11. Introducing Reactor

-As develop application code, we can write 2 styles of code-imperative and reactive:

+**Imperative**: like absurd hypothetical newspaper subscription. It’s a serial set of tasks, each running one at a time, each after the previous task. Data is processed in bulk and can’t be handed over to the next task until the previous task has completed its work on the bulk of data.

+**Reactive**: like a real newspaper subscription. A set of tasks is defined to process data, but those tasks can run in parallel. Each task can process subsets of the data, handing it off to the next task in lined while it continues to work on another subset of the data.

# 11.1 Understanding reactive programming

-**Imperative programming** is intuitive enough that young students are learning it with ease in their school’s STEM program, it’s powerful enough that it makes up the bulk of code that drives the largest enterprises.

+The idea is simple: you write code as a list of instruction to be followed, one at a time, in the order that they’re encountered. A task is performed and the program waits for it to complete before moving on to the next task.

+If it’s **I/O tasks**: writing data to database, fetching data from a remote server, the thread that invoked that task is blocked, unable to do anything else until the task completes -> **wasteful**

+Java support **concurrent programming**: managing concurrency in multiple threads is challenging.

-Reactive programming is functional and declarative in nature. It involves describing a pipeline or stream through which data flows. It processes data as it becomes available. In fact, the incoming data may be endless.

-Consider imperative programming as a water balloon and reactive programming as a garden hose.

## 11.1.1 Defining Reactive Streams

-Reactive is an initiative started in late 2013. Reactive Stream aims to provide a **standard** for **asynchronous stream processing** with **nonblocking backpressure** (which consumers of data can avoid being overwhelmed by an overly fast data source by establishing **limits** on how much they’re willing to **handle**).

-**Java Streams** vs **Reactive Streams**

+Java stream: synchronous and work with a finite set of data. A means of iterating over a collection with function

+Reactive stream: support asynchronous processing of datasets of any size, including infinite datasets. They process data in real time, as it become available, with backpressure to avoid overwhelming their consumers

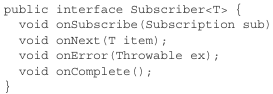
-RS specification 4 interfaces: **Publisher, Subscriber, Subscription, Processor**.

+ A **Publisher** produces data that it sends to a Subscriber per a Subscription.

+**subscribe()** of Publisher: through which a Subscriber can subscribe to Publisher



+Once a Subscriber has subscribed, it can **receive** **events** from **Publisher**. Those events are sent via methods on Subscriber:



When Publisher calls **onSubscribe()**: passes a Subscription object to Subscriber.

+Through Subsription, Subscriber can manage its subscription:



Subscriber call **request()** to request that data be sent, or call cancel to indicate that it’s no longer interested in receiving data and is canceling the subscription. Long n is how many data items Subscriber willing to accept ->backpressure: prevent Publisher sending more data than Subscriber able to handl.

+Once Subscriber has requested data, the data starts flowing through the stream. For every item that’s published by Publisher, **onNext()** called to deliver data to Subscriber. If error -> **onError().** If Publisher() has no more data->**onComplete()** to tell Subscriber that’s end.

+**Processor**: Subscriber+Publisher. As a Subscriber, a Processor will receive data and process it in some way. Then it will switch hats and act as a Publisher to publish the results to its Subscribers.



->Data processing pipeline: Start with a Publisher, pumps data through zero or more Processors, and then drop up the final result off to Subscriber.

# 11.2 Getting started with Reactor

-Reactive programming means building a pipeline through which data will follow.

-Example: take person name, change to uppercase, create a greeting message, print it.

+Imperative programming:



Each line performs a step in same thread, each step blocks the executing thread from moving to next step until complete.

+Reactive code:

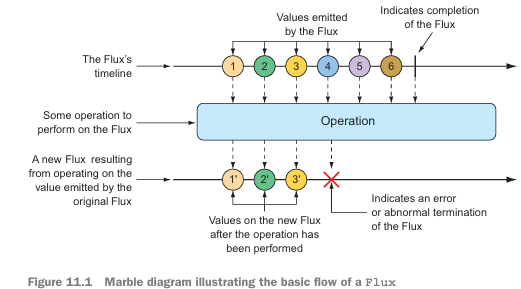


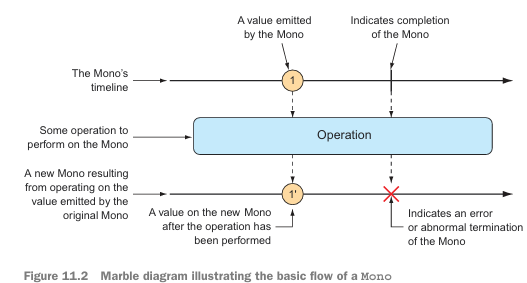
It’s a pipeline that data flow through: At each phase, the data is tweaked somehow, but no assumption can be made about which thread any of the operations are performed one. They may be the same thread or not.

-Reactor’s 2 core types: Both are Publisher implementations  
+**Flux**: pipeline of zero, one, or many data items

+**Mono**: optimized for when the dataset is known to have no more than one data item.

## 11.2.1 Diagraming reactive flows



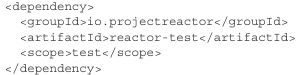


## 11.2.2 Adding Reactor dependencies

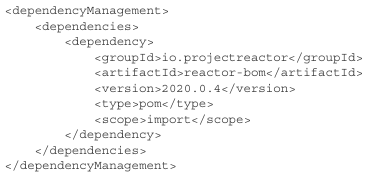
-Add Reactor:



-Reactor also provides testing support. We going to write a lot of tests around Reactor code:



-In Spring Boot project, it handles dependency management so there’s no need to specify <version>. If you want to use Reactor in non-SB project, set up Reactor’s BOM in the build:

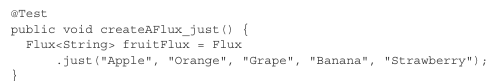


# 11.3 Applying common reactive operations

-Flux and Mono offer more 500 operations, which can categorized: creation, combination, transformation, logic

## 11.3.1 Creating reactive types

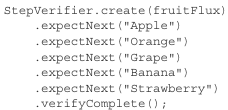
-Creating from objects:



+Add a subscriber:

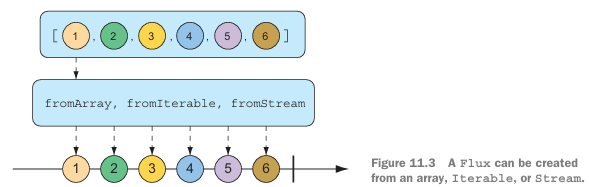


+A better way to test Flux or Mono is use **StepVerifier**. This subscribes to reactive type and then applies assertions against the data as it flows through the stream, finally verifying that the stream completes as expected:

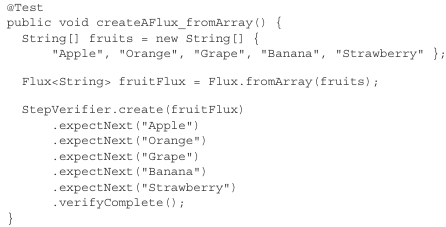


StepVerifier subscribes to Flux and then asserts each item matches the expected fruit name. Finally, it verifies that after Strawberry is produced by Flux, Flux is complete

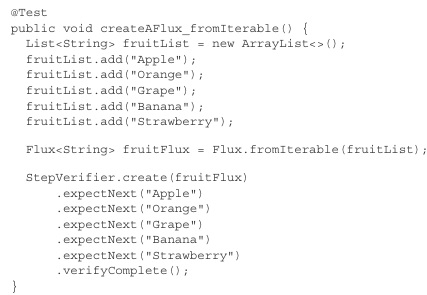
-Creating from Collections:



-fromAray():

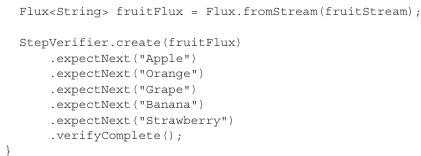


-fromIterable():



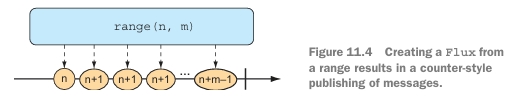
-fromStream():

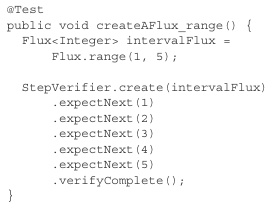




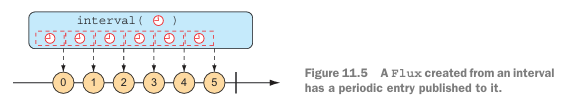
-Generating Flux Data:

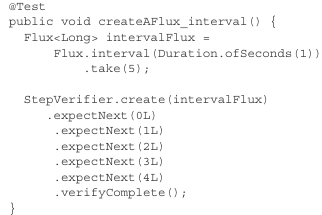
+range()





+iteral()

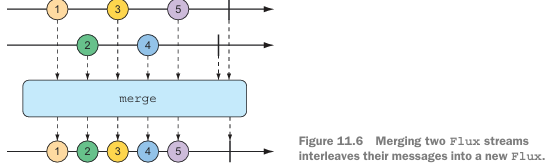


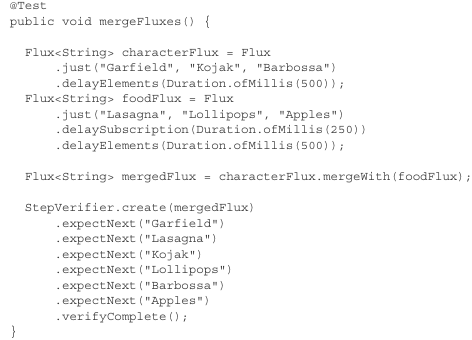


## 11.3.2 Combing reactive types

-Mergeing Reactive types

+mergeWith()



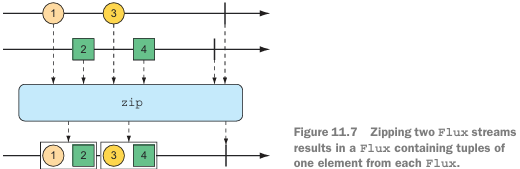


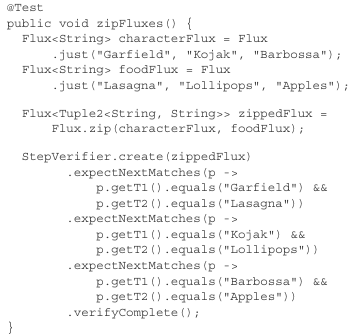
**delayElement()**: slow streams down a little-emitting an entry only every … ms.

**delaySubscription()**: it won’t emit any data until … ms have passed following a subscription

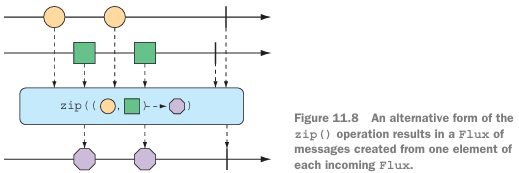
The order of items aligns with the timing of how they’re emitted from sources.

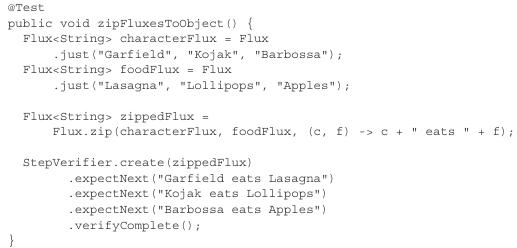
+zip():



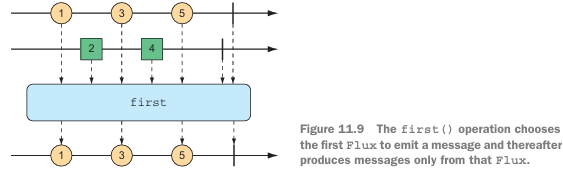


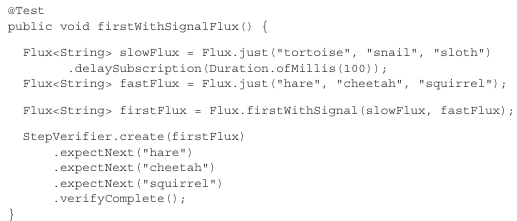
+Function to zip():





-Selecting the 1st Reactive Type to publish:

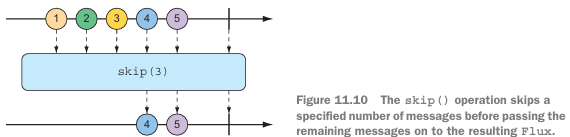


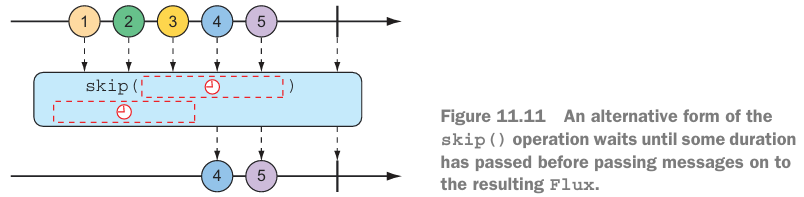
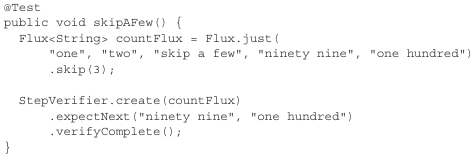


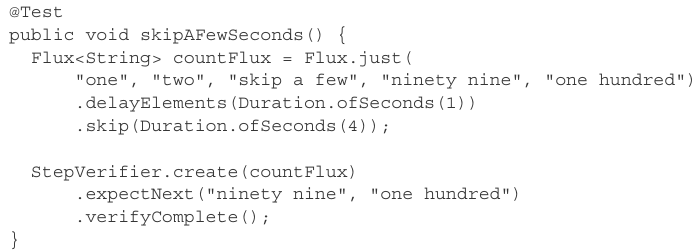
## 11.3.3 Transforming and filtering reactive streams

-Filtering data from Reactive Types:

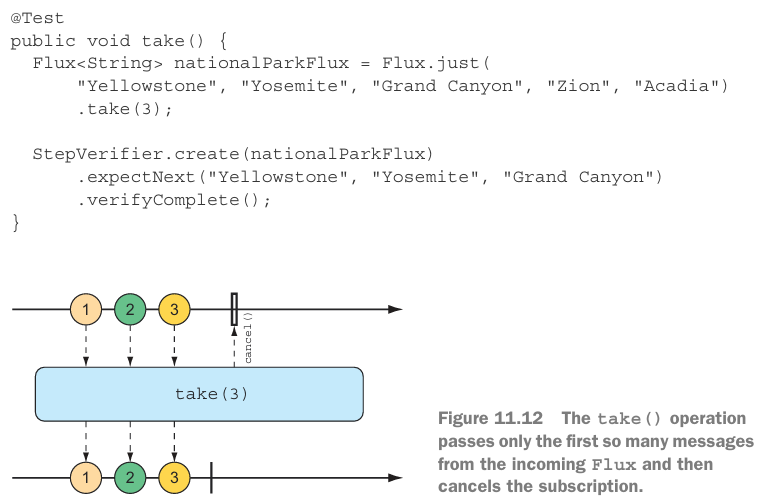
+skip():

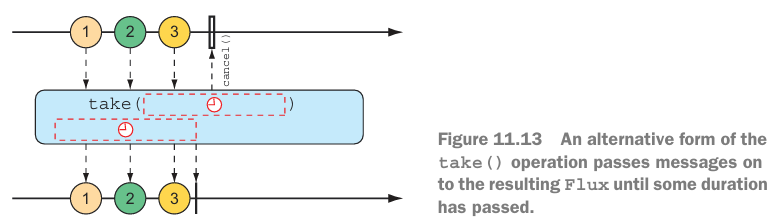




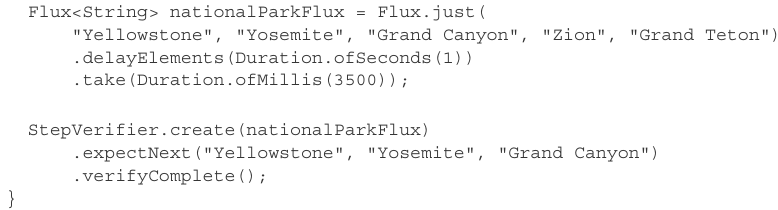


+take():

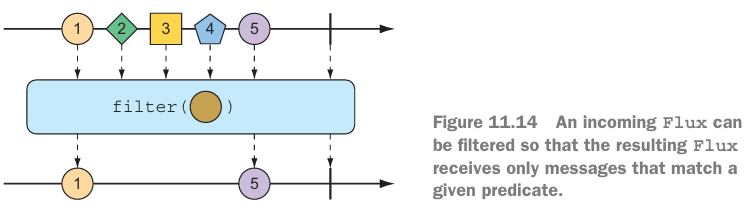


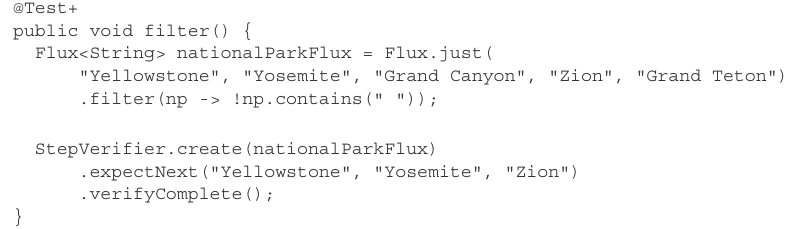




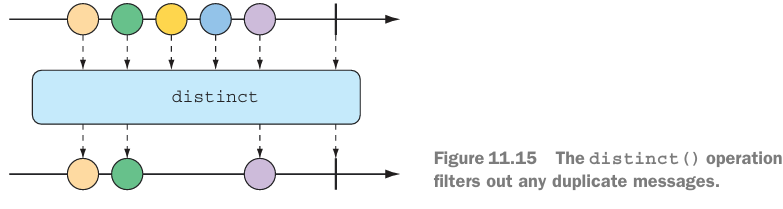


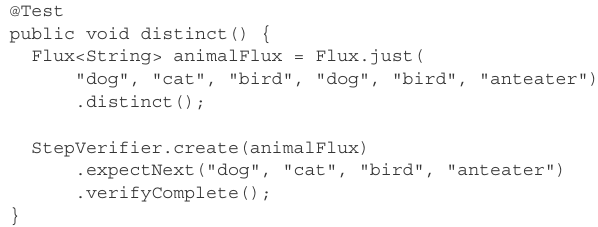
+filter()





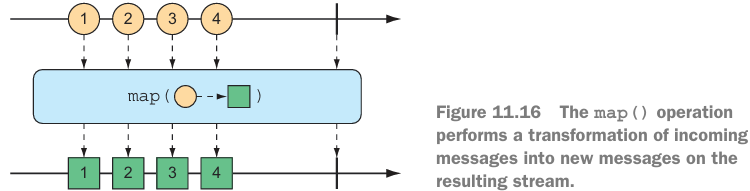
+distinct():



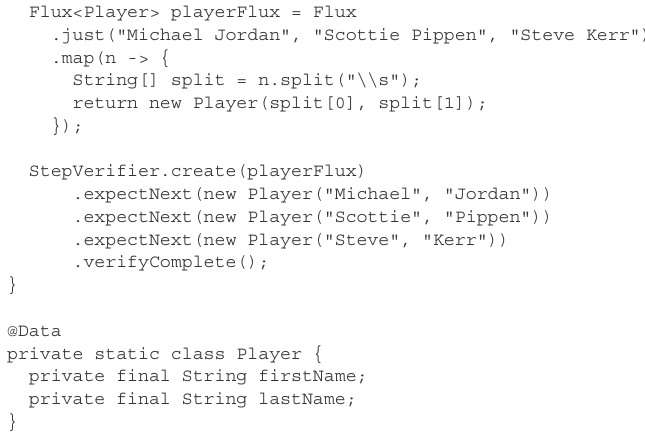


-Mapping reactive data:

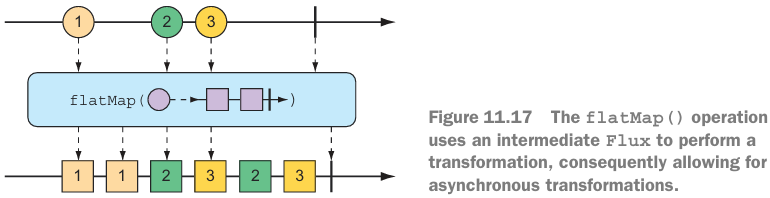
+map():

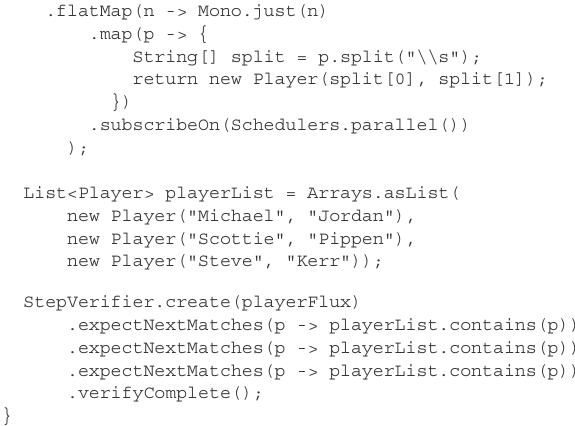
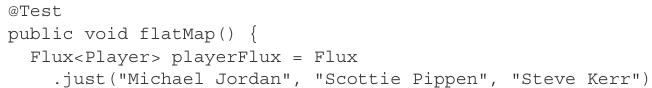




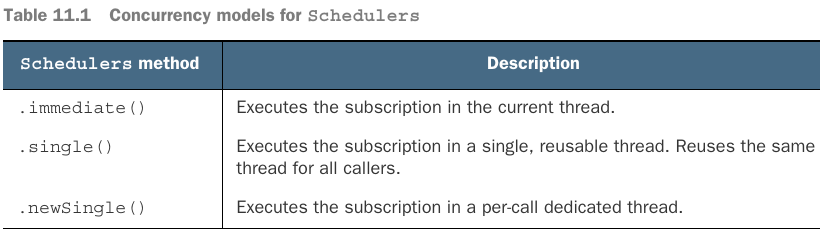


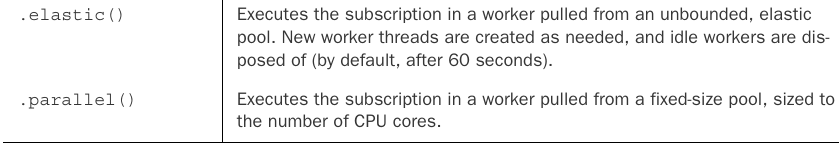
+flatMap():





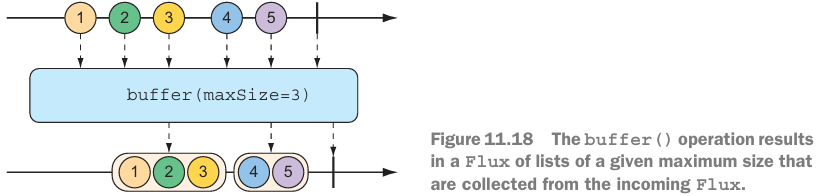
+subscribeOn(): each subscription should take place in a parallel thread. You can use static methods from Schedulers:

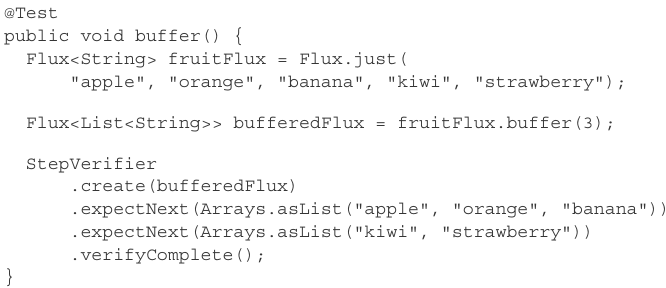




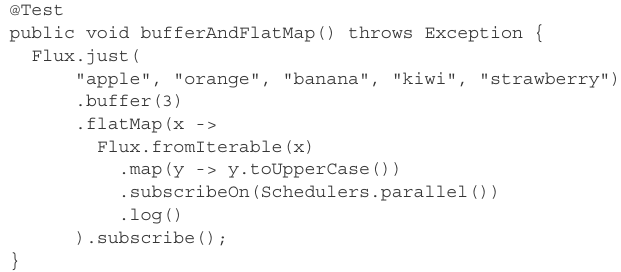
-Buffering data on a reactive stream

+buffer():





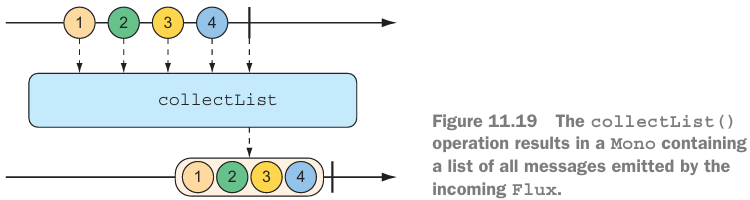
+buffer()+flatMap(): processed in parallel

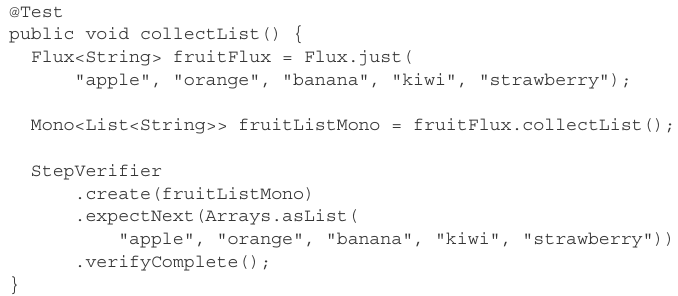


Collect everything in Flux emits into List:

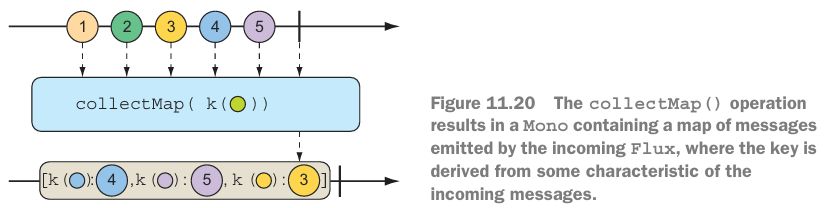


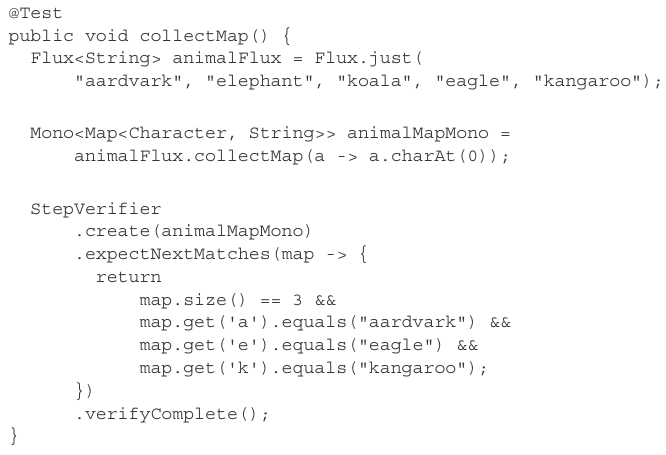
+collectList():





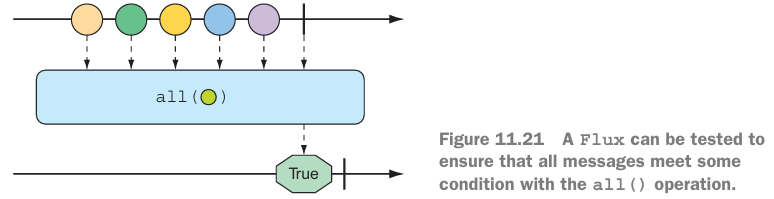
+collectMap():

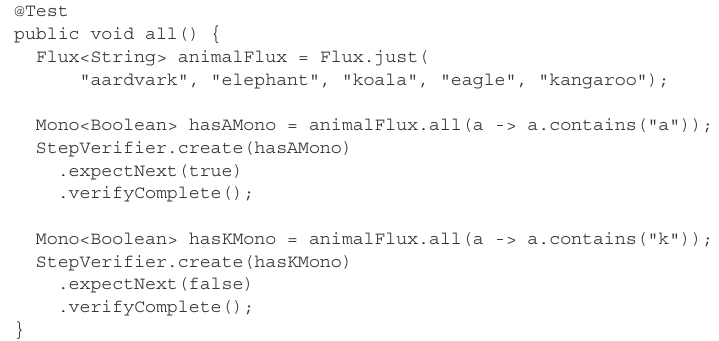




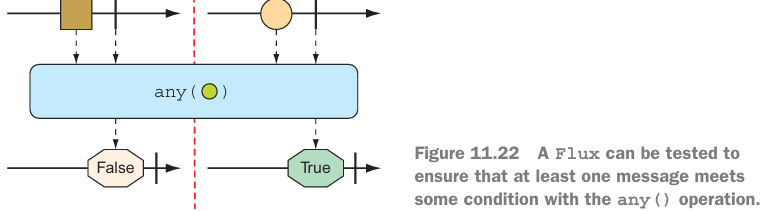
## 11.3.4 Performing logic operations on reactive types

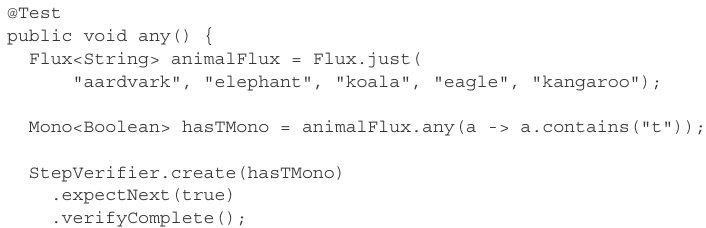
-all()

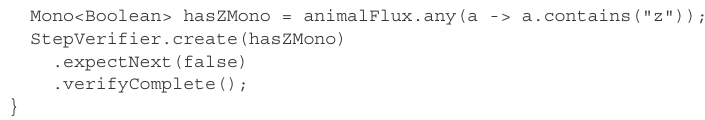




-any()







Summary

