RxJava

Summary

This document is for those who want to understand RxJava and Observables, as they are used heavily in my code. This document will be a very brief overview of what RxJava is and what it's uses are. For a more indepth introduction, see this.

What is RxJava

A Publish/Subscribe library, where `Observable` is the Publisher, and `Observer` is a `Subscriber`. RxJava is a way to create code that 'reacts' to changes. RxJava can allow for more functional programming, as well as supplement object-oriented programming. RxJava also allows a 'push-based' flow of control over Java 8's Stream pull-based flow of control.

Operators

just

Create an Observable that will emit the provided items.

```
Observable.just(1, 2, 3);
```

Will emit 1, 2, and 3.

from

Create an Observable from an existing source. Variants are `fromIterable` or `fromArray`.

```
int[] arr = {1, 2, 3};
Observable.fromArray(arr);
```

Will emit 1, 2, and 3.

map

A mapping of an input to an output.

```
Observable.just(1, 2, 3).map(x -> x + 1);
```

Will emit 2, 3, and 4.

flatMap

A mapping of an input to another Observable, which then 'flattens' into a single Observable.

```
Observable.just("Hello").flatMap(x -> Observable.fromArray(x.toCharArray()));
```

Will emit 'H', 'e', 'l', 'l', 'o'.

filter

Filters out items with some predicate.

```
Observable.just(1, 2, 3).filter(x -> x % 2 == 0);
```

Will emit '2'.

buffer

Buffers the given amount of emitted items into a list. This can be used for batch processing.

```
Observable.just(1, 2, 3).buffer(2);
```

Will emit two lists, containing {1, 2} and {3}.

groupBy

Groups items by some property.

```
Observable.just(1, 2, 3).groupBy(x -> x % 2 == 0);
```

Will emit two `GroupedObservables`, (false, {1, 3}) and (true, {2}).

`GroupedObservables` can be used to process groups of items by the defined property. `GroupedObservables` are still `Observables` and so can be used interchangeably.

subscribe and blockingSubscribe

A terminating operator that consumes the result of applying all other operators

```
Observable.just(1, 2, 3).filter(x -> x % 2 == 0).subscribe(System.out::println);
```

Will only print 2 because 1 and 3 were filtered.

`blockingSubscribe` is like `subscribe` except that it forces the current computations to finish before proceeding. Because `Observables` are lazy by nature and only compute based on need, this forces that need.

ConnectedObservables

`ConnectedObservables` and is used to construct what is referred to as a 'Cold' Observable. An Observable is 'Cold' if it will only begin emitting items when something is listening. A 'Hot' Observable emits items even if no one is listening, and as such can cause them to be missed.

connect

Is used to notify that it should begin emitting items.

replay

Signifies that it should cachce and re-emit all items to a new subscriber, so all subscribers see the same data. Useful when wanting to reprocess the same data over and over like we do.

Schedulers

As all operators for `Observables` pass state from one to the next, they are inherently thread-safe so long as no side-effect inducing operations are performed in them. The two most used schedulers are the IO-Bound `io` and CPU-Bound `computation` schedulers. These can be used to pass work off between different threads.

```
Observable
  .just(1, 2, 3)
  .subscribeOn(Schedulers.io())
  .map(x -> someIOBoundTask(x))
  .observeOn(Schedulers.computation())
  .map(x -> someCPUBoundTask(x));
```

`subscribeOn` is the default scheduler work begins on. `observeOn` is how you exchange work to other schedulers.