Reactive Programming

JHUG May 2017

Anastasopoulos Spyros
@anastasop

What's in a word?

- Reactive Systems
  - Patterns for responsive, resilient, scalable, message driven systems
- Reactive Programming
  - An API for asynchronous programming with observable streams
- Functional Reactive Programming
  - $y[t] = ax[t] + b$
  - $z[t] = y[t] * c$

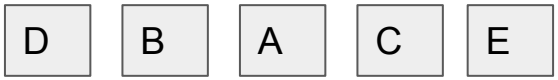
All these are different things and one does not imply the other

Reactive Programming

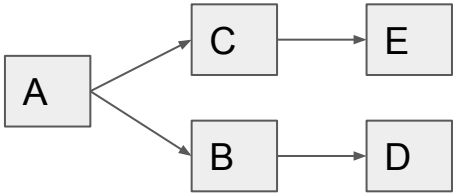
A high level tool for writing concurrent programs

Sequential Programming

Split a large task into smaller tasks



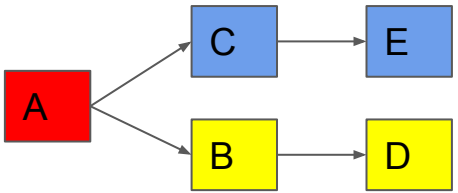
Structure the dependencies and the error boundaries



Weave them into an sequential execution on the CPU

Concurrent Programming

- Modern machines have multiple cores, we can and should use them
- Programming as the composition of independently executing processes
- A process in the generic sense is a locus of points of control (Lamport)
- Processes are executed during overlapping time periods
- Concurrency is not parallelism An excellent talk by Rob Pike
  - Concurrency is about dealing with lots of things at once (composition)
  - Parallelism is about doing lots of things at once (execution)



Concurrent Programming - Is it a choice?

- Modern machines are very fast, my program will not run fast?
  - Resources utilization: machines are faster but traffic volumes are also bigger
  - Even if it runs fast, concurrent programs are better structured
- We program networked services and multi-core machines
  - Inherited distribution and asynchrony can only be handled with concurrency
- Only way to interface with the parallel real world
  - Users, connections, events, devices etc

Concurrent Programming - Is it easy?

Concurrent Programming is hard

Needs good tools

Java 2: `Thread, synchronized`

Java 5: `Lock, Atomic, Future, Executor`

Java 7: `fork/join`

Java 8: `Streams, CompletableFuture`

JavaEE: The frameworks and the application server handle the concurrency

Impossible to reason about the code properties with such low level tools

Concurrent Programming need not be that hard

Must go high level

- Actors (ownership, no sharing)
- CSP - Communicating Sequential Processes (share by communicating)
- DataFlows, Functional Programming (immutability)

Languages like **clojure, elixir, go, scala** use such models

Concurrent Programming in Java

Goals

- Resource utilization - cpus, io should be fully utilized
- Fairness - all threads get equal chances to run
- Convenience - the program is well structured and we can verify properties

Obstacles

- Safety hazards - locks, race conditions, mutable state
- Liveness hazards - locks, deadlocks
- Performance hazards - locks, threading overhead

Concurrent Programming in Java

What do we want? **Scaling**

- Resource utilization - cpus, io are fully utilized
- Fairness - all threads get equal chances to run
- Convenience - the program is well structured

What do we cope with? **Memory Model (java threads communicate by sharing)**

- Safety hazards - locks, race conditions, mutable state
- Liveness hazards - locks, deadlocks
- Performance hazards - locks, threading overhead

Concurrent Programming - What do we want?

A high level tool for concurrency that

- Handles the concurrency and as much as possible hides it
- Makes it easy to write concurrent programs and verify their properties
- Makes it easy to compose concurrent programs

Reactive Extensions (Rx)

Compose asynchronous and event-based programs using observable sequences

- Original design by Microsoft for .Net and JS
- Netflix ported it to Java and popularized it
- A spec is being written and will be included in JDK 9
- Vendors slowly adopt it
  - RxJava
  - Reactive Spring, Spring reactor
  - RxNetty
  - Vert.x Reactive streams
  - Akka streams

The next slides demonstrate the concepts with RxJava and some custom notation

Reactive Extensions (Rx)

Everything is designed around the **`Observable<T>`** and **`Observer<T>`** interfaces

```
Observable.fromArray(1, 2, 3).subscribe(s -> System.out::println)
```

Observable<T>

A *stream*, a possible infinite source of data. Unifies in-memory collections (**Iterable<T>**) and the results of async computations usually associated with a callback (**Observer/subscriber pattern**)

Generalizes Observer pattern for async tasks		
	Single Item	Multiple Items
sync	`T getData()`	`Iterable<T> getData()`
async	`Future<T> getData()`	`Observable<T> getData()`

Turns iterator inside out	
Iterator *pulls*	Observable *pushes*
`T = it.next()`	`observer.onNext(T)`

Observable<T>

- Provides a possibly infinite stream of data to its observers
- Abstracts the source of data from the client
- Has total control of concurrency
- Owner of the API retains control of concurrency behavior
  - Synchronous
  - Asynchronous with event driven IO
  - Asynchronous with thread pool
- The client treats everything as async
- Interactions with the API are async and declarative
  - `Observable.sample().map().filter()`

Observer<T>

An API of 4 callbacks.

`onSubscribe()` // will be called then the Observable is about to send items

`onNext(T t)` // will be called many times with the subsequent item of the stream

`onComplete()` // will be called once when the Observable has no more items

`onError(Throwable t)` // will be called once if the Observable fails

Note: If **`onComplete()`** or **`onError()`** is called, the stream closes

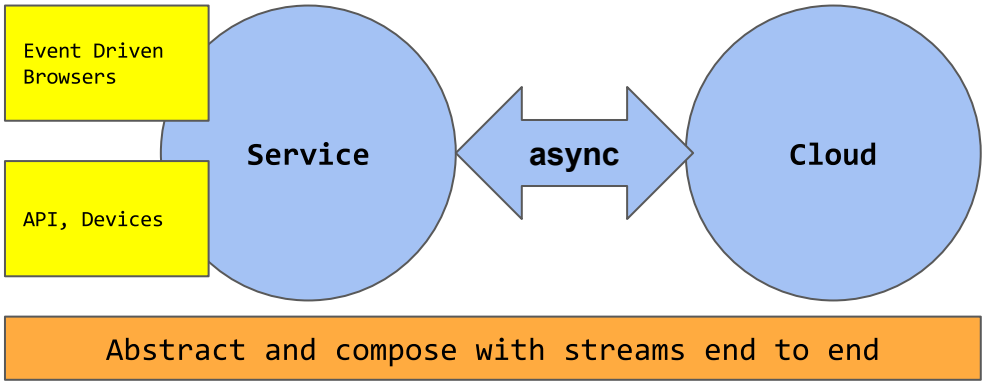
Reactive Extensions (Rx) - Examples

```
Observable<String> titles = Crawler.titlesForHosts(“www.jhug.gr”, ...);
titles.subscribe(title -> System.out::println);

Observable<String> titles = Crawler.titlesForHosts(“www.jhug.gr”, ...);
Observable<String> feeds = Cache.rssForHosts(“www.jhug.gr”, ...);
Observable.zip(titles, feeds, (t, f) -> [t,f]).subscribe(t -> System.out::println);

Observable<String> importantKeywords =
Crawler.contentForHosts(“www.jhug.gr”, ...)
.flatMap(html -> Indexer.keywords(html))
.subscribe(t -> System.out::println)
```

Technicalities

- The emission of data starts after an observer subscribes (cold observable) or when the observable is created (hot observable). In the latter case the observer of course misses the already emitted items.
- The observer callbacks run on the main thread. The API provides the `Scheduler` abstraction to run them on preconfigured pools
- The spec supports *backpressure* i.e the observer can control the flow of data from the observable
- Everything is async, **`blocking`** calls are explicit
- All operations on observables (map, take, filter) are lazy and deferred for when an observer subscribes
- An observable can have many observers and can unicast or multicast

The prototypical Rx application



Composing Observables

The composing of concurrent components is notoriously difficult

- `Future<T>` is hard to compose conditionally and asynchronously
- `Future<T>.get()` is blocking and difficult to decide when to call it
- Callbacks scatter the logic and state among the code
- Every library provides some combination of the above for it's API so combining different libraries is not straightforward

Rx composes observables using functional operators

- Uniform composition model
- Easier to deduce properties of the code like blocking, state mutation, error propagation

Rx Functional Operators - a sample

Create: `create, defer, from`

Transform: `map, flatMap, scan`

Filter: `filter, sample, take`

Combine: `zip, switch, merge`

Conditional: `all, amb`

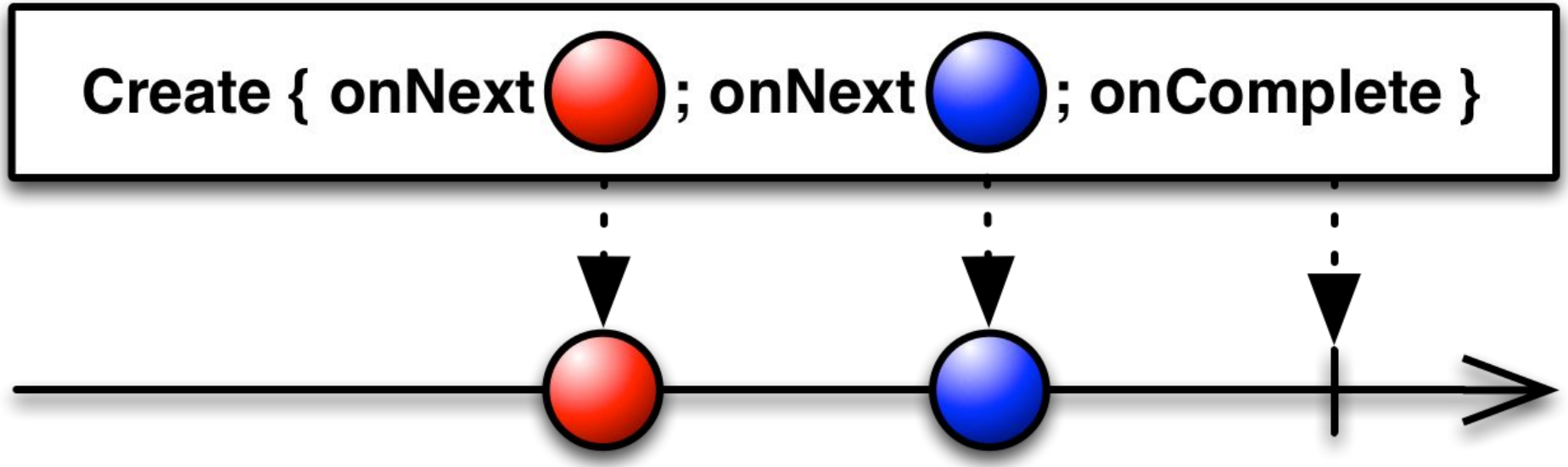
Aggregate: `reduce, concat`

Credits

All the following marble diagrams and descriptions of the operators are from http://reactivex.io licensed under Creative Commons Attribution 3.0

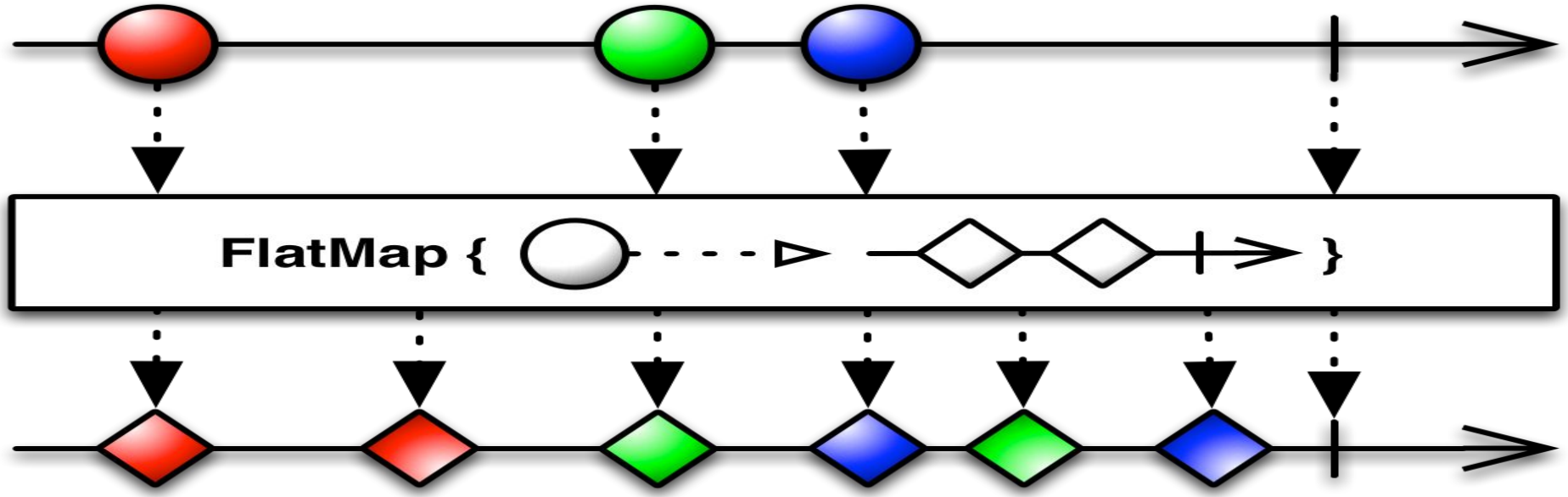
`create`

create an Observable from scratch by means of a function



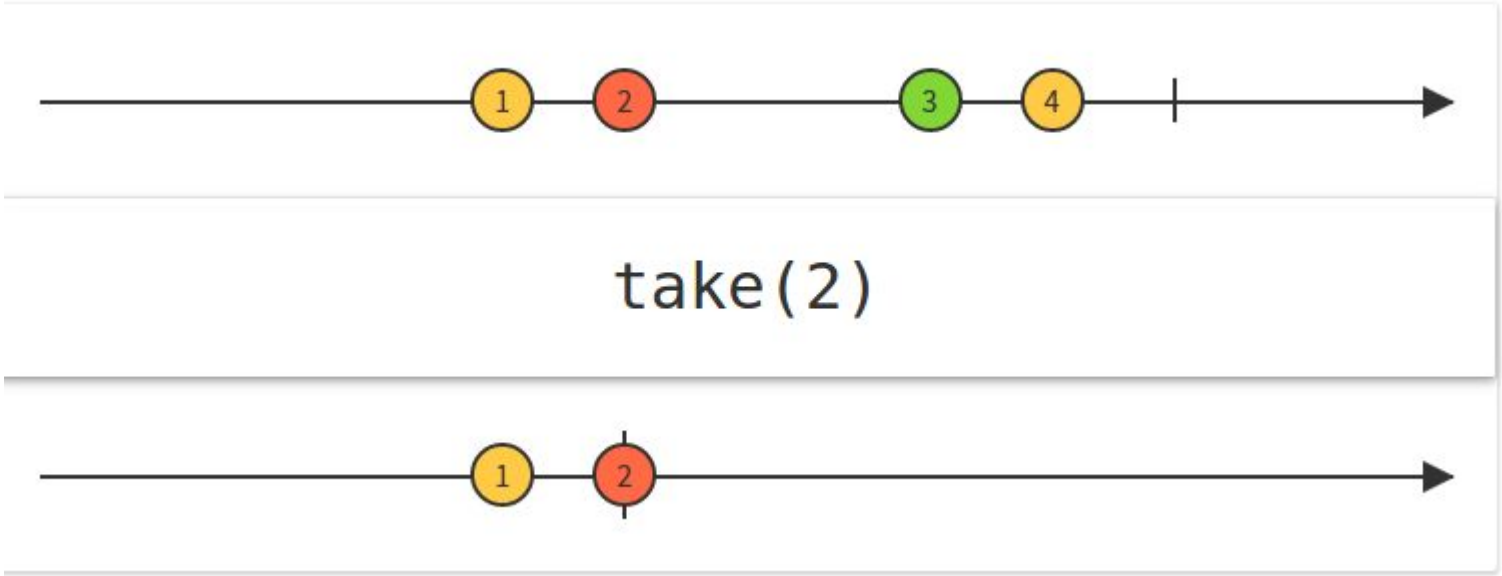
flatMap

transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable



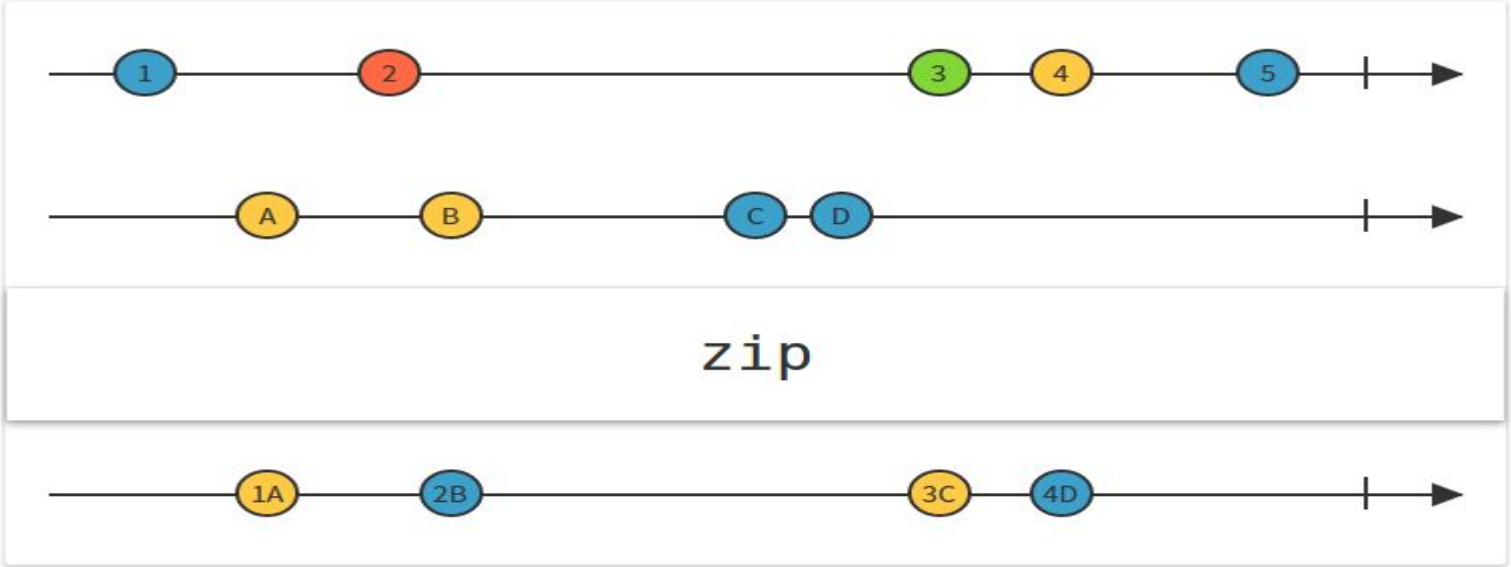
take

emit only the first n items emitted by an Observable



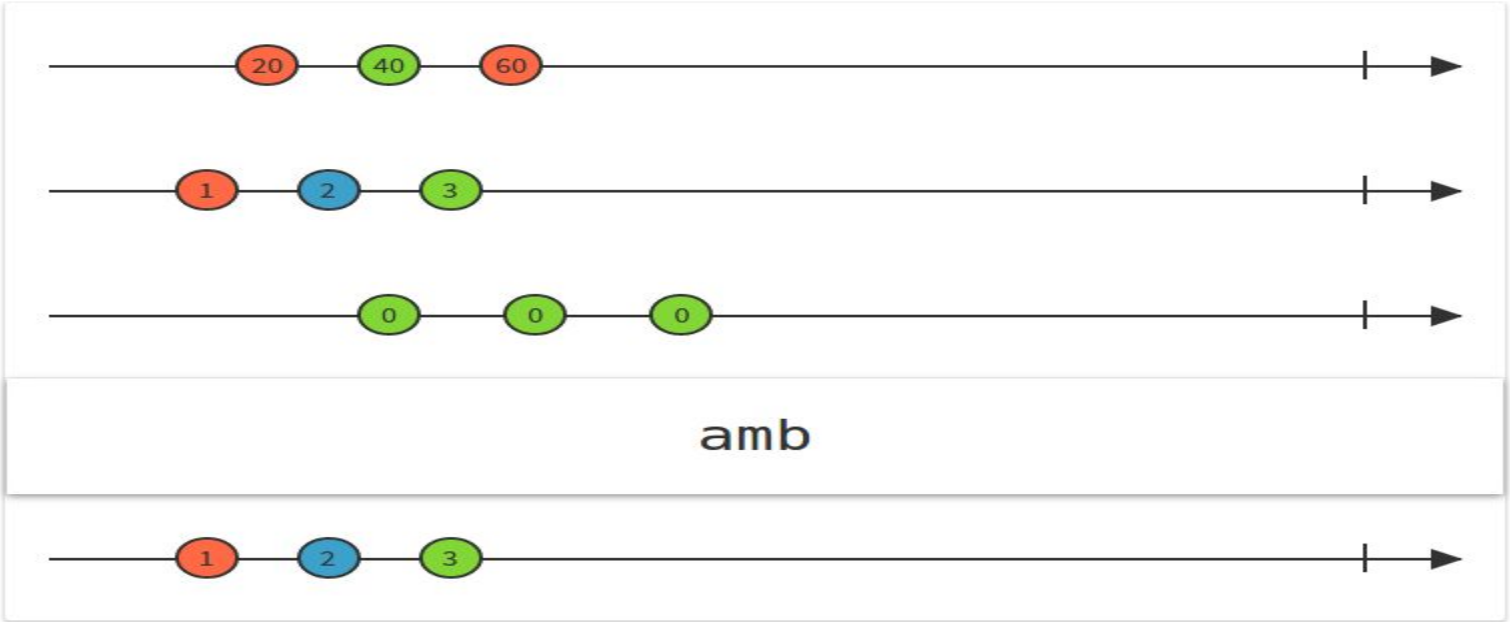
zip

combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function



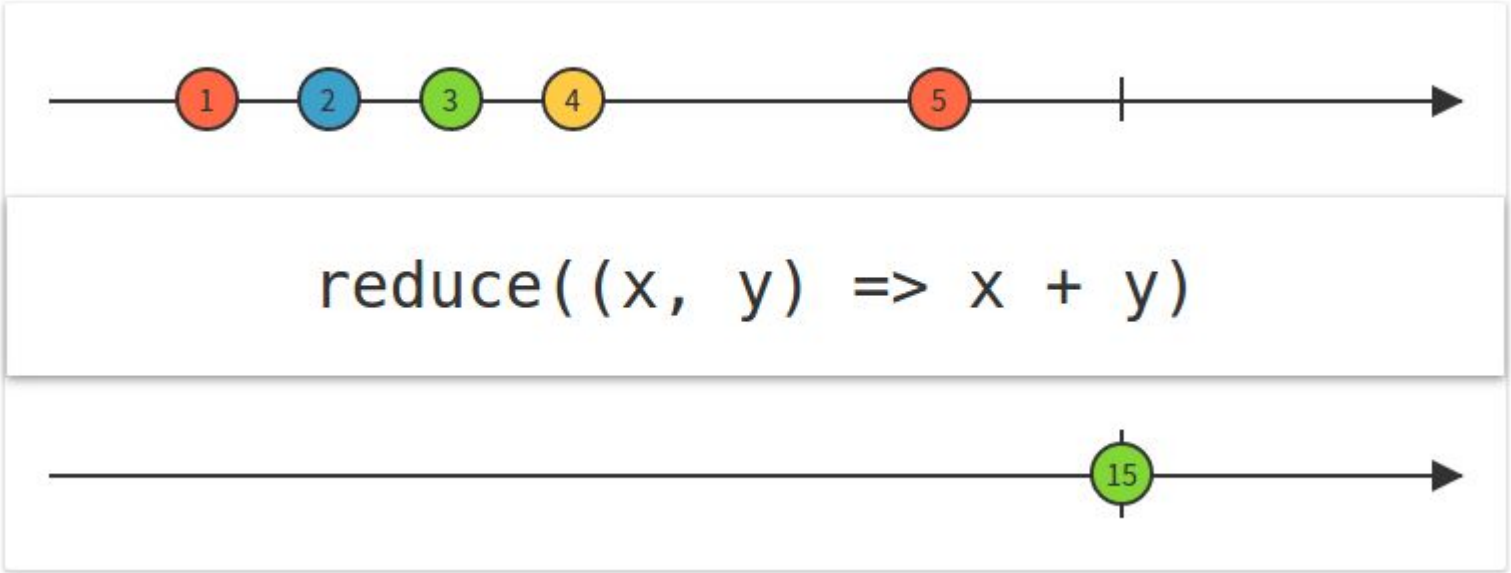
amb

given two or more source Observables, emit all of the items from only the first of these Observables to emit an item or notification



reduce

apply a function to each item emitted by an Observable, sequentially, and emit the final value



The Rx state of mind

1. Design around data flows
    a. Identify event and data streams
    b. The smaller the data size the better, in contrast with traditional threaded apps where data are transferred in large objects and threads are mostly blocked waiting.
2. Implement the Observables
    a. Use available libraries
    b. Redesign your APIs to return and use Observables instead of futures, iterators etc
3. Compose the Observables
    a. The actual computations should be done in the observers and must be non blocking
    b. State should be expressed with function composition
    c. Handle errors in the pipeline. Exception propagation does not help with multiple stacks
4. Return the result as an Observable
    a. Promotes uniformity and composition

An Rx web crawler

Scrap sites to get their titles, rss feeds and number of urls

/index.html

```
<title>Site</title>
<link rel="alternate" type="application/rss+xml" href=""/>
```

robots.txt

```
Sitemap: http://www.site.com/sitemap1.xml.gz
Sitemap: http://www.site.com/sitemap2.xml.gz
```

sitemap.xml

```
<urlset><url><loc>
<sitemapindex><sitemap><loc> // nesting
```

An Rx web crawler

```
class Site
  String host;
  String title;
  String rssFeed;

class Sitemap
  String url;
  List<String> sitemaps;
  int urlCount;

class Fetchers
  Site siteFromHost(String host, BlockingHttpClient cli) // Square OkHttpClient
  List<String> sitemapsUrlsFromRobots(String host, BlockingHttpClient cli)
  Sitemap sitemapFromUrl(String url, BlockingHttpClient cli)
```

An Rx web crawler

Implement

`Observable<Site>`

`Observable<SiteMap>`

For counting `Observable<Url>` would also suffice. The composition is different

- `map(sm -> sm.urlCount).sum()` vs `count()`

The implementation is straightforward and uses an ExecutorCompletionService

An Rx web crawler

```
if (submissions.get() == 0) { emit.onComplete(); return }
Future<Sitemap> taskf = queue.take(); // blocking
submissions.decrementAndGet();
try {
  Sitemap sm = taskf.get();
  if (sm.isIndex()) {
    for (String url : sm.getSitemaps()) {
      queue.submit(new Callable<Sitemap>() {
        public Sitemap call() throws Exception {
          return Fetchers.sitemapFromUrl(url, cli);
        }});
      submissions.incrementAndGet(); } }
      emit.onNext(sm);
} catch (Exception e) { emit.onError(e); }
```

An Rx web crawler

```
Observable<Sitemap> sitemapsOb = Observable.generate(
  new SitemapEmitter(host)).observeOn(Schedulers.io());

Observable<Site> siteOb = Observable.create(new ObservableOnSubscribe<Site>() {
  public void subscribe(ObservableEmitter<Site> emit) throws Exception {
    emit.onNext(Fetchers.siteFromHost(host, new OkHttpClient.Builder().build()));
    emit.onComplete();
  }});

return Observable.combineLatest(siteOb,
    sitemapsOb.map(sm -> sm.getUrlCount()).scan((acc, t) -> acc + t),
    new BiFunction<Site, Integer, Object[]>() {
      public Object[] apply(Site arg0, Integer arg1) throws Exception {
        return new Object[]{arg0, arg1};
      }});
```

Reactive adoption

- Spring
  - Reactor is a 4g Reactive library for building non-blocking applications. Building block for Rx
  - WebFlux is based on Reactor and provides HTTP client and server (req servlet 3.1)
- RxNetty
  - Event driven IO
- Vert.x
  - Event driven IO and non blocking toolkit. Supports reactive extensions and multiple languages
- ReactiveX
  - Implementations for java, scala, js, .net, clojure
- Github
  - ~9000 search results for “reactive”. Most starred are for android and js
  - For redis, es, jdbc, grpc and other services very few official or very popular

References

- reactivex.io the official API site with documentation
- RxJava the java flavor of Rx on github
- Reactive Programming in Netflix API a blog post by netflix
- The speakerdeck of Ben Christensen with many presentation on Rx
- A 15min intro video by Erik Meijer the creator of Rx
- What does it mean to be reactive video by Erik Meijer, for CS purists

Bonus

- Elegance and Power not Rx specific but has great references on asynchronous streams, processes and composition

Questions

```
Obversable.<Question>fromAudience().subscribe(q->speaker.answer())
```