# Key Concurrency & Scheduler Operators in the Flux Class (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

# Learning Objectives in this Part of the Lesson

- Recognize key Flux operators
  - Concurrency & scheduler operators
    - These operators arrange to run other operators in designated threads & thread pools
      - e.g., subscribeOn(), publishOn(), & Schedulers.newParallel()

## Key Concurrency Operators in the Flux Class

# Key Concurrency Operators in the Flux Class

- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param

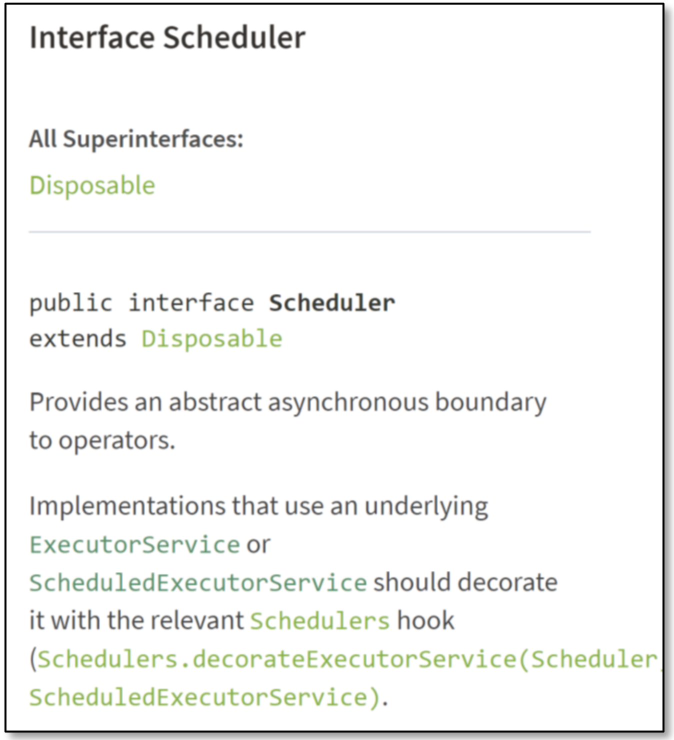
```
Flux<T> subscribeOn
  (Scheduler scheduler)
```

See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#subscribeOn](projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#subscribeOn)

# Key Concurrency Operators in the Flux Class

- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
    - The scheduler param indicates what thread to perform the operation on

```
Flux<T> subscribeOn
   (Scheduler scheduler)
```



Interface Scheduler

**All Superinterfaces:**

Disposable

```
public interface Scheduler
extends Disposable
```

Provides an abstract asynchronous boundary to operators.

Implementations that use an underlying `ExecutorService` or `ScheduledExecutorService` should decorate it with the relevant `Schedulers` hook (`Schedulers.decorateExecutorService(Scheduler, ScheduledExecutorService)`).

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html

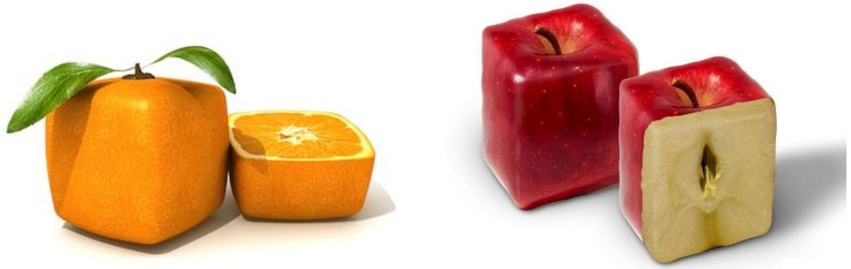
# Key Concurrency Operators in the Flux Class

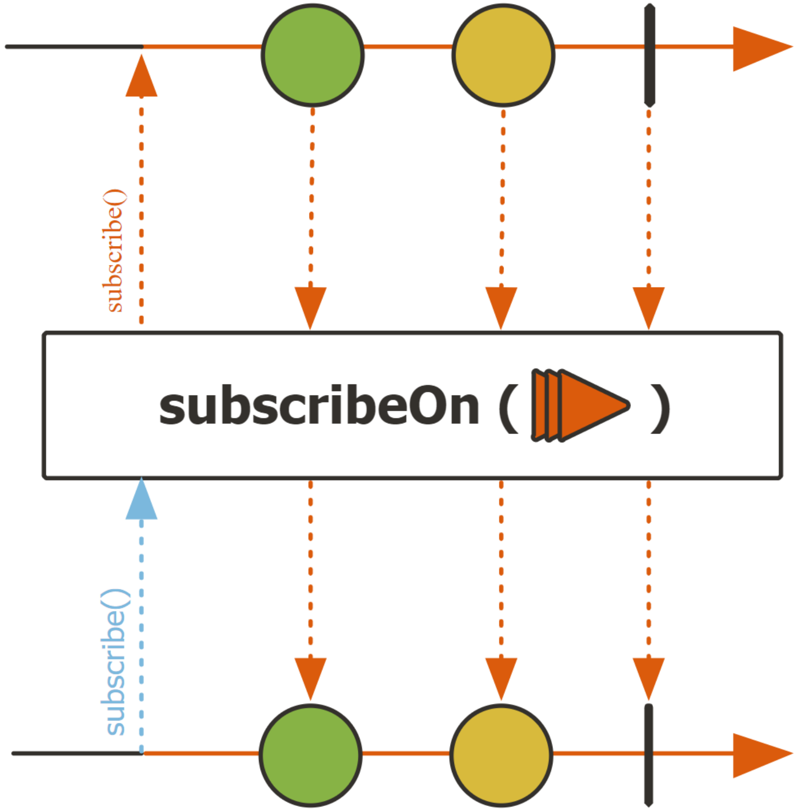
- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
    - The scheduler param indicates what thread to perform the operation on
    - Returns the Flux requesting async processing

```
Flux<T> subscribeOn
  (Scheduler scheduler)
```

# Key Concurrency Operators in the Flux Class

- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
  - The subscribeOn() semantics are a bit unusual





# Key Concurrency Operators in the Flux Class

- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
  - The subscribeOn() semantics are a bit unusual
    - Placing this operator in a chain impacts the execution context of onNext(), onError(), & onComplete() signals

```
Scheduler publisher = Schedulers
   .newParallel("publisher", 1));

Flux
   .range(1, sMAX_ITERATIONS)
   .subscribeOn(publisher)
   .map(__ -> BigInteger
     .valueOf(lowerBound + rand
       .nextInt(sMAX_ITERATIONS)))
   ...
   .doFinally(() -> publisher
              .displose())
   .subscribe(sink::next,
              err -> sink
                     .complete(),
              sink::complete);
```

See Reactive/flux/ex2/src/main/java/FluxEx.java

# Key Concurrency Operators in the Flux Class

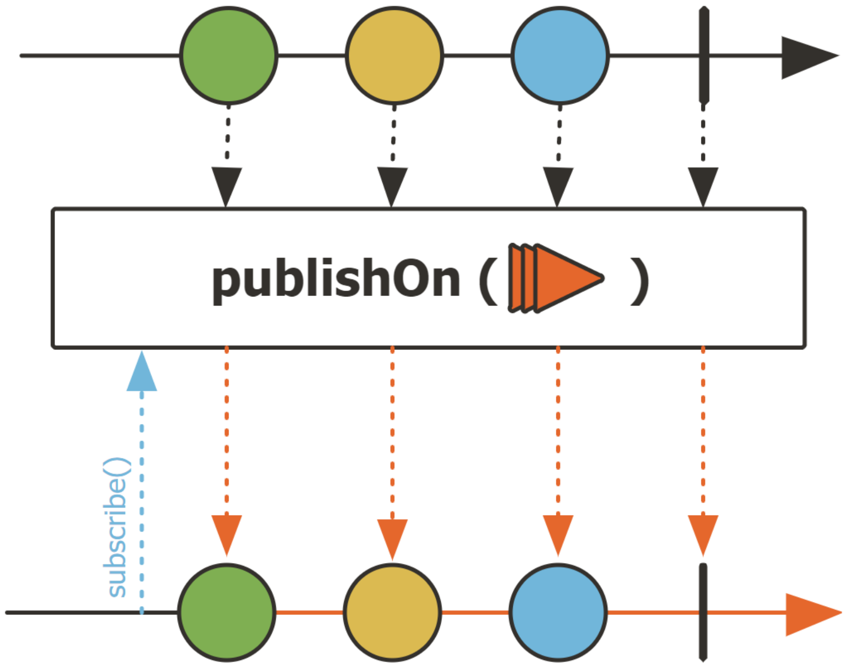
- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
  - The subscribeOn() semantics are a bit unusual
    - Placing this operator in a chain impacts the execution context of onNext(), onError(), & onComplete() signals

```
Scheduler publisher = Schedulers
   .newParallel("publisher", 1));

Flux
   .range(1, sMAX_ITERATIONS)
   .map(__ -> BigInteger
     .valueOf(lowerBound + rand
       .nextInt(sMAX_ITERATIONS)))
   ...
   .doFinally(() -> publisher
                .displose())
   .subscribeOn(publisher)
   .subscribe(sink::next,
              err -> sink
                     .complete(),
              sink::complete);
```

*subscribeOn() can appear later in the chain & have the same effect*

# Key Concurrency Operators in the Flux Class

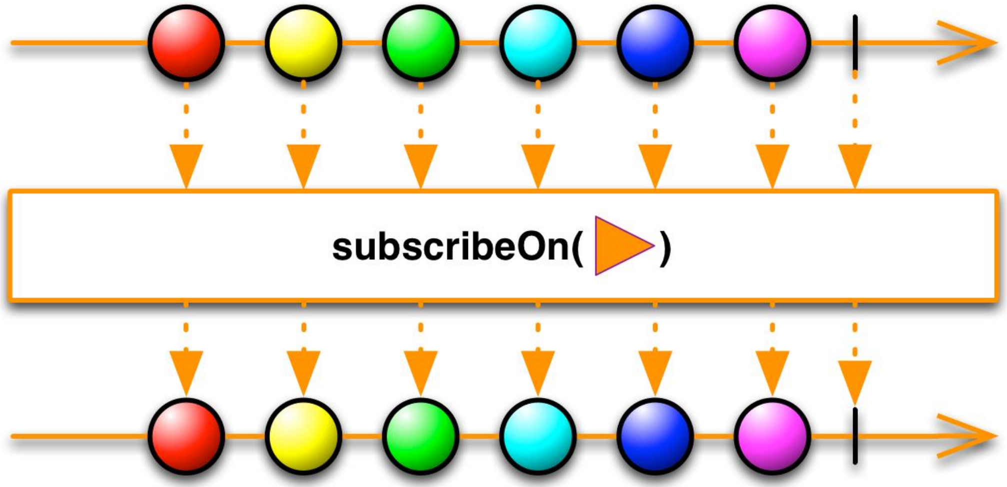
- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
  - The subscribeOn() semantics are a bit unusual
    - Placing this operator in a chain impacts the execution context of onNext(), onError(), & onComplete() signals
      - However, if a publishOn() operator appears later in the chain that can change the threading context where the rest of the operators in the chain below it execute (publishOn() can appear multiple times)



See [projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#publishOn](projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#publishOn)

# Key Concurrency Operators in the Flux Class

- The subscribeOn() operator
  - Run subscribe(), onSubscribe(), & request() on the specified Scheduler param
  - The subscribeOn() semantics are a bit unusual
  - RxJava's Observable. subscribeOn() works the same way



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#subscribeOn

# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param

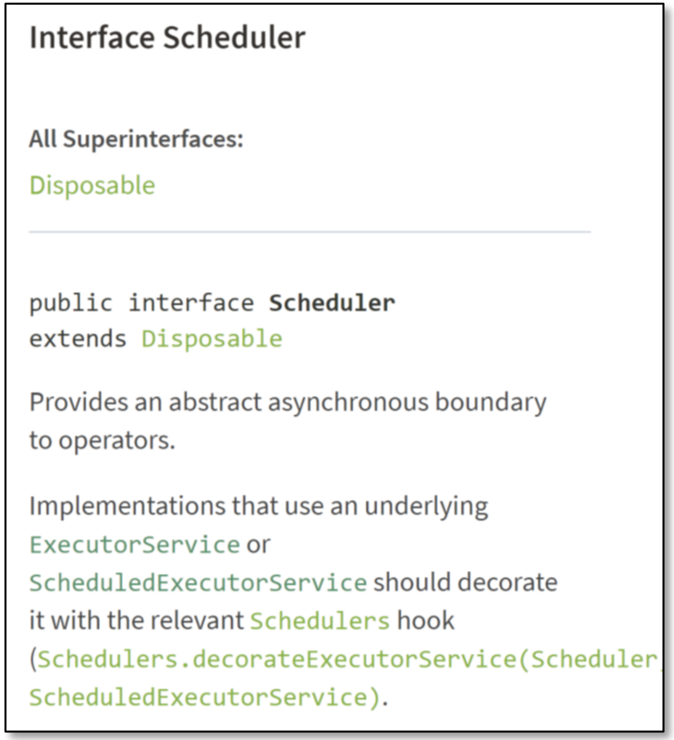
```
Flux<T> publishOn
   (Scheduler scheduler)
```

See projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#publishOn

# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
    - The scheduler param indicates what thread to perform the operation on

```
Flux<T> publishOn
   (Scheduler scheduler)
```



Interface Scheduler

**All Superinterfaces:**

Disposable

```
public interface Scheduler
extends Disposable
```

Provides an abstract asynchronous boundary to operators.

Implementations that use an underlying `ExecutorService` or `ScheduledExecutorService` should decorate it with the relevant `Schedulers` hook (`Schedulers.decorateExecutorService(Scheduler, ScheduledExecutorService)`).

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html

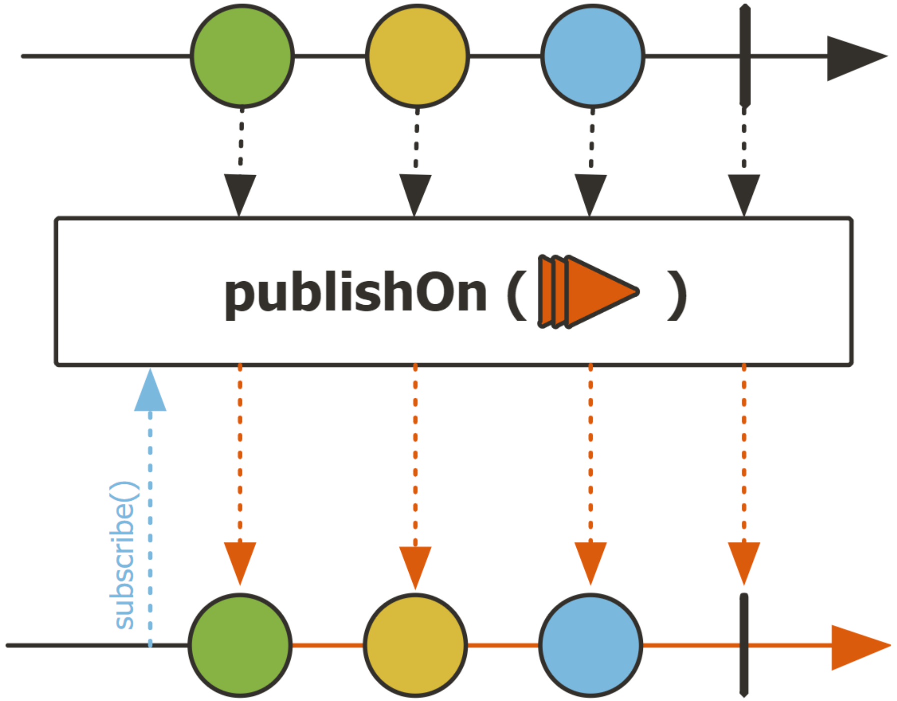
## Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
    - The scheduler param indicates what thread to perform the operation on
    - Returns the Flux requesting async processing

```
Flux<T> publishOn
  (Scheduler scheduler)
```

# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward



# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward
    - It influences the threading context where the rest of the operators in the chain below it execute

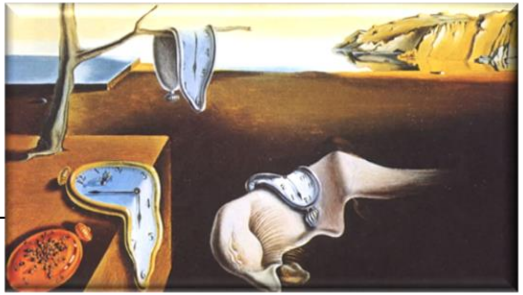
```
Scheduler subscriber = Schedulers
   .newParallel("subscriber",
                1));

return Flux
   .create(makeAsyncFluxSink(sb))
   .publishOn(subscriber)
   .map(bigInteger -> FluxEx
          .checkIfPrime(bigInteger,
                        sb))
   .doOnNext(bigInteger -> FluxEx
       .processResult(bigInteger,
                      sb))
   .doFinally(___ ->
              subscriber.dispose())
   ...
```

See [Reactive/flux/ex2/src/main/java/FluxEx.java](Reactive/flux/ex2/src/main/java/FluxEx.java)

# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward
    - It influences the threading context where the rest of the operators in the chain below it execute
      - Up to any new occurrence of publishOn() (if any)

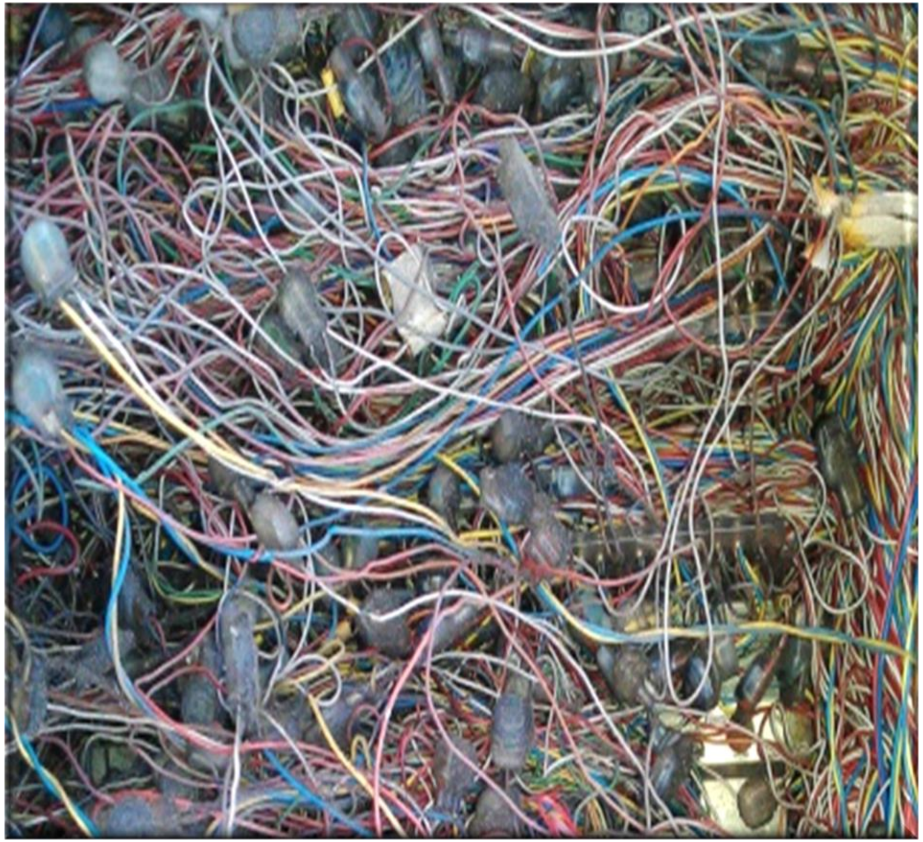
```
Scheduler subscriber = Schedulers
   .newParallel("subscriber",
                2));

return Flux
   .create(makeAsyncFluxSink(sb))
   .publishOn(subscriber)
   .map(bigInteger -> FluxEx
          .checkIfPrime(bigInteger,
                        sb))
   .publishOn(subscriber)
   .doOnNext(bigInteger -> FluxEx
        .processResult(bigInteger,

   ...
```



Beware of publishing on too many different threads!

# Key Concurrency Operators in the Flux Class

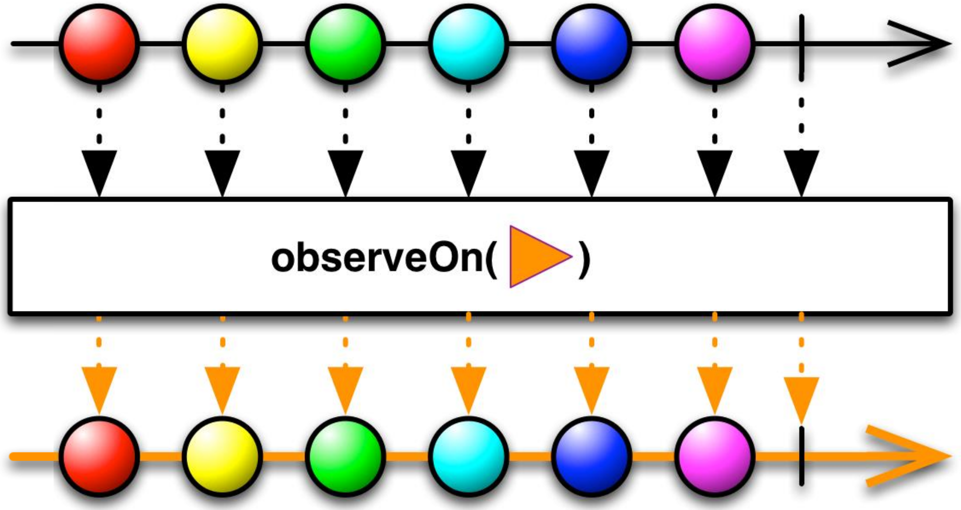
- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward
    - It influences the threading context where the rest of the operators in the chain below it execute
    - Interactions between publishOn() & subscribeOn() are convoluted..



See [www.woolha.com/tutorials/project-reactor-publishon-vs-subscribeon-difference](www.woolha.com/tutorials/project-reactor-publishon-vs-subscribeon-difference)

# Key Concurrency Operators in the Flux Class

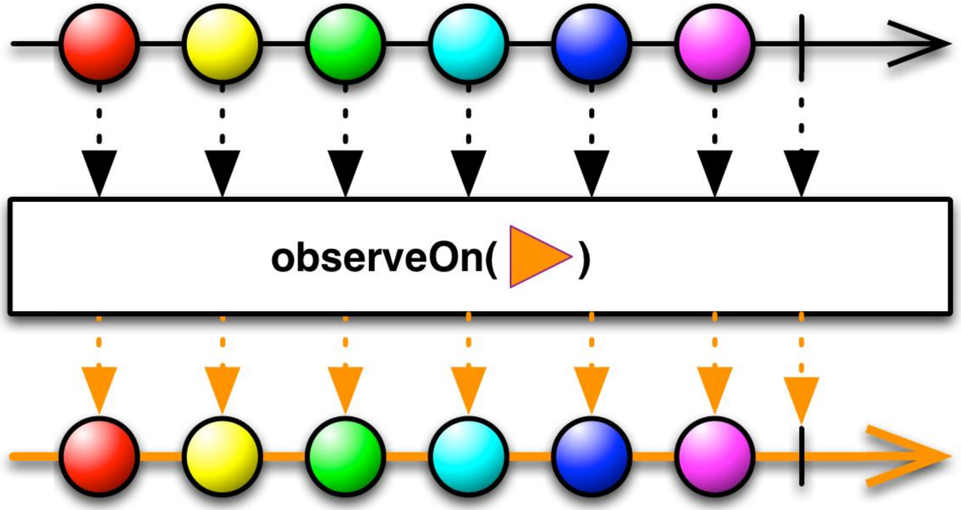
- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward
  - RxJava's Observable.observeOn() operator works the same



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#observeOn

# Key Concurrency Operators in the Flux Class

- The publishOn() operator
  - Run onNext(), onComplete(), & onError() on a supplied Scheduler param
  - The publishOn() semantics are fairly straightforward
  - RxJava's Observable.observeOn() operator works the same
    - Why RxJava & Project Reactor chose different names for this operator is a mystery..



See [reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#observeOn](reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#observeOn)

## Key Scheduler Operators Used By the Flux Class

# Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers

```
static Scheduler newParallel
   (String name,
    int parallelism)
```

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#newParallel

## Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
    - The params (1) give a name for the scheduler & (2) indicate the # of pooled worker threads

```
static Scheduler newParallel
   (String name,
    int parallelism)
```

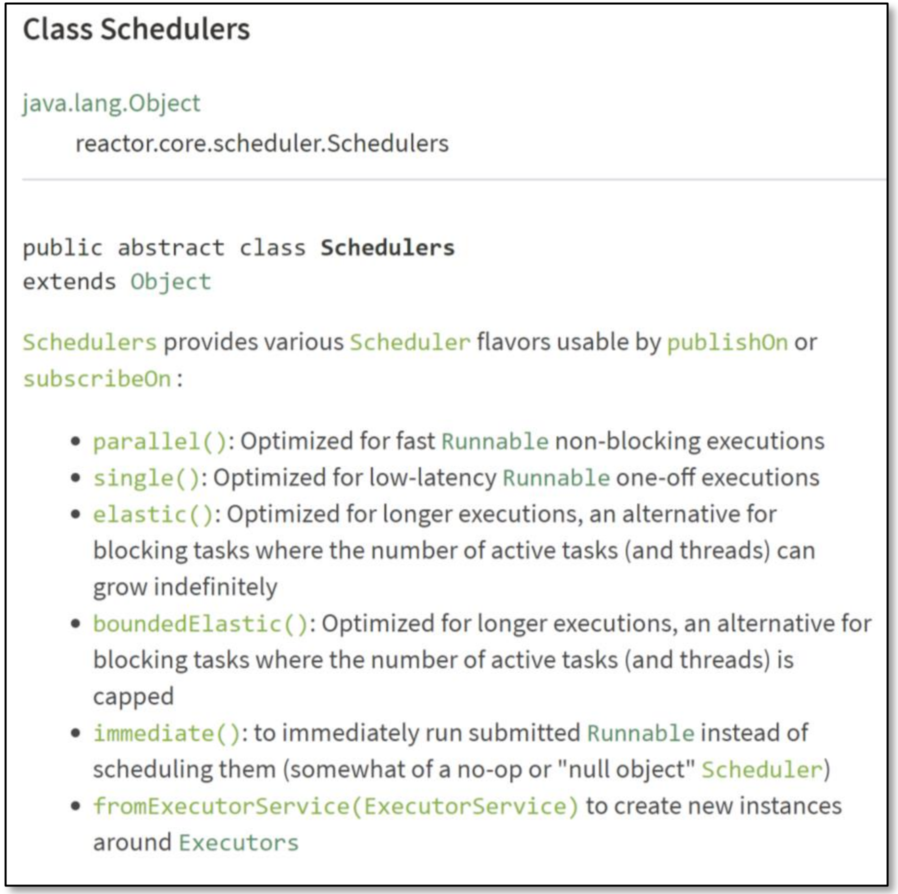
## Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
    - The params (1) give a name for the scheduler & (2) indicate the # of pooled worker threads
    - Returns a new Scheduler suitable for parallel computations

```
static Scheduler newParallel
   (String name,
    int parallelism)
```

# Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
    - The params (1) give a name for the scheduler & (2) indicate the # of pooled worker threads
    - Returns a new Scheduler suitable for parallel computations
      - However, it detects & rejects use of blocking Reactor APIs



Class Schedulers

java.lang.Object
    reactor.core.scheduler.Schedulers

```
public abstract class Schedulers
extends Object
```

Schedulers provides various Scheduler flavors usable by publishOn or subscribeOn :

- parallel(): Optimized for fast Runnable non-blocking executions
- single(): Optimized for low-latency Runnable one-off executions
- elastic(): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) can grow indefinitely
- boundedElastic(): Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) is capped
- immediate(): to immediately run submitted Runnable instead of scheduling them (somewhat of a no-op or "null object" Scheduler)
- fromExecutorService(ExecutorService) to create new instances around Executors

See [projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html](projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html)

## Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler

```
Scheduler publisher = Schedulers
   .newParallel("publisher", 1));
Flux
   .range(1, sMAX_ITERATIONS)
   .map(Integer::toUnsignedLong)
   .subscribeOn(publisher)
   .map(sGenerateRandomBigInt)
   .filter(sOnlyOdd)
   .doFinally(() -> publisher
                 .dispose())
   .subscribe(sink::next,
              error ->
                sink.complete(),
              sink::complete);
```

*Arrange to emit the random big integers in the "publisher" thread*

See Reactive/flux/ex2/src/main/java/FluxEx.java

## Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler
    - Not implemented via a "daemon thread"

See www.baeldung.com/java-daemon-thread

## Key Scheduler Operators Used By the Flux Class

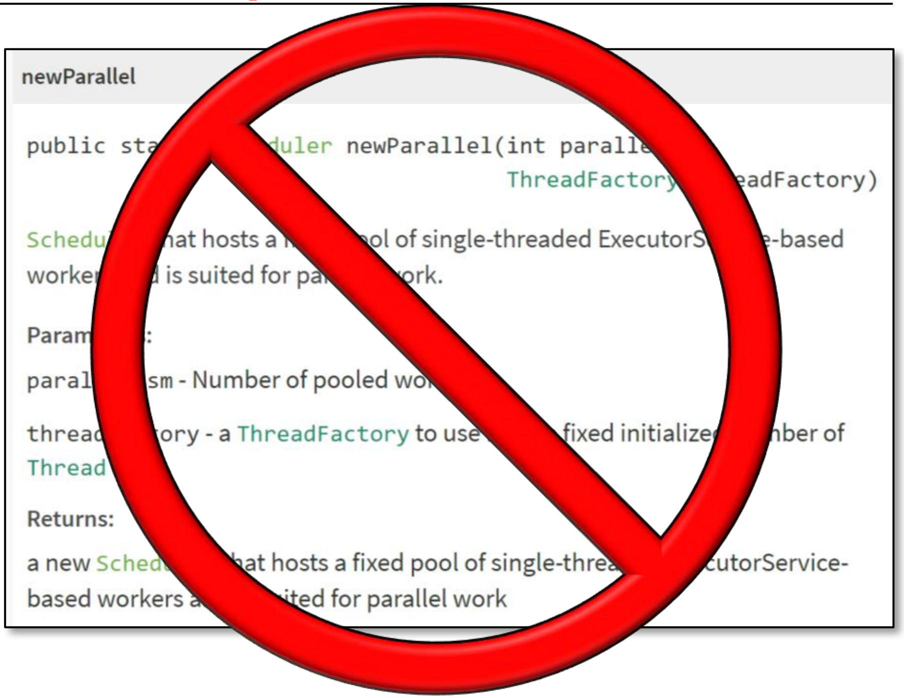
- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler
    - Not implemented via a "daemon thread"
      - i.e., the app will not exit until this pool is disposed of properly & explicitly

```
Scheduler publisher = Schedulers
   .newParallel("publisher", 1));
Flux
   .range(1, sMAX_ITERATIONS)
   .map(Integer::toUnsignedLong)
   .subscribeOn(publisher)
   .map(sGenerateRandomBigInt)
   .filter(sOnlyOdd)
   .doFinally(() -> publisher
                .dispose())
   .subscribe(sink::next,
              error ->
                sink.complete(),
              sink::complete);
```

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Scheduler.html#dispose

# Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler
  - RxJava's Schedulers doesn't have an equivalent method



newParallel

```
public sta    duler newParallel(int paralle
                              ThreadFactory    eadFactory)
```

Schedu    hat hosts a    ol of single-threaded ExecutorS    e-based worke    l is suited for pa    ork.

Param    :

paral    sm - Number of pooled wo

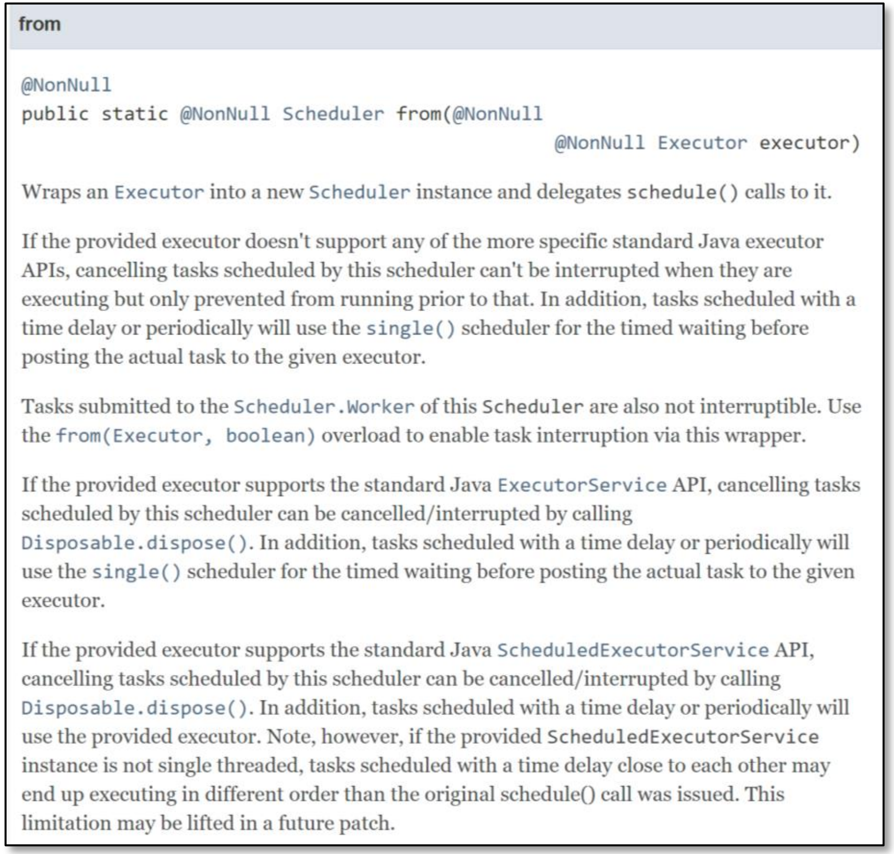
threa    ory - a ThreadFactory to use    fixed initialize    ber of Thread

Returns:

a new Sched    hat hosts a fixed pool of single-threa    cutorService-based workers a    ited for parallel work

# Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler
  - RxJava's Schedulers doesn't have an equivalent method
    - However, the from() method can be used in conjunction with Java's Executor framework



from

```
@NonNull
public static @NonNull Scheduler from(@NonNull
                                      @NonNull Executor executor)
```

Wraps an `Executor` into a new `Scheduler` instance and delegates `schedule()` calls to it.

If the provided executor doesn't support any of the more specific standard Java executor APIs, cancelling tasks scheduled by this scheduler can't be interrupted when they are executing but only prevented from running prior to that. In addition, tasks scheduled with a time delay or periodically will use the `single()` scheduler for the timed waiting before posting the actual task to the given executor.

Tasks submitted to the `Scheduler.Worker` of this `Scheduler` are also not interruptible. Use the `from(Executor, boolean)` overload to enable task interruption via this wrapper.

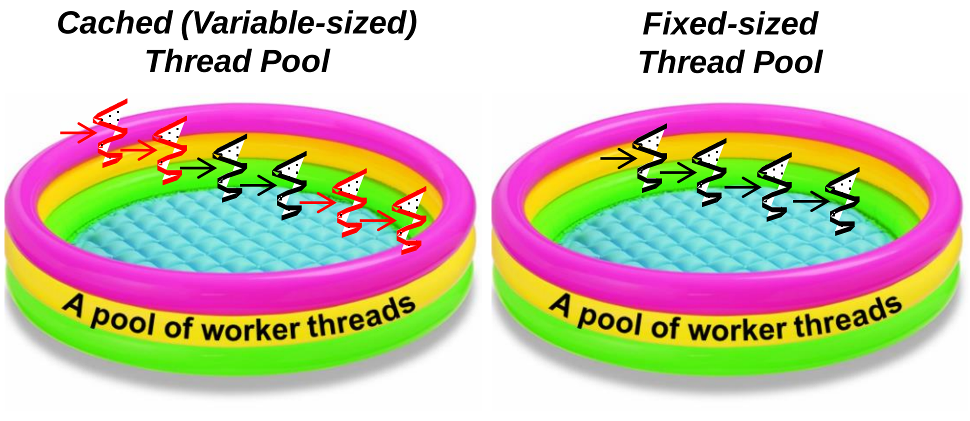
If the provided executor supports the standard Java `ExecutorService` API, cancelling tasks scheduled by this scheduler can be cancelled/interrupted by calling `Disposable.dispose()`. In addition, tasks scheduled with a time delay or periodically will use the `single()` scheduler for the timed waiting before posting the actual task to the given executor.

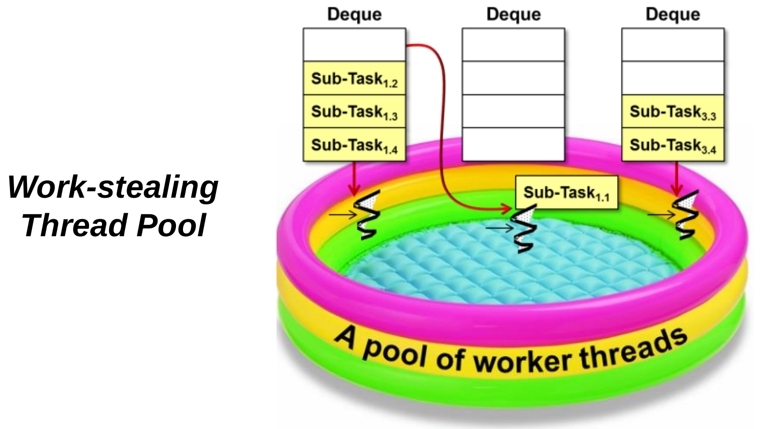
If the provided executor supports the standard Java `ScheduledExecutorService` API, cancelling tasks scheduled by this scheduler can be cancelled/interrupted by calling `Disposable.dispose()`. In addition, tasks scheduled with a time delay or periodically will use the provided executor. Note, however, if the provided `ScheduledExecutorService` instance is not single threaded, tasks scheduled with a time delay close to each other may end up executing in different order than the original schedule() call was issued. This limitation may be lifted in a future patch.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/schedulers/Schedulers.html#from

# Key Scheduler Operators Used By the Flux Class

- The Schedulers.newParallel() operator
  - Hosts a fixed-sized pool of single-threaded ExecutorService-based workers
  - Can be used to create a custom parallel scheduler
  - RxJava's Schedulers doesn't have an equivalent method
    - However, the from() method can be used in conjunction with Java's Executor framework, e.g.





See docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

End of Key Concurrency & Scheduler Operators in the Flux Class (Part 1)