WELCOME
&
THANK YOU

<Cre8tive Devs. Software/>

Reactive & Event Driven Programming

Level 1

MEET YOUR BITTY BYTE TEAM

TANGY F.

CEO

EDDIE K.

DEVELOPER

WILLIAM D.

DEVELOPER

<Cre8tive Devs. Software/>

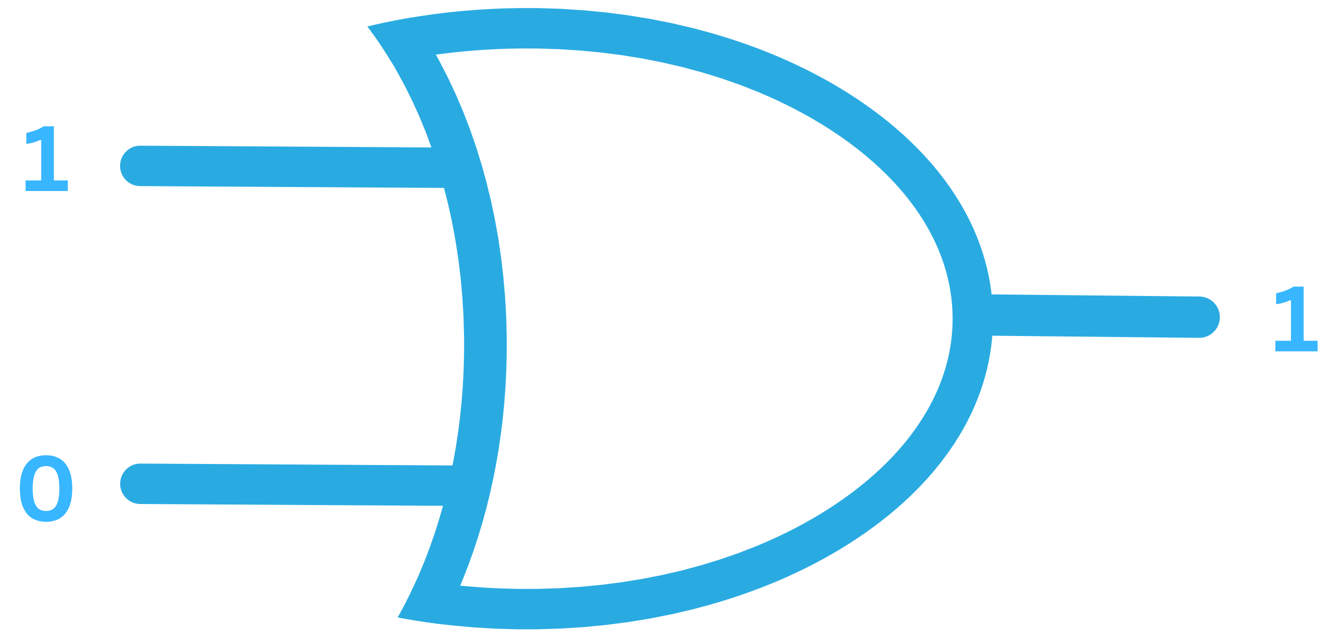
MOBILE APPS•WEBSITES•AR•VIDEO GAMES•SECURITY

Rapid Review

TRIVIA

Rapid Trivia

What is the name of the logic gate bellow?

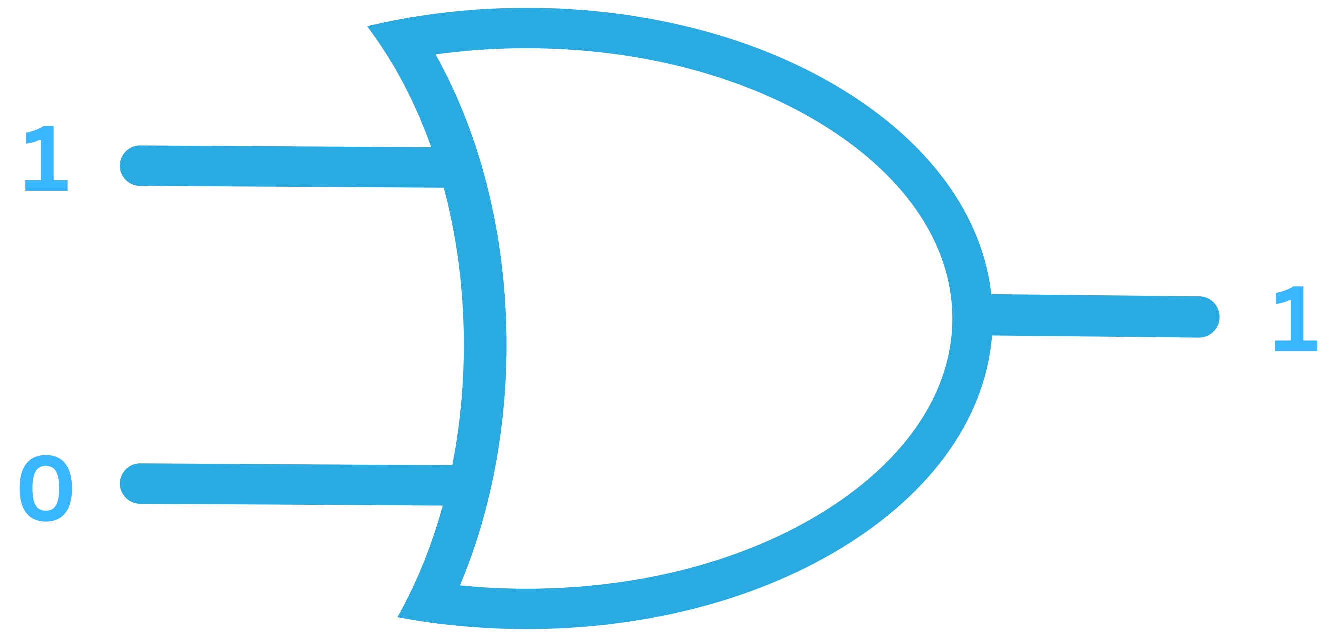


Rapid Review

TRIVIA

Rapid Trivia

This is an OR logic gate



CODING EXERCISE

On the reactive Flux that sends integers from 1 to 50, use **.doOnNext()** to see the order in which execution happens;

- Print “Sending” plus item number before producer sends item
- Print “Receiving” plus item number before consumer process item
- Do not change the current print of the processed item

Let's see the code.

TODAY'S AGENDA

DAY 3

1. Reactive Operators, concat(), distinct(), zip() with Flux, zipWith()
2. Error Handling
3. Scheduler and Threading

<Cre8tive Devs. Software/>

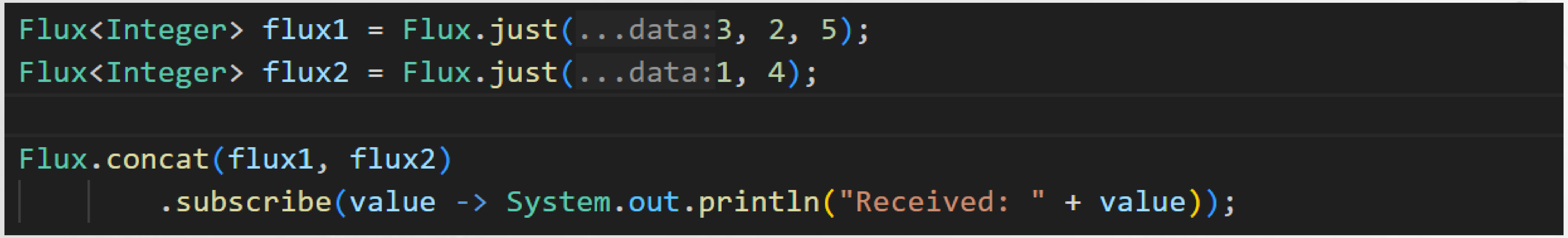
DAY 3
LESSON 1

Operators;

- concat()
- distinct()
- zip() with Flux
- zipWith()

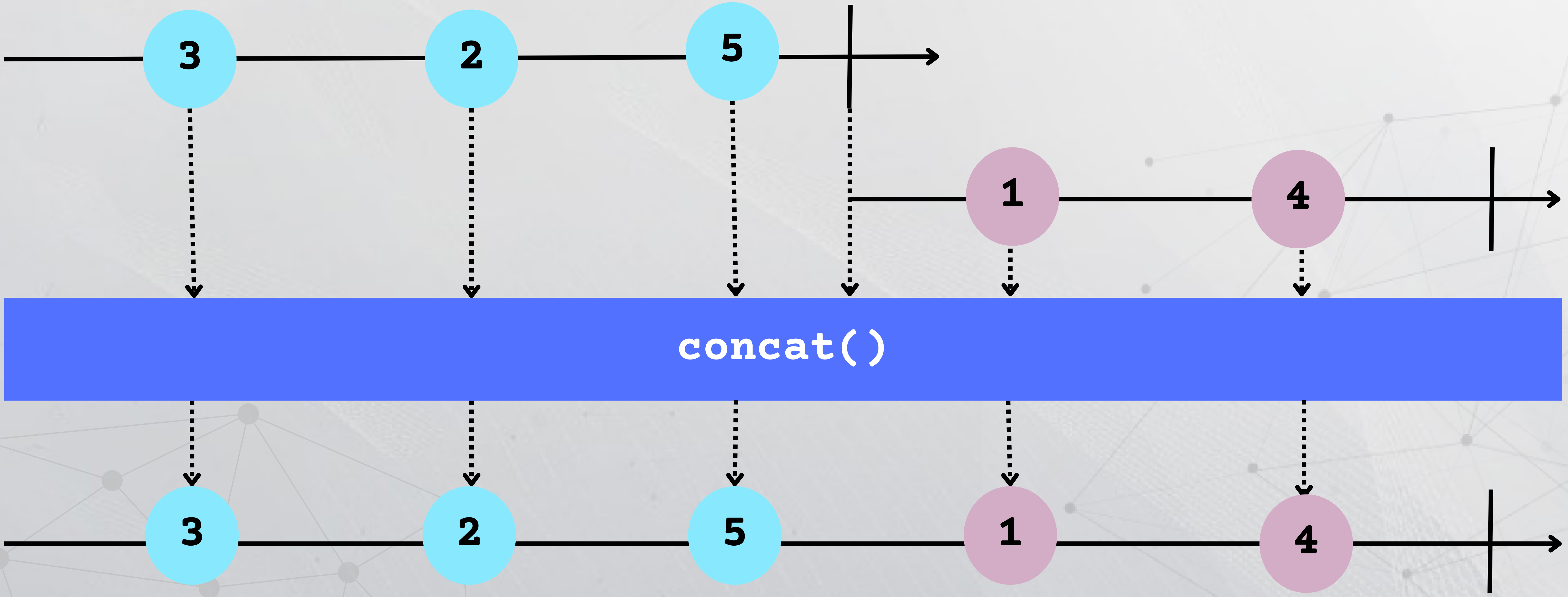
<Cre8tive Devs. Software/>
MOBILE APPS•WEBSITES•AR•VIDEO GAMES•SECURITY

CONCAT OPERATOR



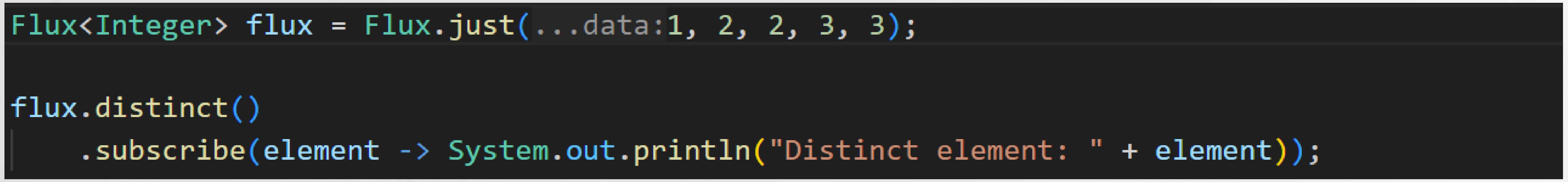
```
Flux<Integer> flux1 = Flux.just(...data:3, 2, 5);
Flux<Integer> flux2 = Flux.just(...data:1, 4);

Flux.concat(flux1, flux2)
        .subscribe(value -> System.out.println("Received: " + value));
```



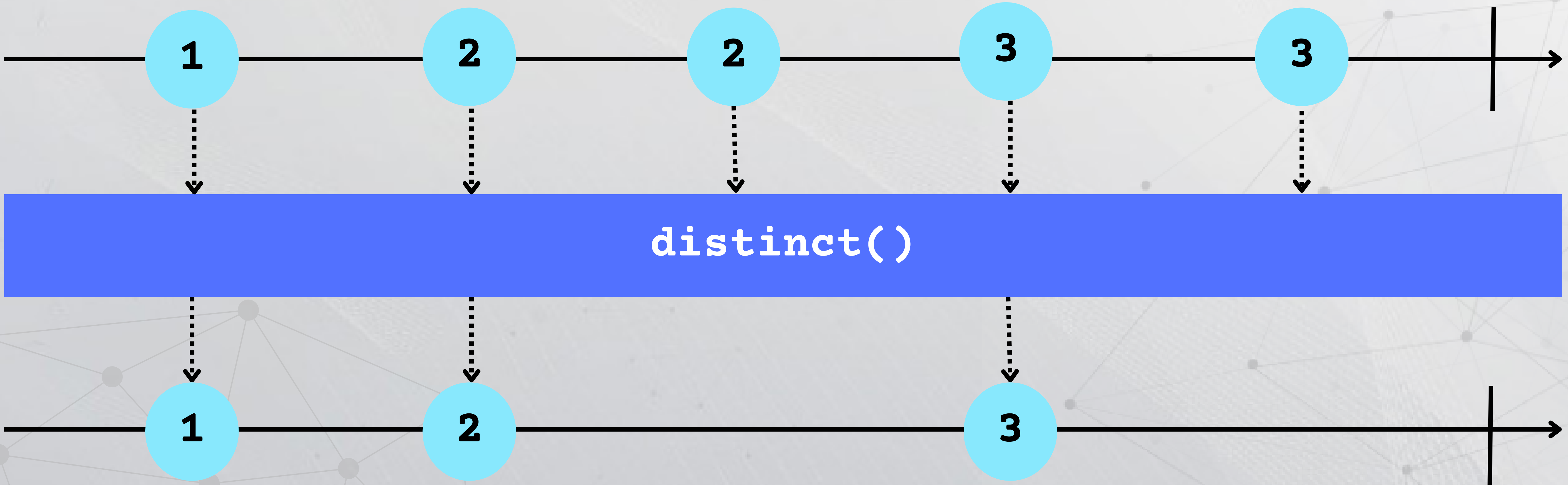
Let's see the code.

DISTINCT OPERATOR



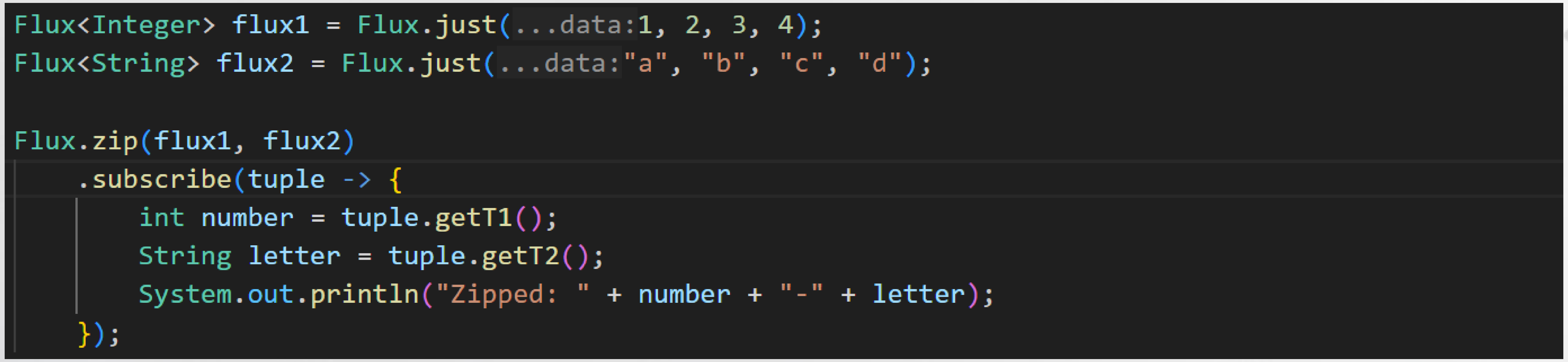
```
Flux<Integer> flux = Flux.just(...data:1, 2, 2, 3, 3);

flux.distinct()
    .subscribe(element -> System.out.println("Distinct element: " + element));
```



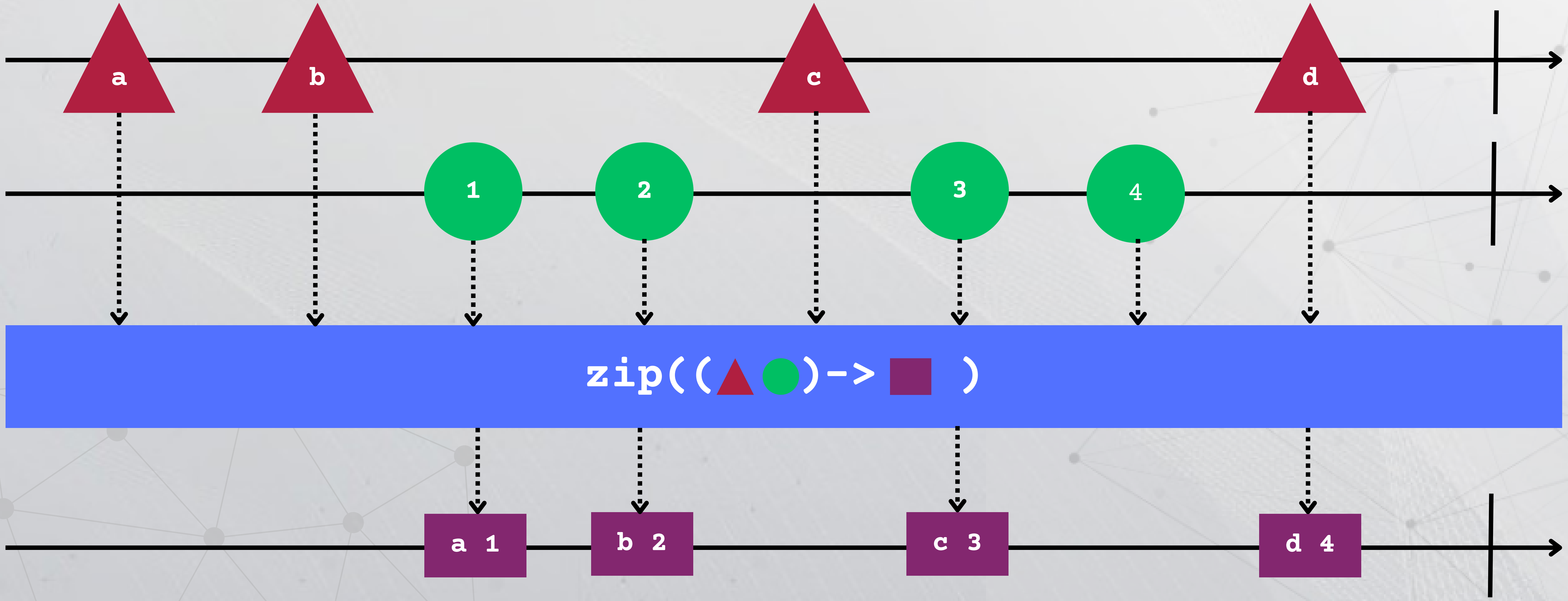
Let's see the code.

ZIP OPERATOR WITH FLUX



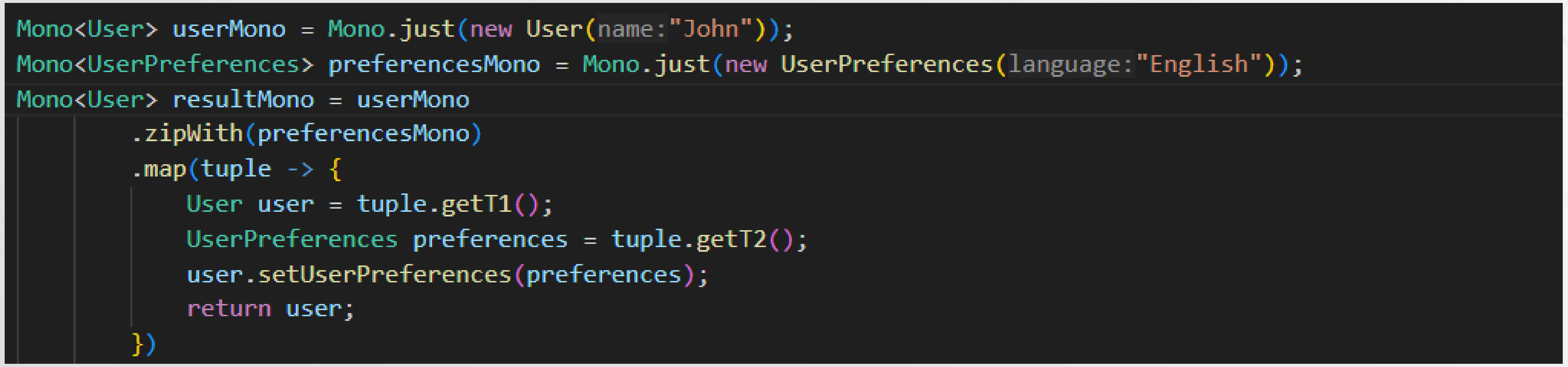
```java
Flux<Integer> flux1 = Flux.just(...data:1, 2, 3, 4);
Flux<String> flux2 = Flux.just(...data:"a", "b", "c", "d");

Flux.zip(flux1, flux2)
    .subscribe(tuple -> {
        int number = tuple.getT1();
        String letter = tuple.getT2();
        System.out.println("Zipped: " + number + "-" + letter);
    });
```

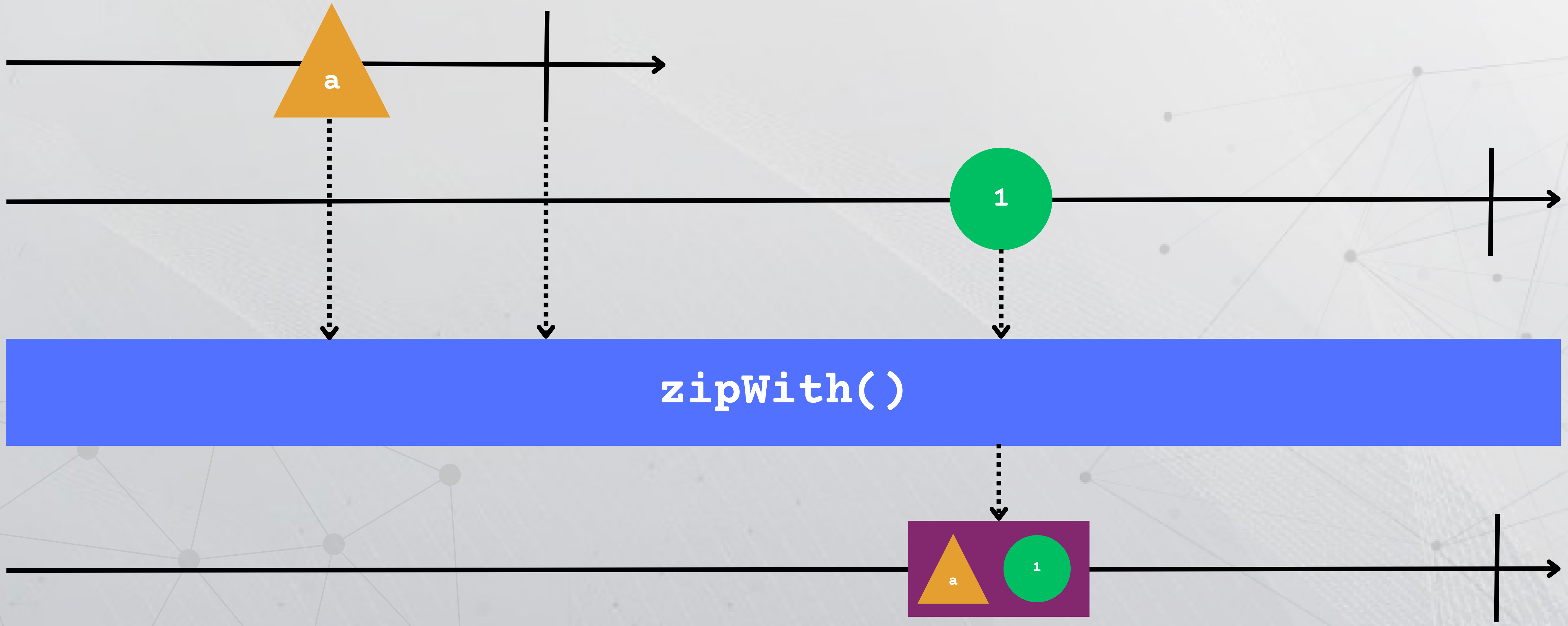


Let's see the code.

ZIPWITH OPERATOR



```java
Mono<User> userMono = Mono.just(new User(name:"John"));
Mono<UserPreferences> preferencesMono = Mono.just(new UserPreferences(language:"English"));
Mono<User> resultMono = userMono
        .zipWith(preferencesMono)
        .map(tuple -> {
            User user = tuple.getT1();
            UserPreferences preferences = tuple.getT2();
            user.setUserPreferences(preferences);
            return user;
        })
```



Let's see the code.

DAY 3
LESSON 2

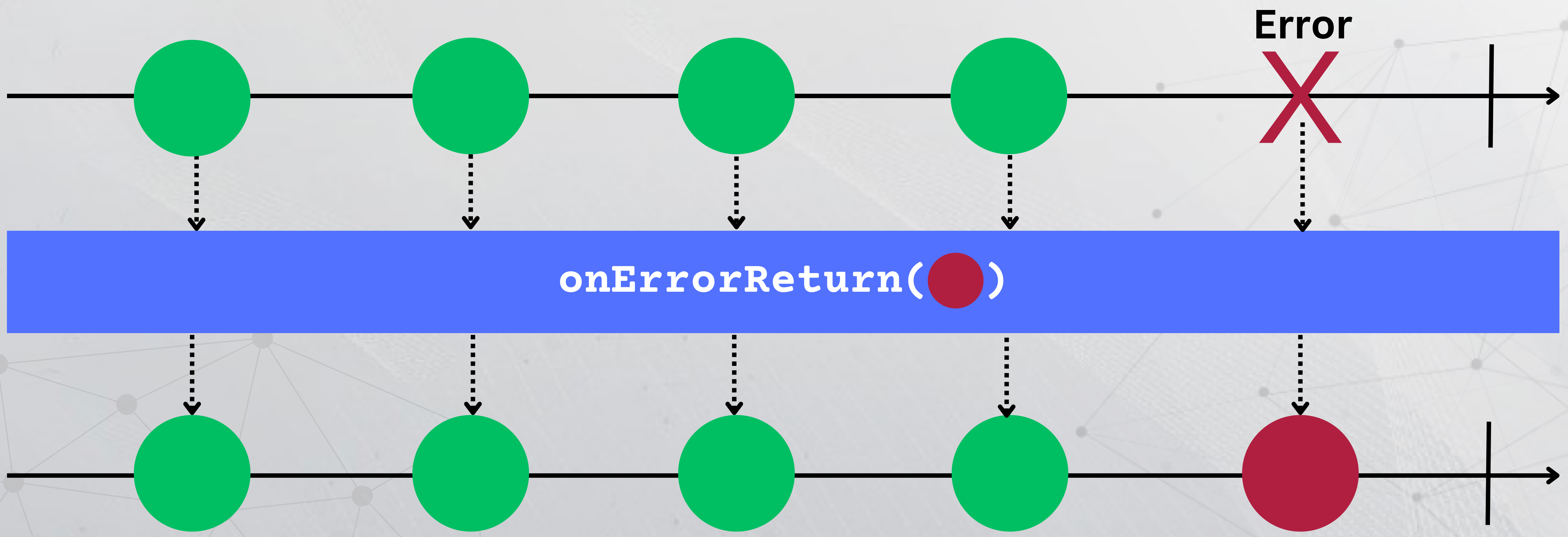
Error Handling

- onErrorReturn
- onErrorResume
- retry

ERROR HANDLING

onErrorReturn

```
flux.onErrorReturn(-1)
        .subscribe(
            item -> System.out.println("Received item: " + item),
            error -> System.err.println("Error: " + error)
        );
```

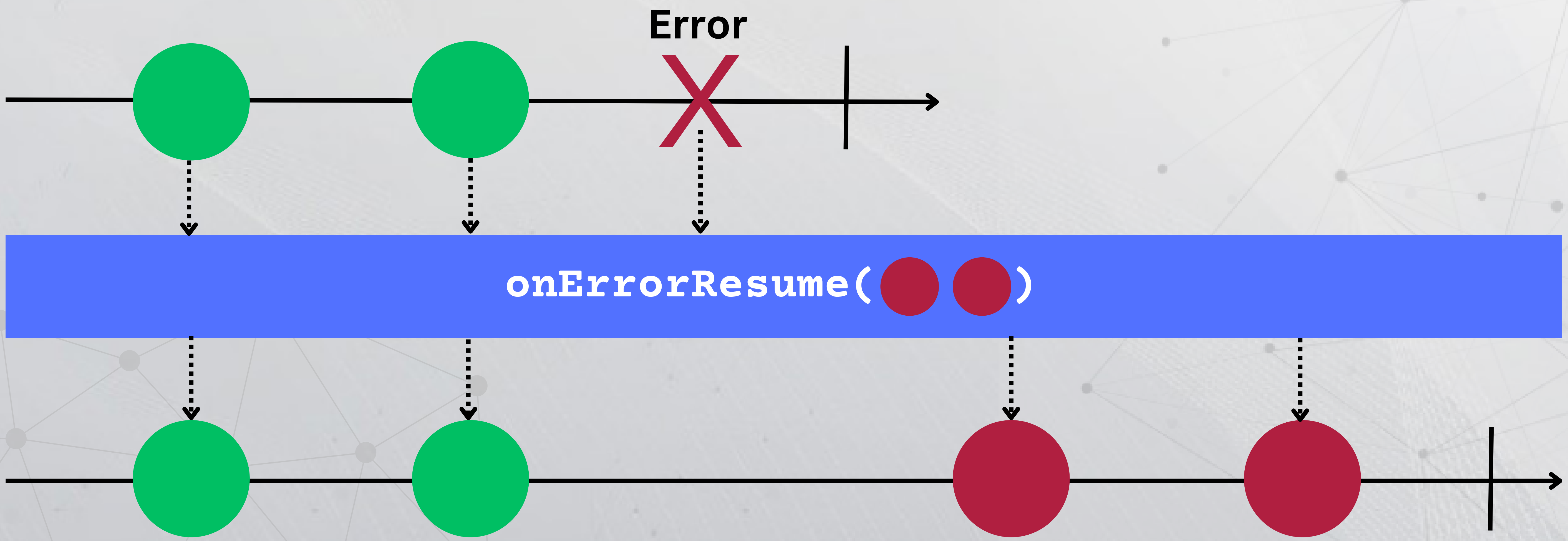


Let's see the code.

ERROR HANDLING

onErrorResume

```
flux.onErrorResume(e -> {
    System.err.println("Handling error: " + e.getMessage());
    return Flux.range(start:100, count:5);  // Provide a fallback publisher
}).subscribe(
    item -> System.out.println("Received item: " + item),
    error -> System.err.println("Error: " + error)
);
```

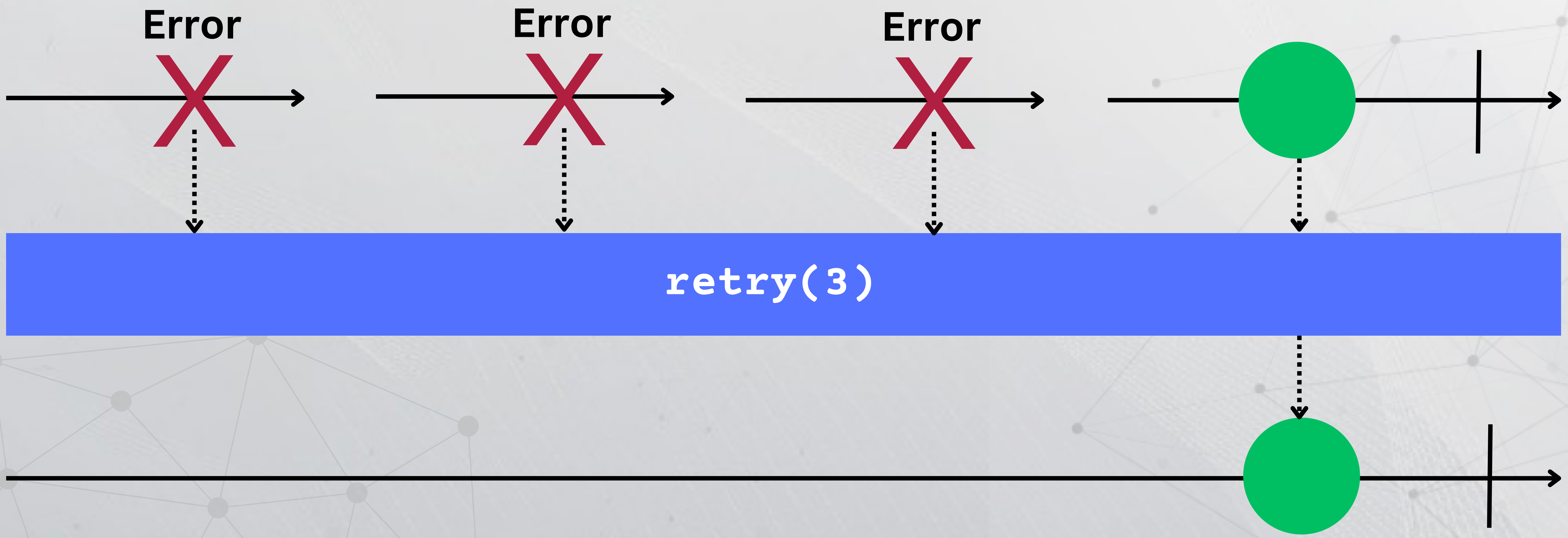


Let's see the code.

ERROR HANDLING

retry

```
flux.retry(numRetries:3)  // Retry 3 times
        .subscribe(
            item -> System.out.println("Received item: " + item),
            error -> System.err.println("Error: " + error)
        );
```

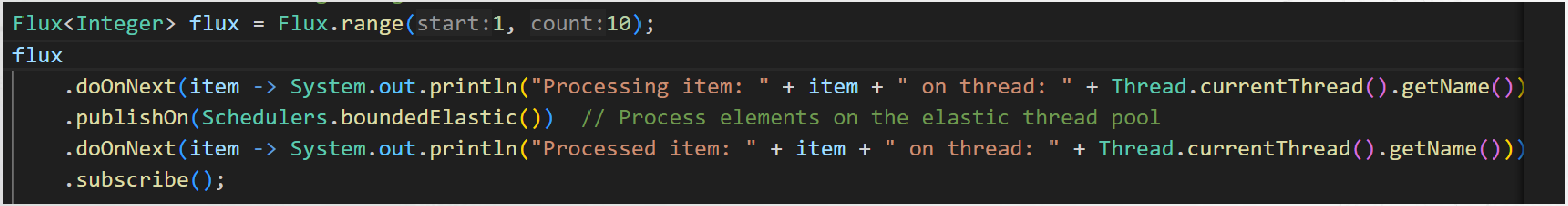


Let's see the code.

DAY 3
LESSON 2

Scheduler & Threading

SCHEDULER AND THREADING



```java
Flux<Integer> flux = Flux.range(start:1, count:10);
flux
    .doOnNext(item -> System.out.println("Processing item: " + item + " on thread: " + Thread.currentThread().getName())
    .publishOn(Schedulers.boundedElastic())  // Process elements on the elastic thread pool
    .doOnNext(item -> System.out.println("Processed item: " + item + " on thread: " + Thread.currentThread().getName()))
    .subscribe();
```

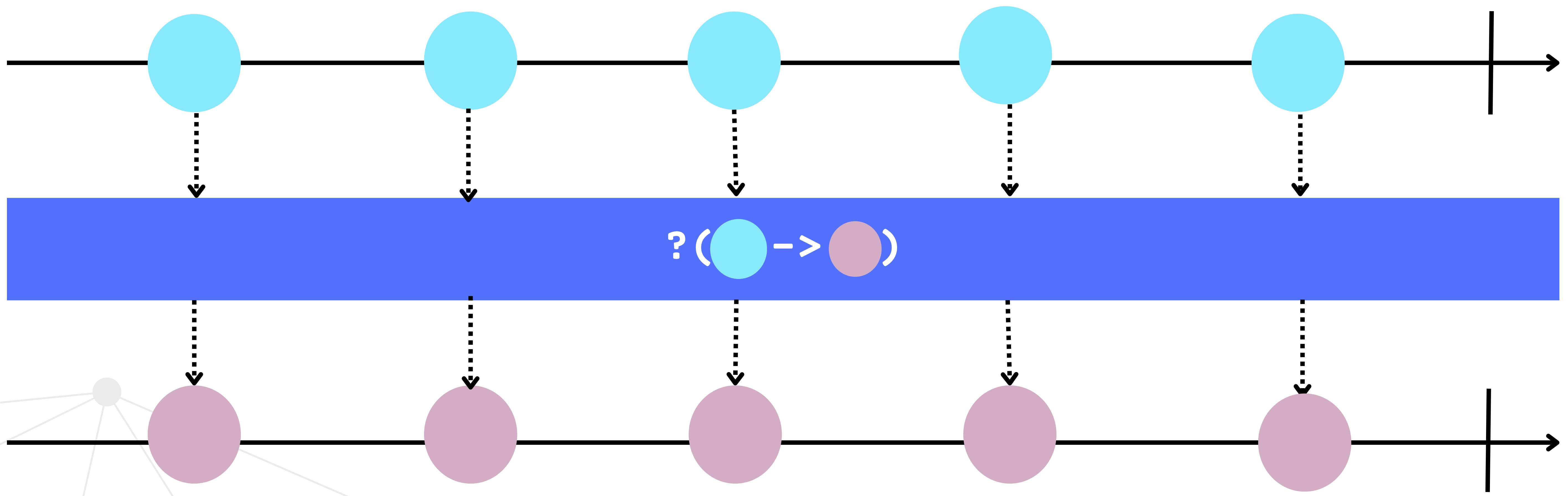
Available Schedulers;

Let's see the code.

- **Immediate:** Execute immediately in current thread
  - For immediate execution of a piece of code, often used for testing, debugging, or situations where you need to inline execution.
- **Single:** Execute in a single reusable thread of execution
  - For sequential processing of a series of operations without worrying about concurrency.
- **Parallel:** Provides a fixed pool of threads based on CPU cores for parallel execution
  - Suitable for CPU-bound tasks with a predictable workload
- **Bounded Elastic:** Provides an elastic thread pool that can dynamically adjust the number of threads based on the workload
  - For situations where the workload may vary or be unpredictable.

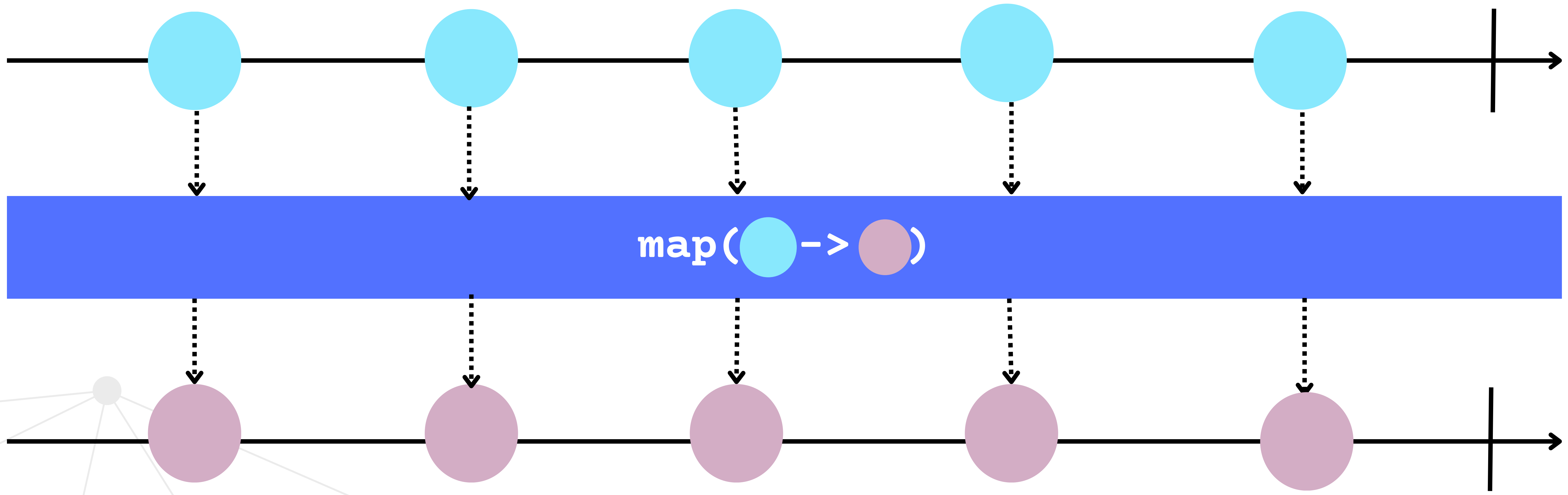
LESSON QUESTION

Which operator will produce the output as described bellow?



LESSON ANSWER

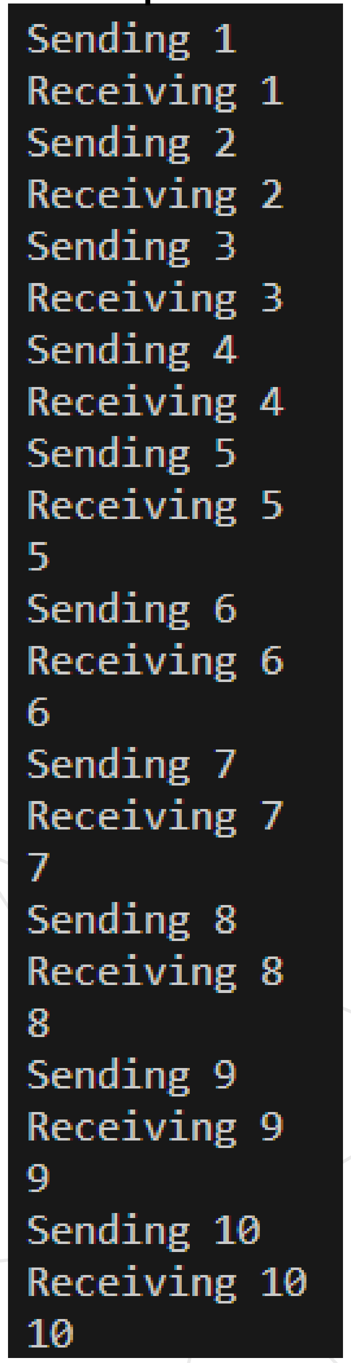
map() operator.



CODING EXERCISE

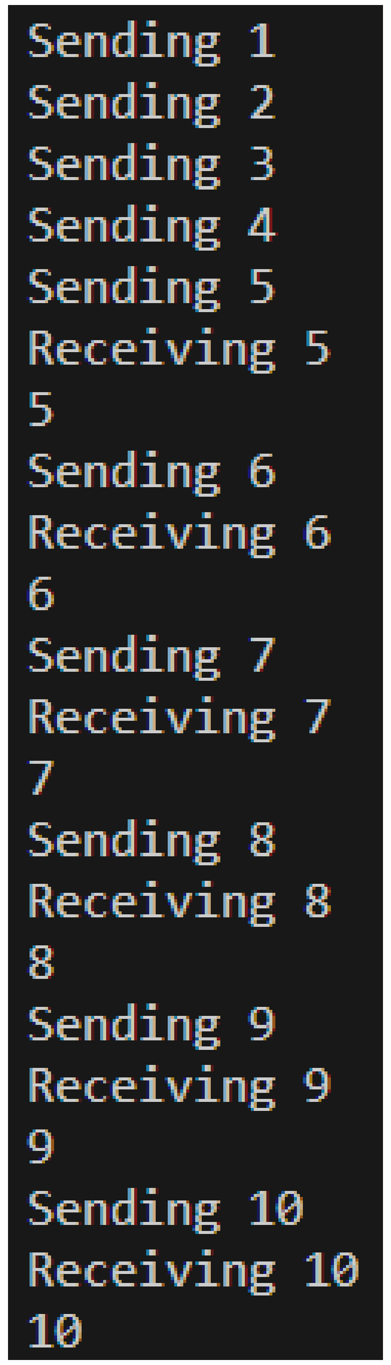
On the reactive Flux that sends integers from 1 to 50, change the limit from 50 to 10, add a filter to include items greater than 5 as follow;

Make the filter to produce this output



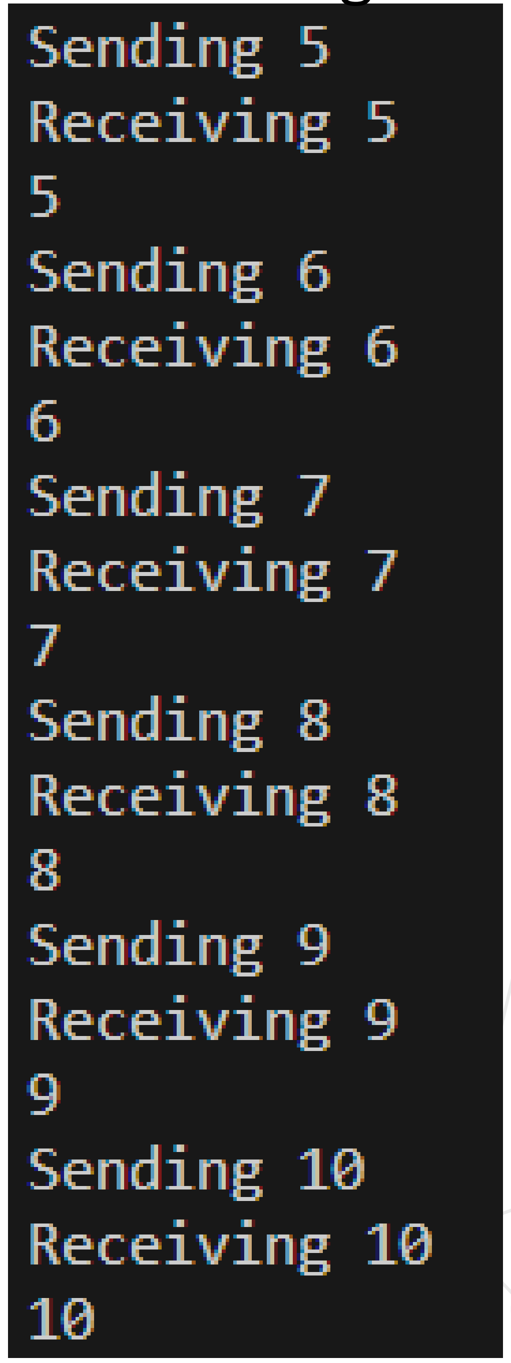
```
Sending 1
Receiving 1
Sending 2
Receiving 2
Sending 3
Receiving 3
Sending 4
Receiving 4
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Change the filter to make this output



```
Sending 1
Sending 2
Sending 3
Sending 4
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Change the filter again to make this output



```
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Let's see the code.

THANK YOU

<Cre8tive Devs. Software/>

Bitty Byte

We will see you soon