**Reactive Programming**

Used for Unlocking and Asynchronous type of implementation

Mono : 0 to 1 elements we use mono as a publisher

Flux : 0 to n elements , to handle data streams we use flux as publisher

**map() :**

The map() operator in reactive programming, is used to **transform the elements of a reactive stream** by applying a function to each element. It operates on each element of a Flux or Mono, transforming it into a new value while preserving the original sequence. It does not alter the original sequence

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3, 4);

flux.map(i -> i \* 2)

.subscribe(System.out::println);

**flatMap() :**

In reactive programming, flatMap() is an operator used to transform elements of a stream, similar to map(). However, the key difference is that flatMap() **is used for asynchronous or multi-step transformations**, where the transformation function returns a Publisher (e.g., Mono or Flux) rather than a simple value.

Ex:

Flux.just("a", "b", "c")

.flatMap(s -> Flux.just(s.toUpperCase(), s.toLowerCase()))

.subscribe(System.out::println);

o/p :

A

a

B

b

C

c

**collectList**() : Collects the output of flatMap() into a list.

o/p:

[A, a, B, b, C, c]

**skip**() :

In reactive programming, skip() is an operator used to **skip a specified number of elements** from the beginning of the stream. It is useful when you want to ignore the first few elements and start processing from a later point in the stream.

Ex:

Flux.just(1, 2, 3, 4, 5)

.skip(2)

.subscribe(System.out::println);

o/p:

3

4

5

**delayElements**() :

In reactive programming, delayElements() is an operator that delays the emission of items in a Flux by a specified duration. It is useful when you want to simulate time-based delays or control the rate of data emission.

Ex:

Flux.just(1, 2, 3, 4, 5)

.delayElements(Duration.ofSeconds(1)) // Delay each element by 1 second

.subscribe(System.out::println);

o/p:

1 (after 1 second)

2 (after 1 second)

3 (after 1 second)

4 (after 1 second)

5 (after 1 second)

**merge**():

In reactive programming, merge() is an operator used to **combine multiple** Publisher **streams into a single stream**. It merges the emitted items from multiple sources into one continuous stream, interleaving them as they arrive.

Ex:

Flux<Long> flux1 = Flux.interval(Duration.ofMillis(500)).take(3); // Emits 0, 1, 2 after 500ms intervals

Flux<Long> flux2 = Flux.interval(Duration.ofMillis(700)).take(3); // Emits 0, 1, 2 after 700ms intervals

Flux.merge(flux1, flux2)

.subscribe(System.out::println);

o/p:

0 (from flux1)

0 (from flux2)

1 (from flux1)

1 (from flux2)

2 (from flux1)

2 (from flux2)

**zip**():

In reactive programming, zip() is an operator used to **combine the emissions of multiple** Publisher **streams** (such as Flux or Mono) in a way that each element from one stream is paired with an element from another stream, creating a tuple (or combination) of values. This ensures that elements are emitted only when all sources have emitted an item, and the values are emitted in a “zipped” manner.

Ex:

Flux<String> flux1 = Flux.just("A", "B", "C");

Flux<Integer> flux2 = Flux.just(1, 2, 3);

Flux.zip(flux1, flux2, (s, i) -> s + i) // Combine each String with its corresponding Integer

.subscribe(System.out::println);

o/p:

A1

B2

C3

**block**():

In reactive programming, block() is a terminal operation that **blocks the execution of the current thread until a value is emitted** from a Mono or Flux. It is typically used when you need to obtain the result of a reactive stream synchronously (e.g., for testing or when integrating with code that is not reactive.

**Dealing with** Mono **and** Flux:

• block() works for Mono when you expect a single value or none.

• blockFirst() and blockLast() can be used for Flux when you want the first or last value in a stream, respectively.

Ex:

Mono<String> mono = Mono.just("Hello, World!");

String result = mono.block();

System.out.println(result);

block() waits for the Mono to emit a value, and returns the result synchronously.

The program will output "Hello, World!"

Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

Integer result = flux.blockFirst();

System.out.println(result);

blockFirst() waits for the first item to be emitted by the Flux and then returns it.

The program will output 1. Similarly with blockLast();

**buffer**():

In reactive programming, buffer() is an operator that **collects emitted items** from a Flux into **batches** (lists), and then emits these batches as new items. This is useful when you want to process or store items in groups, rather than one by one.

Ex:

Flux<Integer> flux = Flux.range(1, 10);

flux.buffer(3) // Buffer items in groups of 3

.subscribe(System.out::println);

o/p:

[1, 2, 3]

[4, 5, 6]

[7, 8, 9]

[10]

**collectMap**():

collectMap() is a terminal operator that collects the emitted items into a Map using a **key extractor function** and a **value extractor function**.

Ex:

Flux<String> flux = Flux.just("apple", "banana", "cherry");

flux.collectMap(s -> s.charAt(0), s -> s.toUpperCase()) // Key by the first letter, value by uppercase string

.subscribe(map -> System.out.println(map));

o/p:

{a=APPLE, b=BANANA, c=CHERRY}

Flux<String> flux = Flux.just("apple", "banana", "apricot", "blueberry");

flux.collectMultimap(s -> s.charAt(0), s -> s.toUpperCase()) // Key by first letter, value by uppercase

.subscribe(map -> System.out.println(map));

{a=[APPLE, APRICOT], b=[BANANA, BLUEBERRY]}

**doOnEach**() :

In reactive programming, doOnEach() is a **side-effect operator** used to execute a specified action on each element emitted by a Mono or Flux. Unlike other operators in the reactive streams, doOnEach() does not modify the stream; it simply allows you to perform additional actions like logging, debugging, or modifying external state as items pass through the stream

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3, 4, 5);

flux.doOnEach(signal -> {

if (signal.isOnNext()) {

System.out.println("Item: " + signal.get());

}

if (signal.isOnComplete()) {

System.out.println("Stream Completed");

}

if (signal.isOnError()) {

System.out.println("Error: " + signal.getThrowable());

}

})

.subscribe();

o/p:

Item: 1

Item: 2

Item: 3

Item: 4

Item: 5

Stream Completed

oOnEach() executes a side-effect action for every item emitted by the Flux. In this case, we print the emitted items, and also log when the stream completes.

signal.isOnNext() checks if the signal is an emitted value (onNext), and signal.get() retrieves the emitted item.

signal.isOnComplete() logs when the stream finishes emitting items, and signal.isOnError() would handle any error signals (though none are emitted here).

**Note**:

doOnNext()**,** doOnError()**,** doOnTerminate(): These operators are more specialized versions of doOnEach():

doOnNext(): Executes only on onNext signals.

doOnError(): Executes only on onError signals.

doOnTerminate(): Executes when the stream terminates, either with onComplete or onError.

doOnSubscribe(): Executes when the stream is subscribed to. It allows you to perform actions (e.g., logging, tracking) before the stream starts emitting values.

doOnCancel(): Executes when the subscription is cancelled, i.e., when the subscriber unsubscribes from the stream, either explicitly or due to back pressure or other conditions.

**Exception Handling:**

Exception handling is an important aspect to ensure that errors in the stream are properly managed. Project Reactor provides several operators to handle exceptions that can occur during the processing of data in a Mono or Flux

1.onErrorResume():

**Purpose**: Provides a fallback in case of an error. You can return another stream (Mono or Flux) when an error occurs.

**Example**: When an error happens, this operator lets you recover and continue the stream with an alternative flow.

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3, 4)

.map(i -> {

if (i == 3) throw new RuntimeException("Error on 3");

return i;

})

.onErrorResume(e -> {

System.out.println("Error occurred: " + e.getMessage());

return Flux.just(10, 20, 30); // Provide a fallback sequence

});

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

o/p:

1

2

Error occurred: Error on 3

10

20

30

2. onErrorReturn():

**Purpose**: Returns a predefined value when an error occurs, instead of emitting the error.

**Example**: In case of an error, you can emit a default value.

Ex:

.map(i -> {

if (i == 3) throw new RuntimeException("Error on 3");

return i;

})

.onErrorReturn(999); // Return 999 when an error occurs

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

o/p:

1

2

999

3.doOnError():

**Purpose**: Executes a side-effect when an error occurs without altering the flow of the stream.

**Example**: Use doOnError() for logging or monitoring the error, but let the error propagate unless specifically handled.

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3)

.map(i -> {

if (i == 3) throw new RuntimeException("Error on 3");

return i;

})

.doOnError(e -> System.out.println("Logged error: " + e.getMessage()));

flux.subscribe(System.out::println, e -> System.out.println("Handled Error: " + e));

o/p:

1

2

Logged error: Error on 3

Handled Error: java.lang.RuntimeException: Error on 3

4.retry():

**Purpose**: Retries the operation a specified number of times or until the stream succeeds.

**Example**: Retry the operation when an error occurs, e.g., retry a failed API request.

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3)

.map(i -> {

if (i == 2) throw new RuntimeException("Error on 2");

return i;

})

.retry(2); // Retry 2 times if there's an error

flux.subscribe(System.out::println, e -> System.out.println("Final error: " + e));

o/p:

1

2

Error on 2

1

2

Error on 2

1

2

Final error: java.lang.RuntimeException: Error on 2

5.retryWhen():

**Purpose**: Provides a more customizable way to handle retries by allowing you to define the retry strategy based on the error.

**Example**: Use retryWhen() to retry a failed operation based on a custom condition, such as a delay between retries

Ex:

Flux<Integer> flux = Flux.just(1, 2, 3)

.map(i -> {

if (i == 2) throw new RuntimeException("Error on 2");

return i;

})

.retryWhen(errors -> errors

.doOnNext(e -> System.out.println("Retrying due to error: " + e.getMessage()))

.take(2)); // Retry up to 2 times

flux.subscribe(System.out::println, e -> System.out.println("Final error: " + e));

o/p:

1

Retrying due to error: Error on 2

1

Retrying due to error: Error on 2

1

Final error: java.lang.RuntimeException: Error on 2

**Backpressure**:

**Backpressure** is a critical concept in reactive programming where the consumer of the data (subscriber) can’t process the emitted items as quickly as they are produced by the publisher (or producer). To prevent memory overload, dropped items, or system crashes, the consumer signals the publisher to slow down or stop sending more items until the consumer is ready.

1. onBackpressureBuffer():

• **Purpose**: Buffers the items when the subscriber can’t keep up, holding them until the subscriber is ready. This can increase memory usage.

• **Example**: The buffer size can be limited to avoid excessive memory consumption.

Ex:

Flux<Integer> flux = Flux.range(1, 100)

.onBackpressureBuffer(10, // Buffer size limit

item -> System.out.println("Dropped item: " + item)); // Drop action when buffer overflows

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

**Use case**: Use this strategy when you want to buffer items and wait for the subscriber to process them.

2.onBackpressureDrop():

• **Purpose**: Drops the items when the subscriber can’t keep up. This avoids memory overhead, but data might be lost.

• **Example**: Drops items beyond a certain threshold.

Ex:

Flux<Integer> flux = Flux.range(1, 100)

.onBackpressureDrop(); // Drop excess items when subscriber can't keep up

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

**Use case**: Use this when you can afford to lose data (e.g., real-time event streams where missed events don’t significantly affect the system).

3.onBackpressureLatest():

• **Purpose**: Keeps only the latest item and drops any items emitted before the subscriber is ready to process it.

• **Example**: When a subscriber is busy, it will only receive the most recent item.

Ex:

Flux<Integer> flux = Flux.range(1, 100)

.onBackpressureLatest(); // Keep the most recent item

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

**Use case**: Use this when the most recent data is more important than earlier items (e.g., in real-time data streams where you only need the latest state).

4.publishOn() **/** subscribeOn():

• **Purpose**: You can control on which thread the publisher and subscriber run, which may help in managing backpressure when switching between threads.

• **Example**: Use these operators to shift work to different threads and balance the workload

Ex:

Flux<Integer> flux = Flux.range(1, 100)

.publishOn(Schedulers.parallel()) // Publisher runs on parallel scheduler

.onBackpressureBuffer(); // Buffer items in case of backpressure

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

.

**Use case**: Use when you want to perform work on different threads to handle backpressure more efficiently.

**Example: Backpressure Handling**

In this example, we simulate backpressure by emitting a large number of items at a faster rate than the subscriber can process:

import reactor.core.publisher.Flux;

import reactor.core.scheduler.Schedulers;

public class BackpressureExample {

public static void main(String[] args) {

Flux<Integer> flux = Flux.range(1, 1000)

.doOnNext(i -> {

try {

Thread.sleep(1); // Simulate slow subscriber by adding delay

} catch (InterruptedException e) {

e.printStackTrace();

}

})

.onBackpressureBuffer(100, // Set buffer size

item -> System.out.println("Dropped: " + item)) // Action when buffer overflow occurs

.publishOn(Schedulers.parallel()); // Run publisher on a different thread

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

}

}

**Explanation:**

• The Flux.range(1, 1000) emits a large number of items, but the subscriber is intentionally slow due to the Thread.sleep(1) delay.

• onBackpressureBuffer(100, ...) ensures that up to 100 items can be buffered before dropping items.

• The publishOn(Schedulers.parallel()) operator shifts the emission of items to a parallel scheduler, which may help with backpressure management.

Yt video :

<https://www.youtube.com/watch?v=y3ySZkSgWik&list=PLdDvyK133QzQB-DIqlcersNmmtLyltbdJ&index=27>