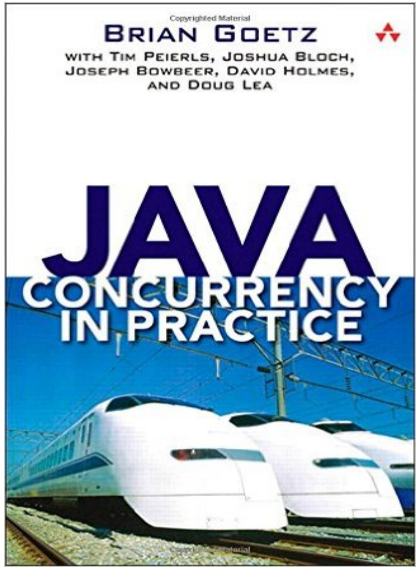
Motivation

Long Running Shift to Concurrency

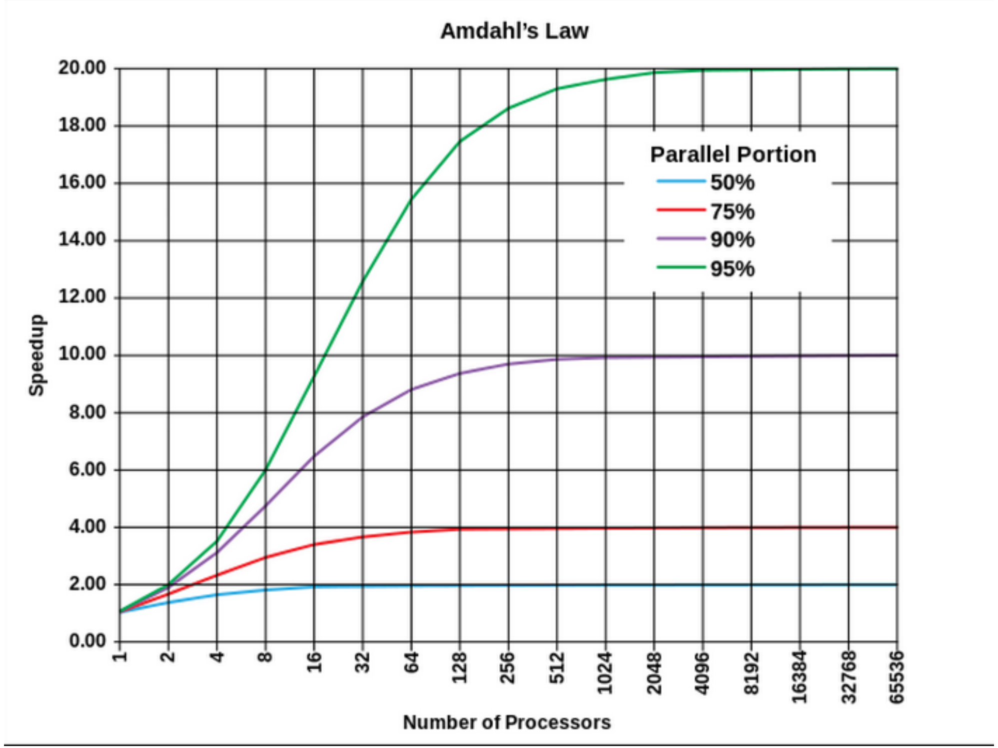
*For 30 years computer performance driven by **Moore's Law**; from now on, it will be driven by **Amdahl's Law**.*



Moore's Law is delivering more cores but not faster cores.

March 2006

Amdahl's Law



Theoretical max speedup using **multiple processors**?

Limited by the time needed for **sequential processing**.

Scale Up

- Remember SOA ? EJB 1 ?
  - distributed components, then services
  - **functional boundaries** between system components
  - location transparent resources **pooled**

- … but it didn’t do well
  - complex / expensive solutions
  - full-stack approach
  - overhead not factored in: **latency failure tolerance**

- Ended up in hype failure mockeries and hacker news rants

Internet Scale

- Social media became mainstream
  - massive scale

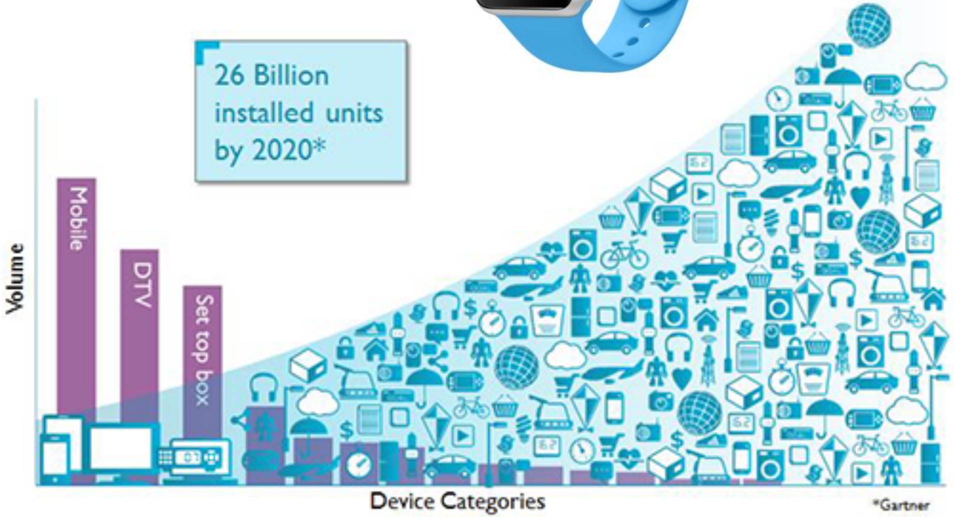
- *Background Processing* and pooled resources regain interest
  - everything **“write-heavy” -- candidate for async handling**
  - e.g. a tweet

- Push-based servers start to proliferate
  - async response handling (e.g. facebook messages)

Cloud Scale

- “Cloud-ready” systems began to spread around 2012
  - AWS, open-source PaaS democratization
- Tech giants expose internal distributed stack
  - Twitter, Google, Netflix…
- Tools to address core concerns of distributed systems
  - **concurrency**, **service discovery, monitoring**

Wait! Another traffic spike: IoT



Scale Out

- There are **limits to scale** the monolith way
  - cost efficiency aside

- **Memory overuse**
  - large thread pools, application footprint

- **CPU underuse**
  - blocking I/O (DB, remote service)

Tales of Time and Space

- Massive scale requires higher efficiency
  - respect physical laws
  - **speed of light dampens inter-service** communication

- Pooling resources simply not enough
  - **need to coordinate, compose, orchestrate data flows**

- **Non-blocking must be embraced** fundamentally

Non-Blocking Architecture

- Non-blocking requires profound change
  - **can’t write imperative code**
  - **can’t assume single thread** of control (e.g. exception handling)

- Must deal with async results
  - **listener/callbacks become unwieldy** with nested composition

- Everything becomes an event stream
  - e.g. from `Input/OutputStream` to **stream of events**

Reactive Programming

Generally speaking…

- **“Reactive”** is used broadly to define event-driven systems
  - UI events, network protocols

- Now also entering the domain of application/business logic

- **Non-blocking** event-driven architecture

Reactive Manifesto

- Reflects the emerging field of scalable, non-blocking applications
- A hark back to other successful manifestos
- Well articulated but very broad strokes
- Defines qualities of **reactive systems**
- For **reactive programming** we need concrete tools

Reactive Programming in Context

- To build reactive apps we need two essential building blocks

- **Contract for interop between non-blocking components**
  - Reactive Streams spec

- **API for composing asynchronous programs**
  - e.g. Reactive Extensions

Myths about “Reactive”

- Async / concurrent == reactive ?
  - easy to end up with too many threads
  - fill up hand-off queue

- Must be async to be reactive ?
  - nope but must be agnostic to source of concurrency

- Reactive is the domain of specific programming languages ?
  - language features can help
  - e.g. JDK 8 lambda

Reactive Streams Specification for the JVM

Specification Goals

- Govern the exchange of data **across async boundaries**
- **Use back-pressure** for flow control
- Fast sources should not overwhelm slower consumers
- Provide **API** types, **TCK**

Back-Pressure

- When publisher maintains **higher rate for extended time**
  - **queues grow** without bounds

- Back-pressure allows a subscriber to **control queue bounds**
  - subscriber **requests #** of elements
  - publisher produces **up to #** of requested

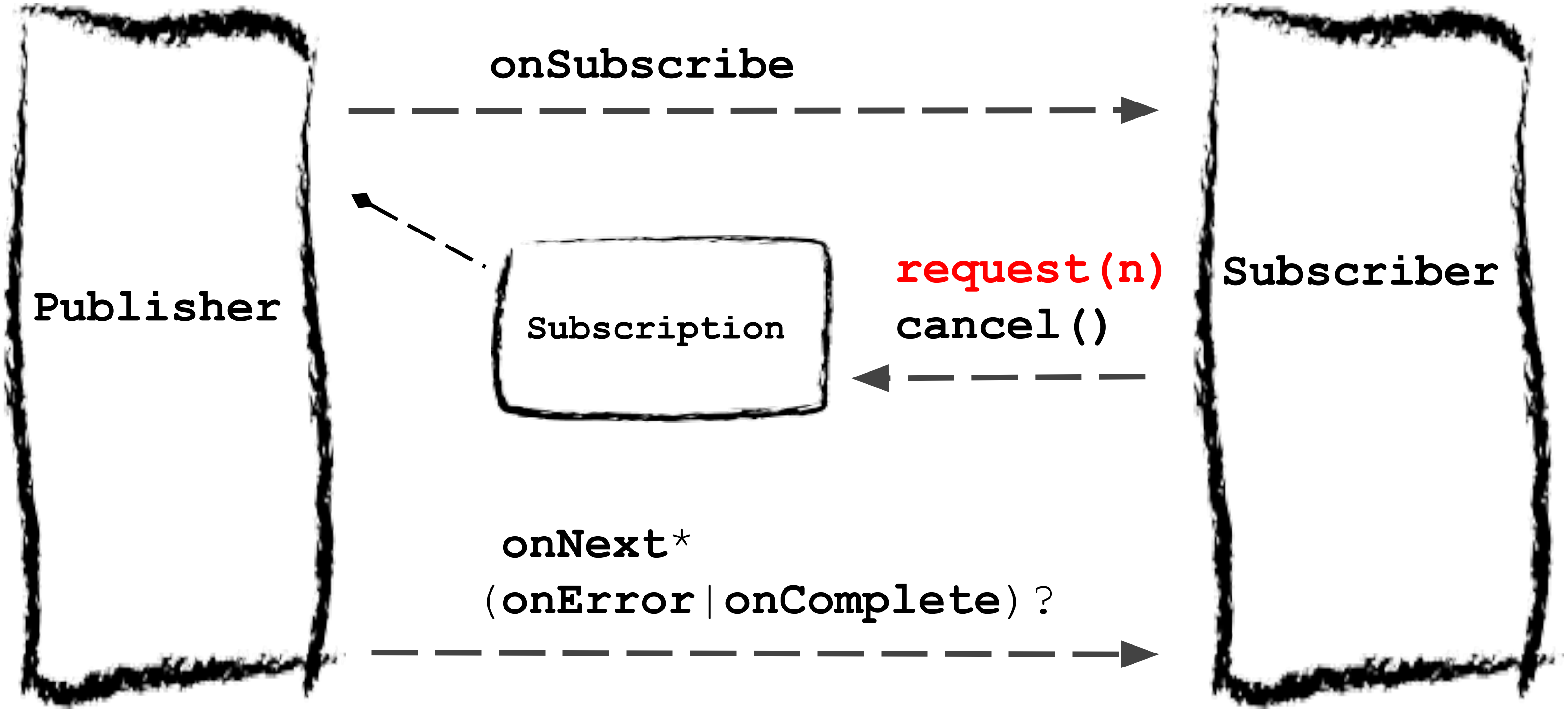
- If source can’t be controlled (e.g. mouse movement clock tick)
  - publisher may buffer or drop
  - must obey # of requested elements

API Types

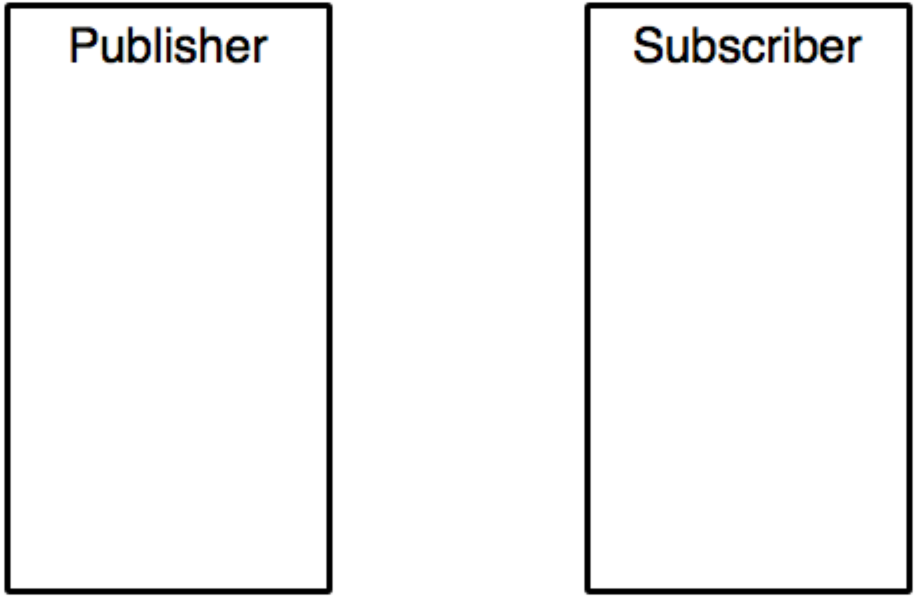
```
public interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
  void onSubscribe(Subscription s);
  void onNext(T t);
  void onError(Throwable t);
  void onComplete();
}

public interface Subscription {
  void request(long n);
  void cancel();
}

public interface Processor<T R> extends Subscriber<T> Publisher<R> {
}
```

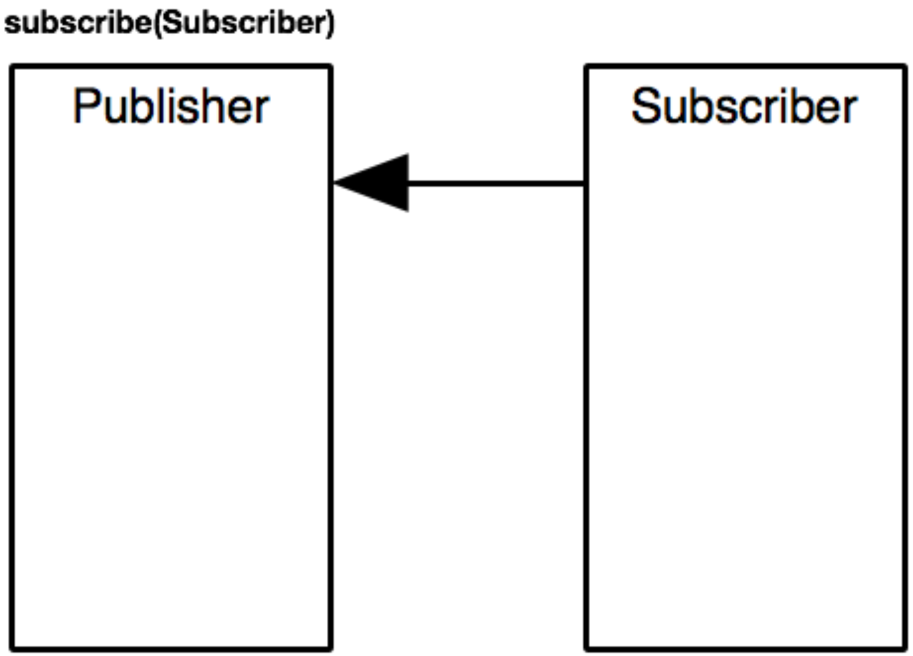
All are `void` methods…
one-way, message style

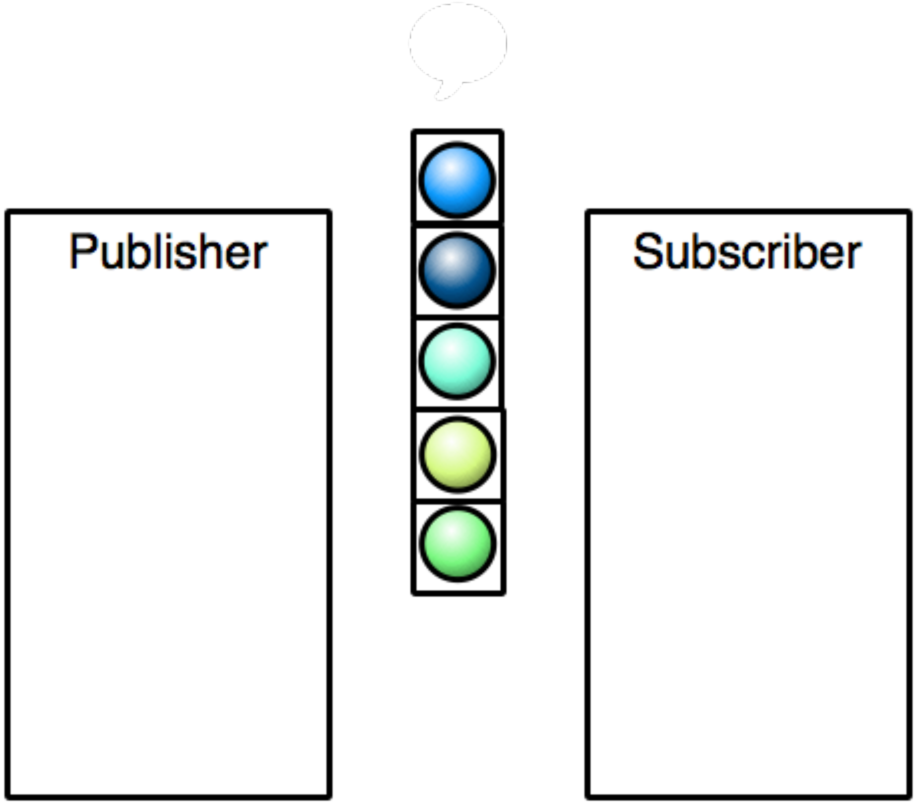
Reactive Streams API

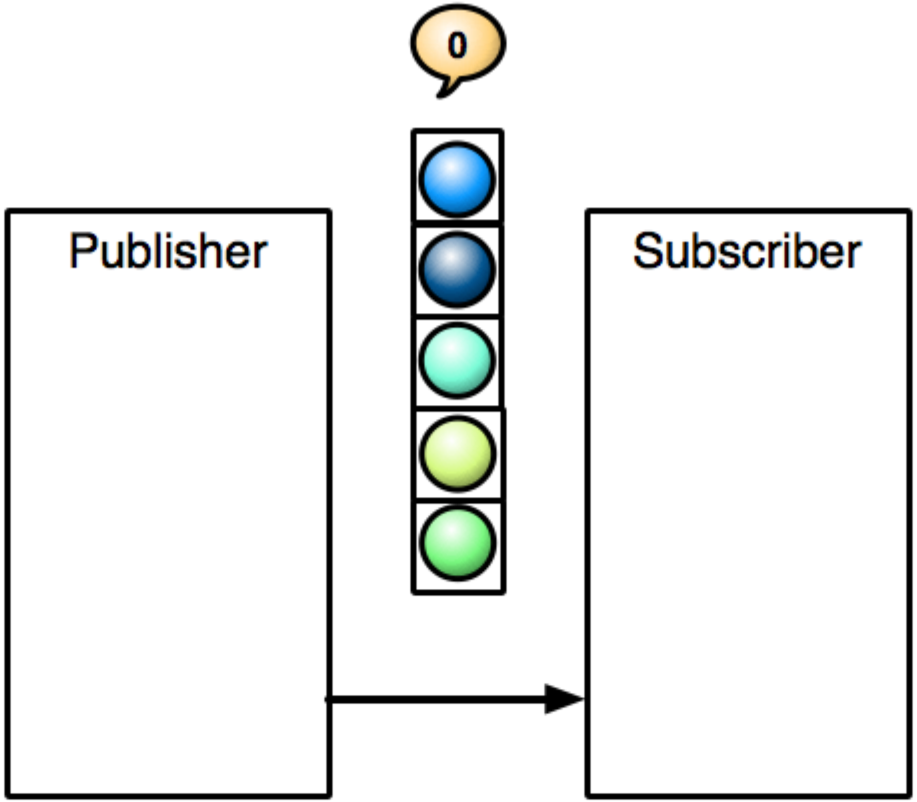


Reactive Streams: Message Passing

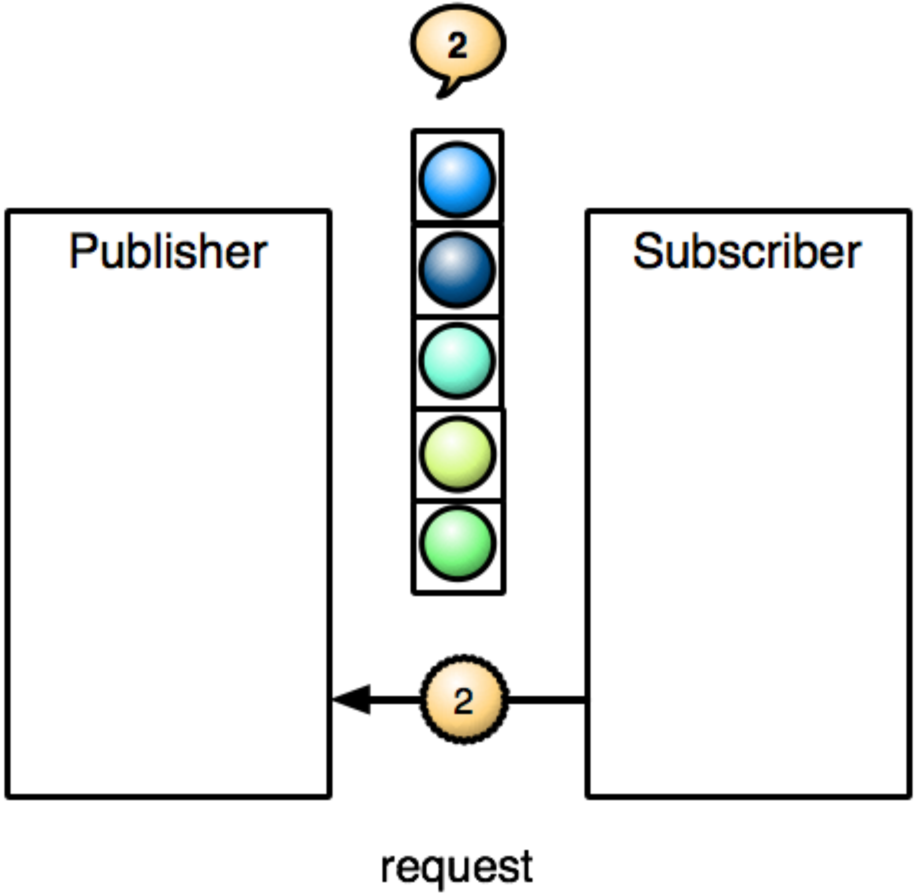


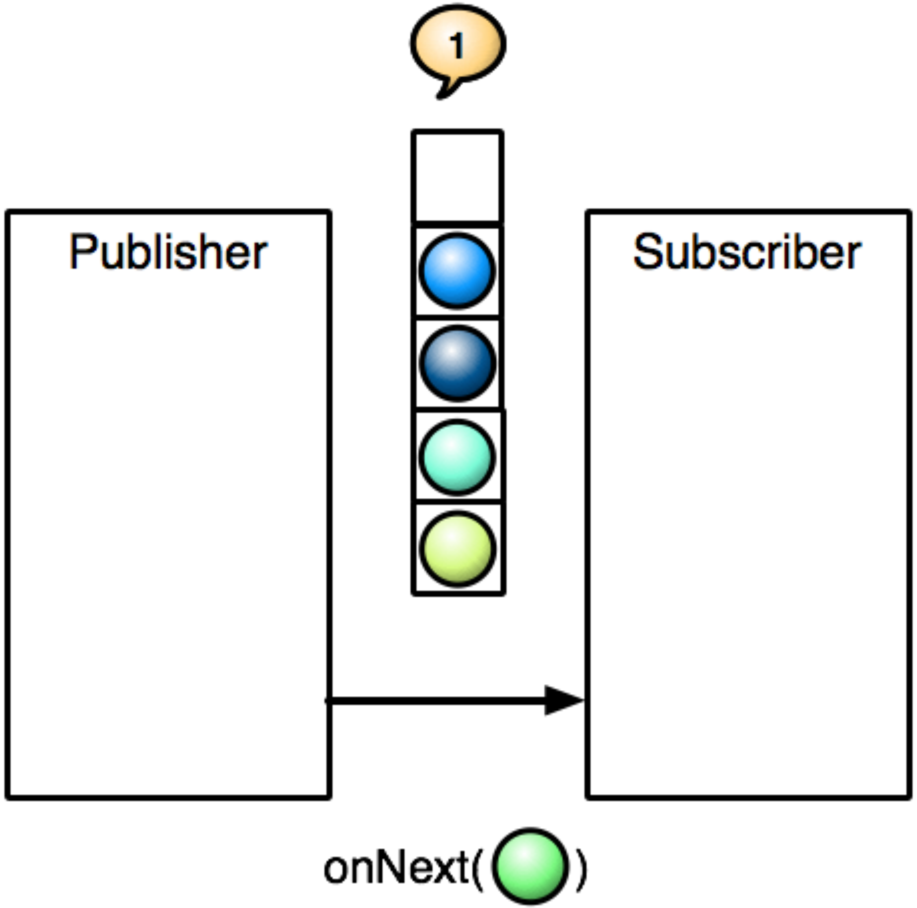

subscribe(Subscriber)
Publisher
Subscriber

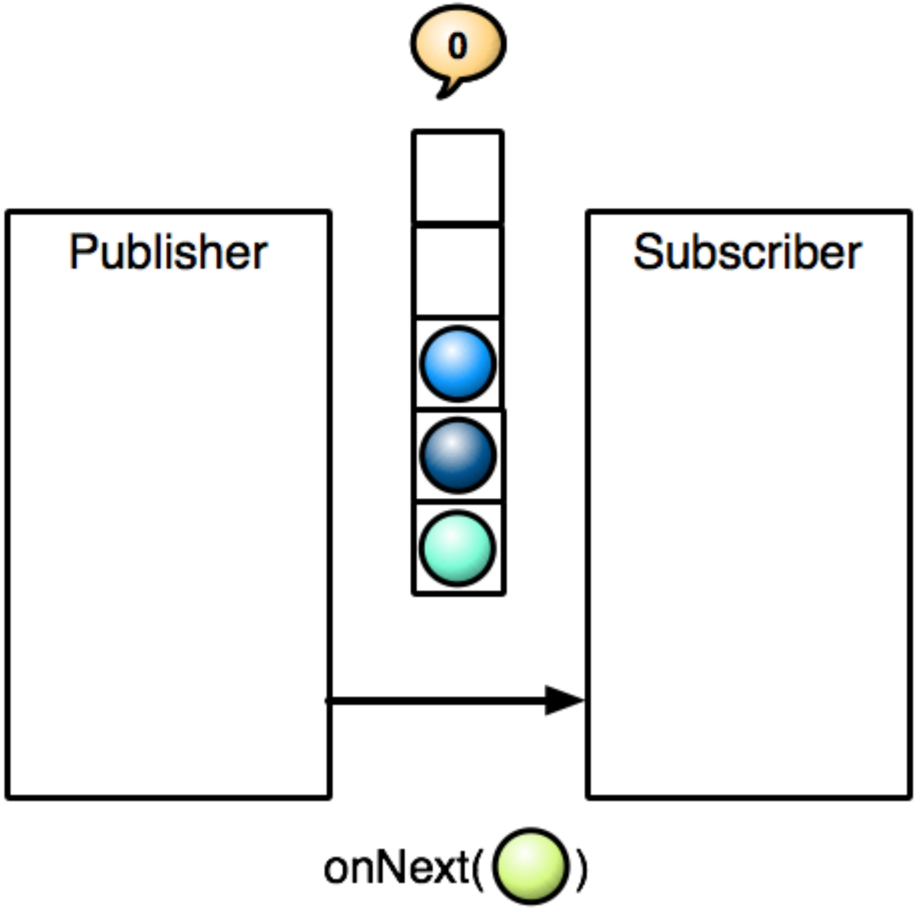

Publisher
Subscriber

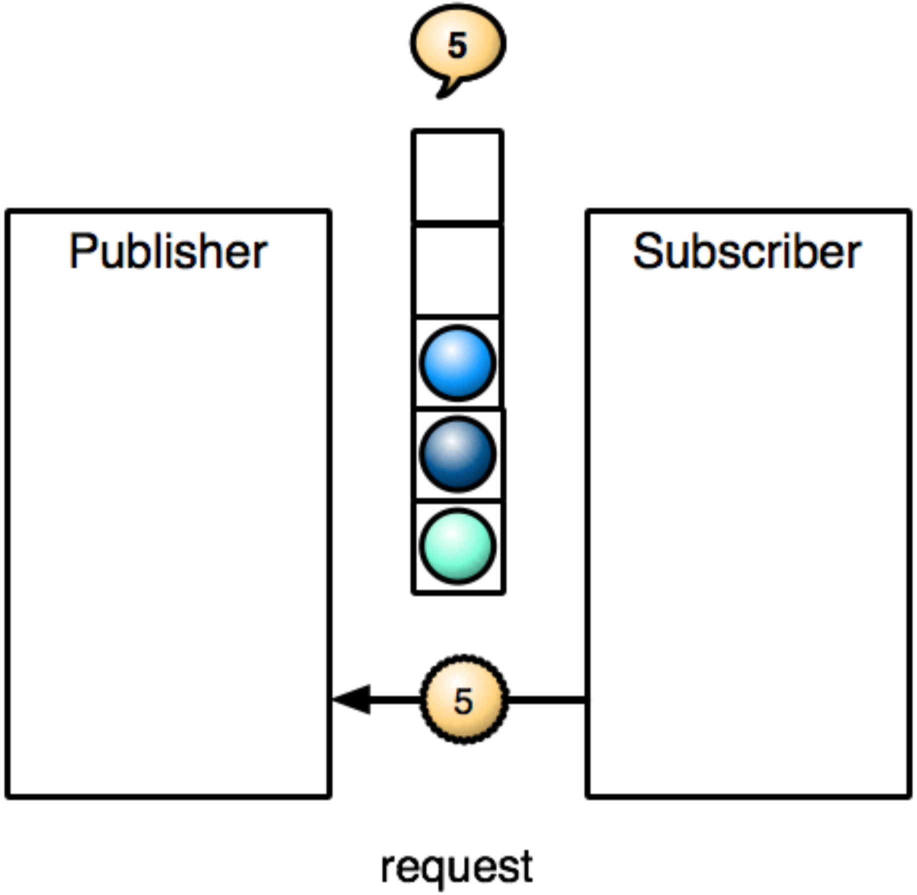


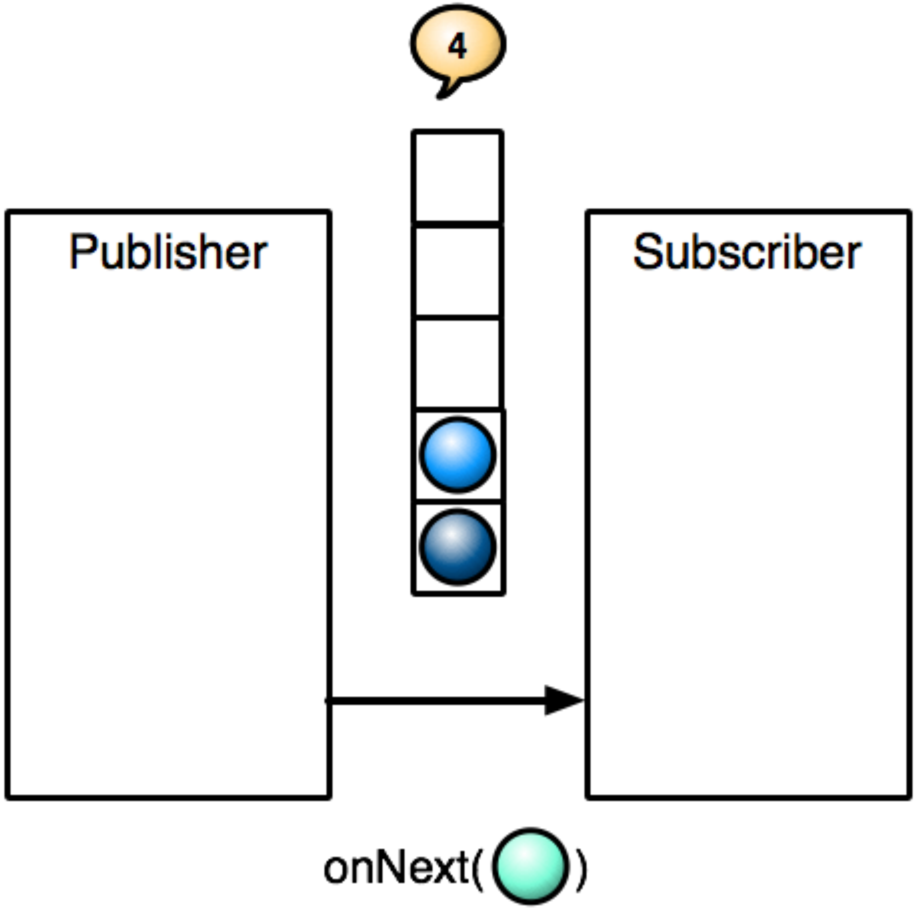
onSubscribe(Subscription)

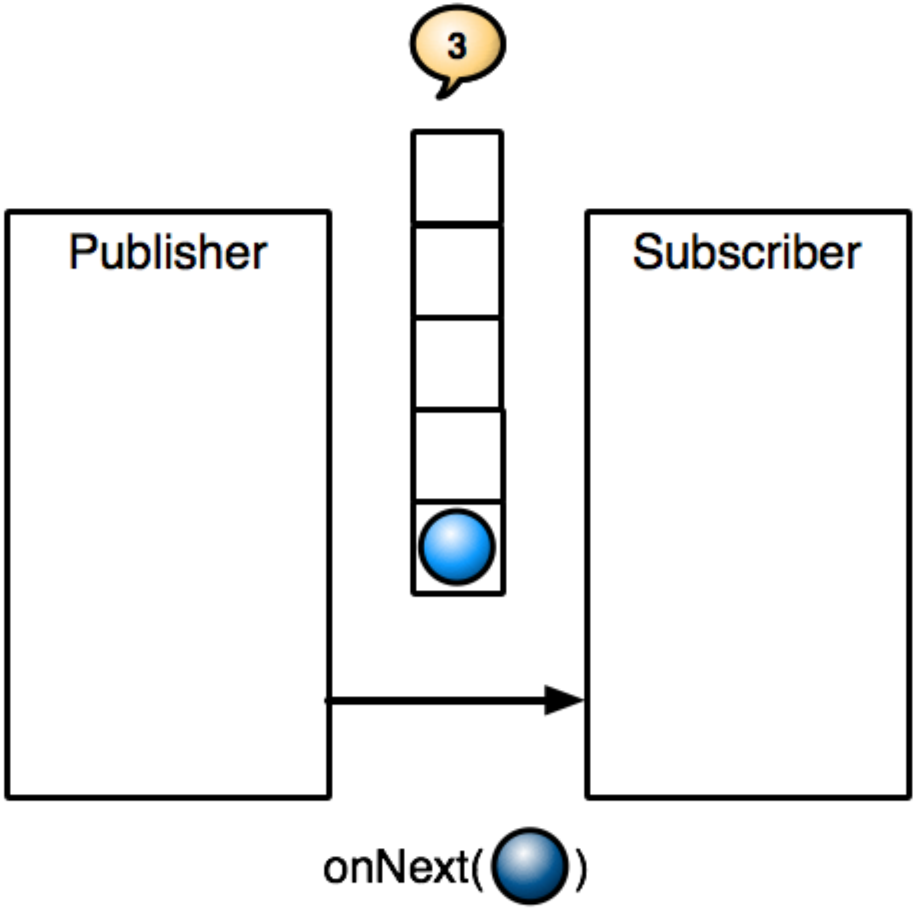

2
Publisher
Subscriber
2
request

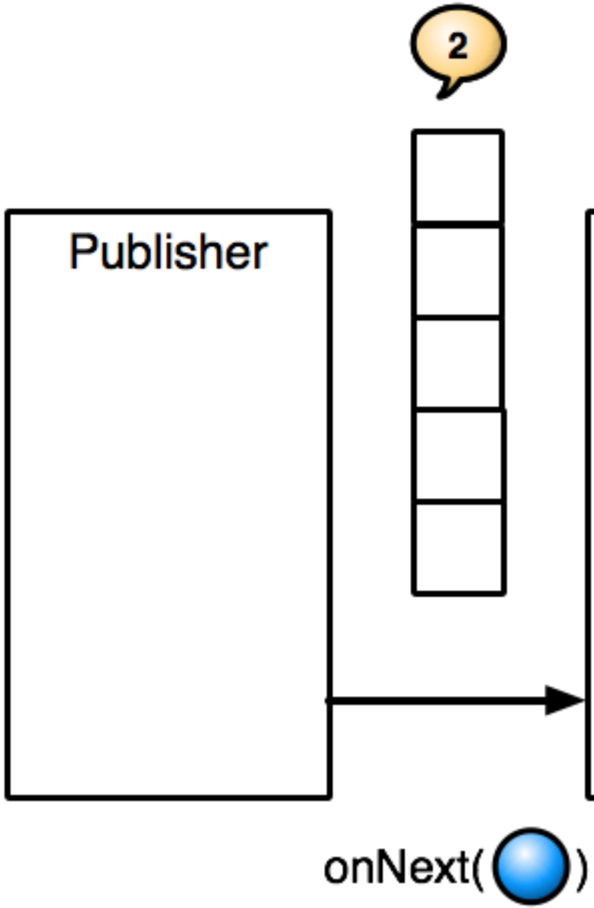

1
Publisher
Subscriber
onNext( )

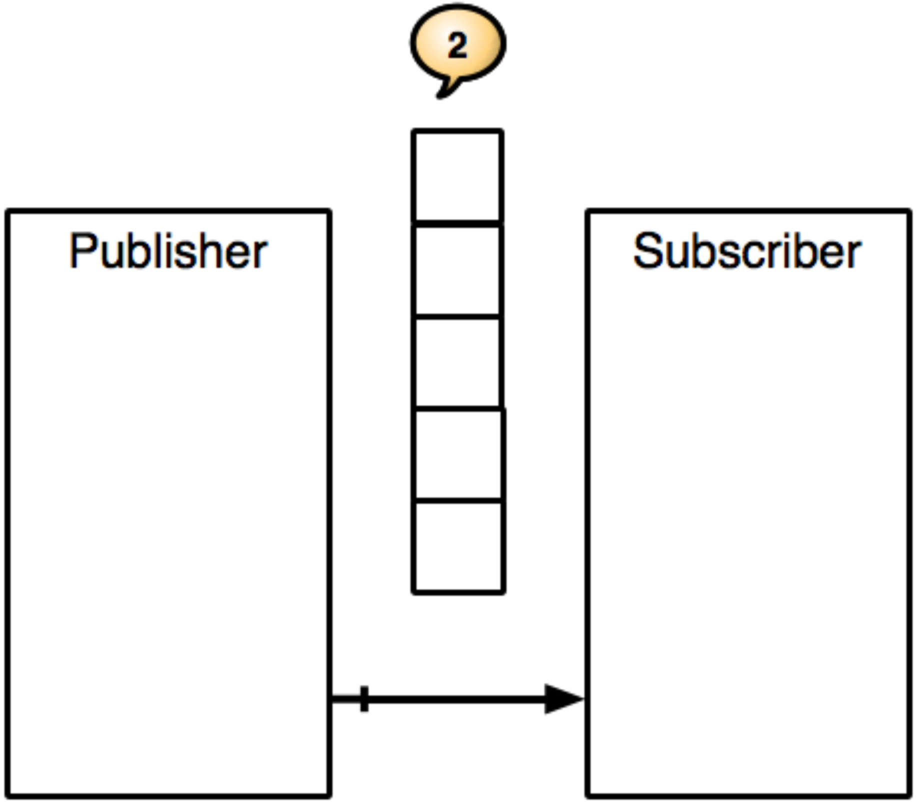

0
Publisher
Subscriber
onNext( )


5
Publisher
Subscriber
5
request


4
Publisher
Subscriber
onNext( )


3
Publisher
Subscriber
onNext( )


2
Publisher
Subscriber
onNext( )



onComplete

Spec Rules

- **No concurrent** calls from Publisher to Subscriber (1.3) or vice versa (2.7)
- A Subscriber may perform work **synchronously or asynchronously**
  - either way **must be non-blocking** (2.2)
- Upper bound on open recursion required (3.3)
  - `onNext` ➜ `request` ➜ `onNext` ➜ … ➜ `StackOverflow`
- No exceptions can be raised except NPE for `null` input (1.9 2.13)
  - Publisher calls `onError(Throwable)`
  - Subscriber cancels subscription

What does `request(n)` mean?

- N represents an element count
  - relative (byte chunks) more so than absolute (# of bytes)

- Boundaries
  - Must be > 0
  - `Long.MAX_VALUE` means unbounded read
  - So does overflow of N

Request Strategies

- Request some receive all request more … (“*stop-and-wait*”)
  - request signal **equivalent to an ack**
- Pre-fill buffer of certain capacity (“*pre-fetch*”)
  - request again **when buffer falls** below some threshold
  - more in-flight data ➜ more risk
- Request using timer/**scheduler** facility (“*poll*”)
- Adapt to network or processing latency (“*dynamic*”)
  - find and pick the **best request size**

Demo:

Publisher & Subscriber, Back-pressure, TCK verification

[Demo source](), [TCK test]()

Async vs Sync

- Sync is explicitly allowed -- 2.2 3.2 3.10 3.11
- If `Publisher/Subscriber` are both sync ➔ open recursion
  - not always an issue e.g. single item `Publisher`
- The picture is different in a processing chain (vs. single Publisher-Subscriber)
  - async hand-off is likely needed at some stage
  - but not at every stage
- Async is not the goal non-blocking is the overall objective

Async vs Sync (continued)

- Implementations are free to choose how to do this
- As long as they comply with all the rules
- Generally speaking the burden is on the `Publisher`
  - make async or sync calls as necessary
  - prevent open recursion
  - sequential signals
- *Subscribers can largely ignore such concerns*

Reactive Streams → JDK 9 Flow.java

- No single best fluent async/parallel API in Java [1]
- `CompletableFuture/CompletionStage`
  - continuation-style programming on futures
- `java.util.stream`
  - multi-stage/parallel, “pull”-style operations on collections
- No "push"-style API for items as they become available from active source

---

Java “concurrency-interest” list:
[1] initial announcement [2] why in the JDK? [3] recent update

JDK 9 `java.util.concurrent`

- java.util.concurrent.Flow
  - `Flow.Publisher Flow.Subscriber Flow.Subscription` …
- Flow.consume(…) and Flow.stream(…)
  - Tie-ins to `CompletableFuture` and `java.util.Stream`
- SubmissionPublisher
  - turn any kind of source to `Publisher`
  - `Executor`-based delivery to subscribers w/ bounded (ring) buffer
  - `submit` and `offer` strategies (block or drop) to publish

API For Async Composition

Application

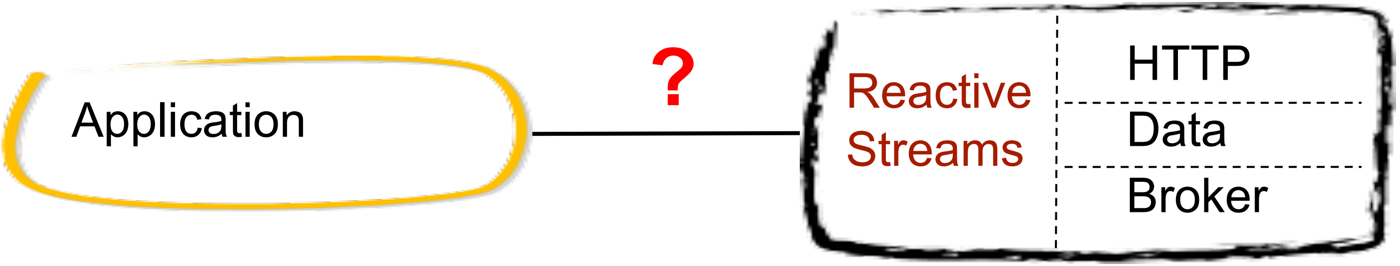
Application

Reactive Streams

HTTP

Data

Broker



API to **compose** asynchronous programs?

Working with Streams

- Non-blocking services return `Publisher<T>` instead of `<T>`
- How do you attach further processing?
- Reactive Streams is a callback-based API
    - becomes very nested quickly
- Need something more declarative
    - it's beyond the scope of Reactive Streams

Stream Operations

- `Publisher` represents a stream of data
- It’s natural to apply operations functional-style
  - like the Java 8 Stream
- Need API for composing async logic
  - rise above callbacks

Reactor Stream

- Project Reactor provides a Stream API
- Reactive Streams `Publisher` + composable operations

```
Streams.just('a' 'b' 'c')
        .take(2)
        .map(Character::toUpperCase)
        .consume(System.out::println);
```

Demo:

Reactor Stream, Back-Pressure

Demo source

Reactive Extensions (Rx)

- Stream composition API based on Observer pattern
- Originated at Microsoft, work by Erik Meijer
- Implemented for different languages -- RxJava, RxJS, Rx.NET, …

```
Observable.just('a', 'b', 'c')
    .take(2)
    .map(Character::toUpperCase)
    .subscribe(System.out::println);
```

RxJava and Reactive Streams

- RxJava 1.x predates Reactive Streams and doesn't implement it directly
- Very similar concepts, different names
  - `Observable-Observer` vs `Publisher-Subscriber`
- RxJava supports "reactive pull" back-pressure
- RxJava 1.x - Reactive Streams bridge

Rx vs Java 8 Streams

- Java 8 Streams best suited for collections
- `Observer` and `Iterator` actually very closely related
  - same except push vs pull
- `Observable` however can represent any source including collections
- RxJava 2 is [planned to support](#) Java 8 `Stream`
  - `Observable.from(Iterable)` + implement `java.util.Stream`

Demo:

RxJava, Back-Pressure, Reactive Streams Bridge

Demo class, *TCK test*

RxJava vs Reactor

- RxJava is a great choice for composition in applications
  - many operators, polyglot (client/server), well-documented

- Reactive Streams is ideal for use in library APIs
  - remain agnostic to composition

- Reactor is positioned as foundation for libraries
  - essential Reactive Streams infrastructure + core operators
  - also used in high volume applications

Head Start with Stream Composition

Operators: reactivex.io Alphabetical List

Aggregate, All, Amb, and_, And, Any, apply, as_blocking, AsObservable, AssertEqual, asyncAction, asyncFunc, Average, averageDouble, averageFloat, averageInteger, averageLong, blocking, Buffer, bufferWithCount, bufferWithTime, bufferWithTimeOrCount, byLine, cache, case, Cast, Catch, catchException, collect, collect (RxScala version of Filter), CombineLatest, combineLatestWith, Concat, concat_all, concatMap, concatMapObserver, concatAll, concatWith, Connect, connect_forever, cons, Contains, controlled, Count, countLong, Create, cycle, Debounce, decode, DefaultIfEmpty, Defer, deferFuture, Delay, delaySubscription, delayWithSelector, Dematerialize, Distinct, DistinctUntilChanged, Do, doAction, doOnCompleted, doOnEach, doOnError, doOnRequest, doOnSubscribe, doOnTerminate, doOnUnsubscribe, doseq, doWhile, drop, dropRight, dropUntil, dropWhile, ElementAt, ElementAtOrDefault, Empty, empty?, encode, ensures, error, every, exclusive, exists, expand, failWith, Filter, filterNot, Finally, finallyAction, finallyDo, find, findIndex, First, FirstOrDefault, firstOrElse, FlatMap, flatMapFirst, flatMapIterable, flatMapIterableWith, flatMapLatest, flatMapObserver, flatMapWith, flatMapWithMaxConcurrent, flat_map_with_index, flatten, flattenDelayError, foldl, foldLeft, for, forall, ForEach, forEachFuture, forIn, forkJoin, From, fromAction, fromArray, FromAsyncPattern, fromCallable, fromCallback, FromEvent, FromEventPattern, fromFunc0, from_future, from_iterable, from_list, fromNodeCallback, fromPromise, fromRunnable, Generate, generateWithAbsoluteTime, generateWithRelativeTime, generator, GetEnumerator, getIterator, GroupBy, GroupByUntil, GroupJoin, head, headOption, headOrElse, if, ifThen, IgnoreElements, indexOf, interleave, interpose, Interval, into, isEmpty, items, Join, join (string), jortSort, jortSortUntil, Just, keep, keep-indexed, Last, lastOption, LastOrDefault, lastOrElse, Latest, latest (Rx.rb version of Switch), length, let, letBind, limit, LongCount, ManySelect, Map, map (RxClojure version of Zip), MapCat, mapCat (RxClojure version of Zip), map-indexed, map_with_index, Materialize, Max, MaxBy, Merge, mergeAll, merge_concurrent, mergeDelayError, mergeObservable, mergeWith, Min, MinBy, MostRecent, Multicast, nest, Never, Next, Next (BlockingObservable version), none, nonEmpty, nth, ObserveOn, ObserveOnDispatcher, observeSingleOn, of, of_array, ofArrayChanges, of_enumerable, of_enumerator, ofObjectChanges, OfType, ofWithScheduler, onBackpressureBlock, onBackpressureBuffer, onBackpressureDrop, OnErrorResumeNext, onErrorReturn, onExceptionResumeNext, orElse, pairs, pairwise, partition, partition-all, pausable, pausableBuffered, pluck, product, Publish, PublishLast, publish_synchronized, publishValue, raise_error, Range, Reduce, reductions, RefCount, Repeat, repeat_infinitely, repeatWhen, Replay, rescue_error, rest, Retry, retry_infinitely, retryWhen, Return, returnElement, returnValue, runAsync, Sample, Scan, scope, Select (alternate name of Map), select (alternate name of Filter), selectConcat, selectConcatObserver, SelectMany, selectManyObserver, select_switch, selectSwitch, selectSwitchFirst, selectWithMaxConcurrent, select_with_index, seq, SequenceEqual, sequence_eql?, SequenceEqualWith, Serialize, share, shareReplay, shareValue, Single, SingleOrDefault, singleOption, singleOrElse, size, Skip, SkipLast, skipLastWithTime, SkipUntil, skipUntilWithTime, SkipWhile, skip_while_with_index, skip_with_time, slice, sliding, slidingBuffer, some, sort, sort-by, sorted-list-by, split, split-with, Start, startAsync, startFuture, StartWith, stringConcat, stopAndWait, subscribe, SubscribeOn, SubscribeOnDispatcher, subscribeOnCompleted, subscribeOnError, subscribeOnNext, Sum, sumDouble, sumFloat, sumInteger, sumLong, Switch, switchCase, switchIfEmpty, switchLatest, switchMap, switchOnNext, Synchronize, Take, take_with_time, takeFirst, TakeLast, takeLastBuffer, takeLastBufferWithTime, takeLastWithTime, takeRight (see also: TakeLast), TakeUntil, takeUntilWithTime, TakeWhile, take_while_with_index, tail, tap, tapOnCompleted, tapOnError, tapOnNext, Then, thenDo, Throttle, throttleFirst, throttleLast, throttleWithSelector, throttleWithTimeout, Throw, throwError, throwException, TimeInterval, Timeout, timeoutWithSelector, Timer, Timestamp, To, to_a, ToArray, ToAsync, toBlocking, toBuffer, to_dict, ToDictionary, ToEnumerable, ToEvent, ToEventPattern, ToFuture, to_h, toIndexedSeq, toIterable, toIterator, ToList, ToLookup, toMap, toMultiMap, ToObservable, toSet, toSortedList, toStream, ToTask, toTraversable, toVector, tumbling, tumblingBuffer, unsubscribeOn, Using, When, Where, while, whileDo, Window, windowWithCount, windowWithTime, windowWithTimeOrCount, windowed, withFilter, withLatestFrom, Zip, zipArray, zipWith, zipWithIndex

Operators: `rx.Observable`

```
all, amb, ambWith, asObservable, buffer, cache, cast, collect, combineLatest, compose,
concat, concatMap, concatWith, contains, count, countLong, create, debounce,
defaultIfEmpty, defer, delay, delaySubscription, dematerialize, distinct,
distinctUntilChanged, doOnCompleted, doOnEach, doOnError, doOnNext, doOnRequest,
doOnSubscribe, doOnTerminate, doOnUnsubscribe, elementAt, elementAtOrDefault, empty,
error, exists, filter, finallyDo, first, firstOrDefault, flatMap, flatMapIterable, from,
groupBy, groupJoin, ignoreElements, interval, isEmpty, join, just, last, lastOrDefault,
lift, limit, map, materialize, merge, mergeDelayError, mergeWith, nest, never,
observeOn, ofType, onBackpressureBlock, onBackpressureBuffer, onBackpressureDrop,
onBackpressureLatest, onErrorResumeNext, onErrorReturn, onExceptionResumeNext, publish,
range, reduce, repeat, repeatWhen, replay, retry, retryWhen, sample, scan,
sequenceEqual, serialize, share, single, singleOrDefault, skip, skipLast, skipUntil,
skipWhile, startWith, subscribeOn, switchIfEmpty, switchMap, switchOnNext, take,
takeFirst, takeLast, takeLastBuffer, takeUntil, takeWhile, throttleFirst, throttleLast,
throttleWithTimeout, timeInterval, timeout, timer, timestamp, toList, toMap, toMultimap,
toSortedList, unsubscribeOn, using, window, withLatestFrom, zip, zipWith
```

Operators: what can you do with sequences?

- Originate -- fixed values, arrays, ranges, from scratch
- Reduce -- filter, accumulate, aggregate, partition
- Transform -- create new type of sequence
- Combine -- concatenate, merge, pair
- Control -- overflow, errors

Compose Non-blocking Service Layer

Blocking:

```
List<String> findUsers(String skill);
LinkedInProfile getConnections(String id);
TwitterProfile getTwitterProfile(String id);
FacebookProfile getFacebookProfile(String id);
```

Non-Blocking:

```
Observable<String> findUsers(String skill);
Observable<LinkedInProfile> getConnections(String id);
Observable<TwitterProfile> getTwitterProfile(String id);
Observable<FacebookProfile> getFacebookProfile(String id);
```

Demo:

Get List of RxJava Operators

[*Demo source*]

Compose Non-Blocking Service Layer

[*Demo source*]

flatMap?!

From `map` to `flatMap`

- `map` is used to apply a function to each element
  - `Function<T, R>`
- The function could return a new sequence
  - `Function<T, Observable<R>>`
- In which case we go from **one** to **many** sequences
- `flatMap` merges (flattens) those back into one
  - i.e. `map` + `merge`

Learn Your Operators

- `flatMap` is essential composing non-blocking services
  - scatter-gather, microservices
- Along with `concatMap, merge, zip`, and others
- Learn and experiment with operators
- Use decision tree for choosing operators on reactivex.io to learn

Stream Types

- **“Cold”** source -- **passive**, subscriber can dictate rate
  - e.g. file, database cursor
  - usually “replayed from the top” per subscriber
- **“Hot”** source -- **active**, produce irrespective of subscribers
  - e.g. mouse events, timer events
  - cannot repeat
- Not always a clear distinction
  - operators further change stream behavior
  - e.g. merge hot + cold

Stream Types & Back-Pressure

- Cold sources are well suited for reactive back-pressure
  - i.e. support `request(n)`
- Hot sources cannot pause, need alternative flow control
  - buffer, sample or drop
- Several operators exist
  - `onBackPressureBuffer`, `onBackPressureDrop`
  - `buffer`, `window`
  - etc.

More on
Stream Composition

How Operators Work

- Each operator is a **deferred** declaration of work to be done

```
observable.map(...).filter(...).take(5)...
```

- No work is done until there is a subscriber

```
observable.map(...).filter(...).take(5) subscribe(...)
```

- Underlying this is the `lift` operator

How the `lift` Operator Works

- Accepts function for decorating `Subscriber`
  - `lift(Function<Subscriber<DOWN>, Subscriber<UP>>)`

- Returns new `Observable` that when subscribed to will use the function to decorate the `Subscriber`

- Effectively each operator decorates the target Subscriber
  - at the time of subscribing

The Subscribe

- For every subscriber the `lift` chain is used (subscriber decorated)
- Data starts flowing to the subscriber
- Every subscriber is decorated independently
- Results in separate data pipelines (N subscribers ➔ N pipelines)

Reasons For Multiple Subscribers

- Different rates of consumption, slow/fast consumers
- Different resource needs, i.e. IO vs CPU bound
- Different tolerance to errors
- Different views on dealing with overflow
- Different code bases

Shared Data Pipeline

- It’s possible to have shared pipeline across subscribers
- Insert shared publisher in the chain
  - vs `lift` which decorates every subscriber
- Both RxJava and Reactor support this
  - `observable.share()` and `stream.process(...)` respectively
- Reactor has processors for fan-out or point-to-point
  - pass all elements to all subscribers
  - distribute elements (“worker queue” pattern)

Demo:

Multiple Streams,
Shared Streams (fan-out, point-to-point)

[*Demo Source*]

Concurrency in Rx

- By default all work performed on subscriber's thread
- Operators generally not concurrent
  - some do accept `rx.Scheduler` (timer-related)
- Two operators for explicit concurrency control
  - `subscribeOn(Scheduler), observeOn(Scheduler)`
- `Schedulers` class with factory methods for `Scheduler` instances
  - `io(), computation(),trampoline(),` …

`observeOn` Operator

- Invoke rest of the chain of operators on specified `Scheduler`
- Inserts async boundary for downstream processing
- Uses a queue to buffer elements
- Temporarily mitigate different upstream/downstream rates

`subscribeOn` operator

- Put source `Observable` on specified `Scheduler`
  - both initial subscribe + subsequent requests
- Order in chain of operators not significant
- Use for Observable that performs blocking I/O
  - e.g. `observable.subscribeOn(Schedulers.io())`

Concurrency in Reactor

- Reactive Streams `Processor` for explicit concurrency control
  - comparable to Rx `observeOn`

```
Streams.period(1)
    .process(RingBufferProcessor.create())
    .consume(...);
```

- `Processor` can also insert async boundary for upstream signals
  - comparable to Rx `subscribeOn`

- Internally some operators may use `Timer` thread
  - the main reason for Rx operators accepting `Scheduler`

Error Handling

- `try-catch-finally` is of limited value in non-blocking app
- Exceptions may occur on different thread(s)
- Notifications are the only path to consistent error handling
- Reactive Streams forbids propagating exceptions through the call-stack
- `Subscriber` receives `onError`

Try but won’t catch

```
try {
  Streams.just(1, 2, 3, 4, 5)
         .map(i -> {
            throw new NullPointerException();
         })
         .consume(
            element -> { … },
            error -> logger.error("Oooh error!", error)
         );
}
catch (Throwable ex) {
  logger.error("Crickets...");
}
```

Recovering from Errors

- There are operators to handle error notifications
- Swallow error + emit backup sequence (or single value)
  - `onErrorResumeNext`, `onErrorReturn`
- Re-subscribe, it might work next time
  - `retry`, `retryWhen`
- Lifecycle related
  - `doOnTerminate/finallyDo`... before/after error-complete

SPRINGONE2GX
WASHINGTON, DC

Learn More. Stay Connected.

@springcentral

Spring.io/video

Follow-up talk on Thursday (10:30am)

“Reactive Web Applications”