Reactive Thinking in Java

Yakov Fain, Farata Systems

Reactive thinking is not new

Reactive thinking is not new

Microsoft Excel - BudgetForecastsXDemoA

File Edit View Insert Format Tools Data Window Help

Verdana 8

	B	C	D	E	F	G	H	I	J	K	L	M	N
2			Happy Valley Farm										
3	Div./Department			Status	1	Enter 1 for completed status.							
4	Cut Flowers												
5	Happy Valley Farm		Start Date	Completed >	Complete								
6			Jun-06										
7	Unit Sales			Jun-06	Jul-06	Aug-06	Sep-06	Oct-06	Nov-06	Dec-06	Jan-07	Feb-07	Mar-07
8	Products	Direct Unit Cost	Totals	1	2	3	4	5	6	7	8	9	10
9	Flowers-Export	$0.27	169,000	0	5,000	6,500	7,500	10,000	20,000	20,000	20,000	20,000	20,000
10	Flowers-Local	$0.43	93,200	0	200	3,500	5,500	4,000	8,000	12,000	12,000	12,000	12,000
11	Flowers-Eldoret	$0.81	151,540	0	40	1,500	5,000	10,000	15,000	20,000	20,000	20,000	20,000
12	Revenue 4	$0.00	0	0	0	0	0	0	0	0	0	0	0
13	Revenues 5	$0.00	0	0	0	0	0	0	0	0	0	0	0
14	Total Units		413,740	0	5,240	11,500	18,000	24,000	43,000	52,000	52,000	52,000	52,000
15	Sales	Unit Prices											
16	Flowers-Export	$2.25	$380,250	$0	$11,250	$14,625	$16,875	$22,500	$45,000	$45,000	$45,000	$45,000	$45,000
17	Flowers-Local	$2.95	$274,940	$0	$590	$10,325	$16,225	$11,800	$23,600	$35,400	$35,400	$35,400	$35,400
18	Flowers-Eldoret	$3.45	$522,813	$0	$138	$5,175	$17,250	$34,500	$51,750	$69,000	$69,000	$69,000	$69,000
19	Revenue 4	$0.00	$0	$0	$0	$0	$0	$0	$0	$0	$0	$0	$0
20	Revenues 5	$0.00	$0	$0	$0	$0	$0	$0	$0	$0	$0	$0	$0
21	Total Sales		$1,178,003	$0	$11,978	$30,125	$50,350	$68,800	$120,350	$149,400	$149,400	$149,400	$149,400
22													
23	Direct Cost of Sales		$208,453	$0	$1,468	$4,475	$8,440	$12,520	$20,990	$26,760	$26,760	$26,760	$26,760
24													
25	Gross Margin		$969,550	$0	$10,510	$25,650	$41,910	$56,280	$99,360	$122,640	$122,640	$122,640	$122,640
26	Gross Margin %		82.3%	0.0%	87.7%	85.1%	83.2%	81.8%	82.6%	82.1%	82.1%	82.1%	82.1%
27													
28	Operating Expenses		$558,977	$24,700	$27,363	$31,415	$35,923	$40,036	$51,526	$58,002	$58,002	$58,002	$58,002
29	Operating Profit/Loss		-$753,566	-$24,700	-$16,853	-$5,765	$5,987	$16,244	$47,834	$64,638	$64,638	$64,638	$64,638
30	Management Charges		$60,624	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9
31	Profit/Loss		$410,507	-$24,700	-$16,854	-$5,767	$5,984	$16,240	$47,829	$64,632	$64,631	$64,630	$64,629
32	Operating Margin %		34.85%	0.00%	-140.77%	-19.14%	11.88%	23.61%	39.74%	43.26%	43.26%	43.26%	43.26%
33													
34				Jun-06	Jul-06	Aug-06	Sep-06	Oct-06	Nov-06	Dec-06	Jan-07	Feb-07	Mar-07
35	Variable Costs Budget	22.29%	Totals										
36	Variable Costs	Variable %	$262,575	$0	$2,663	$6,715	$11,223	$15,336	$26,826	$33,302	$33,302	$33,302	$33,302

License / Welcome / Capacities / Introduction / Excel / Set Up / **Year One** / Years 2-3 / Years 4-10

Ready

A reactive systems is

- **Message-driven** - components communicates via notifications
- **Responsive** - fast
- **Scalable** - works when the amount of data or # of users increases
- **Resilient** - the system is perceived as stable

www.reactivemanifesto.org

Responsivness and Java concurrency

- Blocking I/O is the problem
- `Future.get()`
  - blocks till all threads are complete
- `CompletableFuture.supplyAsync(task).thenAccept(action)`
  - what if the tasks need to fetch millions of records?

Some open-source Rx libraries

http://reactivex.io

- RxJava
  - RxAndroid, RxJavaFX, RxSwing
- Rx.NET, RxCpp, RxJS, Rx.rb, Rx.py, RxSwift, RxScala, RxPHP

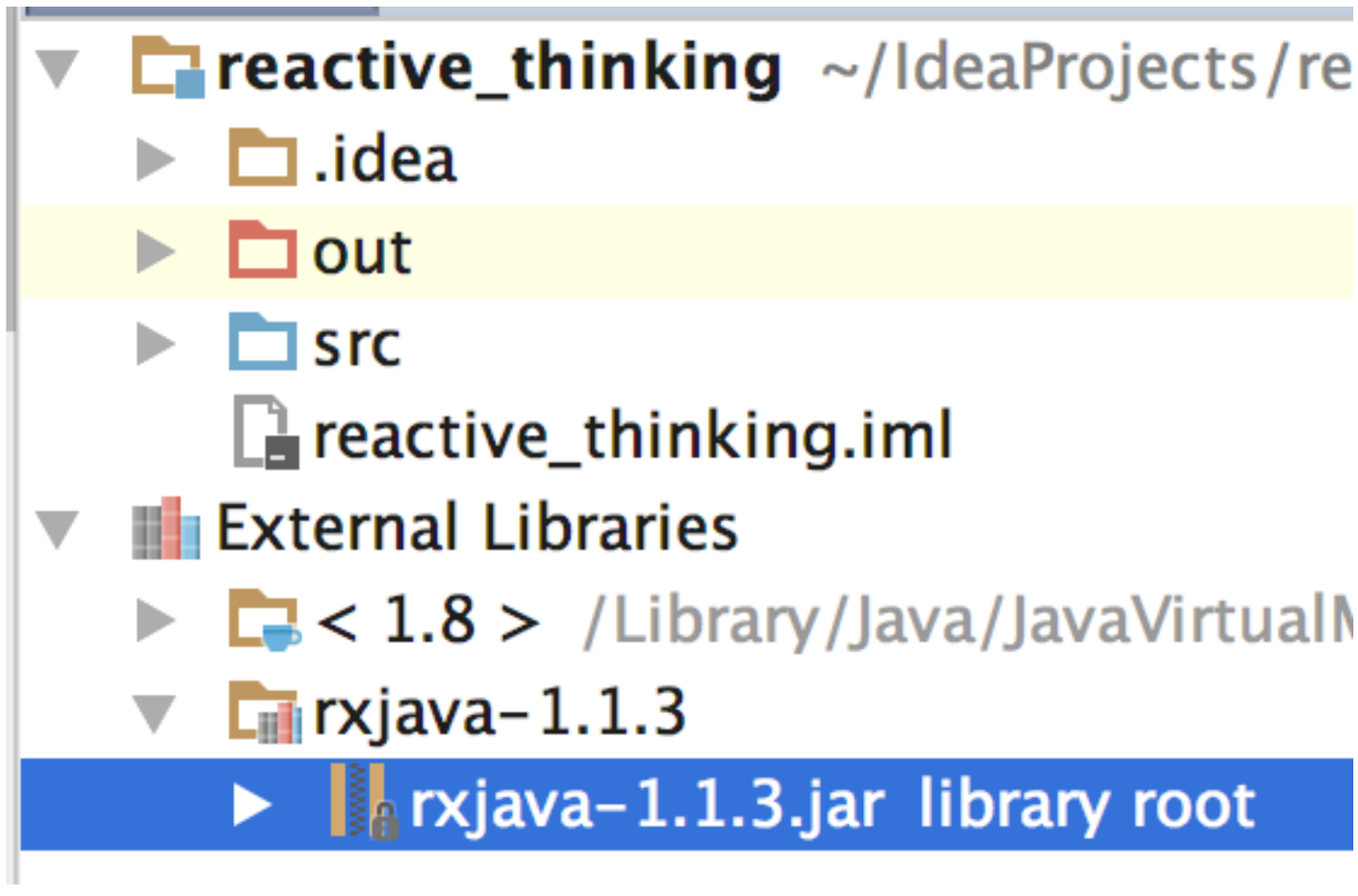
JDK 9 will include reactive streams in java.util.concurrent.Flow

Main Rx players

- **Observable** - producer of data
- **Observer** - consumer of observable sequences
- **Subscriber** - connects observer with observable
- **Operator** - en-route data transformation
- **Scheduler** - multi-threading support

Demo

HelloObservable



Get rxjava.jar on http://search.maven.org

Java Iterable: a pull

```
beers.forEach(brr -> {
            if ("USA".equals(brr.country)){
                americanBeers.add(brr);
            }
        });
```

Java 8 Stream: a pull

```
beers.stream()
        .skip(1)
        .limit(3)
        .filter(b -> "USA".equals(b.country))
        .map(b -> b.name + ": $" + b.price)
        .forEach(beer -> System.out.println(beer));
```

A fool with a tool is still a fool

A pull with a tool is still a pull

©

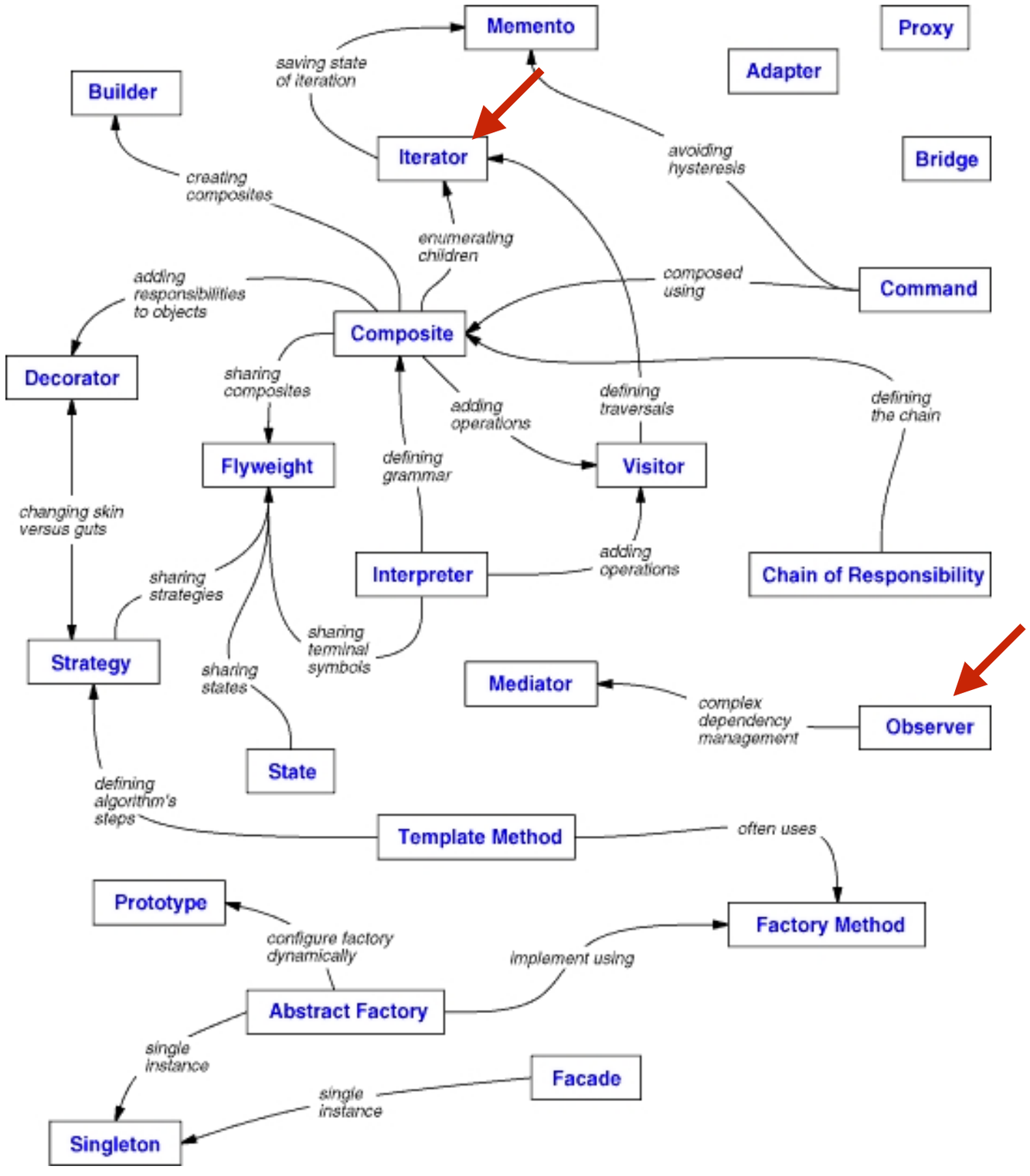
Rx Observable: a push

```
observableBeer
        .skip(1)
        .take(3)
        .filter(b -> "USA".equals(b.country))
        .map(b -> b.name + ": $" + b.price)
        .subscribe(
                beer -> System.out.println(beer),
                err -> System.out.println(err),
                () -> System.out.println("Streaming is complete")
);
```

Rx Observable: a push

```
observableBeer
        .skip(1)
        .take(3)
        .filter(b -> "USA".equals(b.country))
        .map(b -> b.name + ": $" + b.price)
        .subscribe(
                beer -> System.out.println(beer),
                err -> System.out.println(err),
                () -> System.out.println("Streaming is complete")
);
```

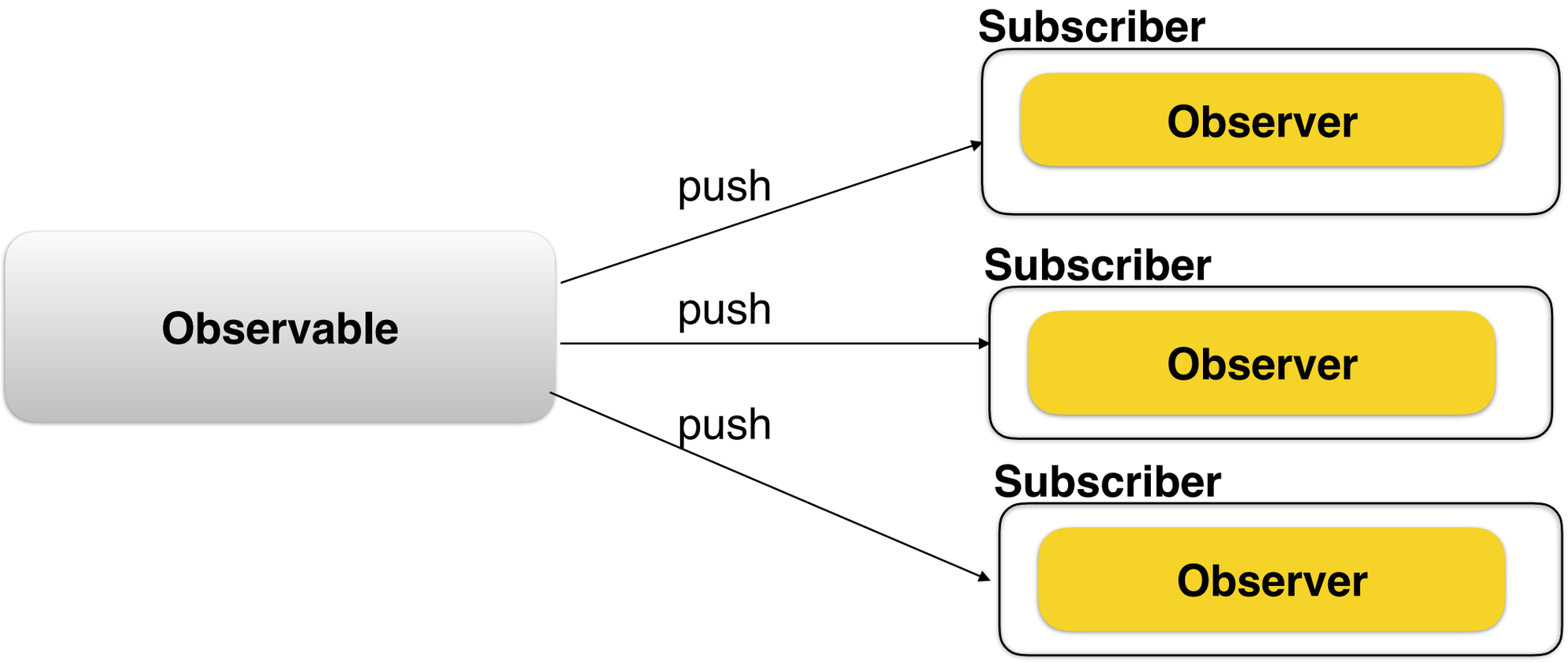
- Sync data push on the same thread
- Async data push from different threads
- Multiple consumers


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
Factory Method
implement using
Abstract Factory
single instance
Facade
single instance
Singleton


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Word	Antonym
Iterable	Observable
Iterator	Observer
Pull	Push

Event-driven push



Subscribe to messages from Observable and handle them by Observer

Learning the terms

Data Flow

Data Source

Data Flow

Observable

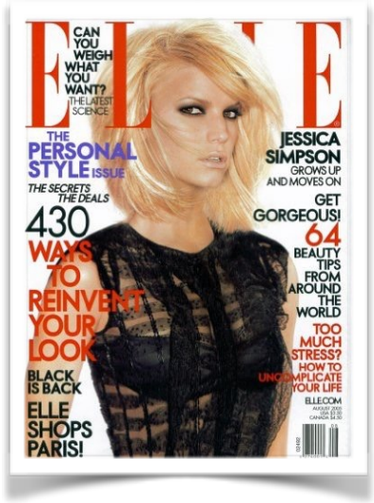
Data Source

Data Flow

Observable

Data Source

next



Data Flow

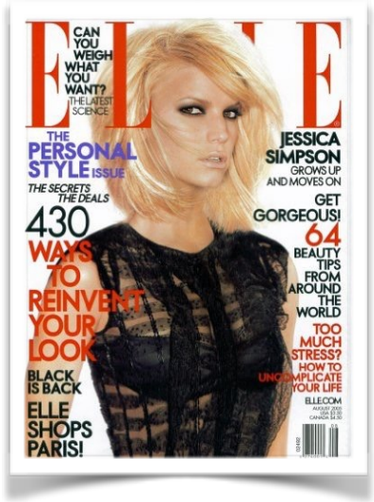
Observer

Subscriber

Observable

Data Source

next



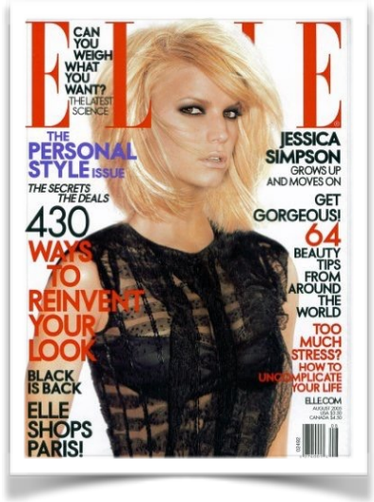
Data Flow

Observer

Subscriber

Observable

Data Source



next()

error()

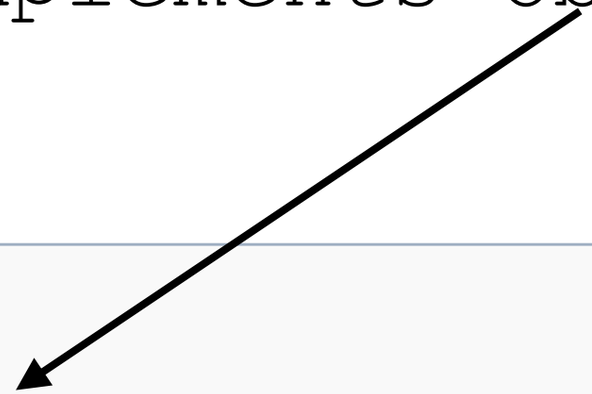
complete()

With Observable you can:

- Subscribe to its data stream
- Emit the next element to the observer
- Notify the observer about errors
- Inform the observer about the stream completion
- Transform the data with operators

```
Observable.subscribe(Subscriber)

class Subscriber implements Observer {}
```



Method Summary

Methods

Modifier and Type	Method and Description
void	**onCompleted**() Notifies the Observer that the **Observable** has finished sending push-based notifications.
void	**onError**(java.lang.Throwable e) Notifies the Observer that the **Observable** has experienced an error condition.
void	**onNext**(**T** t) Provides the Observer with a new item to observe.

Observer is an Iterable inside out

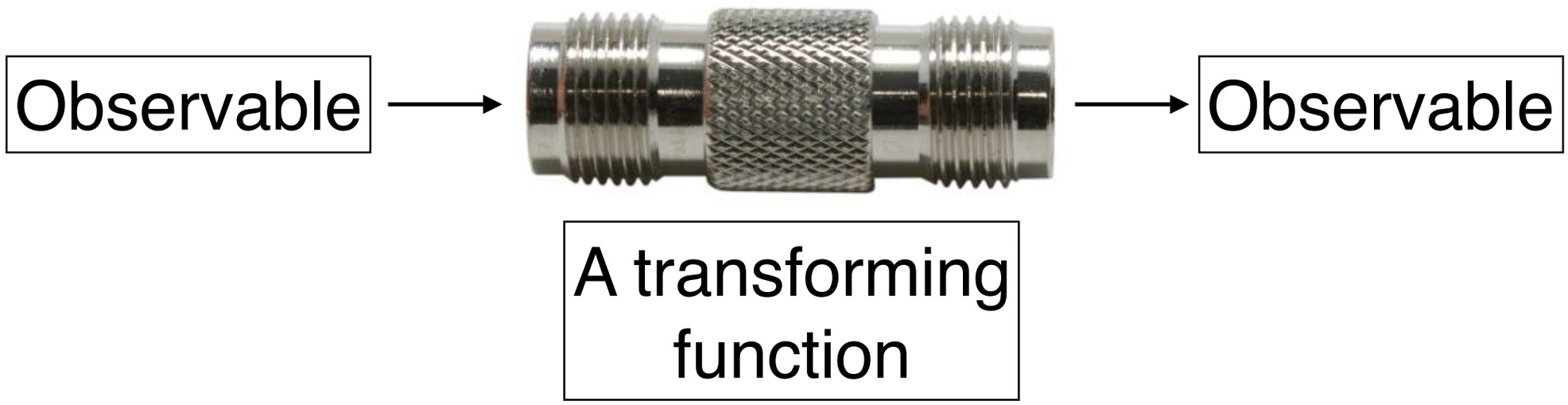
A pure function

- Produces no side effects
- The same input always results in the same output
- Doesn’t modify the input
- Doesn’t rely on the external state

A higher-order function can:

- Take one or more functions as argument(s)
- Return a function

An Operator



An Operator

Observable →  → Observable

A transforming function

pure function

```
observableBeer
        .filter(b -> "USA".equals(b.country))
```

An operator is a higher-order function

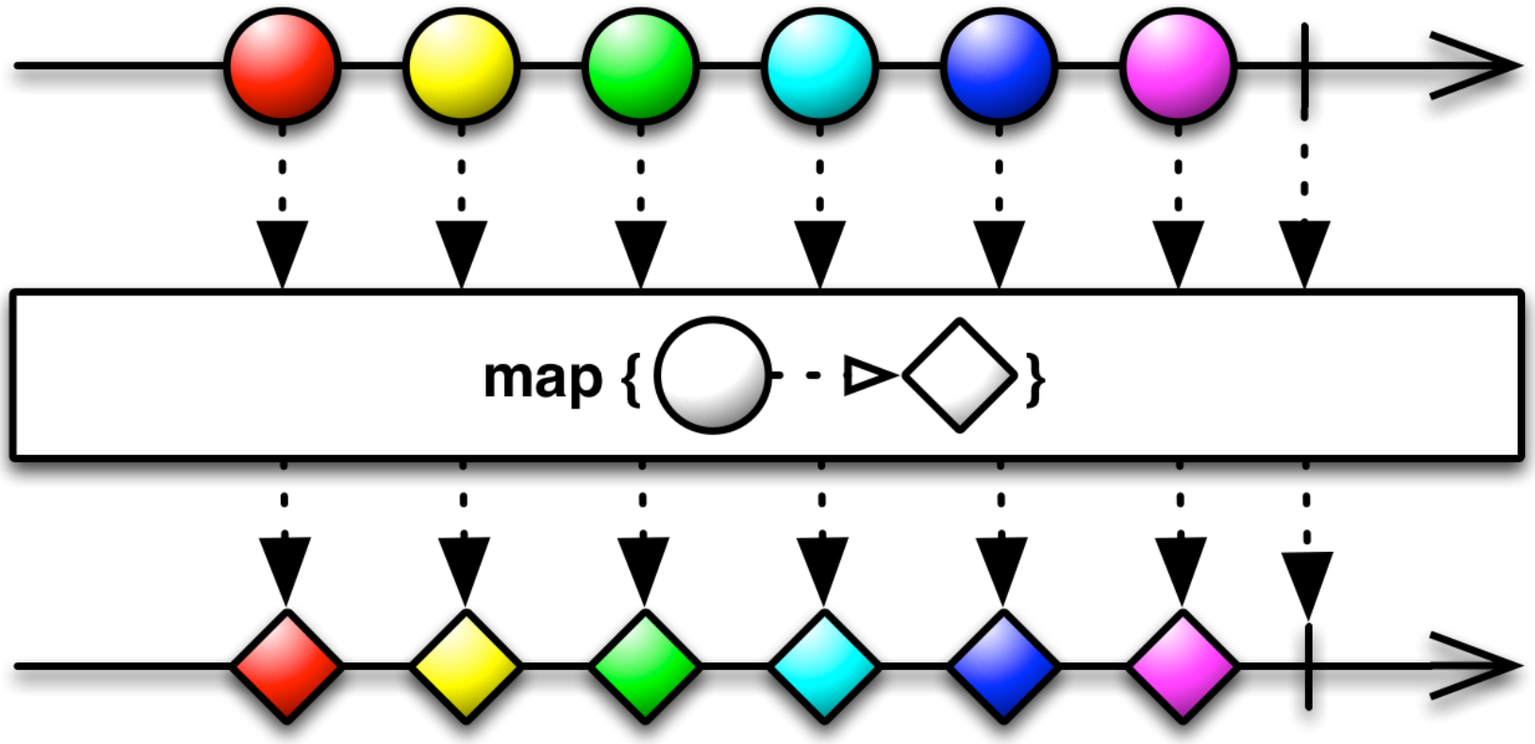
Demo

StreamVsObservable

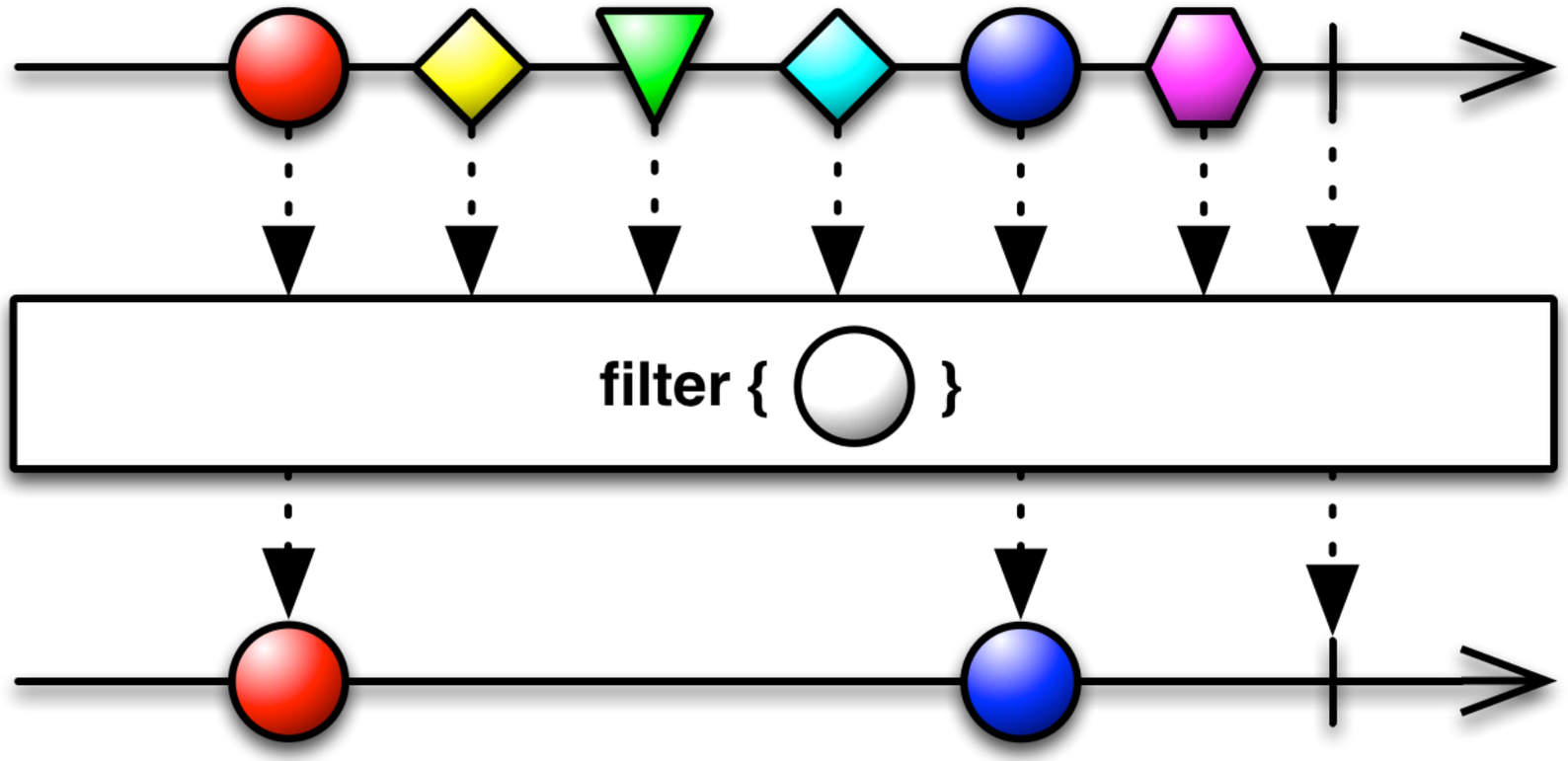
Marble Diagrams

http://rxmarbles.com

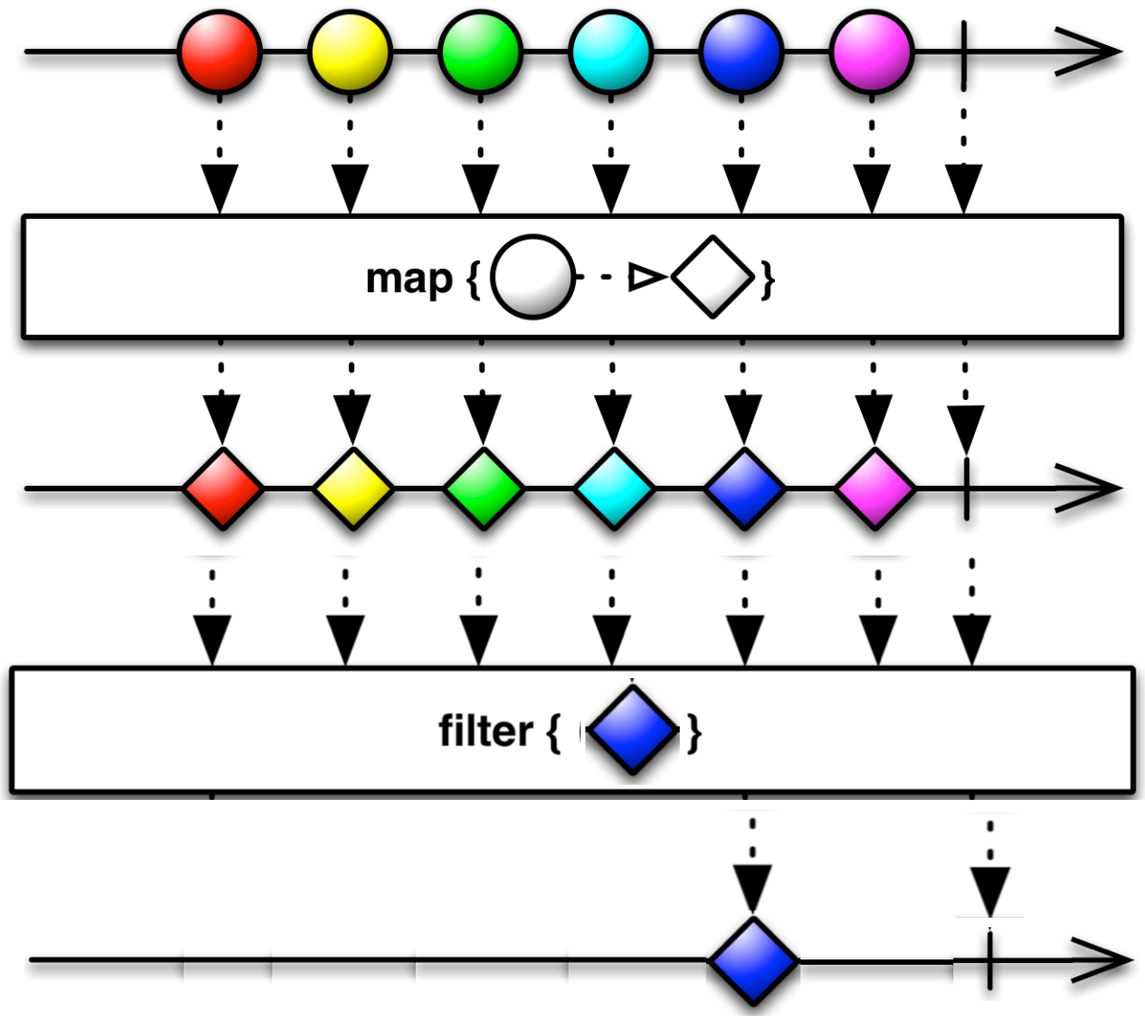
Observable map(function){}



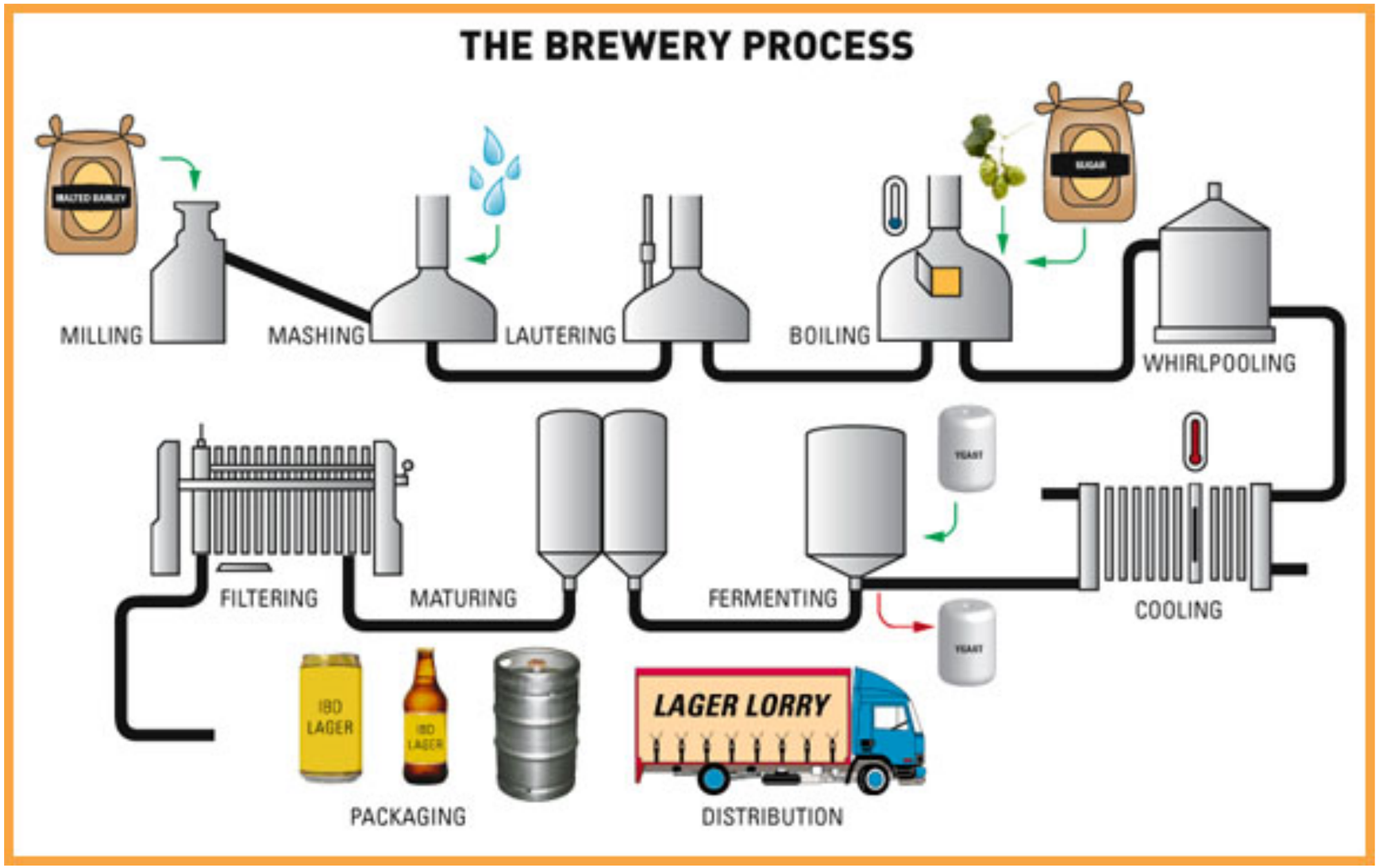
Observable filter(function){}



Operator chaining: map and filter



RX: the data moves across your algorithm



Operator

Operator

Operator

Operator

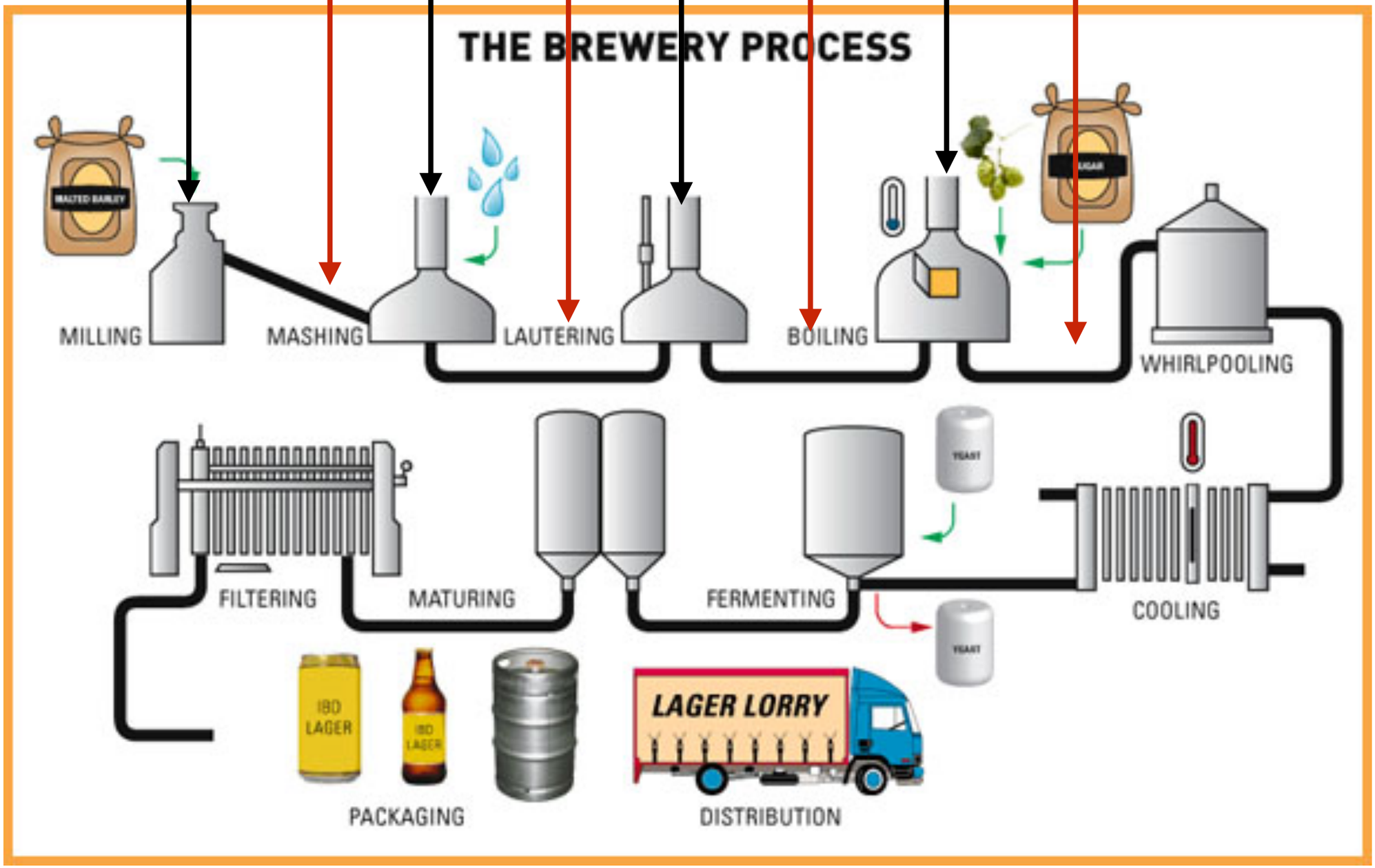
Observable

Observable

Observable

Observable

Observable



Creating an Observable

- Observable.create() - returns Observable that can invoke methods on Observer
- Observable.from() - converts an Iterable or Future into Observable
- Observable.fromCallable() - converts a Callable into Observable
- Observable.empty() - returns empty Observable that invokes onCompleted()
- Observable.range() - returns a sequence of integers in the specified range
- Observable.just() - converts up to 10 items into Observable

Demo

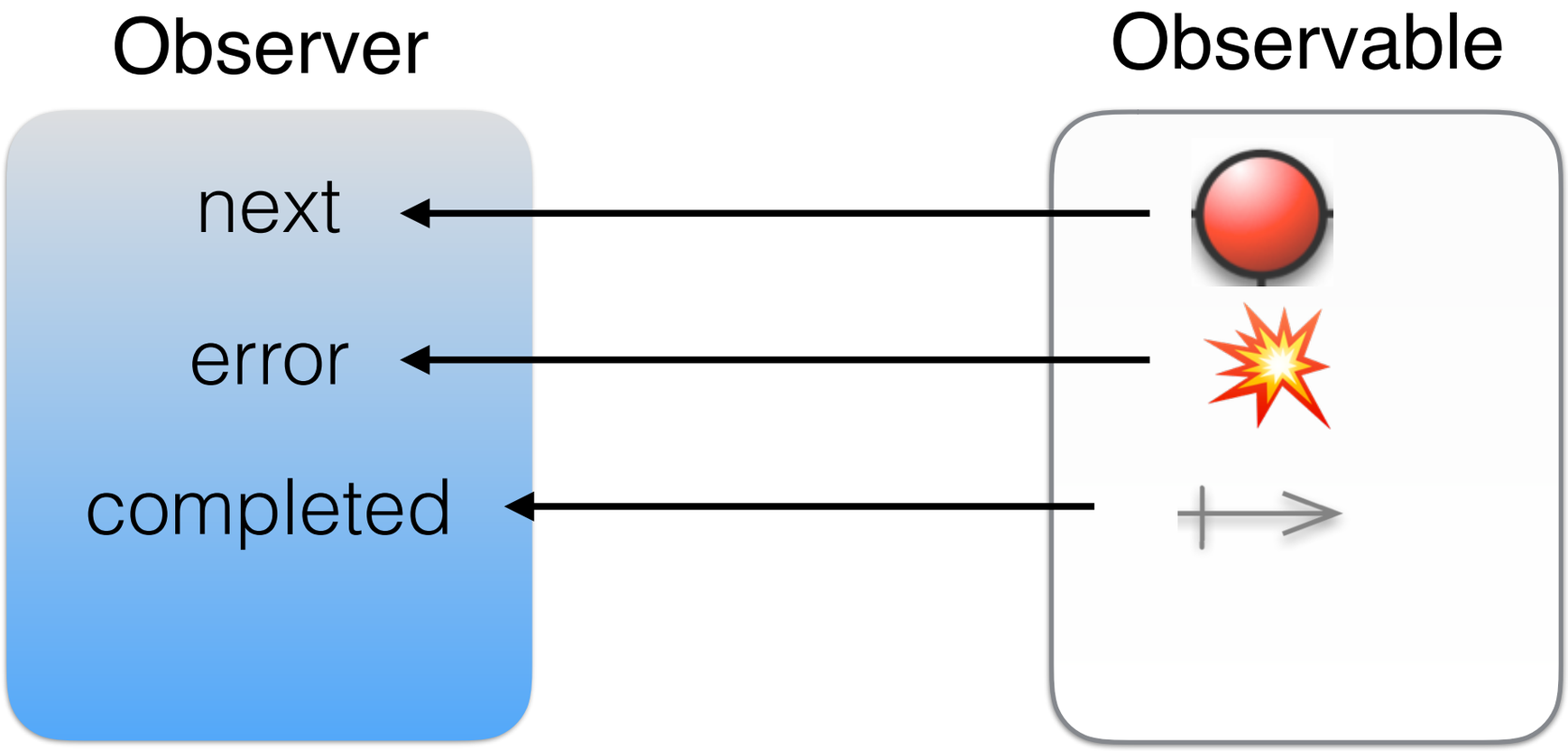
BeerClient

Functions with side effects

Affect environment outside the function.
Don’t affect the Observable items.

- doOnNext()
- doOnError()
- doOnCompleted()
- doOnEach()
- doOnSubscribe()
- doOnXXX()

Error Handling



Error-handling operators

- Errors sent using **onError()** kill the subscription

- **retryWhen()** - intercept, anylize the error, resubscribe

- onErrorResumeNext() - used for failover to another Observable

- onResumeReturn() - returns an app-specific value

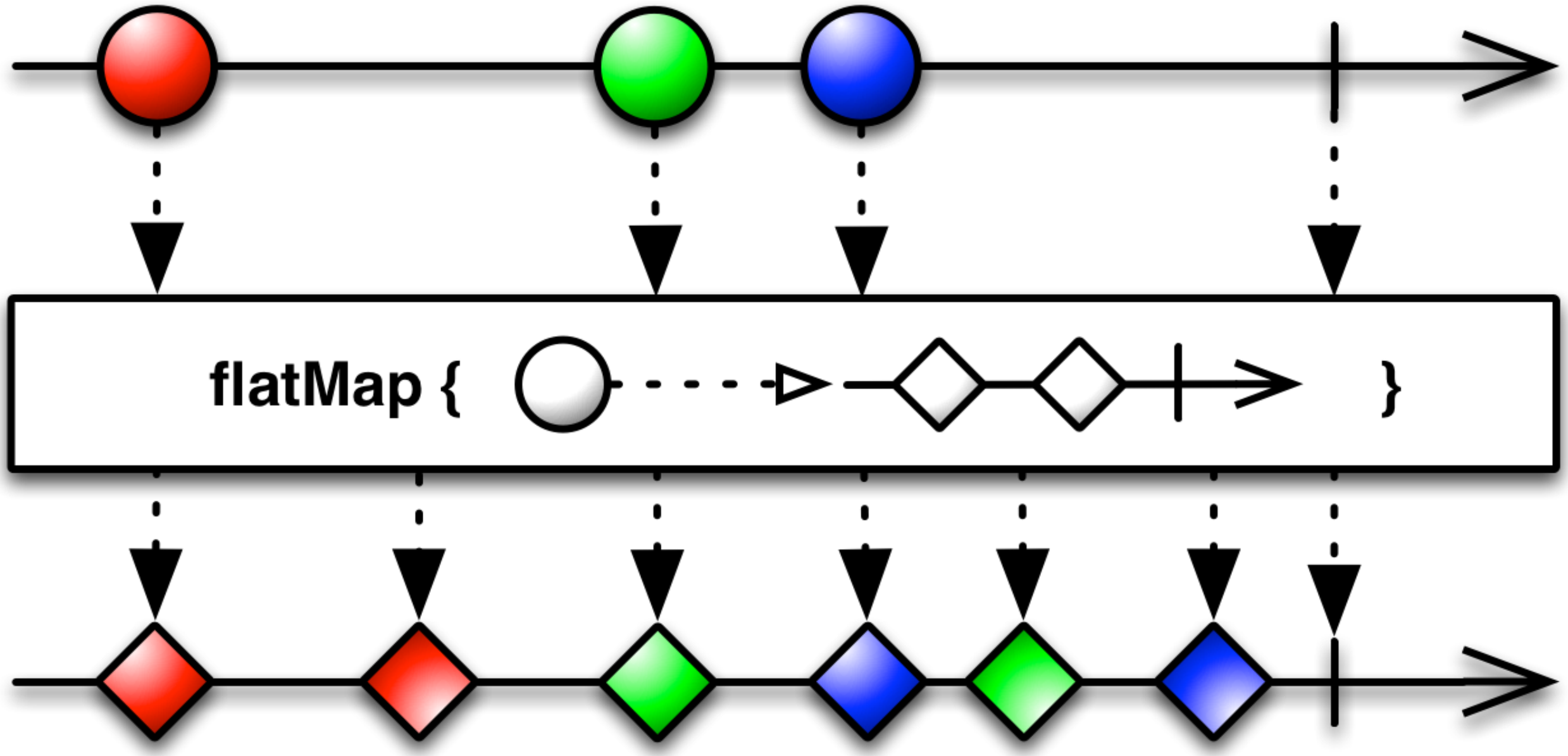
Error-handling operators

- Errors sent using onError() kill the subscription
- **retryWhen()** - intercept/ anylize the error/ resubscribe
- **onErrorResumeNext()** - used for failover to another Observable

Demo

BeerClientWithFailover

The flatMap() operator

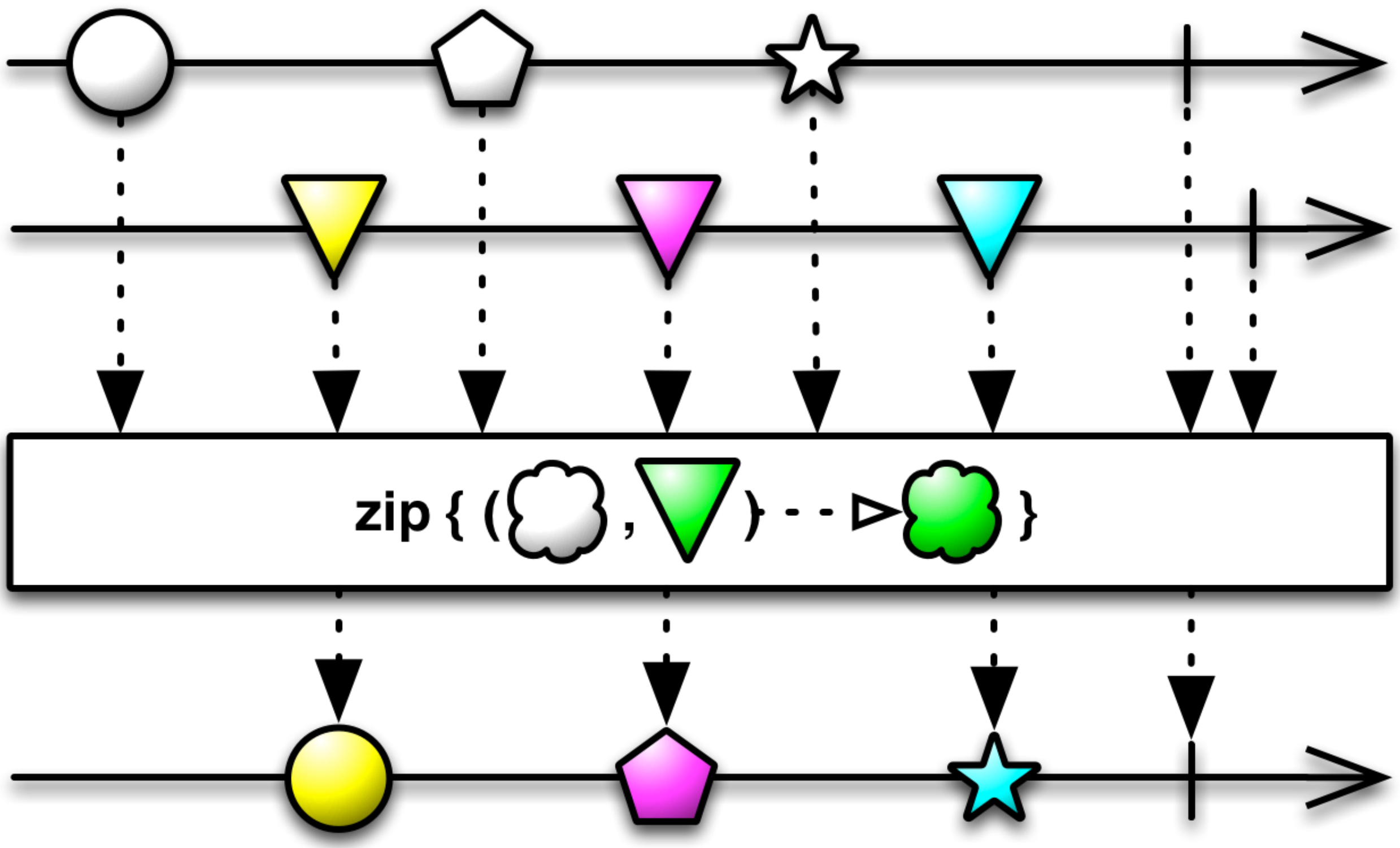


Observable.flatMap()

Demo

ObservableDrinks

The zip() operator



Think Netflix

- take(10) - get a list of lists of videos (the first 10)
- flatmap() - turn them into one list of videos
- map() for each video make nested calls to get some observable metadata (ratings, reviews, etc.)
- zip() - combine observables into a a video object
- observeOn() - switch to the UI thread and render the videos

Schedulers

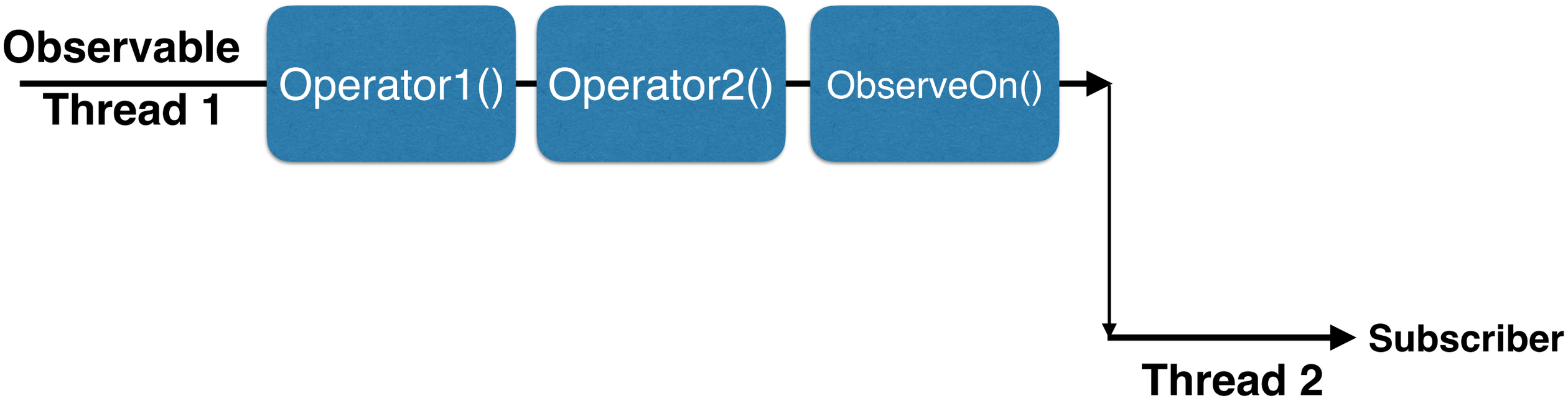
Concurrency with Schedulers

- An observable stream is single-threaded by default
- `subscribeOn(strategy)` - run Observable in a separate thread
- `observeOn(strategy)` - run Observer in a separate thread
- Standard schedulers run on daemon threads

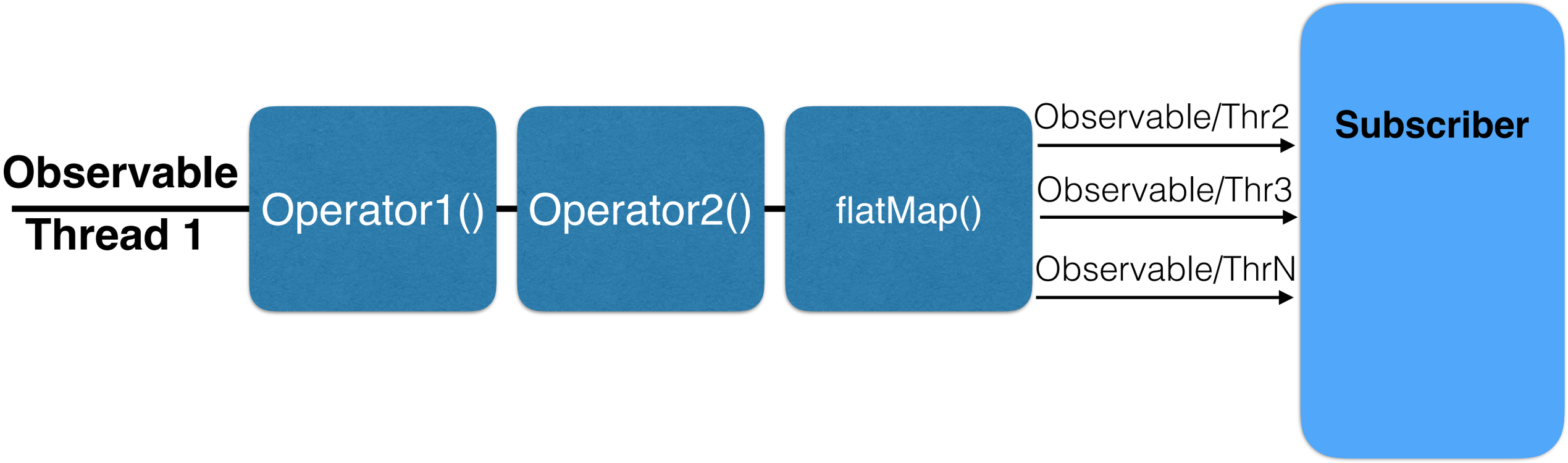
Multi-threading strategies

- Schedulers.computation() - for computations: # of threads <= # of cores
- Schedulers.io() - for long running communications; backed by a thread pool
- Schedulers.newThread() - new thread fo each unit of work
- Schedulers.from(Executor) - a wrapper for Java Executor
- Scedulers.trampoline() - queues the work on the current thread
- AndroidSchedulers.mainThread() - handle data on the main thread (RxAndroid)

Switching threads



Parallel processing



Demo

SubscribeOnObserveOn
ParallelStreams

Hot vs Cold Observables

Cold - begin emitting items when Observer desires.
Observer can control the emission rate.

Hot - begin emitting on creation.
Subscribers get items starting from the current one.

Backpressure

- Too much data is coming in. The consumer is too slow.
- Can happen only in the async mode

Dealing with backpressure

- Throttling: sample(), throttleFirst(), debounce()
- Buffers: buffer()
- Window: window(n)
- Reactive pull: Subscriber.request(n)

https://github.com/ReactiveX/RxJava/wiki/Backpressure

Http and Observables

```
class AppComponent {

  products: Array<string> = [];

  constructor(private http: Http) {

    this.http.get('http://localhost:8080/products')
        .map(res => res.json())
        .subscribe(
            data => {
                this.products=data;
            },
            err =>
              console.log("Can't get products. Error code: %s, URL: %s ",
                                                    err.status, err.url),
            () => console.log('Product(s) are retrieved')
        );
  }
}
```

Observer

Links

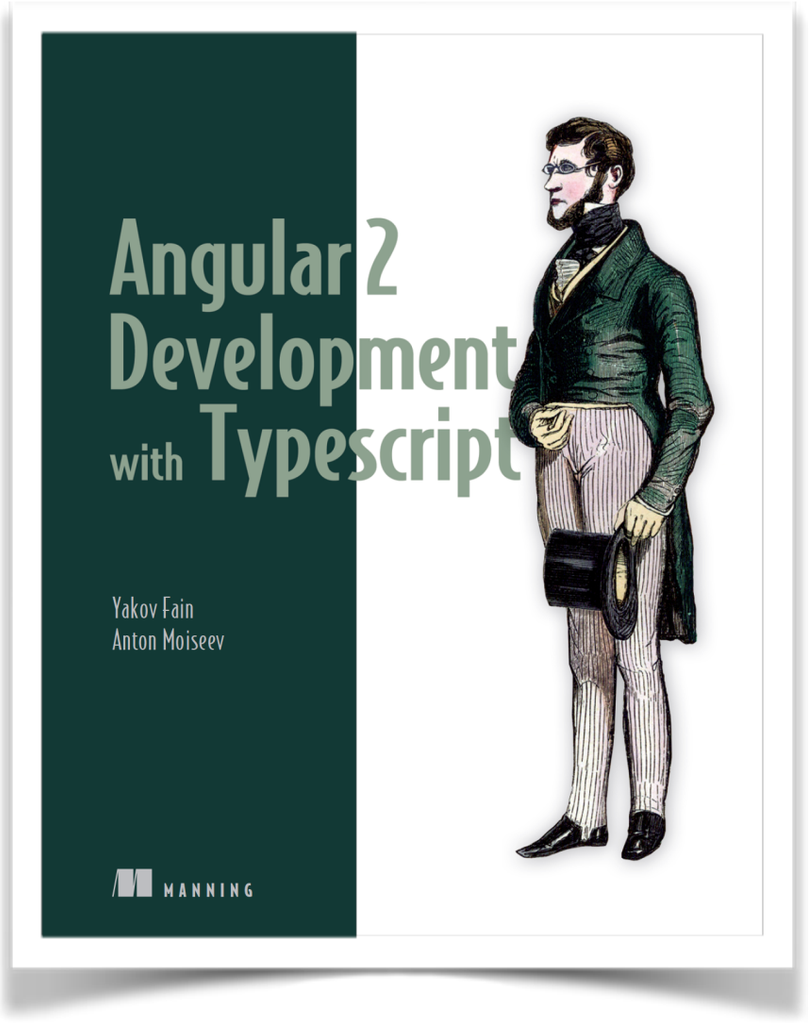
- Code samples:

  https://github.com/yfain/rxjava

- Our company: faratasystems.com

- Blog: yakovfain.com

- Twitter:@yfain



discount code: faindz