

Key Action Operators in the Flux 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 Flux operators
 - Concurrency & scheduler operators
 - Factory method operators
- Action operators
 - These operators don't modify a Flux, but instead use it for side effects
 - i.e., `doFinally()` & `doOnComplete()`



Key Action Operators in the Flux Class

Key Action Operators in the Flux Class

- The doFinally() operator
 - Add a behavior triggered after the Flux terminates for any reason

```
Flux<T> doFinally  
    (Consumer<SignalType> onFinally)
```

Key Action Operators in the Flux Class

- The doFinally() operator
 - Add a behavior triggered after the Flux terminates for any reason
 - The param is called when the Flux signals onError() or onComplete() or is disposed by the downstream

```
Flux<T> doFinally  
(Consumer<SignalType> onFinally)
```

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 docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Key Action Operators in the Flux Class

- The doFinally() operator
 - Add a behavior triggered after the Flux terminates for any reason
 - The param is called when the Flux signals onError() or onComplete() or is disposed by the downstream
 - It is a “callback” that only has side-effects

```
Flux<T> doFinally  
(Consumer<SignalType> onFinally)
```

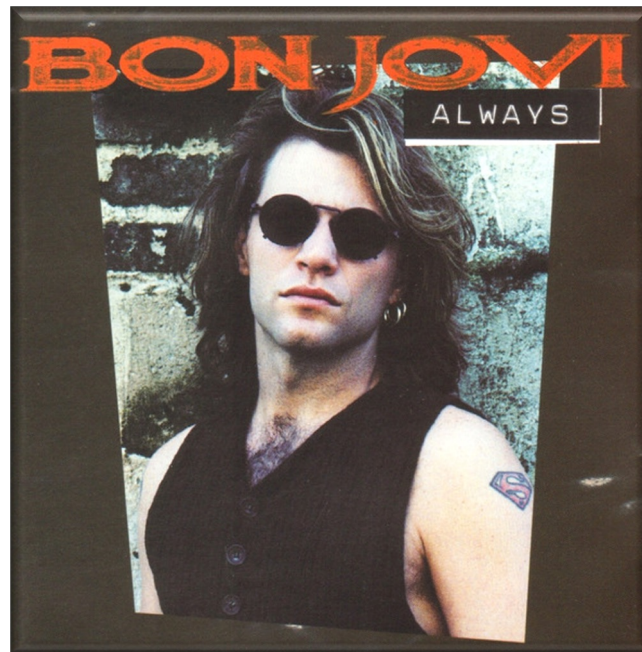


See [en.wikipedia.org/wiki/Callback_\(computer_programming\)](https://en.wikipedia.org/wiki/Callback_(computer_programming))

Key Action Operators in the Flux Class

- The doFinally() operator
 - Add a behavior triggered after the Flux terminates for any reason
 - The param is called when the Flux signals onError() or onComplete() or is disposed by the downstream
 - It is a “callback” that only has side-effects
 - Action is always called regardless of successful or error completion
 - Similar to a C++ destructor

```
Flux<T> doFinally  
(Consumer<SignalType> onFinally)
```



Contrast this doFinally() behavior with the doOnComplete() behavior

Key Action Operators in the Flux Class

- The doFinally() operator
- Add a behavior triggered after the Flux terminates for any reason
 - The param is called when the Flux signals onError() or onComplete() or is disposed by the downstream
- Returns the new unchanged Flux instance

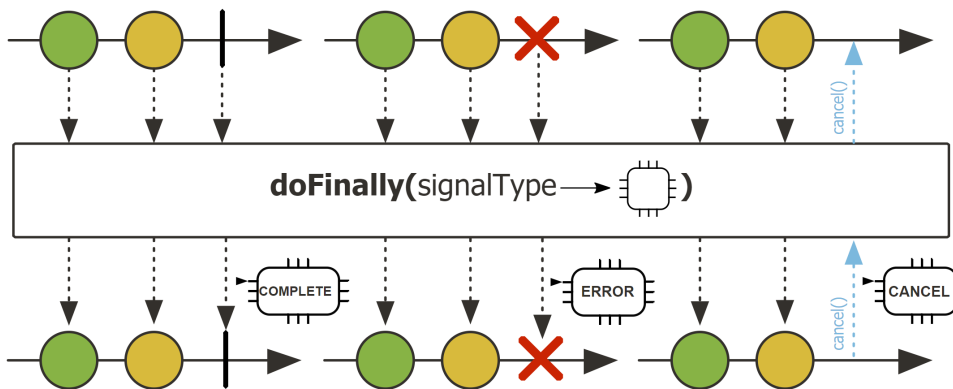
```
Flux<T> doFinally  
    (Consumer<SignalType> onFinally)
```



*The type or the value of elements
that is processed is unchanged*

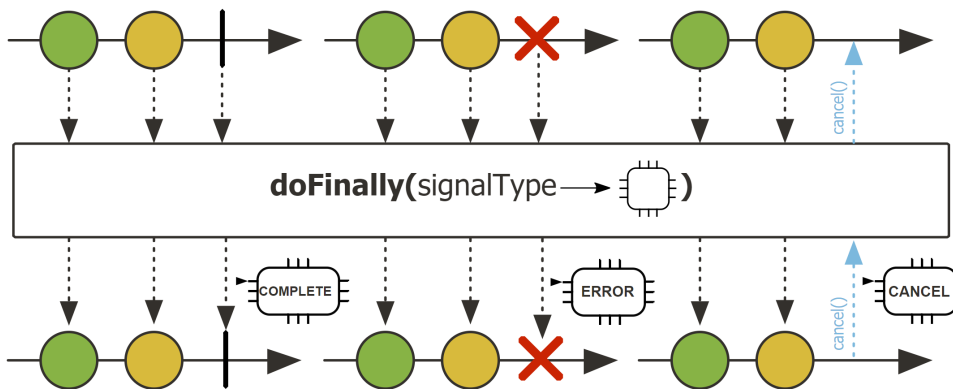
Key Action Operators in the Flux Class

- The `doFinally()` operator
 - Add a behavior triggered after the Flux terminates for any reason
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler



Key Action Operators in the Flux Class

- The `doFinally()` operator
 - Add a behavior triggered after the Flux terminates for any reason
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler



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

```
    .create(makeAsyncFluxSink())
```

```
    ...
```

```
    .publishOn(subscriber)
```

```
    ...
```

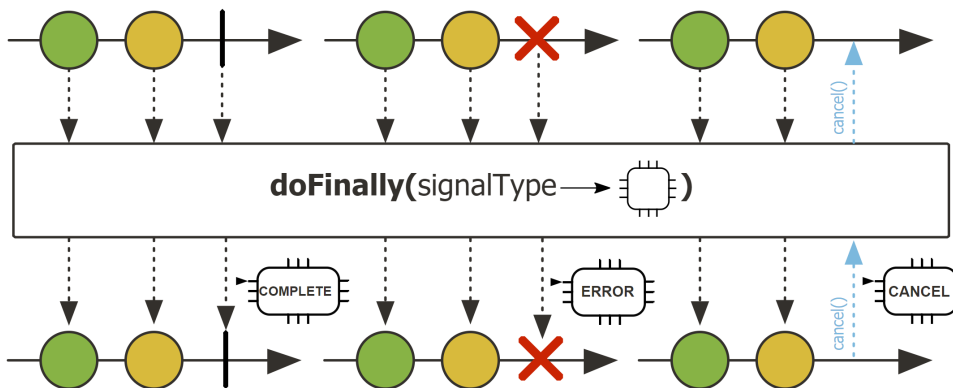
```
    .doFinally(____ -> subscriber.dispose()) ...
```

*This operator is called
after the Flux completes*

See [Reactive/flux/ex2/src/main/java/FluxEx.java](https://github.com/reactive/reactive-streams-examples/blob/master/flux-examples/src/main/java/FluxEx.java)

Key Action Operators in the Flux Class

- The `doFinally()` operator
 - Add a behavior triggered after the Flux terminates for any reason
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler



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

```
    .create(makeAsyncFluxSink())
```

```
    ...
```

```
    .publishOn(subscriber)
```

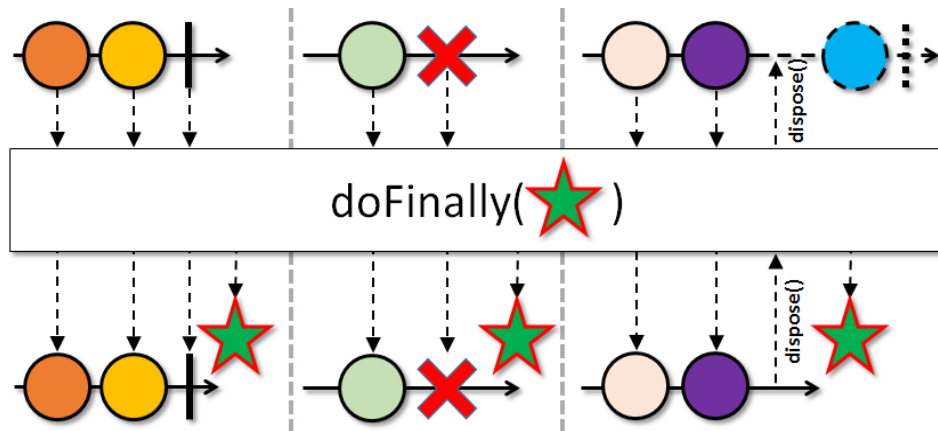
```
    ...
```

```
    .doFinally(____ -> subscriber.dispose()) ...
```

*This callback disposes
the subscriber thread*

Key Action Operators in the Flux Class

- The `doFinally()` operator
 - Add a behavior triggered after the Flux terminates for any reason
 - Does not operate by default on a particular Scheduler
- RxJava's operator `Observable.doFinally()` works the same



`Observable`

```
.create(ObservableEx::emitAsync)
.observeOn(Schedulers.newThread()) ...
.doFinally(() -> BigFractionUtils.display(sb.toString()))
```

Print BigIntegers to aid debugging

Key Action Operators in the Flux Class

- The `doFinally()` operator
 - Add a behavior triggered after the Flux terminates for any reason
 - Does not operate by default on a particular Scheduler
 - RxJava's operator `Observable.doFinally()` works the same
- The Java Streams framework has no operations like `doFinally()`
 - Any cleanup can be done after the stream's terminal operation completes synchronously

Interface `Stream<T>`

Type Parameters:

T - the type of the stream elements

All Superinterfaces:

`AutoCloseable`, `BaseStream<T,Stream<T>>`

```
public interface Stream<T>
extends BaseStream<T,Stream<T>>
```

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using `Stream` and `IntStream`:

```
int sum = widgets.stream()
    .filter(w -> w.getColor() == RED)
    .mapToInt(w -> w.getWeight())
    .sum();
```

In this example, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to `Stream`, which is a stream of object references, there are primitive specializations for `IntStream`, `LongStream`, and `DoubleStream`, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

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

Key Action Operators in the Flux Class

- The doOnComplete() operator
 - Add behavior triggered when the Flux completes successfully

`Flux<T> doOnComplete`
`(Runnable onComplete)`

Key Action Operators in the Flux Class

- The doOnComplete() operator
 - Add behavior triggered when the Flux completes successfully
 - The parameter is called when the Flux signals onComplete()
 - Runnable is a functional interface

Flux<T> doOnComplete
(Runnable onComplete)

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

Functional Interface:

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

```
@FunctionalInterface
```

```
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

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

Key Action Operators in the Flux Class

- The doOnComplete() operator
 - Add behavior triggered when the Flux completes successfully
 - The parameter is called when the Flux signals onComplete()
 - Runnable is a functional interface
 - i.e., it's a callback that only has side-effects

```
Flux<T> doOnComplete  
    (Runnable onComplete)
```



See [en.wikipedia.org/wiki/Callback_\(computer_programming\)](https://en.wikipedia.org/wiki/Callback_(computer_programming))

Key Action Operators in the Flux Class

- The doOnComplete() operator
 - Add behavior triggered when the Flux completes successfully
 - The parameter is called when the Flux signals onComplete()
 - Runnable is a functional interface
 - onComplete() is only called on successful completion, but not when errors occur

`Flux<T> doOnComplete`
`(Runnable onComplete)`



Contrast this doOnComplete() behavior with the doFinally() behavior

Key Action Operators in the Flux Class

- The doOnComplete() operator
 - Add behavior triggered when the Flux completes successfully
 - The parameter is called when the Flux signals onComplete()
 - Returns the new Flux instance

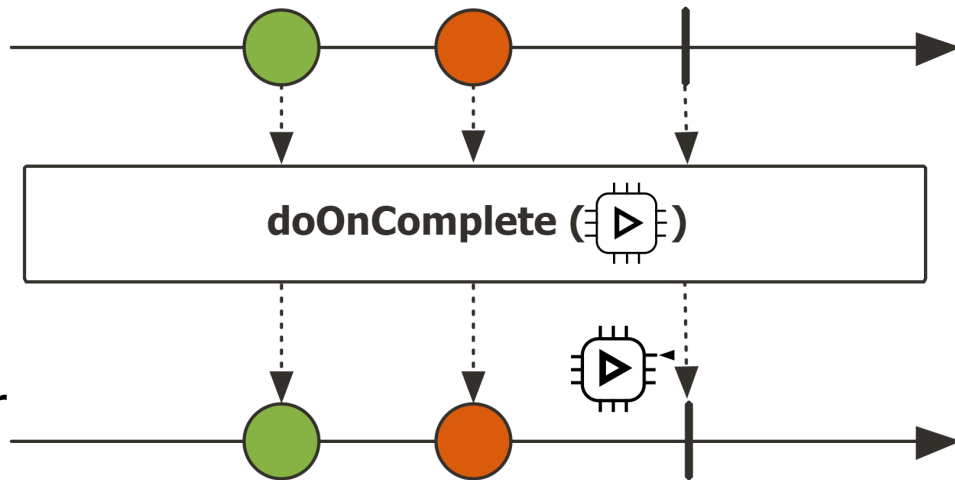
Flux<T> doOnComplete
(Runnable onComplete)



Can't change the type or the value of elements it processes

Key Action Operators in the Flux Class

- The `doOnComplete()` operator
 - Add behavior triggered when the Flux completes successfully
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler



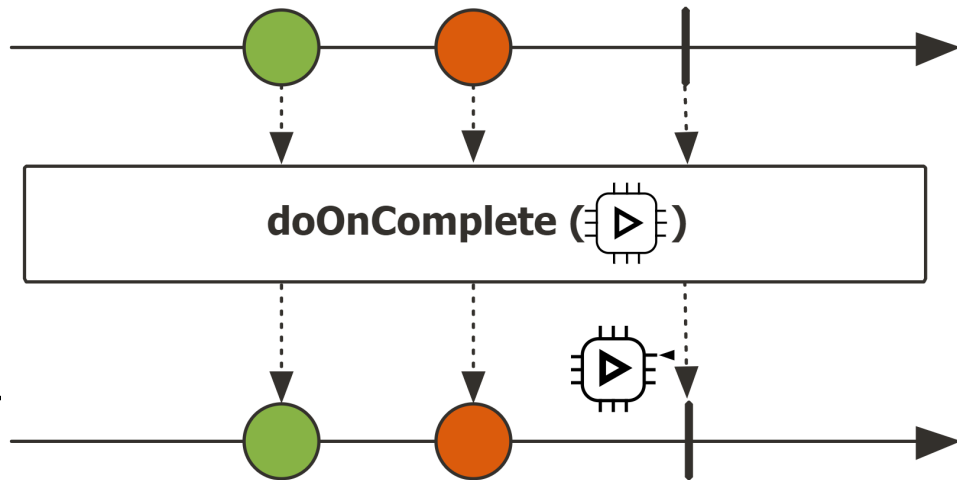
Key Action Operators in the Flux Class

- The `doOnComplete()` operator
 - Add behavior triggered when the Flux completes successfully
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler

Flux

```
.create(makeAsyncFluxSink())  
...  
.map(bigInt -> FluxEx.checkIfPrime(bigInt, sb))  
.doOnComplete(() -> BigFractionUtils.display(sb.toString()))  
...
```

Print BigIntegers when Flux stream completes successfully



See [Reactive/flux/ex2/src/main/java/FluxEx.java](https://github.com/reactive/reactive-examples/blob/master/flux/ex2/src/main/java/FluxEx.java)

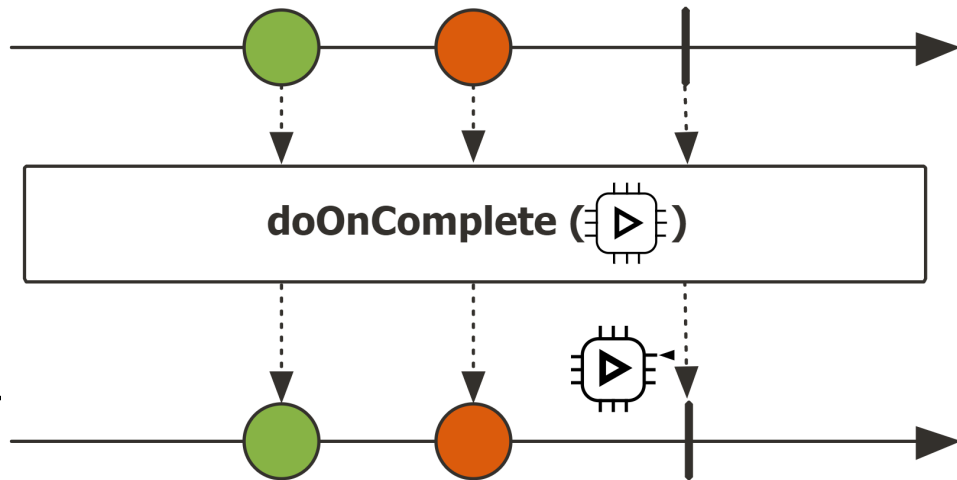
Key Action Operators in the Flux Class

- The `doOnComplete()` operator
 - Add behavior triggered when the Flux completes successfully
 - Does not operate by default on a particular Scheduler
 - i.e., it uses the current scheduler

Flux

```
.create(makeAsyncFluxSink())  
...  
.map(bigInt -> FluxEx.checkIfPrime(bigInt, sb))  
.doOnComplete(() -> BigFractionUtils.display(sb.toString()))  
...
```

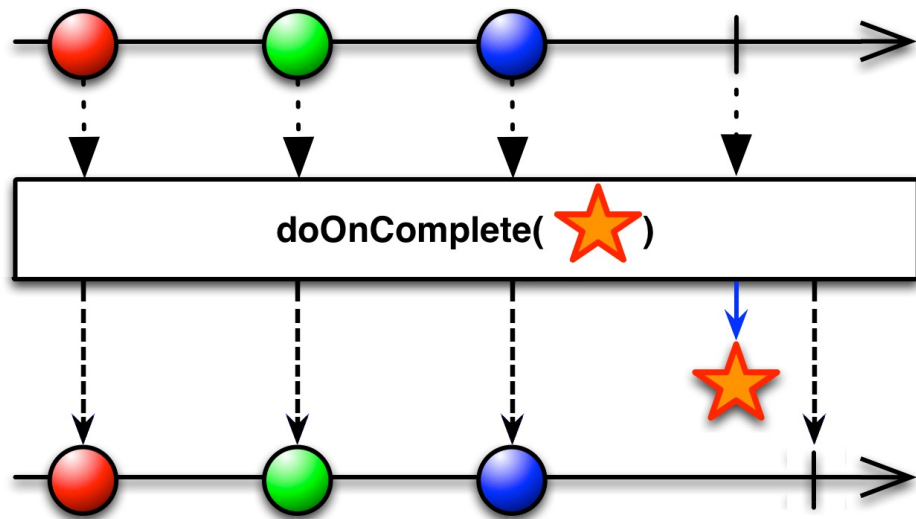
Only a "side-effect"



See [Reactive/flux/ex2/src/main/java/FluxEx.java](https://github.com/reactive/flux-ex2/src/main/java/FluxEx.java)

Key Action Operators in the Flux Class

- The `doOnComplete()` operator
 - Add behavior triggered when the Flux completes successfully
 - Does not operate by default on a particular Scheduler
- RxJava's Observable `doOnComplete()` operator works the same



Observable

```
.create(ObservableEx::emitInterval)
.map(bigInt -> ObservableEx.checkIfPrime(bigInt, sb))
.doOnComplete(() -> BigFractionUtils.display(sb.toString()))
...
```

Print BigIntegers when Observable stream completes successfully

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

End of Key Action Operators in the Flux Class (Part 2)