Key Terminal Operators in the Observable Class (Part 1)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

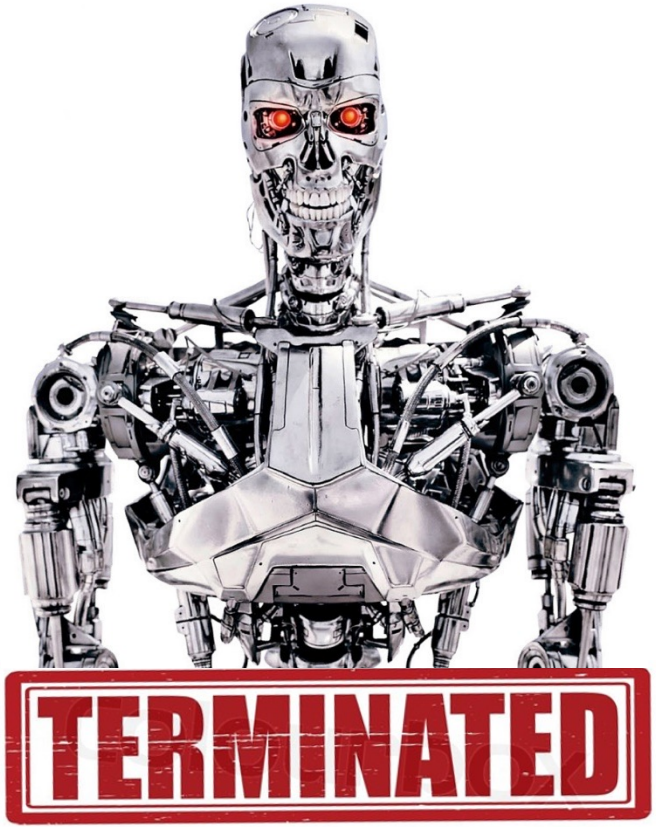
Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize key operators defined in—or used with—Observables
  - Factory method operators
  - Transforming operators
  - Action operators
  - Combining operators
  - Terminal operators
    - Terminate an Observable stream & trigger all the processing of operators in the stream
      - e.g., blockingSubscribe()



Key Terminal Operators in the Observable Class

Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable

```
void blockingSubscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

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

Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion

```
void blockingSubscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

Interface Consumer<T>

Type Parameters:

T - the type of the input to the operation

All Known Subinterfaces:

Stream.Builder<T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/functions/Consumer.html

Key Terminal Operators in the Observable Class

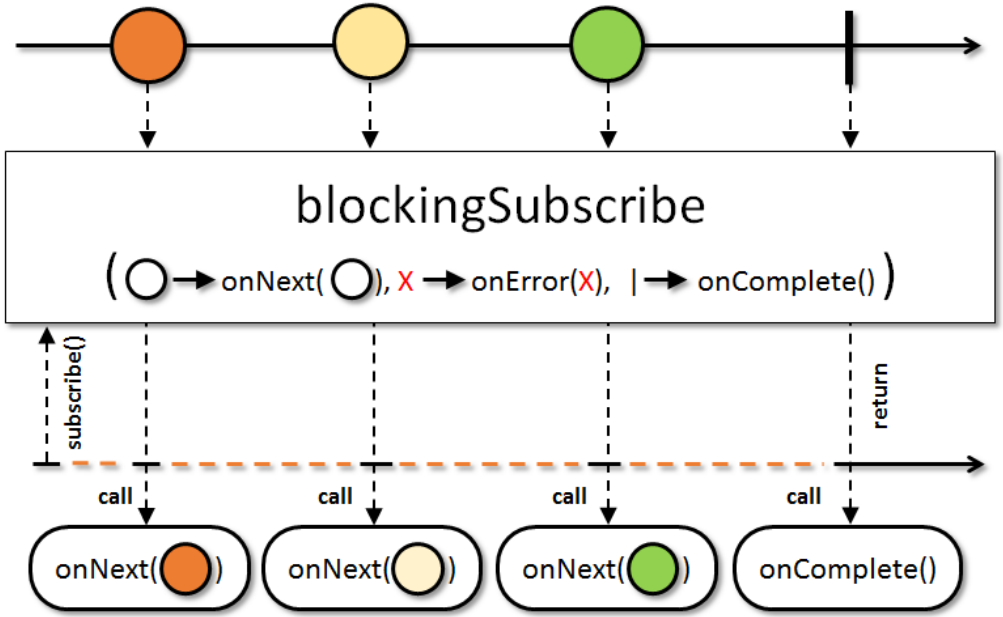
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests "unbounded demand"
        - i.e., Long.MAX_VALUE

```
void blockingSubscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests "unbounded demand"
    - Signals emitted to this operator are represented by the following regular expression:

```
onNext()*(onComplete()|onError())?
```



Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

Initiate processing & handle events

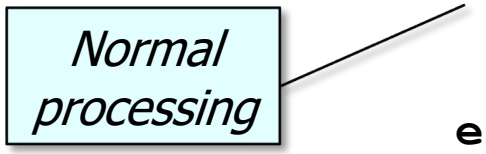
```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFrac))

   .blockingSubscribe
     (fraction -> sb.append(" = "
        + fraction.toMixedString()
        + "\n"),
      error -> {
        sb.append("error"); ...
      },
      () -> BigFractionUtils
       .display(sb.toString()));
```

See Reactive/Observable/ex1/src/main/java/ObservableEx.java

Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFrac))

   .blockingSubscribe
      (fraction -> sb.append(" = "
         + fraction.toMixedString()
         + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
        .display(sb.toString()));
```



Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFrac))

   .blockingSubscribe
      (fraction -> sb.append(" = "
         + fraction.toMixedString()
         + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
        .display(sb.toString()));
```

Error Processing

Key Terminal Operators in the Observable Class

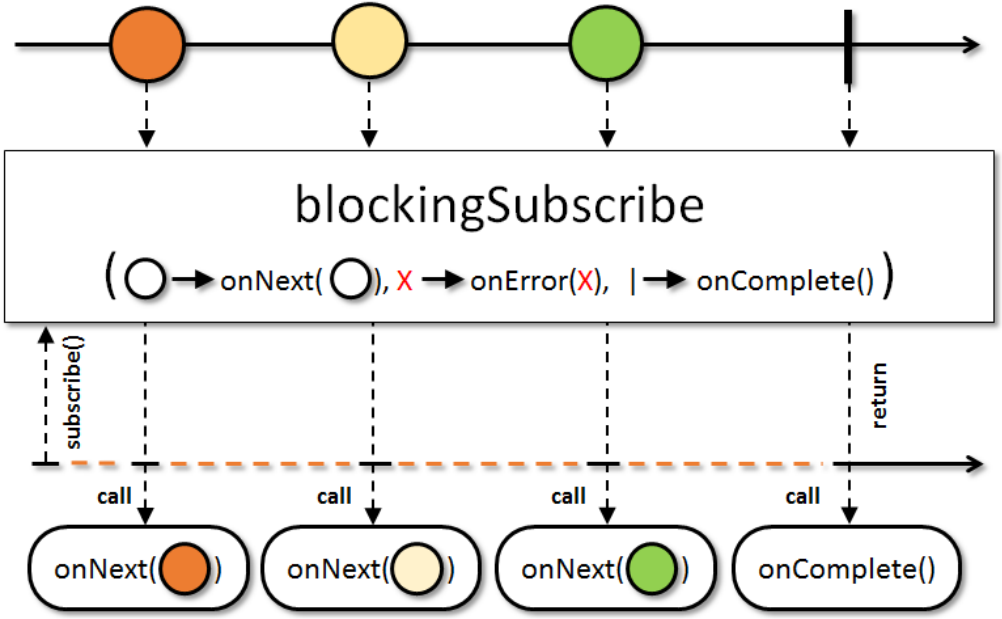
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFrac))

   .blockingSubscribe
      (fraction -> sb.append(" = "
         + fraction.toMixedString()
         + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
        .display(sb.toString()));
```

Completion Processing

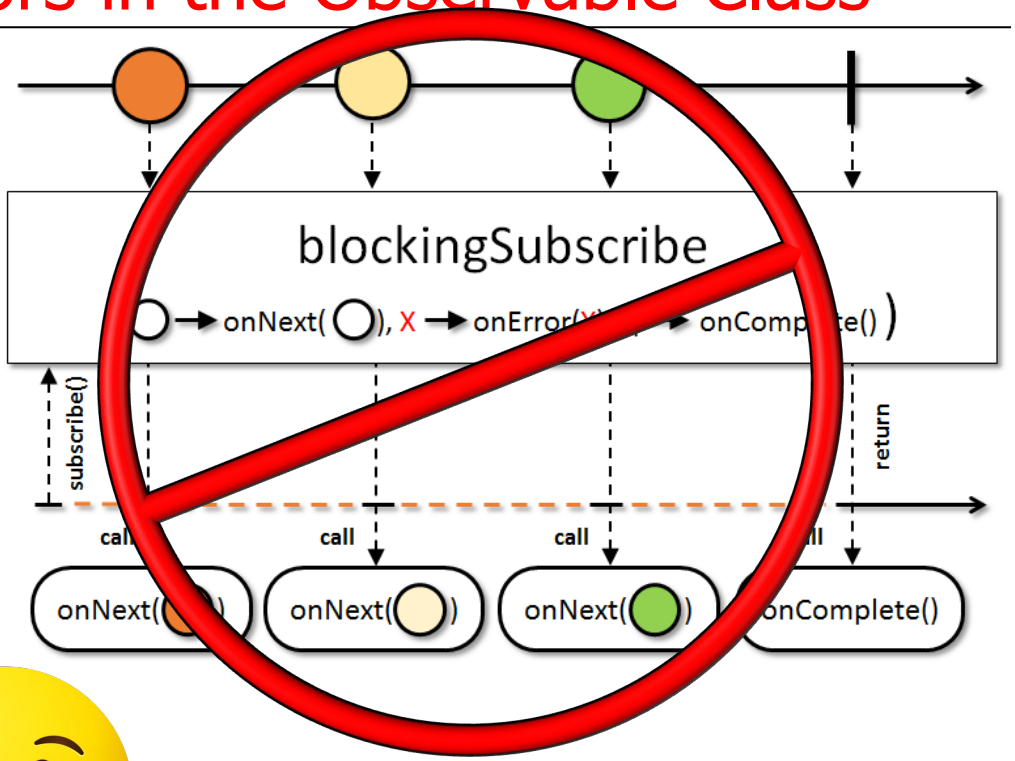
Key Terminal Operators in the Observable Class

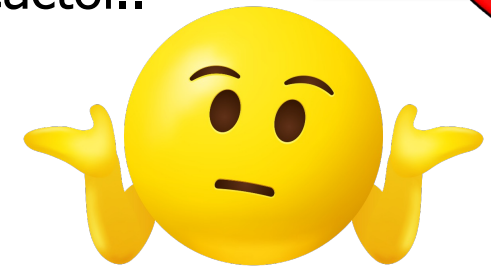
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will block the caller thread
    - Until the upstream terminates normally or with an error



Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will block the caller thread
- Oddly, there is no equivalent operator in Project Reactor..





Key Terminal Operators in the Observable Class

- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will block the caller thread
- Oddly, there is no equivalent operator in Project Reactor..
  - This omission complicates testing a bit, until you're comfortable with StepVerifier

`public interface StepVerifier`

A `StepVerifier` provides a declarative way of creating a verifiable script for an async `Publisher` sequence, by expressing expectations about the events that will happen upon subscription. The verification must be triggered after the terminal expectations (completion, error, cancellation) have been declared, by calling one of the `verify()` methods.

- Create a `StepVerifier` around a `Publisher` using `create(Publisher)` or `withVirtualTime(Supplier<Publisher>)` (in which case you should lazily create the publisher inside the provided `lambda`).
- Set up individual value expectations using `expectNext`, `expectNextMatches(Predicate)`, `assertNext(Consumer)`, `expectNextCount(long)` or `expectNextSequence(Iterable)`.
- Trigger subscription actions during the verification using either `thenRequest(long)` or `thenCancel()`.
- Finalize the test scenario using a terminal expectation: `expectComplete()`, `expectError()`, `expectError(Class)`, `expectErrorMatches(Predicate)`, or `thenCancel()`.
- Trigger the verification of the resulting `StepVerifier` on its `Publisher` using either `verify()` or `verify(Duration)`. (note some of the terminal expectations above have a "verify" prefixed alternative that both declare the expectation and trigger the verification).
- If any expectations failed, an `AssertionError` will be thrown indicating the failures.

See projectreactor.io/docs/test/release/api/reactor/test/StepVerifier.html

End of Key Terminal Operators in the Observable Class (Part 1)

Key Concurrency Operators for the Observable 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 operators defined in—or used with—Observables
  - Concurrency operators
    - These operators arrange to run other operators in designated threads & thread pools
      - e.g., subscribeOn() & observeOn()

Key Concurrency Operators in the Observable Class

Key Concurrency Operators in the Observable Class

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

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

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

Key Concurrency Operators in the Observable Class

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

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

Class Scheduler

java.lang.Object
io.reactivex.rxjava3.core.Scheduler

Direct Known Subclasses:
TestScheduler

```
public abstract class Scheduler
extends Object
```

A `Scheduler` is an object that specifies an API for scheduling units of work provided in the form of `Runnables` to be executed without delay (effectively as soon as possible), after a specified time delay or periodically and represents an abstraction over an asynchronous boundary that ensures these units of work get executed by some underlying task-execution scheme (such as custom Threads, event loop, `Executor` or Actor system) with some uniform properties and guarantees regardless of the particular underlying scheme.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Scheduler.html

Key Concurrency Operators in the Observable Class

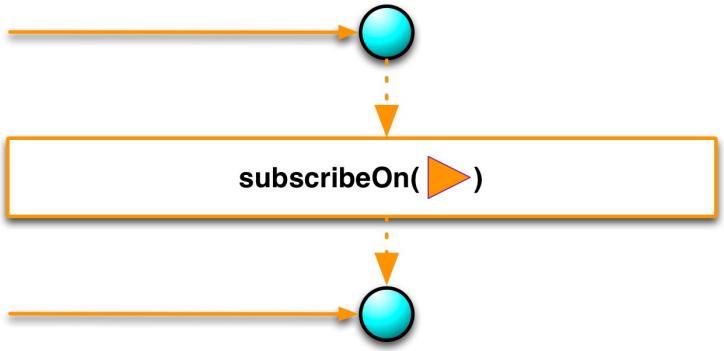
- The subscribeOn() operator
  - Run the subscribe(), request(), & onSubscribe() methods on the specified Scheduler worker
    - The scheduler param indicates what thread to perform the operation on
      - Scheduler is parameterized so that these mechanisms can also be reused in the Single class

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

subscribeOn

```
@CheckReturnValue
 @NonNull
 @SchedulerSupport(value="custom")
public final @NonNull Single<T> subscribeOn(@NonNull Scheduler scheduler)
```

Asynchronously subscribes `SingleObservers` to this `Single` on the specified `Scheduler`.



Scheduler:
You specify which `Scheduler` this operator will use.

Parameters:
`scheduler` - the `Scheduler` to perform subscription actions on

Returns:
the new `Single` instance

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Single.html

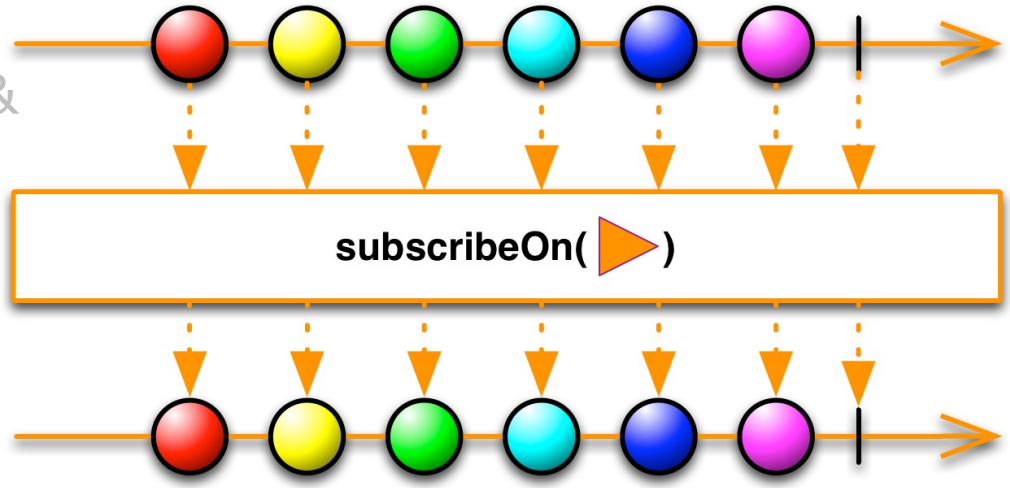
Key Concurrency Operators in the Observable Class

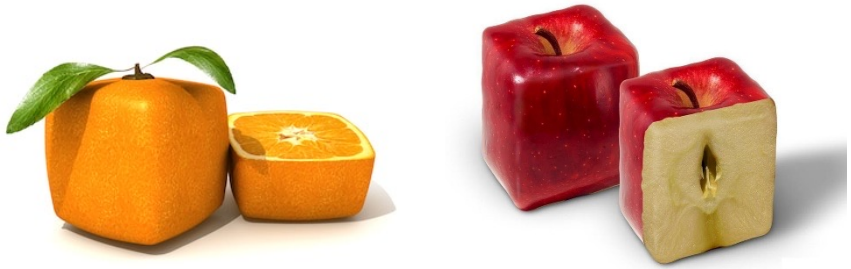
- The subscribeOn() operator
  - Run the subscribe(), request(), & onSubscribe() methods on the specified Scheduler worker
    - The scheduler param indicates what thread to perform the operation on
    - Returns the Observable requesting async processing

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

Key Concurrency Operators in the Observable Class

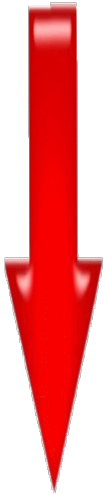
- The subscribeOn() operator
  - Run the subscribe(), request(), & onSubscribe() methods on the specified Scheduler worker
  - The subscribeOn() semantics are a bit unusual





Key Concurrency Operators in the Observable Class

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

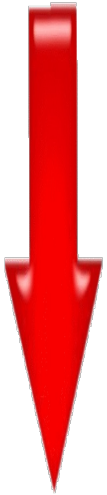


```
Observable
   .range(1, sMAX_ITERATIONS)
   .subscribeOn(Schedulers
      .newThread())
   .map(__ -> BigInteger
     .valueOf(lowerBound + rand
       .nextInt(sMAX_ITERATIONS)))
   .doOnNext(s ->
        ObservableEx.print(s, sb))
   .subscribe(emitter::next,
              error ->
              emitter.complete(),
              emitter::complete);
```

See Reactive/Observable/ex2/src/main/java/ObservableEx.java

Key Concurrency Operators in the Observable Class

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

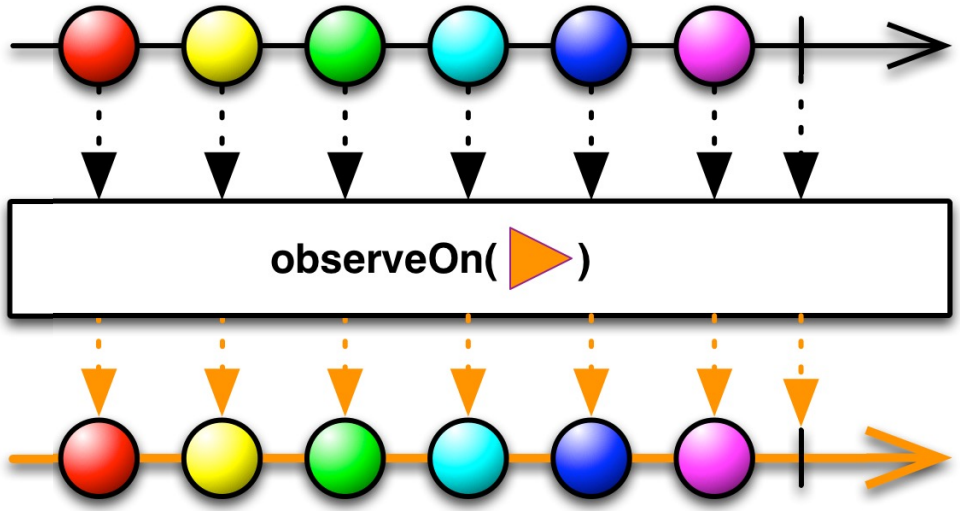


```
Observable
   .range(1, sMAX_ITERATIONS)
   .map(__ -> BigInteger
     .valueOf(lowerBound + rand
       .nextInt(sMAX_ITERATIONS)))
   .doOnNext(s ->
        ObservableEx.print(s, sb))
   .subscribeOn(Schedulers
     .newThread())
   .subscribe(emitter::next,
              error ->
              emitter.complete(),
              emitter::complete);
```

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

Key Concurrency Operators in the Observable Class

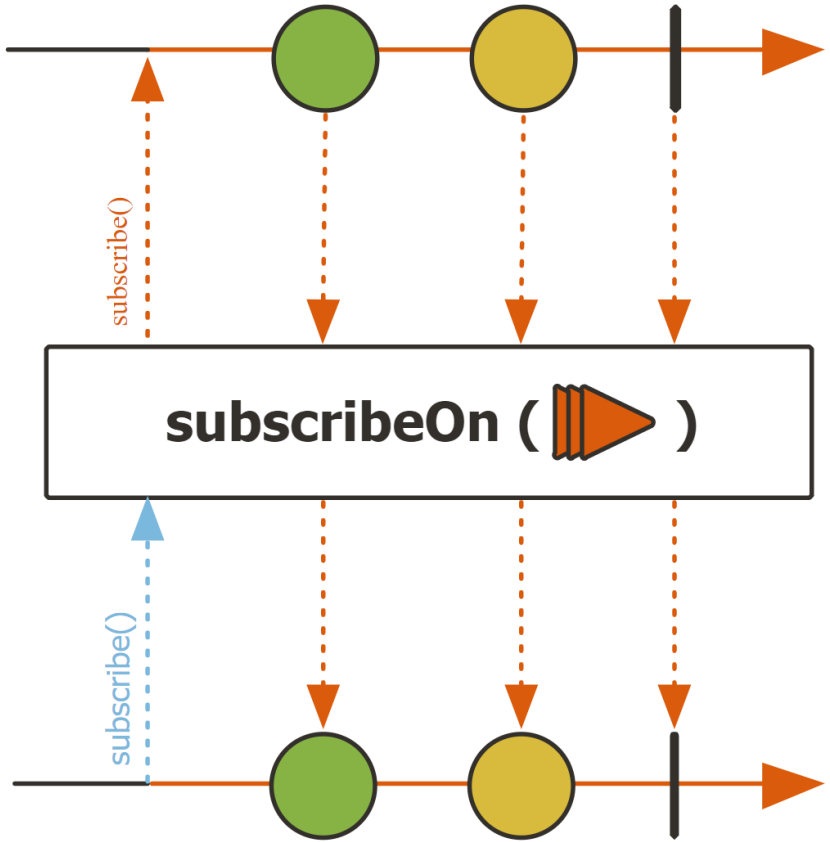
- The subscribeOn() operator
  - Run the subscribe(), request(), & onSubscribe() methods on the specified Scheduler worker
  - The subscribeOn() semantics are a bit unusual
    - Placing this operator in a chain impacts the execution context of the onNext(), onError(), & onComplete() signals
      -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



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

Key Concurrency Operators in the Observable Class

- The subscribeOn() operator
  - Run the subscribe(), request(), & onSubscribe() methods on the specified Scheduler worker
  - The subscribeOn() semantics are a bit unusual
  - Project Reactor's operator Flux.subscribeOn() works the same



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

Key Concurrency Operators in the Observable Class

- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker

```
Observable<T>
observeOn(Scheduler scheduler)
```

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

Key Concurrency Operators in the Observable Class

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

```
Observable<T>
observeOn(Scheduler scheduler)
```

Class Scheduler

java.lang.Object
io.reactivex.rxjava3.core.Scheduler

Direct Known Subclasses:
TestScheduler

```
public abstract class Scheduler
extends Object
```

A `Scheduler` is an object that specifies an API for scheduling units of work provided in the form of `Runnables` to be executed without delay (effectively as soon as possible), after a specified time delay or periodically and represents an abstraction over an asynchronous boundary that ensures these units of work get executed by some underlying task-execution scheme (such as custom Threads, event loop, `Executor` or Actor system) with some uniform properties and guarantees regardless of the particular underlying scheme.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Scheduler.html

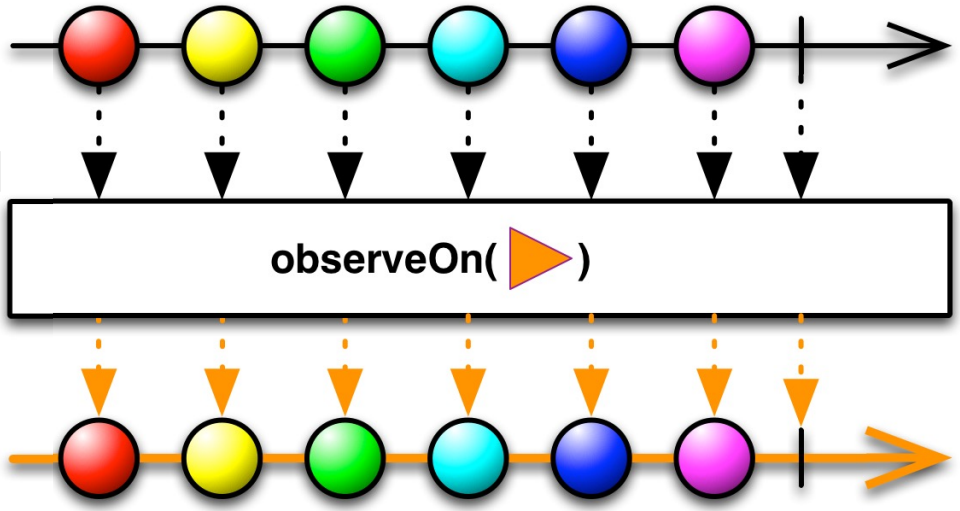
Key Concurrency Operators in the Observable Class

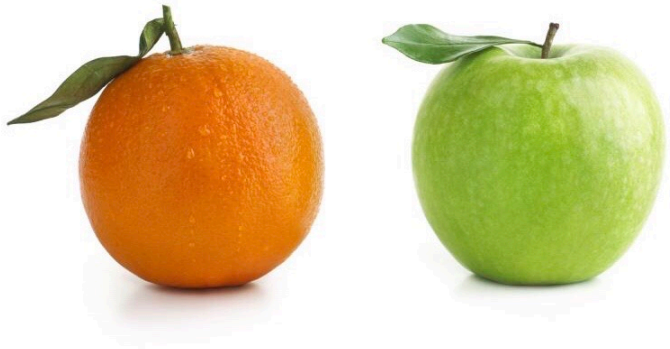
- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker
    - The scheduler param indicates what thread to perform the operation on
    - Returns the Observable requesting async processing

```
Observable<T>
observeOn(Scheduler scheduler)
```

Key Concurrency Operators in the Observable Class

- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker
  - The observeOn() semantics are fairly straightforward





Key Concurrency Operators in the Observable Class

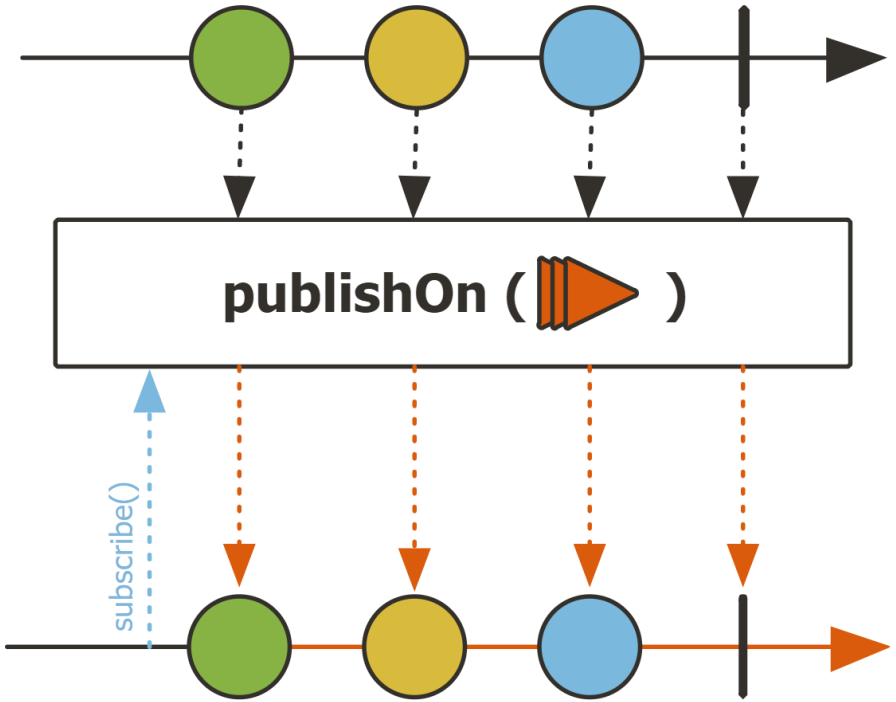
- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker
  - The observeOn() semantics are fairly straightforward
    - It influences the threading context where the rest of the operators in the chain below it execute
      - i.e., up to a new occurrence of observeOn() in a chain (if any)

```
return Observable
   .create(ObservableEx::emitAsync)
   .observeOn(Schedulers
      .newThread())
   .map(bi -> ObservableEx
        .checkIfPrime(bi, sb))
   .doOnNext(bi -> ObservableEx
      .processResult(bi, sb))
   .doOnComplete(() ->
         BigFractionUtils
         .display(sb.toString()))
   .count()
   .ignoreElement();
```

See Reactive/Observable/ex2/src/main/java/ObservableEx.java

Key Concurrency Operators in the Observable Class

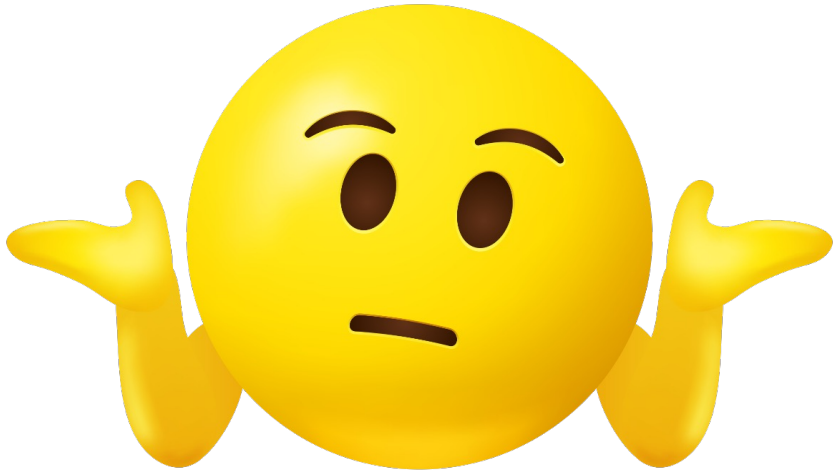
- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker
  - The observeOn() semantics are fairly straightforward
  - Project Reactor's operator Flux.publishOn() works the same



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

Key Concurrency Operators in the Observable Class

- The observeOn() operator
  - Run the onNext(), onComplete(), & onError() methods on a supplied Scheduler worker
  - The observeOn() semantics are fairly straightforward
  - Project Reactor's operator Flux. publishOn() works the same
    - It's unclear why this operator is named differently from RxJava's observeOn() operator



End of Key Concurrency Operators for the Observable Class (Part 1)

Key Suppressing Operators in the Observable Class

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 Observable operators
  - Concurrency & scheduler operators
  - Factory method operators
  - Action operators
  - Suppressing operators
    - These operators create an Observable and/or Single that changes or ignores (portions of) its payload
      - e.g., take() & ignoreElements()

Key Suppressing Operators in the Observable Class

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available

```
Observable<T>
take(long n)
```

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

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
    - The param is the # of items to emit from this Observable

```
Observable<T>
take(long n)
```

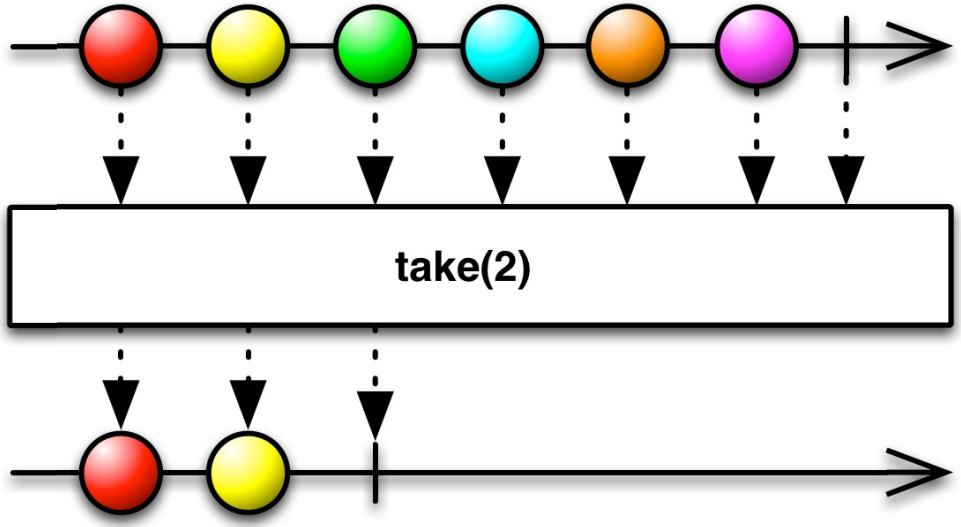
Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
    - The param is the # of items to emit from this Observable
    - Returns an Observable limited to size N

```
Observable<T>
take(long n)
```

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
  - Used to limit otherwise "infinite" streams



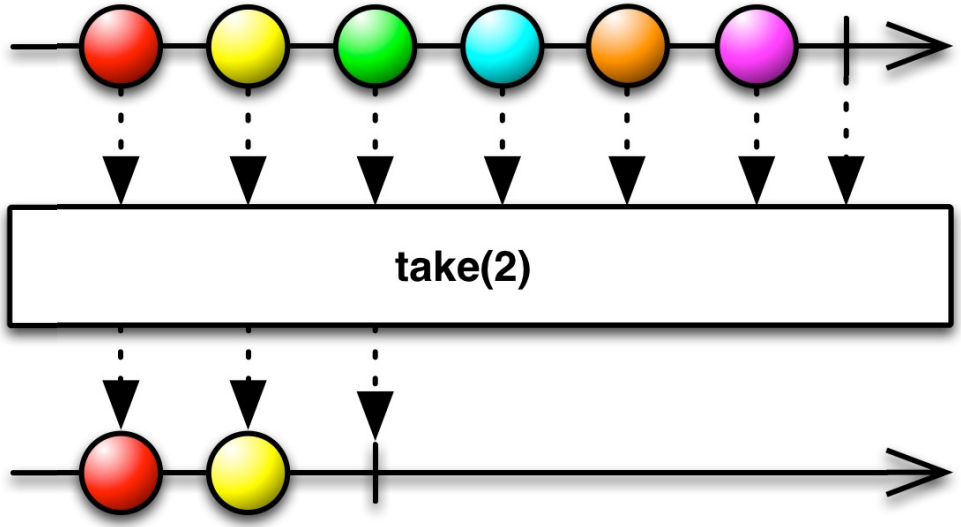
Generate an infinite series of integers periodically in a background thread

```
Observable
   .interval(sSLEEP.toMillis(),
             MILLISECONDS)
   ...
   .take(sMAX_ITERATIONS)
```

See previous discussion of the Observable.interval() method

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
  - Used to limit otherwise "infinite" streams



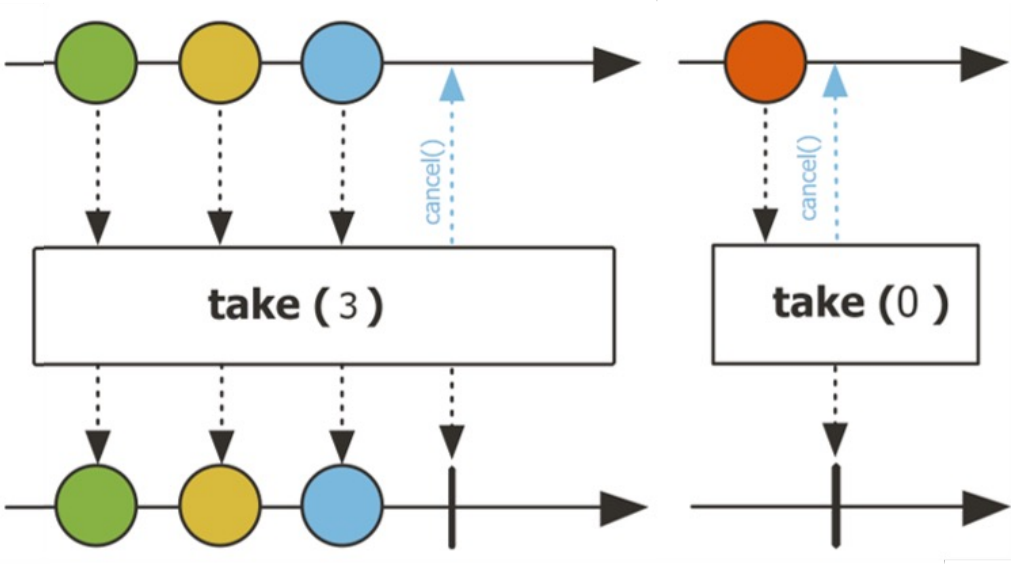

```
Observable
   .interval(sSLEEP.toMillis(),
                MILLISECONDS)
   ...
   .take(sMAX_ITERATIONS)
```

Stop emitting after sMAX_ITERATIONS

See Reactive/Observable/ex2/src/main/java/ObservableEx.java

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
  - Used to limit otherwise "infinite" streams
  - Project Reactor's Flux.take() operator works the same




```
Flux
   .interval
      (sSLEEP_DURATION)
   ...
   .take(sMAX_ITERATIONS)
   ...
```

Only process sMAX_ITERATIONS # of emitted values from interval()

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

Key Suppressing Operators in the Observable Class

- The take() operator
  - Take only the first N values from this Observable, if available
  - Used to limit otherwise "infinite" streams
  - Project Reactor's Flux.take() operator works the same
- Similar to Stream.limit() in Java Streams

limit

`Stream<T> limit(long maxSize)`

Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.

This is a short-circuiting stateful intermediate operation.

```
List<Long> oddNumbers = Stream
   .iterate(1L, l -> l + 1)
   .filter(n -> (n & 1) != 0)
   .limit(100)
   .collect(toList());
```

Only emit 100 odd #'s

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#limit

Key Suppressing Operators in the Observable Class

- The ignoreElements() operator
  - Ignores all items emitted by the current Observable

```
Completable
ignoreElements()
```

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

Key Suppressing Operators in the Observable Class

- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
    - It only calls onComplete() or onError()
      - But not onNext()!

```
Completable
ignoreElements()
```

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

Key Suppressing Operators in the Observable Class

- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
    - It only calls onComplete() or onError()
  - Returns a new Completable instance
    - i.e., emits no value, but only completion or error

`Completable`

`ignoreElements()`

Class Completable

java.lang.Object
io.reactivex.rxjava3.core.Completable

All Implemented Interfaces:
CompletableSource

Direct Known Subclasses:
CompletableSubject

```
public abstract class Completable
extends Object
implements CompletableSource
```

The `Completable` class represents a deferred computation without any value but only indication for completion or exception.

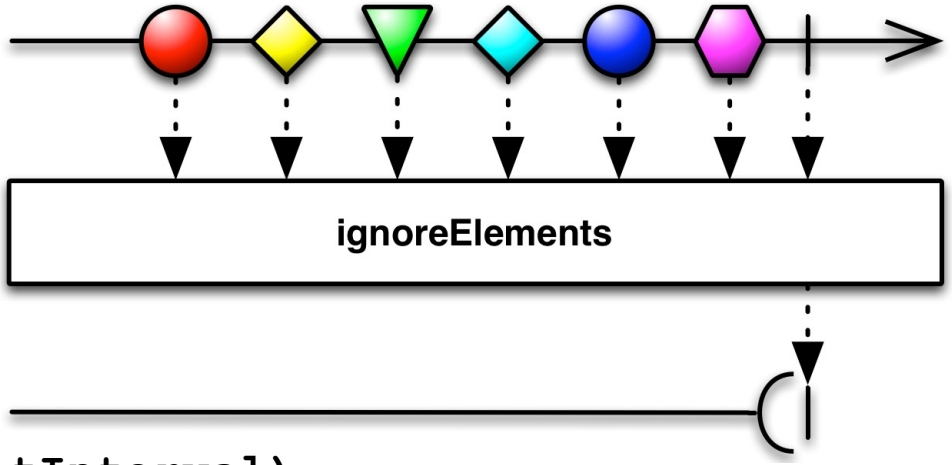
`Completable` behaves similarly to `Observable` except that it can only emit either a completion or error signal (there is no `onNext` or `onSuccess` as with the other reactive types).

The `Completable` class implements the `CompletableSource` base interface and the default consumer type it interacts with is the `CompletableObserver` via the `subscribe(CompletableObserver)` method. The `Completable` operates with the following sequential protocol:

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Completable.html

Key Suppressing Operators in the Observable Class

- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
  - This "data-suppressing" operator ignores its payload



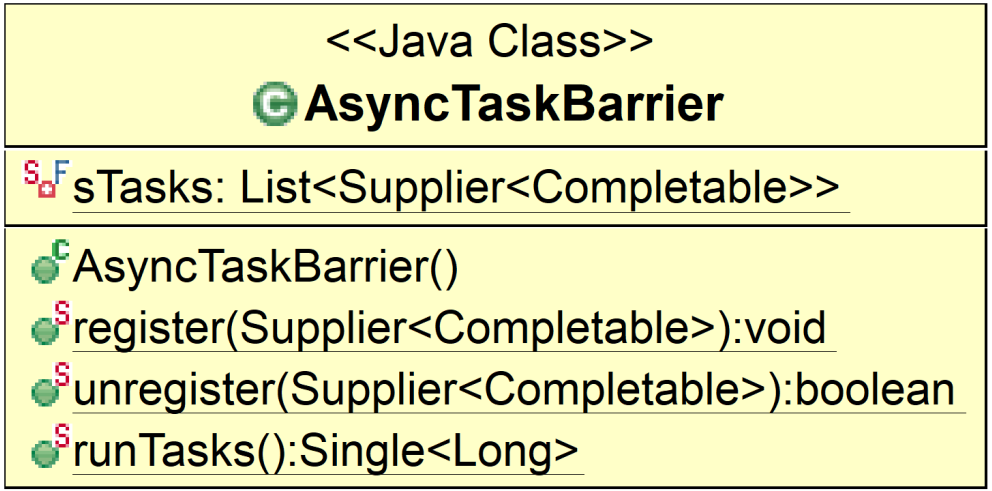

```
return Observable
        .create(ObservableEx::emitInterval)
        .map(bigInteger -> ObservableEx
                .checkIfPrime(bigInteger, sb))
        .doOnComplete(() -> BigFractionUtils
                        .display(sb.toString()))
        .ignoreElements();
```

Indicate an async operation completed

See Reactive/Observable/ex2/src/main/java/ObservableEx.java

Key Suppressing Operators in the Observable Class

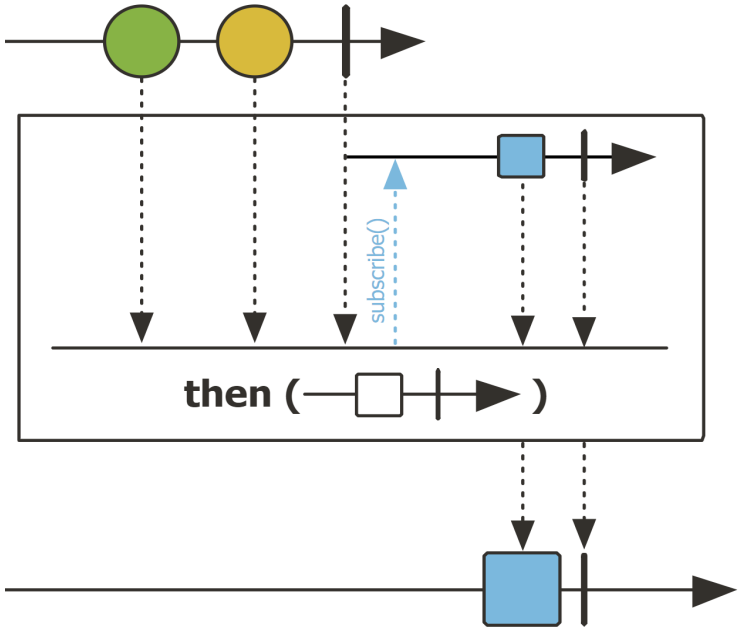
- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
  - This "data-suppressing" operator ignores its payload
    - Used by the AsyncTaskBarrier framework to determine when an async task completes



See Reactive/Observable/ex2/src/main/java/utils/AsyncTaskBarrier.java

Key Suppressing Operators in the Observable Class

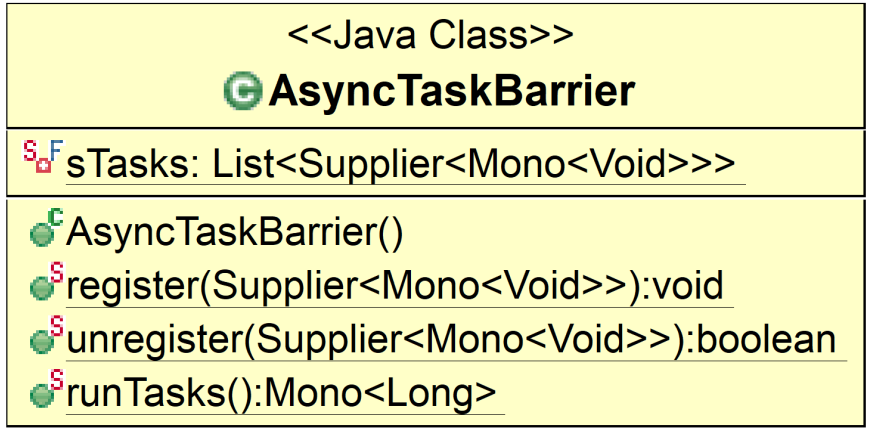
- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
  - This "data-suppressing" operator ignores its payload
  - Project Reactor doesn't really have an equivalent, though its then() operator can be used in a similar way

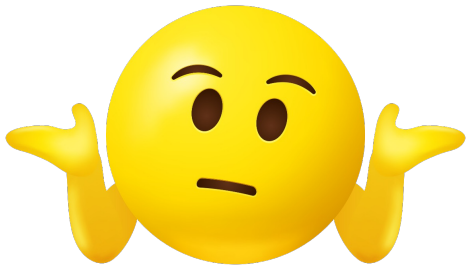


See projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#then

Key Suppressing Operators in the Observable Class

- The ignoreElements() operator
  - Ignores all items emitted by the current Observable
  - This "data-suppressing" operator ignores its payload
  - Project Reactor doesn't really have an equivalent, though its then() operator can be used in a similar way
    - Also used by the AsyncTaskBarrier to determine when an async task completes





See Reactive/flux/ex2/src/main/java/utils/AsyncTaskBarrier.java

End of Key Suppressing Operators in the Observable Class

Key Terminal Operators in the Observable Class (Part 2)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

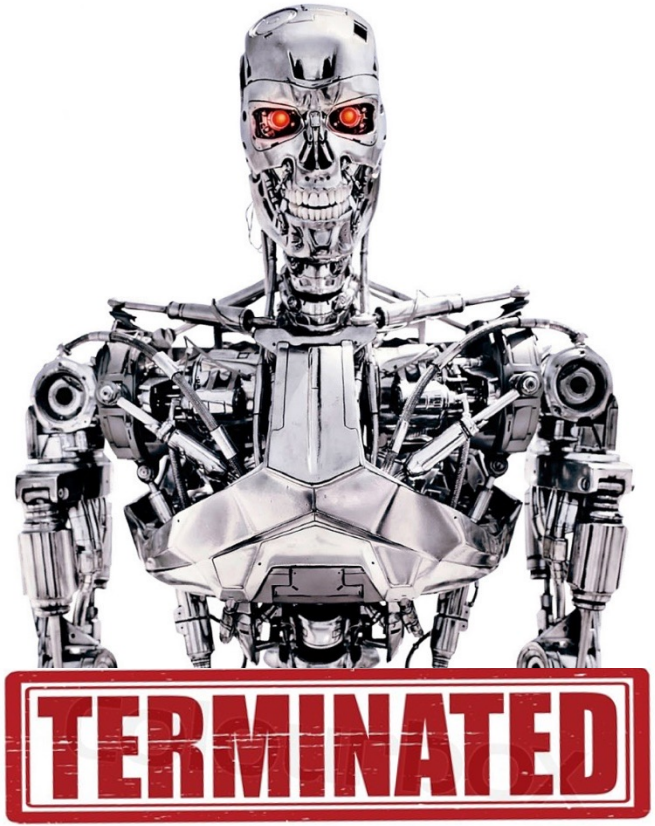
Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize key Observable operators
  - Concurrency & scheduler operators
  - Factory method operators
  - Action operators
  - Suppressing operators
  - Terminal operators
    - Terminate an Observable stream & trigger all the processing of operators in the stream
      - e.g., subscribe()



The subscribe() operator is non-blocking, unlike blockingSubscribe()

Key Terminal Operators in the Observable Class

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable

```
Disposable subscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

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

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion

```
Disposable subscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

Interface Consumer<T>

Type Parameters:

T - the type of the input to the operation

All Known Subinterfaces:

Stream.Builder<T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/functions/Consumer.html

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests unbounded demand
        - i.e., Long.MAX_VALUE

```
Disposable subscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests unbounded demand
    - Signals emitted to this operator are represented by the following regular expression:

      `onNext()*(onComplete()|onError())?`

subscribe

```
@CheckReturnValue
 @SchedulerSupport(value="none")
 @NonNull
public final @NonNull Disposable subscribe(@NonNull @NonNull Consumer<? super T> onNext,
        @NonNull @NonNull Consumer<? super Throwable> onError,
        @NonNull @NonNull Action onComplete)
```

Subscribes to the current `Observable` and provides callbacks to handle the items it emits and any error or completion notification it signals.

Scheduler:

`subscribe` does not operate by default on a particular `Scheduler`.

Parameters:

`onNext` - the `Consumer<T>` you have designed to accept emissions from the current `Observable`

`onError` - the `Consumer<Throwable>` you have designed to accept any error notification from the current `Observable`

`onComplete` - the `Action` you have designed to accept a completion notification from the current `Observable`

Returns:

the new `Disposable` instance that can be used to dispose the subscription at any time

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
    - A Disposable is returned, which indicates a task or resource that can be cancelled/disposed

```
Disposable subscribe
   (Consumer<? super T> consumer,
    Consumer<? super Throwable>
      errorConsumer,
    Runnable completeConsumer)
```

```
@FunctionalInterface
public interface Disposable
```

Indicates that a task or resource can be cancelled/disposed.

Call to the dispose method is/should be idempotent.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/disposables/Disposable.html

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
    - A Disposable is returned, which indicates a task or resource that can be cancelled/disposed
      - Disposables can be accumulated & disposed in one fell swoop!

```
CompositeDisposable
  mDisposables
    (mPublisherScheduler,
     mSubscriberScheduler,
     mSubscriber);
...

mDisposables.dispose();
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/disposables/CompositeDisposable.html

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

Initiate stream processing & handle events

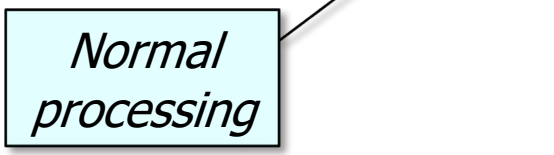
```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFraction))

   .subscribe
      (fraction -> sb.append(" = "
       + fraction.toMixedString()
       + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
             .display(sb.toString()));
```

See Reactive/Obervable/ex2/src/main/java/ObservableEx.java

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFraction))

   .subscribe
      (fraction -> sb.append(" = "
       + fraction.toMixedString()
       + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
             .display(sb.toString()));
```



Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFraction))

   .subscribe
      (fraction -> sb.append(" = "
       + fraction.toMixedString()
       + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
             .display(sb.toString()));
```

Error Processing

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

```
Observable
   .fromIterable(bigFractionList)

   .map(fraction -> fraction
      .multiply(sBigReducedFraction))

   .subscribe
      (fraction -> sb.append(" = "
       + fraction.toMixedString()
       + "\n"),
       error -> {
         sb.append("error"); ...
       },
       () -> BigFractionUtils
                .display(sb.toString()));
```

Completion Processing

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will *not* block the caller thread
    - Until upstream terminates normally or with an error

subscribe

```
@CheckReturnValue
 @SchedulerSupport(value="none")
 @NonNull
public final @NonNull Disposable subscribe(@NonNull Consumer<? super T> onNext,
        @NonNull Consumer<? super Throwable> onError)
```

Subscribes to the current `Observable` and provides callbacks to handle the items it emits and any error notification it signals.

Scheduler:

`subscribe` does not operate by default on a particular `Scheduler`.

Parameters:

`onNext` - the `Consumer<T>` you have designed to accept e… …gren…
`Observable`
`onError` - the `Consumer<Throwable>` you have desig… to a… …catio…
from the current `Observable`

Returns:

the new `Disposable` instance that can be used to disp… the …ription at …me

Throws:

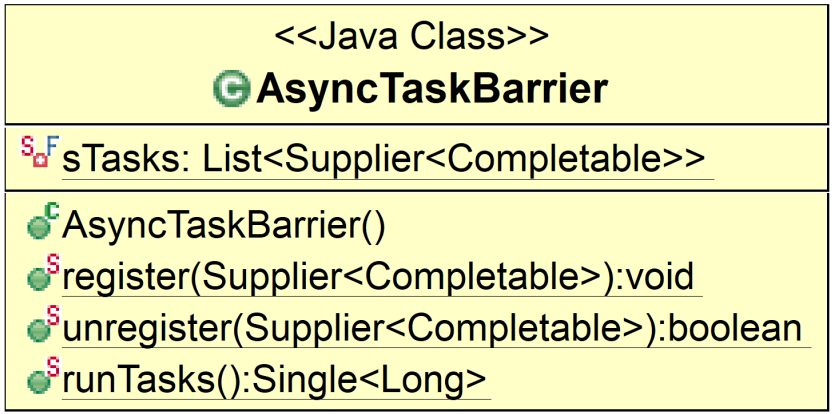
`NullPointerException` - if `onNext` or `onError` is `null`

See Also:

ReactiveX operators documentation: Subscribe

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will *not* block the caller thread
    - Until upstream terminates normally or with an error
    - These semantics motivate the need for the AsyncTaskBarrier framework!



See Reactive/Observable/ex2/src/main/java/utils/AsyncTaskBarrier.java

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will *not* block the caller thread
  - Other versions of subscribe() support different capabilities

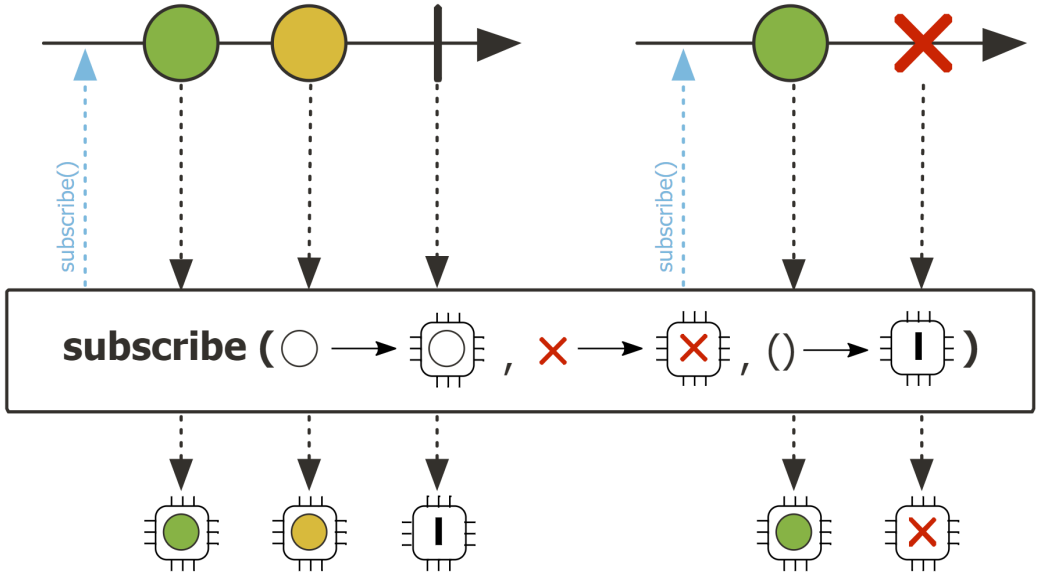
```
void subscribe
   (Observer<? super T> observer)
```

Subscribes the given Observer to this ObservableSource instance, which provides additional capabilities for receiving push-based notifications

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

Key Terminal Operators in the Observable Class

- The subscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain
  - Calling this operator will *not* block the caller thread
  - Other versions of subscribe() support different capabilities
  - Project Reactor's operator Flux .subscribe() works the same



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

End of Key Terminal Operators in the Observable Class (Part 2)

Key Transforming Operators in the Observable Class (Part 2)

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 operators defined in—or used with—Observables
  - Factory method operators
  - Transforming operators
    - Transform the values and/or types emitted by an Observable
      - e.g., flatMapCompletable()

Key Transforming Operators in the Observable Class

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable

```
Completable
flatMapCompletable
   (Function<? super T,
                ? extends
                  CompletableSource>
    mapper))
```

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

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable, e.g.,
    - Maps each element of the current Observable into CompletableSource objects

```
Completable
flatMapCompletable
   (Function<? super T,
             ? extends
               CompletableSource>
    mapper))
```

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable, e.g.,
    - Maps each element of the current Observable into CompletableSource objects
    - Subscribes to them & waits for the completion of the upstream & all CompletableSource objects

```
Completable
flatMapCompletable
   (Function<? super T,
             ? extends
               CompletableSource>
    mapper))
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/CompletableSource.html

Key Transforming Operators in the Observable Class

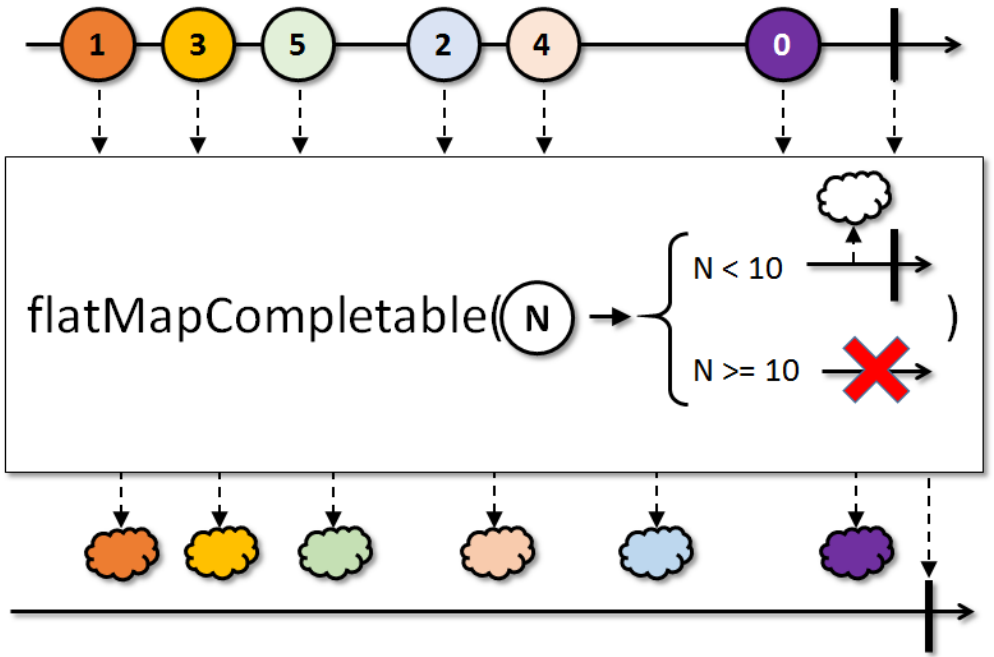
- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable, e.g.,
    - Maps each element of the current Observable into CompletableSource objects
    - Subscribes to them & waits for the completion of the upstream & all CompletableSource objects
    - Returns the new Completable instance

```
Completable
flatMapCompletable
   (Function<? super T,
             ? extends
               CompletableSource>
    mapper))
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Completable.html

Key Transforming Operators in the Observable Class

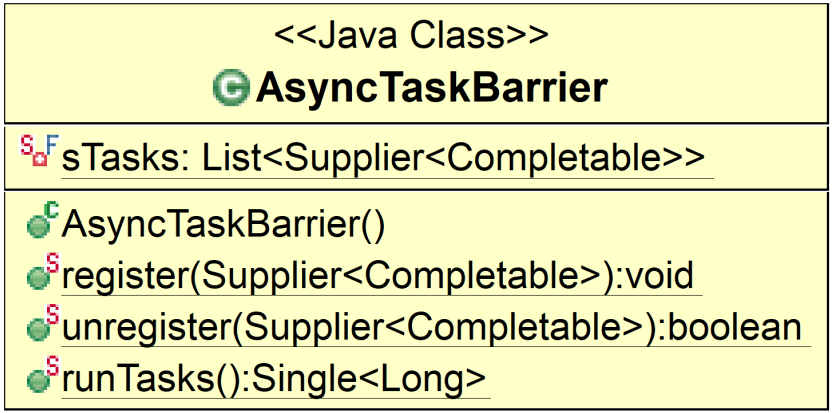
- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())



See medium.com/@daniel.rodak/combining-rxjava2-completable-with-observable-6dda410a3c83

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
    - Used to integrate w/the RxJava AsyncTaskBarrier framework



See Reactive/Observable/ex3/src/main/java/utils/AsyncTaskBarrier.java

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
    - Used to integrate w/the RxJava AsyncTaskBarrier framework
      - i.e., the Completable isn't triggered until all async processing is finished

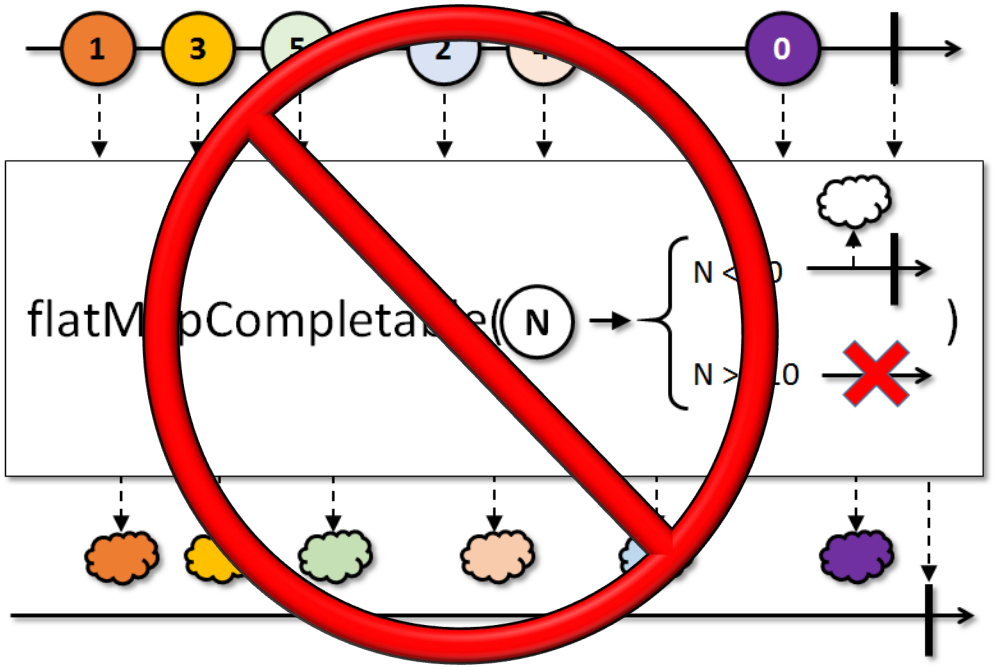
```
Observable
    .fromIterable(sTasks)

    .map(Supplier::get)

    .flatMapCompletable(c -> c)

    .toSingleDefault((long)
                          sTasks.size());
```

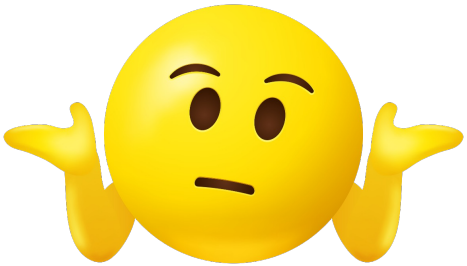
Map each Observable element into a CompletableSource, subscribes to them, & wait until the upstream & all CompletableSource objects complete

See Reactive/Observable/ex3/src/main/java/utils/AsyncTaskBarrier.java

Key Transforming Operators in the Observable Class

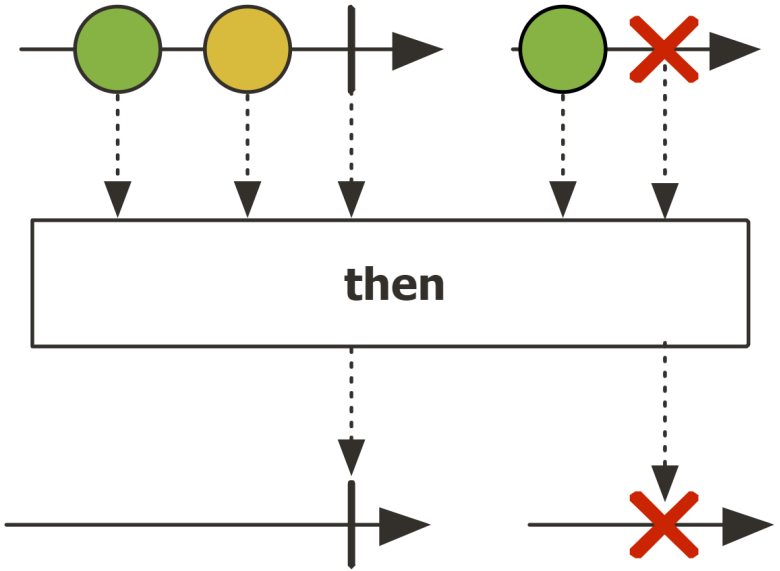
- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
  - Project Reactor has no operator like flatMapCompletable()





Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
  - Project Reactor has no operator like flatMapCompletable()
    - However, Project Reactor's Flux.then() & Mono.then() operators provide a similar capability when used in conjunction with flatMap()



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

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
  - Project Reactor has no operator like flatMapCompletable()
    - However, Project Reactor's Flux.then() & Mono.then() operators provide a similar capability when used in conjunction with flatMap()
      - Used to integrate w/the Project Reactor AsyncTaskBarrier framework

```
Flux
    .fromIterable(sTasks)

    .flatMap(Supplier::get)

    .collectList()

    .onErrorContinue(errorHandler)

    .flatMap(__ -> ...);
```

See Reactive/flux/ex3/src/main/java/utils/AsyncTaskBarrier.java

Key Transforming Operators in the Observable Class

- The flatMapCompletable() operator
  - "flatMaps" an Observable into a Completable
  - The Completable returned waits for the upstream's Observable terminal event (onComplete())
  - Project Reactor has no operator like flatMapCompletable()
  - The CompletableFuture.allOf() method can be combined with the Java Streams collector framework for a similar effect

```
Stream
   .generate(() ->
      makeBigFraction
         (new Random(), false))

   .limit(sMAX_FRACTIONS)

   .map(reduceAndMultiplyFraction)

   .collect(FuturesCollector
            .toFuture())

   .thenAccept
      (this::sortAndPrintList);
```

See Java8/ex19/src/main/java/utils/FuturesCollector.java

End of Key Transforming Operators in the Observable Class (Part 2)

Key Scheduler Operators for the Observable Class (Part 2)

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 operators defined in—or used with—Observables
  - Factory method operators
  - Transforming operators
  - Scheduler operators
    - These operators provide the context to run other operators in designated threads & thread pools
      - e.g., Schedulers.computation()

These operators also work with the Flowable, ParallelFlowable, Single, & Maybe classes

Key Scheduler Operators for the Observable Class

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers

```
static Scheduler computation()
```

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

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for parallel work

```
static Scheduler computation()
```

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for parallel work
      - Optimized for fast Runnable non-blocking executions

Class Schedulers

java.lang.Object
io.reactivex.rxjava3.schedulers.Schedulers

```
public final class Schedulers
extends Object
```

Static factory methods for returning standard Scheduler instances.

The initial and runtime values of the various scheduler types can be overridden via the `RxJavaPlugins.setInit(scheduler name)SchedulerHandler()` and `RxJavaPlugins.set(scheduler name)SchedulerHandler()` respectively.

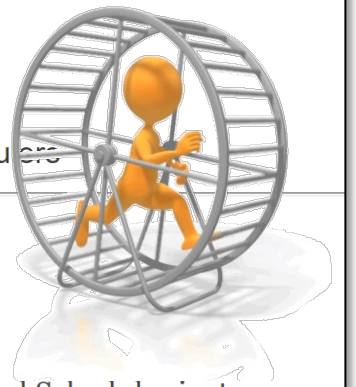
See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/schedulers/Schedulers.html

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for parallel work
      - Optimized for fast Runnable non-blocking executions
        - i.e., compute-/CPU-bound tasks, *not* I/O-bound tasks!

Class Schedulers

java.lang.Object
io.reactivex.rxjava3.schedulers.Schedulers



```
public final class Schedulers
extends Object
```

Static factory methods for returning standard Scheduler instances.

The initial and runtime values of the various scheduler types can be overridden via the `RxJavaPlugins.setInit(scheduler name)SchedulerHandler()` and `RxJavaPlugins.set(scheduler name)SchedulerHandler()` respectively.

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work

```
return Observable
   .fromIterable(bigFractionList)

   .flatMap(bf -> Observable
      .fromCallable(() -> bf
         .multiply(sBigFraction))

      .subscribeOn
         (Schedulers
          .computation()))

   .reduce(BigFraction::add)
```

Arrange to multiply a List of Big Integer objects in a background thread in computation thread pool

See Reactive/Observable/ex3/src/main/java/ObservableEx.java

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work

```
return Observable
   .fromIterable(bigFractionList)

   .flatMap(bf -> Observable
      .fromCallable(() -> bf
         .multiply(sBigFraction))

      .subscribeOn
         (Schedulers
          .computation()))

   .reduce(BigFraction::add)
```

Each BigFraction emitted via from Callable() is multiplied in parallel within the computation thread pool

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work

```
return Observable
   .fromIterable(bigFractionList)

   .flatMap(bf -> Observable
      .fromCallable(() -> bf
         .multiply(sBigFraction))

      .subscribeOn
         (Schedulers
          .computation()))

   .reduce(BigFraction::add)
```

fromCallable() is a "lazy" factory method so multiply() runs in the computation thread pool even though subscribeOn() comes after

Key Scheduler Operators for the Observable Class

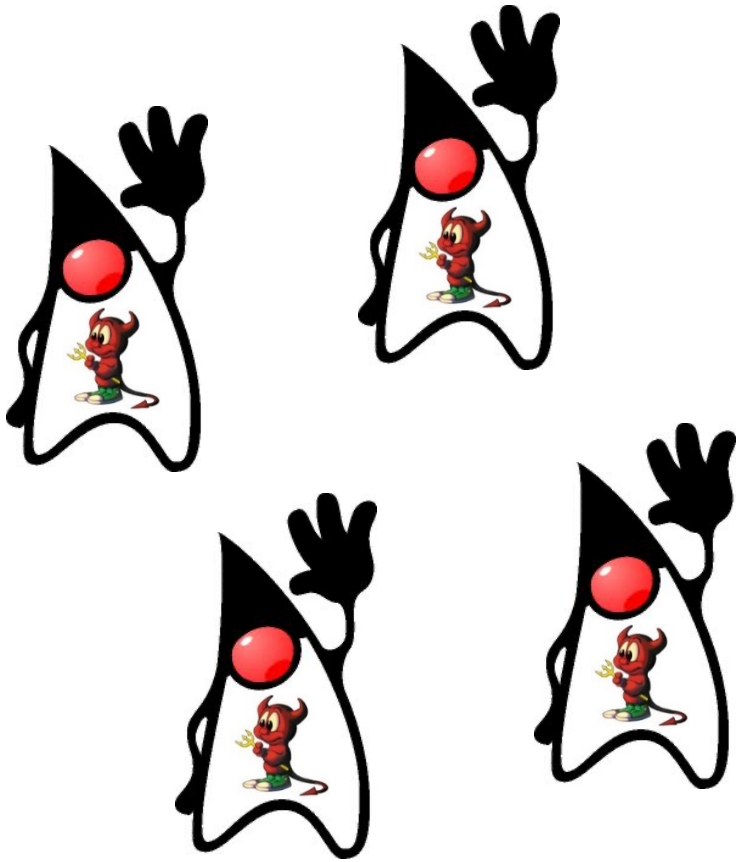
- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work

```
return Observable
   .fromIterable(bigFractionList)

   .flatMap(bf -> Observable
      .fromCallable(() -> bf
         .multiply(sBigFraction))

      .subscribeOn
         (Schedulers
          .computation()))

   .reduce(BigFraction::add)
```

Only one thread runs reduce() after all other computations are done

Key Scheduler Operators for the Observable Class

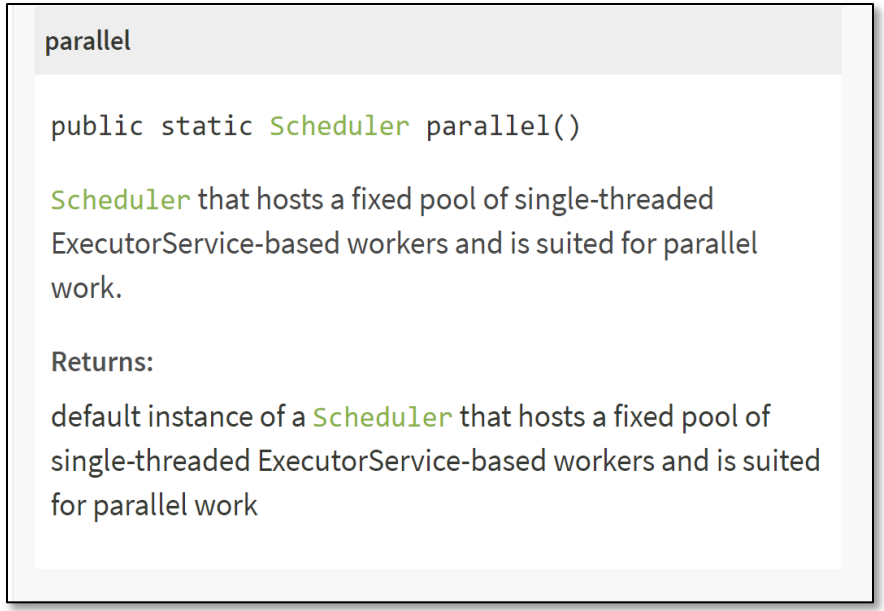
- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work
  - Implemented via "daemon threads"
    - i.e., won't prevent the app from exiting even if its work isn't done



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

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work
  - Implemented via "daemon threads"
- The Schedulers.parallel() operator in Project Reactor is similar
  - i.e., intended for compute-/CPU-bound tasks, not I/O-bound tasks



parallel

```
public static Scheduler parallel()
```

`Scheduler` that hosts a fixed pool of single-threaded ExecutorService-based workers and is suited for parallel work.

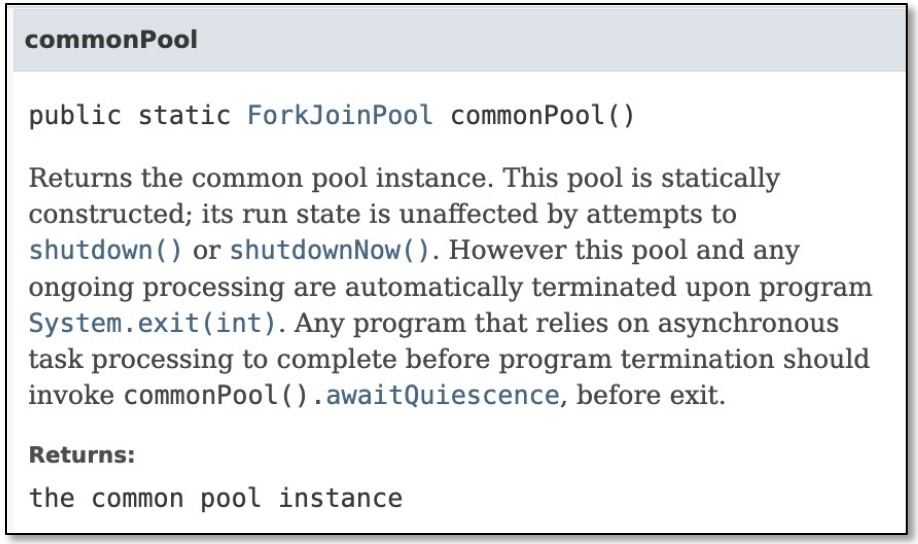
Returns:

default instance of a `Scheduler` that hosts a fixed pool of single-threaded ExecutorService-based workers and is suited for parallel work

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

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work
  - Implemented via "daemon threads"
  - The Schedulers.parallel() operator in Project Reactor is similar
- The Java common fork-join pool is also similar wrt CPU-bound tasks



commonPool

```
public static ForkJoinPool commonPool()
```

Returns the common pool instance. This pool is statically constructed; its run state is unaffected by attempts to `shutdown()` or `shutdownNow()`. However this pool and any ongoing processing are automatically terminated upon program `System.exit(int)`. Any program that relies on asynchronous task processing to complete before program termination should invoke `commonPool().awaitQuiescence`, before exit.

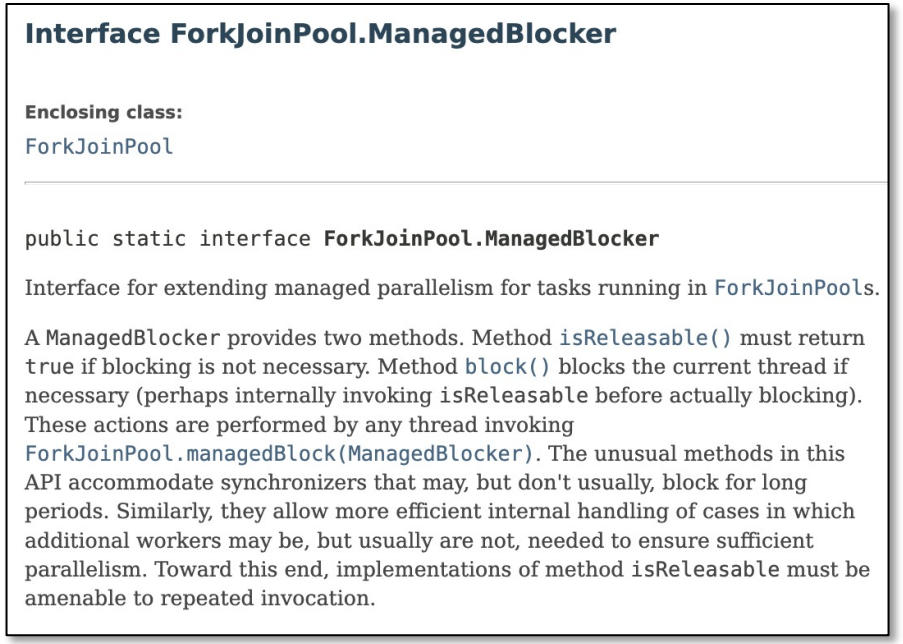
Returns:

the common pool instance

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool

Key Scheduler Operators for the Observable Class

- The Schedulers.computation() operator
  - Hosts a fixed-size pool of single-threaded Executor Service-based workers
  - Used for event-loops, callbacks, & other computational work
  - Implemented via "daemon threads"
  - The Schedulers.parallel() operator in Project Reactor is similar



Interface ForkJoinPool.ManagedBlocker

Enclosing class:

ForkJoinPool

public static interface **ForkJoinPool.ManagedBlocker**

Interface for extending managed parallelism for tasks running in ForkJoinPools.

A ManagedBlocker provides two methods. Method isReleasable() must return true if blocking is not necessary. Method block() blocks the current thread if necessary (perhaps internally invoking isReleasable before actually blocking). These actions are performed by any thread invoking ForkJoinPool.managedBlock(ManagedBlocker). The unusual methods in this API accommodate synchronizers that may, but don't usually, block for long periods. Similarly, they allow more efficient internal handling of cases in which additional workers may be, but usually are not, needed to ensure sufficient parallelism. Toward this end, implementations of method isReleasable must be amenable to repeated invocation.

- The Java common fork-join pool is also similar wrt CPU-bound tasks
  - However, ManagedBlocker enables it to also work with I/O-bound tasks

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html

End of Key Scheduler Operators for the Observable Class (Part 2)

Key Combining Operators in the Observable Class (Part 2)

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 operators defined in—or used with—Observables
  - Factory method operations
  - Transforming operators
  - Concurrency & scheduler operators
  - Error handling operators
  - Combining operators
    - This operator creates a Maybe by accumulating elements in an Observable stream
      - e.g., reduce()

Key Combining Operators in the Observable Class

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items

```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```

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

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param

```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```

Interface BiFunction<T,U,R>

Type Parameters:

T - the type of the first argument to the function

U - the type of the second argument to the function

R - the type of the result of the function

All Known Subinterfaces:

BinaryOperator<T>

Functional Interface:

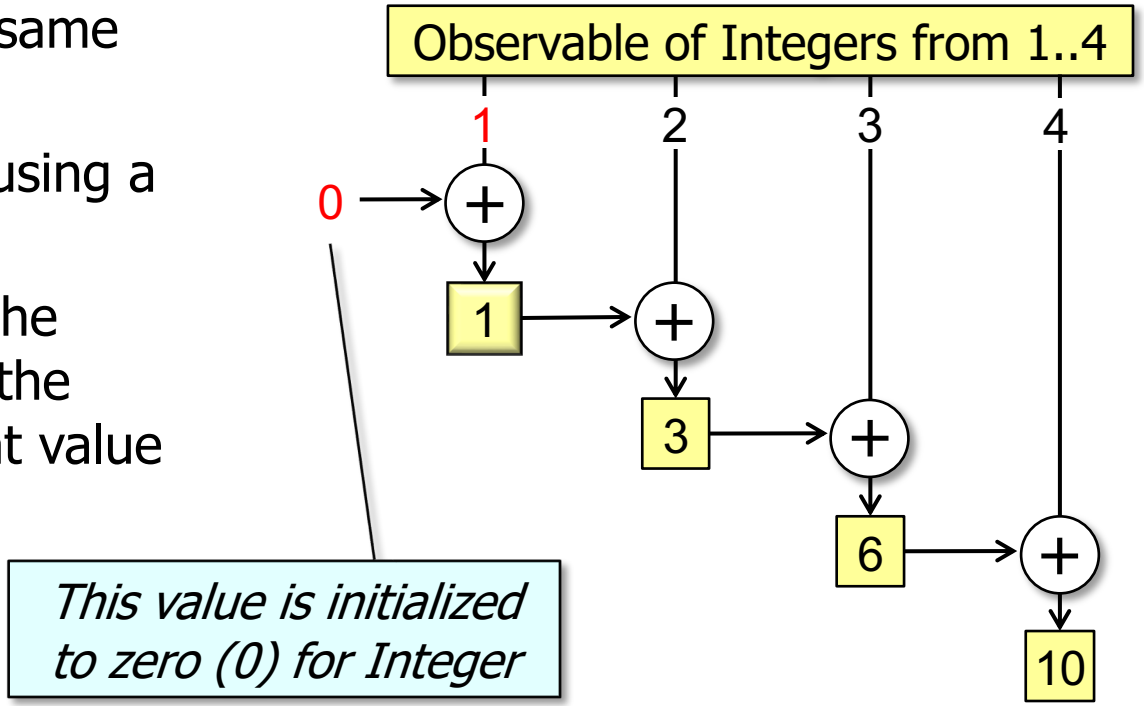
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/functions/BiFunction.html

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value

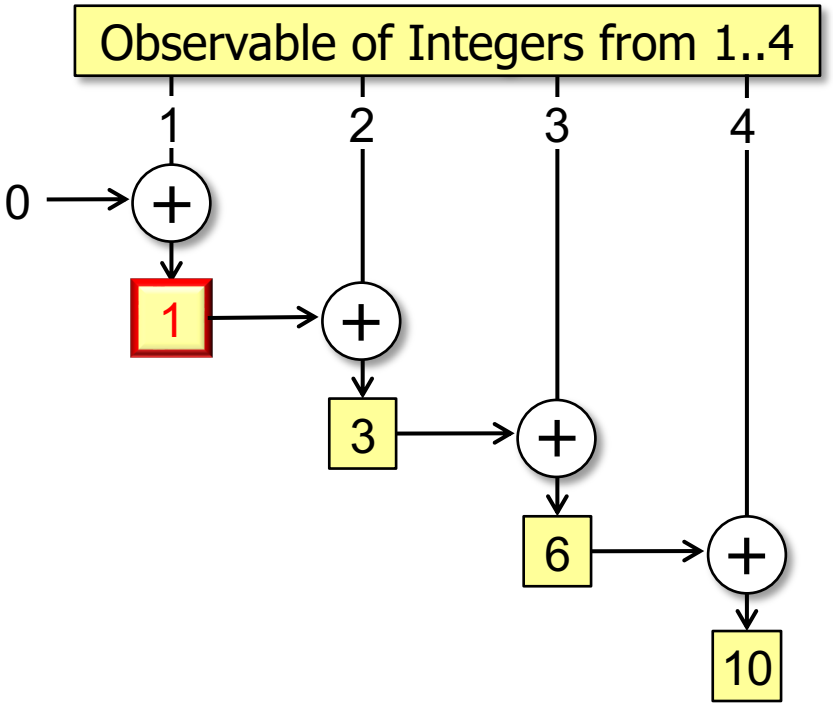
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction

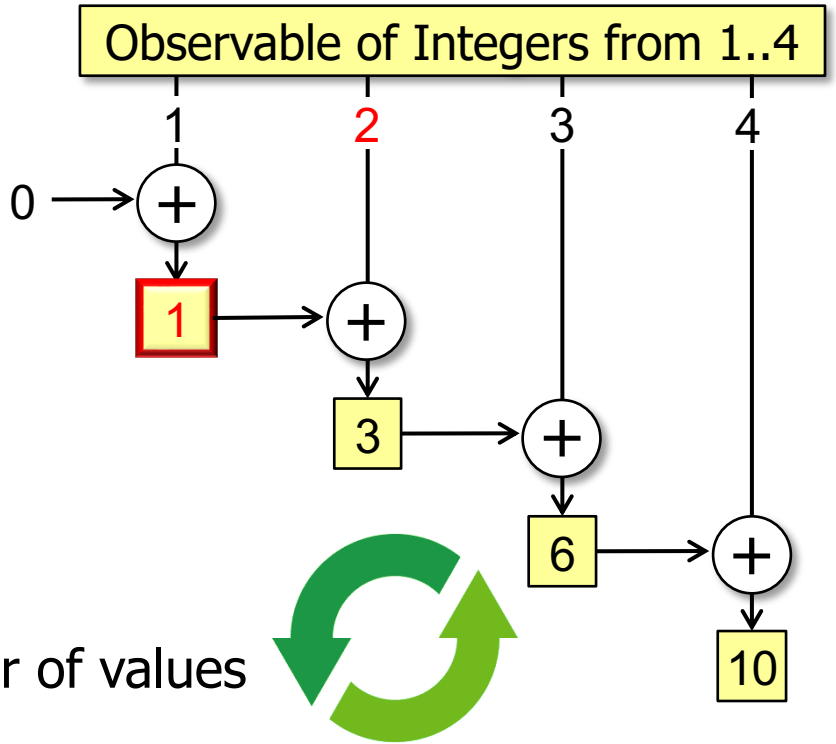
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

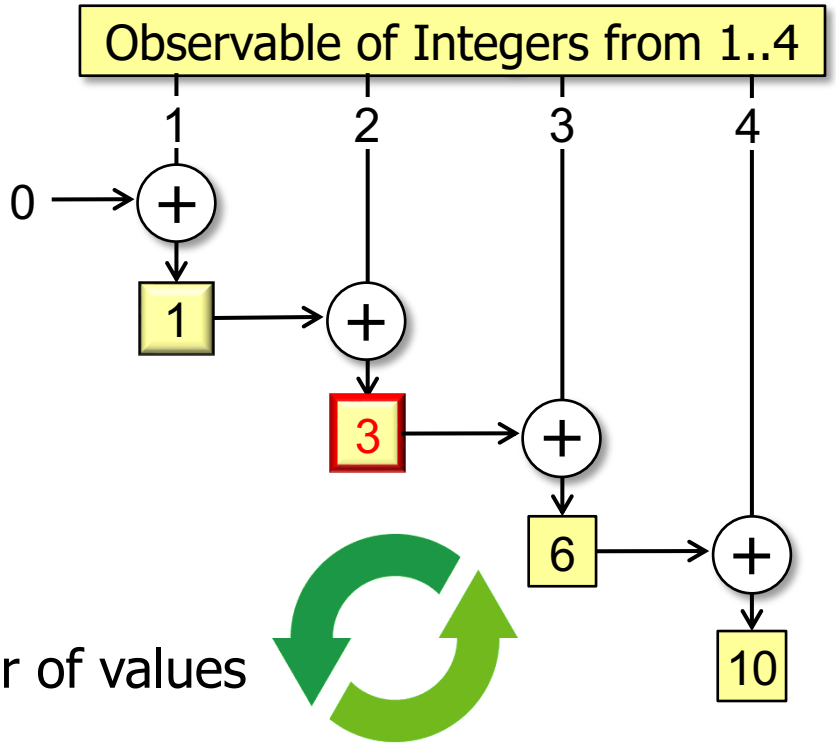
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

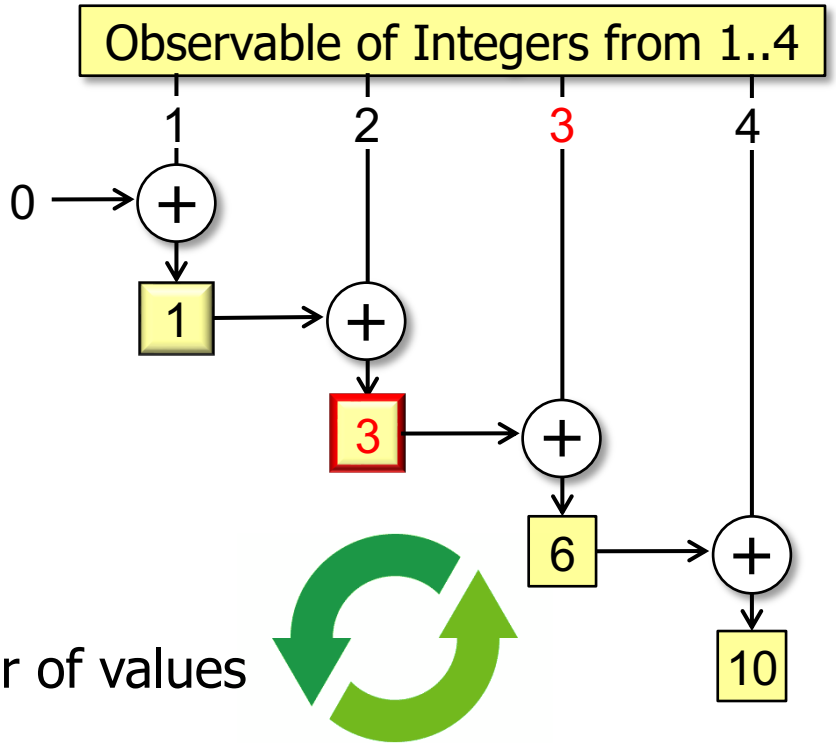
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

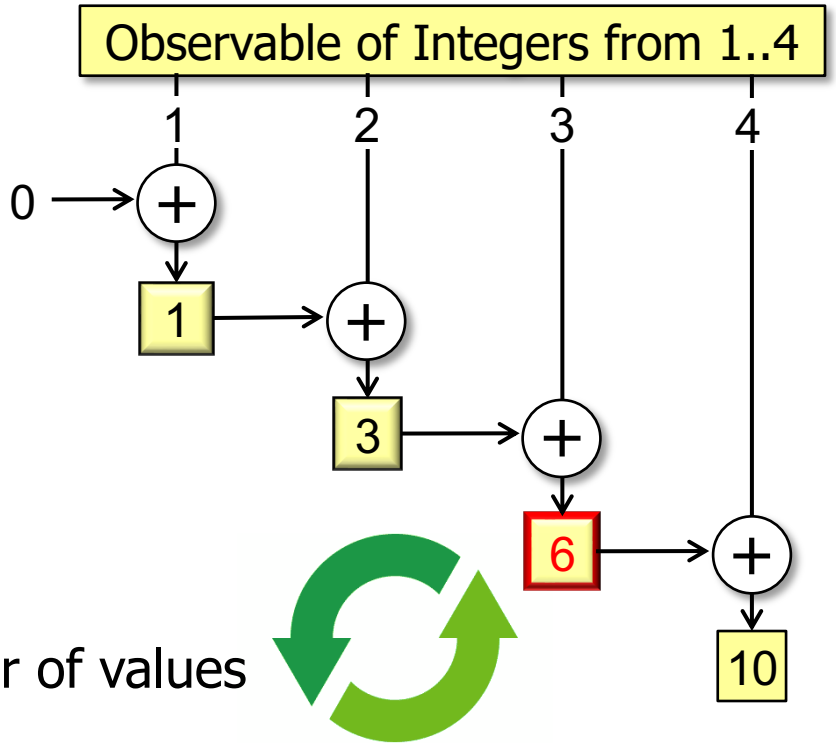
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

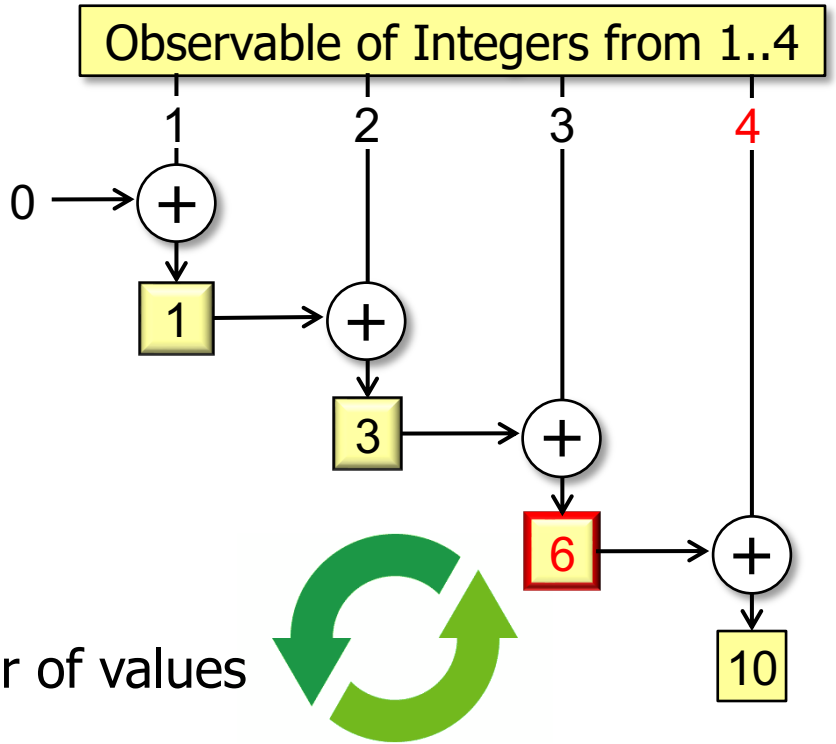
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

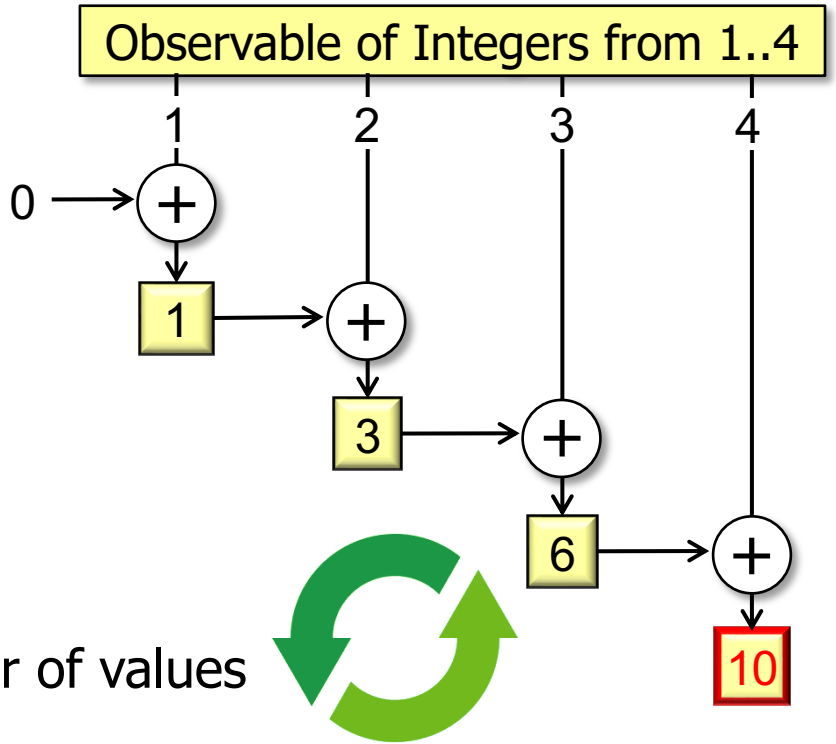
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
      - This param is passed the intermediate result of the reduction & the current value
        - It returns the next intermediate value of the reduction
      - This process repeats for each pair of values

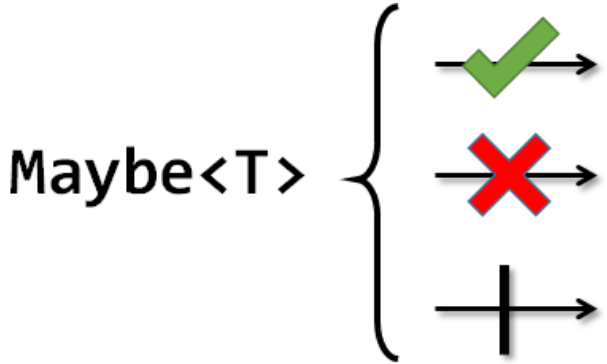
```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
  - The final result is emitted from the final call as the sole item of a Maybe

```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Maybe.html

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
  - The final result is emitted from the final call as the sole item of a Maybe
    - An empty Maybe will be returned if the Observable emits no items

```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```

Key Combining Operators in the Observable Class

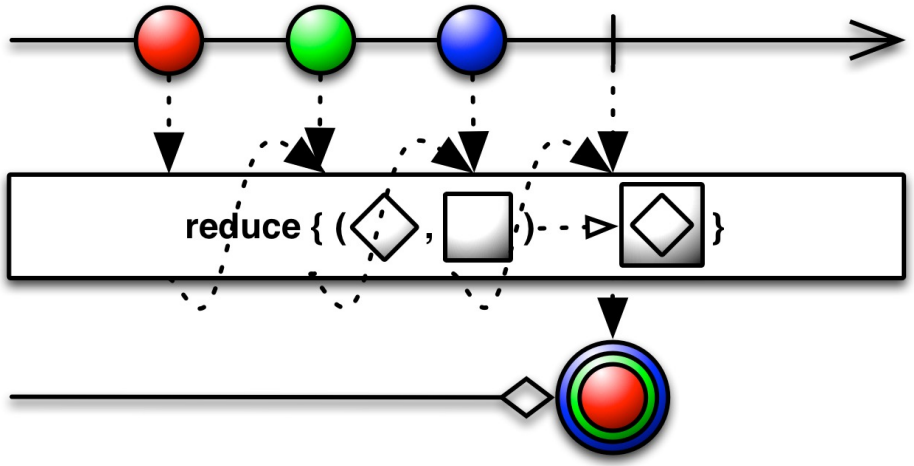
- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
    - Reduction is performed using a BiFunction param
  - The final result is emitted from the final call as the sole item of a Maybe
    - An empty Maybe will be returned if the Observable emits no items

```
Maybe<T> reduce
   (BiFunction<T, T, T> reducer)
```

- The internally accumulated value is discarded upon cancellation or error

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
  - Upstream must signal onComplete() before accumulator can be emitted



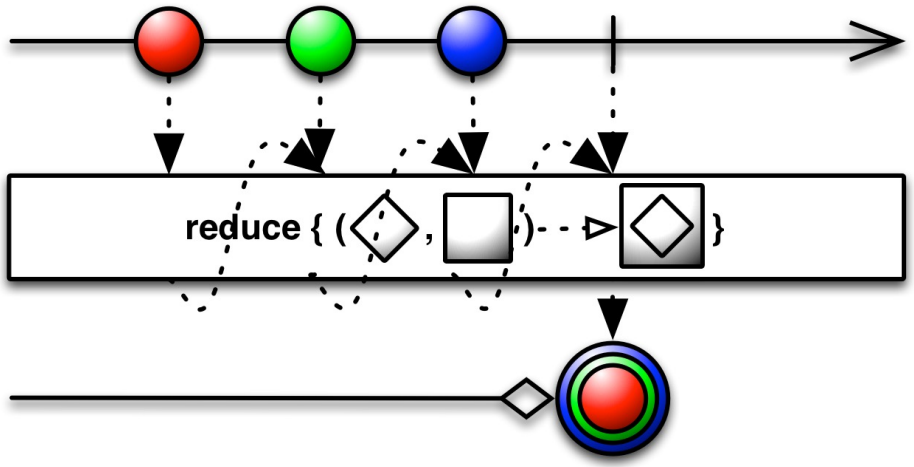

```
return Observable
   .fromArray(bigFractions)
   ...
   .flatMap(bf ->
            multiplyFractions(bf, Schedulers.computation()))
   .reduce(BigFraction::add)
   ...
```

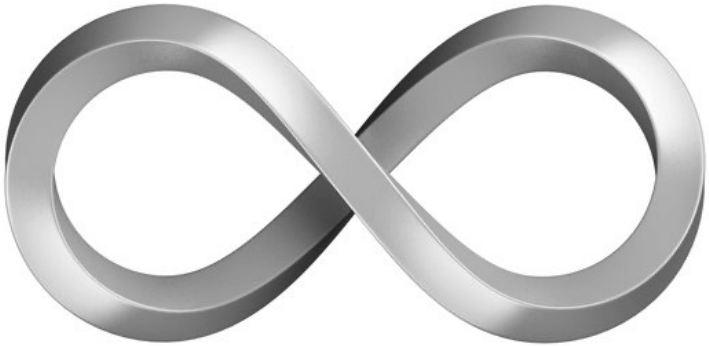
Sum the results of async multiplications

See Reactive/Observable/ex3/src/main/java/ObserableEx.java

Key Combining Operators in the Observable Class

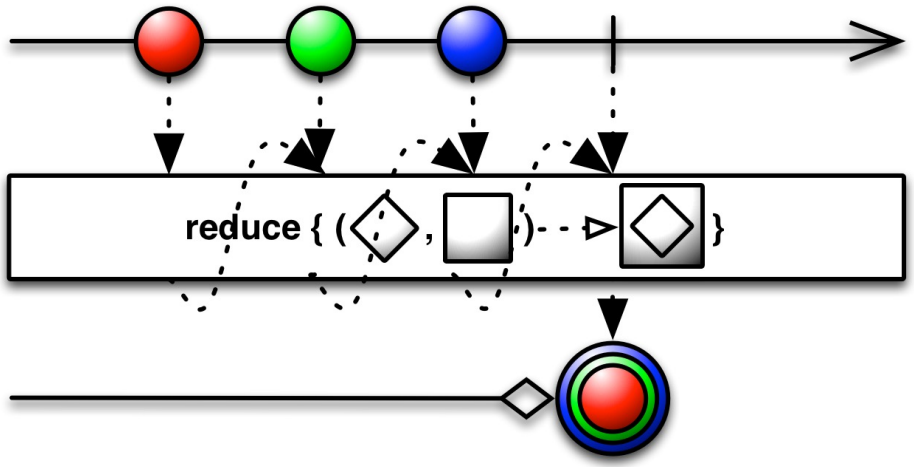
- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
  - Upstream must signal onComplete() before accumulator can be emitted
    - Sources that are infinite & never complete will never emit anything through this operator

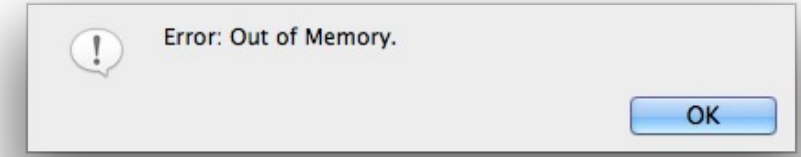




Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
  - Upstream must signal onComplete() before accumulator can be emitted
    - Sources that are infinite & never complete will never emit anything through this operator
      - An infinite source may lead to a fatal OutOfMemoryError

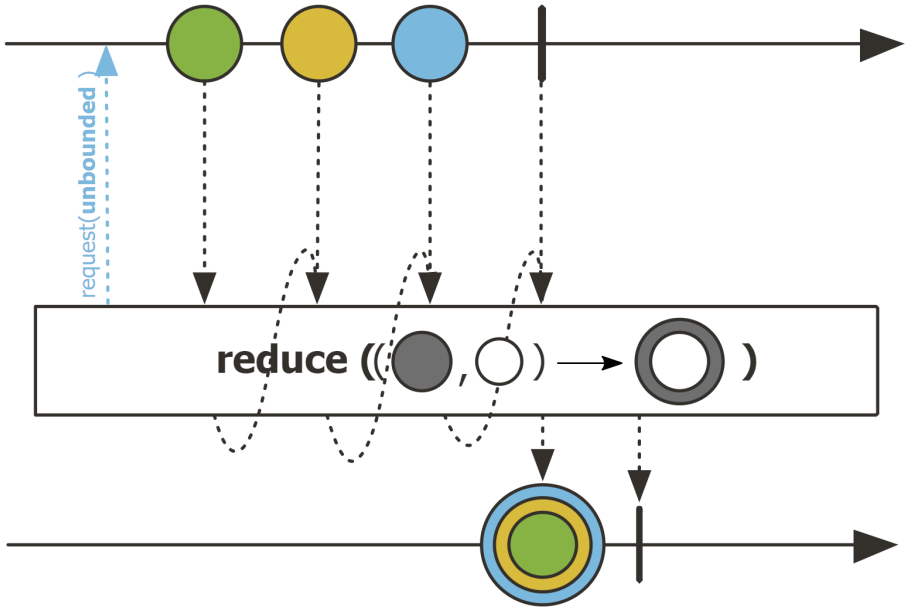




See docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
  - Upstream must signal onComplete() before accumulator can be emitted
  - Project Reactor's Flux.reduce() operator works the same




```
return Flux
    .fromArray(bigFractions)
    .flatMap(bf -> multiplyFractions(bf, Schedulers.parallel()))
    .reduce(BigFraction::add)
    ...
```

Sum results of async multiplications

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

Key Combining Operators in the Observable Class

- The reduce() operator
  - Reduce this Observable's values into a single object of the same type as the emitted items
  - Upstream must signal onComplete() before accumulator can be emitted
  - Project Reactor's Flux.reduce() operator works the same
  - Similar to the Stream.reduce() method in Java Streams

reduce

`Optional<T> reduce(BinaryOperator<T> accumulator)`

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream) {
    if (!foundAny) {
        foundAny = true;
        result = element;
    }
    else
        result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

```
int result = Stream.of(bigFractions)
   .parallel().map(multiplyFractions)
   .reduce(0, Math::addExact);
```

Sum the List values

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce

End of Key Combining Operators in the Observable Class (Part 2)

Key Combining Operators in the Observable Class (Part 3)

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 operators defined in—or used with—Observables
  - Factory method operations
  - Transforming operators
  - Concurrency & scheduler operators
  - Error handling operators
  - Combining operators
    - These operators create a Single by accumulating elements in an Observable stream
      - e.g., reduce(), collectInto(), & collect()

Key Combining Operators in the Observable Class

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure

```
Single<U> collectInto
   (U initialItem,
    BiConsumer<? super U, ? super T>
    collector)
```

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

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
    - The 1st param is the mutable data structure that accumulates (collects) the items

```
Single<U> collectInto
   (U initialItem,
    BiConsumer<? super U, ? super T>
    collector)

...
.collectInto
   (new ArrayList<BigFraction>(),
    List::add)
...
```

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
    - The 1st param is the mutable data structure that accumulates (collects) the items
  - The 2nd param is a BiConsumer that accepts the accumulator & an emitted item
    - The accumulator is modified accordingly

```
Single<U> collectInto
   (U initialItem,
    BiConsumer<? super U, ? super T>
    collector)

...
.collectInto
   (new ArrayList<BigFraction>(),
    List::add)
...
```

Interface BiConsumer<T1,T2>

Type Parameters:

T1 - the first value type

T2 - the second value type

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/functions/BiConsumer.html

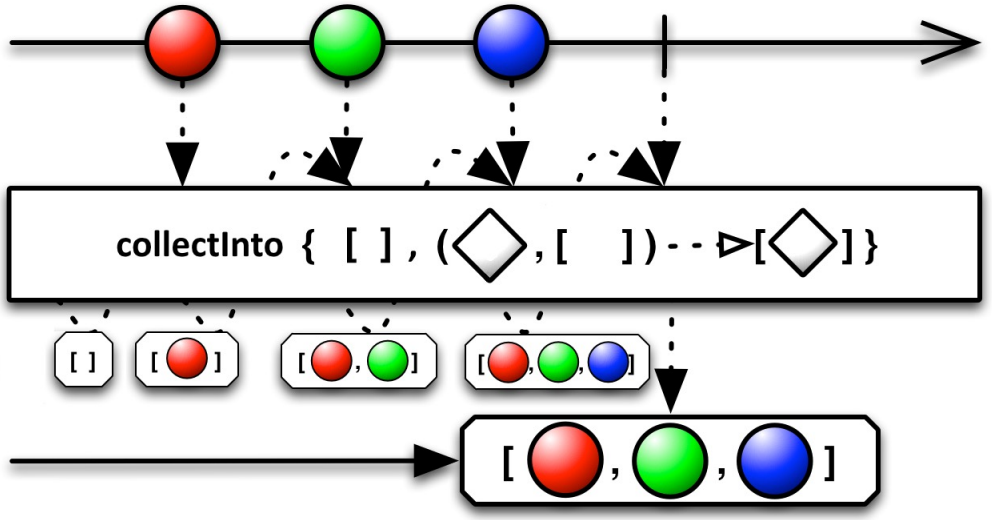
Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
    - The 1st param is the mutable data structure that accumulates (collects) the items
    - The 2nd param is a BiConsumer that accepts the accumulator & an emitted item
  - Returns a Single that emits the mutable data structure

```
Single<U> collectInto
   (U initialItem,
    BiConsumer<? super U, ? super T>
    collector)
```

Key Combining Operators in the Observable Class

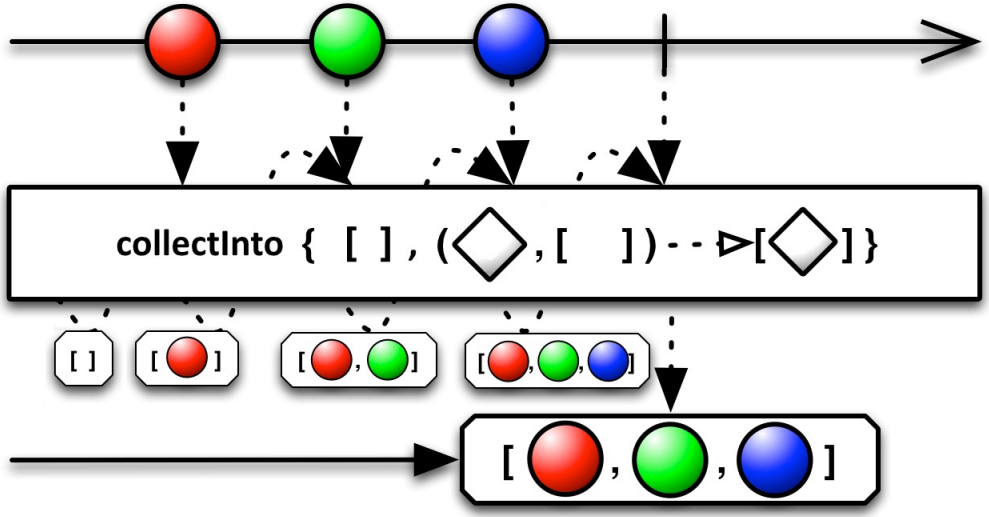
- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
  - This operator is a simplified version of reduce() that does not need to return the state on each pass



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

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
  - This operator is a simplified version of reduce() that does not need to return the state on each pass



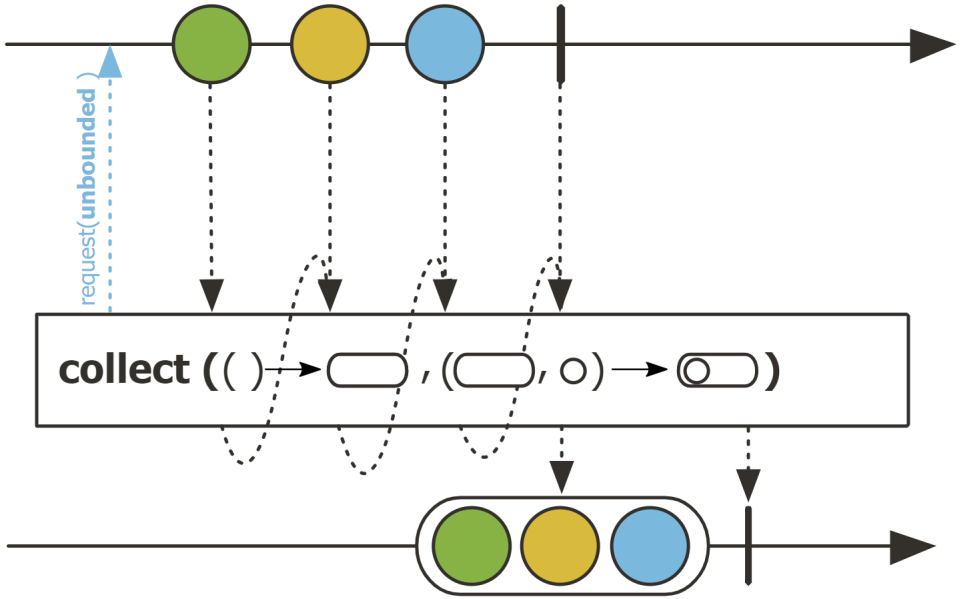

```
Observable
   .fromIterable(bigFractions)
   .flatMap(...)
   .filter(fraction -> fraction.compareTo(0) > 0)
   .collectInto(new ArrayList<BigFraction>(), List::add)
   ...
```

Collect filtered BigFractions into a list

See Reactive/Observable/ex3/src/main/java/ObservableEx.java

Key Combining Operators in the Observable Class

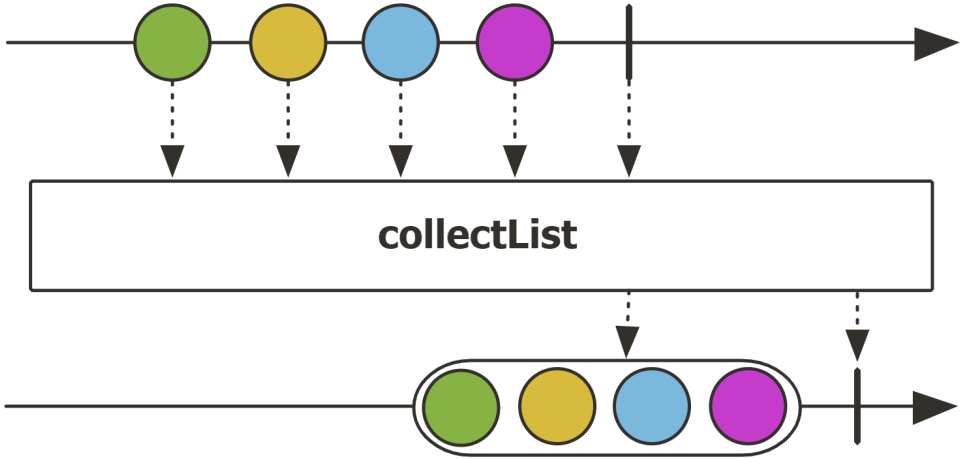
- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
  - Project Reactor's Flux.collect() operator works the same way



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

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
  - Project Reactor's Flux.collect() operator works the same way
    - Flux.collectList() is a more concise (albeit more limited) option



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

Key Combining Operators in the Observable Class

- The collectInto() operator
  - Collects items emitted by the finite source Observable into a single mutable data structure
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
  - Project Reactor's Flux.collect() operator works the same
  - Similar to the Stream.collect() method in Java Streams

collect

`<R,A> R collect(Collector<? super T,A,R> collector)`

Performs a mutable reduction operation on the elements of this stream using a `Collector`. A `Collector` encapsulates the functions used as arguments to `collect(Supplier, BiConsumer, BiConsumer)`, allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

```
List<Integer> evenNumbers = List
   .of(1, 2, 3, 4, 5, 6)
   .stream()
   .filter(x -> x % 2 == 0)
   .toList();
```

Collect even #'d Integers into a List

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

Key Combining Operators in the Observable Class

- The collect() operator
  - Collects the finite upstream's values into a container

```
<R, A> Single<U> collect
  (Collector<? super T,
             A,
             R> collector)
```

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

Key Combining Operators in the Observable Class

- The collect() operator
  - Collects the finite upstream's values into a container
    - The param is the Java Stream Collector interface defining the container supplier, accumulator, & finisher functions

```
<R, A> Single<U> collect
  (Collector<? super T,
             A,
             R> collector)
```

Interface Collector<T,A,R>

Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

---

public interface **Collector<T,A,R>**

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Key Combining Operators in the Observable Class

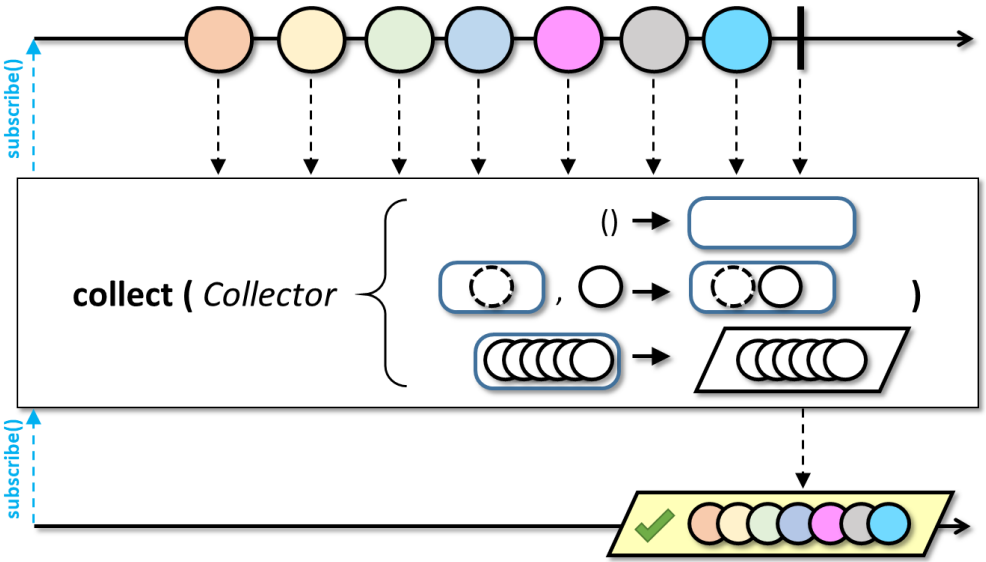
- The collect() operator
  - Collects the finite upstream's values into a container
    - The param is the Java Stream Collector interface defining the container supplier, accumulator, & finisher functions
  - Returns a Single that emits the container

```
<R, A> Single<U> collect
  (Collector<? super T,
             A,
             R> collector)
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Single.html

Key Combining Operators in the Observable Class

- The collect() operator
  - Collects the finite upstream's values into a container
  - This operator is a simplified version of reduce() that does not need to return the state on each pass



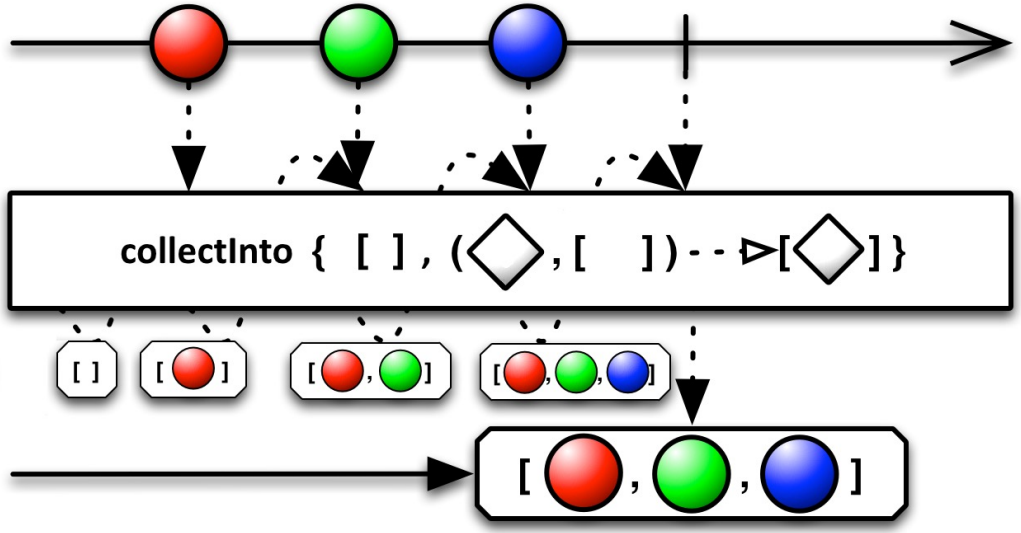

```
Observable
   .generate(emitter)
   .take(sMAX_FRACTIONS)
   .flatMap(...)
   .collect(toList())
   .flatMapCompletable(...);
```

Collect all the processed BigFractions into a List

See Reactive/Observable/ex3/src/main/java/ObservableEx.java

Key Combining Operators in the Observable Class

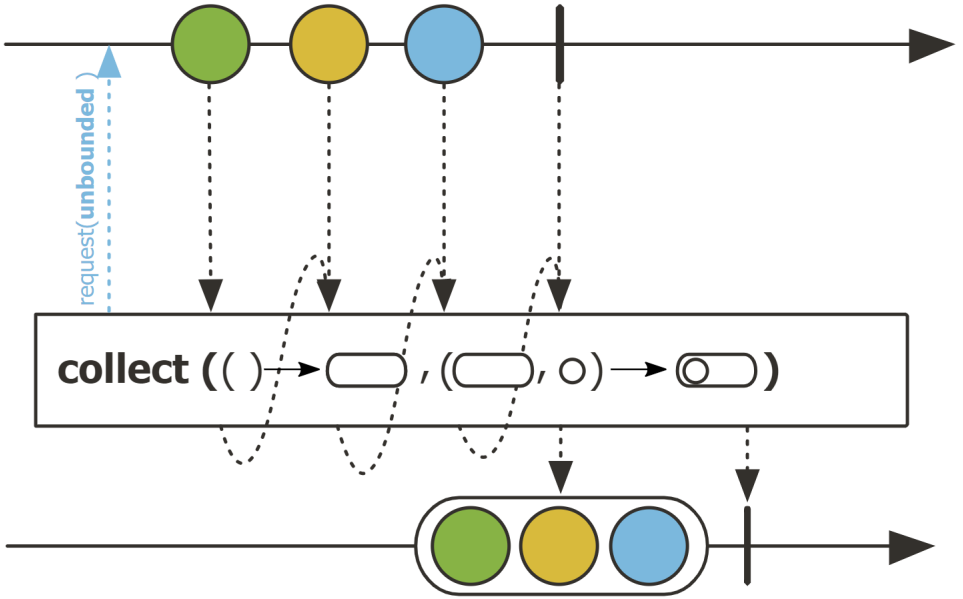
- The collect() operator
  - Collects the finite upstream's values into a container
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
    - It's also similar to operator Observable.collectInto()



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

Key Combining Operators in the Observable Class

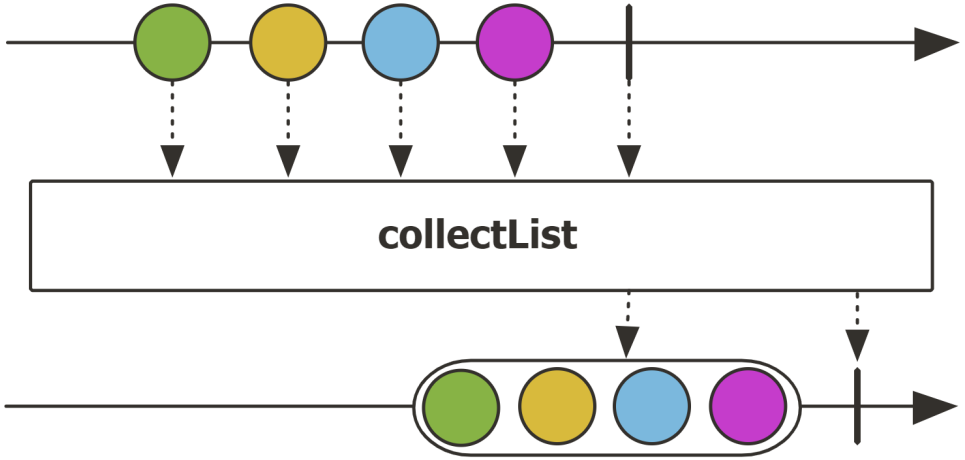
- The collect() operator
  - Collects the finite upstream's values into a container
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
  - Project Reactor's Flux.collect() operator works the same way



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

Key Combining Operators in the Observable Class

- The collect() operator
  - Collects the finite upstream's values into a container
  - This operator is a simplified version of reduce() that does not need to return the state on each pass



- Project Reactor's Flux.collect() operator works the same way
  - Flux.collectList() is a more concise (albeit more limited) option

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

Key Combining Operators in the Observable Class

- The collect() operator
  - Collects the finite upstream's values into a container
  - This operator is a simplified version of reduce() that does not need to return the state on each pass
  - Project Reactor's Flux.collect() operator works the same
- Similar to the Stream.collect() method in Java Streams

collect

`<R,A> R collect(Collector<? super T,A,R> collector)`

Performs a mutable reduction operation on the elements of this stream using a `Collector`. A `Collector` encapsulates the functions used as arguments to `collect(Supplier, BiConsumer, BiConsumer)`, allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

```
Set<Integer> evenNumbers = List
   .of(1, 2, 2, 3, 4, 4, 5, 6, 6)
   .stream()
   .filter(x -> x % 2 == 0)
   .collect(toSet());
```

Collect even #'d Integers into a Set

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

End of Key Combining Operators in the Observable Class (Part 3)