Reactor

Advanced Java I. Functional, Asynchronous, Reactive Java

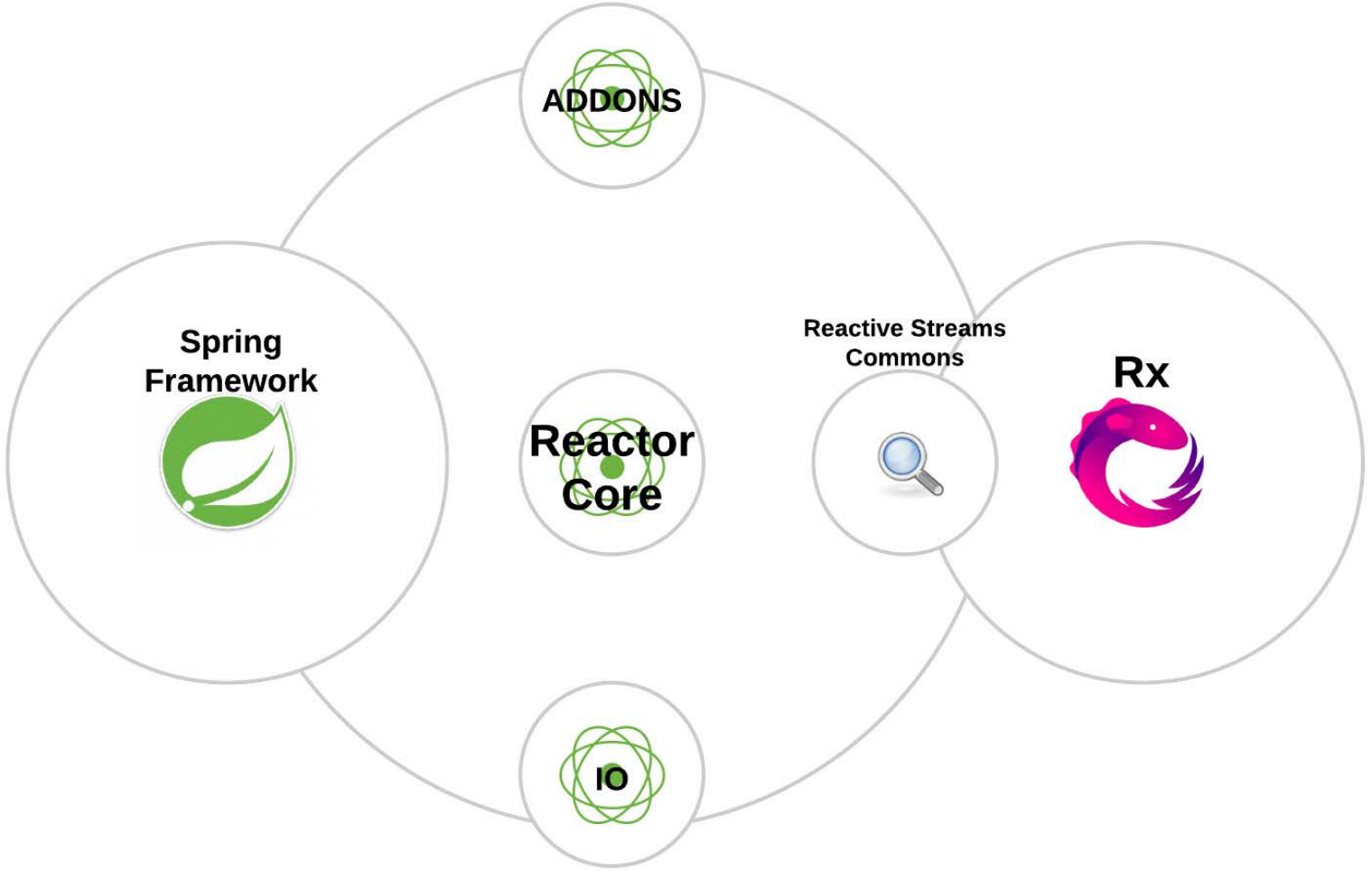
Module 6

think.
create.
accelerate.

Luxoft | training
A DXC Technology Company

Introducing Reactor

- **Project Reactor** is a Spring implementation of Reactive Streams.
- It has two main publishers:
  - `Flux<T>`
  - `Mono<T>`
- It uses Schedulers similar to RxJava.



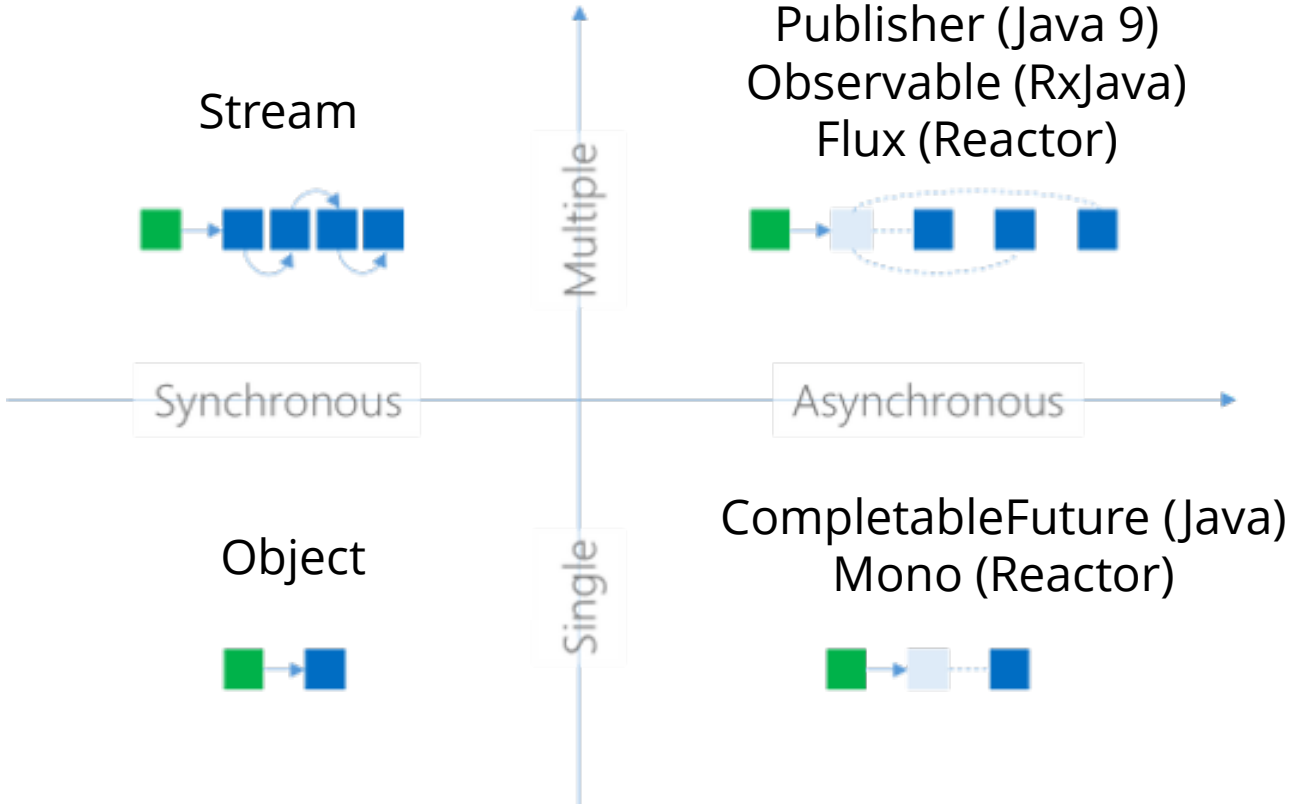
Reactor maven dependencies

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
  <version>3.3.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-test</artifactId>
  <version>3.3.1.RELEASE</version>
  <scope>test</scope>
</dependency>
```

Flux

- `Flux<T>` - the main entry point for Reactor reactive streams
- Similar to RxJava `Observable`
- `Mono<T>` - like a `Flux` for 0 or 1 element
- `Mono` and `Flux` implement `org.reactivestreams.Publisher`

Function may return…



Flux example

```
Flux.range(1, 10)
        .subscribe(System.out::println);
```

Results:

1
2
3
4
5
6
7
8
9
10

Flux interval example

```
Flux.interval(Duration.ofSeconds(1))
        .take(3)
        .doOnNext(System.out::println)
        .blockLast();
```

We see the differences from RxJava:

- Interval is using Java 8 Duration instead of TimeUnit
- We don’t have blockingSubscribe() and need to use blockLast()

Mono example

```
Mono.just("example")
    .subscribe(System.out::println);
```

Result:
example

Mono methods:

- `justOrEmpty(T)` - takes a nullable value and converts it into a `Mono`. If null, the result is the same as `Mono.empty()`.
- `justOrEmpty(Optional)` - takes an `Optional` and converts into a `Mono` directly.

Flux creation patterns

```
Flux.just("one", "two")
    .subscribe(System.out::println);
List<String> iterable = Arrays.asList("three“, "four");
Flux.fromIterable(iterable)
    .subscribe(System.out::println);
Flux.range(1, 10)
    .subscribe(System.out::println);
```

Results:

one
two
three
four
1
2
3
4
5

Mono creation patterns

```
Mono.empty()
    .subscribe(System.out::println);
Mono.just("example")
    .subscribe(System.out::println);
```

Results:
example

Flux generate method

```
Flux.generate(
        AtomicInteger::new,
        (item, square) -> {
            int i = item.getAndIncrement();
            square.next(i * i);
            if (i == 10) square.complete();
            return item;
        })
    .subscribe(System.out::println);
```

Results:

0
1
4
9
16
25
36
49
64
81
100

Flux create method

```
Flux.create(sink -> {
    for (int i = 0; i < 3; i++) {
        sink.next("i=" + i);
    }
    sink.complete();
}).subscribe(System.out::println);
```

Results:

i=0
i=1
i=2

Reactor operators to work with Flux

Operator	Explanation
empty()	Creates a Flux that gets completed without emitting any element.
concat()	Facilitates the concatenation of all sources.
create()	Facilitates the creation of a Flux that has the capability of multiple emissions.
delay()	Signals to the subscriber to delay Flux until the given period has elapsed.
interval()	Creates a Flux that emits data of type Long for the given interval.
just()	Creates a Flux that emits the specific number of items.
merge()	Merges the emitted publisher sequences.

Reactor operators to work with Mono

Operator	Explanation
create()	Creates a deferred emitter that can be used with callback-based APIs.
and()	Combines the results of two Monos.
empty()	Creates a Mono that gets completed without emitting any element.
otherwise()	Gets unsubscribed to a returned publisher when any error that matches the given type occurs.

Handling backpressure

Method	Explanation
onBackpressureBuffer()	Buffers all items until they can be handled downstream
onBackpressureBuffer(maxSize)	Buffers items up to the given count
onBackpressureBuffer(maxSize, BufferOverflowStrategy)	Buffers items up to the given count and allows you to specify the strategy to use when and if the buffer is full
onBackpressureLatest()	If the downstream does not keep up with upstream, only the latest element will be given downstream

Handling backpressure

Method	Explanation
onBackpressureError()	Ends the Flux with an error (calling the downstream Subscriber's onError) with an IllegalStateException from Exceptions. failWithOverflow() if more items were produced upstream than requested downstream
onBackpressureDrop()	Drops any items produced above what was requested. This would be useful, for example, in UI code to drop user input that can't be handled immediately
onBackpressureDrop(Consumer)	Drops any items produced above what was requested and calls the given Consumer for each dropped item

BufferOverflowStrategy enum

Strategy	Explanation
DROP_OLDEST	Drops the oldest items in the buffer
DROP_LATEST	Drops the oldest items in the buffer
ERROR	Terminate the stream with an error

BackpressureDemo

Reactor Schedulers (frequently used)

Sheduler	Explanation
Schedulers.**immediate**()	Running on the current Thread (no-op Scheduler).
Schedulers.**single**()	A single, reusable thread. this method reuses the same thread for all callers, until the Scheduler is disposed.
Schedulers.**parallel**()	Creates as many workers as you have CPU cores.
Schedulers .**boundedElastic**()	Creates new worker pools as needed and reuses idle ones. Worker pools that stay idle for too long (the default is 60s) are also disposed. it has a cap on the number of backing threads it can create (default is number of CPU cores x 10). This is a better choice for I/O blocking work. handy way to give a blocking process its own thread so that it does not tie up other resources, but doesn't pressure the system too much with new threads.

Reactor Schedulers (others)

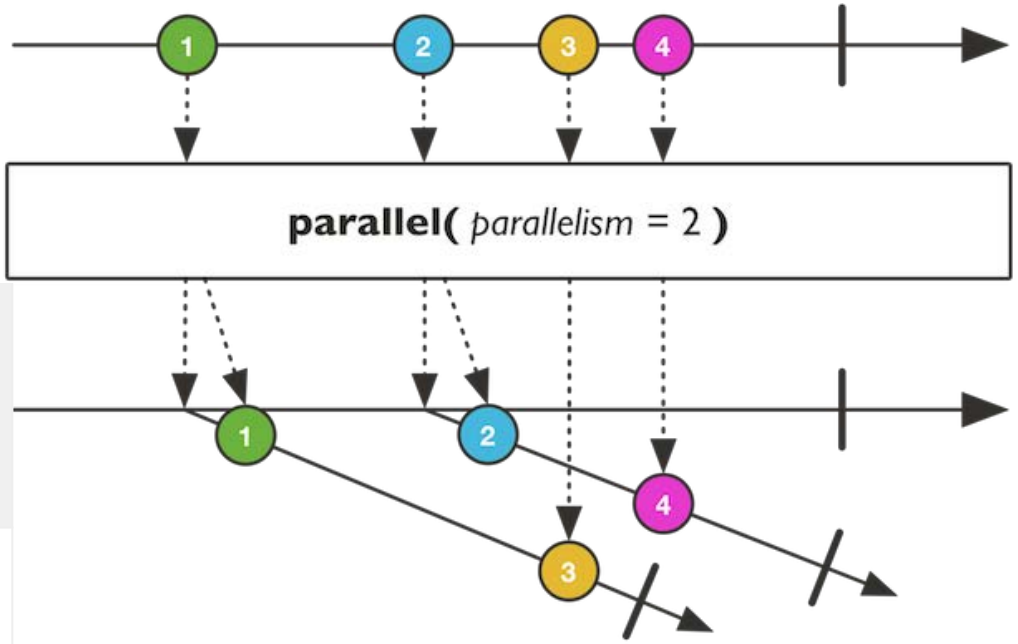
Sheduler	Explanation
Schedulers .**fromExecutorService**(ExecutorService es)	Scheduler from pre-existing ExecutorService
Schedulers.**newParallel**(yourScheduleName, [parallelism], [daemon])	Creates a new parallel scheduler named yourScheduleName. Parallelism allows to set number of threads. Daemon is false by default.
Schedulers .**newBoundedElastic**(threadCap, queuedTaskCap [name], [ttlSeconds], [daemon])	**threadCap** - maximum number of underlying threads to create **queuedTaskCap** - maximum number of tasks to enqueue when no more threads can be created **ttlSeconds** - Time-to-live for an idle
Schedulers.**elastic**(), Schefulers.**newElastic**()	Unbounded elastic thread pool, not recommended.

SchedulerDemo

Parallel Flux

Use Parallel flux to perform parallel calculations for the Flux data.

parallel()	Prepare this Flux by dividing data on a number of 'rails' matching the number of CPU cores, in a round-robin fashion.
parallel(int parallelism)	Prepare this Flux by dividing data on a number of 'rails' matching the provided parallelism parameter, in a round-robin fashion.
runOn(Scheduler scheduler)	Specifies where each 'rail' will observe its incoming values with no work-stealing and default prefetch amount.
sequential()	Merges the values from each 'rail' in a round-robin or same-order fashion and exposes it as a regular Publisher sequence, running with a default prefetch value for the rails.




```
Flux.range(1, 10)
  .parallel(4)
  .runOn(Schedulers.parallel())
  .subscribe(i -> System.out.println(
        Thread.currentThread()
        .getName() +
              " -> " + i));
```

ParallelFluxDemo

Results:

```
parallel-1 -> 1
parallel-4 -> 4
parallel-2 -> 2
parallel-3 -> 3
parallel-2 -> 6
parallel-4 -> 8
parallel-1 -> 5
parallel-1 -> 9
parallel-2 -> 10
parallel-3 -> 7
```

Wrapping a synchronous, blocking call

```
Mono blockingWrapper = Mono.fromCallable(() -> {
    return /* make a remote synchronous call */
});
blockingWrapper = blockingWrapper
        .subscribeOn(Schedulers.boundedElastic());
```

Testing with reactor

- Main uses of reactor-test:
  - Testing that a sequence follows a given scenario with **`StepVerifier`**
  - Producing data in order to test the behavior of operators downstream with **`TestPublisher`**

`StepVerifier`

- Reactor `StepVerifier` can be used to verify the behavior of a Reactor Publisher (Flux or Mono).
- `StepVerifier` is an interface used for testing that can be created using one of several static methods on `StepVerifier` itself.

StepVerifier

```
Mono<String> message =
                Mono.just("message");

StepVerifier.create(message)
        .expectNext("message")
        .verifyComplete();
```

TestDemo testMonoValue()

StepVerifier

```
Mono<String> monoException =
      Mono.error(
            new RuntimeException("exception"));

StepVerifier.create(monoException)
        .expectErrorMessage("exception")
          .verify();
```

TestDemo testMonoException()

StepVerifier

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

StepVerifier.create(flux)
        .expectNext(1)
        .expectNext(2)
        .expectNext(3)
        .expectComplete()
        .verify(Duration.ofSeconds(10));
```

TestDemo testStepVerifier()

TestVerifier

```
TestPublisher<Object> publisher =
      TestPublisher.create();
Flux<Object> stringFlux = publisher.flux();
List list = new ArrayList();

stringFlux.subscribe(
      next -> list.add(next),
      ex -> ex.printStackTrace());
publisher.emit("one", "two");

assertEquals(2, list.size());
assertEquals("one", list.get(0));
assertEquals("two", list.get(1));
```

TestDemo testPublisher()

Reactor processors

Processors are **Publisher that are also a Subscriber**.

ProcessorsDemo

Processor	Explanation
Direct Processor	A **direct Processor** is a processor that can dispatch signals to zero or more Subscribers.
Unicast Processor	Deal with backpressure by using an **internal buffer**. The trade-off is that it can have **at most one Subscriber**. By default, buffer is **unbounded**: if you push any amount of data through it while its Subscriber has not yet requested data, it buffers all of the data. We can provide an optional queue to buffer the events. After the buffer is full, the processor starts to **reject elements**.
Emitter Processor	Emit to several subscribers while **honoring backpressure for each of its subscribers**.
Replay Processor	Caches elements and **replays them to late subscribers**. Can be created in various configurations: • Caching a **single element** (cacheLast). • Caching a **limited history** (create(int)) or an **unbounded history** (create()). • Caching a **time-based replay window** (createTimeout(Duration)). • Caching a **combination of history size and time window** (createSizeOrTimeout(int, Duration)).

RxJava and Reactor

- Similar from a functional perspective
- `Flow` is part of a reactive streams specification, bundled into JDK.
- APIs are similar, they both support basic operators like `map()`, `filter()`, `flatMap()`, and more advanced ones.
- RxJava needs Java 6+
- Reactor needs Java 8+
- Reactor integrates very well with Spring
- There are many independent projects that choose RxJava as their API (MongoDB, RxNetty - Reactor extension adaptor for Netty)

Summary: what is Flux?

Flux is a mix of Stream + CompletableFuture, with:

- a lot of powerful operators
- backpressure support
- control over publisher and subscriber behavior
- control over the notion of time (buffering windows of values, adding timeouts and fallbacks, etc)
- something tailored for async sequences fetched over the network (from a database or a remote Web API)

Thank You!

think.
create.
accelerate.