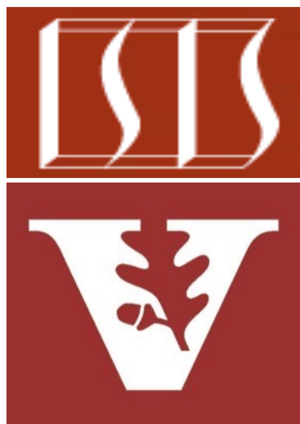
Key Factory Method Operators in the Observable Class (Part 1)

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

Professor of Computer Science

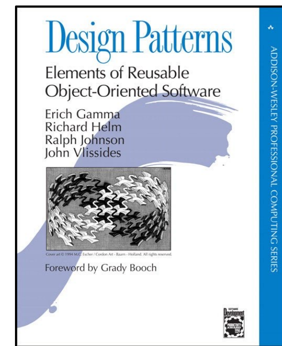
Institute for Software Integrated Systems

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
    - These operators create reactive Observable streams in various ways from non-reactive input sources
      - e.g., just() & fromCallable()



See en.wikipedia.org/wiki/Factory_method_pattern

Key Factory Method Operators in the Observable Class

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes

```
static <T> Observable<T>
           just(T... data)
```

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

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
    - The param(s) are the elements to emit, as a varargs param

```
static <T> Observable<T>
           just(T... data)
```

See www.baeldung.com/java-varargs

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
    - The param(s) are the elements to emit, as a varargs param
  - Returns a new Observable that's captured at "assembly time"
    - i.e., it's "eager"

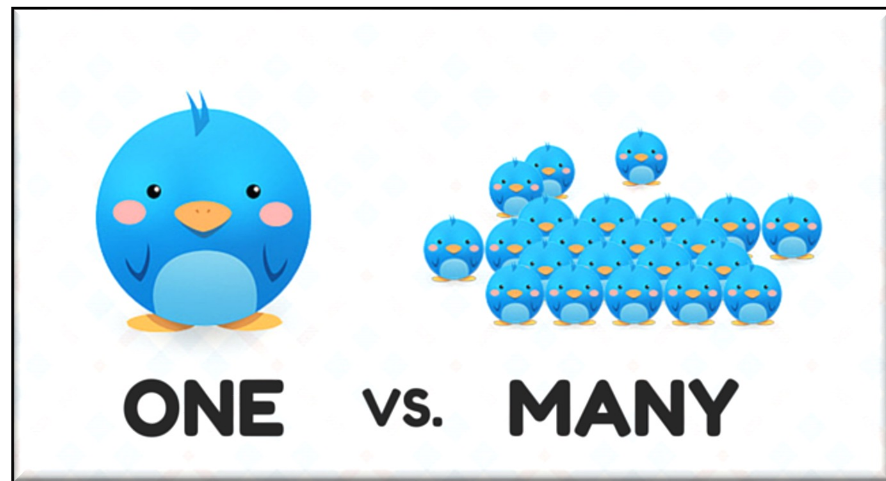
```
static <T> Observable<T>
           just(T... data)
```

Contrast with the discussion of the Observable.fromCallable() operator later in this lesson

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
    - The param(s) are the elements to emit, as a varargs param
    - Returns a new Observable that's captured at "assembly time"
    - Multiple elements can be emitted, unlike the Single.just() operator

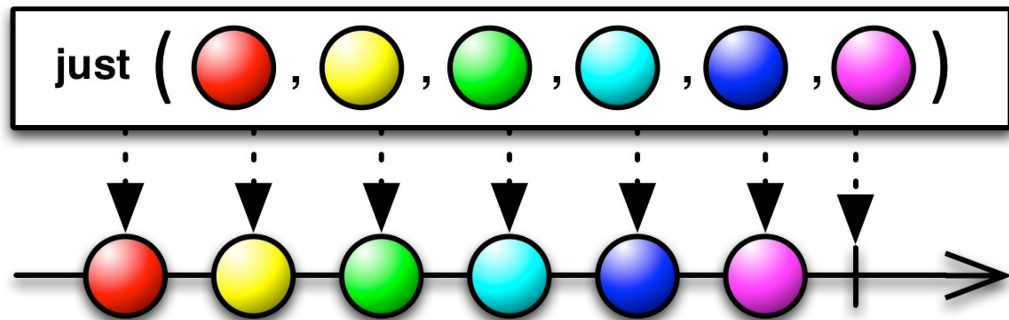
```
static <T> Observable<T>
           just(T... data)
```



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

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model



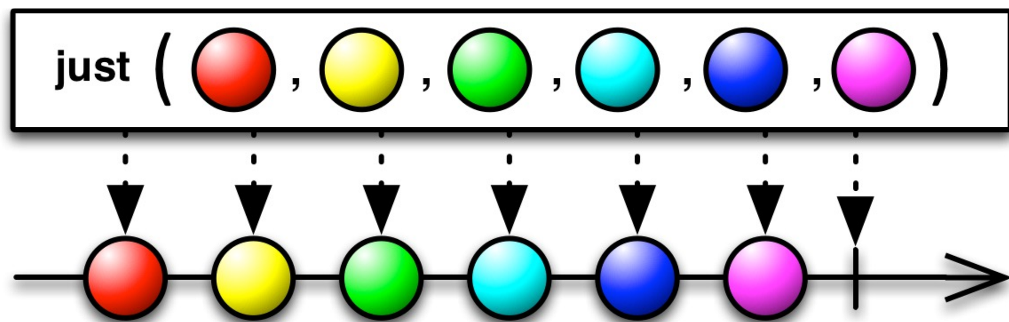

```
Observable
  .just(BigFraction.valueOf(100,3),
        BigFraction.valueOf(100,4),
        BigFraction.valueOf(100,2),
        BigFraction.valueOf(100,1))
  ...
```

Create an Observable stream of four BigFraction objects

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

Key Factory Method Operators in the Observable Class

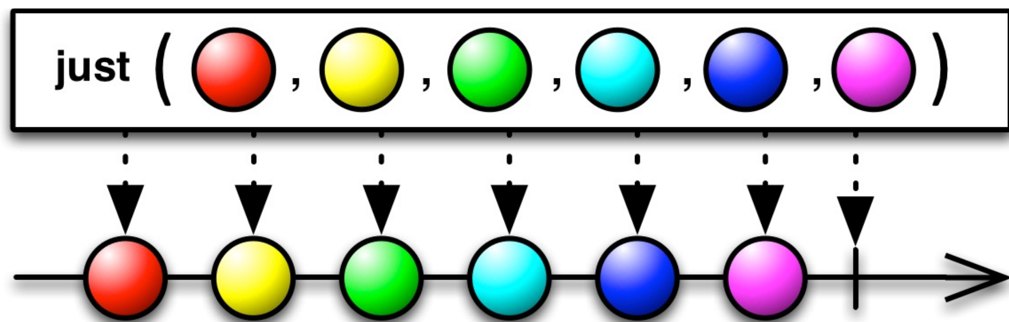
- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model
    - just() is evaluated eagerly at "assembly time"



See proandroiddev.com/operator-fusion-in-rxjava-2-dcd6612cffae

Key Factory Method Operators in the Observable Class

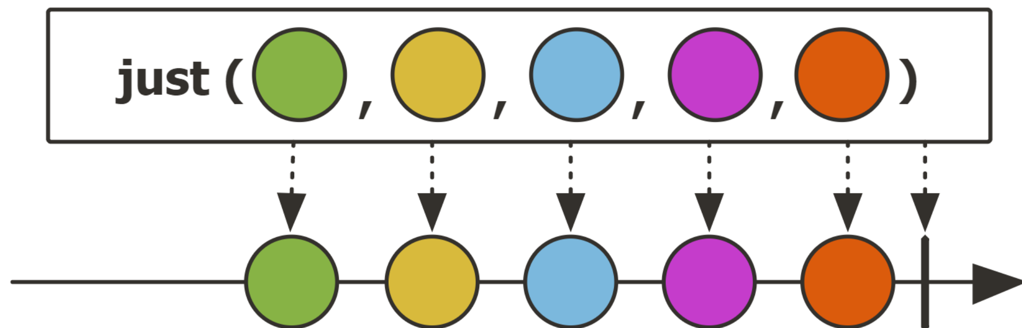
- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model
    - just() is evaluated eagerly at "assembly time"
    - It therefore always runs in the context of the thread where the Observable is instantiated



The fromIterable() & fromArray() factory method operators also evaluate eagerly

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model
  - Project Reactor's Flux.just() operator works the same




```
Flux
  .just(BigFraction.valueOf(100,3),
        BigFraction.valueOf(100,4),
        BigFraction.valueOf(100,2),
        BigFraction.valueOf(100,1))
  ...
```

Create a Flux stream of four BigFraction objects

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

Key Factory Method Operators in the Observable Class

- The just() operator
  - Creates an Observable that emits the given element(s) & then completes
  - This factory method adapts non-reactive input sources into the reactive model
  - Project Reactor's Flux.just() operator works the same
  - Similar to Stream.of() factory method in Java Streams

of

```
@SafeVarargs
static <T> Stream<T> of(T... values)
```

Returns a sequential ordered stream whose elements are the specified values.

Type Parameters:

T - the type of stream elements

Parameters:

values - the elements of the new stream

Returns:

the new stream

```
Stream
  .of(BigFraction.valueOf(100,3),
      BigFraction.valueOf(100,4),
      BigFraction.valueOf(100,2),
      BigFraction.valueOf(100,1))
  ...
```

Create a stream of 4 BigFraction objects

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

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things

```
static <T> Observable<T>
  fromCallable(Callable<? extends T>
                              callable)
```

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

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
    - Invokes a Callable param

```
static <T> Observable<T>
  fromCallable(Callable<? extends T>
                         callable)
```

Interface Callable<V>

Type Parameters:

V - the result type of method call

All Known Subinterfaces:

DocumentationTool.DocumentationTask, JavaCompiler.CompilationTask

Functional Interface:

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

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

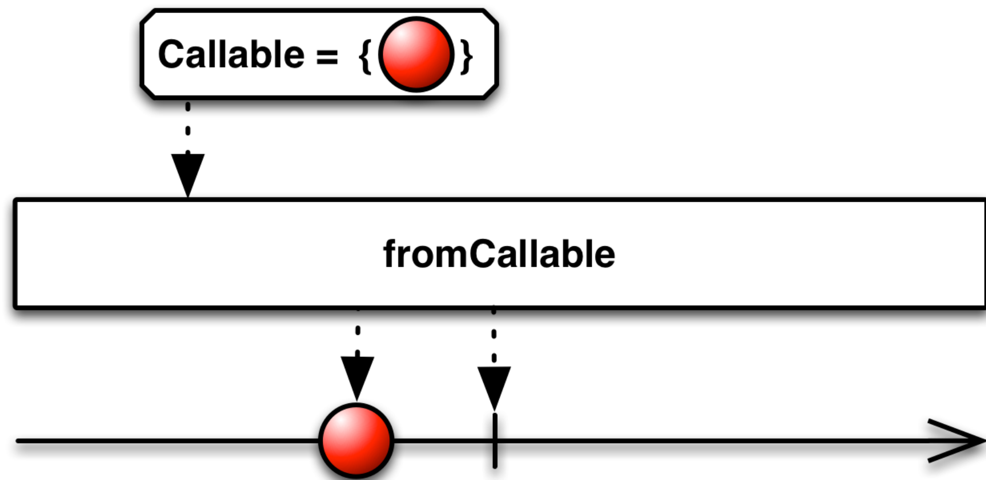
Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
    - Invokes a Callable param
    - The returned Observable emits the value returned from the Callable

```
static <T> Observable<T>
  fromCallable(Callable<? extends T>
                              callable)
```

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
  - This factory method adapts non-reactive input sources into the reactive model




```
Observable
  .fromCallable
    (()
     -> BigFractionUtils
       .makeBigFraction(random,
                        true))
```

Create an Observable that emits one random BigFraction

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

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
  - This factory method adapts non-reactive input sources into the reactive model
  - This operator defers executing the Callable until an observer subscribes to the Observable
    - i.e., it is "lazy"

```
Observable
  .fromCallable
    (()
     -> BigFractionUtils
         .makeBigFraction(random,
                          true))
```

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
  - This factory method adapts non-reactive input sources into the reactive model
  - This operator defers executing the Callable until an observer subscribes to the Observable
    - i.e., it is "lazy"

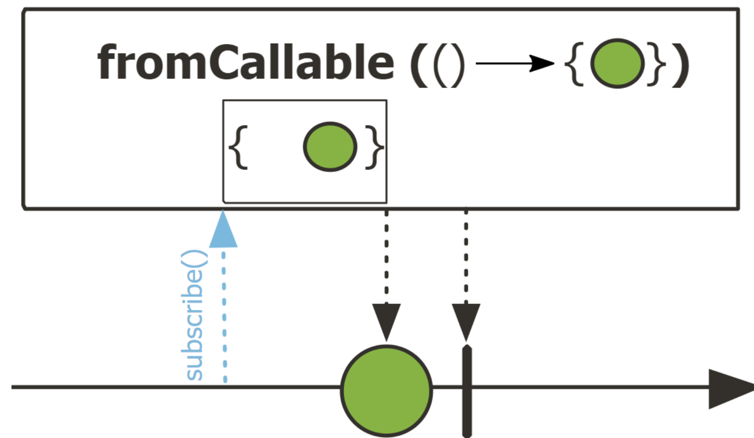
Conversely, Observable.just() is "eager"

```
Observable
   .just(BigFraction.valueOf(100,3),
         BigFraction.valueOf(100,4),
         BigFraction.valueOf(100,2),
         BigFraction.valueOf(100,1))
   ...
```

Contrast with "eager" Observable factory method operators earlier in this lesson

Key Factory Method Operators in the Observable Class

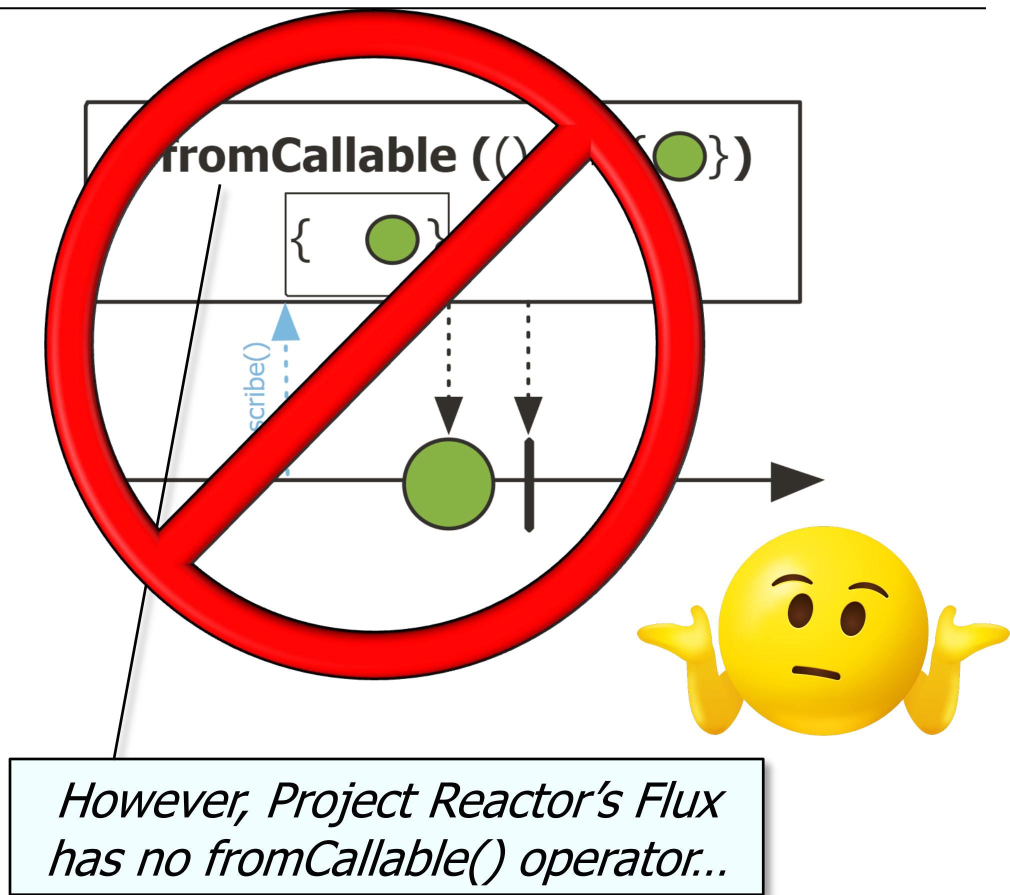
- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
  - This factory method adapts non-reactive input sources into the reactive model
  - This operator defers executing the Callable until an observer subscribes to the Observable
  - Project Reactor's operator Mono .fromCallable() is similar



See projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#fromCallable

Key Factory Method Operators in the Observable Class

- The fromCallable() operator
  - Returns an Observable that, when an observer subscribes to it, does certain things
  - This factory method adapts non-reactive input sources into the reactive model
  - This operator defers executing the Callable until an observer subscribes to the Observable
  - Project Reactor's operator Mono .fromCallable() is similar



However, Project Reactor's Flux has no fromCallable() operator...

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

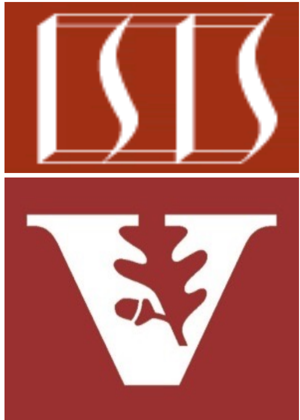
Key Transforming Operators in the Observable Class (Part 1)

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
    - Transform the values and/ or types emitted by an Observable
      - e.g., map()

Key Transforming Operators in the Observable Class

Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable

```
<V> Observable<V> map
  (Function<? super T,? extends V>
   mapper)
```

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

Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable
    - Applies a synchronous function to transform each item

```
<V> Observable<V> map
  (Function<? super T,? extends V>
   mapper)
```

Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

All Known Subinterfaces:

UnaryOperator<T>

Functional Interface:

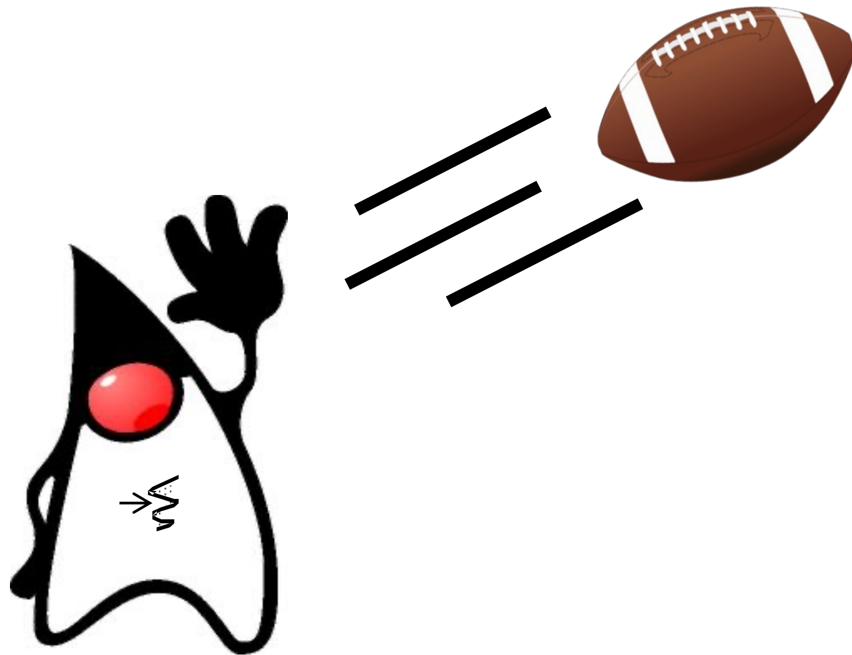
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

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

Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable
    - Applies a synchronous function to transform each item
      - map() can terminate if mapper throws an exception

```
<V> Observable<V> map
  (Function<? super T,? extends V>
   mapper)
```



Key Transforming Operators in the Observable Class

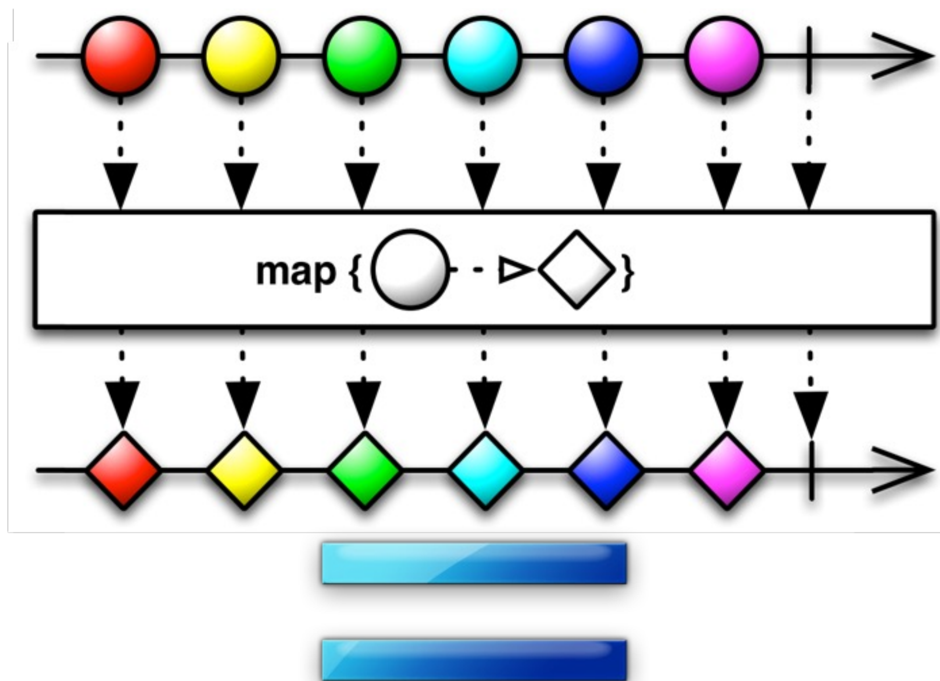
- The map() operator
  - Transform the item(s) emitted by this Observable
    - Applies a synchronous function to transform each item
    - Returns a transformed Observable

```
<V> Observable<V> map
  (Function<? super T,? extends V>
   mapper)
```

Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable
  - The # of output items must match the # of input items

```
Observable
   .fromIterable
     (bigFractionList)
   ...
   .map(fraction -> fraction
          .multiply(sBigReducedFrac))
   ...
```

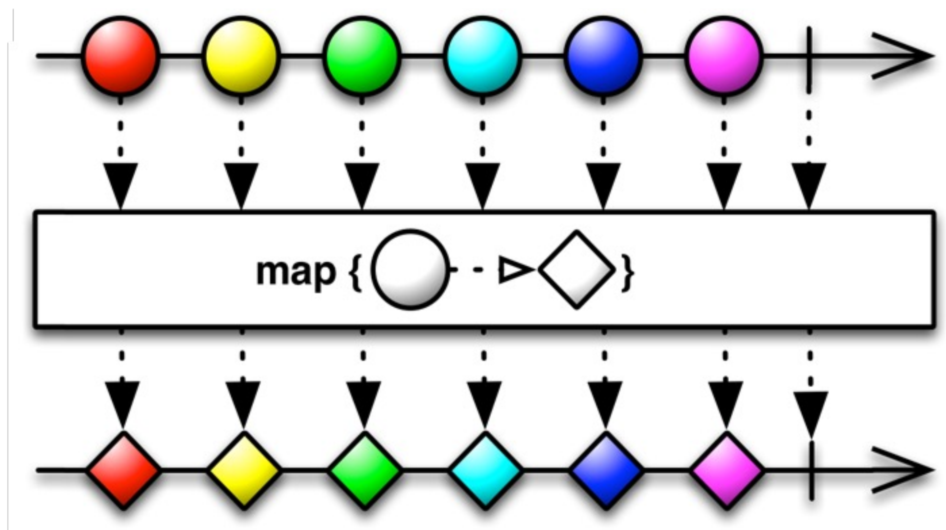


Multiply each element in the Observable stream by a constant

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

Key Transforming Operators in the Observable Class

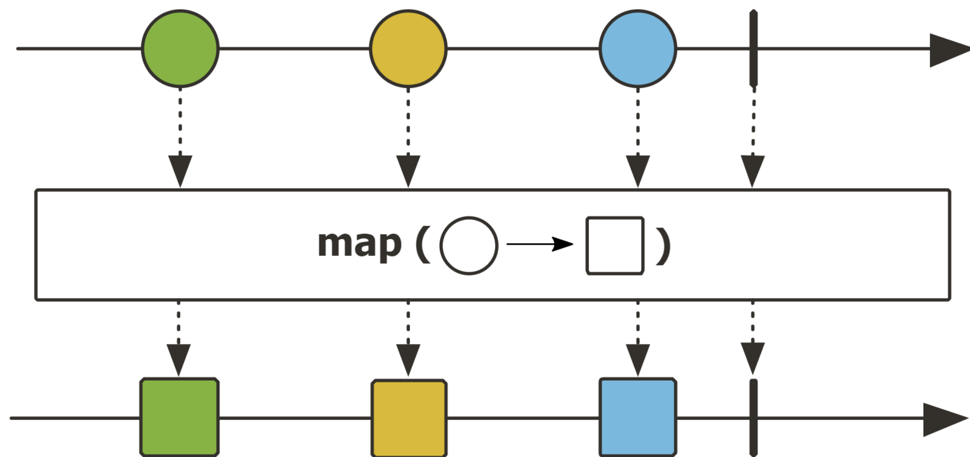
- The map() operator
  - Transform the item(s) emitted by this Observable
  - The # of output items must match the # of input items
    - map() can transform the type and/or value of elements it processes



Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable
  - The # of output items must match the # of input items
  - Project Reactor's Flux.map() operator works the same

```
Flux
   .fromIterable
     (bigFractionList)
   ...
   .map(fraction -> fraction
          .multiply(sBigReducedFrac))
   ...
```



Multiply each element in the Flux stream by a constant

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

Key Transforming Operators in the Observable Class

- The map() operator
  - Transform the item(s) emitted by this Observable
  - The # of output items must match the # of input items
  - Project Reactor's Flux.map() operator works the same
  - Similar to Stream.map() method in Java Streams

map

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an intermediate operation.

Type Parameters:

R - The element type of the new stream

Parameters:

mapper - a non-interfering, stateless function to apply to each element

```
List<String> collect = List
  .of("a", "b", "c").stream()
  .map(String::toUpperCase).toList();
```

Uppercase each string in a stream

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

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

Key Combining 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
    - These operators create an Observable from multiple iterations or sources
      - e.g., mergeWith()

Key Combining Operators in the Observable Class

Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param

```
Observable<T> mergeWith
  (ObservableSource<? extends T>
     other)
```

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

Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
    - The param is the Observable Source to merge with

```
Observable<T> mergeWith
   (ObservableSource<? extends T>
      other)
```

```
@FunctionalInterface
public interface ObservableSource<T>
```

Represents a basic, non-backpressured `Observable` source base interface, consumable via an `Observer`.

Since:
2.0

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
`void`	`subscribe(@NonNull Observer<? super T> observer)` Subscribes the given `Observer` to this `ObservableSource` instance.

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

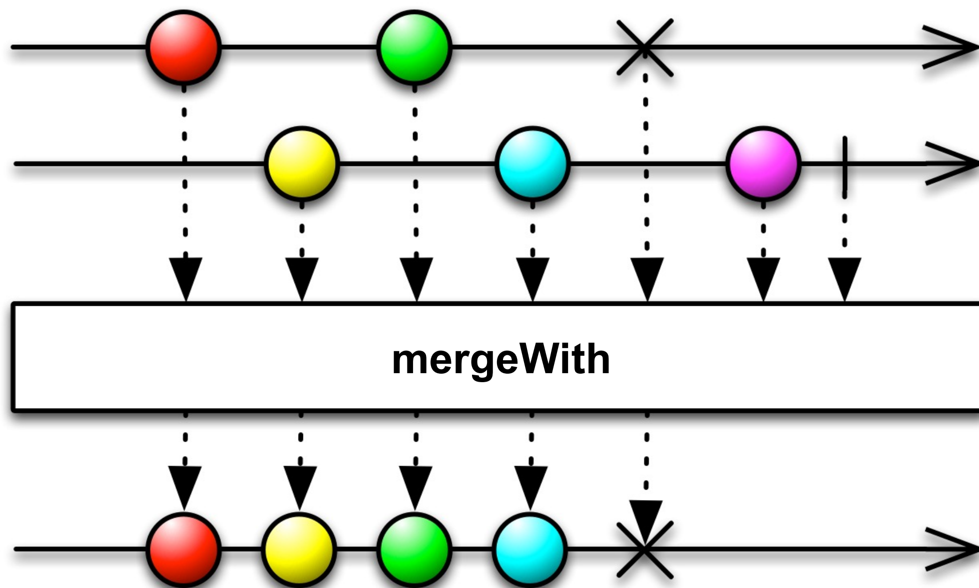
Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
    - The param is the Observable Source to merge with
    - Returns the new merged Observable instance

```
Observable<T> mergeWith
  (ObservableSource<? extends T>
     other)
```

Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
  - This operator combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource

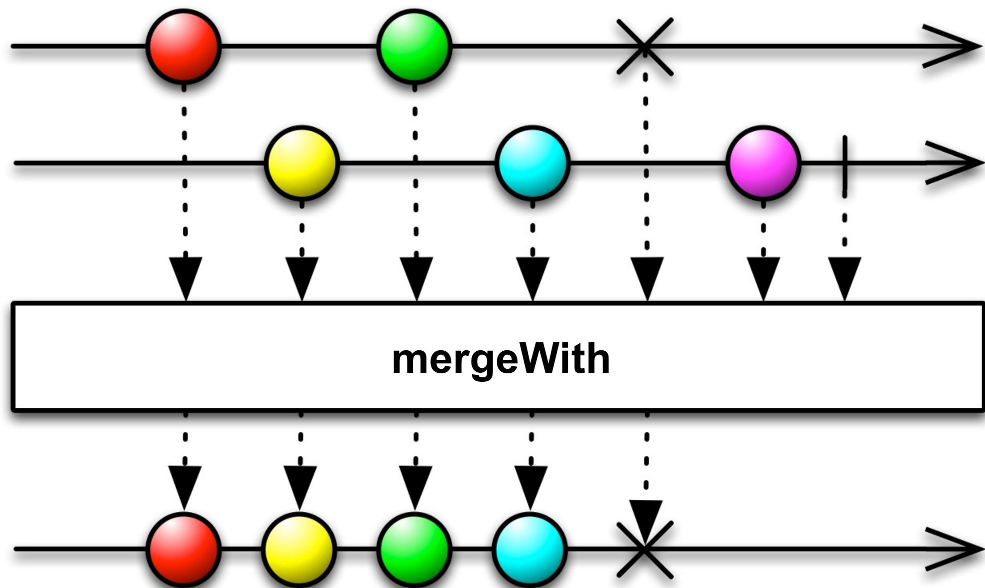



```
Observable<BigFraction> o1 ...
Observable<BigFraction> o2 ...
o1.mergeWith(o2)...
```

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

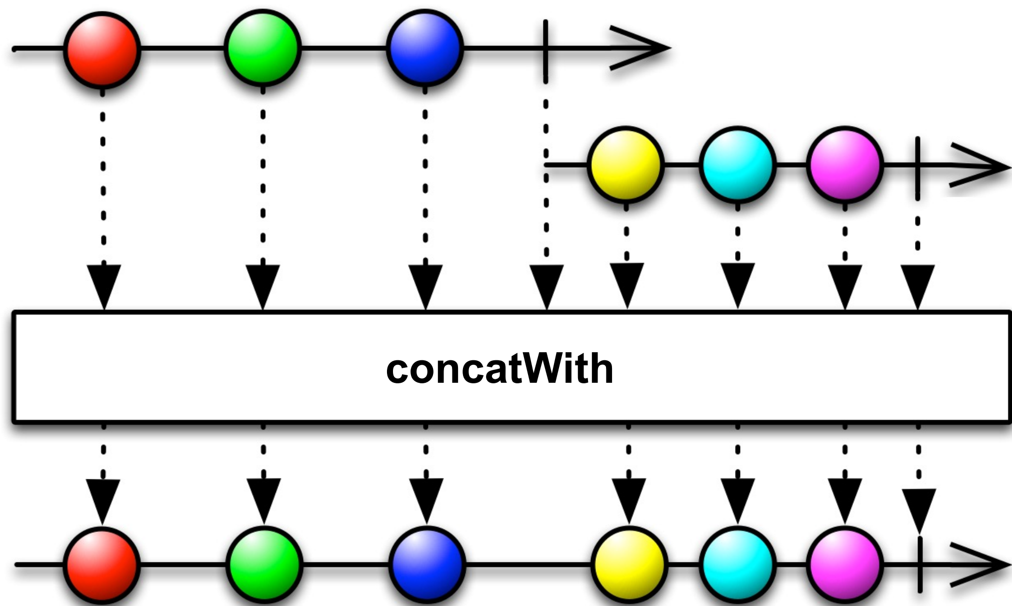
Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
  - This operator combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
    - This merging may interleave the items



Key Combining Operators in the Observable Class

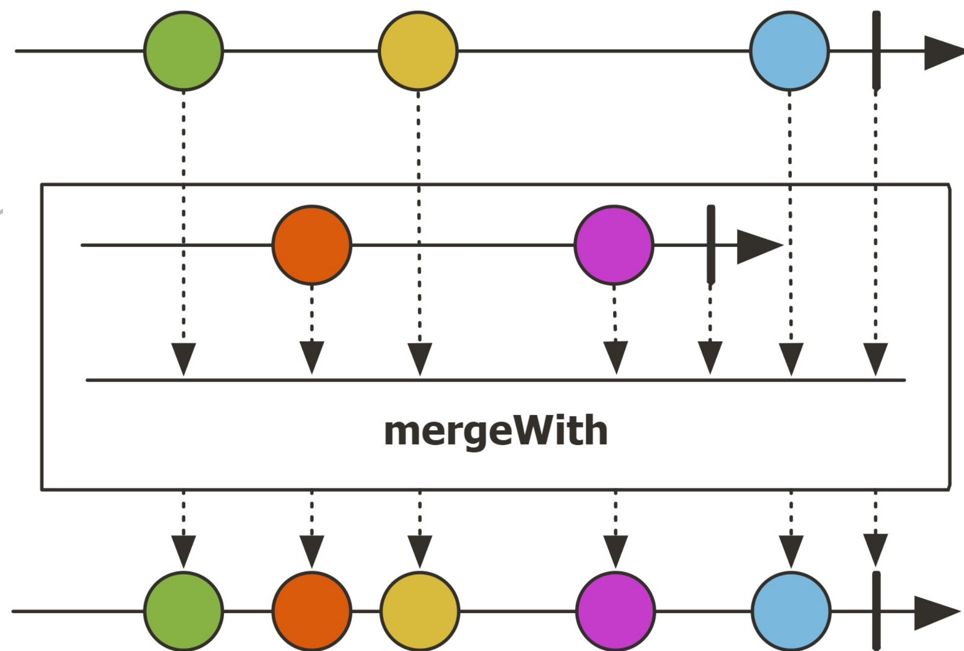
- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
  - This operator combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
    - This merging may interleave the items
    - Use concatWith() to avoid interleaving



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

Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
  - This operator combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
  - Project Reactor's operator Flux.mergeWith() works the same




```
Flux<BigFraction> f1 ...
Flux<BigFraction> f2 ...
f1.mergeWith(f2)...
```

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

Key Combining Operators in the Observable Class

- The mergeWith() operator
  - Merges the sequence of items of this Observable with the success value of the other param
  - This operator combines items emitted by multiple Observable Sources so that they appear as a single ObservableSource
  - Project Reactor's operator Flux.mergeWith() works the same
  - Similar to the Stream.concat() method in Java Streams

concat

```
static <T> Stream<T> concat(Stream<? extends T> a,
                            Stream<? extends T> b)
```

Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.

```
List<String> concats
          (List<String> l, int n) {
  Stream<String> s = Stream.empty();
  while (--n >= 0)
    s = Stream.concat(s, l.stream());
  return s.toList();
}
```

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

End of Key Combining Operators in 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., filter()

Key Suppressing Operators in the Observable Class

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate

```
Observable<T> filter
  (Predicate<? super T> p)
```

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

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
    - If predicate test succeeds, the value is emitted

```
Observable<T> filter
   (Predicate<? super T> p)
```

Interface Predicate<T>

Type Parameters:

T - the type of the input to the predicate

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/Predicate.html

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
    - If predicate test succeeds, the value is emitted
    - If predicate test fails, the value is ignored & a request of 1 is made upstream

```
Observable<T> filter
   (Predicate<? super T> p)
```

Interface Predicate<T>

Type Parameters:

T - the type of the input to the predicate

Functional Interface:

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

Key Suppressing Operators in the Observable Class

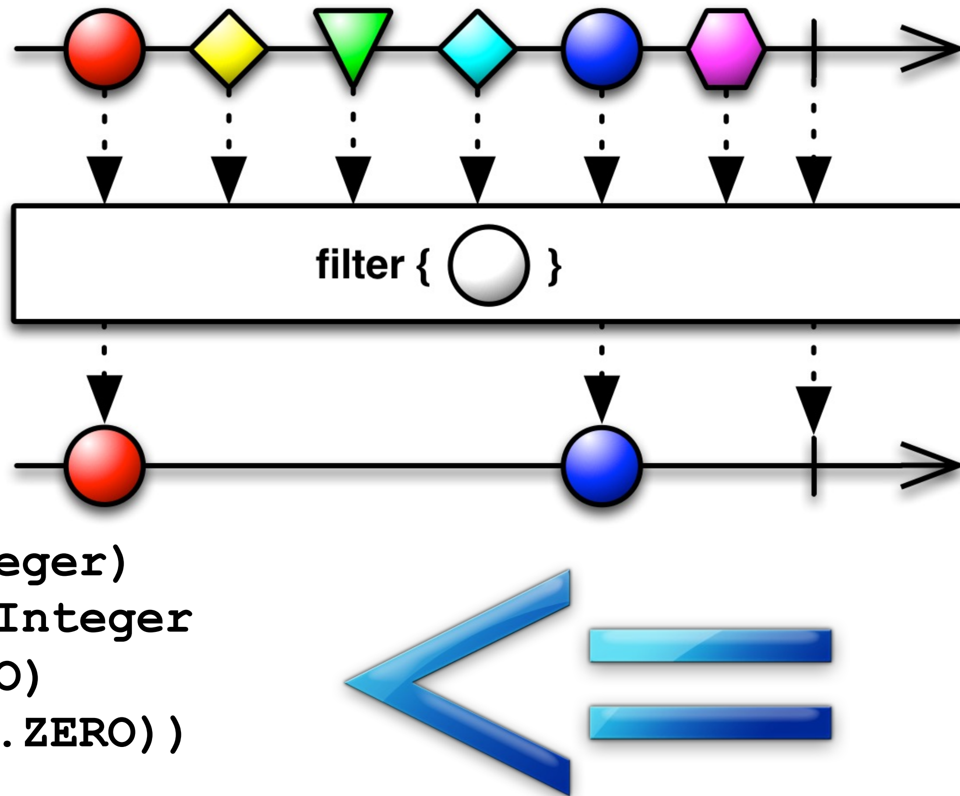
- The filter() operator
  - Evaluate each source value against the given Predicate
    - If predicate test succeeds, the value is emitted
    - If predicate test fails, the value is ignored & a request of 1 is made upstream
    - Returns a new Observable containing only values that pass the predicate test

```
Observable<T> filter
  (Predicate<? super T> p)
```

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
  - The # of output elements may be < than # of input elements

```
Observable
   .rangeLong(1, sMAX_ITERS)
   ...
   .map(sGenerateRandomBigInteger)
   .filter(bigInteger -> !bigInteger
           .mod(BigInteger.TWO)
           .equals(BigInteger.ZERO))

   .subscribe(...);
```



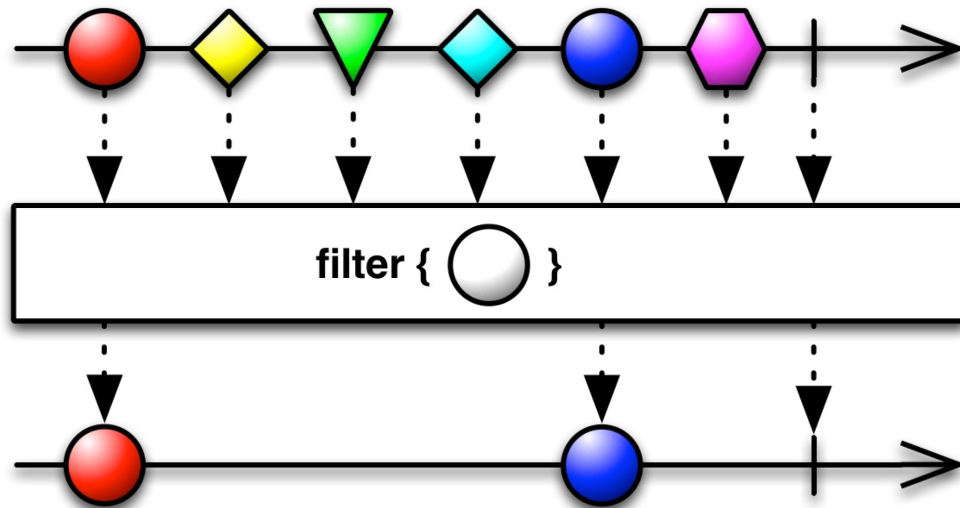
See Reactive/Observable/ex2/src/main/java/ObservableEx.java

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
  - The # of output elements may be < than # of input elements

```
Observable
   .rangeLong(1, sMAX_ITERS)
   ...
   .map(sGenerateRandomBigInteger)
   .filter(bigInteger -> !bigInteger
                  .mod(BigInteger.TWO)
                  .equals(BigInteger.ZERO))

   .subscribe(...);
```

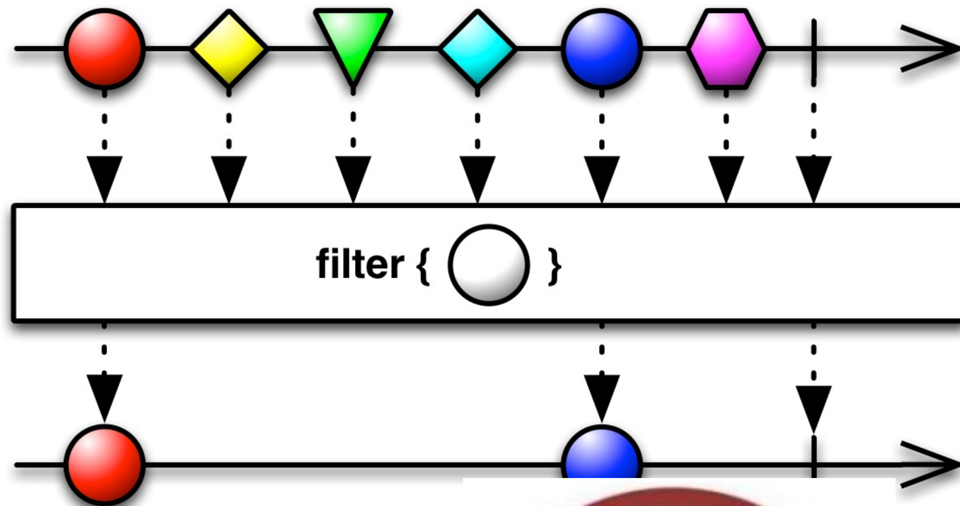
Only emit odd numbers



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

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
  - The # of output elements may be < than # of input elements



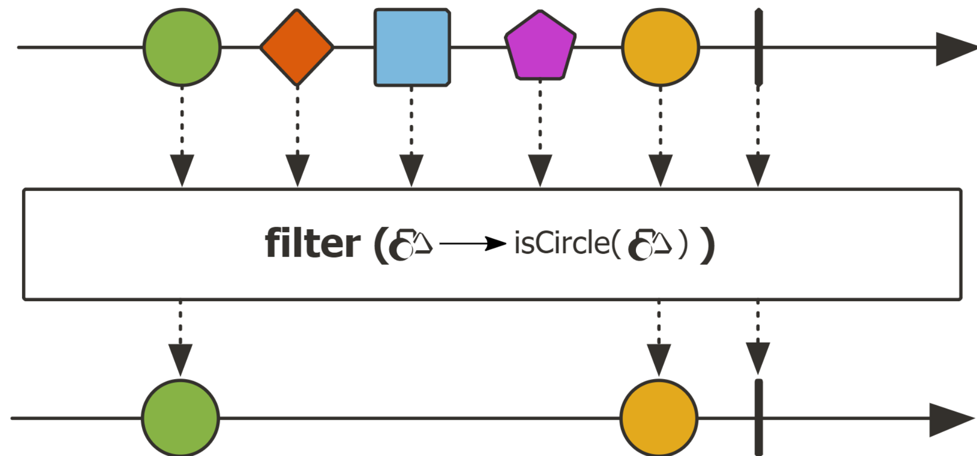

```
Observable
   .rangeLong(1, sMAX_ITERS)
   ...
   .map(sGenerateRandomBigInteger)
   .filter(bigInteger -> !bigInteger
               .mod(BigInteger.TWO)
               .equals(BigInteger.ZERO))

   .subscribe(...);
```

filter() can't change the type or value of elements it processes

Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
  - The # of output elements may be < than # of input elements
  - Project Reactor's Flux.filter() operator works the same way

```
Flux
   .range(1, sMAX_ITERATIONS)
   ...
   .map(sGenerateRandomBigInteger)
   .filter(bigInteger -> !bigInteger.mod(BigInteger.TWO)
                                    .equals(BigInteger.ZERO))
   .subscribe(...);
```



Key Suppressing Operators in the Observable Class

- The filter() operator
  - Evaluate each source value against the given Predicate
  - The # of output elements may be < than # of input elements
  - Project Reactor's Flux.filter() operator works the same way
  - Similar to Stream.filter() method in Java Streams

filter

```
Stream<T> filter(Predicate<? super T> predicate)
```

Returns a stream consisting of the elements of this stream that match the given predicate.

This is an intermediate operation.

Parameters:

predicate - a non-interfering, stateless predicate to apply to each element to determine if it should be included

Returns:

the new stream

```
List<Long> oddNumbers =
  LongStream
    .rangeClosed(1, 100)
    .filter(n -> (n & 1) != 0)
    .toList();
```

Only emit odd #'s

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

End of Key Suppressing Operators in the Observable Class

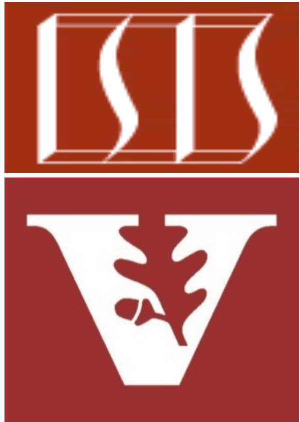
Key Transforming Operators in the Observable Class (Part 2)

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., flatMap()

Key Transforming Operators in the Observable Class

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously

```
<R> Observable<R> flatMap
  (Function
     <? super T,
      ? extends ObservableSource
                   <? extends R>>
   mapper)
```

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

Key Transforming Operators in the Observable Class

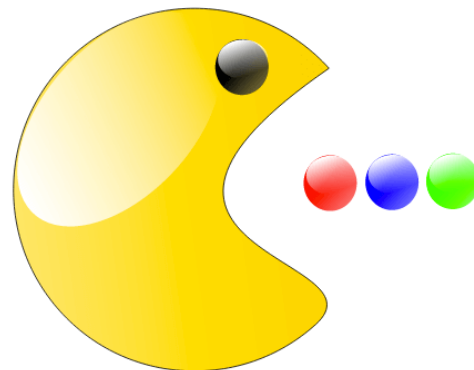
- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable

```
<R> Observable<R> flatMap
  (Function
     <? super T,
      ? extends ObservableSource
                       <? extends R>>
   mapper)
```

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable
    - That function returns an ObservableSource
      - An ObservableSource can be consumed by an Observable

```
<R> Observable<R> flatMap
  (Function
     <? super T,
      ? extends ObservableSource
                    <? extends R>>
   mapper)
```



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

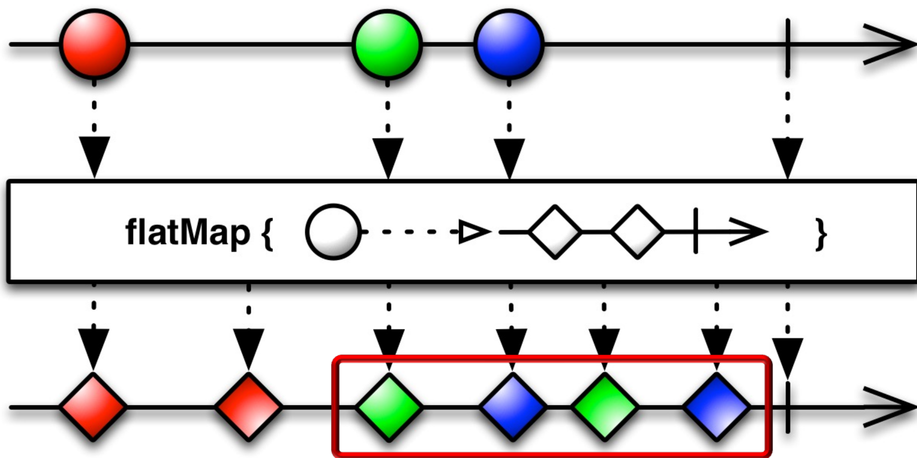
Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable
    - That function returns an ObservableSource
    - The returned ObservableSources are merged & the results of this merger are "flattened" & emitted

```
<R> Observable<R> flatMap
  (Function
     <? super T,
      ? extends ObservableSource
                   <? extends R>>
 mapper)
```

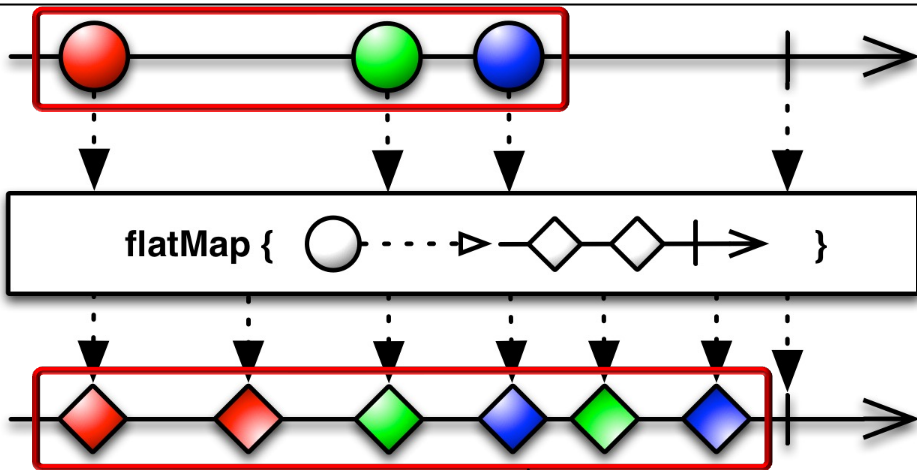
Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable
    - That function returns an ObservableSource
    - The returned ObservableSources are merged & the results of this merger are "flattened" & emitted
      - They thus can interleave



Key Transforming Operators in the Observable Class

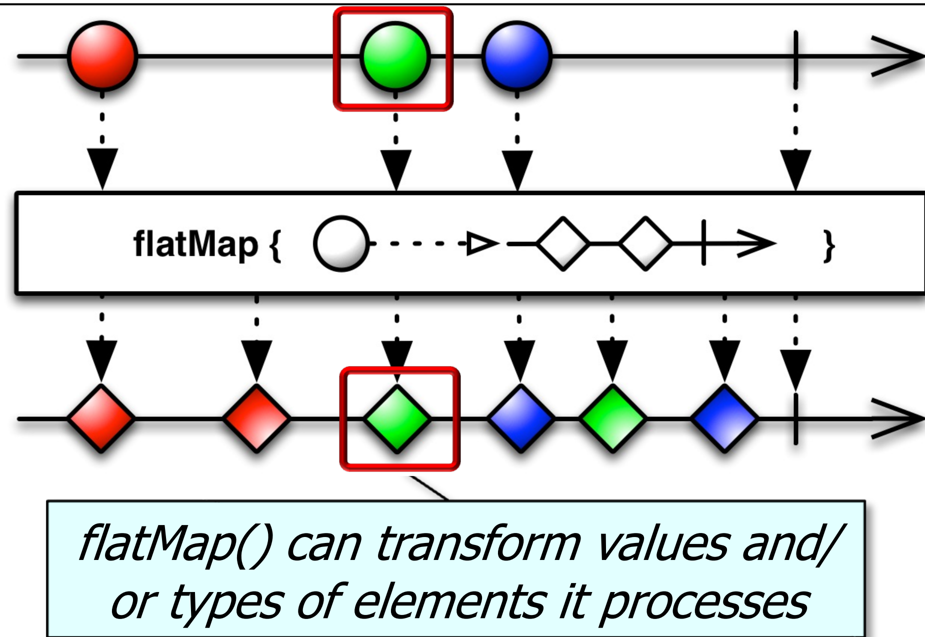
- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable
    - That function returns an ObservableSource
    - The returned ObservableSources are merged & the results of this merger are "flattened" & emitted
      - They thus can interleave



The # of output elements may differ from the # of input elements

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
    - Items are emitted based on applying a function to each item emitted by this Observable
    - That function returns an ObservableSource
    - The returned ObservableSources are merged & the results of this merger are "flattened" & emitted
      - They thus can interleave



Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing

```
return Observable
    .fromIterable(bigFractionList)

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

       .subscribeOn
          (Schedulers
           .computation()))

    .reduce(BigFraction::add)
```

See next part of the lesson on the RxJava flatMap() concurrency idiom

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing

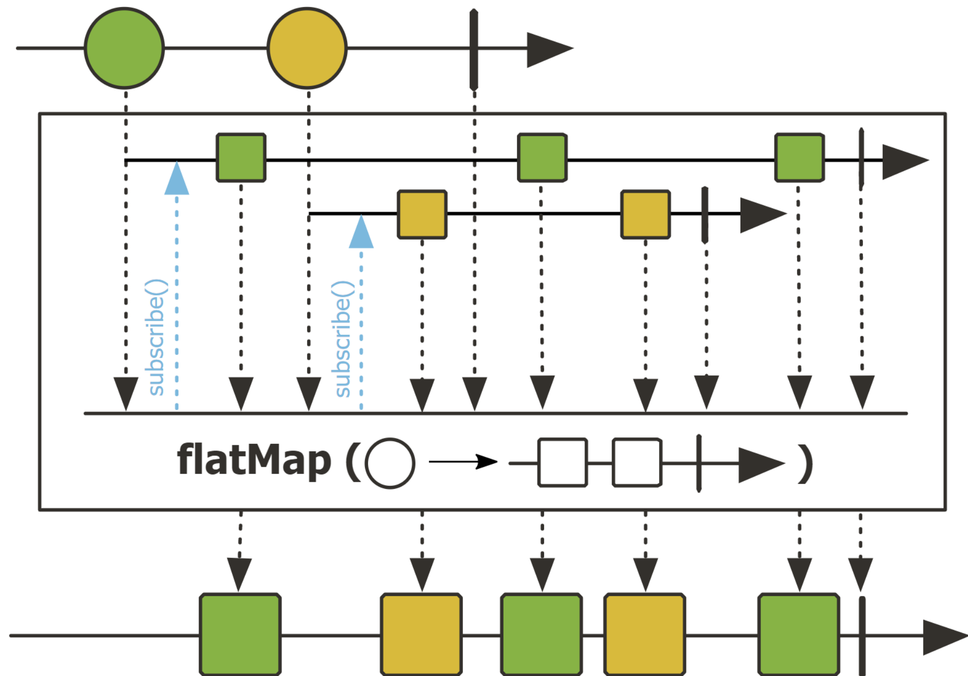
```
return Observable
  .fromIterable(bigFractionList)

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

     .subscribeOn
        (Schedulers
         .computation()))

  .reduce(BigFraction::add)
```

Return an Observable that emits multiplied BigFraction objects via the RxJava flatMap() concurrency idiom

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

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing
  - Project Reactor's Flux.flatMap() operator works the same way



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

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing
  - Project Reactor's Flux.flatMap() operator works the same way
  - Similar to the Stream.flatMap() method in Java Streams

flatMap

```
<R> Stream<R> flatMap(
Function<? super T,? extends Stream<? extends R>> mapper)
```

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

```
List<String> a = List.of("d", "g");
List<String> b = List.of("a", "c");
Stream
     .of(a, b)
     .flatMap(List::stream)
     .sorted()
     .forEach(System.out::println);
```

Flatten, sort, & print two lists of strings

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

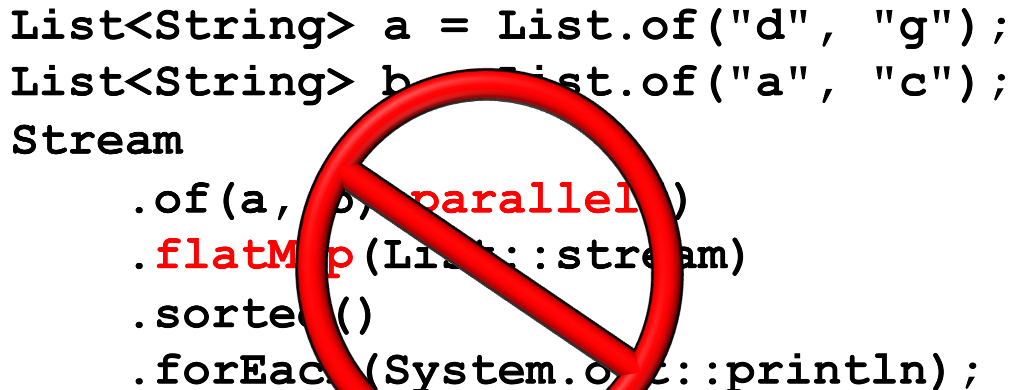
Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing
  - Project Reactor's Flux.flatMap() operator works the same way
  - Similar to the Stream.flatMap() method in Java Streams
    - However, Stream.flatMap() doesn't support parallelism..

flatMap

```
<R> Stream<R> flatMap(
Function<? super T,? extends Stream<? extends R>> mapper)
```

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)



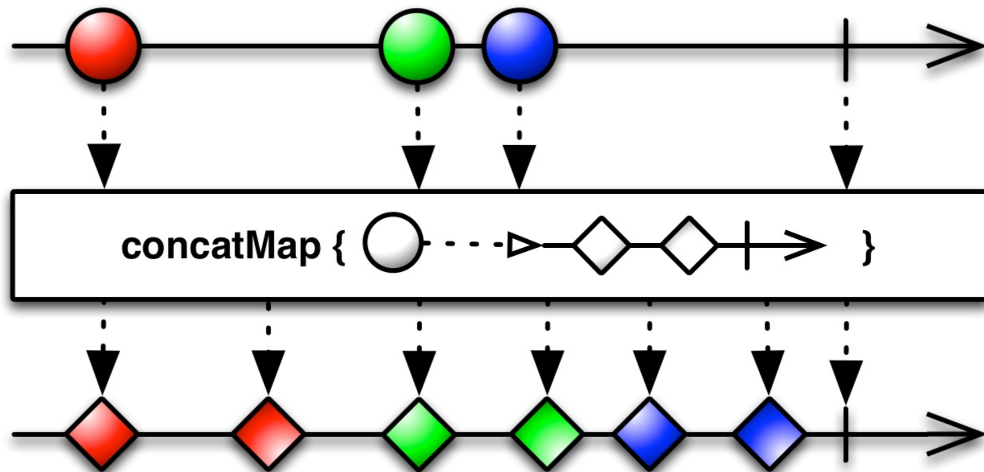
See stackoverflow.com/questions/45038120/parallel-flatmap-always-sequential/66386078

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronously
  - This operator is often used to trigger concurrent processing
  - Project Reactor's Flux.flatMap() operator works the same way
  - Similar to the Stream.flatMap() method in Java Streams
  - flatMap() doesn't ensure the order of the items in the resulting stream

Key Transforming Operators in the Observable Class

- The flatMap() operator
  - Transform the elements emitted by this Observable asynchronous
  - This operator is often used to trigger concurrent processing
  - Project Reactor's Flux.flatMap() operator works the same way
  - Similar to the Stream.flatMap() method in Java Streams
  - flatMap() doesn't ensure the order of the items in the resulting stream
    - Use concatMap() if order matters

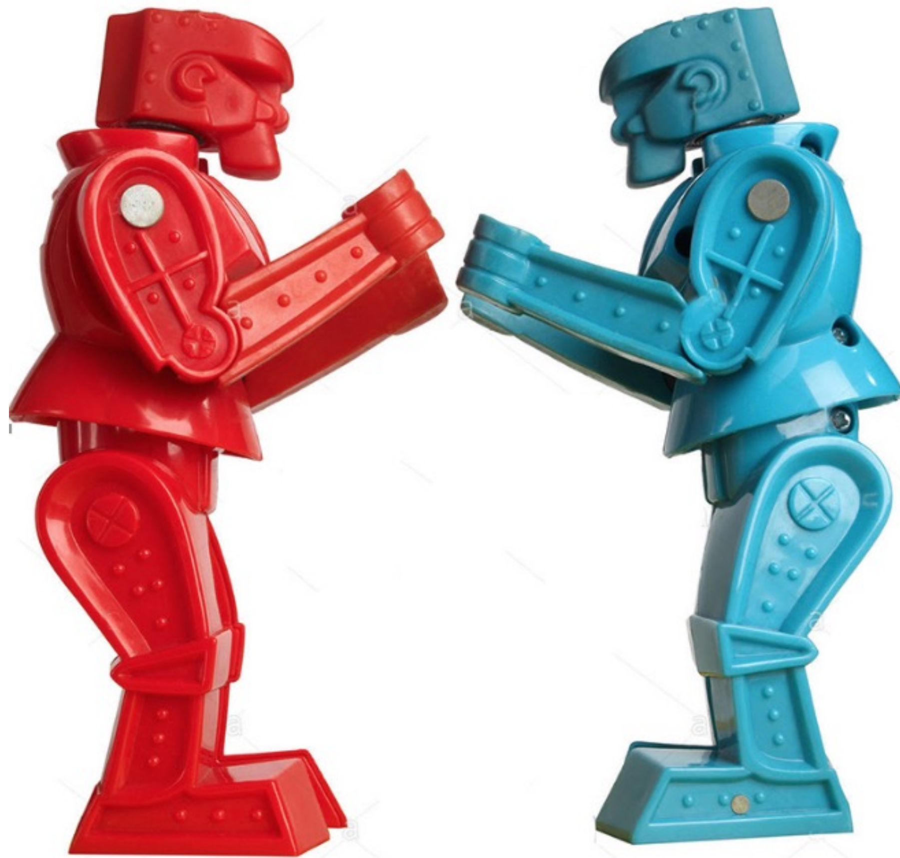


See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#concatMap

Comparing Observable map() & flatMap()

Comparing Observable map() & flatMap()

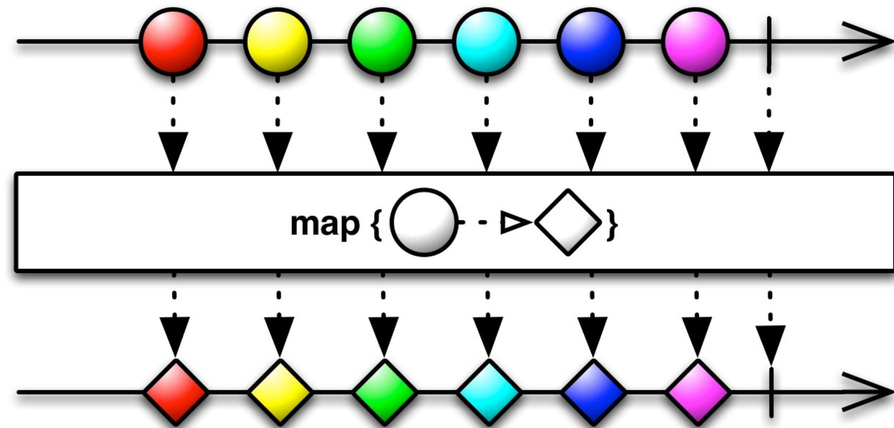
- The map() vs. flatMap() operators



See en.wikipedia.org/wiki/Rock_'Em_Sock_'Em_Robots

Comparing Observable map() & flatMap()

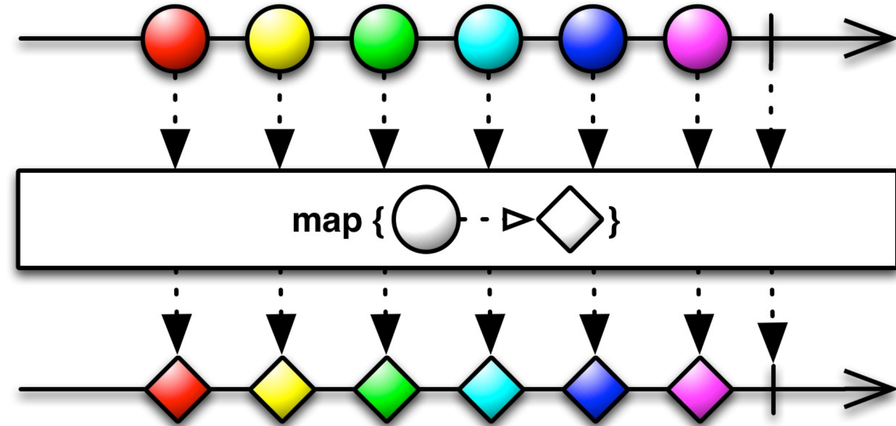
- The map() vs. flatMap() operators
  - map() transforms each value in an Observable stream into one value



See medium.com/mindorks/rxjava-operator-map-vs-flatmap-427c09678784

Comparing Observable map() & flatMap()

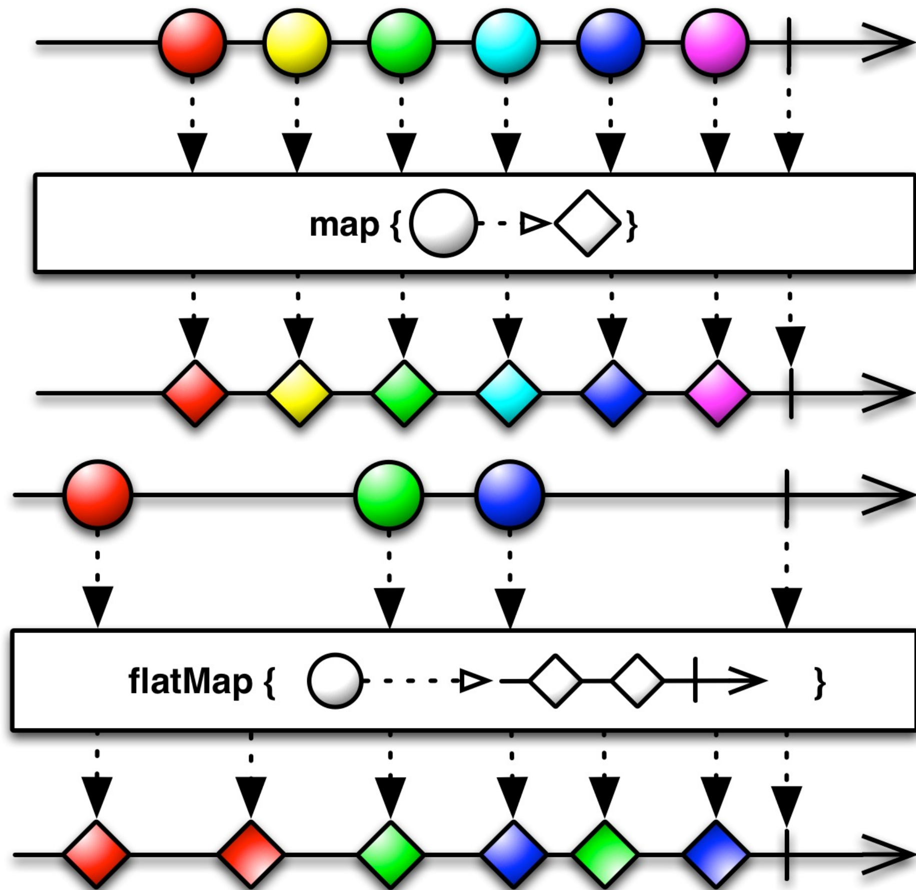
- The map() vs. flatMap() operators
  - map() transforms each value in an Observable stream into one value
    - e.g., used for synchronous 1-to-1 transformations



The # of output elements equal the # of input elements

Comparing Observable map() & flatMap()

- The map() vs. flatMap() operators
  - map() transforms each value in an Observable stream into one value
  - flatMap() transforms each value in an Observable stream into an arbitrary number (0+) values



See medium.com/mindorks/rxjava-operator-map-vs-flatmap-427c09678784

Comparing Observable map() & flatMap()

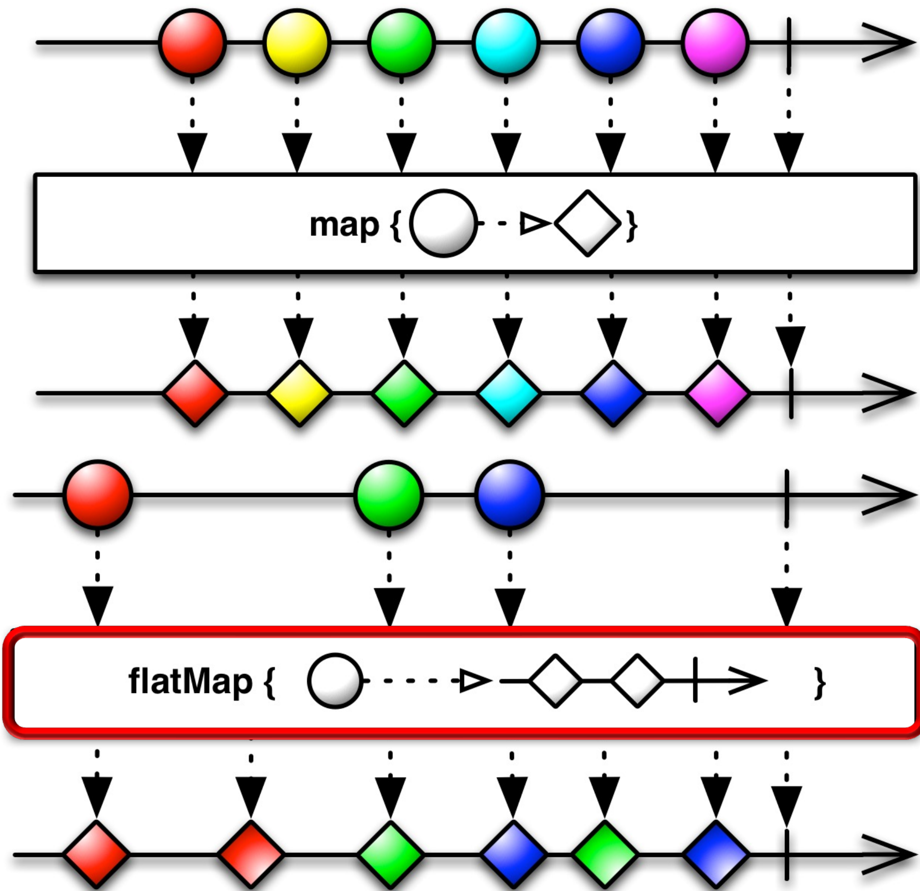
- The map() vs. flatMap() operators
  - map() transforms each value in an Observable stream into one value
  - flatMap() transforms each value in an Observable stream into an arbitrary number (0+) values
    - e.g., intended for asynchronous 1-to-N transformations

The # of output elements may differ from the # of input elements

Comparing Observable map() & flatMap()

- The map() vs. flatMap() operators
  - map() transforms each value in an Observable stream into one value
  - flatMap() transforms each value in an Observable stream into an arbitrary number (0+) values
  - flatMap() is used extensively in RxJava

POPULAR



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

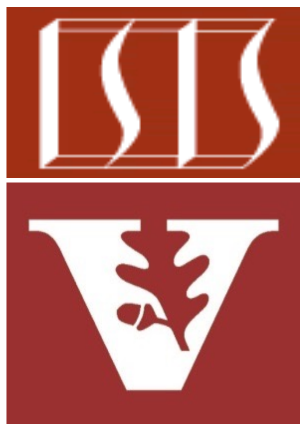
Key Error Handling Operators in the Observable Class

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 operators defined in—or used with—Observables
  - Factory method operators
  - Transforming operators
  - Concurrency & scheduler operators
  - Error handling operators
    - These operators handle exceptions that occur in an Observable chain
      - e.g., onErrorReturnItem()

Key Error Handling Operators in the Observable Class

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())

```
Observable<T>
onErrorReturnItem(T item)
```

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

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
    - The param value is emitted along via a regular onComplete() when the Observable signals an exception

```
Observable<T>
onErrorReturnItem(T item)
```

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

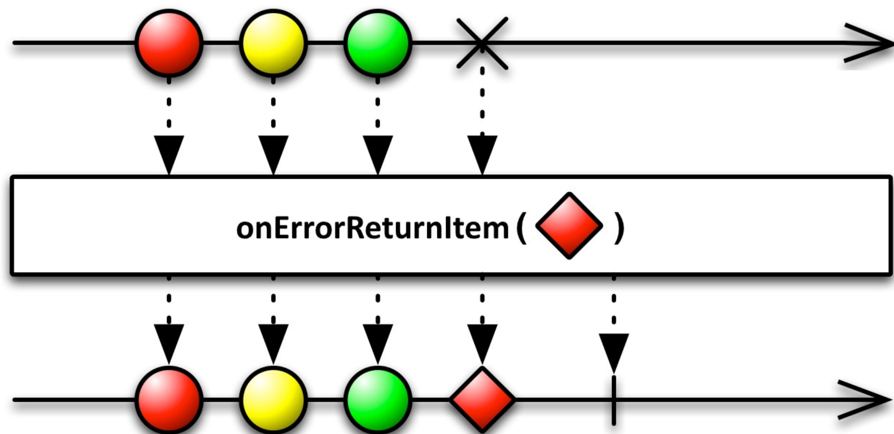
Key Error Handling Operators in the Observable Class

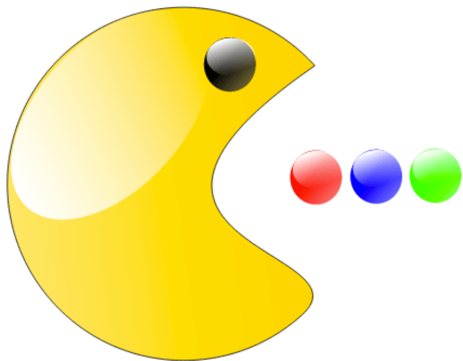
- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
    - The param value is emitted along via a regular onComplete() when the Observable signals an exception
    - Returns a new Observable that emits the given item

```
Observable<T>
onErrorReturnItem(T item)
```

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
  - This operator "swallows" the exception so it won't propagate up the call chain/stack further

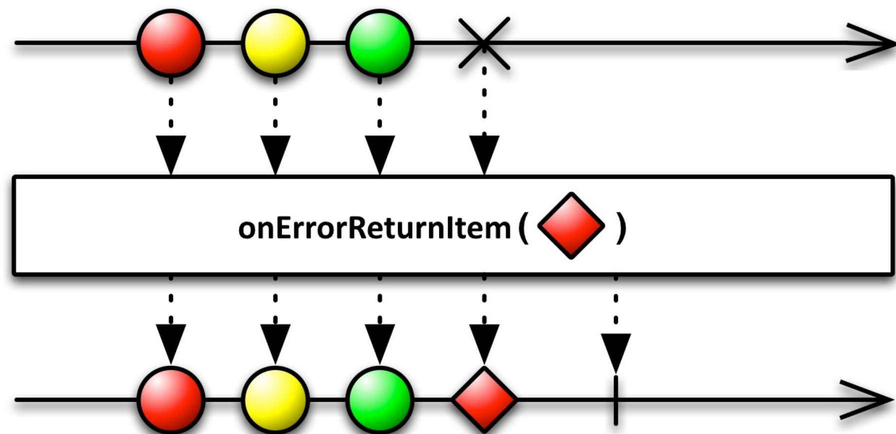




See en.wikipedia.org/wiki/Error_hiding

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
  - This operator "swallows" the exception so it won't propagate up the call chain/stack further



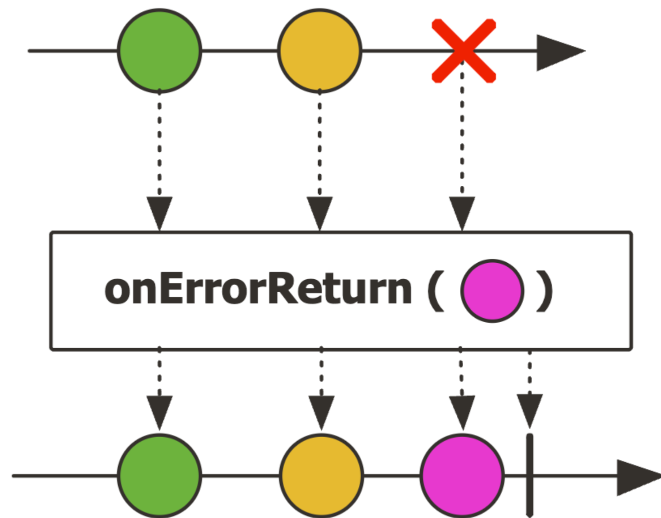

```
return Observable
   .fromCallable(BigFraction
            .valueOf(Math.abs(sRANDOM.nextInt()), denominator))
   .subscribeOn(Schedulers.computation())
   .onErrorReturnItem(ZERO)
   .map(multiplyBigFractions);
```

Convert ArithmeticException to ZERO when denominator == 0

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

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
  - This operator "swallows" the exception so it won't propagate up the call chain/stack further
  - Project Reactor's operator Flux .onErrorReturn() works the same



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

Key Error Handling Operators in the Observable Class

- The onErrorReturnItem() operator
  - Ends the flow with the given item when the Observable fails (instead of signaling the error via onError())
  - This operator "swallows" the exception so it won't propagate up the call chain/stack further
  - Project Reactor's operator Flux .onErrorReturn() works the same
  - The Java CompletableFuture exceptionally() method is similar

exceptionally

```
CompletionStage<T> exceptionally(
Function<Throwable,? extends T> fn)
```

Returns a new CompletionStage that, when this stage completes exceptionally, is executed with this stage's exception as the argument to the supplied function. Otherwise, if this stage completes normally, then the returned stage also completes normally with the same value.

Parameters:

fn - the function to use to compute the value of the returned CompletionStage if this CompletionStage completed exceptionally

Returns:

the new CompletionStage

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#exceptionally

End of Key Error Handling Operators in the Observable Class

Key Blocking Operators in the Single 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 Single operators
  - Concurrency & scheduler operators
  - Blocking operators
    - These operators block awaiting a Single to emit its value
      - e.g., blockingGet()

The Single that emits a value typically runs asynchronously in a different thread of control

Key Blocking Operators in the Single Class

Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception

```
T blockingGet()
```

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

Key Blocking Operators in the Single Class

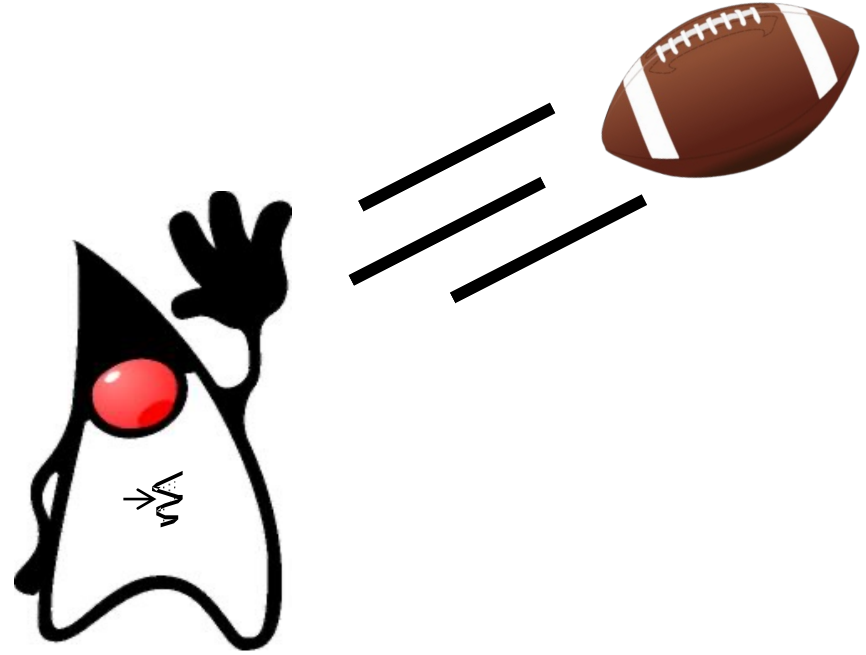
- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received

```
T blockingGet()
```

Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received
    - If the source signals errors, the original exception is thrown

```
T blockingGet()
```



Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received
    - If the source signals errors, the original exception is thrown
      - A checked exception is wrapped in a RuntimeException

```
T blockingGet()
```

```
public class RuntimeException
extends Exception
```

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

`RuntimeException` and its subclasses are *unchecked exceptions*. Unchecked exceptions do *not* need to be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

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

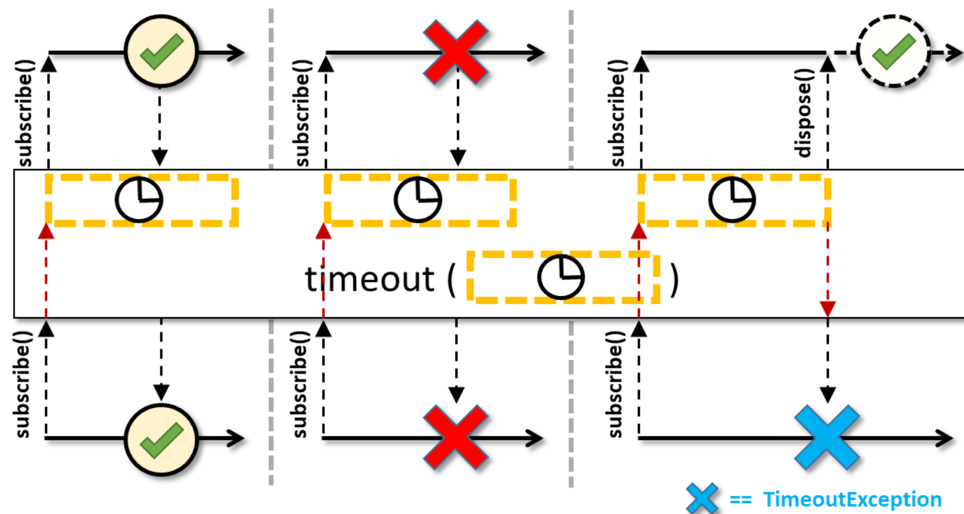
Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received
    - If the source signals errors, the original exception is thrown
    - There is no timed version of blockingGet()

```
T blockingGet()
```

Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received
    - If the source signals errors, the original exception is thrown
    - There is no timed version of blockingGet()
      - However, there are timeout() operators



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

Key Blocking Operators in the Single Class

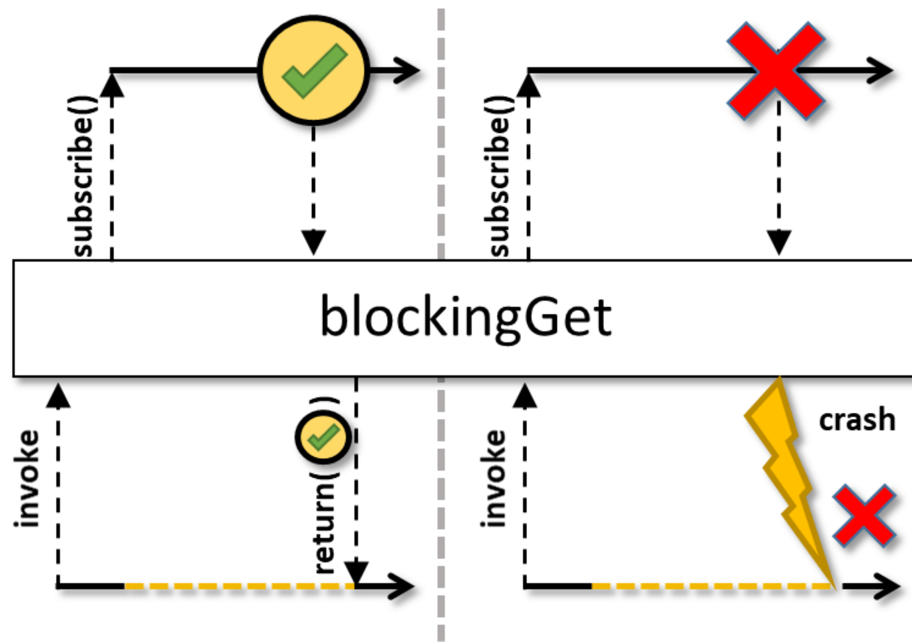
- The blockingGet() operator
  - Block until current Single signals a success value or an exception
    - Returns the value received
    - If the source signals errors, the original exception is thrown
    - There is no timed version of blockingGet()
    - blockingGet() internally calls subscribe() to initiate the Single processing chain

```
final T blockingGet() {
  BlockingMultiObserver<T>
    observer = new
      BlockingMultiObserver<>();
  subscribe(observer);
  return observer.blockingGet();
}
```

See github.com/ReactiveX/RxJava/blob/3.x/src/main/java/io/reactivex/rxjava3/core/Single.java

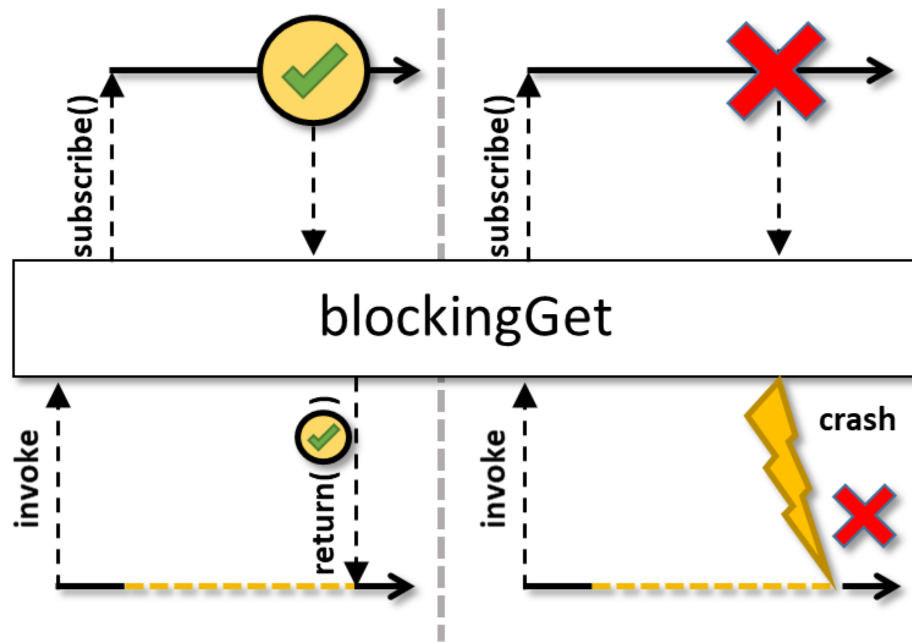
Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
  - This operator does not operate by default on a particular Scheduler



Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
  - This operator does not operate by default on a particular Scheduler
    - However, the Single that emits a value often runs asynchronously in a different thread of control



Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
  - This operator does not operate by default on a particular Scheduler
  - Should only be used if a value is needed before proceeding

```
BigFraction result = Single
  .fromCallable(call)

  .subscribeOn
     (Schedulers.single())

  .blockingGet();
```

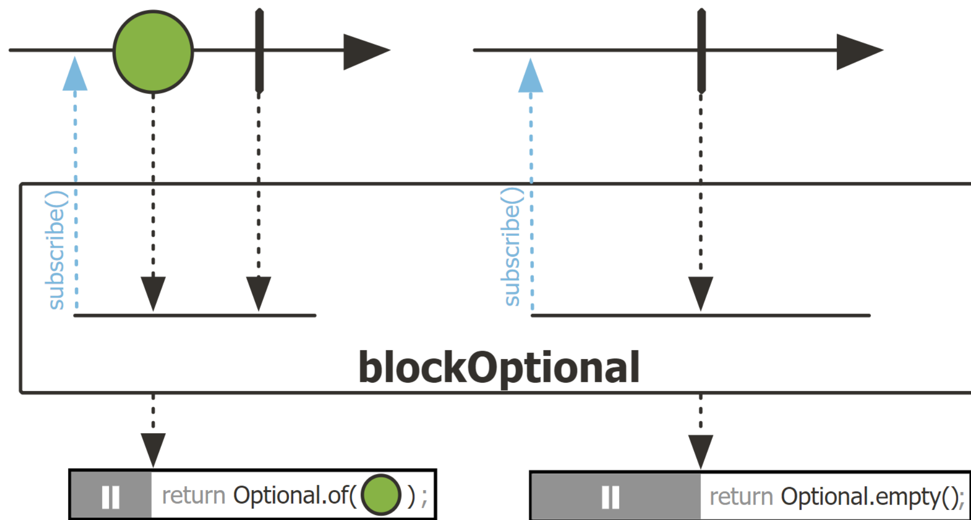
Block caller until the back ground operation completes

```
System.out.println
  (result::toMixedString);
```

See [Reactive/Single/ex2/src/main/java/SingleEx.java]()

Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
  - This operator does not operate by default on a particular Scheduler
  - Should only be used if a value is needed before proceeding
  - Project Reactor's operator Mono. blockOptional() is similar
    - i.e., it blocks indefinitely until a next signal is received or the Mono completes empty



See projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#blockOptional

Key Blocking Operators in the Single Class

- The blockingGet() operator
  - Block until current Single signals a success value or an exception
  - This operator does not operate by default on a particular Scheduler
  - Should only be used if a value is needed before proceeding
  - Project Reactor's operator Mono.blockOptional() is similar
  - Similar to the join() method in Java CompletableFuture

```
CompletableFuture<BigFraction> f
  = CompletableFuture
     .supplyAsync(() -> {
        BigFraction bf1 = new
          BigFraction(sF1);
        BigFraction bf2 = new
          BigFraction(sF2);
        return bf1.multiply(bf2);
     });
...
System.out.println
   ("result = "
    + f.join().toMixedString());
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html#join

End of Key Blocking Operators in the Single Class

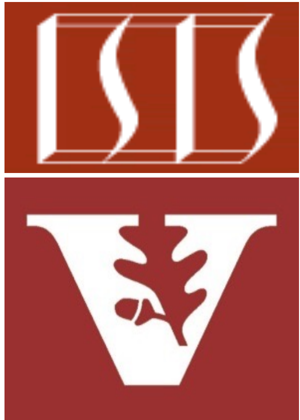
Overview of the ParallelFlowable Class

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

- Understand the capabilities of the ParallelFlowable class

Class ParallelFlowable<T>

java.lang.Object
    io.reactivex.rxjava3.parallel.ParallelFlowable<T>

Type Parameters:
    `T` - the value type

```
public abstract class ParallelFlowable<T>
extends Object
```

Abstract base class for parallel publishing of events signaled to an array of `Subscribers`.

Use `from(Publisher)` to start processing a regular `Publisher` in 'rails'. Use `runOn(Scheduler)` to introduce where each 'rail' should run on thread-vise. Use `sequential()` to merge the sources back into a single `Flowable`.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html

Learning Objectives in this Part of the Lesson

- Understand the capabilities of the ParallelFlowable class
  - Simplifies parallel processing *cf.* the flatMap() concurrency idiom

SIMPLE

```
return Flowable
    .fromArray(bigFractionArray)
    .parallel()
    .runOn(Schedulers.computation())
    .map(bf -> bf.multiply(sBigReducedFrac))
    .reduce(BigFraction::add)
    .firstElement()...
```

```
return Observable
    .fromArray(bigFractionArray)
    .flatMap(bf -> Observable
        .fromCallable(() -> bf
            .multiply(sBigFraction))
        .subscribeOn(Schedulers
                       .computation()))
    .reduce(BigFraction::add)...
```

See earlier lesson on *"Key Transforming Operators in the Observable Class (Part 3)"*

Overview of the ParallelFlowable Class

Overview of the ParallelFlowable Class

- The RxJava flatMap() concurrency idiom performs relatively well, but is also somewhat convoluted..

```
return Observable
   .fromArray(bigFractionArray)

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

      .subscribeOn(Schedulers
                   .computation()))

   .reduce(BigFraction::add)
   ...
```

Return an Observable that emits multiplied BigFraction objects via the RxJava flatMap() concurrency idiom

See previous lessons on *"Key Transforming Operators in the Observable Class (Part 3)"*

Overview of the ParallelFlowable Class

- The RxJava flatMap() concurrency idiom performs relatively well, but is also somewhat convoluted..
  - Particularly in comparison with Java parallel streams

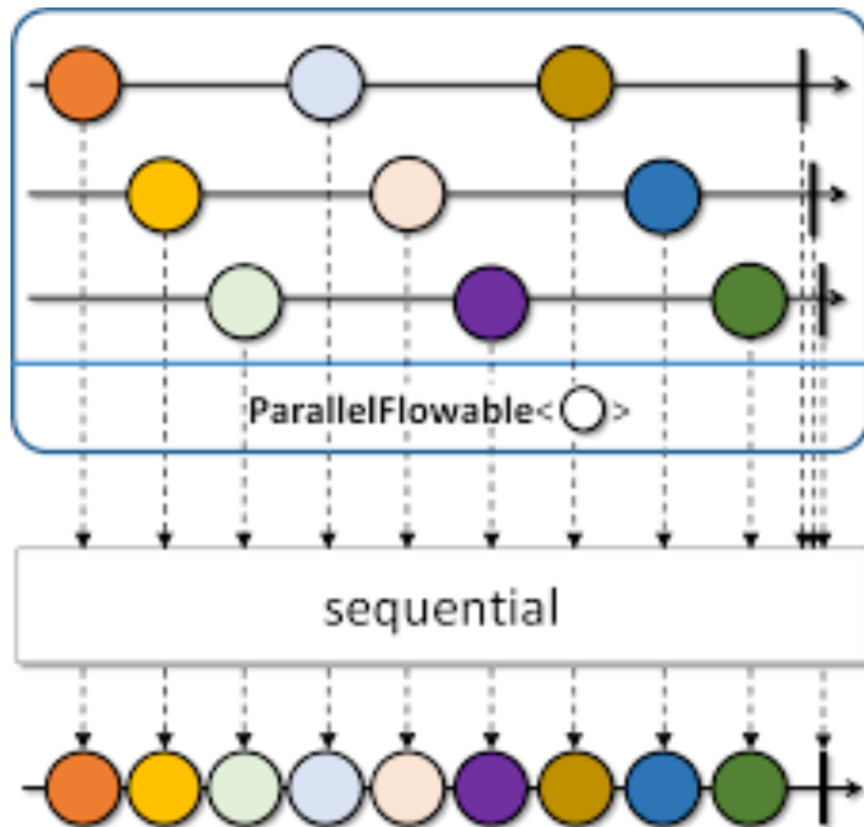
```
return Stream
   .of(bigFractionArray)

   .parallel()

   .map(bf -> bf
        .multiply(sBigFraction))

   .reduce(ZERO, BigFraction::add)
```

```
return Observable
   .fromArray(bigFractionArray)

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

      .subscribeOn(Schedulers
                   .computation()))

   .reduce(BigFraction::add)
   ...
```

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Overview of the ParallelFlowable Class

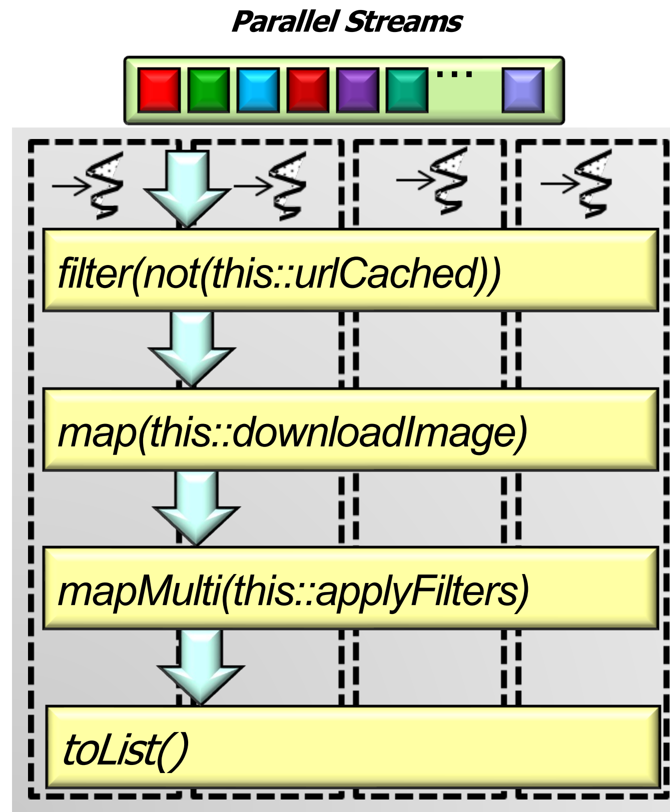
- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams



See dzone.com/articles/rxjava-idiomatic-concurrency-flatmap-vs-parallel

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
    - i.e., intended for "embarrassingly parallel" tasks

"Embarrassingly parallel" tasks have little/no dependency or need for communication between tasks or for sharing results between them

See en.wikipedia.org/wiki/Embarrassingly_parallel

Overview of the ParallelFlowable Class

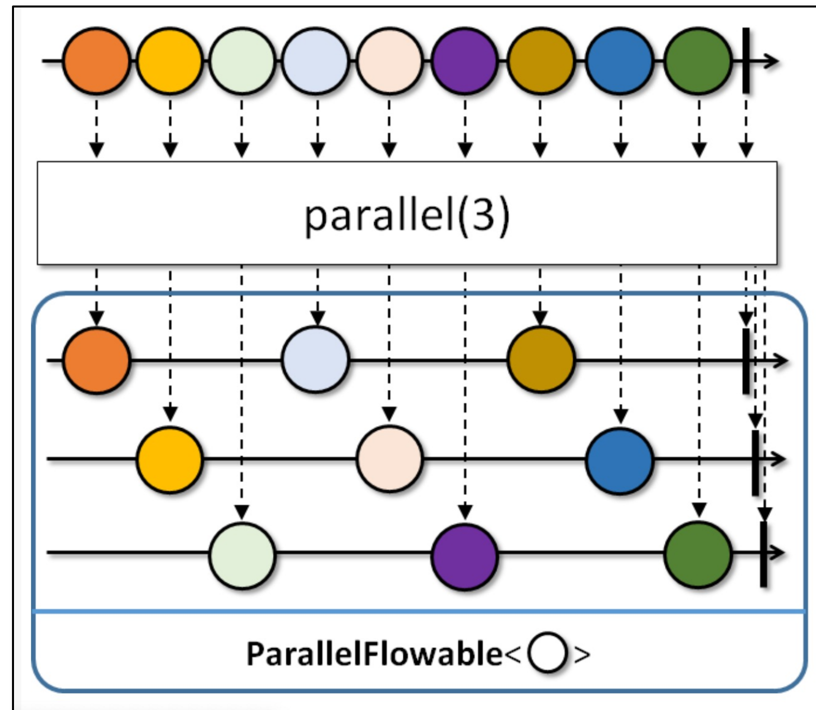
- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
  - Avoids the convoluted syntax of the flatMap() concurrency idiom

See dzone.com/articles/rxjava-idiomatic-concurrency-flatmap-vs-parallel

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
  - Avoids the convoluted syntax of the flatMap() concurrency idiom
  - The Flowable.parallel() factory method creates a ParallelFlowable

```
ParallelFlowable<T> parallel()
```

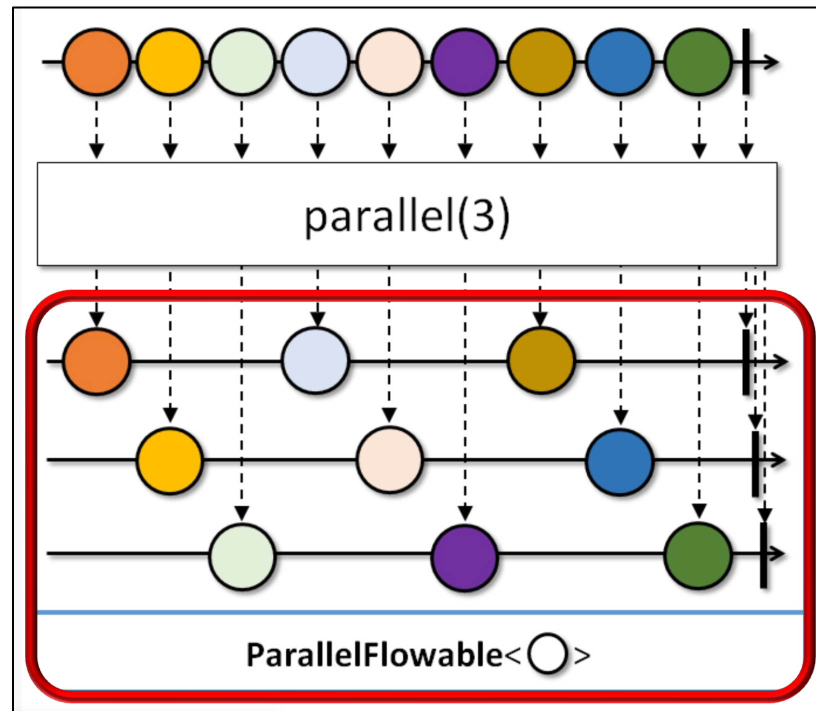


See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html#parallel

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
  - Avoids the convoluted syntax of the flatMap() concurrency idiom
  - The Flowable.parallel() factory method creates a ParallelFlowable
    - Elements are processed in parallel via 'rails' in round-robin order

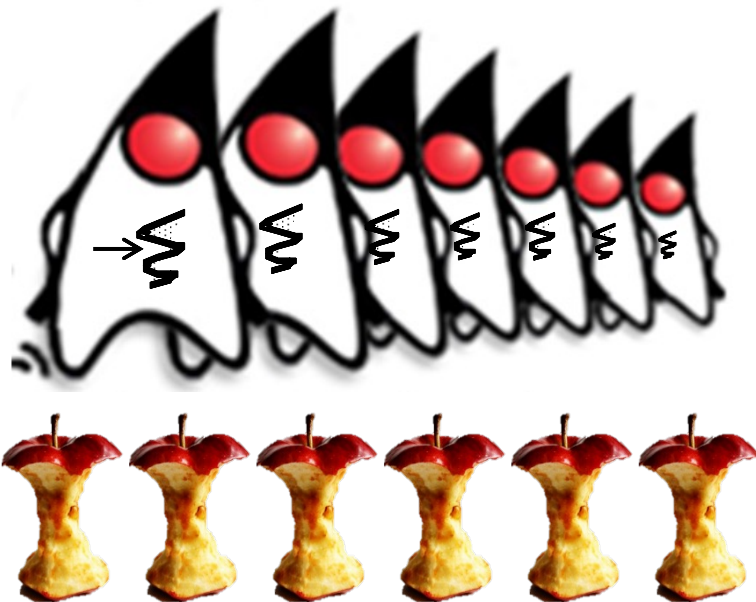
```
ParallelFlowable<T> parallel()
```



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html#parallel

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
  - Avoids the convoluted syntax of the flatMap() concurrency idiom
  - The Flowable.parallel() factory method creates a ParallelFlowable
    - Elements are processed in parallel via 'rails' in round-robin order
    - By default, the # of rails is set to the # of available CPU cores



See docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#availableProcessors

Overview of the ParallelFlowable Class

- ParallelFlowable is a subset of Flowable that provides a more concise means of processing multiple values in parallel
  - Similar to Java parallel streams
  - Avoids the convoluted syntax of the flatMap() concurrency idiom
  - The Flowable.parallel() factory method creates a ParallelFlowable
    - Elements are processed in parallel via 'rails' in round-robin order
    - By default, the # of rails is set to the # of available CPU cores
      - This setting can be changed programmatically

parallel

```
@BackpressureSupport(value=FULL)
 @SchedulerSupport(value="none")
 @CheckReturnValue
 @NonNull
public final @NonNull ParallelFlowable<T> parallel(int parallelism)
```

Parallelizes the flow by creating the specified number of 'rails' and dispatches the upstream items to them in a round-robin fashion.

Note that the rails don't execute in parallel on their own and one needs to apply `ParallelFlowable.runOn(Scheduler)` to specify the `Scheduler` where each rail will execute.

To merge the parallel 'rails' back into a single sequence, use `ParallelFlowable.sequential()`.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html#parallel

Key Operators in the ParallelFlowable Class

Key Operators in the ParallelFlowable Class

- ParallelFlowable supports a subset of Flowable operators that process elements in parallel across the rails
  - e.g., map(), filter(), concatMap(), flatMap(), collect(), & reduce()

See github.com/ReactiveX/RxJava/wiki/Parallel-flows

Key Operators in the ParallelFlowable Class

- The runOn() operator specifies where each 'rail' will observe its incoming elements

```
ParallelFlowable<T> runOn(Scheduler
                          scheduler)
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#runOn

Key Operators in the ParallelFlowable Class

- The runOn() operator specifies where each 'rail' will observe its incoming elements
  - Specified via a Scheduler that performs no work-stealing

```
ParallelFlowable<T> runOn(Scheduler
                          scheduler)
```

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

Key Operators in the ParallelFlowable Class

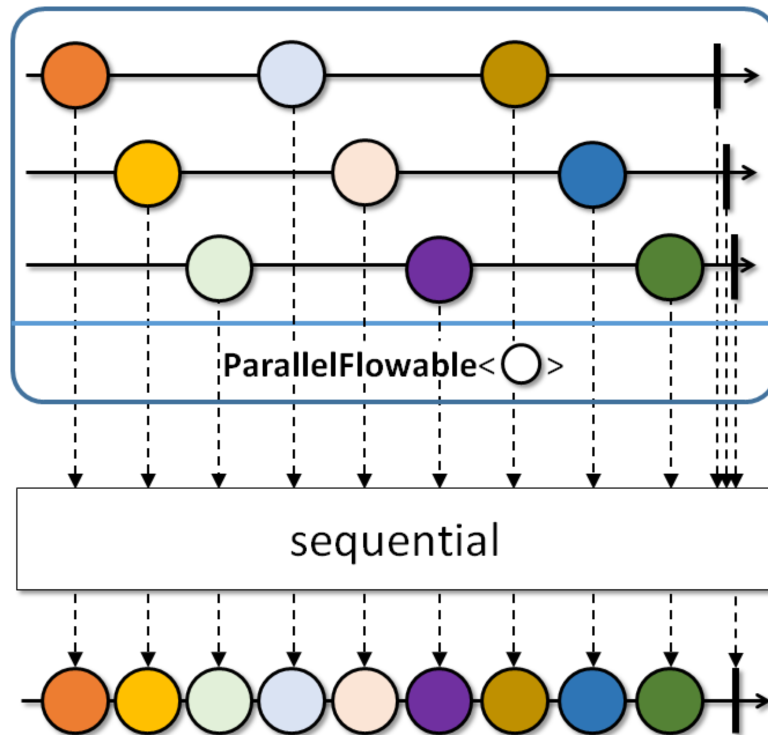
- The runOn() operator specifies where each 'rail' will observe its incoming elements
  - Specified via a Scheduler that performs no work-stealing
  - Returns the new Parallel Flowable instance

```
ParallelFlowable<T> runOn(Scheduler
                          scheduler)
```

Key Operators in the ParallelFlowable Class

- A ParallelFlowable can be converted back into a Flowable via sequential()

`Flowable<T> sequential()`

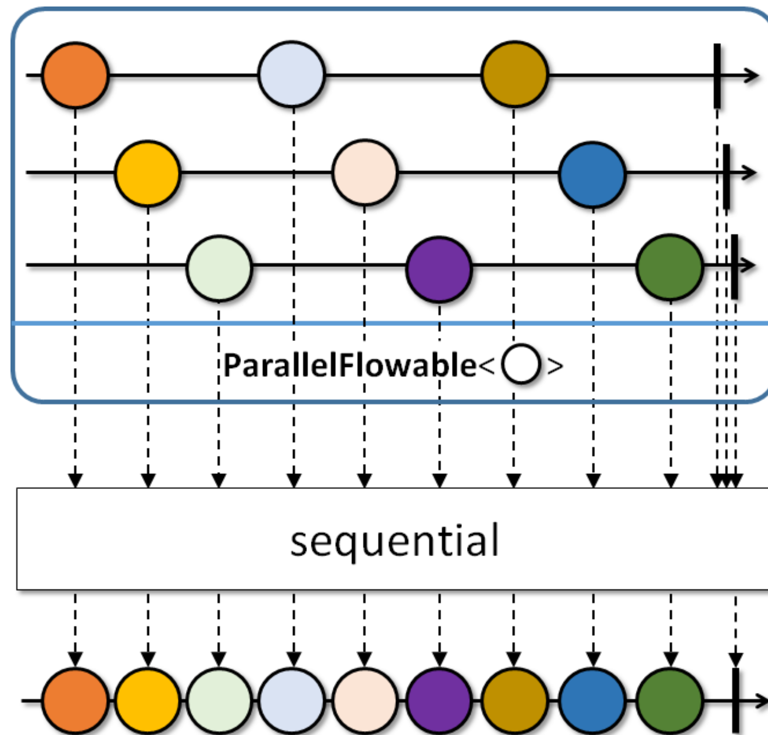


See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#sequential

Key Operators in the ParallelFlowable Class

- A ParallelFlowable can be converted back into a Flowable via sequential()
  - Merge the values from each 'rail' in a round-robin fashion

```
Flowable<T> sequential()
```



See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#sequential

Key Operators in the ParallelFlowable Class

- ParallelFlowable.reduce() can also be used to convert back into a Flowable

reduce

```
@CheckReturnValue
 @NonNull
 @BackpressureSupport(value=UNBOUNDED_IN)
 @SchedulerSupport(value="none")
public final @NonNull Flowable<T> reduce(@NonNull BiFunction<T,T,T> reducer)
```

Reduces all values within a 'rail' and across 'rails' with a reducer function into one `Flowable` sequence.

Note that the same reducer function may be called from multiple threads concurrently.

Backpressure:
The operator honors backpressure from the downstream and consumes the upstream rails in an unbounded manner (requesting `Long.MAX_VALUE`).

Scheduler:
`reduce` does not operate by default on a particular `Scheduler`.

Parameters:
`reducer` - the function to reduce two values into one.

Returns:
the new `Flowable` instance emitting the reduced value or empty if the current `ParallelFlowable` is empty

Throws:
`NullPointerException` - if `reducer` is `null`

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#reduce

Key Operators in the ParallelFlowable Class

- ParallelFlowable.reduce() can also be used to convert back into a Flowable
  - Reduces all values within a 'rail' & across 'rails' into a Flowable

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

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#reduce

Key Operators in the ParallelFlowable Class

- ParallelFlowable.reduce() can also be used to convert back into a Flowable
  - Reduces all values within a 'rail' & across 'rails' into a Flowable
  - The BiFunction param reduces two values into one successively

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

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

Key Operators in the ParallelFlowable Class

- ParallelFlowable.reduce() can also be used to convert back into a Flowable
  - Reduces all values within a 'rail' & across 'rails' into a Flowable
  - The BiFunction param reduces two values into one successively
  - Return a regular Flowable that contains just one element

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

End of Overview of the ParallelFlowable Class

Key Scheduler Operators for RxJava Reactive Types (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—ParallelFlowables
  - Scheduler operators
    - These operators provide the context to run other operators in designated threads & thread pools
      - e.g., Schedulers.io()

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

Key Scheduler Operators for RxJava Reactive Types

Key Scheduler Operators for RxJava Reactive Types

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

```
static Scheduler io()
```

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

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for I/O-bound work

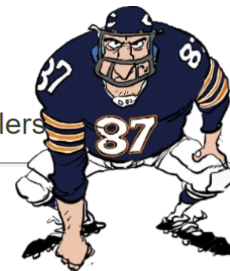
```
static Scheduler io()
```

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking operations

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.

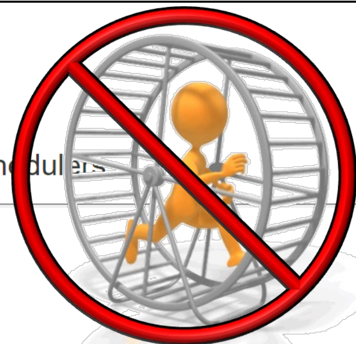
See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/schedulers/Schedulers.html

Key Scheduler Operators for RxJava Reactive Types

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

Class Schedulers

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



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

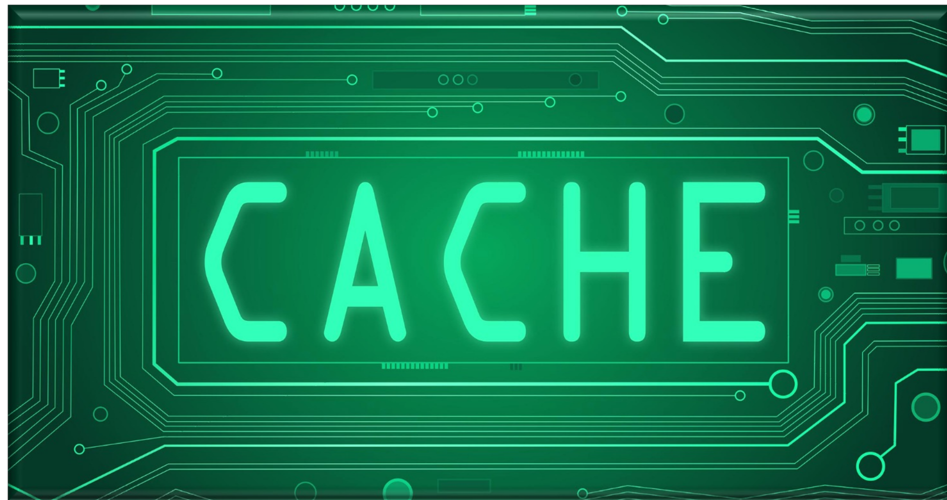
Static factory methods for returning standard Scheduler instances.

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.

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

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking operations
      - Either starts a new thread or reuses an idle one from a cache



Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking operations
    - Either starts a new thread or reuses an idle one from a cache
      - The goal is to maximally utilize the CPU cores

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
   .getUrlFlowable()

   .parallel()

   .runOn(Schedulers.io())

   .map(downloadAndStoreImage)

   .sequential()

   .collect(Collectors.toList())

   .doOnSuccess(...)
```

Download images from remote web servers in parallel & store them on the local computer

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex4

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
    .getUrlFlowable()

    .parallel()

    .runOn(Schedulers.io())

    .map(downloadAndStoreImage)

    .sequential()

    .collect(Collectors.toList())

    .doOnSuccess(...)
```

Create a Flowable containing URLs to download from remote web servers

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
   .getUrlFlowable()

   .parallel()

   .runOn(Schedulers.io())

   .map(downloadAndStoreImage)

   .sequential()

   .collect(Collectors.toList())

   .doOnSuccess(...)
```

Convert the Flowable into a ParallelFlowable

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
    .getUrlFlowable()

    .parallel()

    .runOn(Schedulers.io())

    .map(downloadAndStoreImage)

    .sequential()

    .collect(Collectors.toList())

    .doOnSuccess(...)
```

Designate the I/O Scheduler that will download & store each image in parallel

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlowable.html#runOn

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
   .getUrlFlowable()

   .parallel()

   .runOn(Schedulers.io())

   .map(downloadAndStoreImage)

   .sequential()

   .collect(Collectors.toList())

   .doOnSuccess(...)
```

Download & store images in parallel

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
    .getUrlFlowable()

    .parallel()

    .runOn(Schedulers.io())

    .map(downloadAndStoreImage)

    .sequential()

    .collect(Collectors.toList())

    .doOnSuccess(...)
```

Merge the values from each 'rail' in a round-robin fashion & expose it as a regular Flowable sequence

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
   .getUrlFlowable()

   .parallel()

   .runOn(Schedulers.io())

   .map(downloadAndStoreImage)

   .sequential()

   .collect(Collectors.toList())

   .doOnSuccess(...)
```

Collect the Flowable into a List

Key Scheduler Operators for RxJava Reactive Types

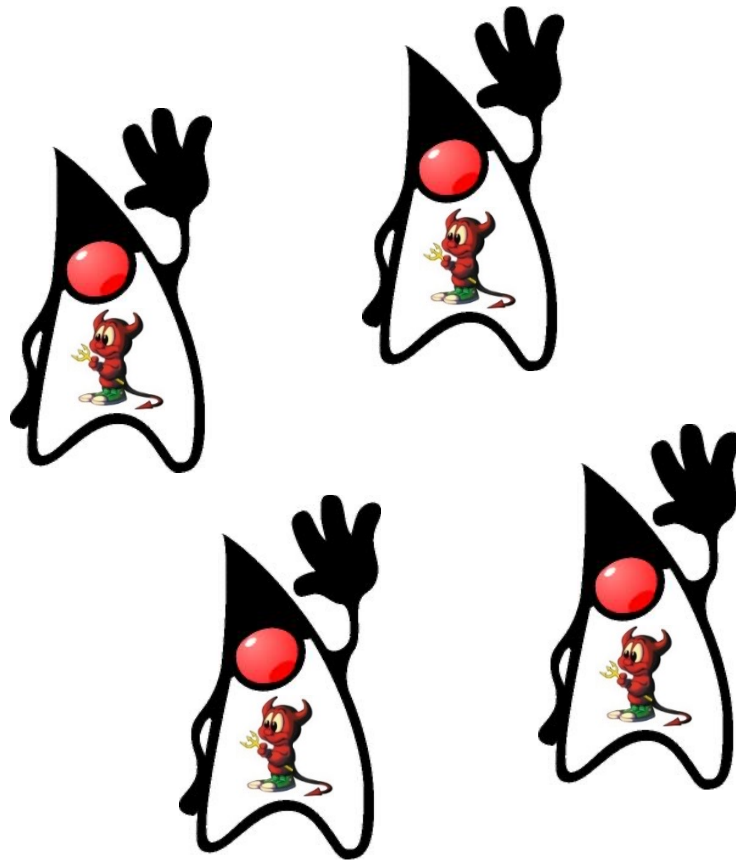
- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.

```
return Options.instance()
   .getUrlFlowable()

   .parallel()

   .runOn(Schedulers.io())

   .map(downloadAndStoreImage)

   .sequential()

   .collect(Collectors.toList())

   .doOnSuccess(...)
```

Handle the final 'reduced' results

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - 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 RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
  - The Schedulers.boundedElastic() operator in Project Reactor is similar

boundedElastic

```
public static Scheduler boundedElastic()
```

The common *boundedElastic* instance, a `Scheduler` that dynamically creates a bounded number of ExecutorService-based Workers, reusing them once the Workers have been shut down. The underlying daemon threads can be evicted if idle for more than `60` seconds.

The maximum number of created threads is bounded by a `cap` (by default ten times the number of available CPU cores, see `DEFAULT_BOUNDED_ELASTIC_SIZE`). The maximum number of task submissions that can be enqueued and deferred on each of these backing threads is bounded (by default 100K additional tasks, see `DEFAULT_BOUNDED_ELASTIC_QUEUESIZE`). Past that point, a `RejectedExecutionException` is thrown.

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

Key Scheduler Operators for RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single-threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
  - The Schedulers.boundedElastic() operator in Project Reactor is similar
  - The Java common fork-join pool is also similar

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 RxJava Reactive Types

- The Schedulers.io() operator
  - Hosts a variable-size pool of single -threaded Executor Service-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
  - The Schedulers.boundedElastic() operator in Project Reactor is similar
  - The Java common fork-join pool is also similar
    - When used with the ManagedBlocker mechanism..

Interface ForkJoinPool.ManagedBlocker

Enclosing class:
ForkJoinPool

`public static interface` **`ForkJoinPool.ManagedBlocker`**

Interface for extending managed parallelism for tasks running in `ForkJoinPool`s.

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.

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

End of Key Scheduler Operators for RxJava Reactive Types (Part 3)

Key Composing Operators in the Flowable 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 operators defined in—or used with—ParallelFlowables
  - Scheduler operators
  - Composing operators
    - These operators make it possible to implement custom Flowable operators
      - e.g., compose()

Implementing Your Own Operators

Jump to bottom

David Gross edited this page on May 19, 2015 · 22 revisions

You can implement your own Observable operators. This page shows you how.

If your operator is designed to *originate* an Observable, rather than to transform or react to a source Observable, use the `create( )` method rather than trying to implement `Observable` manually. Otherwise, you can create a custom operator by following the instructions on this page.

If your operator is designed to act on the individual items emitted by a source Observable, follow the instructions under *Sequence Operators* below. If your operator is designed to transform the source Observable as a whole (for instance, by applying a particular set of existing RxJava operators to it) follow the instructions under *Transformational Operators* below.

See github.com/ReactiveX/RxJava/wiki/Implementing-Your-Own-Operators

Key Composing Operators in the Flowable Class

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function

```
<R> Flowable<R> compose
   (FlowableTransformer
      <? super T,
       ? extends R>
    composer)
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Flowable.html#compose

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
    - This function param transforms the current Flowable

```
<R> Flowable<R> compose
  (FlowableTransformer
     <? super T,
      ? extends R>
   composer)
```

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

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
    - This function param transforms the current Flowable
    - Returns a transformed Flowable

```
<R> Flowable<R> compose
  (FlowableTransformer
     <? super T,
      ? extends R>
   composer)
```

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators

Don't break the chain: use RxJava's compose() operator

Dan Lew
Mar 2, 2015 · 4 min read

See blog.danlew.net/2015/03/02/dont-break-the-chain

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators

```
var rateF = Flowable
    .just("LDN:NYC")
    .parallel()
    .compose(RxUtils
             .commonPoolFlowable())
    .map(this::findBestPrice)
    .sequential()
    .timeout(2,
             TimeUnit.SECONDS,
             sDEFAULT_RATE_F);
```

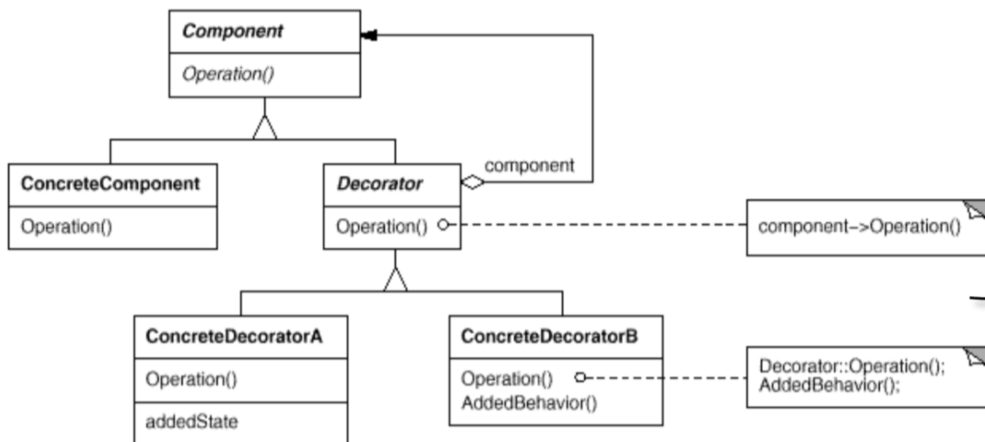
Asynchronously determine exchange rate from British pounds to US dollars via the Java common fork-Join pool

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex5

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators

```
var rateF = Flowable
    .just("LDN:NYC")
    .parallel()
    .compose(RxUtils
              .commonPoolFlowable())
    .map(this::findBestPrice)
    .sequential()
    .timeout(2,
              TimeUnit.SECONDS,
              sDEFAULT_RATE_F);
```

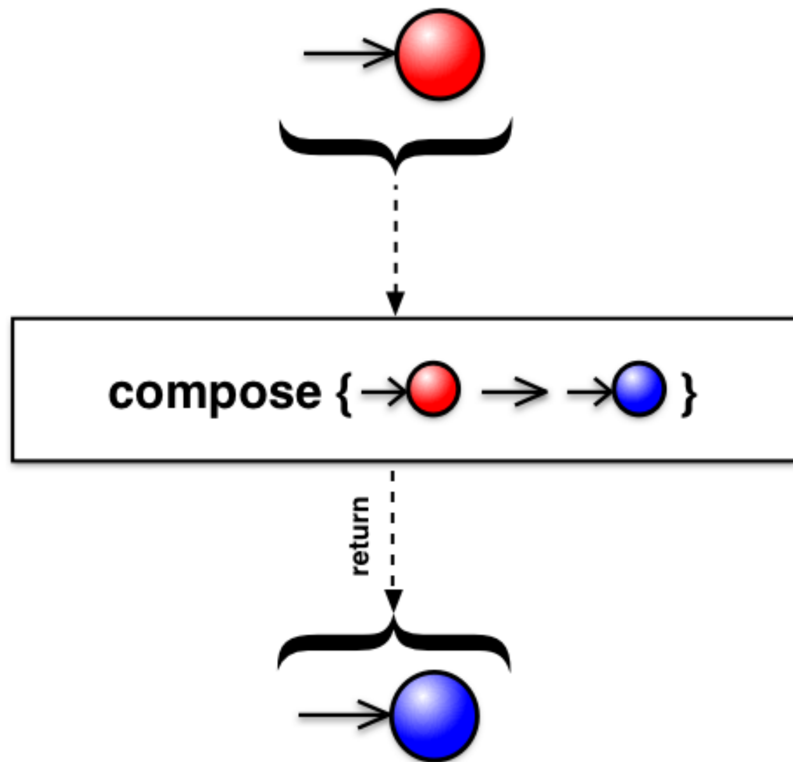


Implements the Decorator pattern that adds behavior to an object dynamically

See en.wikipedia.org/wiki/Decorator_pattern

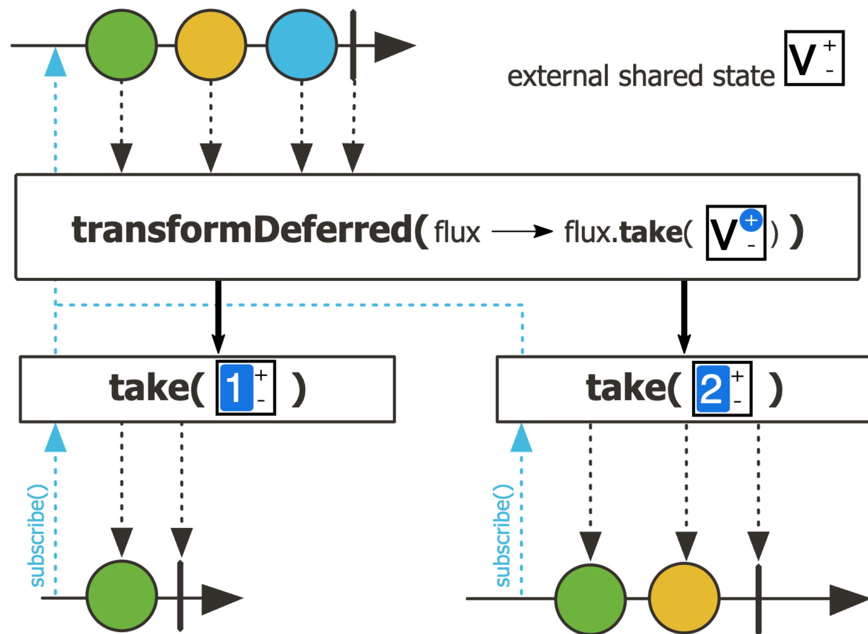
Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators
  - Does not operate by default on a particular Scheduler



Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators
  - Does not operate by default on a particular Scheduler
  - Project Reactor's operator Flux .transformDeferred() works the same



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

Key Composing Operators in the Flowable Class

- The compose() operator
  - Transform the Flowable by applying the FlowableTransformer function
  - Can be used to define "custom" operators that are chained together alongside standard RxJava operators
  - Does not operate by default on a particular Scheduler
  - Project Reactor's operator Flux .transformDeferred() works the same
  - The proposed Java Streams' Gatherer API is similar

Interface Gatherer<T,A,R>

Type Parameters:

T - the type of input elements to the gatherer operation

A - the potentially mutable state type of the gatherer operation (often hidden as an implementation detail)

R - the type of output elements from the gatherer operation

```
public interface Gatherer<T,A,R>
```

An intermediate operation that processes input elements, optionally mutating intermediate state, optionally transforming the input elements into a different type of output elements, and optionally applies final actions at end-of-upstream. Gatherer operations can be performed either sequentially, or be parallelized -- if a combiner function is supplied.

Examples of gathering operations include, but is not limited to: grouping elements into batches, also known as windowing functions; de-duplicating consecutively similar elements; incremental accumulation functions; incremental reordering functions, etc. The class `Gatherers` provides implementations of common gathering operations.

API Note:

A `Gatherer` is specified by four functions that work together to process input elements, optionally using intermediate state, and optionally perform a final operation at the end of input. They are:

- creation of a new, potentially mutable, state (`initializer()`)
- integrating a new input element (`integrator()`)
- combining two states into one (`combiner()`)
- performing an optional final operation (`finisher()`)

See openjdk.org/jeps/461

End of Key Composing Operators in the Flowable Class

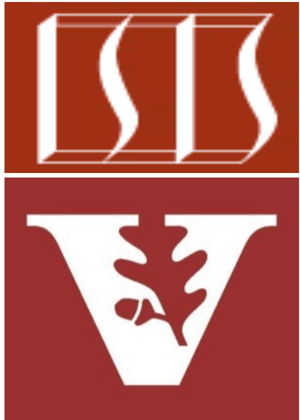
Applying Key Operators in the Flowable Class: Case Study ex5

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



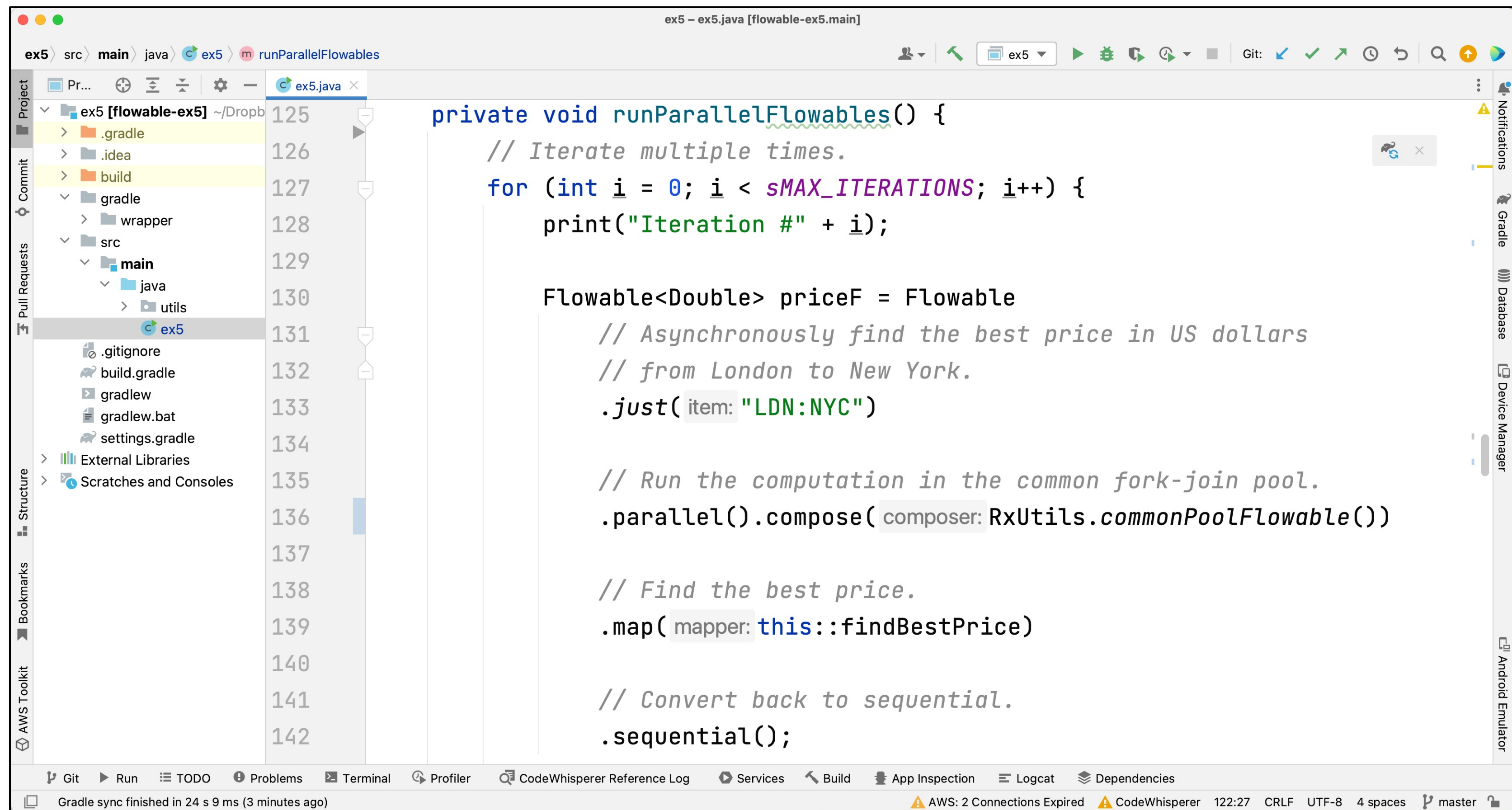
Learning Objectives in this Part of the Lesson

- Case study ex5 shows how to apply timeouts & compose() with the async Single & ParallelFlowable classes in the RxJava framework

```
var rateF = Flowable
   .just("LDN:NYC")
   .parallel()
   .compose(RxUtils
              .commonPoolFlowable())
   .map(this::findBestPrice)
   .sequential()
   .timeout(2,
              TimeUnit.SECONDS,
              sDEFAULT_RATE_F);
```

Applying Key Operators in the Flowable Class to ex5

Applying Key Operators in the Flowable Class to ex5



See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/Flowable/ex5

End of Applying Key Operators in the Flowable Class: Case Study ex5