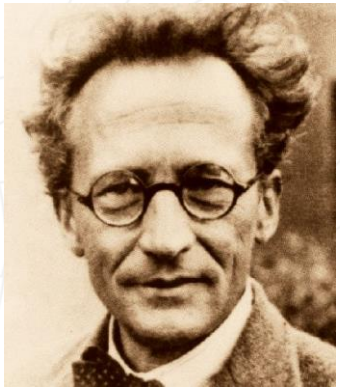
REACTIVE JAVA

TOMASZ KOWALCZEWSKI

RX JAVA BY NETFLIX

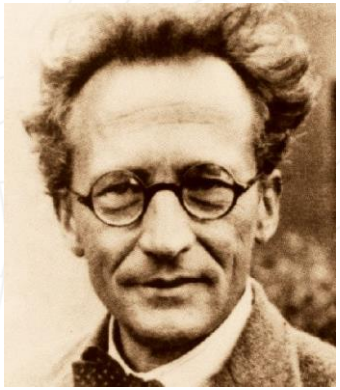
- Open source project with Apache License.
- Java implementation of Rx Observables from Microsoft
- The Netflix API uses it to make the entire service layer asynchronous
- Provides a DSL for creating computation flows out of asynchronous sources using collection of operators for filtering, selecting, transforming and combining that flows in a lazy manner
- These flows are called Observables – collection of events with push semantics (as oposed to pull in Iterator)
- Targets the JVM not a language. Currently supports Java, Groovy, Clojure, and Scala

geecon



codewise

geecon

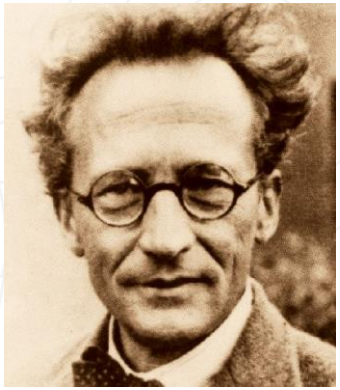


OBSERVABLE ->

codewise

geecon

OBSERVER ->



OBSERVABLE ->

codewise

SERVICE RETURNING OBSERVABLE

```
public interface ShrödingersCat {
   boolean alive();
}

public interface ShrödingersCat {
   Future<Boolean> alive();
}

public interface ShrödingersCat {
   Iterator<Boolean> alive();
}
```

REACTIVE

“readily responsive to a stimulus”

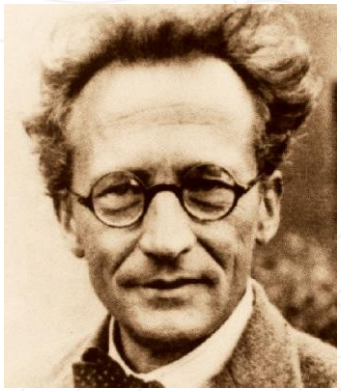
Merriam-Webster dictionary

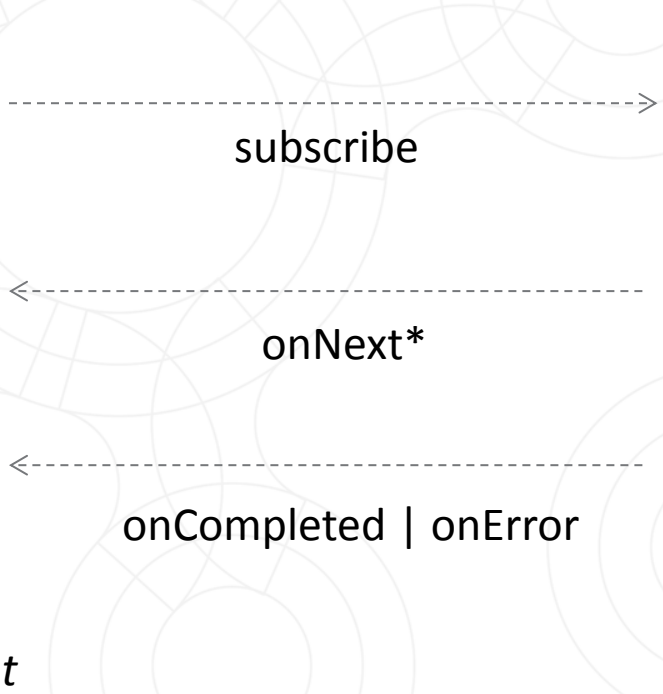
SERVICE RETURNING OBSERVABLE

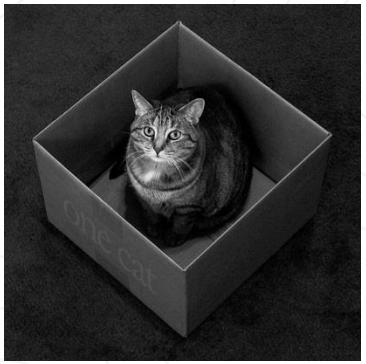
```
public interface ShrödingersCat {
   Observable<Boolean> alive();
}
```

codewise

SUBCRIPTIONS AND EVENTS







SERVICE RETURNING OBSERVABLE

```
public interface ShrödingersCat {
   Observable<Boolean> alive();
}

cat
   .alive()
   .subscribe(status -> System.out.println(status));
```

codewise

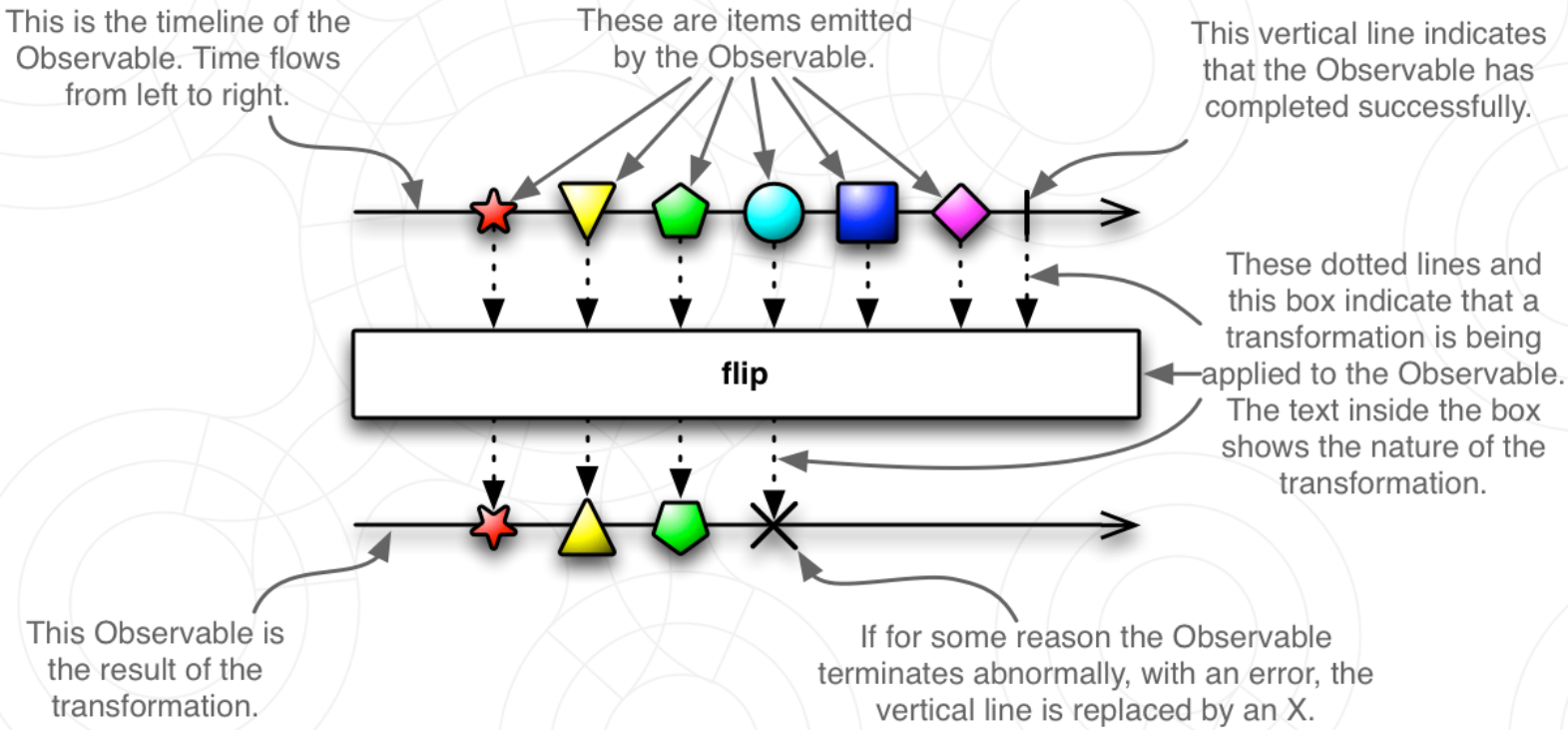
SERVICE RETURNING OBSERVABLE

```
public interface ShrödingersCat {
   Observable<Boolean> alive();
}

cat
   .alive()
   .throttleWithTimeout(250, TimeUnit.MILLISECONDS)
   .distinctUntilChanged()
   .filter(isAlive -> isAlive)
   .map(Boolean::toString)
   .subscribe(status -> display.display(status));
```

HOW IS THE OBSERVABLE IMPLEMENTED?

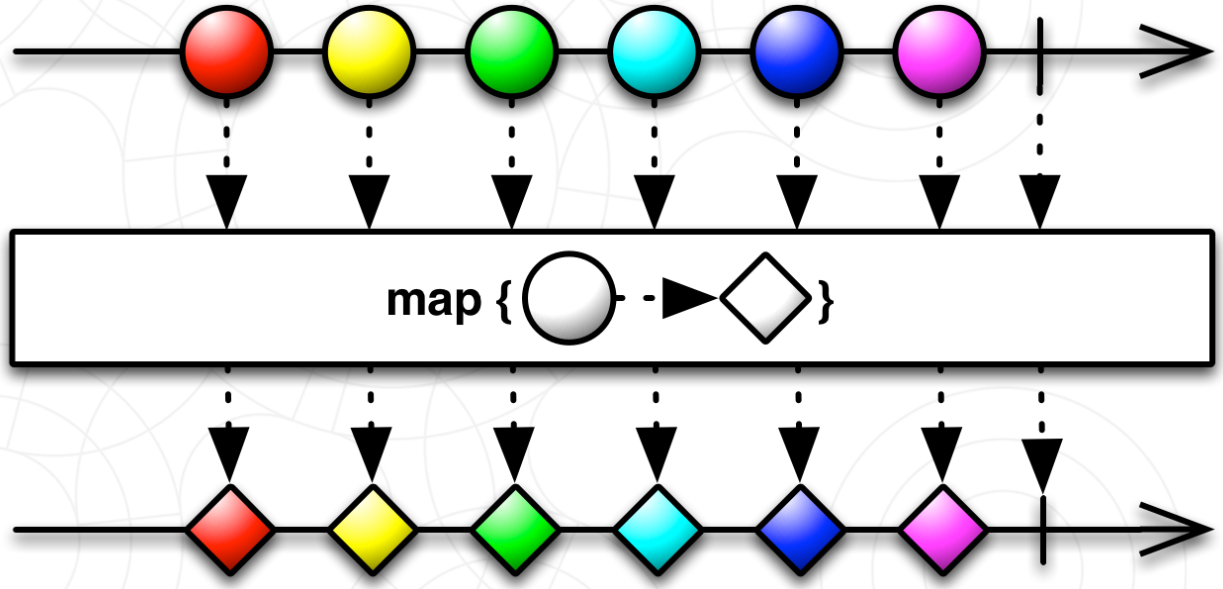
- Maybe it executes its logic on subscriber thread?
- Maybe it delegates part of the work to other threads?
- Does it use NIO?
- Maybe its an actor?
- Does it return cached data?
- Observer does not care!

MARBLE DIAGRAMS



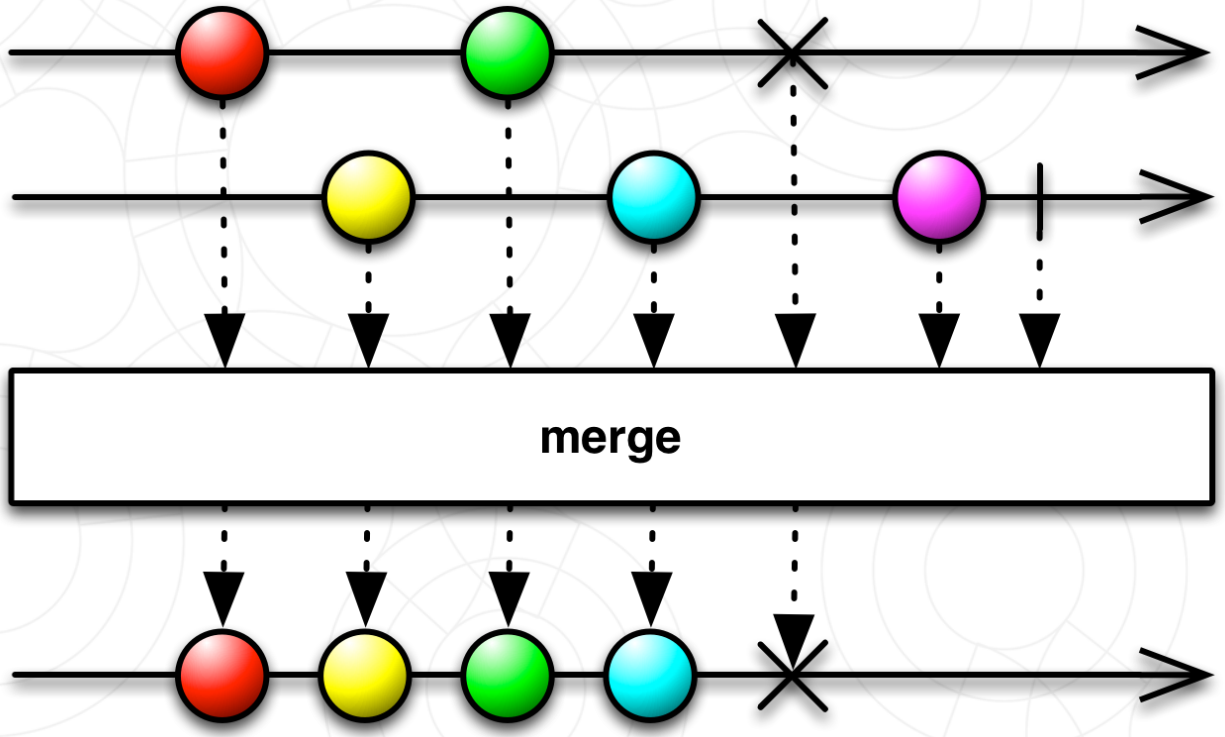
geecon

MAP(FUNC1)

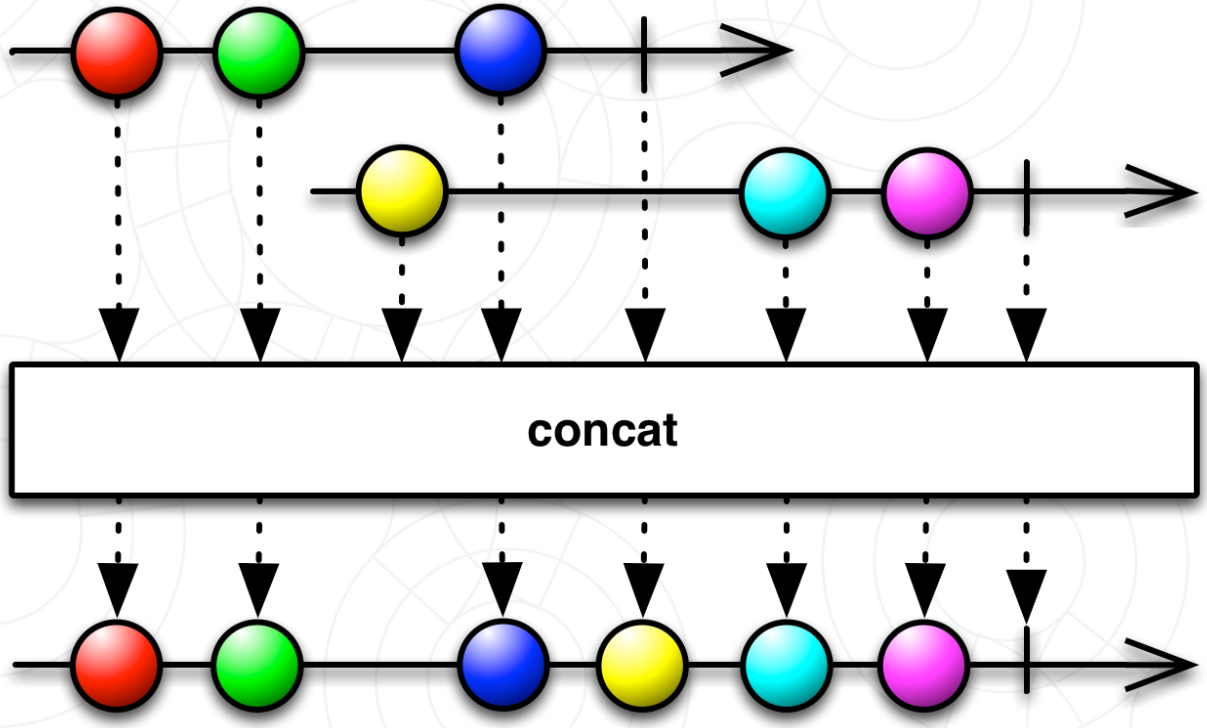


codewise

MERGE(OBSERVABLE...)

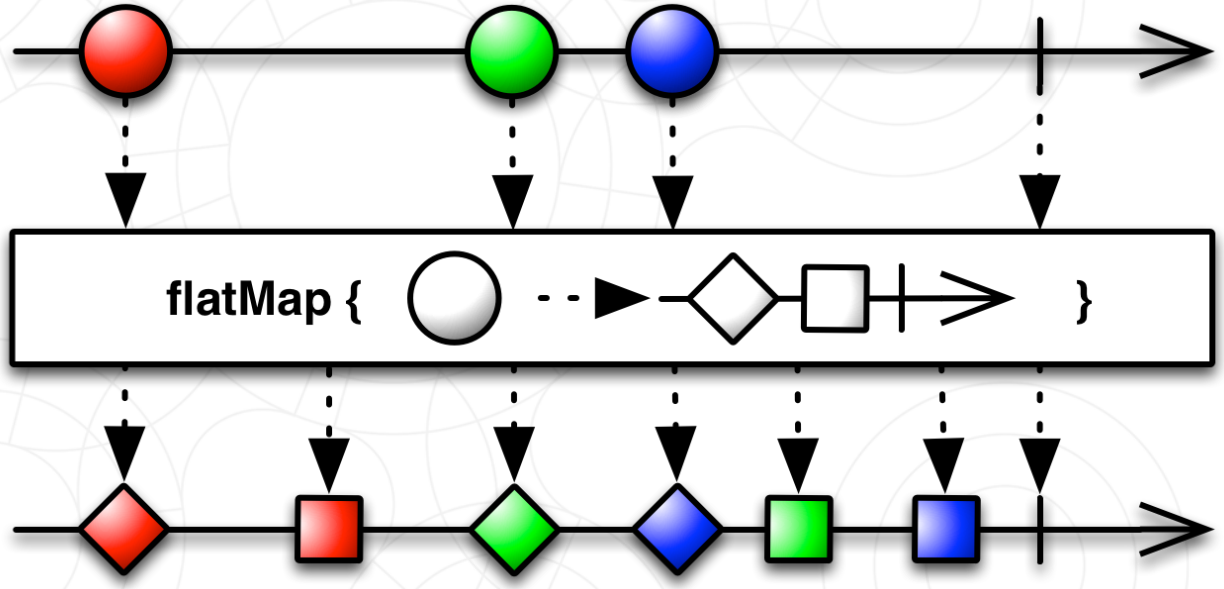


CONCAT(OBSERVABLE...)



geecon

FLATMAP(FUNC)



codewise

FLATMAP(FUNC)

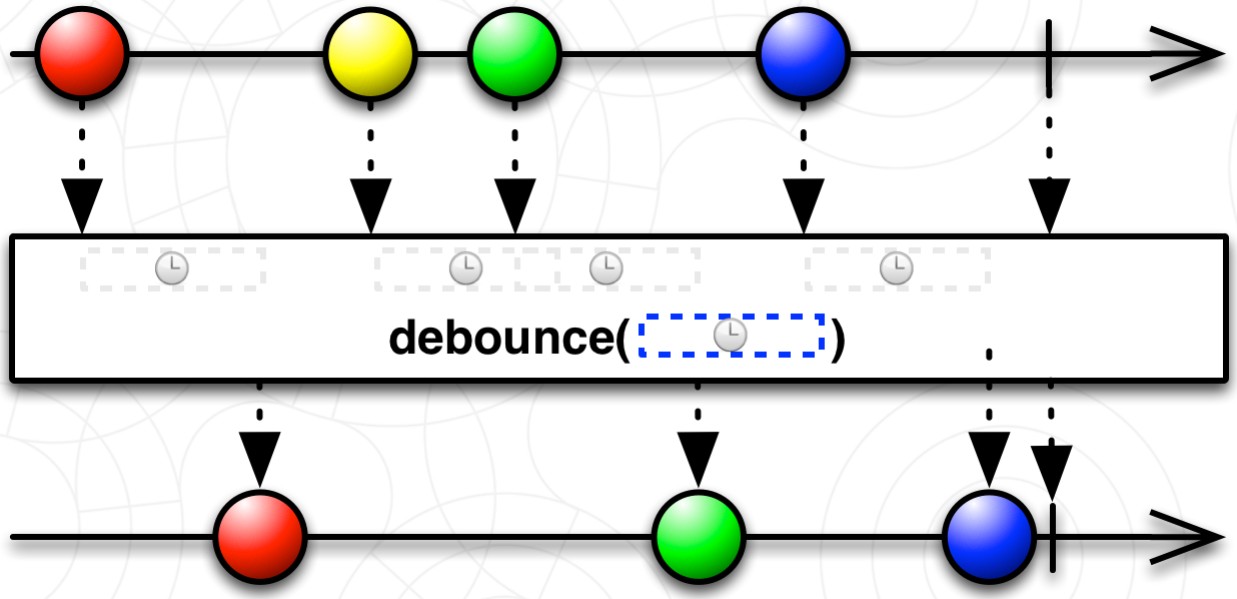
```
Observable<ShrödingersCat> cats = listAllCats();


cats
    .flatMap(cat ->
          Observable
          .from(catService.getPicturesFor(cat))
          .filter(image -> image.size() < 100 * 1000)
    )
).subscrbie();
```

CACHE

```
Random random = new Random();
Observable<Integer> observable = Observable
                                    .range(1, 100)
                                    .map(random::nextInt)
                                    .cache();

observable.subscribe(System.out::println);
observable.subscribe(System.out::println);
...
```

- Always prints same values

DEBOUNCE(LONG TIMEOUT, TIMEUNIT TU)

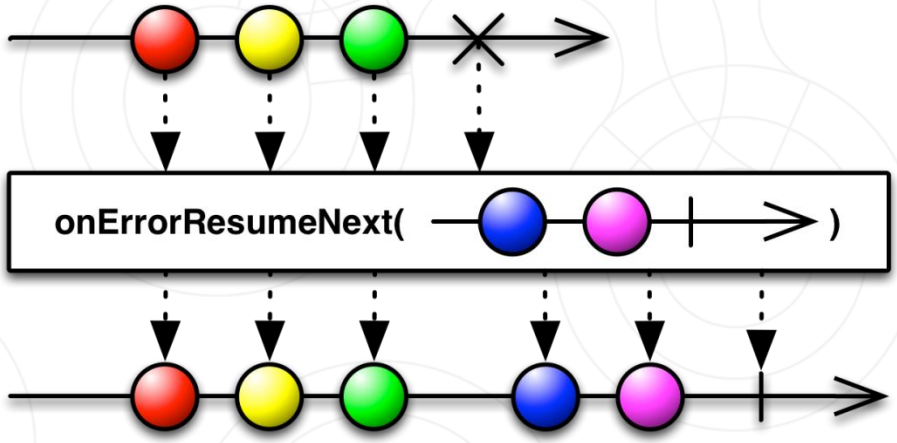


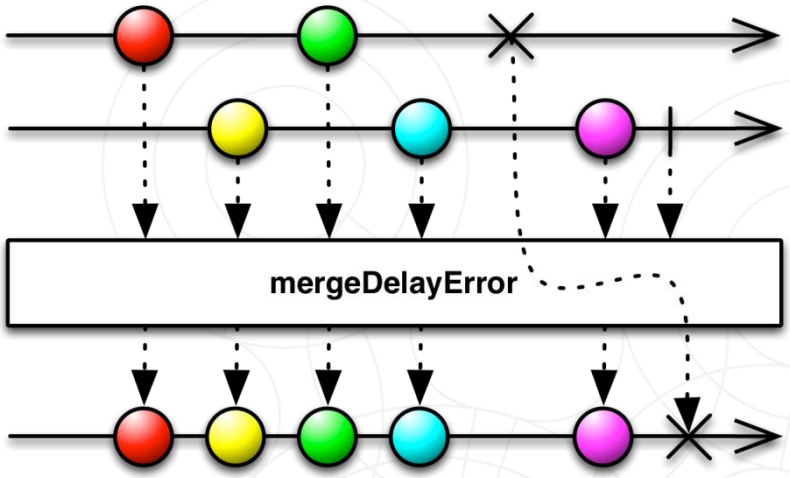
INJECTING CUSTOM OPERATORS USING LIFT

```
class InternStrings implements Observable.Operator<String, String> {
    public Subscriber<String> call(Subscriber<String> subscriber) {
        return new Subscriber<String>() {
            public void onCompleted() { subscriber.onCompleted(); }
            public void onError(Throwable e) { subscriber.onError(e); }

            public void onNext(String s) {
                subscriber.onNext(s.intern());
            }
        };
    }
}

Observable
    .from("AB", "CD", "AB", "DE")
    .lift(new InternStrings())
    .subscribe();
```

ERROR HANDLING

- Correctly implemented observable will not produce any events after error notification
  - Operators available for fixing observables not adhering to this rule
- Pass custom error handling function to *subscribe*
- Transparently substite failing observable with another one
- Convert error into regular event
- Retry subscription in hope this time it will work...





ESCAPING THE MONAD

```
Iterable<String> strings = Observable.from(1, 2, 3, 4)
        .map(i -> Integer.toString(i))
        .toBlockingObservable()
        .toIterable();
// or (and many more)
T firstOrDefault(T defaultValue, Func1 predicate)
Iterator<T> getIterator()
Iterable<T> next()
```

- Inverses the dependency, will wait for next item, then execute
- Usually to interact with other, synchronous APIs
- While migrating to reactive approach in small increments
- To trigger early evaluation while debugging

OBSERVER

```
public interface Observer<T> {
   void onCompleted();
   void onError(Throwable e);
   void onNext(T args);
}
```

CREATING OBSERVABLES

```
Observable<Boolean> watchTheCat =
   Observable.create(observer -> {
        observer.onNext(cat.isAlive());
        observer.onCompleted();
   });
```

- *create* accepts *OnSubscribe* function
- Executed for every subscriber upon subscription
- This example is not asynchronous

CREATING OBSERVABLES

```
Observable.create(observer -> {
   Future<?> brighterFuture = executorService.submit(() -> {
        observer.onNext(cat.isAlive());
        observer.onCompleted();
   });
   subscriber.add(Subscriptions.from(brighterFuture));
});
```

- Executes code in separate thread (from thread pool *executorService*)
- Stream of events is delivered by the executor thread
- Thread calling *onNext()* runs all the operations defined on observable
- Future is cancelled if client unsubscribes

CREATING OBSERVABLES

```
Observable<Boolean> watchTheCat =
   Observable.create(observer -> {
        observer.onNext(cat.isAlive());
        observer.onCompleted();
   })
   .subscribeOn(scheduler);
```

- Subscribe function is executed on supplied scheduler (thin wrapper over *java.util.concurrent.Executor*)

SUBSCRIPTION

```
public interface Subscription {
    void unsubscribe();
    boolean isUnsubscribed();
}
```

UNSUBSCRIBING

```
Observable.create(subscriber -> {
   for (long i = 0; !subscriber.isUnsubscribed(); i++) {
        subscriber.onNext(i);
        System.out.println("Emitted: " + i);
   }
   subscriber.onCompleted();
})
.take(10)
.subscribe(aLong -> {
   System.out.println("Got: " + aLong);
});
```

- *Take* operator unsubscribes from observable after 10 iterations

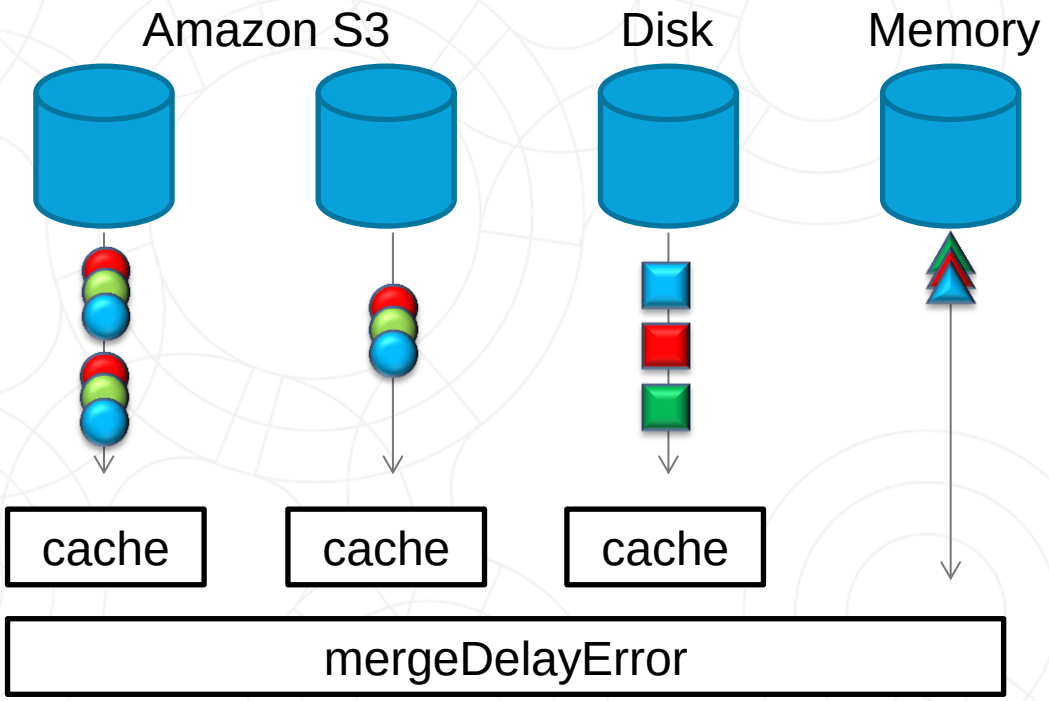
CONCURRENCY

- Synchronous vs. asynchonous, single or multiple threaded is implementation detail of service provider (Observable)
  - As long as onNext calls are not executed concurrently
  - So the framework does not have to synchronize everything
  - Operators combining many Observables ensure serialized access
- In face of misbehaving observable serialize() operator forces correct behaviour
- Passing pure functions to Rx operators is always the best bet

LESSONS LEARNED

- In our use cases performance profile is dominated by other system components
- Performance depends on implementation of used operators and may vary
- Contention points on operators that merge streams
- Some operators require scheduler, default is NewThreadScheduler
  - Creaing 1000s of threads and reaching `ulimit –u` - system almost freezes :)
- Debugging and reasoning about subscriptions is not always easy.
- Insert doOnEach or doOnNext calls for debugging
- IDE support not satisfactory, problems in placing breakpoints inside closures – IntelliJ IDEA 13 has smart step into closures which my help

HOW WE USE IT – DATA FILE DISCOVERY



MORE INFORMATION

- https://github.com/Netflix/RxJava
- https://github.com/Netflix/RxJava/wiki
- http://www.infoq.com/author/Erik-Meijer
- React conference http://www.youtube.com/playlist?list=PLSD48HvrE7-Z1stQ1vIlBumB0wK0s8llY
- Cat picture taken from http://www.teckler.com/en/Rapunzel