Key Terminal Operators in the Observable Class (Part 1)

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

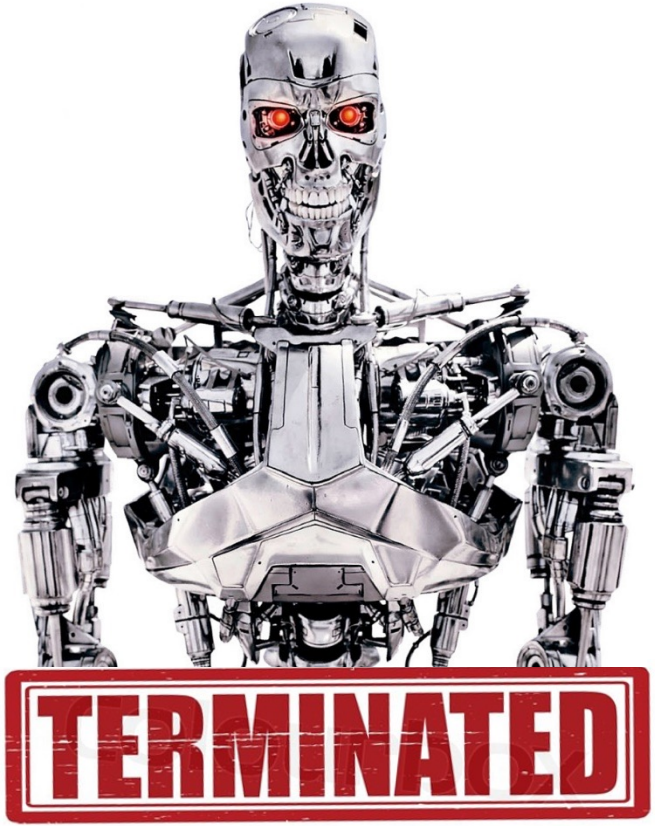
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize key Observable operators
  - 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

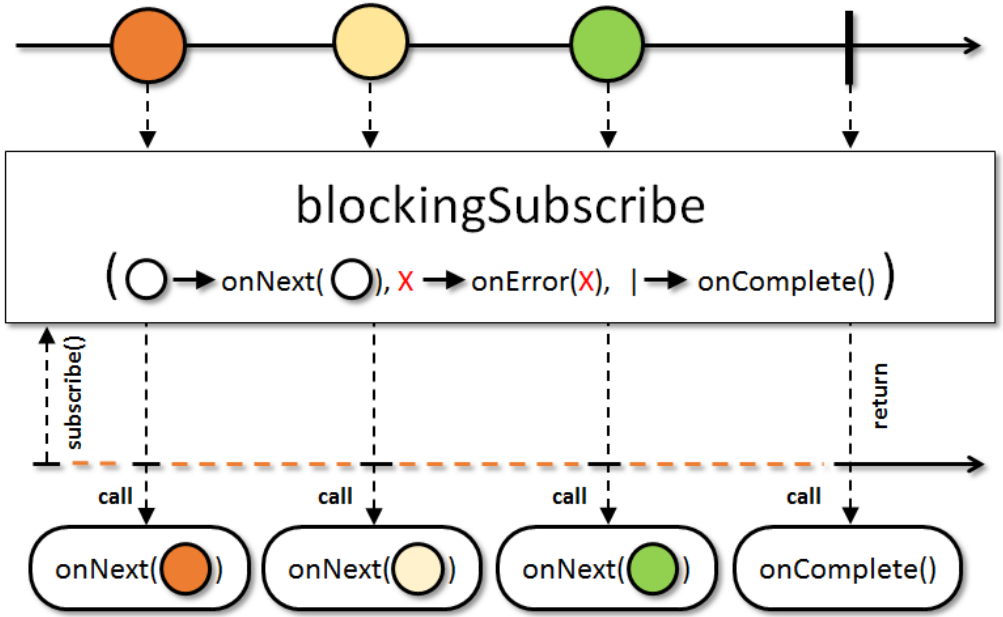
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests "unbounded demand"
        - i.e., Long.MAX_VALUE

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

Key Terminal Operators in the Observable Class

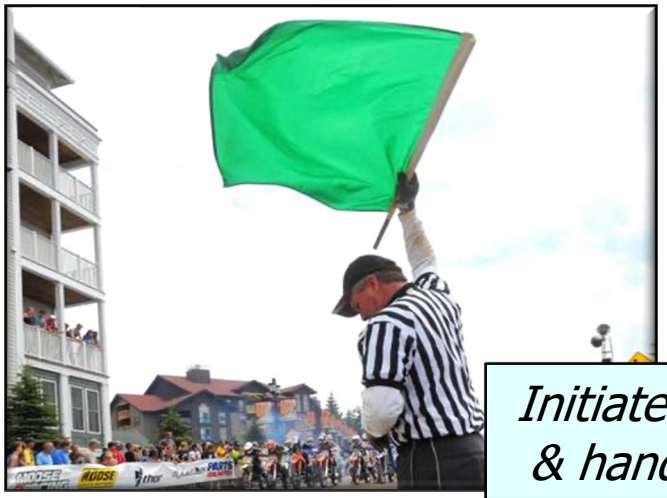
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
    - The params consume all elements in the sequence, handle errors, & react to completion
      - This subscription requests "unbounded demand"
    - Signals emitted to this operator are represented by the following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



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

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

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

Initiate processing & handle outputs

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

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

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

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

   .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

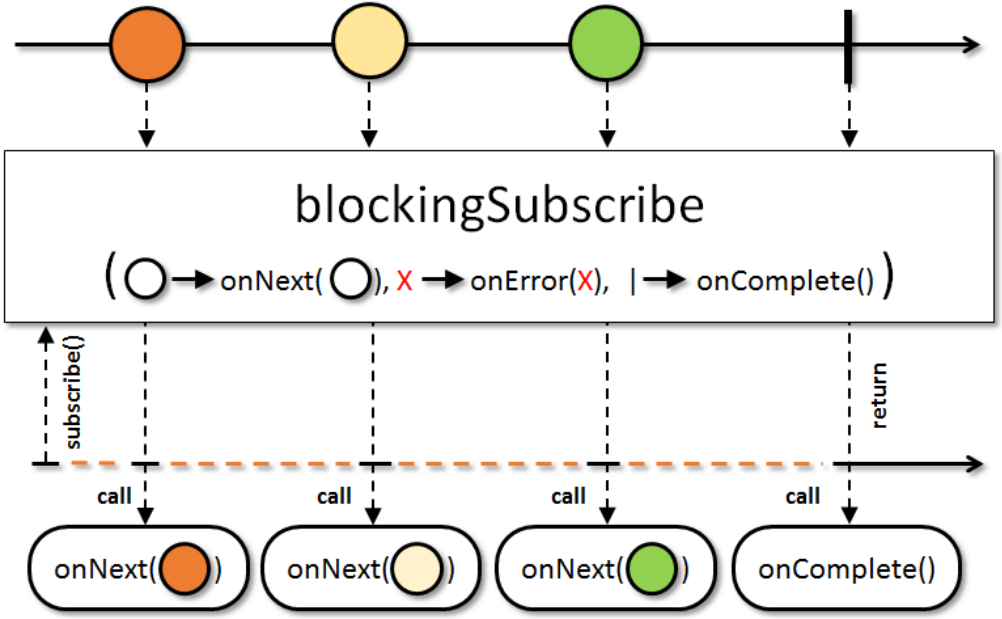
- The blockingSubscribe() operator
  - Subscribe Consumers & a Runnable to this Observable
  - This operator triggers all the processing in a chain

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

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

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

Completion Processing

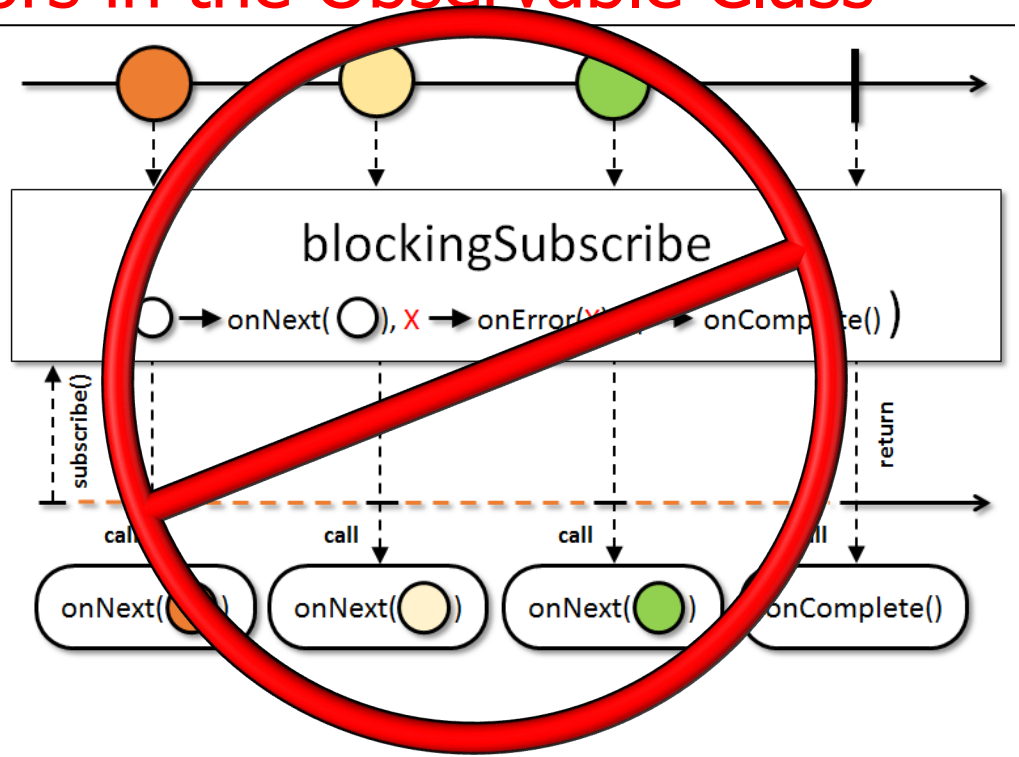
Key Terminal Operators in the Observable Class

- 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)