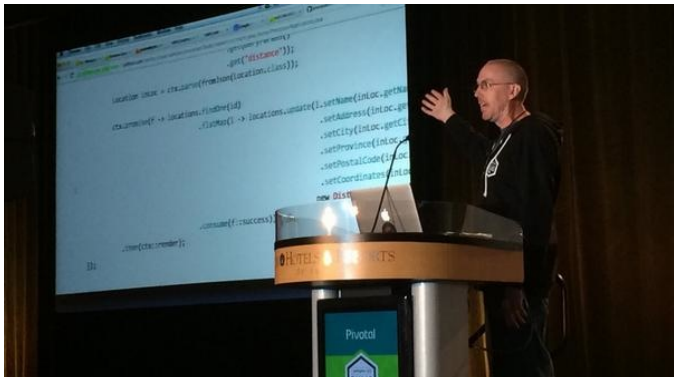
# Building #uberfastdata Applications with: @ProjectReactor

**Jon Brisbin**

Reactor Project Lead

***Pivotal***

# The Reactor Team



@j_brisbin - 100% asynchronous poet

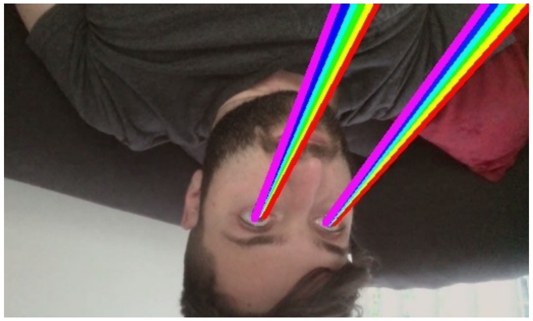
Reactor Committer I

Reactive-Streams Contributor

@smaldini - solve 9 issues, create 10 problems

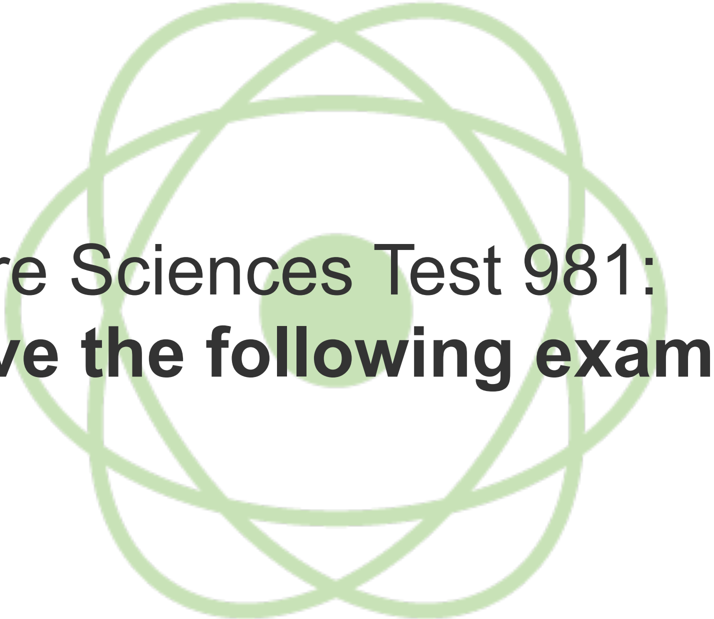
Reactor Committer II

Reactive-Streams Contributor



SAY MICROSERVICE
ONE MORE TIME!
memegenerator.net

# NanoService, MicroService, NotTooBigService™…



Aperture Sciences Test 981:
**Observe the following examples**

# NanoService, MicroService, NotTooBigService™…

```
cat file.csv | grep ‘doge’ | sort
```

## NanoService, MicroService, NotTooBigService™…

*POST [json]* http://dogecoin.money/send/id

—> GET [json] http://dogeprofile.money/id

—> POST [json] http://nsa.gov.us/cc/trace/id

# NanoService, MicroService, NotTooBigService™…

```
userService.auth(username,password)
  —> userService.hashPassword(password)
  —> userService.findByNameAndHash(name)
```

# NanoService, MicroService, NotTooBigService™…

- A SomethingService will **always need to interact**
  - With the user
  - With other services

- The **boundary** between services is the real deal
  - A big danger lurks in the dark
  - More breakdown => More boundaries

And this threat has a name

**Latency**

**UberFact** : *Humans don’t really enjoy waiting*

***Neither do The Machines***

## What is latency doing to you ?

- **Loss of revenues**
  - because users **switched to another site/app**
  - because services are **compounding inefficiency**
  - because **aggressive scaling** will be needed
  - because dev budget will sink into **addressing this a postiori**

**Out of the realm of possibility? :**

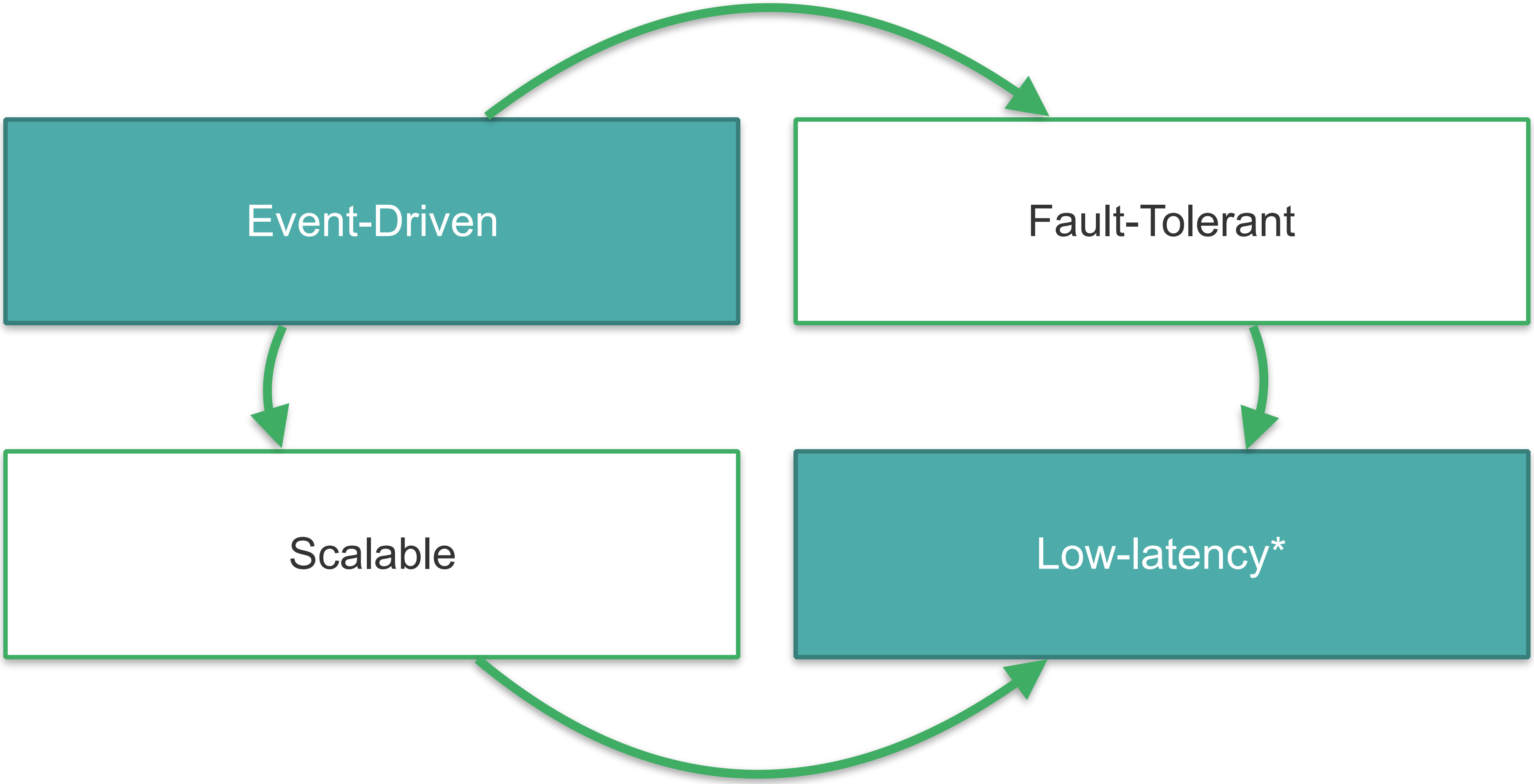
tech team turnover will fuel whining about how crappy the design is

Loading

## All hail Reactive Programming

- A possible answer to this issue
- The very nature of **Reactor**, look at the name dude
- A fancy buzz-word that might work better than MDA or SOA
- A simple accumulation of years of engineering

## What is Reactive Programming ?



Event-Driven

Fault-Tolerant

Scalable

Low-latency*

## Reactive Architecture ?

- A **Reactive** system **MUST** be resilient
  - splitting concerns to achieve error bulk-heading and modularity

- A **Reactive** system **MUST** be scalable
  - scale-up : partition work across CPUs
  - scale-out : distribute over peer nodes

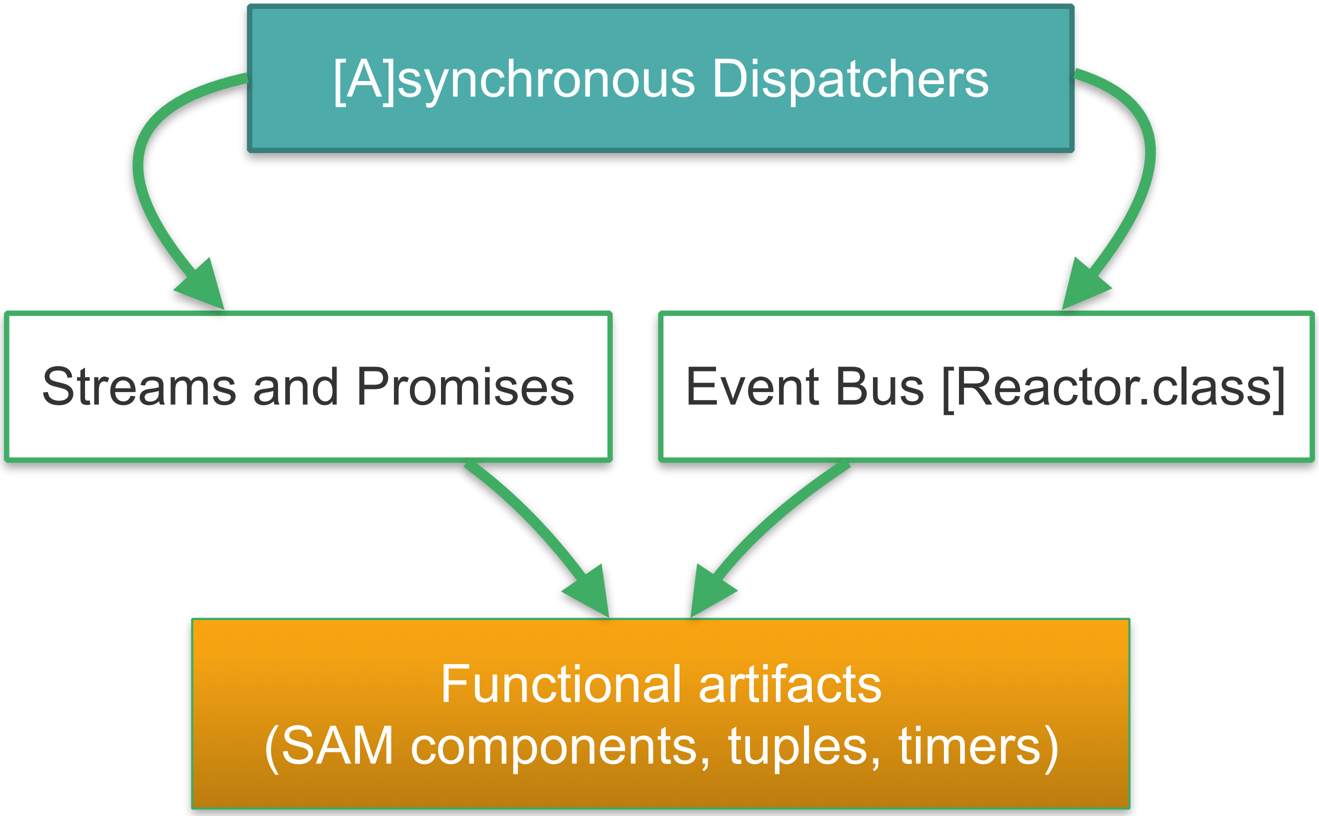
## Reactive Architecture !

- Asynchronous Programming is core to Reactive Architecture
  - *Immediate* answer to the originating publisher
  - *Context segregation* to avoid cascade failure as possible

- Functional Programming fits perfectly as it is stimulus based
  - **Related: F**unctional **R**eactive **P**rogramming

# Reactor has 99 problems but Latency isn’t one

# Reactor-Core features



[A]synchronous Dispatchers

Streams and Promises

Event Bus [Reactor.class]

Functional artifacts
(SAM components, tuples, timers)

Fast IO
[buffer, net,
persistent queues,
codec]

Fast Data
[allocators, batch-processors]

# Dispatching model matters

- **Context switching** hurts performance
- **Locking** hurts performance
- **Message passing** hurts performance
- **Blocking** for a thread hurts performance
- **Resources** cost

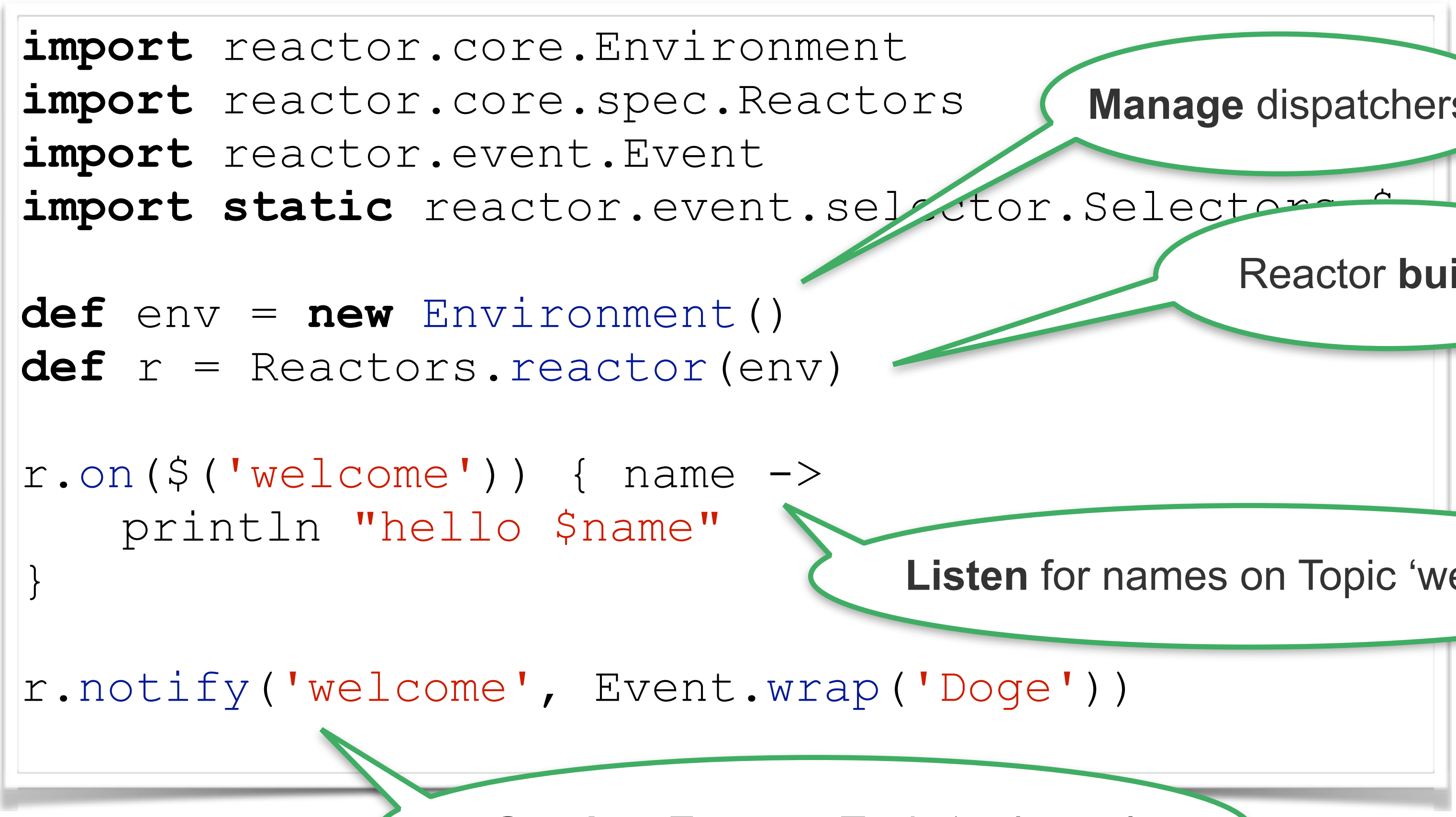
# Built-in UberFast™ dispatcher

- **LMAX Disruptor** deals with message passing issues
  - Based on a **Ring Buffer** structure
  - “**Mechanical Sympathy**” in Disruptor
  - http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf
  - http://mechanitis.blogspot.co.uk/2011/06/dissecting-disruptor-whats-so-special.html

# Message Passing matters

- **Pull** vs **Push** patterns
  - Push:
    - Non blocking programming (e.g. lock-free LMAX RingBuffer)
    - Functional programming
    - Best for in-process short access
  - Pull:
    - Queue based decoupling
    - Best for slow services or blocking connections

# Reactor event bus in action In Groovy because Java doesn't fit into the slide



```
import reactor.core.Environment
import reactor.core.spec.Reactors
import reactor.event.Event
import static reactor.event.selector.Selectors.$

def env = new Environment()
def r = Reactors.reactor(env)

r.on($('welcome')) { name ->
    println "hello $name"
}

r.notify('welcome', Event.wrap('Doge'))
```

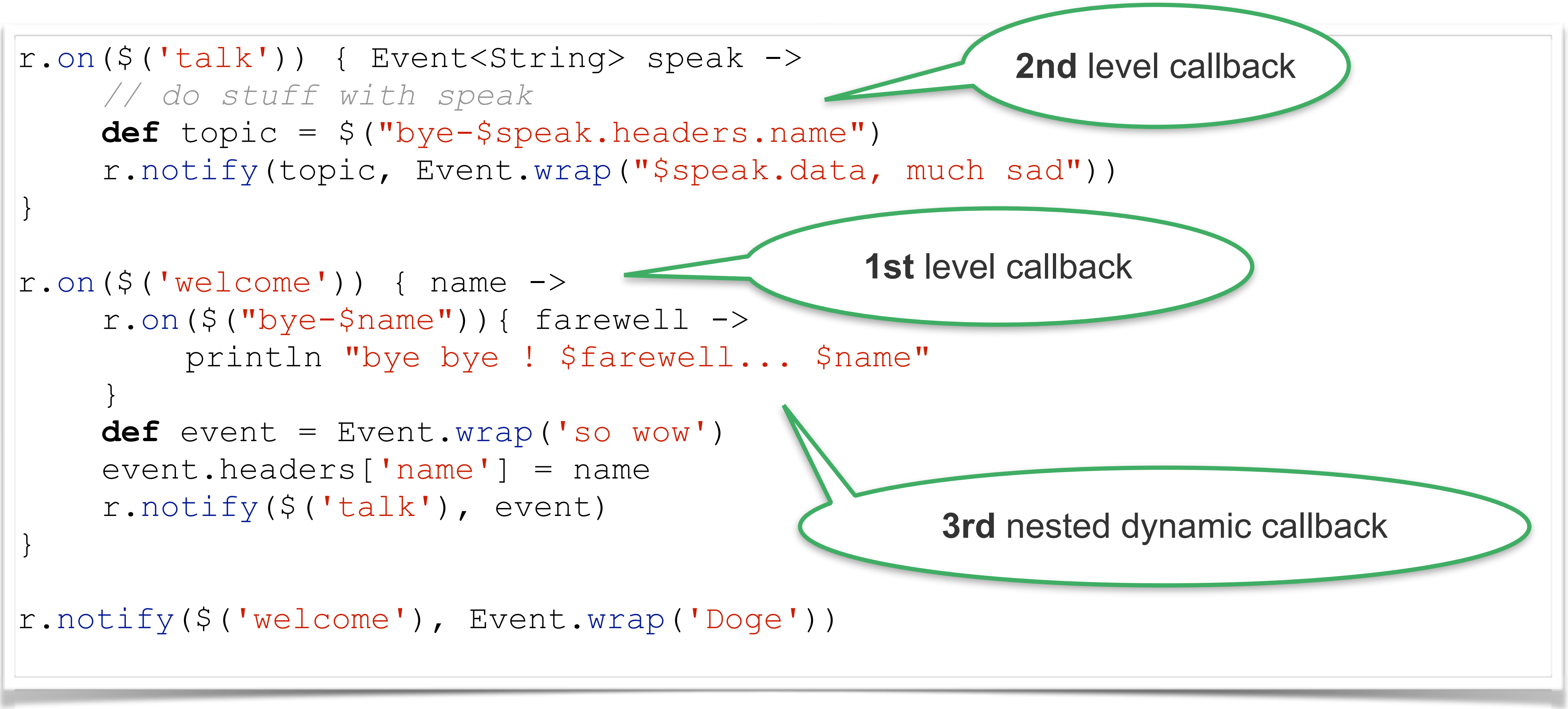
**Manage** dispatchers

Reactor **builder**

**Listen** for names on Topic 'welcome'

**Send** an **Event** to Topic 'welcome'

# Reactor callback hell In Groovy because Java doesn't fit into the slide



```
r.on($('talk')) { Event<String> speak ->
    // do stuff with speak
    def topic = $("bye-$speak.headers.name")
    r.notify(topic, Event.wrap("$speak.data, much sad"))
}

r.on($('welcome')) { name ->
    r.on($("bye-$name")){ farewell ->
        println "bye bye ! $farewell... $name"
    }
    def event = Event.wrap('so wow')
    event.headers['name'] = name
    r.notify($('talk'), event)
}

r.notify($('welcome'), Event.wrap('Doge'))
```

**2nd** level callback

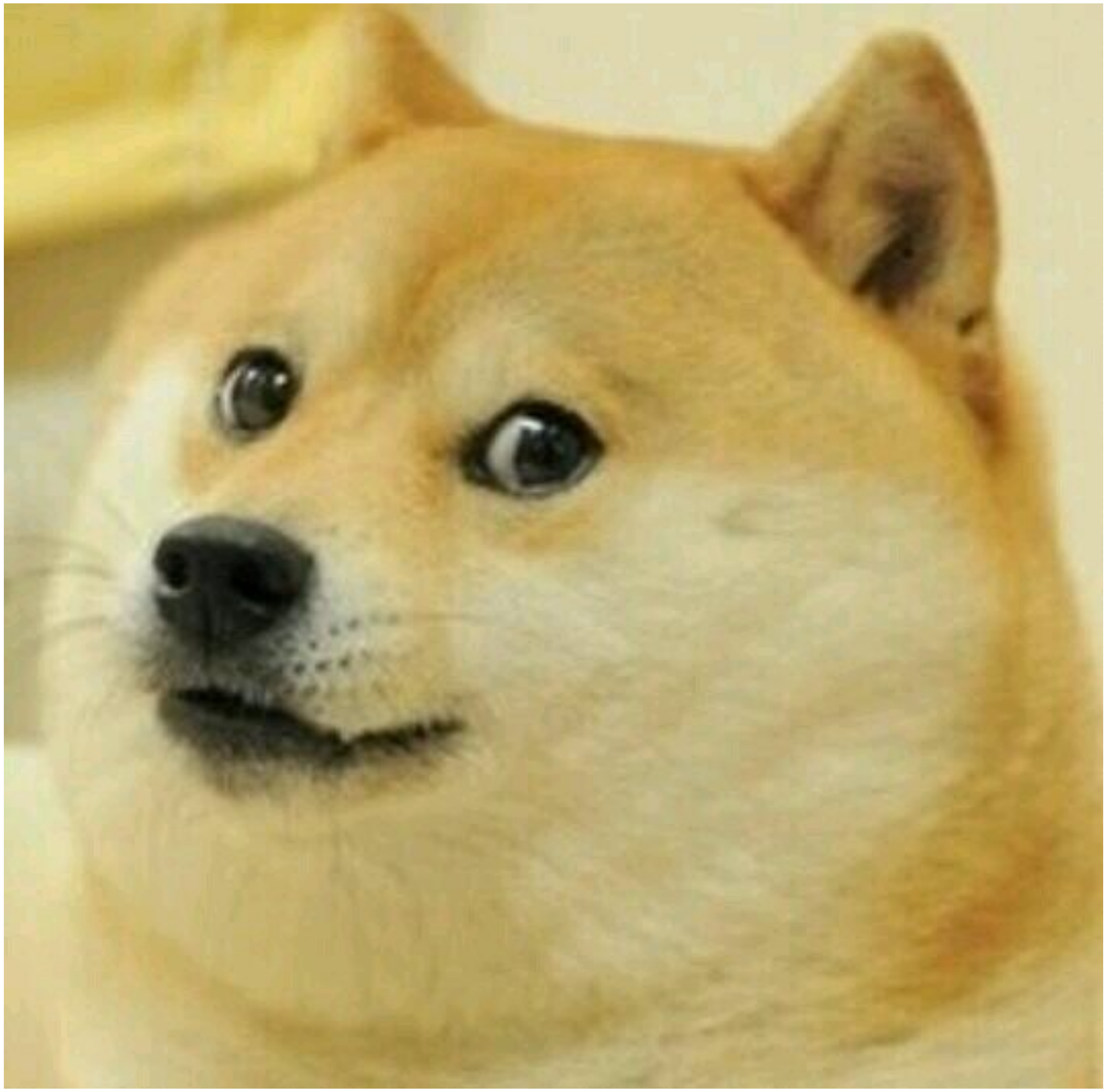
**1st** level callback

**3rd** nested dynamic callback

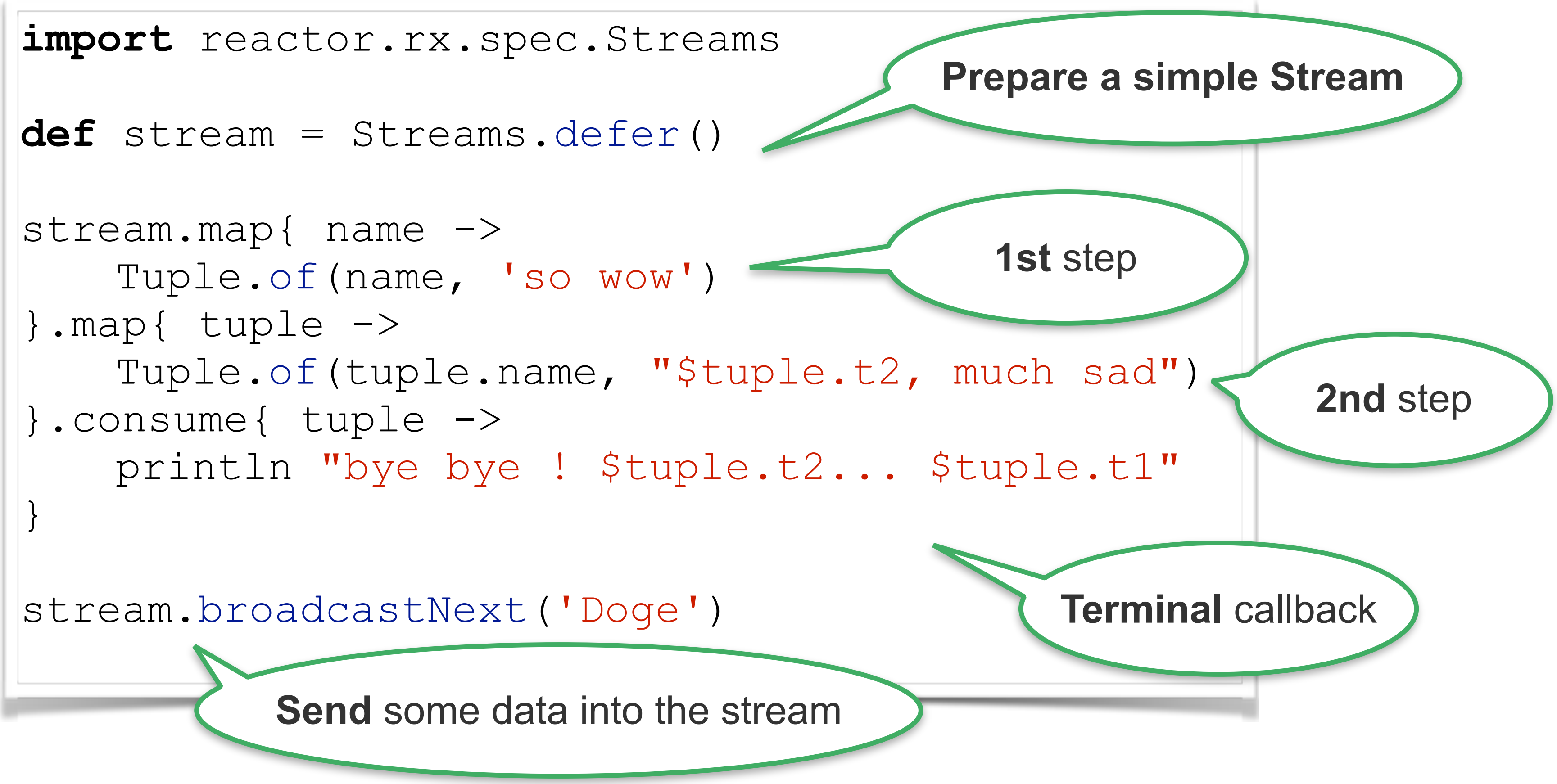
# Stream?

# Stream

# Stream!



# Solving callback hell



```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.name, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```

**Prepare a simple Stream**

**1st** step

**2nd** step

**Terminal** callback

**Send** some data into the stream

# Using a Stream ?

- Embedded data-processing
- Event Handling
- Metrics, Statistics
- Micro-Batching
- Composition
- Feedback-Loop

## Defining a Stream

- Represents a **sequence of data,** possibly **unbounded**
- Provide for processing API such as **filtering and enrichment**
- **Not** a ***Collection,* not** a ***Storage***

## Stream VS Event Bus [Reactor]

- Works great combined (stream distribution)
- Type-checked flow
- Publisher/Subscriber tight control
- No Signal concurrency

**Rule of thumb:**

if nested event composition > 2, switch to Stream

## Hot Stream vs Cold Stream

- An Hot Stream multi-casts real-time signals
  - think **Trade, Tick, Mouse Click**

- A Cold Stream uni-casts deferred signals
  - think **File, Array, Random**

# Introducing Reactive Streams Specification !

# What is defined by Reactive Streams ?

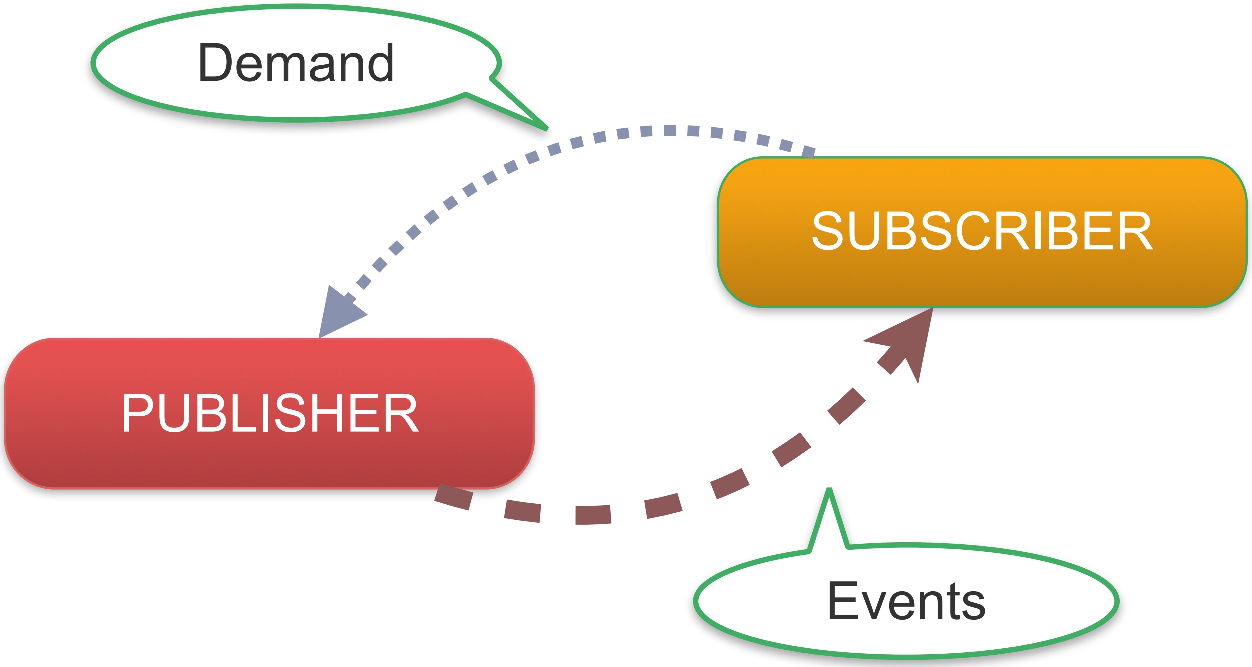
Async non-blocking data sequence

Interoperable protocol
(Threads, Nodes…)

Async non-blocking flow-control

Minimal resources requirement

# Reactive-Streams: Dynamic Message-Passing

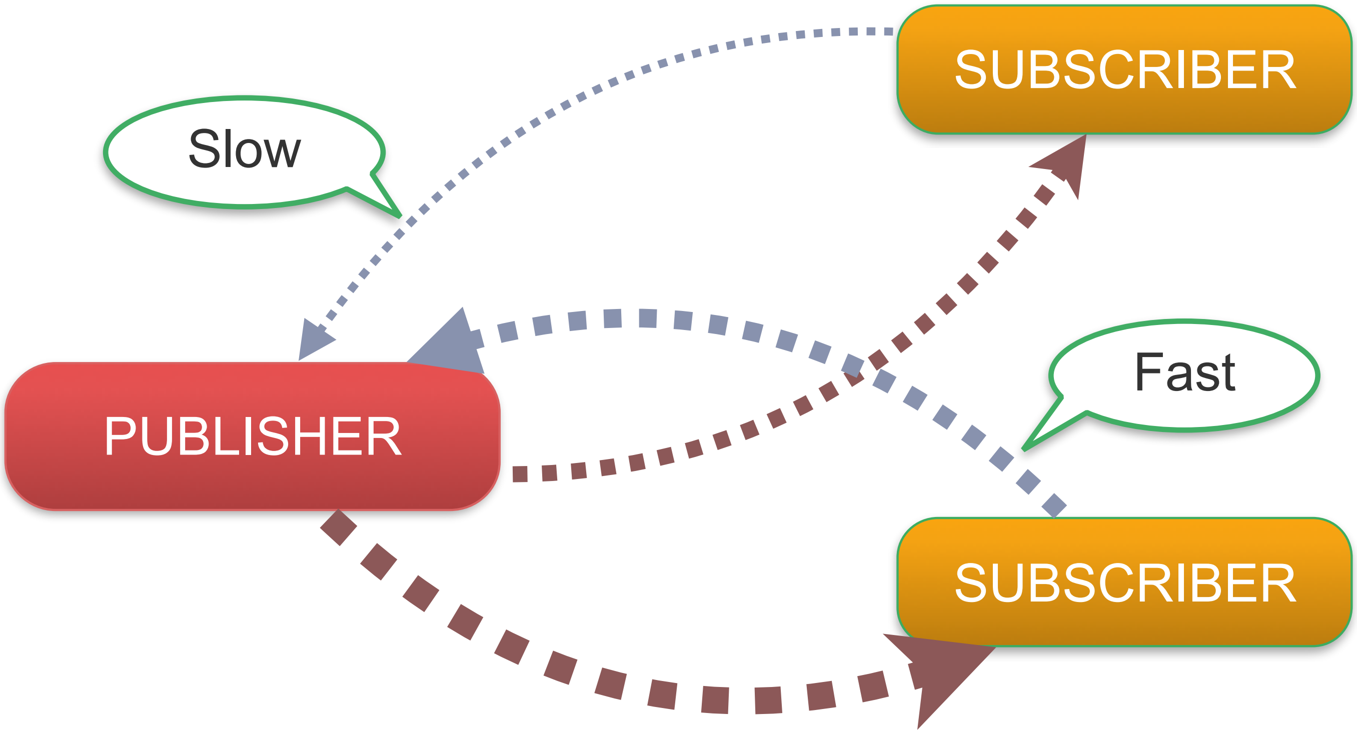


## Now You Know

- It is not only queue-based pattern:
  - Signaling demand on a slower **Publisher** == no buffering
  - Signaling demand on a faster **Publisher** == buffering

- Data volume is bounded by a **Subscriber**
  - Scaling dynamically if required

# Out Of The Box : Flow Control



# Reactive Streams: Batch Processing ?

- Requesting sized demand allows for **batch publishing** optimizations
  - Could be adapted dynamically based on criteria such as network bandwidth…
  - A Publisher could decide to apply grouped operations (aggregating, batch serializing)

# Reactive Streams: What is in the data sequence ?

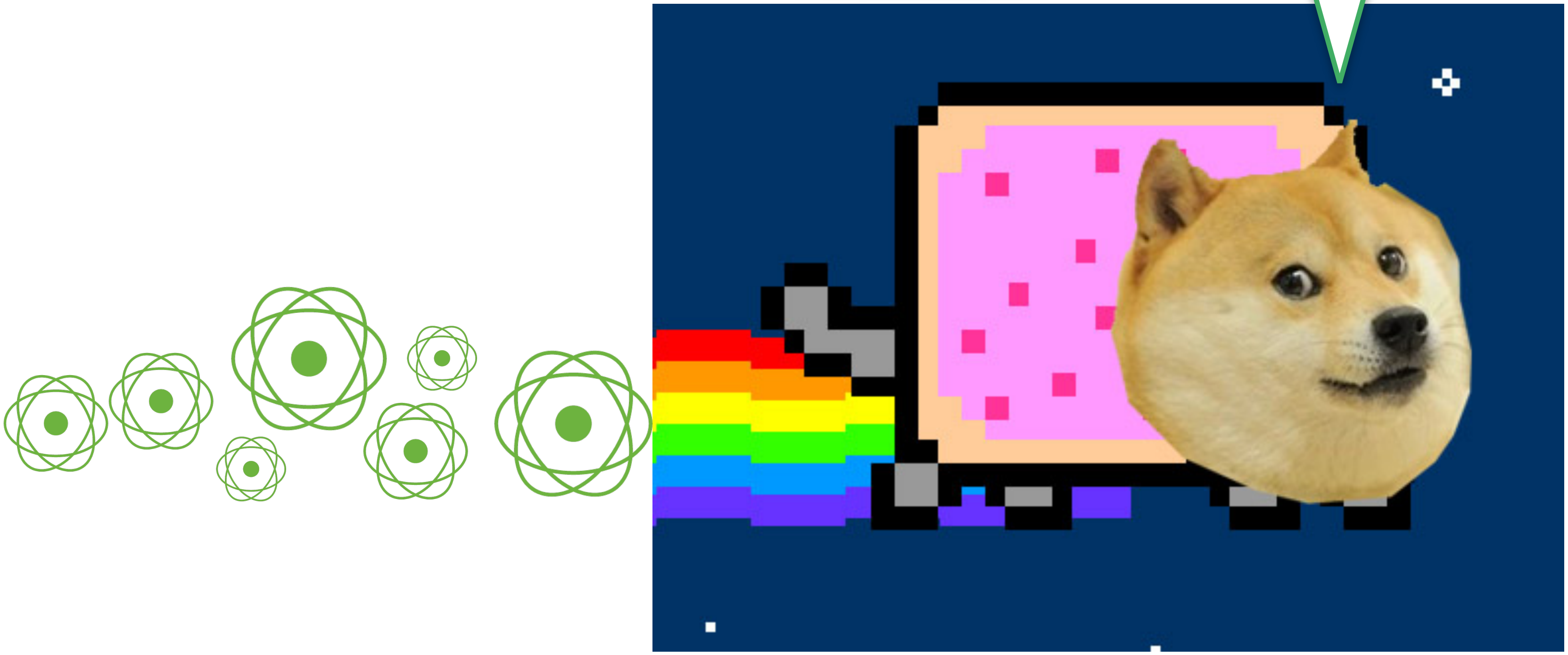
- If the Publisher is an **Hot Stream**
  - Sequence will be defined by **when** the Subscriber is connected

- If the Publisher is a **Cold Stream**
  - Sequence will be **similar** for every Subscriber regardless of when they connected

# Reactive Streams: Transformations ?

- Does not specify any transformation, only essentials components to enable this protocol…
- …But define a staging component **Processor :**
  - **Both A Subscriber AND a Publisher**

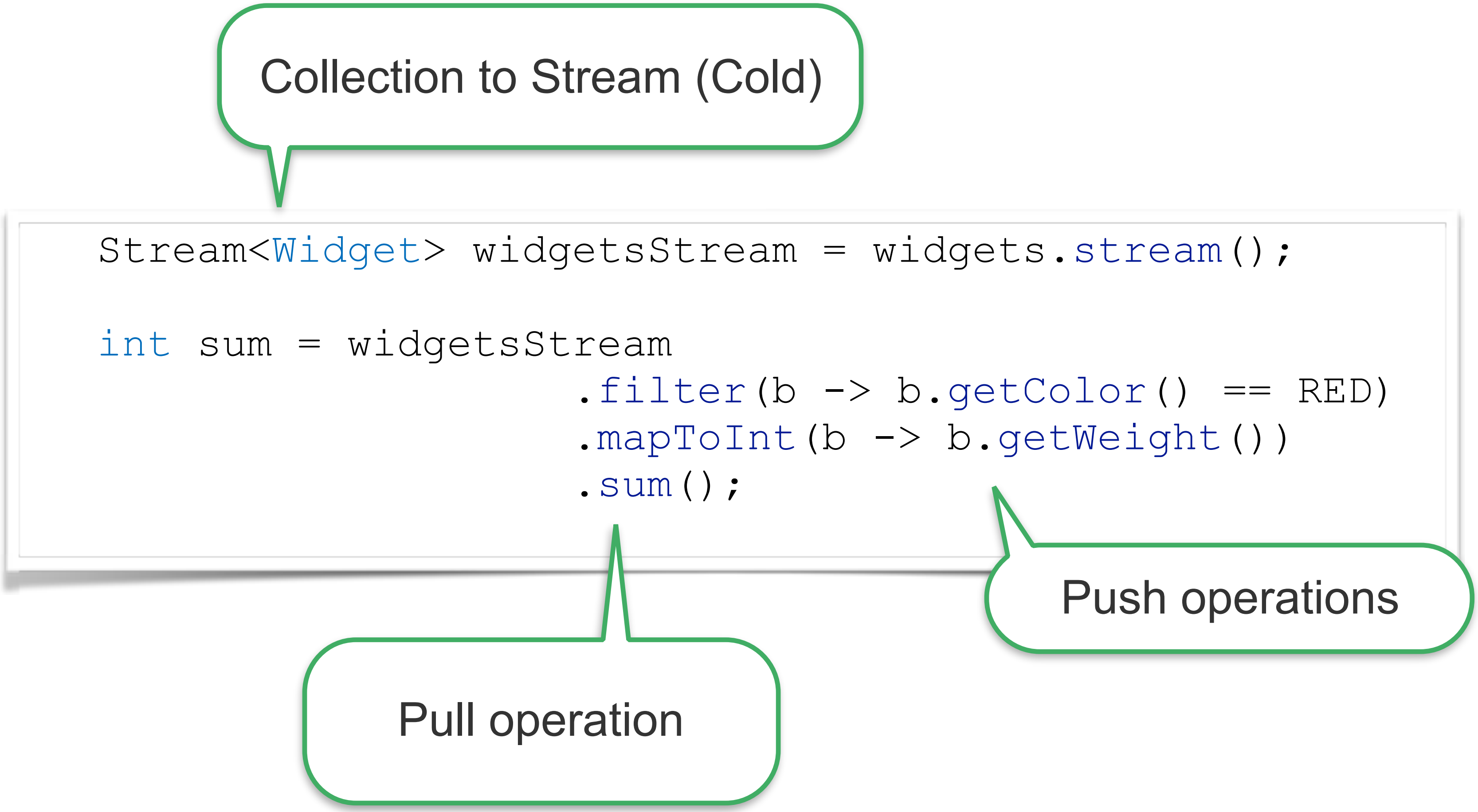
Hold on a minute buddy, there are already **Streams in Java 8 !**



# All the “Streaming” tech around… Java 8

- **Java 8** introduces **java.util.stream.Stream**
  - Functional DSL to **support transformations and stages**
  - P**roactive fetching**
  - **No dynamic message passing**
  - **Fits nicely** with event-driven libraries such as Reactor and RxJava.

# All the “Streaming” tech around… Java 8



Collection to Stream (Cold)

```
Stream<Widget> widgetsStream = widgets.stream();

int sum = widgetsStream
                .filter(b -> b.getColor() == RED)
                .mapToInt(b -> b.getWeight())
                .sum();
```

Push operations

Pull operation

What about **RxJava mate !** All those hipsters use it.

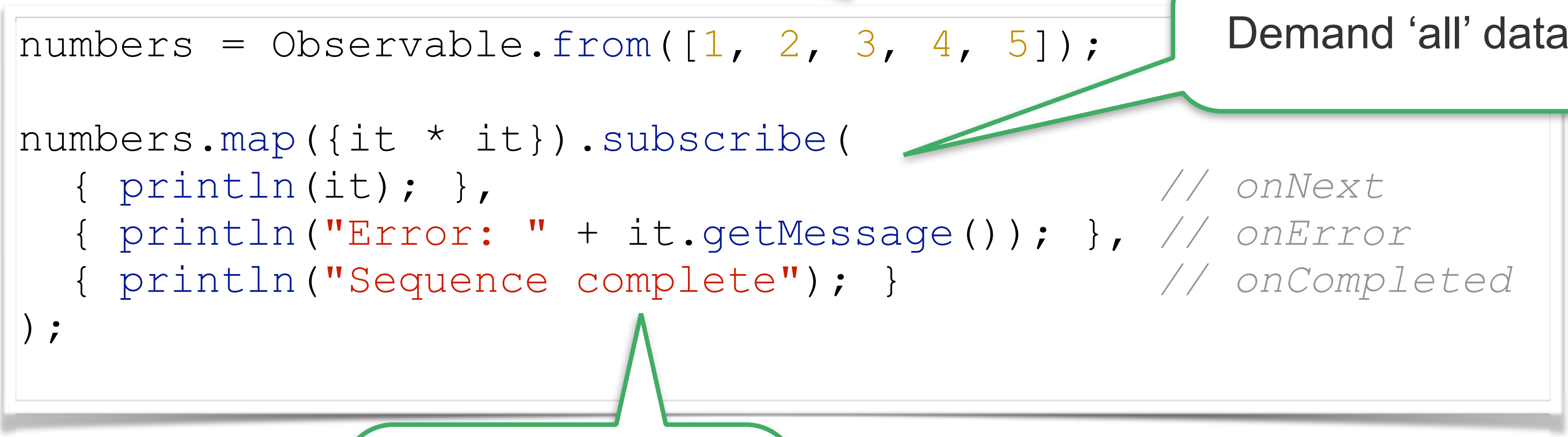
# All the “Streaming” tech around… RxJava

- **RxJava** provides the now famous **Reactive Extensions**
  - **Rich** transformations
  - **Event-Driven** (onNext, onError, onComplete)
  - **Flexible** scheduling
  - **No dynamic demand** OoB(possibly unbounded in-flight data)

# All the “Streaming” tech around… Java 8

Observe a ***Cold*** Stream

Demand ‘all’ data



```
numbers = Observable.from([1, 2, 3, 4, 5]);

numbers.map({it * it}).subscribe(
  { println(it); },                          // onNext
  { println("Error: " + it.getMessage()); }, // onError
  { println("Sequence complete"); }          // onCompleted
);
```

Push operations

# Other “Streaming” tech around…

- Streaming data is all the rage in modern frameworks:
  - **Akka, Storm, Reactor**, **Ratpack**, …

- Semantics compare with **Enterprise Integration Patterns**:
  - **Camel, Spring Integration/XD, …**
  - However it would technically **take multiple *channels* to model** a single Reactive Streams component (filter, transformer…)

# Reactive Streams: Joining forces

Typesafe

ORACLE

Pivotal

NETFLIX

spray

twitter

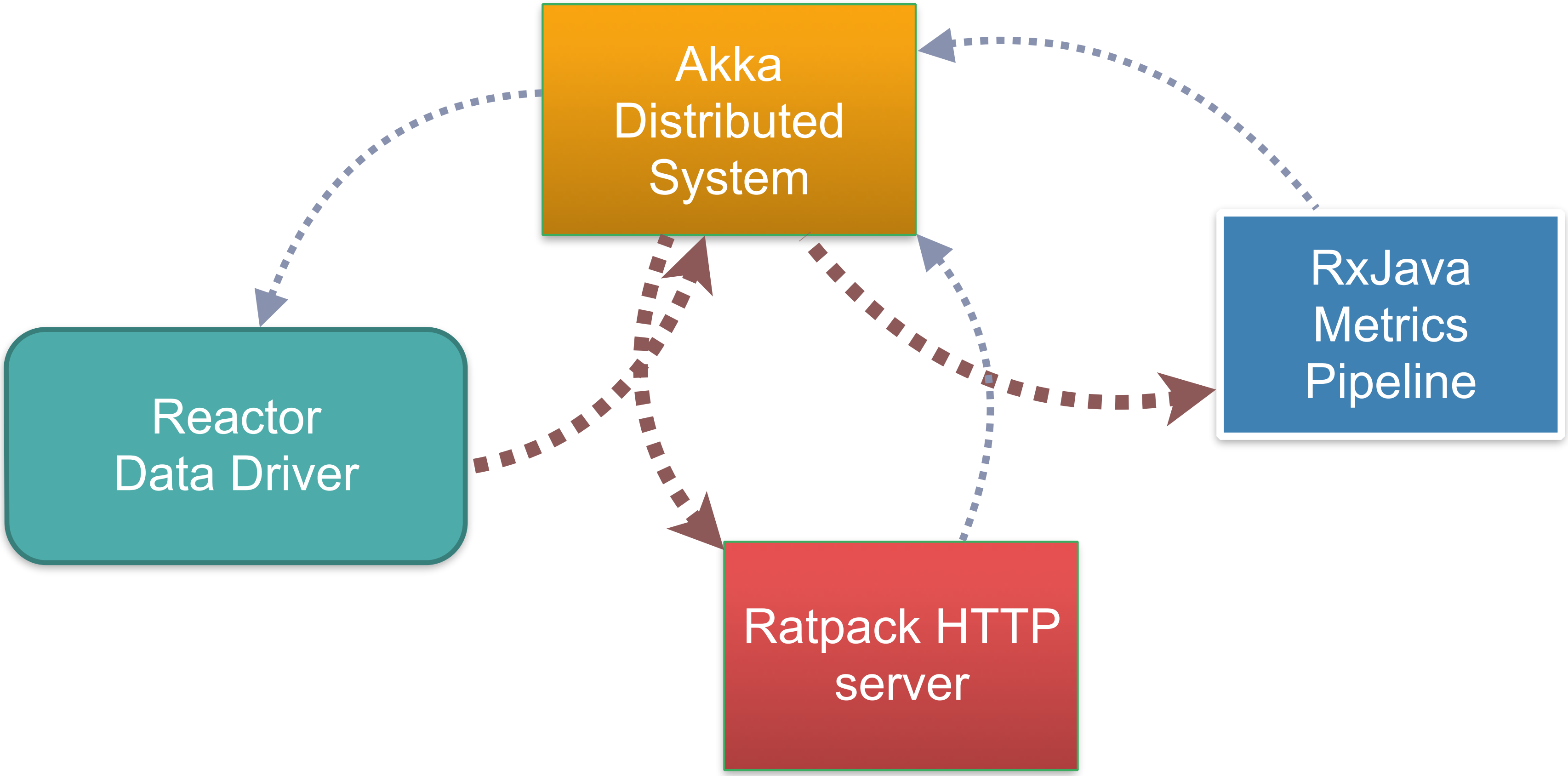
redhat

Applied Duality, Inc.

KAAZING

Doug Lea – SUNY Oswego

# Reactive Streams: Joining forces



## Reactive Streams: Joining forces

- **Smart** solution and pattern to **all reactive** applications
- Writing a **standard protocol** works best when it is used (!)
- **Beyond the JVM**, initial discussions for network stack started

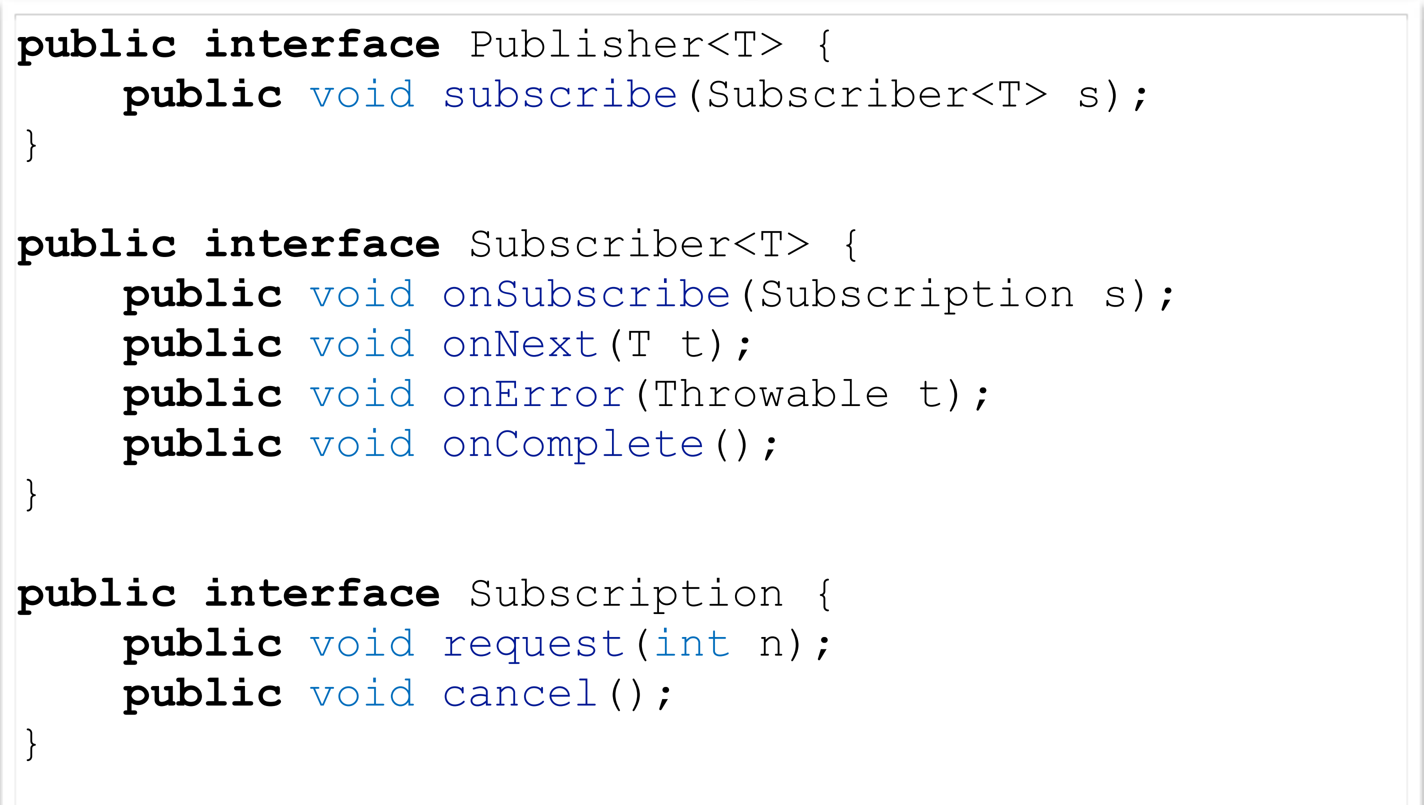
# Reactive Streams: Joining forces

- **Semantics**
  - **Single document** listing full rules
  - Open enough to allow for various patterns

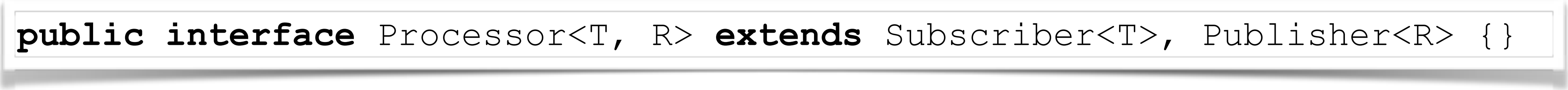
- 4 **API** Interfaces
  - **Publisher**, **Subscriber**, **Subscription**, ***Processor***

- **TCK** to verify implementation behavior

# Reactive Streams: org.reactivestreams

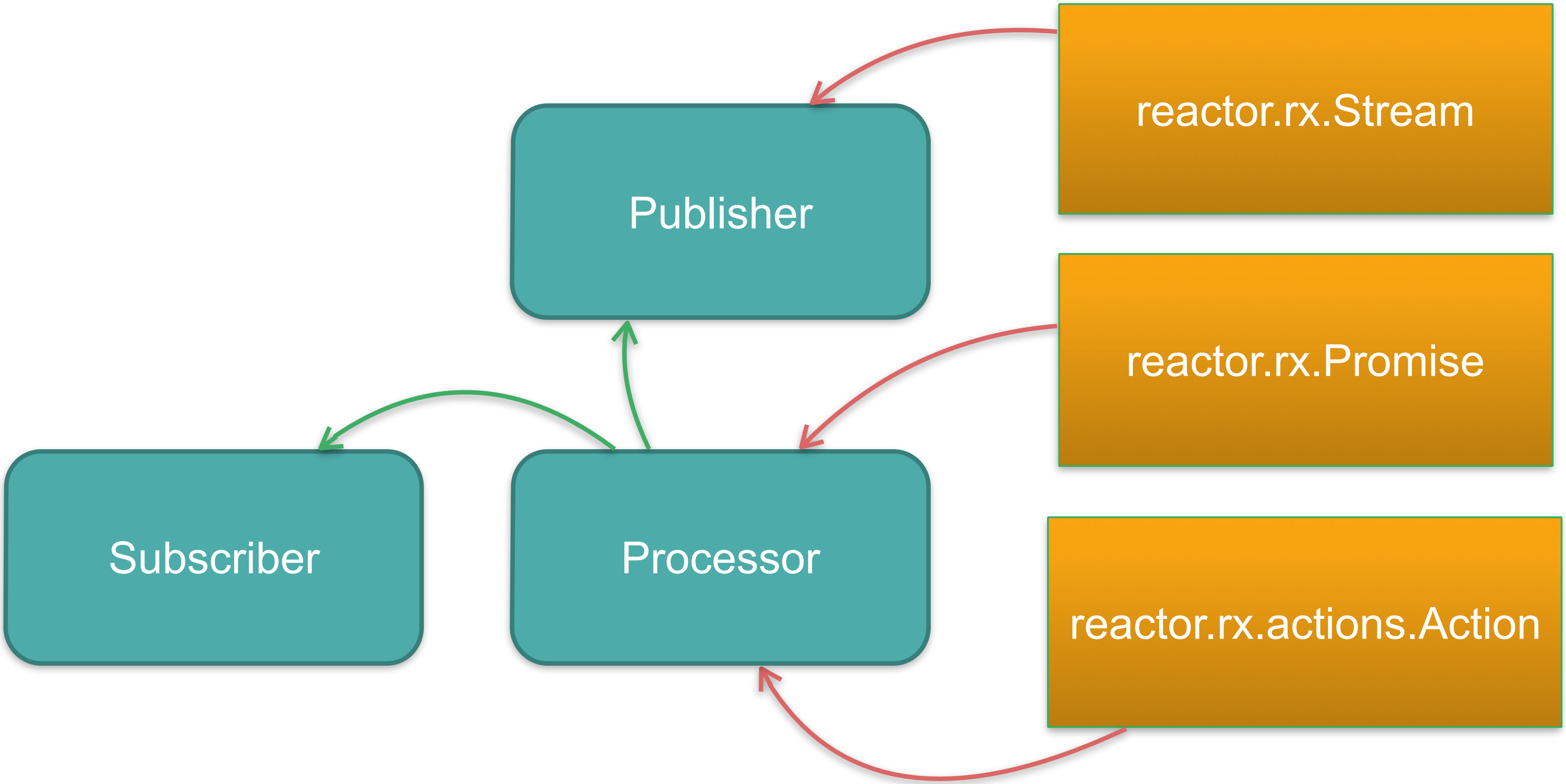


```java
public interface Publisher<T> {
    public void subscribe(Subscriber<T> s);
}

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

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



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

# Reactive Streams: Reactor mapping



# Reactive Streams: Execution Model

- **Publisher** creates a capacity-aware container
  - **Subscription per Subscriber**

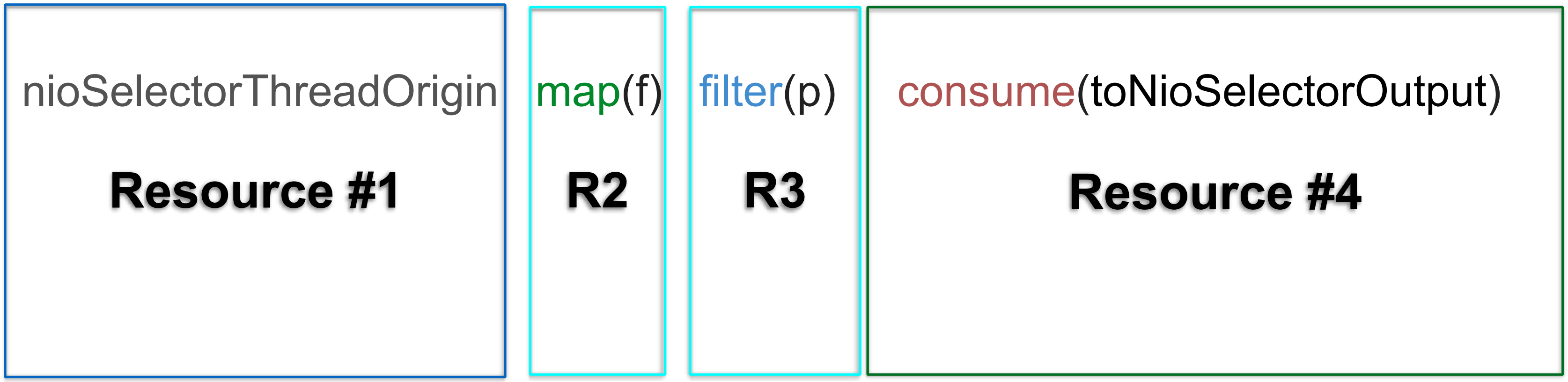
- **No concurrent** notifications on a same **Subscriber**

- **Asynchronous or Synchronous**
  - must not impact negatively calling code

# Reactive Streams: Signals

onError | (onSubscribe onNext* (onError | onComplete)?)

# Reactive Streams: Async Boundaries



nioSelectorThreadOrigin

**Resource #1**

map(f)

**R2**

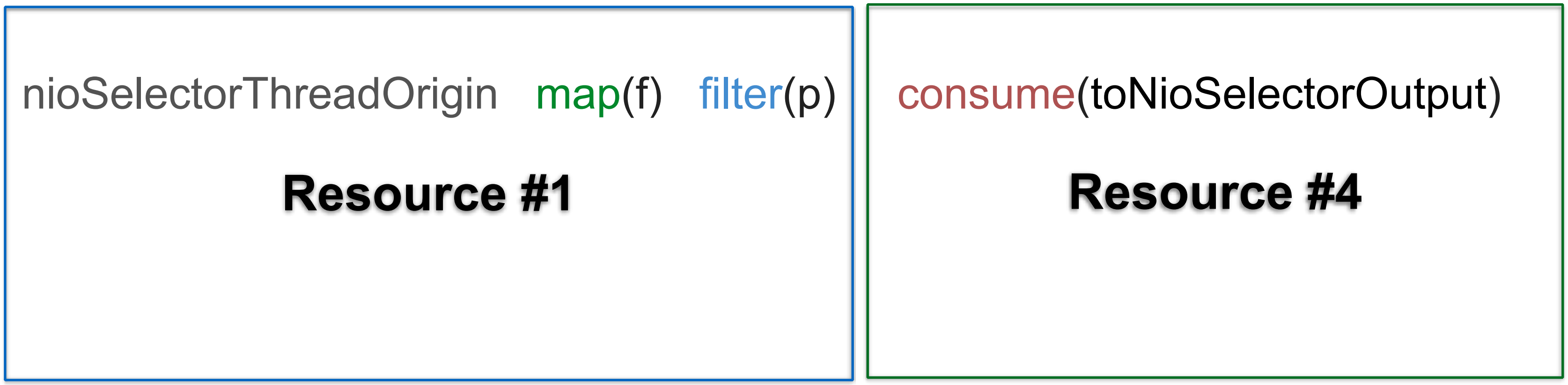
filter(p)

**R3**

consume(toNioSelectorOutput)

**Resource #4**

# Reactive Streams: Async Boundaries



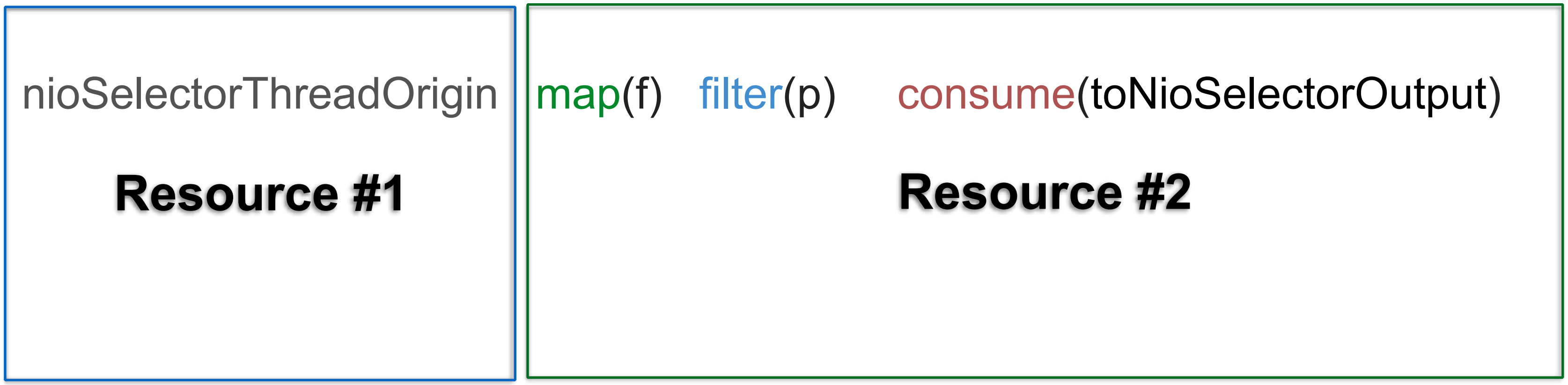
nioSelectorThreadOrigin map(f) filter(p)

**Resource #1**

consume(toNioSelectorOutput)

**Resource #4**

# Reactive Streams: Async Boundaries



nioSelectorThreadOrigin

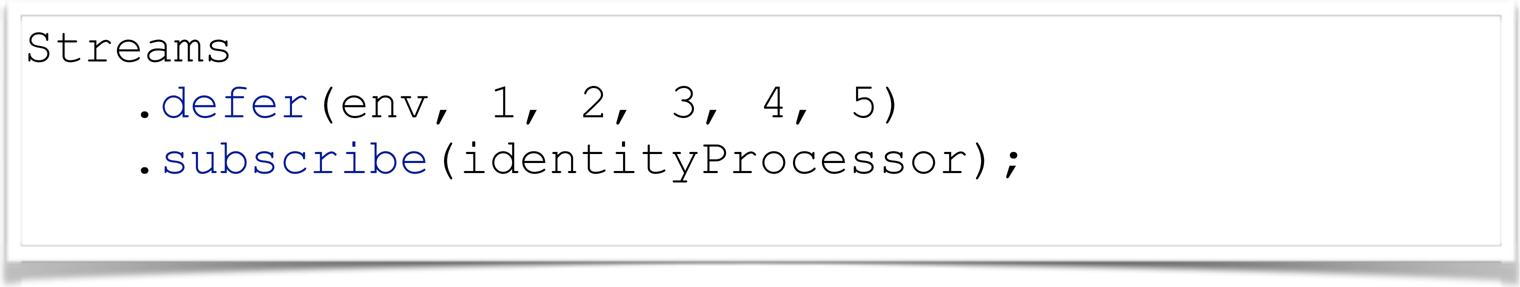
**Resource #1**

map(f) filter(p) consume(toNioSelectorOutput)

**Resource #2**

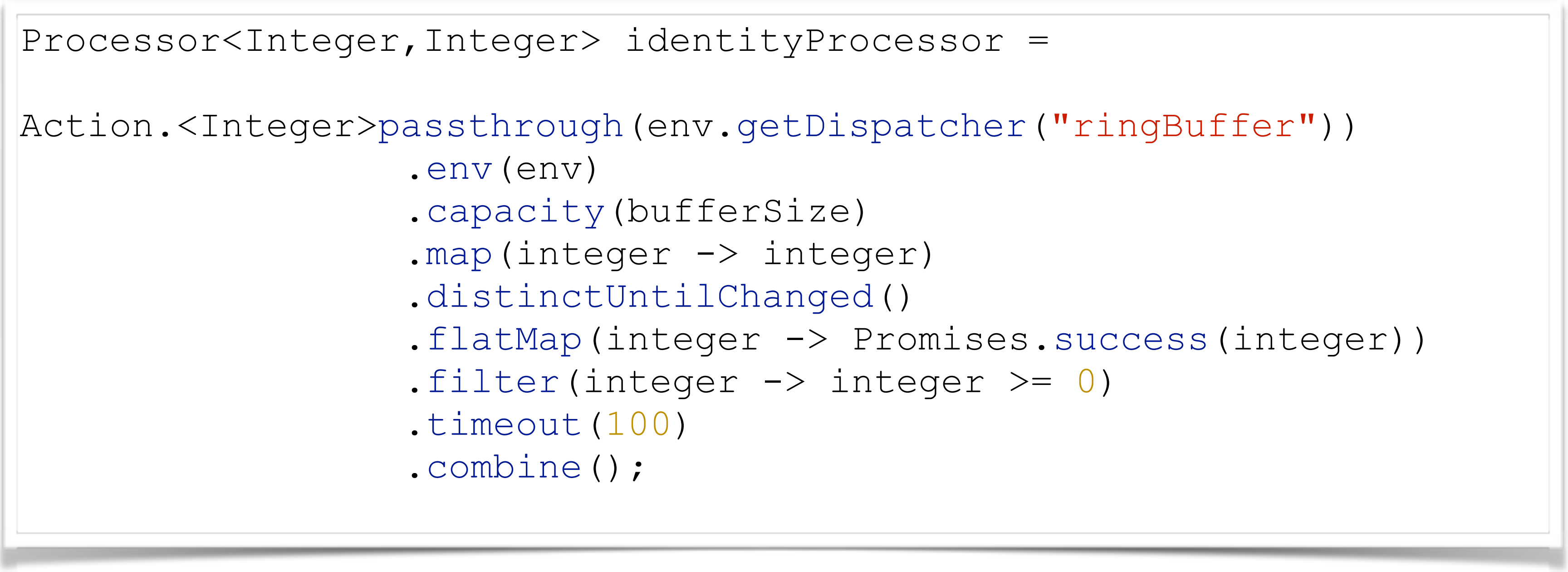
**10** slides and a **demo** to go :):):)

# Reactor : Iterable Cold Stream



```
Streams
    .defer(env, 1, 2, 3, 4, 5)
    .subscribe(identityProcessor);
```

# Reactor : Building blackbox processors



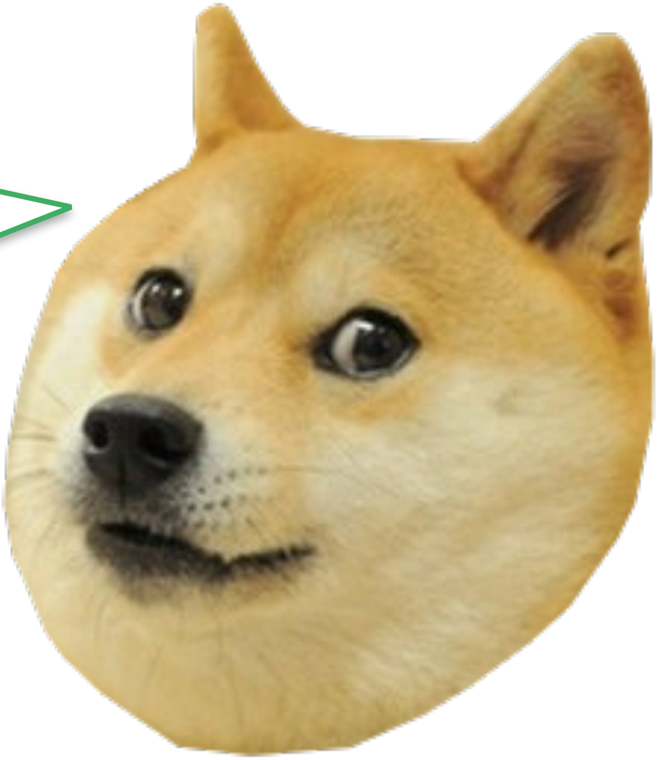
```
Processor<Integer,Integer> identityProcessor =

Action.<Integer>passthrough(env.getDispatcher("ringBuffer"))
                 .env(env)
                 .capacity(bufferSize)
                 .map(integer -> integer)
                 .distinctUntilChanged()
                 .flatMap(integer -> Promises.success(integer))
                 .filter(integer -> integer >= 0)
                 .timeout(100)
                 .combine();
```

# A Full Slide Just To Talk About *flatMap()*

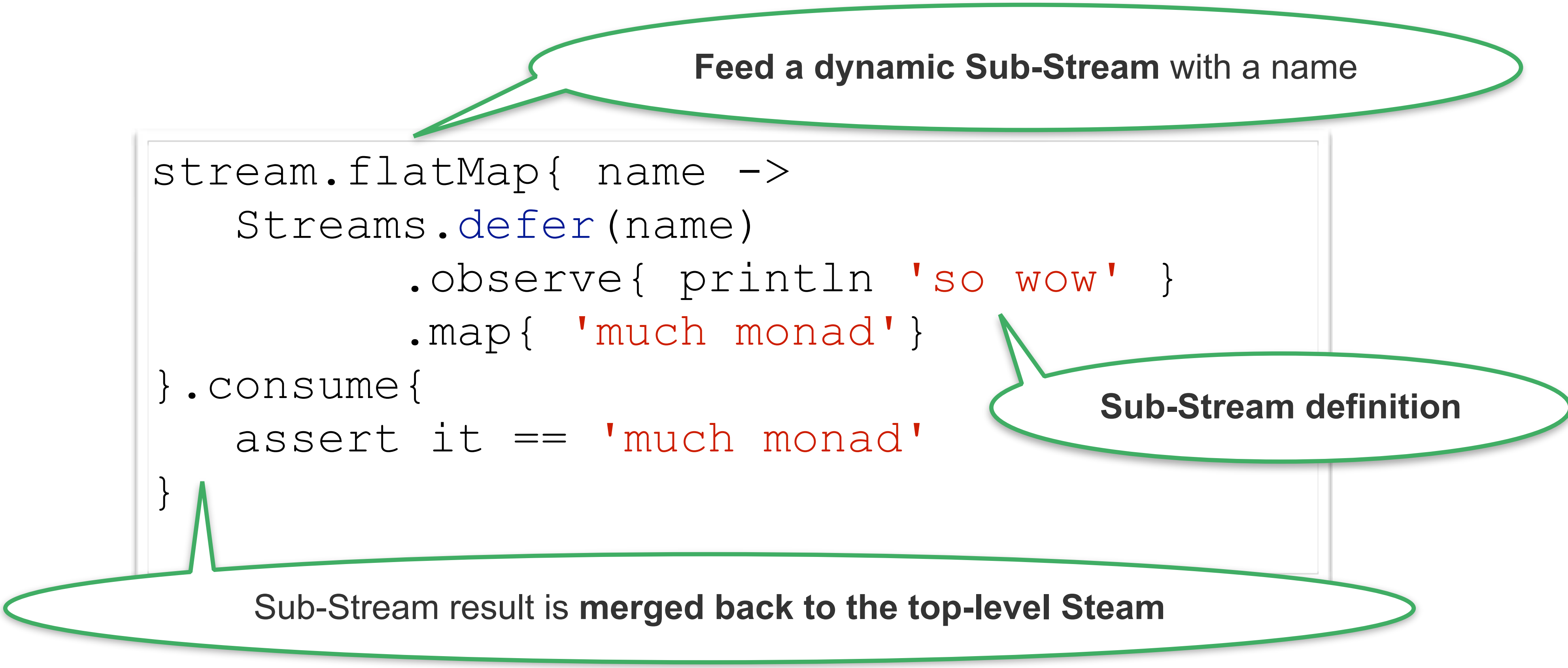
FlatMap Bucket Challenge ! Nominate 3 friends to explain *flatMap()*

# Another Slide Just To Talk About *flatMap()*

*flatMap()* is nothing more than the functional alternative to RPC. Just a way to say “Ok bind this incoming data to this sub-flow and listen for the result, dude”.



# The Last Slide about *flatMap()*



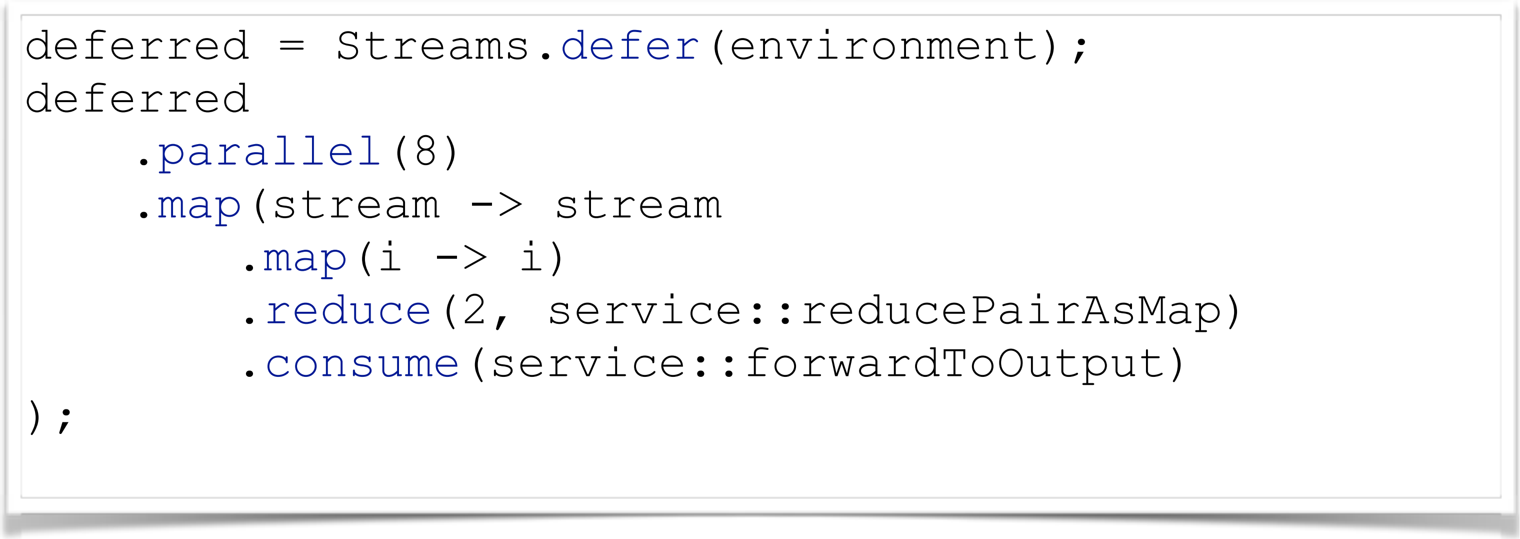
**Feed a dynamic Sub-Stream** with a name

```
stream.flatMap{ name ->
   Streams.defer(name)
          .observe{ println 'so wow' }
          .map{ 'much monad'}
}.consume{
   assert it == 'much monad'
}
```

**Sub-Stream definition**

Sub-Stream result is **merged back to the top-level Steam**

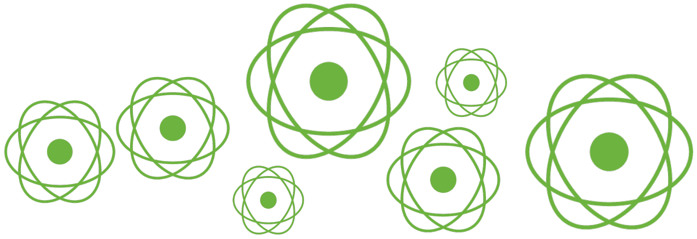
# Reactor does also Scale-Up

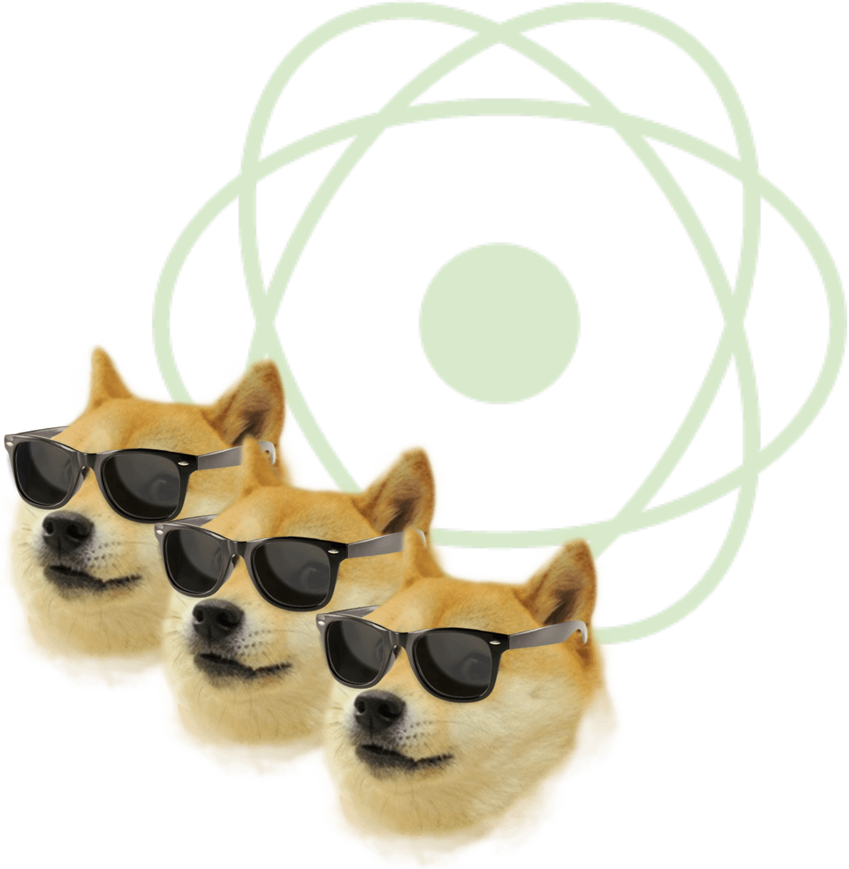


```
deferred = Streams.defer(environment);
deferred
    .parallel(8)
    .map(stream -> stream
        .map(i -> i)
        .reduce(2, service::reducePairAsMap)
        .consume(service::forwardToOutput)
);
```

# DEMO

Ratpack





# Early adopters

- **Checkpoint**
  - **Reactor 2.0.0.M1** implements **0.4.0.M2 - TCK OK**
  - **Akka Streams** implements **0.4.0.M2 - TCK OK**
  - **Experiments** started by **RxJava**
  - **Ratpack 0.9.9.SNAPSHOT** implements **0.4.0.M2 - TCK WIP**

- **Links**
  - **https://github.com/Netflix/RxJava**
  - **http://typesafe.com/blog/typesafe-announces-akka-streams**
  - **https://github.com/reactor/reactor**
  - **http://www.ratpack.io/manual/0.9.9/streams.html**

# ReactiveStreams.onSubscribe(Resources)

- **www.reactive-streams.org**
- **https://github.com/reactive-streams/reactive-streams**

- **on maven central : 0.4.0.M2**
  - **org.reactivestreams/reactive-streams**

- Current TCK preview on repo.akka.io : 0.4.0.M2-SNAPSHOT
  - org.reactivestreams/reactive-streams-tck

## ReactiveStreams.onNext(Roadmap)

- Discussed for inclusion in JDK
- Close to release: 0.4.0
  - Evaluating TCK before going 0.4 final
    - TCK coverage by **Akka Streams** and **Reactor**
    - **Need 3 passing implementations before going 1.0.0.M1**

# Reactor.onSubscribe(Resources)

- http://projectreactor.cfapps.io/
- https://github.com/reactor
- Twitter: @ProjectReactor

- on maven central : 2.0.0.M1

```
   org.projectreactor:reactor-*
```

## Reactor.onNext(Roadmap)

- Versions
    - 2.0.0.M1 out now
    - 2.0.0.M2 - November(ish)
    - **2.0.0.RELEASE** - Late 2014 early 2015
        - WIP: **additional Stream operations**, **ecosystem upgrade**, **new starting guides**

## Reactor.onError(issues)

- Tracker:
  - https://github.com/reactor/reactor/issues
- Group:
  - https://groups.google.com/forum/?#!forum/reactor-framework

# Q & A

# session.onComplete()