WELCOME
&
THANK YOU

Reactive & Event Driven Programming

Level 1

MEET A FEW OF OUR TEAM MEMBERS

TANGY F.
CEO

LINA G
PROJECT MANAGER

NARI S.
TECH ASSISTANT

EDDIE K.
DEVELOPER

WILLIAM D.
DEVELOPER

- *Founded in South Florida in 2018*
- *Evolved from Software Development Expanded into a Technical Training firm Providing tailored technical training solutions for businesses*

Creative Devs. Software
MOBILE APPS•WEBSITES•AR•VIDEO GAMES•SECURITY

THE CRASH COURSE

Bitty Byte is our
bite size lighting training

The Duration is 1 hour a day
for 5 days

One Topic~
In this case it's Docker

WE LISTENED TO YOU & YOUR COLLEAGUES

1. We interviewed practitioners and your leads
2. We reviewed anonymous data
3. Came up with a tailored curriculum based on real-time data

Creative Devs. Software

THE SPACE

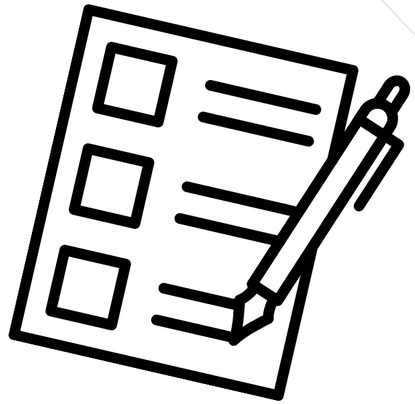
A safe space where our goal is to help you

We are here to help you grow. If there is something you don't understand this is the place to ask.

This is a 'No Judgment Zone'

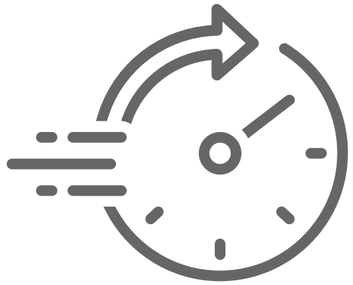
<Creative Devs. Software/>

TIME SET ASIDE



Pre Assessment

PRE-ASSESSMENT



Event Driven * Reactive

On a blank document
Type your name
&
answers the questions
Add your doc.
to the shared drive

QUESTIONS 1 & 2 PRE- ASSESSMENT

Full Name: _____

1. What is the name given to reactive programming data stream diagrams?

_____

_____

_____

2.In the code shown below, in which statement does the producer will start sending the collection items to its consumer? Provide line number or statement.

```
2   Iterable<Integer> collection = List.of(1, 2, 3, 4, 5);
3
4   Flux<Integer> flux = Flux.fromIterable(collection);
5
6   flux.subscribe(
7           item -> System.out.println("Received: " + item),
8           error -> System.err.println("Error: " + error),
9           () -> System.out.println("Processing completed")
10  );
```

QUESTIONS 3 & 4 PRE- ASSESSMENT

3. Based on the origin of source, what are the two main types of events in event-driven programming?

Based on the timing of handling, what are the two main types of events in event-driven programming

QUESTIONS 5 & 6 PRE- ASSESSMENT

5. What is backpressure within the context of reactive programming.

6. What is the difference between declarative vs. imperative programming.

MEET YOUR CRASH COURSE TEAM

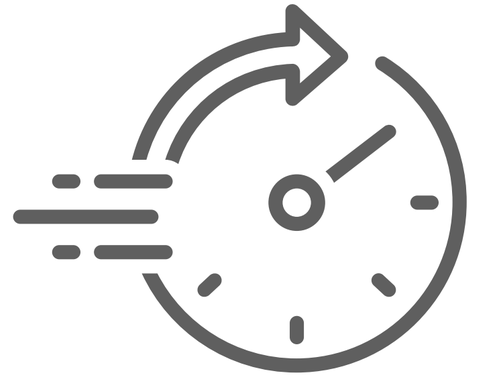
TANGY F.
CEO

EDDIE K.
DEVELOPER

WILLIAM D.
DEVELOPER

Rapid Review

TRIVIA

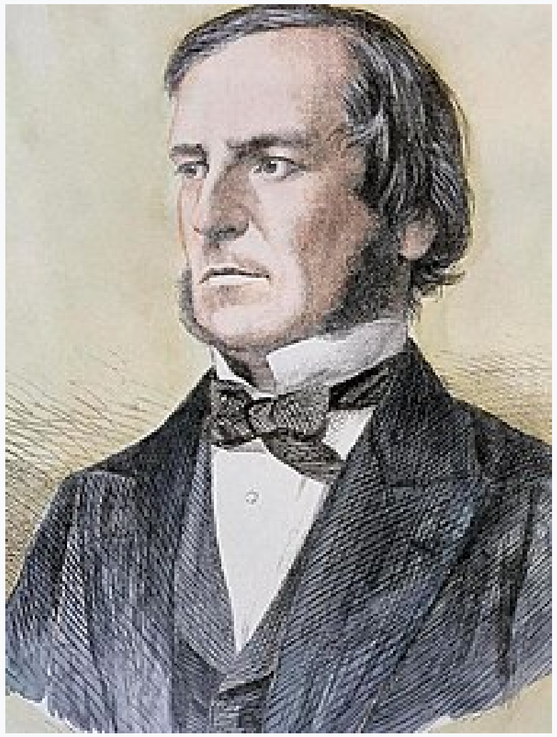


Rapid Trivia

What was George Boole contribution to the Information Age

Rapid Review

TRIVIA



George Boole 1815 – 1864

Author of “The Laws of Thought” which contains the foundation of the Boolean algebra of logic

TODAY'S AGENDA

TRAINING DAY ?

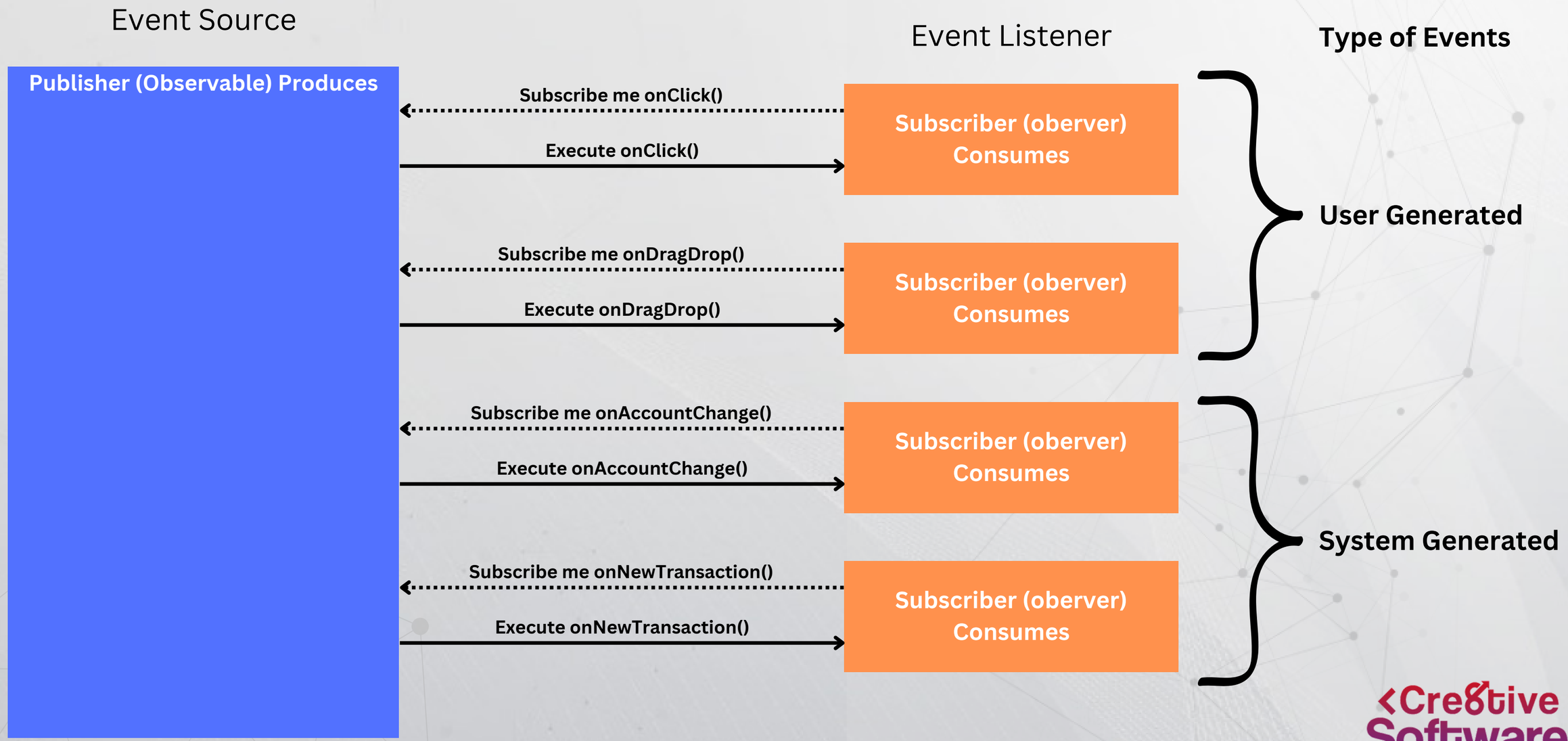
1. Reactive Basic Concepts
2. Anatomy of Reactor Declarative Statements
3. Flux & Mono

DAY 1
LESSON 1

Reactive Basic Concepts;

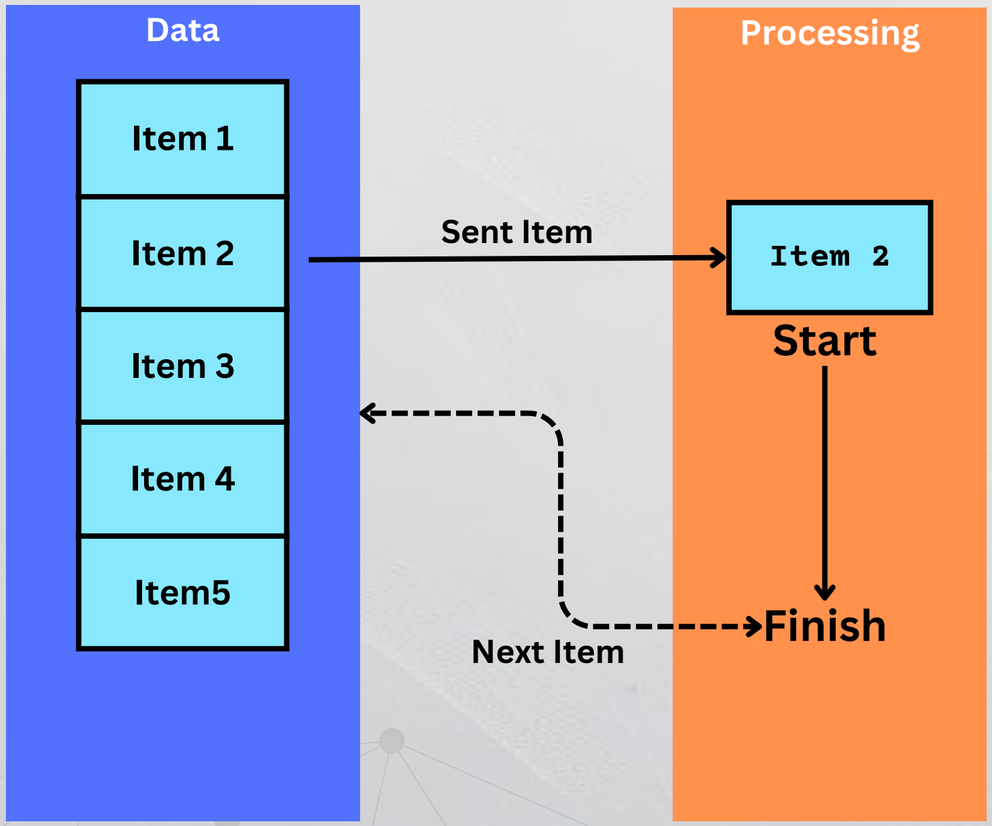
- Event Driven
- Data Streams
- Declarative Coding
- Asynchronous Non-Blocking

EVENT DRIVEN AND TYPE OF EVENTS

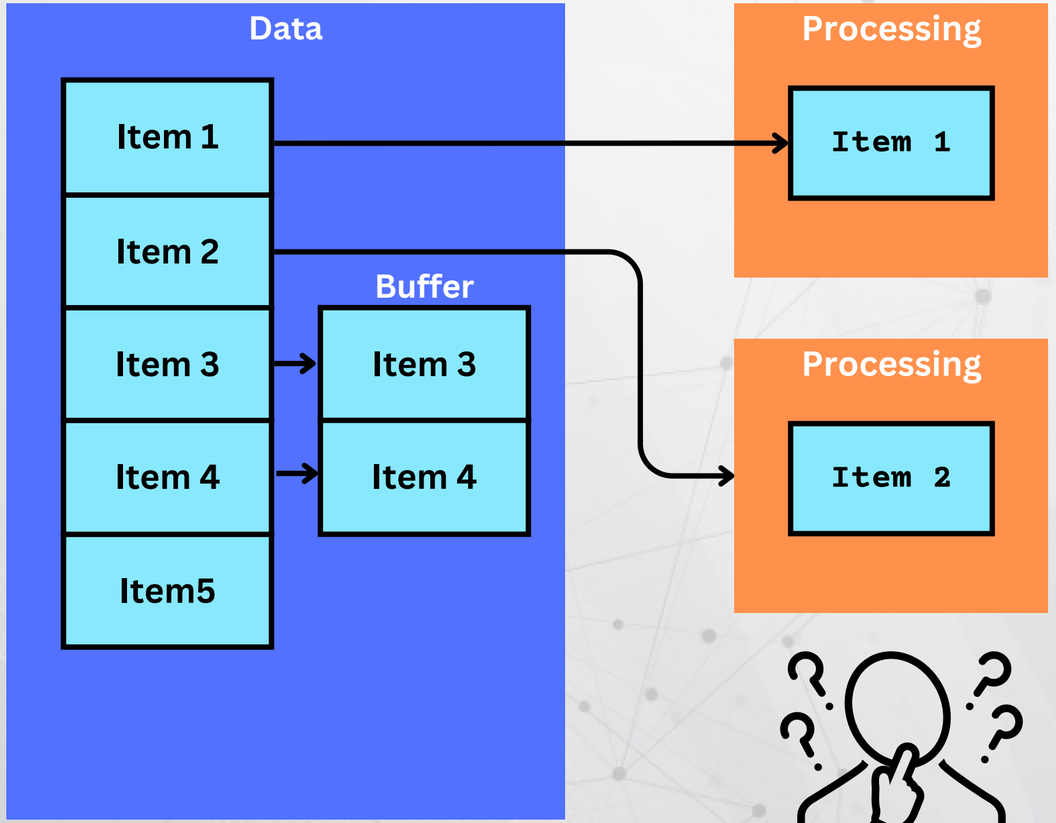


ASYNCHRONOUS DATA STREAMS

Traditional



Asynchronous



DECLARATIVE VS. IMPERATIVE

Imperative

```
List<Client> filteredCli = filterByAge(clients);
```

```
private static List<Client> filterByAge(List<Client> clients) {

    List<Client> filteredClients = new ArrayList<>();
    for (Client client : clients) {
        if (client.getAge() >= 30) {
            filteredClients.add(client);
        }
    }
    return filteredClients;

}
```

Declarative

```
List<Client> filteredClients = clients.stream()
    .filter(client -> client.getAge() >= 30)
    .collect(Collectors.toList());
```

ASYNCHRONOUS NON-BLOCKING

Synchronous Blocking

```
int id = profile.getId();
```

↓

```
Cust cust = collect.getCust(id);
```

↓

```
Intrst intrst = calc.intCalc(id);
```

↓

```
rewards.checkRewards(id, cust);
```

Asynchronous Non-Blocking

```
int id = profile.getId();
```

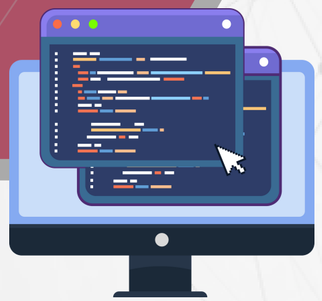
Asynchronous

```
Cust cust = collect.getCust(id);
```

```
Intrst intrst = calc.intCalc(id);
```

```
rewards.checkRewards(id, cust);
```

WHAT IS REACTIVE PROGRAMMING ?



Is a a declarative programming paradigm focused on handling and reacting to asynchronous data streams and events with non-blocking executions.

DAY 1
LESSON 2

Anatomy of Reactor Declarative Statements

REACTOR DECLARATIVE STATEMENTS

Publisher

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

Method Chain / Reactive Chain

Subscriber

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

Method Chain / Reactive Chain

REACTOR DECLARATIVE STATEMENTS

.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.

.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.

REACTOR DECLARATIVE STATEMENTS

Publisher *.operators*

1. **map:** Transforms each emitted item using a provided function.
2. **flatMap:** Maps each element into a publisher and flattens the result.
3. **filter:** Filters elements based on a predicate.
4. **onErrorResume:** Resumes with a fallback publisher when an error occurs.
5. **doOnNext:** Performs an action for each element.
6. **merge:** Merges multiple publishers into one.

Subscriber *.operators*

1. **onSubscribe:** Invoked by the publisher to pass the Subscription.
2. **onNext:** Invoked for each element in the stream.
3. **onError:** Invoked when an error occurs.
4. **onComplete:** Invoked when the stream is successfully completed.

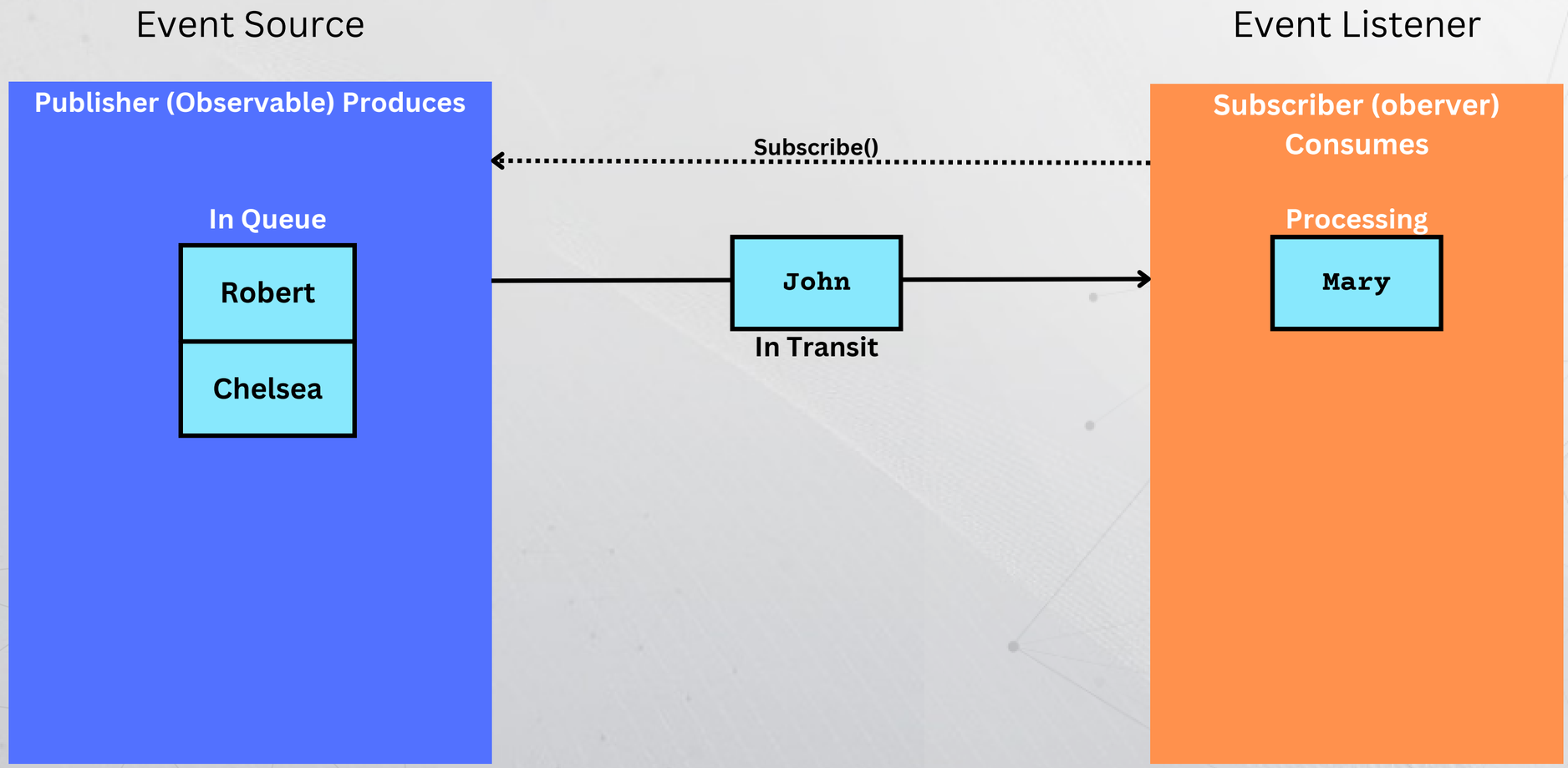
DAY 1
LESSON 3

Flux & Mono

FLUX & MONO

```
Flux<String> flux =
    Flux.just(...data:"Mary", "John", "Robert", "Chelsea");
```

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



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.

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 bellow “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.

THANK YOU

Bitty Byte

We will see you soon