Rxing like a Ninja

What is Reactive Programming?

What is Rx?

Rx Building Blocks

The build blocks for Rx

Emitters	Operators	Listeners
Observable - Single - Maybe - etc	Filter - Map - Count - Replay - etc	Observers - Consumers - etc

Simple example

- Emitters

  Observable.just(takes from 1 to 10 items)

  Observable.just(1,2,3,4,5)

- Operators

  filter(Predicate<Same as ObservableType>)

  .filter(integer -> integer%2 == 0)

Cont. Simple example

- Listeners

  Observer<Same as ObservableType>

```
Observer<Integer> integerObserver = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable disposable) {

    }

    @Override
    public void onNext(Integer integer) {

    }

    @Override
    public void onError(Throwable throwable) {

    }

    @Override
    public void onComplete() {

    }
};
```

Wrap up

```
Observable.just(1,2,3,4,5)
        .filter(integer -> integer%2 == 0)
```

```
.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable disposable) {
        System.out.println("On Subscribe");
    }

    @Override
    public void onNext(Integer integer) {
        System.out.println(integer);
    }

    @Override
    public void onError(Throwable throwable) {
        System.out.println(throwable.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("On Complete");
    }
});
```

Result is :-

On Subscribe

2

4

On Complete

A Fight

```
Observable.just(1,2,(3/0),4,5)
        .filter(integer -> integer%2 == 0)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable disposable) {
                System.out.println("On Subscribe");
            }

            @Override
            public void onNext(Integer integer) {
                System.out.println(integer);
            }

            @Override
            public void onError(Throwable throwable) {
                System.out.println(throwable.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("On Complete");
            }
        });
```

A Fight

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at examples.Observables.main(Observables.java:385)
```

Observable Types

Observable types

Cold

Hot

Connectable

Cold Observables

- Wait for a subscription to start emitting
- Replay emissions to each Observer to ensure that all Observers got all the data
- Most data-driven Observables are cold

Cold Observables

```
Observable<String> source = Observable.just("One", "Two", "Three", "Four");
source.subscribe(s -> System.out.println("Observer 1 Received: " + s));
source.subscribe(s -> System.out.println("Observer 2 Received: " + s));
```

```
Observer 1 Received: One
Observer 1 Received: Two
Observer 1 Received: Three
Observer 1 Received: Four
Observer 2 Received: One
Observer 2 Received: Two
Observer 2 Received: Three
Observer 2 Received: Four
```

Hot Observables

- If an Observer subscribes to a hot Observable, receives some emissions, and then another Observer comes in afterwards, that second Observer will miss those emissions.
- Represents events rather than finite datasets. The events can carry data with them, but there is a time-sensitive component where late observers can miss previously emitted data.

Connectable Observables

- A helpful form of hot Observable is ConnectableObservable. It takes any Observable, even the cold ones, and make it hot so that all emissions are played to all Observers at once.
- To do this conversion, you simply need to call publish() on any Observable, and it will yield a ConnectableObservable.

Cont. Connectable Observables

- Subscribing will not start the emissions yet. You need to call its connect() method to start firing the emissions.

```
ConnectableObservable<Integer> integerConnectableObservable = Observable.just(1,2,3,4,5).publish();
integerConnectableObservable.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable disposable) {}
    @Override
    public void onNext(Integer integer) {
        System.out.println(integer);
    }
    @Override
    public void onError(Throwable throwable) {}
    @Override
    public void onComplete() {}
});
integerConnectableObservable.connect();
```

Cont. Connectable Observables

```
Observable<Integer> threeIntegers = Observable.range( start: 1,
        count: 3);
threeIntegers.subscribe(i -> System.out.println("Observer One: " + i));
threeIntegers.subscribe(i -> System.out.println("Observer Two: " + i));
```

```
Observer One: 1
Observer One: 2
Observer One: 3
Observer Two: 1
Observer Two: 2
Observer Two: 3
```

Cont. Connectable Observables

```
ConnectableObservable<Integer> threeIntegers = Observable.range( start: 1,  count: 3).publish();
threeIntegers.subscribe(i -> System.out.println("Observer One: " + i));
threeIntegers.subscribe(i -> System.out.println("Observer Two: " + i));
threeIntegers.connect();
```

```
Observer One: 1
Observer Two: 1
Observer One: 2
Observer Two: 2
Observer One: 3
Observer Two: 3
```

Cont. Connectable Observables

```
Observable<Integer> threeRandoms = Observable.range( start: 1, count: 3)
        .map(i -> (int) (Math.random() * 10000));

threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
```

```
Observer 1: 7795
Observer 1: 5656
Observer 1: 5924
Observer 2: 3621
Observer 2: 5002
Observer 2: 8030
```

Cont. Connectable Observables

```
ConnectableObservable<Integer> threeInts = Observable.range( start: 1, count: 3).publish();
Observable<Integer> threeRandoms = threeInts.map(i -> (int) (Math.random() * 10000));;
threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
threeInts.connect();
```

```
Observer 1: 7795
Observer 1: 5656
Observer 1: 5924
Observer 2: 3621
Observer 2: 5002
Observer 2: 8030
```

Cont. Connectable Observables

```
ConnectableObservable<Integer> threeRandoms = Observable.range( start: 1, count: 3)
        .map(i -> (int)(Math.random()*10000)).publish();
threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
threeRandoms.connect();
```

```
Observer 1: 1565
Observer 2: 1565
Observer 1: 7532
Observer 2: 7532
Observer 1: 8086
Observer 2: 8086
```

Observable Sources

Observable sources

- Observable.just(from 1 to 10 items)
  The one we used before

- Observable.fromIterable(Iterable item)
  You can pass a list or any iterable item here. It will iterate on each item and emit it.

Cont. Observable sources

- Observable.interval(int period, int timeUnit)
  it will take the specified period to emit numbers starting from zero.

  it’s good to know that interval works with timers and timers requires to be in a separate thread which is computational thread by default.

Cont. Observable sources

- Observable.fromCallable(Callable<T>)
  If you need to perform a calculation or action and then emit it, you can use Observable.just(). But sometimes, we want to do this in a lazy or deferred manner.

  Also, if that procedure throws an error, we want it to be emitted up the Observable chain through onError() rather than throw the error at that location in traditional Java fashion.

Cont. Observable sources

```
Callable<Integer> integerCallable = () -> 1 / 0;
Observable.fromCallable(integerCallable)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable disposable) {
                System.out.println("On Subscribe");
            }

            @Override
            public void onNext(Integer integer) {
                System.out.println(integer);
            }

            @Override
            public void onError(Throwable throwable) {
                System.out.println(throwable.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("On Complete");
            }
        });
```

Result is :-

On Subscribe

/ by zero

A Fight

```
Observable intervalObservable = Observable.interval( period: 3, TimeUnit.SECONDS);
intervalObservable.subscribe(new Observer<Long>() {
    @Override
    public void onSubscribe(Disposable disposable) { System.out.println("On Subscribe"); }

    @Override
    public void onNext(Long aLong) { System.out.println("First Observer :" + aLong.toString()); }

    @Override
    public void onError(Throwable throwable) { System.out.println("First Observer:"+throwable.getMessage()); }

    @Override
    public void onComplete() { System.out.println("First Observer: On Complete"); }
});

try {
    sleep( millis: 5000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

intervalObservable.subscribe(new Observer<Long>() {
    public void onSubscribe(Disposable disposable) { System.out.println("On Subscribe"); }

    @Override
    public void onNext(Long aLong) { System.out.println("Second Observer :" + aLong.toString()); }

    @Override
    public void onError(Throwable throwable) { System.out.println("Second Observer:"+throwable.getMessage()); }

    @Override
    public void onComplete() { System.out.println("Second Observer: On Complete"); }
});

try {
    sleep( millis: 15000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Observable Flavors

Observable Flavors

- Single
  Emits only one item. So we can use Single.just() like that Single.just(1). But we cannot use it like that ~~Single.just(1,2,3,4,5)~~

- SingleObserver
  It has onSubscriber, onSuccess (which consolidates both onNext and onComplete as we have only one emission), and onError

Cont. Observable Flavors

- Maybe
  Is like Single except that may not emit anything.

- MaybeObserver
  Is like the ordinary observers except that onNext is called onSuccess as it may emit something or may not. It has onSubscribe onError and onCompleted like our ordinary observers

Cont. Observable Flavors

- Completable
  Is simply concerned with an action being executed, but it does not receive any emissions.

- CompletableObserver
  Logically, it does not have onNext() or onSuccess() to receive emissions, but it does have onSubscribe(), onError() and onComplete()

Cont. Observable Flavors

- When to use Observable Flavors?

  Imagine we have a user that has interests and we need to :-

  - Get user info

    Single<UserInfo>
  - Get user interests

    Maybe<UserInterests>
  - Update user info

    Completable

Cont. Observable Flavors

- User APIs

```
public interface UserAPIs {
    @GET("/v1/user")
    Single<UserInfo> getUser();

    @GET("/v1/user/{id}/interest")
    Maybe<UserInterests> getUserInterests(@Path("id") int userId);

    @PUT("/v1/user/{id}")
    Completable updateUser(@Path("id") int userId, @Body UserInfo userInfo);
}
```

Operators

Operators

Suppressing

Transforming

Reducing

Collection

Error recovery

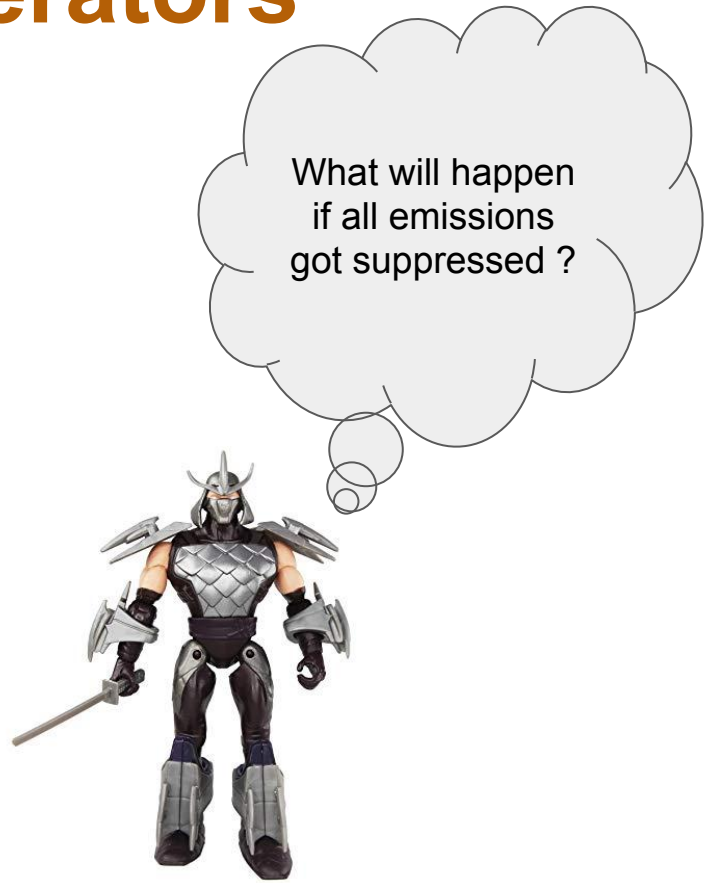
Action

Suppressing Operators

- filter()
  The filter() operator accepts Predicate<T> for a given Observable<T>. Predicate is a functional interface that accepts a lambda for the test method which filters the emitted item based upon a condition and will pass it or return.

Cont. Suppressing Operators

- filter()



Cont. Suppressing Operators

- filter()
  If all emissions fail to meet your criteria, the returned Observable will be empty, with no emissions occurring before, onComplete() is called.

Cont. Suppressing Operators

- take()

  It has two overloads:-

  The first is number of emissions you want to take. For example take(3) will take only first 3 emissions.

  The Second is specified time interval, for example take(3, TimeUnit.SECONDS) which will take emissions for 3 seconds.

Cont. Suppressing Operators

- distinct()
  will emit each unique emission if we have 122347789 it will emit 1234789 and skip duplicates.

Keep in mind that if you have a wide, diverse spectrum of unique values, distinct() can use a bit of memory.

Transforming Operators

- map()
  we use map to transform emitted data by using a lambda function. For example we may transform the emitted string to be the length of that string so we use map like this
  map(string -> string.length())

Cont. Transforming Operators

- switchIfEmpty()
  Specifies a different Observable to emit values from. If the source Observable is empty, this allows you to specify a different sequence of emissions.

```
Observable.empty()
        .switchIfEmpty(Observable.just("switched"))
        .subscribe(System.out::println);
```

Cont. Transforming Operators

- sorted()
  If you have a finite Observable<T> emitting items that implement Comparable<T> , you can use sorted() to sort the emissions. Internally, it will collect all the emissions and then re-emit them in sorted order.

If you use this against an infinite Observable , you may get an OutOfMemory error

Cont. Transforming Operators

- scan()
  Is a rolling aggregator. For every emission, you add it to an accumulation you can emit the sum of emits.

```
Observable.just(5, 3, 7, 10, 2, 14)
        .scan((accumulator, next) -> accumulator + next)
        .subscribe(s -> System.out.println("Received: " + s));
```

The Result :-
5, 8,15, 25, 27, 41

Reducing Operators

- reduce()
  Is syntactically identical to scan() , but it only emits the final accumulation when the source calls onComplete()

```
Observable.just(5, 3, 7, 10, 2, 14)
        .reduce((accumulator, next) -> accumulator + next)
        .subscribe(System.out::println);
```

The Result :-
41

Cont. Reducing Operators

- contains()
  Checks whether a specific element (based on the hashCode()/equals() implementation) ever emits from an Observable . It will return a Single<Boolean> that will emit true if it is found and false if it is not.

Collection Operators

- toList()
  For a given Observable<T> , it will collect incoming emissions into a List<T> and then push that entire List<T> as a single emission (through Single<List<T>>)

```
Observable.just(5, 3, 7, 10, 2, 14)
        .toList()
        .subscribe(System.out::println);
```

The Result :-
[5,3,7,10,2,14]

Cont. Collection Operators

- collect()
  You can always use the collect() operator to specify a different type to collect items into. For example, there is no toSet() operator to collect emissions into a Set<T>.

Cont. Collection Operators

- collect()
  You will need to specify two arguments that are built with lambda expressions:-

  InitialValueSupplier, which will provide a new HashSet for a new Observer.

  Collector, which specifies how each emission is added to that HashSet.

Cont. Collection Operators

```
Observable.just("a", "b", "c", "d")
        .collect(ArrayList::new, (list, value) -> {
    System.out.println(String.format("Inserting '%s' into ArrayList", value));
    list.add(value); })
        .subscribe(System.out::println);
```

```
Observable.just("a", "b", "c", "d")
        .collect(HashSet::new, HashSet::add)
        .subscribe(System.out::println);
```

Error Recovery Operators

- onErrorReturn() and onErrorReturnItem()
  Which replaces onError with a single onNext(value) followed by onCompleted().

```
Observable.just(100)
        .map(integer -> (integer / 0))
        .onErrorReturn(throwable -> {
            if (throwable instanceof ArithmeticException)
                return 100;
            else
                return 0;
        })
        .subscribe(System.out::println);
```

Cont. Error Recovery Operators

- onErrorResumeNext()
  accepts another Observable as a parameter to emit potentially multiple values, not a single value, in the event of an exception.

  We can also pass Observable.empty() to quietly stop emissions in the event where there is an error and gracefully call the onComplete() function

Cont. Error Recovery Operators

- onErrorResumeNext()

```
userAPIs.getUser()
        .onErrorResumeNext(userDAO.getUserInfo())
        .subscribe(System.out::println);
```

Action Operators

- doOn*()
  * refers to Next, Subscribe, Complete, etc

  To perform and action in a specific event

Cont. Action Operators

```
userAPIs.getUser()
        .doOnSuccess(userInfo -> userDAO.insertUserInfo(userInfo))
        .onErrorResumeNext(userDAO.getUserInfo())
        .subscribe(System.out::println);
```

Concurrency
&
Parallelization

Concurrency & Parallelization

RxJava supports multi-threading through SubscribeOn() and ObserveOn() using various schedulers.

Note that merging operators like merge(), zip(), etc can merge between two threads.

Cont. Concurrency & Parallelization

- Schedulers different types
  - Computational

    This will maintain a fixed number of threads based on the processor count available to your Java session, making it appropriate for computational tasks such as math, algorithms, and complex logic.

Cont. Concurrency & Parallelization

- Computational
  When you are unsure how many tasks will be executed concurrently or are simply unsure which Scheduler is the right one to use, prefer the computation one by default.

Cont. Concurrency & Parallelization

- IO
  Tasks such as reading and writing databases, web requests, and disk storage are less expensive on the CPU and often have idle time waiting for the data to be sent or come back.

  It will maintain as many threads as there are tasks and will dynamically grow, cache, and reduce the number of threads as needed.

Cont. Concurrency & Parallelization

- New thread
  It will create a new thread for each Observer and then destroy the thread when it is done.

  It does not attempt to persist and cache threads for reuse.

- Single
  When you want to run tasks sequentially on a single thread.

Cont. Concurrency & Parallelization

- subscribeOn()
  It works all the way upstream so any operation gonna be done before calling subscribeOn() will be dones in the specified thread.

  you should always call subscribeOn() near the source of observables.

  if you called many subscibeOn() methods with different schedulers it will use the nearest one to the source.

Cont. Concurrency & Parallelization

- observeOn()
  The observeOn() operator, will intercept emissions at that point in the Observable chain and switch them to a different Scheduler going forward.

  Unlike subscribeOn() , the placement of observeOn() matters. It will leave all operations upstream on the default or subscribeOn() defined Scheduler, but will switch to a different Scheduler downstream.

Cont. Concurrency & Parallelization

- observeOn()
  You can actually use multiple observeOn() operators to switch Schedulers more than once.

  This is helpful in case of you have many operations to be done on the stream and each of them has an appropriate Scheduler to be done on. So you are free to switch the used Scheduler by using observeOn()

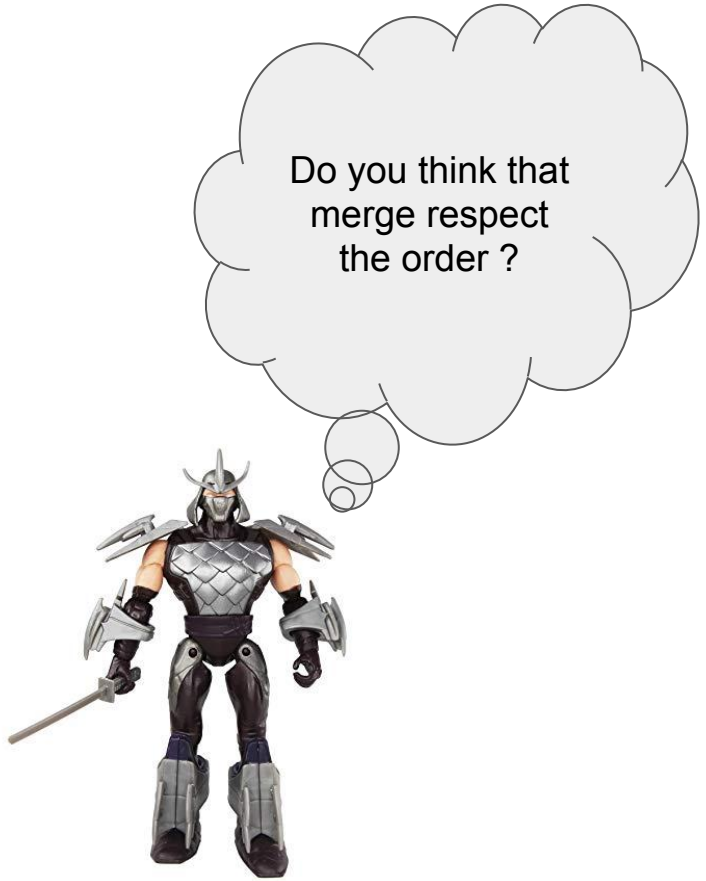
Combining

Combining

- Merging
  A common task done in ReactiveX takes two or more Observable<T> instances and merges them into one Observable<T> .

- Observable.merge() and mergeWith()
  Merge or merge with can merge from two up to four observables, the merged observable subscribe to all of them simultaneously.

Cont. Combining



Cont. Combining

```
Observable firstObservable =
        Observable.interval( period: 1, TimeUnit.SECONDS).map(aLong -> "first : " + aLong)
                .take(4);
Observable secondObservable =
        Observable.interval( period: 2, TimeUnit.SECONDS).map(aLong -> "second : " + aLong)
                .take(2);
Observable.merge(firstObservable, secondObservable).subscribe(System.out::println);
sleep( millis: 8000);
```

```
first : 0
first : 1
second : 0
first : 2
first : 3
second : 1
```

Cont. Combining

- Concatenation

Is the same as merging but it emits observables in the specified order. so it will not move to the next observable until the first one calls onComplete().

Remember that this is a poor choice if you target an infinite observables as it will emit infinitely and won’t call onComplete and hold up the queue.

Cont. Combining

```
Observable firstObservable =
        Observable.interval( period: 1, TimeUnit.SECONDS).map(aLong -> "first : " + aLong)
                .take(4);
Observable secondObservable =
        Observable.interval( period: 2, TimeUnit.SECONDS).map(aLong -> "second : " + aLong)
                .take(2);
Observable.concat(firstObservable, secondObservable).subscribe(System.out::println);
sleep( millis: 8000);
```

```
first : 0
first : 1
first : 2
first : 3
second : 0
second : 1
```

Cont. Combining

- flatMap()
  It is an operator that performs a dynamic Observable.merge() by taking each emission and mapping it to an Observable . Then, it merges the emissions from the resulting Observables into a single stream.

Cont. Combining

```
Observable.just("One", "Two", "Three")
        .flatMap(s -> Observable.fromArray(s.split( regex: "")))
        .subscribe(System.out::println);
```

The Result :-

O
n
e
T
etc

Cont. Combining

- Observable.combineLatest()
  For every emission that fires from one of the sources, it will immediately couple up with the latest emission from every other source.

  CombineLatest emits when all of the observables start emitting items.

  It combines from 2 up to 9 Observables.

Cont. Combining

```
Observable first =
        Observable.interval( period: 300, TimeUnit.MILLISECONDS);
Observable second =
        Observable.interval( period: 1, TimeUnit.SECONDS);
Observable.combineLatest(first, second, (01, 02) -> "First: " + 01 + " Second: " + 02)
        .subscribe(System.out::println);
sleep( millis: 3000);
```

The Result :-

First: 2 Second: 0
First: 3 Second: 0
First: 4 Second: 0
First: 5 Second: 0
First: 5 Second: 1

What is the first emission of this ?

Subjects

Subjects

They are both an Observer and Observable

There are a couple implementations of Subject , which is an abstract type that implements both Observable and Observer

This means that you can manually call onNext(), onComplete(), and onError() on a Subject , and it will in turn pass those events downstream toward their Observers.

Cont. Subjects

- When to use subjects
  to subscribe to unknown number of observables so you consolidate them to a single observable since subject are an observer too, so you can pass it in the subscribe method
- When you shouldn’t use Subjects
  Subjects are not disposable so don’t use them when you are not sure 100% that there is something that gonna be emitted without your control.

Cont. Subjects

- Common Subjects
  - Publish Subject

The simplest Subject type is the PublishSubject

```
Subject<String> subject = PublishSubject.create();
subject.subscribe(System.out::println);
subject.onNext( t: "Begin");
Observable.interval( period: 1, TimeUnit.SECONDS).map(aLong -> "One : "+aLong).subscribe(subject);
Observable.interval( period: 2, TimeUnit.SECONDS).map(aLong -> "Two : "+aLong).subscribe(subject);
```

Cont. Subjects

```
Subject<String> subject = PublishSubject.create();
subject.subscribe(System.out::println);
subject.onNext( t: "Begin");
Observable.interval( period: 1, TimeUnit.SECONDS).map(aLong -> "One : "+aLong).subscribe(subject);
Observable.interval( period: 2, TimeUnit.SECONDS).map(aLong -> "Two : "+aLong).subscribe(subject);
```

Cont. Subjects

- Behavior Subject

  As PublishSubject , but it will replay the last emitted item to each new Observer downstream.

```
BehaviorSubject<Integer> behaviorSubject =
        BehaviorSubject.create();
behaviorSubject.subscribe(integer ->
        System.out.println("First Observer: " + integer));
behaviorSubject.onNext( t: 1);
behaviorSubject.onNext( t: 2);
behaviorSubject.onNext( t: 3);
behaviorSubject.subscribe(integer ->
        System.out.println("Second Observer: " + integer));
behaviorSubject.onNext( t: 4);
behaviorSubject.onComplete();
```

```
First Observer: 1
First Observer: 2
First Observer: 3
Second Observer: 3
First Observer: 4
Second Observer: 4
```

Cont. Subjects

Do you think subjects are hot or cold observables?

Cont. Subjects

```
PublishSubject<Integer> integerPublishSubject = PublishSubject.create();
integerPublishSubject.onNext( t: 1);
integerPublishSubject.subscribe(System.out::println);
integerPublishSubject.onNext( t: 2);
integerPublishSubject.onNext( t: 3);
integerPublishSubject.subscribe(System.out::println);
```

The Result :-

2

3

Cont. Subjects

```
BehaviorSubject<Integer> behaviorSubject =
        BehaviorSubject.create();
behaviorSubject.onNext( t: 1);
behaviorSubject.subscribe(integer ->
        System.out.println("First Observer: " + integer));
behaviorSubject.onNext( t: 1);
behaviorSubject.onNext( t: 2);
behaviorSubject.onNext( t: 3);
behaviorSubject.subscribe(integer ->
        System.out.println("Second Observer: " + integer));
behaviorSubject.onNext( t: 4);
behaviorSubject.onComplete();
```

The Result :-

1
1
2
3
3
4
4

Cont. Subjects

```
BehaviorSubject<Integer> behaviorSubject =
        BehaviorSubject.create();
behaviorSubject.onNext( t: 0);
behaviorSubject.onNext( t: 1);
behaviorSubject.subscribe(integer ->
        System.out.println("First Observer: " + integer));
behaviorSubject.onNext( t: 1);
behaviorSubject.onNext( t: 2);
behaviorSubject.onNext( t: 3);
behaviorSubject.subscribe(integer ->
        System.out.println("Second Observer: " + integer));
behaviorSubject.onNext( t: 4);
behaviorSubject.onComplete();
```

The Result :-

1
1
2
3
3
4
4

Thank You ^_^

Resources and recommendations

- For nerds
  Learning RxJava by Thomas Neild

- For geeks
  Learn RxJava Talk by Kaushik Gopal

- For both
  RxMarbles application