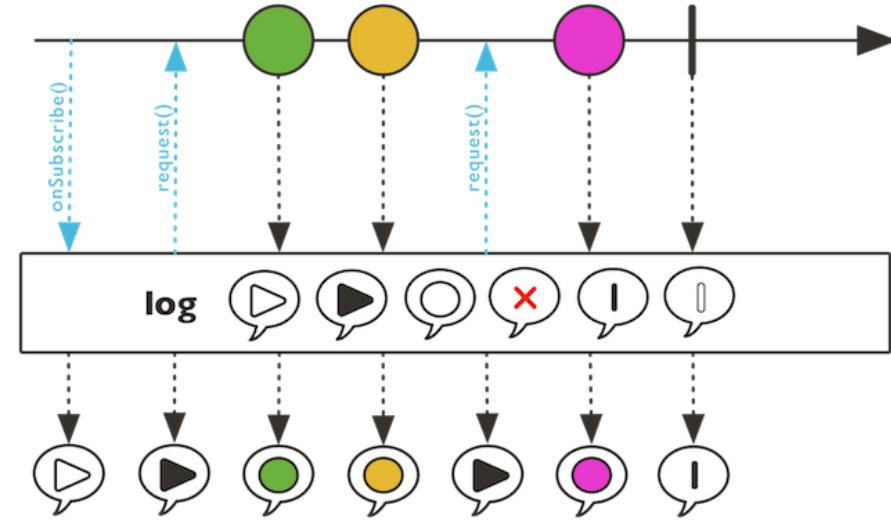
```
System.out.print("\n\nFlux to Mono - Then: \n");  
ex.fluxThen().subscribe(System.out::print);
```

```
Flux to Mono - Then:  
12345 Done
```

Flux & Mono Logging

- Logging all subscription activity
 - `log()`
 - observes all reactive stream signals
 - uses Logger which can be set to categorized logging
 - Logger is set to INFO level
 - Log message format:
<date> <level> <thread> reactor.Flux.<Impl>. <count> <event>
 - thread – is the running thread name
 - Impl – is the actual Flux inner implementation – mostly Map
 - count – is the serial number of the inner implementation instance
 - `log(Logger logger, Level level, SignalType...signalTypes)`

```
flux.log(myCustomLogger, Level.INFO, SignalType.ON_NEXT, SignalType.ON_ERROR)
```



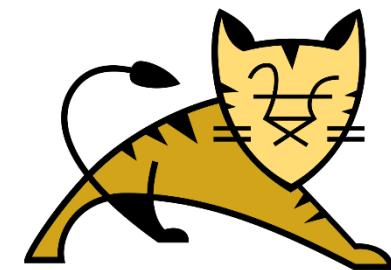
Spring WebFlux

Spring WebFlux

- Uses HTTP2 communication and data streaming
- Reactive web is not for executing faster
- Reactive web is for scaling with a limited number of threads
- WebFlux uses fixed-size thread pool (event loop workers) to handle requests
 - default size is available processors
- All SpringMVC forms & REST annotations are valid and relevant

Spring WebFlux

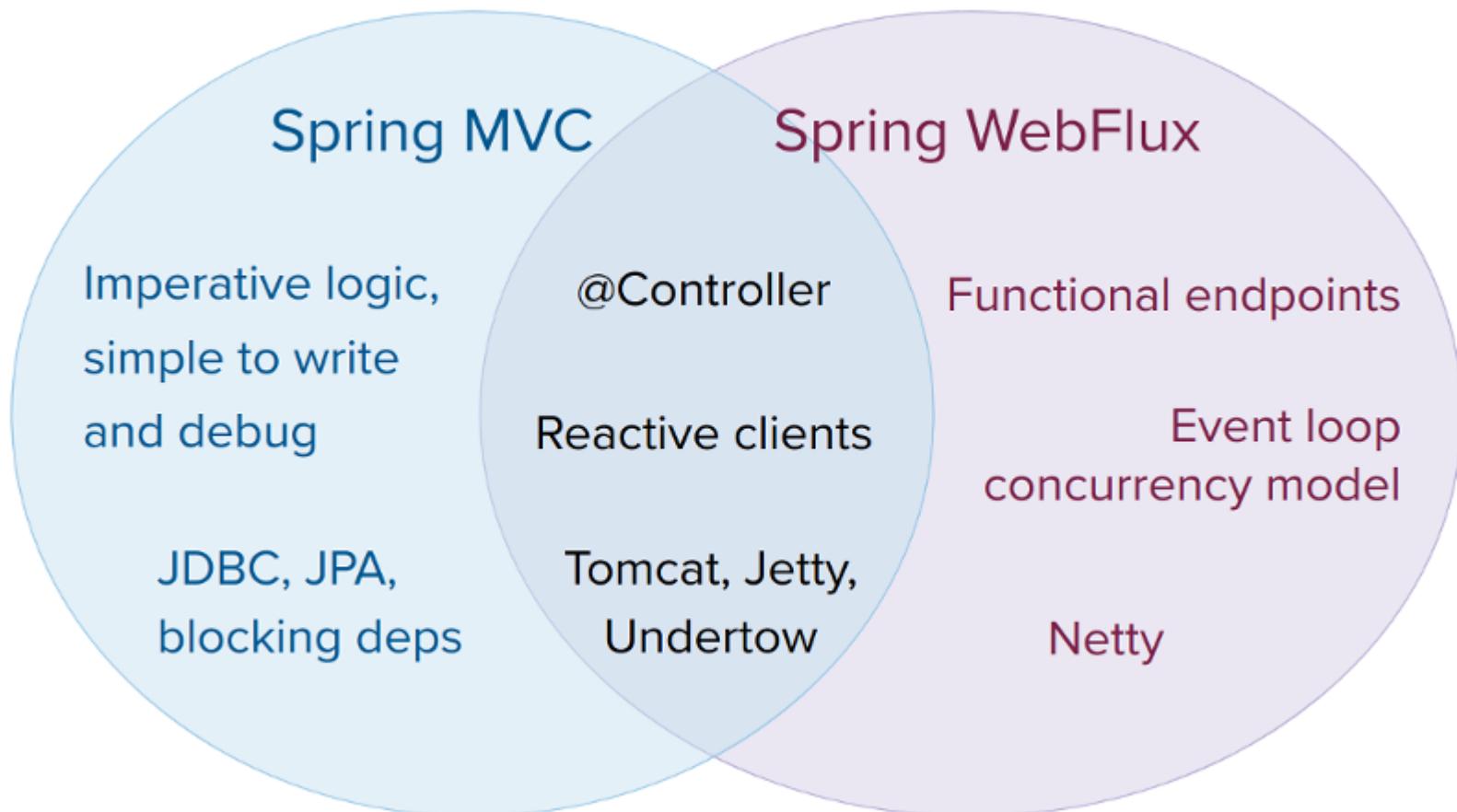
- Supported servers:
 - Netty (default)
 - Tomcat
 - Jetty
- Tomcat & Jetty
 - are based on Servlet API
 - relevant when using both SpringMVC & WebFlux
 - WebFlux uses low-level Servlet TCP communication for streaming
 - must support Servlet 3.1 (NIO)



jetty://

Spring WebFlux

- SpringMVC / Spring WebFlux



Spring WebFlux

- Configuration

- Maven dependency
 - Includes embedded Netty

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

- Spring property `spring.main.web-application-type` is set to ‘reactive’ by default
 - Set the web engine to be reactive
 - Disabling Spring-MVC is a must!

```
spring.main.web-application-type=reactive
```

Spring WebFlux

- Reactive Controller
 - also denoted with `@RestController`
 - returns `Mono<T>` & `Flux<T>` instead of `T` or `ResponseEntity<?>`
 - uses HTTP2 STREAM capable MIME types
 - `TEXT_EVENT_STREAM_VALUE` – for plain text & JSONS
 - `APPLICATION_OCTET_STREAM_VALUE` – for binary streams

Spring WebFlux

- Reactive Controller Example:

```
@RestController
public class ReactiveController{

    @GetMapping(value="item", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
    public Mono<Item> item(){
        ....
    }

    @GetMapping(value="items", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Item> items(){
        ....
    }
}
```

```
public class Item {

    private long id;
    private String name;

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Spring WebFlux

- JSON streaming results differently:

- `MediaType.APPLICATION_JSON` :

- browser blocks until response load completes EVEN when resulting with `Mono<T>` or `Flux<T>`
- Mono & Flux doesn't effect

```
[{"id":1,"name":"item1"}, {"id":2,"name":"item2"}, {"id":3,"name":"item3"}]
```

- `MediaType.TEXT_EVENT_STREAM_VALUE` :

- browser uses streaming for Mono & Flux
- Each stream emitted item gets 'data:'

```
data: {"id":1,"name":"item1"}
```

```
data: {"id":1,"name":"item1"}
```

```
data: {"id":2,"name":"item2"}
```

```
data: {"id":3,"name":"item3"}
```

Spring WebFlux

- JSON streaming results differently:
 - MediaType.TEXT_EVENT_STREAM_VALUE :
 - Same goes for simple types and primitives

data: 1

data: 2

data: 3

data: item1

data: item2

data: item3

Spring WebFlux

- `block()`, `blockFirst()`, `blockLast()` not supported for http streaming
 - these methods forces a thread to wait for outcomes
 - It is totally unwanted when
 - using limited number of threads
 - computation might take time
- Spring WebFlux will throw `IllegalStateException` in such cases

```
java.lang.IllegalStateException: block()/blockFirst()/blockLast() are blocking,  
which is not supported in thread reactor-http-nio-2
```

Spring WebFlux

- WebFlux reactor distributes events to handlers ('ctor-http-nio')
- This can be shown via logging web stream activity (*log()*)
- Example of handling http request for Flux<Value>:

```
INFO 23052 --- [main] o.s.b.web.embedded.netty.NettyWebServer : Netty started on port(s): 8080
INFO 23052 --- [main] react.Application : Started Application in 5.172 seconds (JVM running for 5.85)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onSubscribe(TakeUntilPredicateSubscriber)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : request(32)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@23f3a0b3)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@6f394543)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@1c6ca1f6)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@7780fdb5)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@49c83dce)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onNext/react.web.Value@6cfe6ae9)
INFO 23052 --- [ctor-http-nio-1] reactor.Flux.TakeUntil.1 : onComplete()
```

Parallel Mono/Flux

Parallel Flux/Mono

- Usually, subscriber thread is used to consume values from Flux/Mono
- If needed, streams may be handled with separate thread/s
- This can be done in 2 ways
 - Specify different workers to handle stream
 - Turn Flux<T> into ParallelFlux<T>
 - publishes to an array of Subscribers, in parallel 'rails' (or 'groups')
 - Scheduler objects helps in specifying workers

Parallel Flux

- Scheduler
 - Provides asynchronous context to operations
 - Uses reactor.schedulers thread pool
 - Default pool size is set to no' of available processors
 - No' of thread can be set via spring property:

```
reactor.schedulers.defaultPoolSize = 7
```
 - Can be factored via **Schedulers** utility class:
 - parallel() : hosts a single-threaded executor ready to participate in parallel execution
 - newParallel() : does the same but provides a new Scheduler instance for each subscriber

Parallel Flux/Mono

- Schedulers – factors scheduler objects:
 - fromExecutorService(ExecutorService)
 - uses given executor to handle stream
 - parallel() – uses reactor common pool
 - newParallel(String poolName)
 - newParallel(String poolName, int parallelism)
 - newParallel(String poolName, int parallelism, Boolean daemon)
 - does the same but with given name & number of executor-services – parallel capable
 - Unlike parallel(), instantiates a executors for any request context

Parallel Flux/Mono

- Example:
 - `Schedulers.parallel(...)` – handling elements with separate thread pool

```
public void handlingElementsInSeperateThread() {  
    Flux.just("yellow", "red", "white", "blue")  
        .subscribeOn(Schedulers.parallel())  
        .map(String::toUpperCase)  
        .log().subscribe();  
}
```

```
System.out.print("\n\nParallel: Handling all elements in seperate thread: \n");  
pex.handlingElementsInSeperateThread();
```

```
Parallel: Handling all elements in seperate thread:  
INFO 21096 --- [ main] reactor.Flux.Map.1 : onSubscribe(FluxMap.MapSubscriber)  
INFO 21096 --- [ main] reactor.Flux.Map.1 : request(unbounded)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.1 : onNext(YELLOW)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.1 : onNext(RED)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.1 : onNext(WHITE)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.1 : onNext(BLUE)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.1 : onComplete()
```

Parallel Flux/Mono

- Example:
 - `Schedulers.parallel(...)` – handling each element with available thread

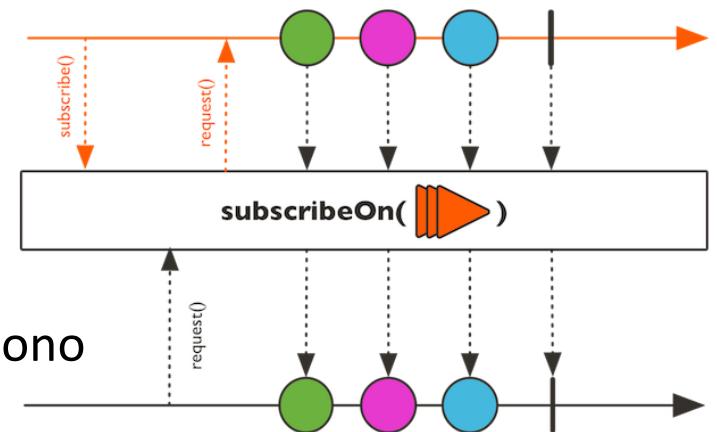
```
public void handleEachElementInSeparateThread() {  
    Flux.just("yellow", "red", "white", "blue")  
        .flatMap(s -> Mono.just(s))  
        .subscribeOn(Schedulers.parallel()).delayElement(Duration.ofSeconds((int)(Math.random()*4)))  
        .map(String::toUpperCase)  
        .log().subscribe();  
}
```

```
System.out.print("\n\nParallel: Handling each element in separate thread: \n");  
pex.handleEachElementInSeparateThread();
```

```
Parallel: Handling each element in separate thread:  
INFO 21096 --- [ main] reactor.Flux.Map.2 : onSubscribe(FluxMap.MapSubscriber)  
INFO 21096 --- [ main] reactor.Flux.Map.2 : request(unbounded)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.2 : onNext(YELLOW)  
INFO 21096 --- [ parallel-1] reactor.Flux.Map.2 : onNext(WHITE)  
INFO 21096 --- [ parallel-2] reactor.Flux.Map.2 : onNext(BLUE)  
INFO 21096 --- [ parallel-2] reactor.Flux.Map.2 : onNext(RED)  
INFO 21096 --- [ parallel-2] reactor.Flux.Map.2 : onComplete()
```

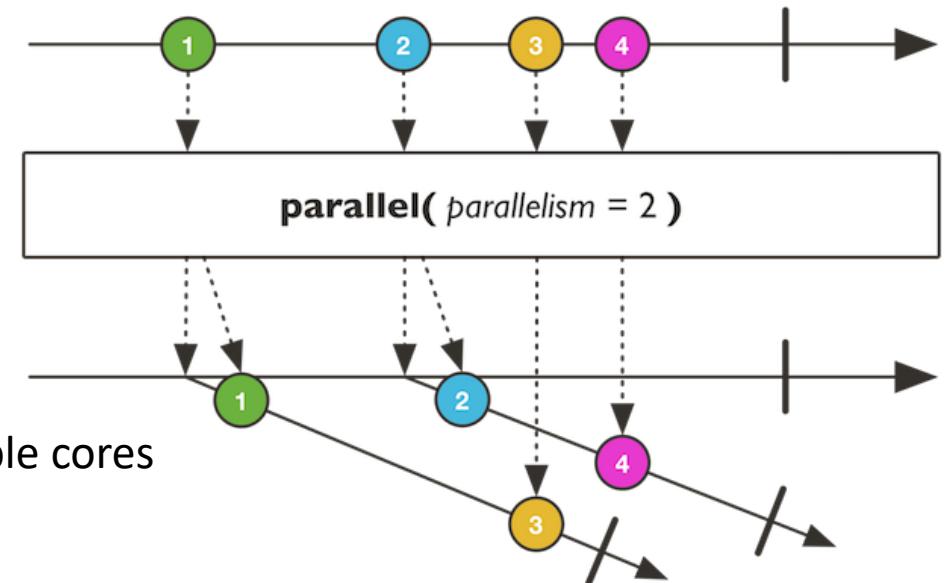
Parallel Flux

- Setting Flux/Mono with workers
 - Forking stream element processing:
 - This is done via `subscribeOn(Scheduler asynchHandler)` method
 - Actually – activates the parallel stream
 - Forking each element handling in a stream
 - For this we map `Flux<T>` into `Flux<Mono<T>>` and fork each Mono
 - Or use built-in Flux parallel mechanism



Parallel Flux

- Flux<T> into ParallelFlux<T>
 - parallel()
 - parallel (int parallelism)
 - parallel(int parallelism, int prefetch)
 - Prepares ParallelFlux<T> from current Flux<T>
 - default number of workers is set according to available cores
 - specify number of workers via ‘parallelism’
 - mechanism is round-robin-based
 - ‘prefetch’ sets the max amount of values for each worker at a time
 - Caller thread is used to dispatch work
 - Must use ParallelFlux.runOn(Scheduler schedule) to enable parallel subscription



Parallel Flux

- Example:
 - parallel(...)

```
public void parallelFlux() {  
    Flux.just("yellow", "red", "white", "blue")  
        .parallel(2)  
        .map(String::toUpperCase)  
        .runOn(Schedulers.parallel())  
        .log().subscribe();  
}
```

```
System.out.print("\n\nParallel: ParallelFlux: \n");  
pex.parallelFlux();
```

```
Parallel: ParallelFlux:  
INFO 22704 --- [ main] reactor.Parallel.RunOn.3 : onSubscribe([Fuseable]  
INFO 22704 --- [ main] reactor.Parallel.RunOn.3 : request(unbounded)  
INFO 22704 --- [ main] reactor.Parallel.RunOn.3 : onSubscribe([Fuseable]  
INFO 22704 --- [ main] reactor.Parallel.RunOn.3 : request(unbounded)  
INFO 22704 --- [ parallel-1] reactor.Parallel.RunOn.3 : onNext(YELLOW)  
INFO 22704 --- [ parallel-2] reactor.Parallel.RunOn.3 : onNext(RED)  
INFO 22704 --- [ parallel-1] reactor.Parallel.RunOn.3 : onNext(WHITE)  
INFO 22704 --- [ parallel-2] reactor.Parallel.RunOn.3 : onNext(BLUE)  
INFO 22704 --- [ parallel-1] reactor.Parallel.RunOn.3 : onComplete()  
INFO 22704 --- [ parallel-2] reactor.Parallel.RunOn.3 : onComplete()
```

Parallel Flux

- Reducing ‘rails’ from parallel back to Flux<T>
 - ParallelFlux.sequential()
 - Returns Flux<T>
 - Merges values from all ‘rails’ in a round-robin fashion

Parallel Flux

- Example:
 - sequential()

Parallel: ParallelFluxSequential:

```
INFO 29432 --- [main] reactor.Flux.Map.5 : onSubscribe(FluxMap.MapSubscriber)
INFO 29432 --- [main] reactor.Flux.Map.5 : request(unbounded)
INFO 29432 --- [main] reactor.Parallel.RunOn.4 : onSubscribe([Fuseable])
INFO 29432 --- [main] reactor.Parallel.RunOn.4 : request(256)
2019-06-04 20:22:03.621 INFO 29432 --- [           main] reactor.Parallel.RunOn.4 : onSubscribe([Fuseable])
2019-06-04 20:22:03.622 INFO 29432 --- [           main] reactor.Parallel.RunOn.4 : request(256)
2019-06-04 20:22:03.624 INFO 29432 --- [parallel-2] reactor.Parallel.RunOn.4 : onNext(YELLOW)
2019-06-04 20:22:03.624 INFO 29432 --- [parallel-3] reactor.Parallel.RunOn.4 : onNext(RED)
2019-06-04 20:22:03.625 INFO 29432 --- [parallel-2] reactor.Flux.Map.5 : onNext(yellow)
2019-06-04 20:22:03.625 INFO 29432 --- [parallel-3] reactor.Parallel.RunOn.4 : onNext(BLUE)
2019-06-04 20:22:03.625 INFO 29432 --- [parallel-2] reactor.Flux.Map.5 : onNext(red)
2019-06-04 20:22:03.626 INFO 29432 --- [parallel-2] reactor.Parallel.RunOn.4 : onNext(WHITE)
2019-06-04 20:22:03.626 INFO 29432 --- [parallel-3] reactor.Flux.Map.5 : onNext(blue)
2019-06-04 20:22:03.626 INFO 29432 --- [parallel-2] reactor.Parallel.RunOn.4 : onComplete() (rail-1 ends)
2019-06-04 20:22:03.626 INFO 29432 --- [parallel-3] reactor.Flux.Map.5 : onNext(white)
2019-06-04 20:22:03.627 INFO 29432 --- [parallel-3] reactor.Parallel.RunOn.4 : onComplete() (rail-2 ends)
2019-06-04 20:22:03.628 INFO 29432 --- [parallel-3] reactor.Flux.Map.5 : onComplete()
```

```
public void parallelFluxSequential() {
    Flux.just("yellow", "red", "white", "blue")
        .parallel(2)
        .map(String::toUpperCase)
        .runOn(Schedulers.parallel())
        .log()
        .sequential()
        .map(String::toLowerCase)
        .log()
        .subscribe();
}
```

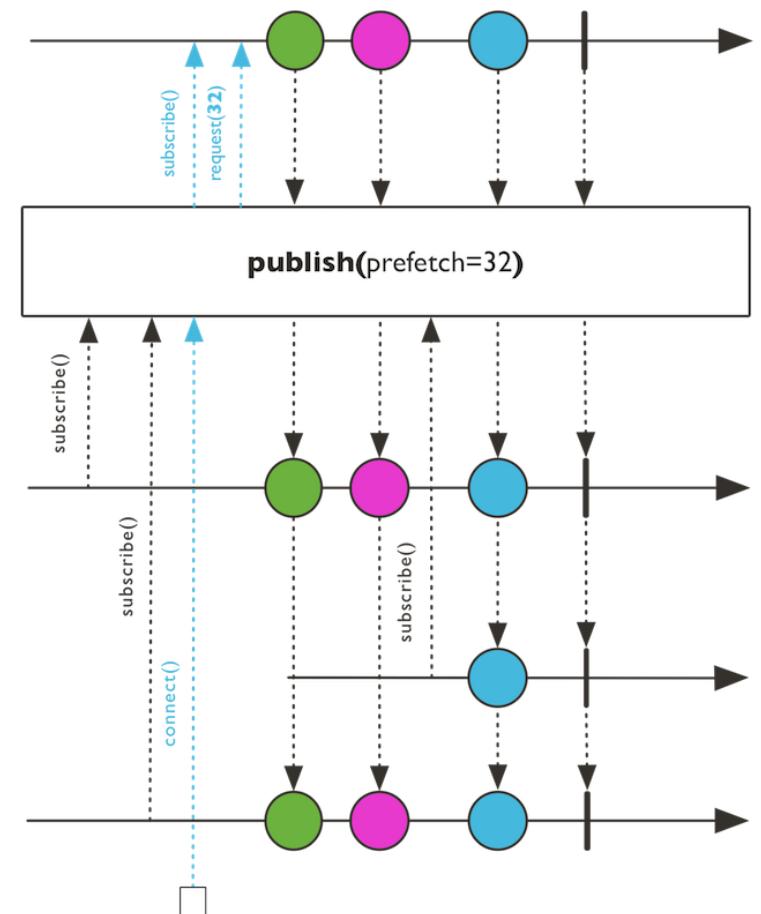
Connectable Flux

Connectable Flux

- Connectable Flux provides the following capabilities:
 - Join upstream during on-going subscription – Hot Stream
 - Cold Streams – which are the default, forces subscribers to take from the beginning of the stream
 - Hot Streams – once connected, allows subscribers to join at any time while streaming data
 - Delay on-going subscription until minimum number of subscribers performs
 - Turn a whole Flux<T> or parts of it into hot streams - replaying

Connectable Flux

- Flux<T> can be transformed into ConnectableFlux<T>
 - publish()
 - publish(int prefetch)
 - returns ConnectableFlux<T>
 - connectable Flux is not ‘turned-on’ yet
 - time to connect...
 - NOTE: Connectable Flux are parallel capable



Connectable Flux

- ConnectableFlux<T> must connect upstream in order to provide Flux<T>
 - autoConnect()
 - autoConnect(int minSubscribers)
 - automatically connects upstream when specified amount of 'minSubscribers' subscribes
 - connect()
 - automatically connects upstream
 - refCount()
 - refCount(int minSubscribers)
 - automatically connects upstream after specified amount of 'minSubscribers' and disconnects after all subscribers canceled or completed

Connectable Flux

- Example:
 - `publish(...)` & `connect`

```
public void connectableExample() {  
    Flux<String> source = Flux.just("aaa", "bbb", "ccc", "ddd", "eee", "fff")  
        .delayElements(Duration.ofSeconds(1))  
        .doOnNext(System.out::println)  
        .map(String::toUpperCase);  
ConnectableFlux<String> connectable = source.publish();  
    //subscription is now enabled  
connectable.connect();  
    //start with single subscriber  
    connectable.subscribe(d -> System.out.println("Subscriber 1: "+d));  
    //delay before adding subscriber  
    try {Thread.sleep(2000);} catch(Exception e) {}  
    //join with another subscriber  
    connectable.subscribe(d -> System.out.println("Subscriber 2: "+d));  
}
```

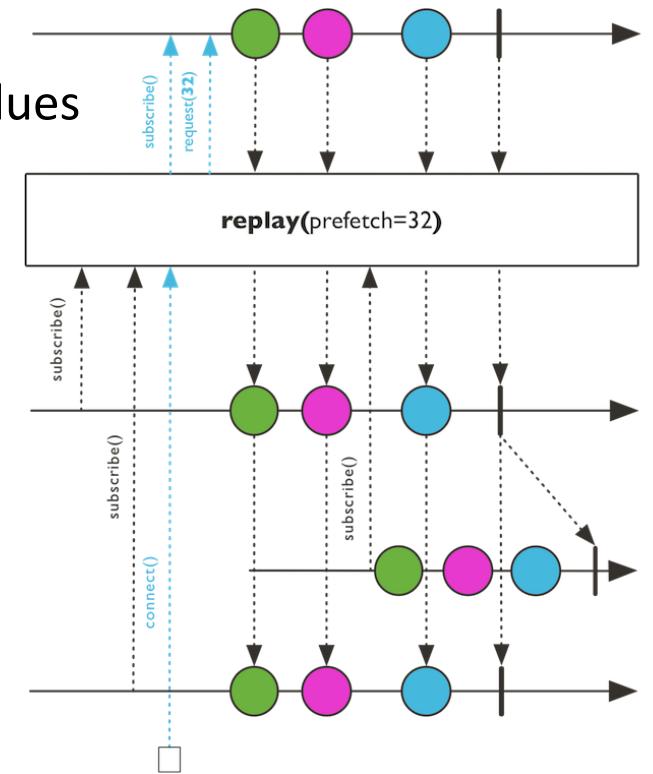
```
System.out.print("\n\nConnectable: ConnectableFlux: \n");  
cex.connectableExample();
```

Connectable: ConnectableFlux:
aaa
Subscriber 1: AAA
bbb
Subscriber 1: BBB
Subscriber 2: BBB
ccc
Subscriber 1: CCC
Subscriber 2: CCC
ddd
Subscriber 1: DDD
Subscriber 2: DDD
eee
Subscriber 1: EEE
Subscriber 2: EEE
fff
Subscriber 1: FFF
Subscriber 2: FFF

Connectable Flux

- Turning parts of `Flux<T>` into `ConnectableFlux<T>`

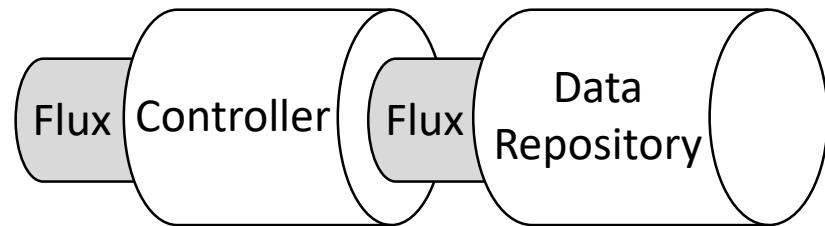
- `replay()`
 - returns `ConnectableFlux` which contains all further subscribed values
 - grows infinitely
 - complete and error terminations are also streamed
- `replay(int history)`
- `replay(Duration ttl)`
 - same but keeps some history
 - history – holds specified amount of latest subscribed values
 - ttl - holds latest subscribed values according to Time-To-Live



Reactive Repository with MongoDB

Reactive Repository with MongoDB

- Reactive supported DBs provides a Reactor mechanism as well
 - DB Server plays the role of the Reactor
 - The reactor distributes events to Connections which are the handlers
 - Connections uses different thread resources to prevent reactor blocking
- Spring reactive repository
 - ReactiveCrudRepository interface
 - Results with Mono<T> & Flux<T> instead of T & List<T>
 - Implements Reactor DP, uses reactive driver implementation



Reactive Repository with MongoDB

- MongoDB
 - DB server may be standalone or embedded
 - Extends ReactiveCrudRepository with ReactiveMongoRepository
 - Offers more querying capabilities
 - Example – connecting and querying log results show reactor & handler threads:



```
INFO 7072 --- [localhost:27017] org.mongodb.driver.connection : Opened connection  
[connectionId{localValue:1, serverValue:29}] to localhost:27017  
INFO 7072 --- [localhost:27017] org.mongodb.driver.cluster : Monitor thread successfully connected to server  
with description ServerDescription{  
    address=localhost:27017, type=STANDALONE, state=CONNECTED, ok=true,  
    version=ServerVersion{versionList=[4, 0, 1]}, minWireVersion=0, maxWireVersion=7,  
    maxDocumentSize=16777216, logicalSessionTimeoutMinutes=30, roundTripTimeNanos=2740937}  
INFO 7072 --- [ntLoopGroup-2-2] org.mongodb.driver.connection
```

Reactive Repository with MongoDB

- Configuration

- MongoDB Server

- Standalone

- Download from: <https://www.mongodb.com/download-center#community>

- Embedded MongoDB

- Add Maven dependencies:

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <version>1.50.5</version>
</dependency>
<dependency>
    <groupId>cz.jirutka.spring</groupId>
    <artifactId>embedmongo-spring</artifactId>
    <version>RELEASE</version>
</dependency>
```

Reactive Repository with MongoDB

- Configuration
 - Maven dependency
 - Reactive driver for MongoDB & Spring reactive repositories
 - If MongoDB is installed – this is the only dependency required

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Reactive Repository with MongoDB

- Configuration
 - Spring property `spring.data.mongodb.database` is set to ‘reactive’ by default
 - Sets/creates DB name
 - Optional, default name is ‘test’
 - `spring.data.mongodb.database=reactiveDB`
 - SpringBoot also uses ‘localhost’ and 27017 as default port

Reactive Repository with MongoDB

- Creating repository

```
public interface ReactivePersonRepository extends ReactiveCrudRepository<Person, String>{  
    Flux<Person> findAllByAge(int age);  
}
```

- Remember: Flux are executed lazily

- Use subscribe() to get it going

```
@Autowired  
private ReactivePersonRepository repo;  
...  
Iterable people = //populate with Person instances  
repo.saveAll(people).subscribe();  
...  
repo.deleteAll().subscribe();  
}
```

```
@Document  
public class Person {  
  
    @Id  
    private String id;  
    private String name;  
    private int age;  
  
    //getters & setters  
    ...  
}
```

Reactive Repository with MongoDB

- Using reactive repository with REST

```
@RestController
@RequestMapping("person")
public class ReactiveTodoController {

    @Autowired
    private ReactiveTodoRepository repo;

    @GetMapping(value="add/{id}/{name}/{age}", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
    public Mono<Person> add( @PathVariable("id") String id,
                             @PathVariable("name") String name,
                             @PathVariable("age") int age) {
        Person p=new Person();
        p.setId(id);
        p.setName(name);
        p.setAge(age);
        return repo.save(p); //results with Mono<T> & invoked via web subscription
    }

    ...
}
```

Reactive Repository with MongoDB

- Using reactive repository with REST – cont.

```
...  
  
@GetMapping(value="all", produces=MediaType.TEXT_EVENT_STREAM_VALUE)  
public Flux<Person> getAll() {  
    return repo.findAll(); //results with Flux<T> & invoked via web subscription  
}  
  
@GetMapping(value="all/age/{age}", produces=MediaType.TEXT_EVENT_STREAM_VALUE)  
public Flux<Person> getAll(@PathVariable("age") int age) {  
    return repo.findAllByAge(age); //same here...  
}  
}
```

Reactive Repository with MongoDB

- Using reactive repository with REST – Sorting
 - Reactive DB are parallel by nature – which might lead to records disorder
 - For sorted results use: repo.findAll(Sort)
 - Must use ReactiveMongoRepository – not ReactiveCrudRepository

```
...
@GetMapping(value="all", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Person> getAll() {
    return repo.findAll(Sort.by(Sort.Direction.ASC, "id"));
}
```

Reactive Repository with MongoDB

- Testing:
 - /person/add/a/David/20

```
data:{"id":"a", "name":"David", "age":20}
```

- /person/all

```
data:{"id":"b", "name":"Bob", "age":35}
```

```
data:{"id":"y", "name":"Eve", "age":20}
```

```
data:{"id":"w", "name":"Dan", "age":45}
```

```
data:{"id":"a", "name":"David", "age":20}
```

- /person/all/age/20

```
data:{"id":"y", "name":"Eve", "age":20}
```

```
data:{"id":"a", "name":"David", "age":20}
```