WELCOME
&
THANK YOU


# Reactive & Event Driven Programming

**Level 1**

# MEET YOUR BITTY BYTE TEAM

**TANGY F.**

**CEO**

**EDDIE K.**

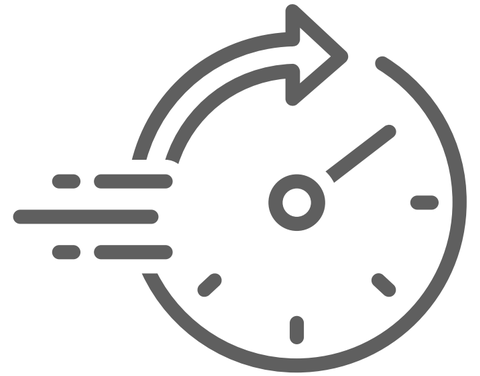
**DEVELOPER**

**WILLIAM D.**

**DEVELOPER**

Rapid Review

TRIVIA

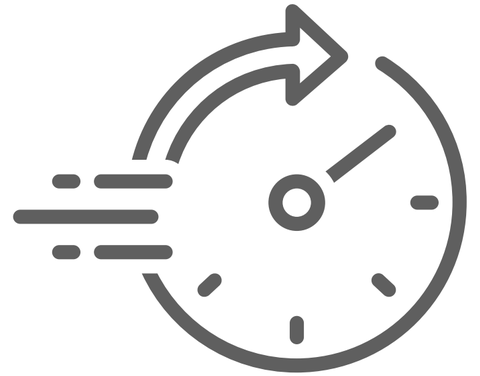


# Rapid Trivia

## What is a logic gate

Rapid Review

TRIVIA



## Rapid Trivia

**A logic gate is a non-physical switch that performs Boolean logic operation on one or more binary inputs, producing a binary output based on well-defined rules.**

# CODING EXERCISE

Write your first **one line** “Hello World” code using java reactor “Mono”

### Publisher

```
[Flux | Mono]<T> reactiveObject = [Flux | Mono].factoryOperator(value1, value2, value3)
                        .operator()
                        .operator()
                        .operator();
```

Chain / Reactive Chain

### Subscriber

```
reactiveObject.operator()
              .operator()
              .operator()
              .subscribe();
```

Chain / Reactive Chain

### *.factoryOperator* for Mono

1. **Mono.just:** Creates a **Mono** that emits the provided element and completes.
2. **Mono.fromSupplier:** Creates a **Mono** by invoking a supplier.
3. **Mono.fromCallable:** Creates a **Mono** from a **Callable**.
4. **Mono.fromFuture:** Creates a **Mono** from a **CompletableFuture**.
5. **Mono.fromRunnable:** Creates a **Mono** that completes when the provided runnable is executed.

**Gradle;**

```
implementation 'io.projectreactor:reactor-core:3.4.10'
```

**Maven;**

```
<dependencies>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.4.10</version>
  </dependency>
</dependencies>
```

# CODING EXERCISE

Right below “Hello World”, add a reactive Flux that list numbers from 1 to 50

### Publisher

```
[Flux | Mono]<T> reactiveObject = [Flux | Mono].factoryOperator(value1, value2, value3)
                        .operator()
                        .operator()
                        .operator();
```

Chain / Reactive Chain

### Subscriber

```
reactiveObject.operator()
              .operator()
              .operator()
              .subscribe();
```

Chain / Reactive Chain

## *.factoryOperator* for Flux

1. **Flux.just:** Creates a Flux that emits the provided elements and completes.
2. **Flux.fromArray:** Creates a Flux from an array of elements.
3. **Flux.fromIterable:** Creates a Flux from an Iterable.
4. **Flux.fromStream:** Creates a Flux from a Stream.
5. **Flux.from(FluxSource):** Converts a source FluxSource into a Flux.
6. **Flux.range:** Creates a Flux that emits a range of integer values.

**Gradle;**

```
implementation 'io.projectreactor:reactor-core:3.4.10'
```

**Maven;**

```
<dependencies>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.4.10</version>
  </dependency>
</dependencies>
```

Let's see the code.

# REINFORSMENT REVIEW

Type of events based on **origin of source**;

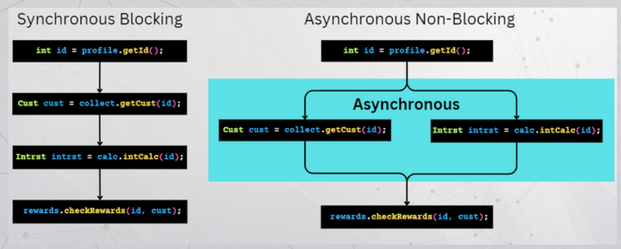
- User
- System

Type of events based on **timing of handling**;

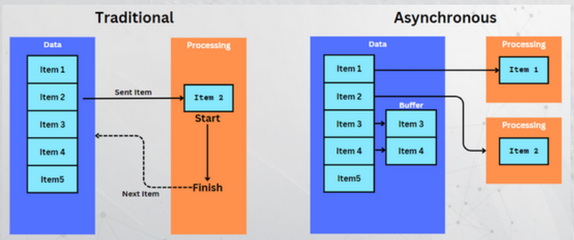
- Synchronous
- Asynchronous

Do not confuse synchronous and asynchronous events with;

Synchronous & Asynchronous Execution




Synchronous & Asynchronous Data Streams



# TODAY'S AGENDA

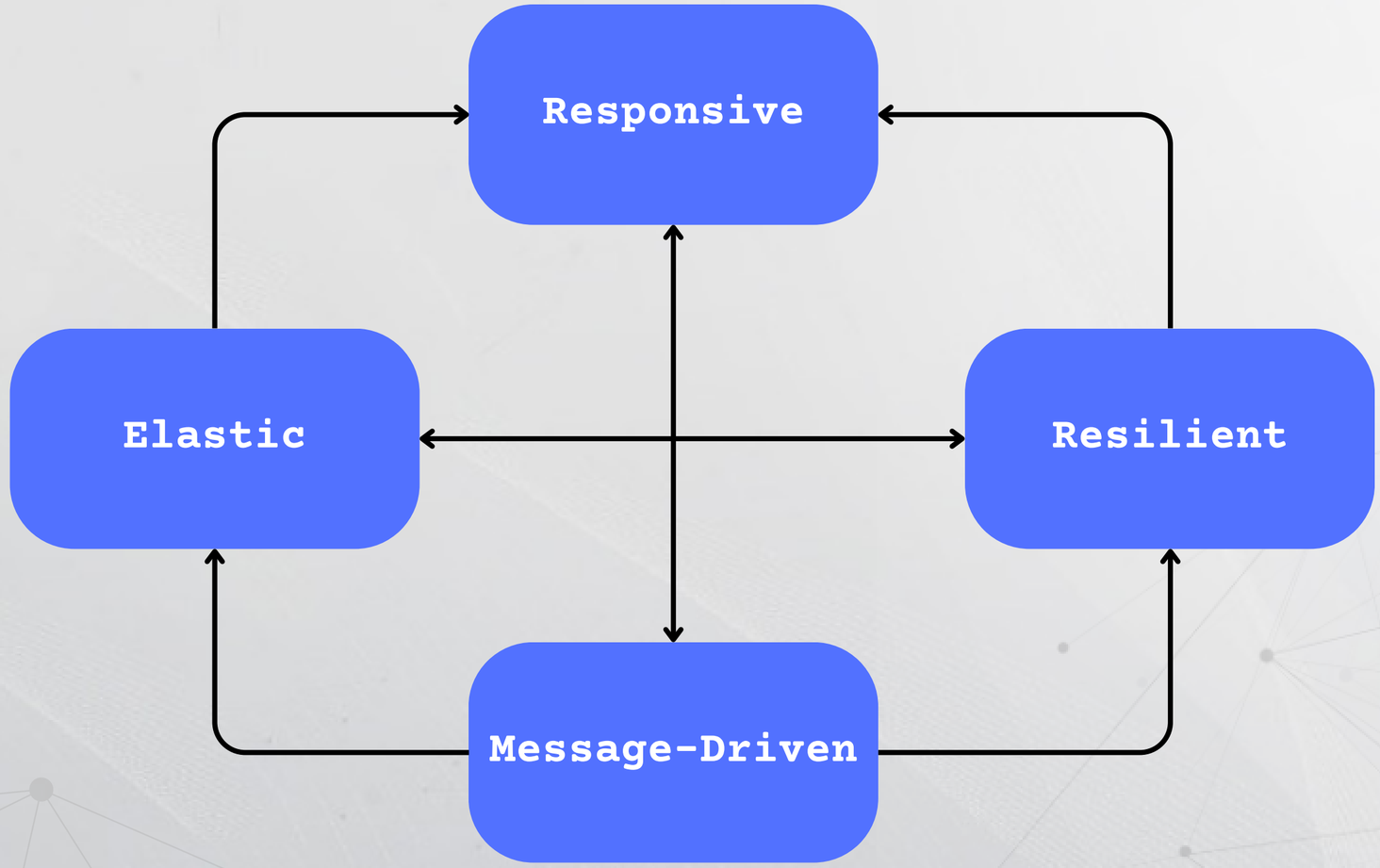
DAY 2

1. Characteristics of Reactive Systems
2. Operators, map, filter,flatMap & doOnNext
3. Flux & Mono Asynchronous Execution

# DAY 2
# LESSON 1

## Characteristics of Reactive Systems

# CHARACTERISTICS OF REACTIVE SYSTEMS

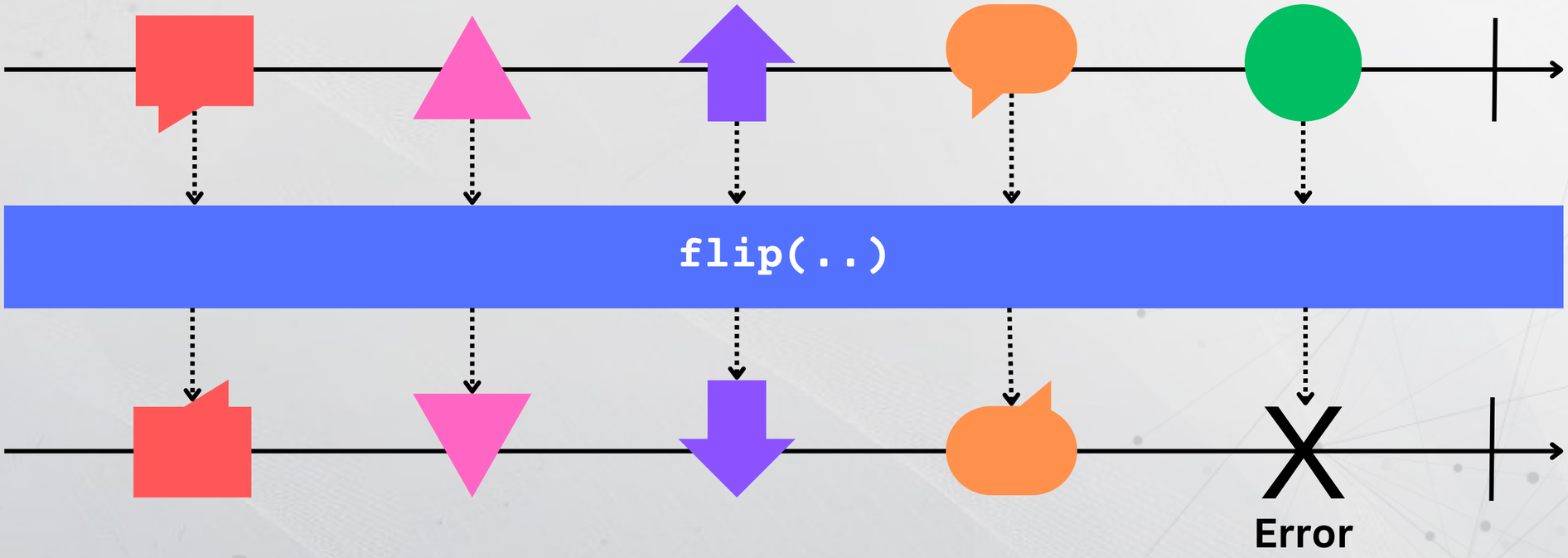


# DAY 2
# LESSON 2

## Operators;

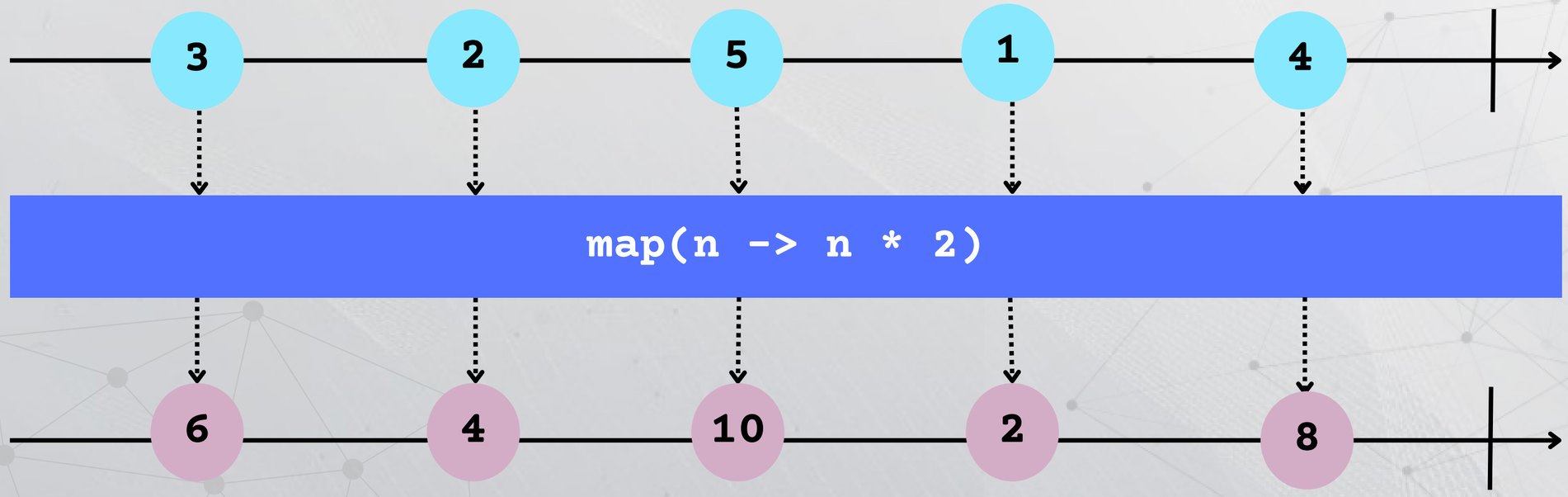
- map()
- filter()
- flatMap()
- doOnNext()

# MARBLE DIAGRAM



# MAP OPERATOR

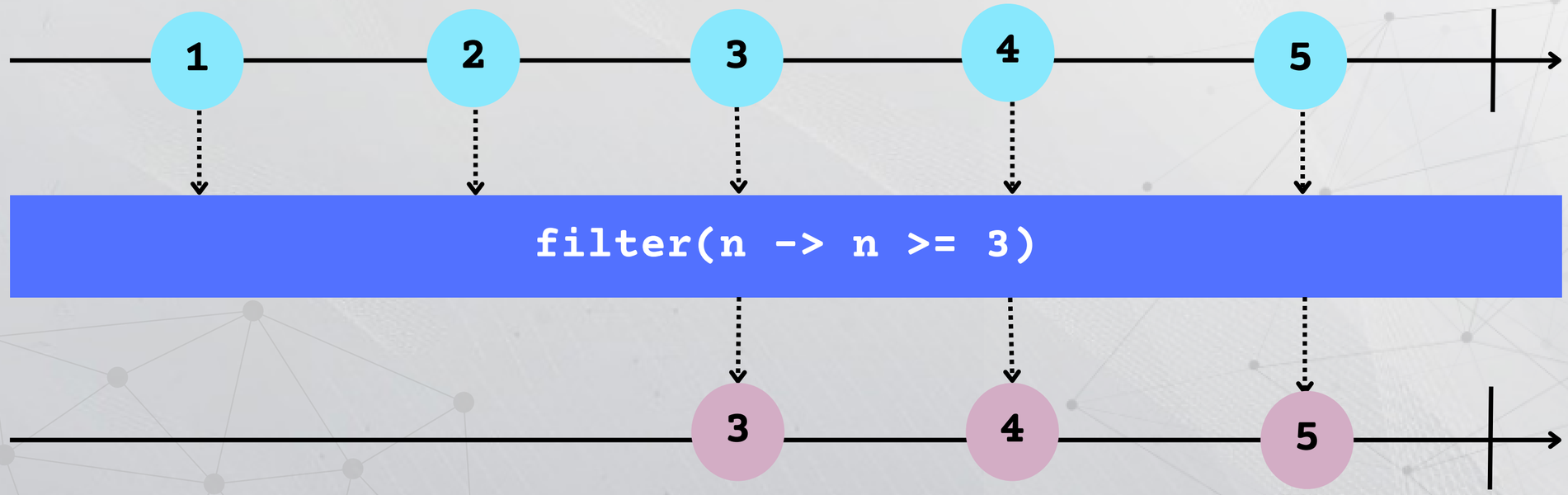
```
Flux<String> flux = Flux.range(start:1, count:10)
        .map(n -> "Number: " + n * 2);

flux.subscribe(System.out::println);
```




Let's see the code.

# FILTER OPERATOR

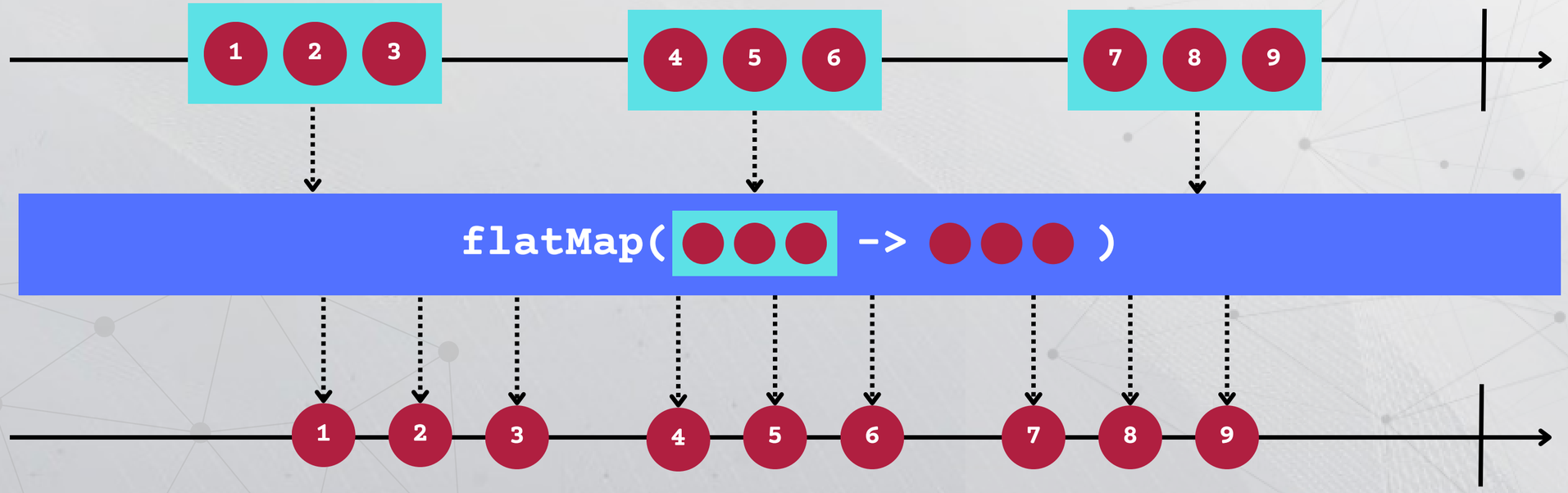
```
Flux<String> flux = Flux.just(...data:"Maria", "", "John", "", "")
        .filter(n -> !n.isEmpty());

flux.subscribe(item -> System.out.println("Received: " + item));
```




Let's see the code.

# FLATMAP OPERATOR

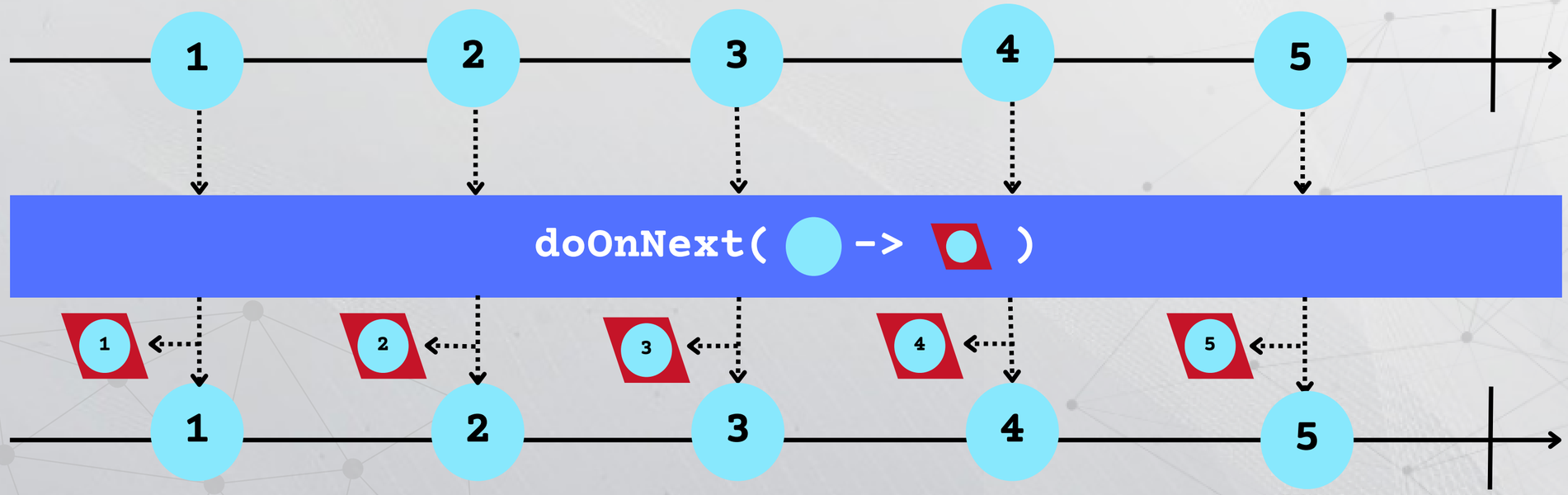
```java
List<List<String>> listOfLists = Arrays.asList(
    Arrays.asList("Element 1", "Element 2", "Element 3"),
    Arrays.asList("Element 4", "Element 5", "Element 6"),
    Arrays.asList("Element 7", "Element 8", "Element 9")
);

Flux<String> flattenedFlux = Flux.fromIterable(listOfLists)
    .flatMap(list -> Flux.fromIterable(list));

flattenedFlux.subscribe(System.out::println);
```




Let's see the code.

# DOONNEXT OPERATOR

```
Flux<Integer> flux1 = Flux.range(start:1, count:10)
        .doOnNext(item -> System.out.println("Loop 1: " + item))
        .subscribeOn(Schedulers.parallel());
```

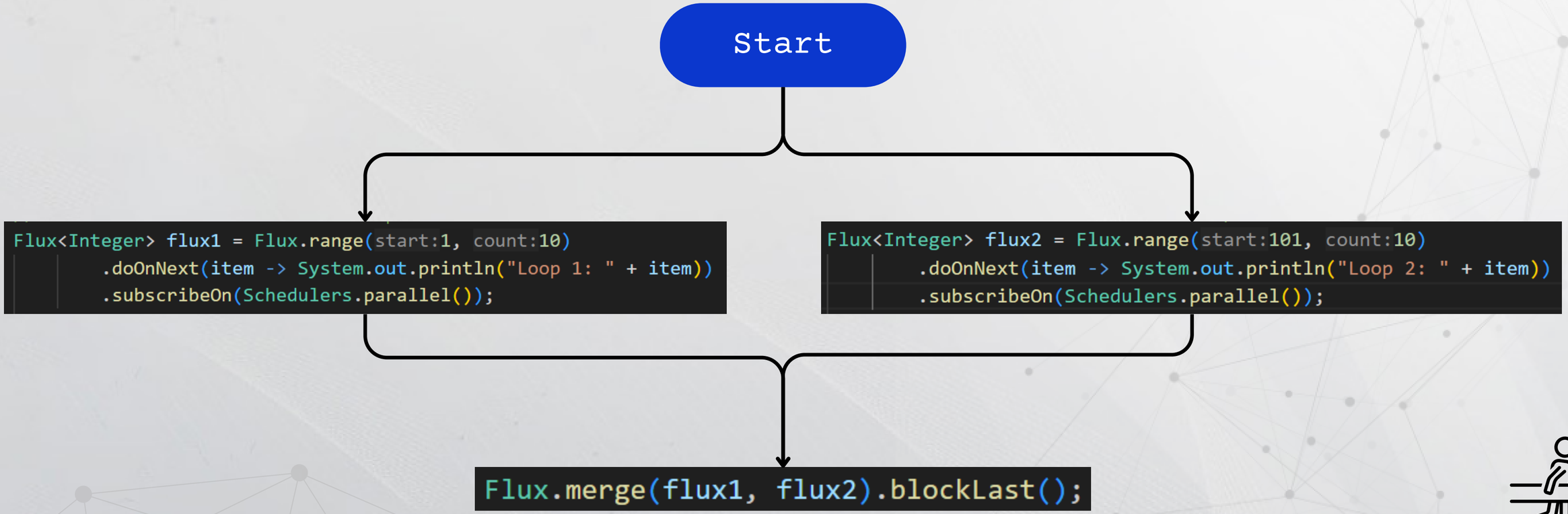


# DAY 2
# LESSON 3

## Flux & Mono Asynchronous

# ASYNCHRONOUS WITH FLUX

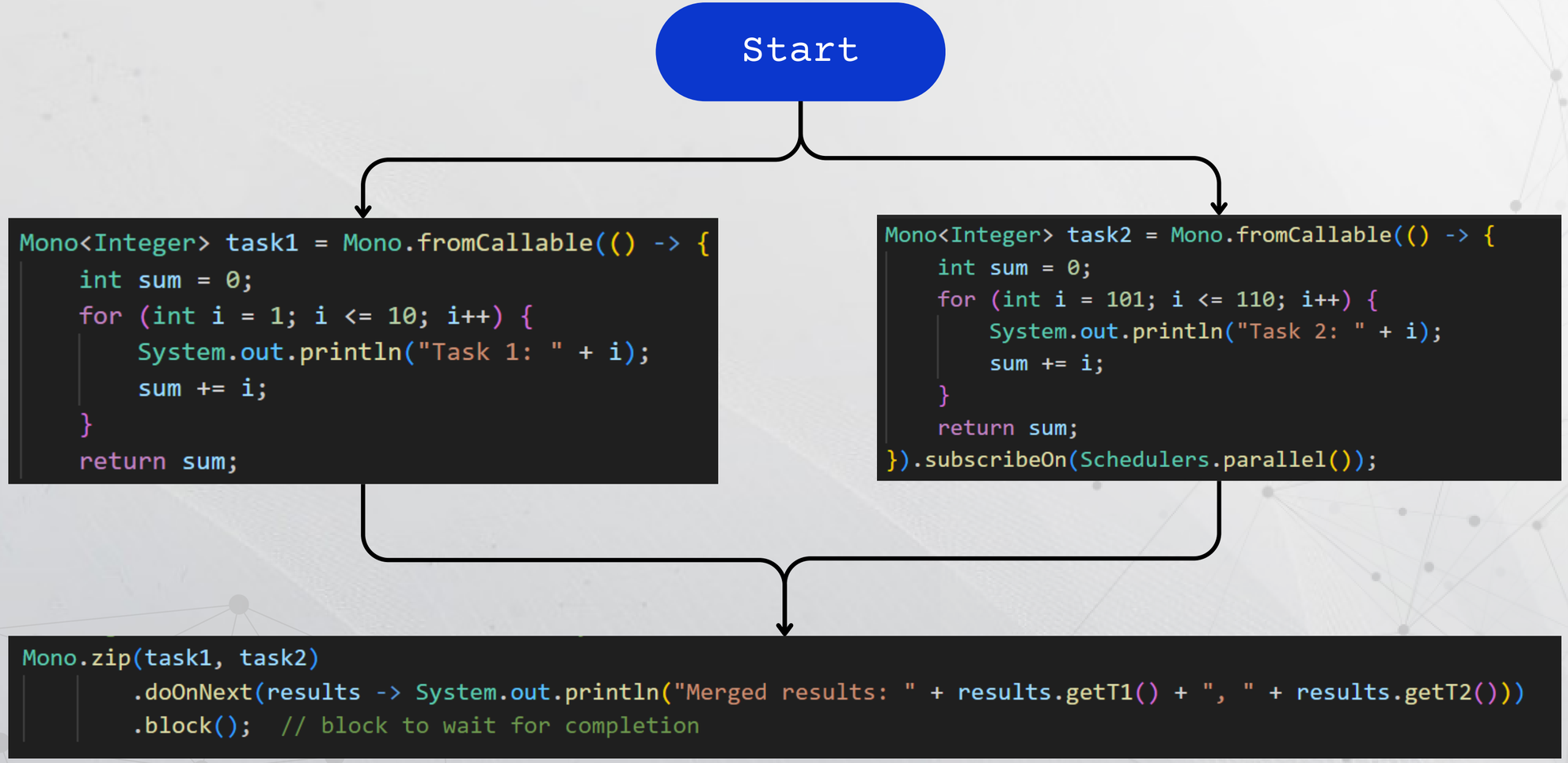
## Using Flux.merge() operator for concurrency




Let's see the code.

# ASYNCHRONOUS WITH MONO

## Using Mono.fromCallable() and Mono.zip() operator for concurrency




Let's see the code.

# LESSON QUESTION

What is the difference between a factory operator and regular operator?

# LESSON ANSWER

**Factory operators** create a new data stream or publisher, while regular **operators** work on an existing data stream, transforming or processing the data emitted by it.

# LESSON QUESTION

Describe the difference between declarative and imperative programming

# LESSON ANSWER

- Declarative: What to do.
  - Specify exact steps to achieve
- Imperative: How is done.
  - describe the desired outcome without specifying the exact steps to achieve

# 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.

# THANK YOU


Bitty Byte

**We will see you soon**