~RxJava-Android~

What is Reactive Programming?

:~It is all about *responding to value changes*. For example, let's say we define $x = y+z$. When we change the value of y or z, the value of x automatically changes. This can be done by observing the values of y and z.

Why doing asynchronous programming

Reactive programming provides a simple way of asynchronous programming.

1. This allows to simplify the asynchronously processing of potential long running operations.
2. It also provides a defined way of handling multiple events, errors and termination of the event stream.
3. Reactive programming provides also a simplified way of running different tasks in different threads. For example, widgets in SWT and
4. Android have to be updated from the UI thread and reactive programming provides ways to run observables and subscribers in different threads.
5. It is also possible to convert the stream before its received by the observers. And you can chain operations, e.g., if a API call depends on

the call of another API Last but not least, reactive programming reduces the need for state variables, which can be the source of errors

Reactive Extensions is a library that follows Reactive Programming principles to compose asynchronous and event-based programs by using observable sequence.

RxJava is a Java based implementation of Reactive Programming.

RxAndroid is specific to Android platform which utilises some classes on top of the RxJava library.

The building blocks of RxJava are:

1. **Observables** representing sources of data:
2. **Subscribers (or observers)** listening to the observables
3. A set of methods for modifying and composing the data

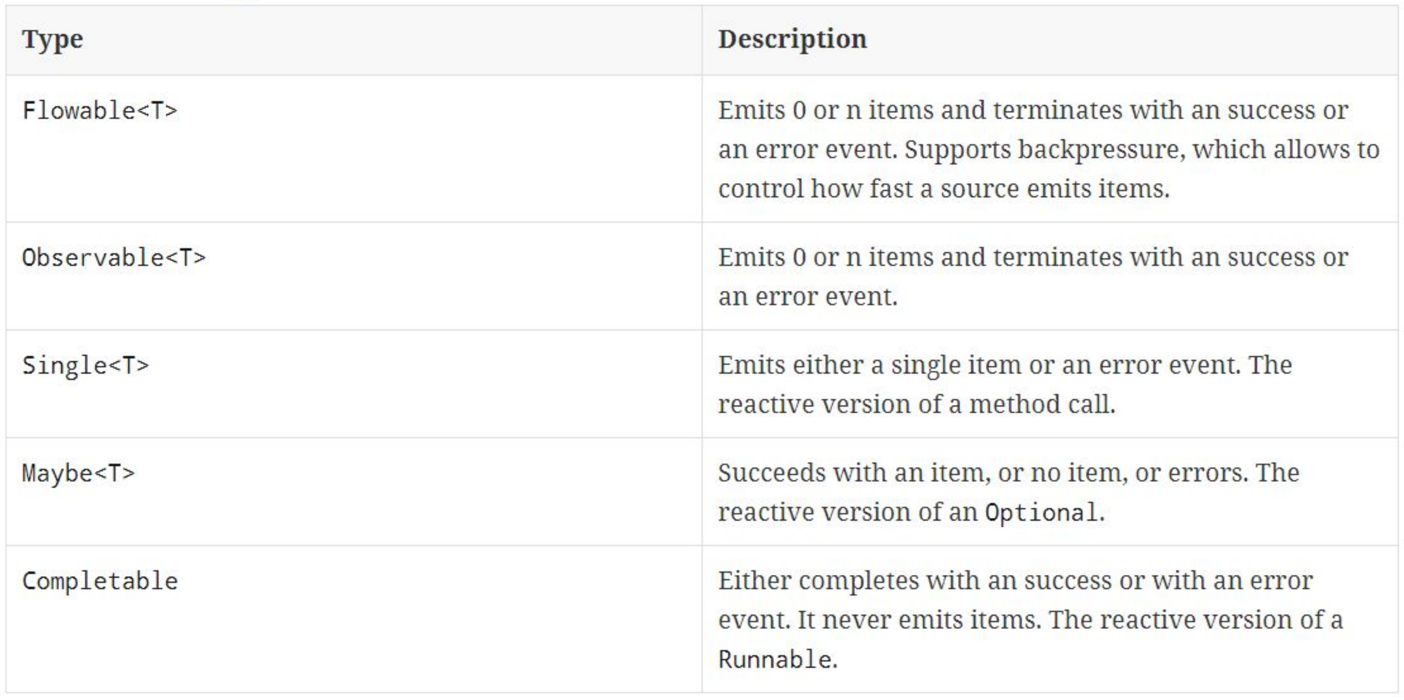
Observable: Observables are the sources for the data. Usually they start providing data once a subscriber starts listening. An observable may emit any number of items (including zero items). It can terminate either successfully or with an error.

::class that emits a stream of data or events. i.e. a class that can be used to perform some action, and publish the result. Ex:~

```
Observable observable = Observable.just("A", "B", "C", "D", "E",
"F");
```

You can create different types of observables:~

Table 1. Obervable types



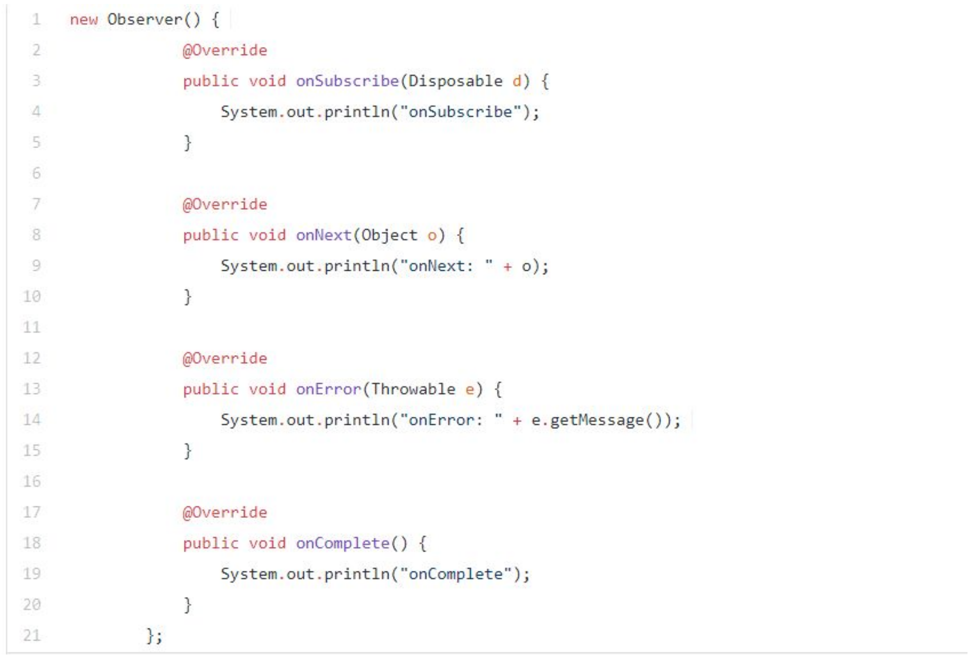
Type	Description
`Flowable<T>`	Emits 0 or n items and terminates with an success or an error event. Supports backpressure, which allows to control how fast a source emits items.
`Observable<T>`	Emits 0 or n items and terminates with an success or an error event.
`Single<T>`	Emits either a single item or an error event. The reactive version of a method call.
`Maybe<T>`	Succeeds with an item, or no item, or errors. The reactive version of an `Optional`.
`Completable`	Either completes with an success or with an error event. It never emits items. The reactive version of a `Runnable`.

Observer(Subscriber): class that receivers the events or data and acts upon it. i.e. a class that waits and watches the Observable, and reacts whenever the Observable publishes results.

The **Observer** has 4 interface methods to know the different states of the **Observable**.

- **onSubscribe():** This method is invoked when the Observer is subscribed to the Observable.
- **onNext():** This method is called when a new item is emitted from the Observable.
- **onError():** This method is called when an error occurs and the emission of data is not successfully completed.
- **onComplete():** This method is called when the Observable has successfully completed emitting all items.

we hold the reference to the returned Subscription object and later invoke `subscription#unsubscribe()` when necessary. In Android, this is best invoked within `Activity#onDestroy()` or `Fragment#onDestroy()`.


```java
new Observer() {
            @Override
            public void onSubscribe(Disposable d) {
                System.out.println("onSubscribe");
            }

            @Override
            public void onNext(Object o) {
                System.out.println("onNext: " + o);
            }

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

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

::~**Points to remember**

- A observable can have any number of subscribers.
- If a new item is emitted from the observable, the `onNext()` method is called on each subscriber.
- If the observable finishes its data flow successful, the `onComplete()` method is called on each subscriber.
- If the observable finishes its data flow with an error, the `onError()` method is called on each subscriber.

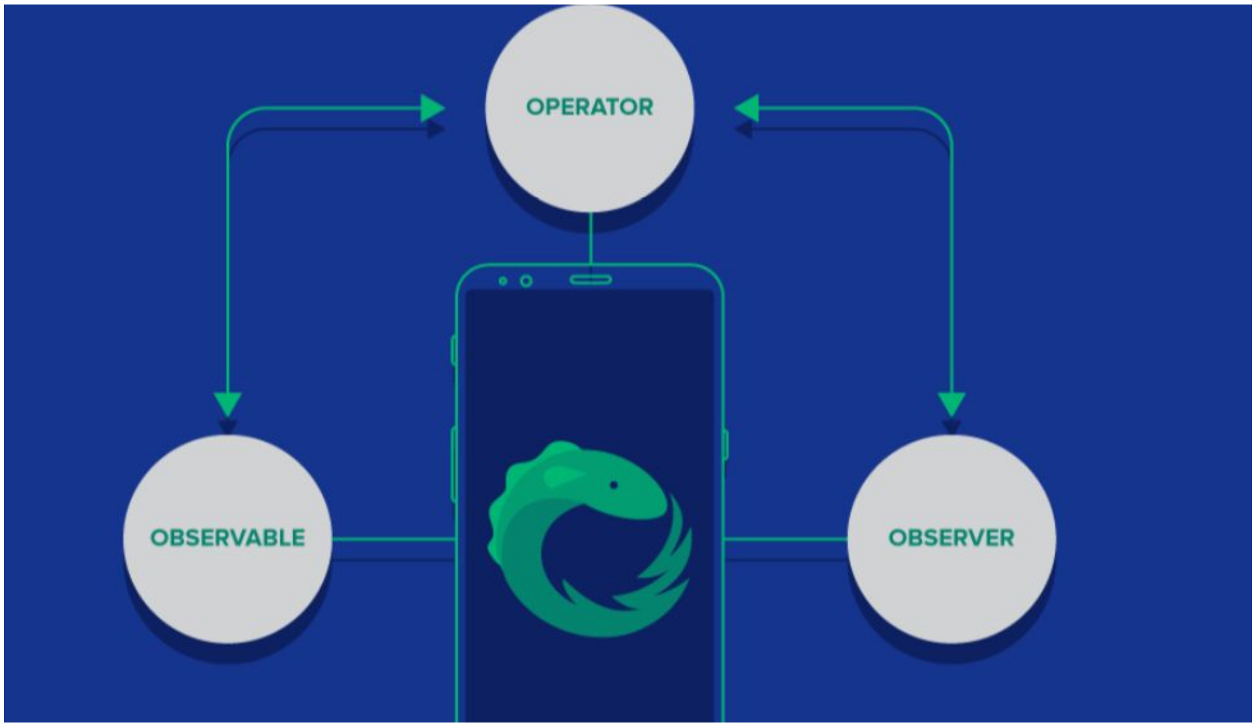
Basic Example:

NOte:`// Call unsubscribe when appropriate subscription.unsubscribe();`

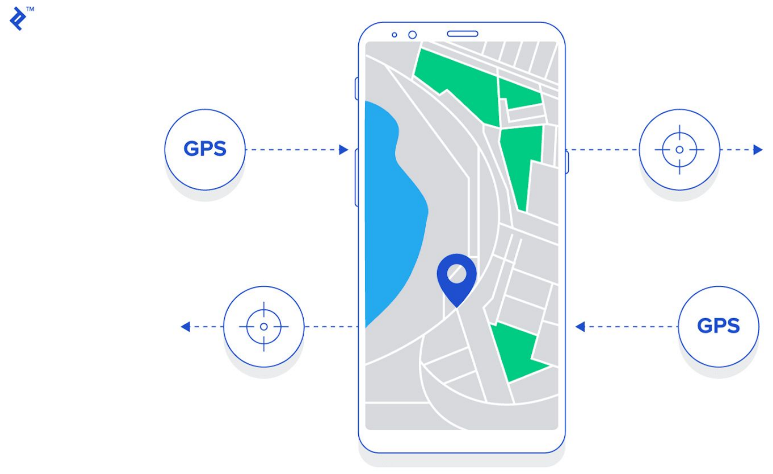
```
Observable<Integer> observable = Observable.just(1, 2, 3);

observable.subscribe(new Observer<Integer>() {

    @Override public void onCompleted() {

        Log.d("Test", "In onCompleted()");

    }

@Override public void onError(Throwable e) {

        Log.d("Test", "In onError()");

    }

@Override public void onNext(Integer integer) {

        Log.d("Test", "In onNext():" + integer);

    }
});
```

Rx- Operators:

Operators allow you to manipulate the data that was emitted or create new Observables.



If you think about it, a stream is not a new concept: click events can be a stream, location updates can be a stream, push notifications can be a stream, and so on.



Some of the most common operations found in functional programming (such as map, filter, reduce, etc.) can also be applied to an Observable stream. Let's look at map as an example:

```
Observable.just(1, 2, 3, 4, 5).map(new Func1<Integer, Integer>() {
   @Override public Integer call(Integer integer) {
       return integer * 3;
   }
}).subscribe(new Observer<Integer>() {
   @Override public void onCompleted() {
       // ...
   }

   @Override public void onError(Throwable e) {
       // ...
   }

   @Override public void onNext(Integer integer) {
       // ...
   }
});
```

Given the stream above, say we wanted to only receive even numbers. This can be achieved by chaining a *filter* operation.

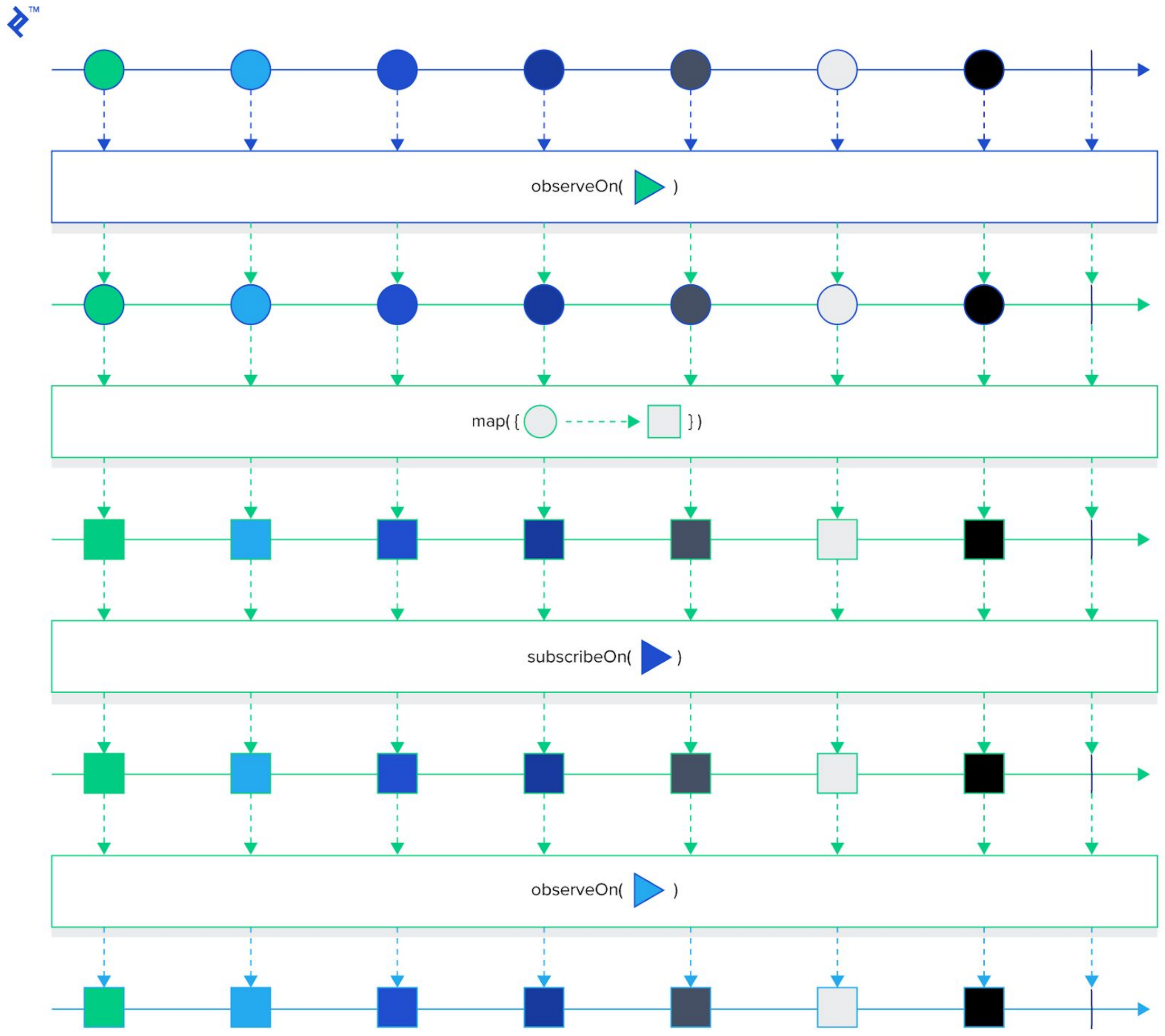
```
Observable.just(1, 2, 3, 4, 5).map(new Func1<Integer, Integer>() {
   @Override public Integer call(Integer integer) {
       return integer * 3;
   }
}).filter(new Func1<Integer, Boolean>() {
   @Override public Boolean call(Integer integer) {
       return integer % 2 == 0;
   }
}).subscribe(new Observer<Integer>() {
   @Override public void onCompleted() {
       // ...
   }

   @Override public void onError(Throwable e) {
       // ...
   }

   @Override public void onNext(Integer integer) {
       // ...
   }
});
```

Multithreading with RxJava

- Controlling the thread within which operations occur in the Observable chain is done by specifying the Scheduler within which an operator should occur. Essentially, you can think of a Scheduler as a thread pool that, when specified, an operator will use and run on.
- By default, if no such Scheduler is provided, the Observable chain will operate on the same thread where `Observable#subscribe(...)` is called. Otherwise, a Scheduler can be specified via `Observable#subscribeOn(Scheduler)` and/or `Observable#observeOn(Scheduler)` wherein the scheduled operation will occur on a thread chosen by the Scheduler.

NOTE**

- The key difference between the two methods is that `Observable#subscribeOn(Scheduler)` instructs the source Observable which Scheduler it should run on. The chain will continue to run on the thread from the Scheduler specified in `Observable#subscribeOn(Scheduler)` until a call to `Observable#observeOn(Scheduler)` is made with a different Scheduler. When such a call is made, all observers from there on out (i.e., subsequent operations down the chain) will receive notifications in a thread taken from the `observeOn` Scheduler.



**As shown in above example, In the context of Android, if a UI operation needs to take place as a result of a long operation, we'd want that operation to take place on the UI thread. For this purpose, we can use `AndroidScheduler#mainThread()`, one of the Schedulers provided in the RxAndroid library.

Let's create a demo App:

In this example, we will look at Retrofit, an HTTP client open sourced by Square which has built-in bindings with RxJava to interact with GitHub's API. Specifically, we'll create a simple app that presents all the starred repositories for a user given a GitHub username. If you want to jump ahead, the source code is available here- manishkaushik Github repo.

Operators for creating Observables:

1. Create
2. Defer
3. From
4. Interval
5. Just
6. Range
7. Repeat
8. Timer

1.**Create**:

This operator creates an Observable from scratch by calling observer methods programmatically. An emitter is provided through which we can call the respective interface methods when needed.
The below sample creates an Observable using Observable.create() method. The create() method does not have an option to pass values. So we have to create the list beforehand

and perform operations on the list inside the onNext() method. The below code will print each item from the list.

```
final List<String> alphabets = getAlphabetList();

    /*
     * Observable.create() -> We will need to call the
     * respective methods of the emitter such as onNext()
     * & onComplete() or onError()
     *
     * */
    Observable observable = Observable.create(new ObservableOnSubscribe() {
      @Override
      public void subscribe(ObservableEmitter emitter) {

        try {

          /*
           * The emitter can be used to emit each list item
           * to the subscriber.
           *
           * */
          for (String alphabet : alphabets) {
            emitter.onNext(alphabet);
          }

          /*
           * Once all the items in the list are emitted,
           * we can call complete stating that no more items
           * are to be emitted.
           *
           * */
          emitter.onComplete();

        } catch (Exception e) {

          /*
           * If an error occurs in the process,
            * we can call error.
            *
            * */
           emitter.onError(e);
        }
    }
});


/*
 * We create an Observer that is subscribed to Observer.
 * The only function of the Observer in this scenario is
 * to print the valeus emitted by the Observer.
 *
 * */
Observer observer = new Observer() {
    @Override
    public void onSubscribe(Disposable d) {
        System.out.println("onSubscribe");
    }

    @Override
    public void onNext(Object o) {
        System.out.println("onNext: " + o);
    }

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

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

/*
 * We can call this method to subscribe
```

```
        * the observer to the Observable.
        * */
       observable.subscribe(observer);
```

2. Defer

This operator does not create the **Observable** until the **Observer** subscribes. The only downside to defer() is that it creates a new **Observable** each time you get a new **Observer**. **create**() can use the same function for each subscriber, so it's more efficient.

```
public class DeferClass {
  private String value;

  public void setValue(String value) {
    this.value = value;
  }

/*
 * DeferClass instance = new DeferClass();
 * Observable<String> value = instance.valueObservable();
 * instance.setValue("Test Value");
 * value.subscribe(System.out::println); //null will be printed
 *
 * Suppose we call the below class and create an Observable,
 * when we call print, it will print null, since value had yet
 * to be initialized when Observable.just() is called.
 */
  public Observable<String> valueObservable() {
    return Observable.just(value);
  }
}


public class DeferClass {
  private String value;

  public void setValue(String value) {
    this.value = value;
  }

/*
 * Now if we wrap the code inside
 * Observable.defer(), none of the code inside of defer()
 * is executed until subscription.
 */
  public Observable<String> valueObservable() {
    return Observable.defer(() -> Observable.just(value));
  }
}
```

3.FROM

This operator creates an Observable from set of items using an Iterable, which means we can pass a list or an array of items to the Observable and each item is emitted one at a time. Some of the examples of the operators include fromCallable(), fromFuture(), fromIterable(), fromPublisher(), fromArray().

```
Observable.fromArray(new String[]{"A", "B", "C", "D", "E", "F"})
        .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(String string) {
                System.out.println("onNext: " + string);
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("onError: " + e.getMessage());
            }
        @Override
        public void onComplete() {

        }
    });
```

4.Interval

This operator creates an Observable that emits a sequence of integers spaced by a particular time interval.

Sample Implementation: The below sample creates an Observable using Observable.interval() method. The below code will print values from 0 after every second.

```
/*

 * This will print values from 0 after every second.

 */

Observable.interval(1, TimeUnit.SECONDS)

        .subscribe(new Observer<Long>() {

            @Override

            public void onSubscribe(Disposable d) { }



            @Override
        public void onNext(Long value) {

            System.out.println("onNext: " + value);

        }



        @Override

        public void onError(Throwable e) {

            System.out.println("onError: " + e.getMessage());

        }



        @Override

        public void onComplete() {}

    });
```

5.Just

This operator takes a list of arguments (maximum **10**) and converts the items into Observable items. just() makes only 1 emission. For instance, If an array is passed as a parameter to the just() method, the array is emitted as single item instead of individual numbers. Note that if you pass null to just(), it will return an Observable that *emits* null as an item.

```
Observable.just(new String[]{"A", "B", "C", "D", "E", "F"})
        .subscribe(new Observer<String[]>() {
            @Override
            public void onSubscribe(Disposable d) { }

            @Override
            public void onNext(String[] strings) {
                System.out.println("onNext: " + Arrays.toString(strings));
            }

            @Override
            public void onError(Throwable e) {   }

            @Override
            public void onComplete() { }
        });
```

6. Range

This operator creates an Observable that emits a range of sequential integers. The function takes two arguments: the **starting number** and **length**.

```java
Observable.range(2, 5)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

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

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
        });
```

Operators for Transforming Observables

1. Buffer
2. Map
3. FlatMap
4. SwitchMap
5. ConcatMap
6. GroupBy
7. Scan

We will discuss a few of them only:

1.Buffer

This operator periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time.
The below code will emit 2 items at a time since the buffer is specified as 2.

```
Observable.just("A", "B", "C", "D", "E", "F")
        .buffer(2)
        .subscribe(new Observer<List<String>>() {
          @Override
          public void onSubscribe(Disposable d) {}

          @Override
          public void onNext(List<String> strings) {
            System.out.println("onNext(): ");
            for (String s : strings) {
              System.out.println("String: " + s);
            }
          }

          @Override
          public void onError(Throwable e) { }

          @Override
          public void onComplete() {}
        });
```

Ouput:

```
onNext():
String: A
String: B onNext(): etc
```

2.**Map**

This operator transforms the items emitted by an Observable by applying a function to each item. map() operator allows for us to modify the emitted item from the Observable and then emits the modified item.

```
getOriginalObservable()
        .map(new Function<Integer, Integer>() {
            @Override
            public Integer apply(final Integer integer)  {
                return (integer * 2);
            }
        })
        .subscribeOn(Schedulers.io())
        .subscribe(new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

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

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
 });


private Observable<Integer> getOriginalObservable() {
    final List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 6);

    return Observable
        .create(new ObservableOnSubscribe<Integer>() {
```

```
        @Override
        public void subscribe(ObservableEmitter<Integer> emitter) {
            for(Integer integer : integers) {

                if (!emitter.isDisposed()) {
                    emitter.onNext(integer);
                }
            }

            if(!emitter.isDisposed()) {
                emitter.onComplete();
            }
        }

    });
}
```

Scenarios we can use the different operators:

- **Map** operator can be used when we fetch items from the server and need to modify it before emitting to the UI.
- **FlatMap** operator can be used when we know that the order of the items are not important.
- **SwitchMap** is best suited for scenarios such as a feed page, when pull to refresh is enabled. When user refreshes the screen, the older feed response is ignored and only the latest request results are emitted to the UI when using a SwitchMap.