RxJava : Introduction.

- RxJava is actually based on an old technogly going back to patterns of the 1980's. More specifically it is a reengineering of the Observer pattern, with a few additional features.
- It was documented in the GoF desigh pattern book " *Design Patterns: Elements of Reusable Object-Oriented Software" released in 1995*. (Did anybody read all of it at the time, and understand all the patterns? Who was even programming at the time?)
- The observer pattern allowed you to push data from a subject to one or many observers if data in the subject changes. Normally when programming we just pull data from somewhere. Think of pulling values, from an array, or from a file. Makes synchronizing of thread difficult perhaps, especially when also writing values?
- When you push data and the data does not change, synchronisation becomes less of a problem? Data not changing ,sounds a bit functional to me?
- Perhaps the following are examples of pushing data.
    - Perhaps we can view the UI as pushing mouse data, keyboard data?
    - Could we view asynchronous operations, as pushing data?
- Perhaps we could chain, operations easily, since we know who the data is going to be pushed to? Could we do this if we used callbacks?
- RxJava can.

RxJava: Simple Old Observable Implementation.

- If we look at diagram at https://en.wikipedia.org/wiki/Observer_pattern, we can get some idea of a simple implementation for an Observer.
- Basically the subject keeps a list of observers, and when an event happens this passed onto all the observers in the list.
- Notice it we have no way of indicating errors, we have to register/unregister to get anything to work, no mention of which thread the observers are working on. Probably just the main thread, as that will be the one the observers where probably created on.
- All in all a bit awkward to use, and looks marginally useful ☹.

RxJava: A Bit Of History.

- Finally along comes Erik Meijer at Microsoft who took a new look at the observer pattern.
- Created Observer, and Observable(Subject) interfaces.
- Added synchronization across threads, i.e. the ability to add observers to observables, even if they are both in different threads.
- Made it easy to spread Observers/Observables out into different threads if required.
- Dictated that data passed around should always be safe, because it should not change.
- Added an error channel.
- Added a channel that could indicate when the Observable had finished sending out data.
- Created a lot of useful objects that could take data, manipulate it then pass it on. Allowing us to create streams of data. Many of these functions look familiar to functional programmers.
- Microsoft open sourced their version of the C# reactive functional programming. Netflix took it up, and it converted it to Java and many other languages. The distributed nature and useful functionality fitted well with there distributed video systems, which they rewrote using functional relative programming methods.

RxJava: A Very Simple Program.

- The simplest program has to be

  ```
  Observable<String> observable = just(“hello”, “world”);
  Observer observer = observable.subscribe( str -> log.w( str ));
  ```

  Notice -> we are using lambda calculus, it makes code much shorter.
- The just emits the strings “hello”, “world” one after another in a stream.
- If we look at the marble diagram for “just”at http://reactivex.io/documentation/operators/just.html
- We can see it takes 1 or more items, and just emits the items.
- Once the items have been emitted, the Observable completes.
- Once we complete the observable completes the observer unsubscribes from the observable, which means the observable is now dead. So the observer and observable can be garbage collected. The unsubscribe is automatic on completion, and errors.

RxJava : Subscription

- If we are interested in when we complete, or is there is an error we can create code like this

```
Observable.just(“hello”, “ ”, “world”) .subscribe(new Subscriber<IString>() {
                @Override public void onNext(Integer item) { System.out.println("Next: " + item); }
                @Override public void onError(Throwable error) { System.err.println("Error: " + error.getMessage()); }
                @Override public void onCompleted() { System.out.println("Sequence complete."); } });
or
Observable.just(“hello”, “ ”, “world”) .subscribe(
                 item->System.out.println("Next: " + item); ,
                error-> System.err.println("Error: " + error.getMessage());,
                 ()->System.out.println("Sequence complete."); } });
```

- Note both programs are exactly the same. The use of lambda calculus in Java makes the code shorter and easier to understand.
- We have the options as before of intercepting the onNext or 1st parameter, which we just log to console.
- We have the options of intercepting the onComplete or “()->”. This might be a useful time to close a database in which we are saving data.
- Similarly we can intercept the onError() or “error->”.

RxJava:creating observables.

- You can always create your own observable. You might want to convert a database into a stream, then pass all this data onto a number of subscribers. Each subscriber does its own thing, this can be efficient, as you will be going through the database once.
- If we look at http://reactivex.io/documentation/operators/create.html we can see how to create a simple observable.
- Observables can be hot or cold.
  - A hot observable continuously emits events irrespective of any observer. If multiple observers subscribe at different times, they are likely to get different emitted values from the stream. An observable that emits mouse positions, would be a hot observable.
  - A cold observable just emits values when an observer subscribes to it. All observers get same emitted objects.

Rxjava : Transforming, filtering, combining, and other Observables

- RxJava provides a large number of observables which can be chained together. It would take all week to go through them all, so we will just look at a few. Many of the observable map on to concepts familiar to functional programmers.
- A transforming Observable : map
  http://reactivex.io/documentation/operators/map.html takes an item and applies a function to it returning a new item.
- A filtering Observable : filter
  http://reactivex.io/documentation/operators/filter.html
  Takes an item and applies a predicate. If the predicative is satisfied it is emitted (passed to the next observable in the stream).
- A combining Observable : CombineLatest
  http://reactivex.io/documentation/operators/combinelatest.html.
- This will take the last emitted items from an Observable, when one of the observables has just emitted an item.
- Many more go to http://reactivex.io/documentation/operators.html to see them.

RxJava:Schedulers multithreading made easy.

- RxJava supports the concept of schedulers.
- You have 5 standard schedulers, but you can also create your own.
- .io() : Used if doing io. The work is done on a thread, selected from a thread pool which can expand providing new threads. Old threads that are no longer used may simply be reused or die. As you do more io the thread pool can increase indefinitely until the system crashes. But io will not block anything.
- Schedulers.computaional() : Used for calculation. Usually has a thread pool such that one thread is on each core of a multiprocessor. Saves on swapping contexts.
- Schedulers.immediate() : Use the current thread to execute the observable. Executes immediately.
- Schedulers.newThread() : Create a new thread for each of work. Gets expensive. Threads will simply be destroyed after use.
- Schedulers.trampoline() : Puts work into a queue, for execution on this thread, when this thread is done as a piece of work.
- And for Android users we have observeOn(AndroidSchedulers.mainThread(), which makes sure any code is executed on the main thread. Important for the GUI, where any access in the non main thread will most likely cause a crash.
- Most observables will take a parameter which will dictate what scheduler to use.
- Alternatively you can tell a chain of observables, what scheduler to use. See later.
- RxJava is single threaded by default.
- Some objects have sensible schedule settings. But we can also create our own schedulers.
  - In some cases we might want to create a new thread for each bit of data, we pass around i.e. if the computation takes along time (http requests of bitmap data).
  - Other times all data might go though a single thread (inserting data into a database in a single thread, would require no more synchronisation that provided by RxJava).
  - Other times we might want to limit the number of threads. In the case of http requests of bitmaps, we might to limit the number of requesting threads to 4. An unlimited number of requests on an unlimited number of threads might not be good for the machines.

RxJava : A final example.

-
  ```
  Observable<int> observable = just(1,2,3,4,5)
  .subscribeOn(Schedulers.computaional() )
  .filter( v -> x%2==0 )
  .subscribeOn(Schedulers.io())
  .map(v->v*2)
  .observeOn(AndroidSchedulers.mainThread());
  Observer observer = observable.subscribe( v -> log.w( v ));
  ```

- Notice how everything is chained.
- The subscribeOn, and observerOn dictate what threads are used.
- Even though using threads, it is easy to see what is happening.