Reactive Extensions

RxJava / RxAndroid

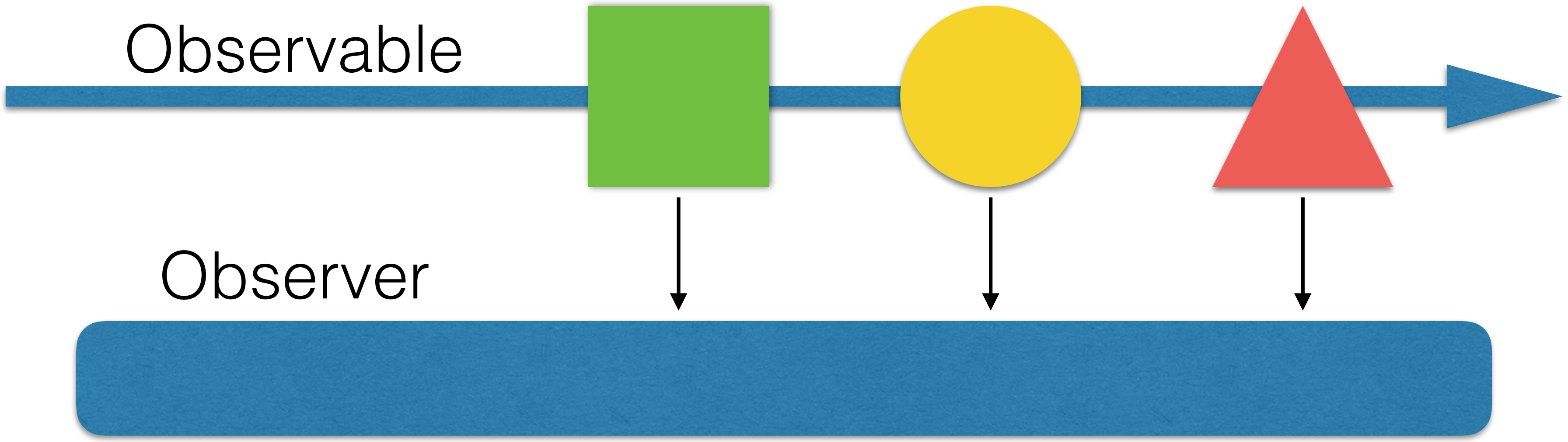
What is Reactive?

Reactive programming is programming with asynchronous data streams

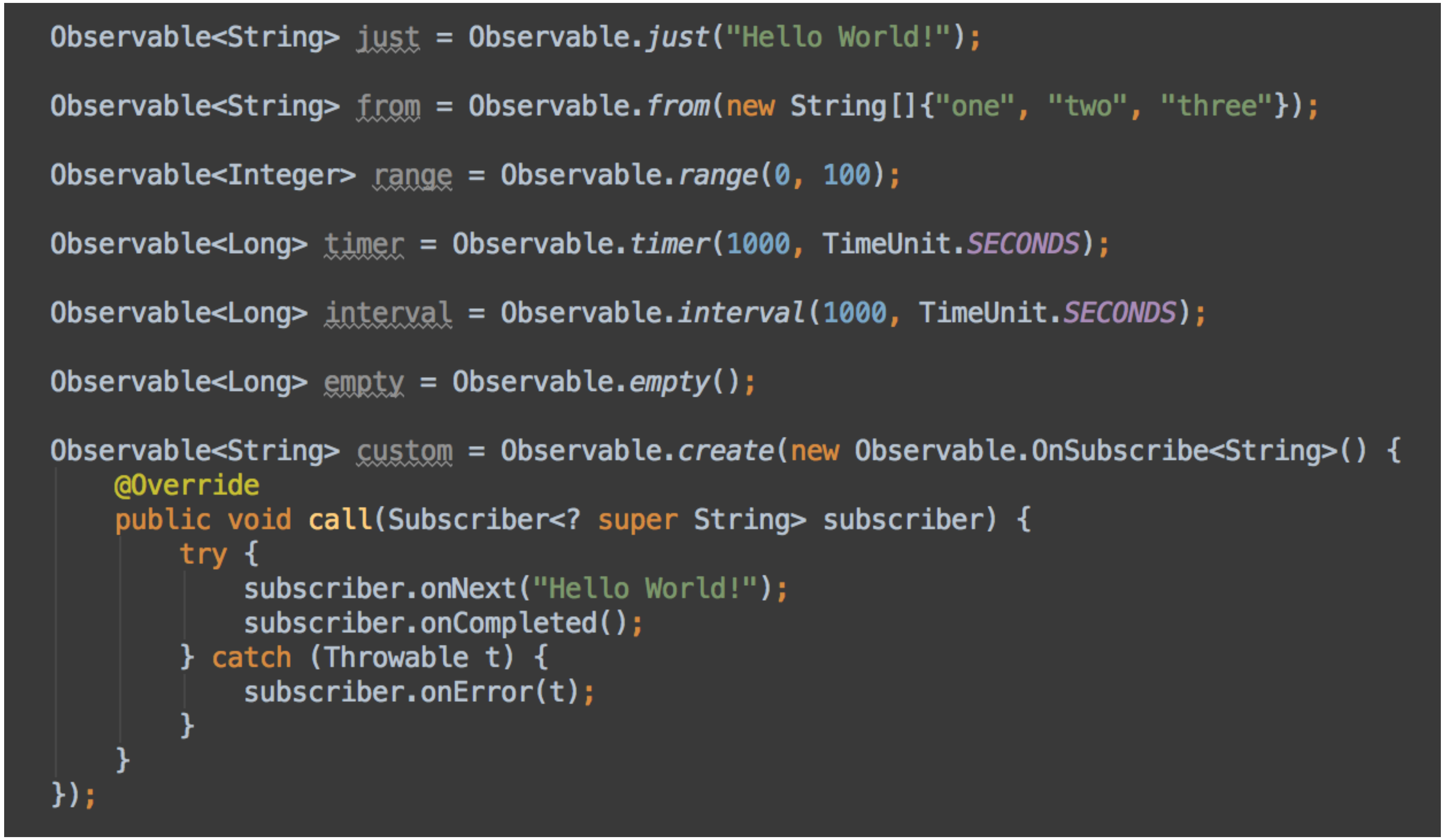
Main components

Observable (emits async data stream)

Observer (consumes async data stream)

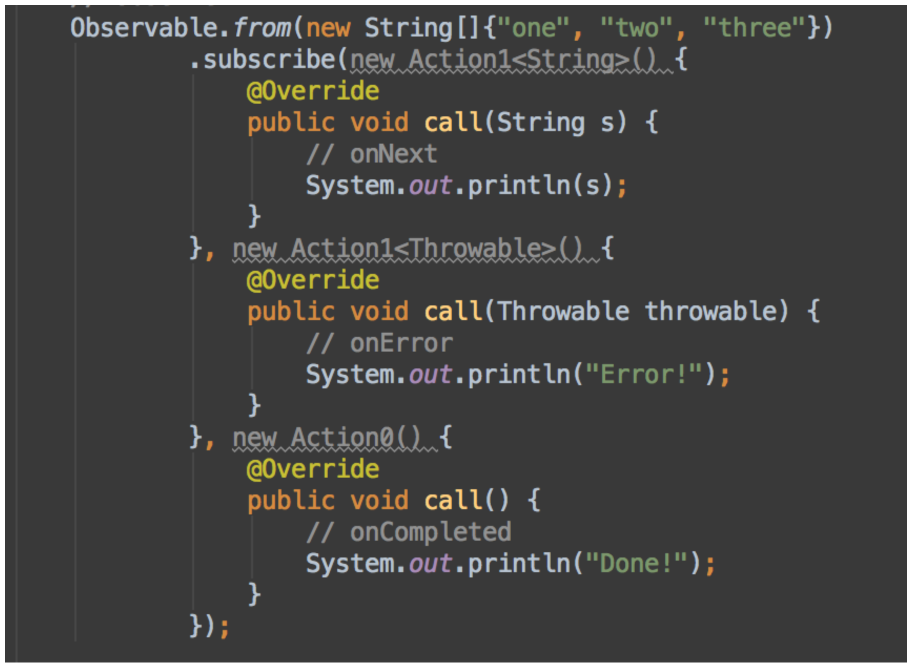


Observables

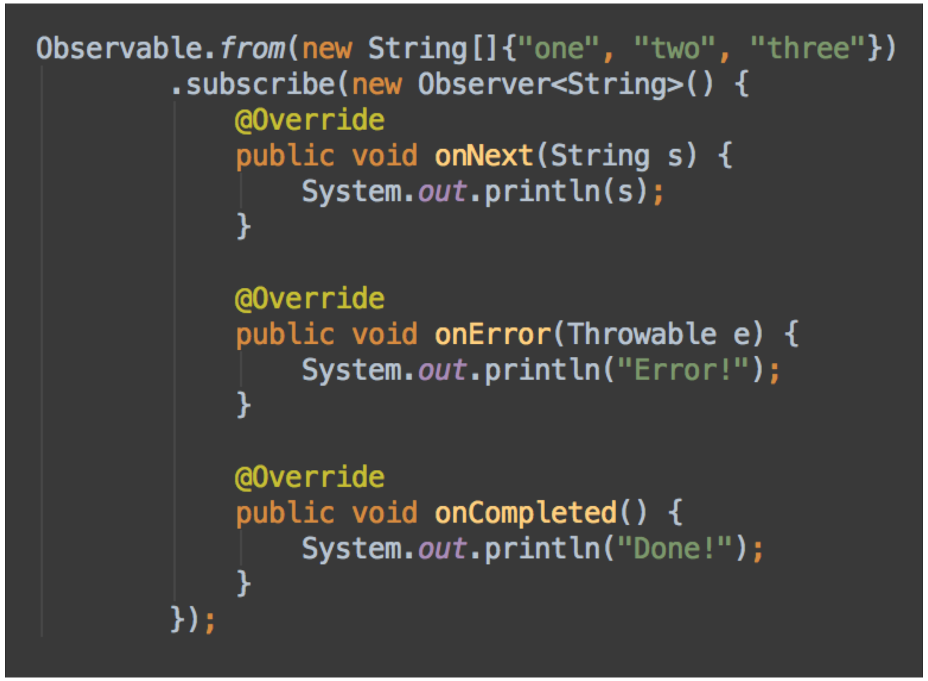


```
Observable<String> just = Observable.just("Hello World!");

Observable<String> from = Observable.from(new String[]{"one", "two", "three"});

Observable<Integer> range = Observable.range(0, 100);

Observable<Long> timer = Observable.timer(1000, TimeUnit.SECONDS);

Observable<Long> interval = Observable.interval(1000, TimeUnit.SECONDS);

Observable<Long> empty = Observable.empty();

Observable<String> custom = Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        try {
            subscriber.onNext("Hello World!");
            subscriber.onCompleted();
        } catch (Throwable t) {
            subscriber.onError(t);
        }
    }
});
```

Observers

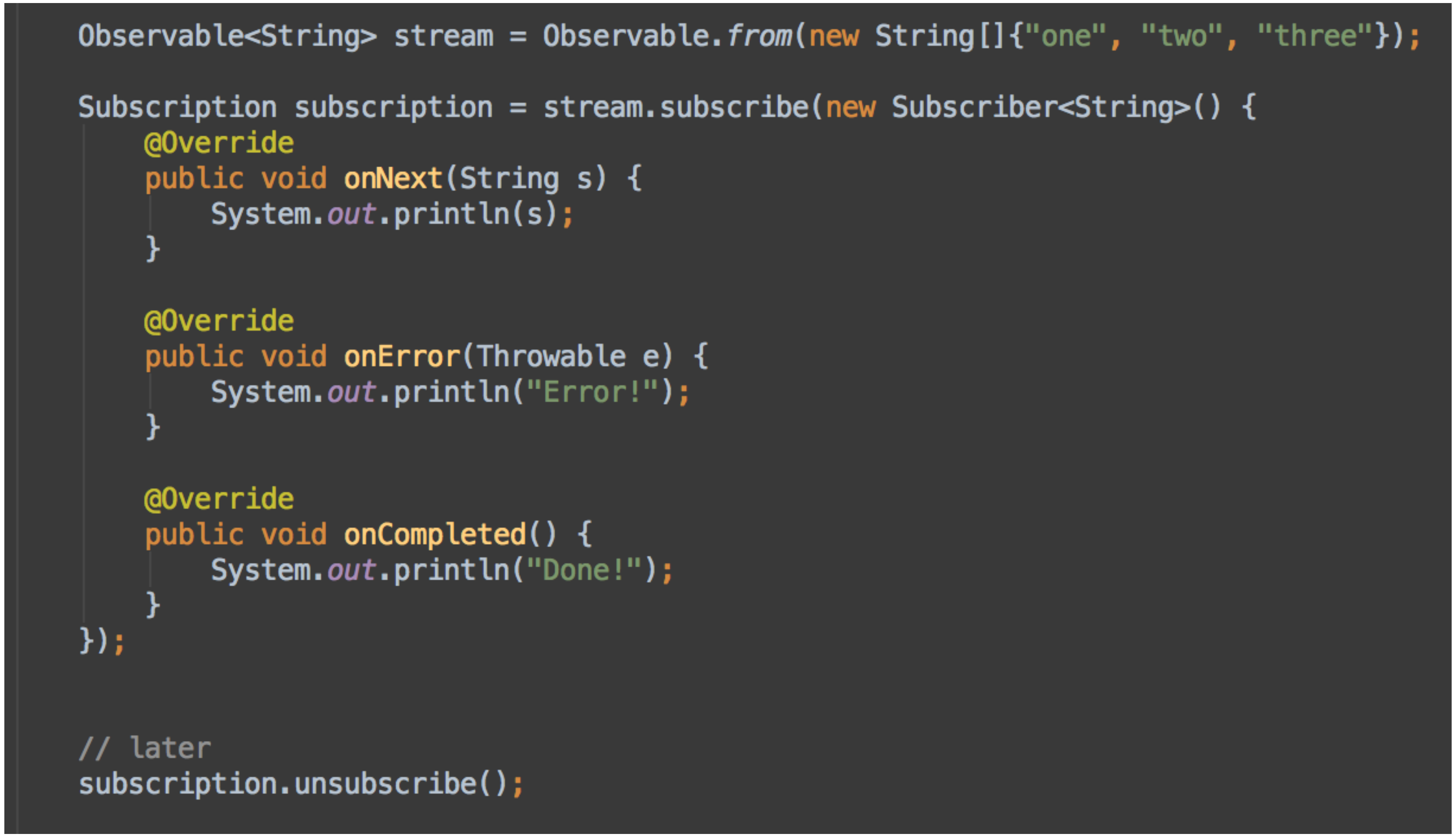


```
Observable.from(new String[]{"one", "two", "three"})
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                // onNext
                System.out.println(s);
            }
        }, new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                // onError
                System.out.println("Error!");
            }
        }, new Action0() {
            @Override
            public void call() {
                // onCompleted
                System.out.println("Done!");
            }
        });
```



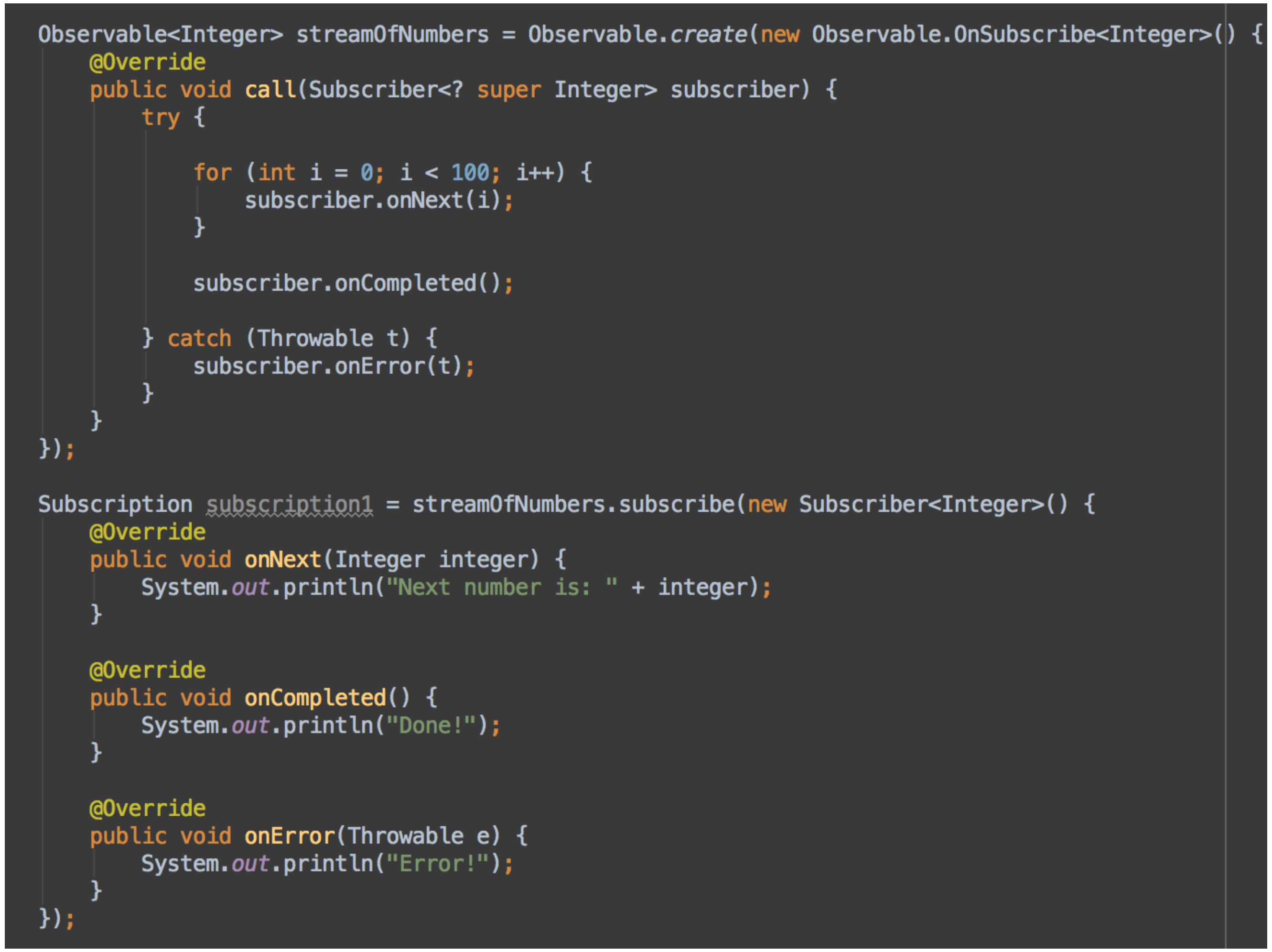
```
Observable.from(new String[]{"one", "two", "three"})
        .subscribe(new Observer<String>() {
            @Override
            public void onNext(String s) {
                System.out.println(s);
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("Error!");
            }

            @Override
            public void onCompleted() {
                System.out.println("Done!");
            }
        });
```

Subscriber and Subscription



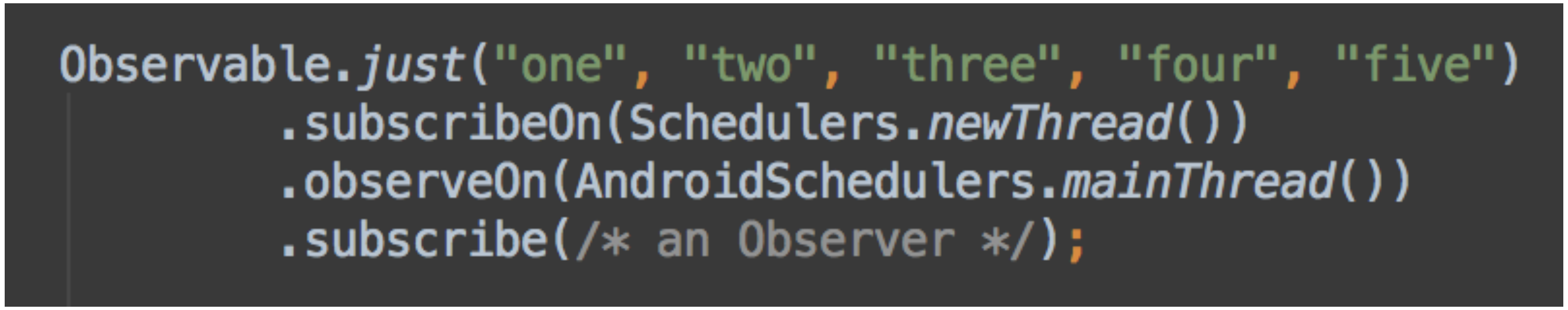
```java
Observable<String> stream = Observable.from(new String[]{"one", "two", "three"});

Subscription subscription = stream.subscribe(new Subscriber<String>() {
    @Override
    public void onNext(String s) {
        System.out.println(s);
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Error!");
    }

    @Override
    public void onCompleted() {
        System.out.println("Done!");
    }
});


// later
subscription.unsubscribe();
```

Observable, Subscriber and Subscription



```java
Observable<Integer> streamOfNumbers = Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        try {

            for (int i = 0; i < 100; i++) {
                subscriber.onNext(i);
            }

            subscriber.onCompleted();

        } catch (Throwable t) {
            subscriber.onError(t);
        }
    }
});

Subscription subscription1 = streamOfNumbers.subscribe(new Subscriber<Integer>() {
    @Override
    public void onNext(Integer integer) {
        System.out.println("Next number is: " + integer);
    }

    @Override
    public void onCompleted() {
        System.out.println("Done!");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Error!");
    }
});
```

Schedulers



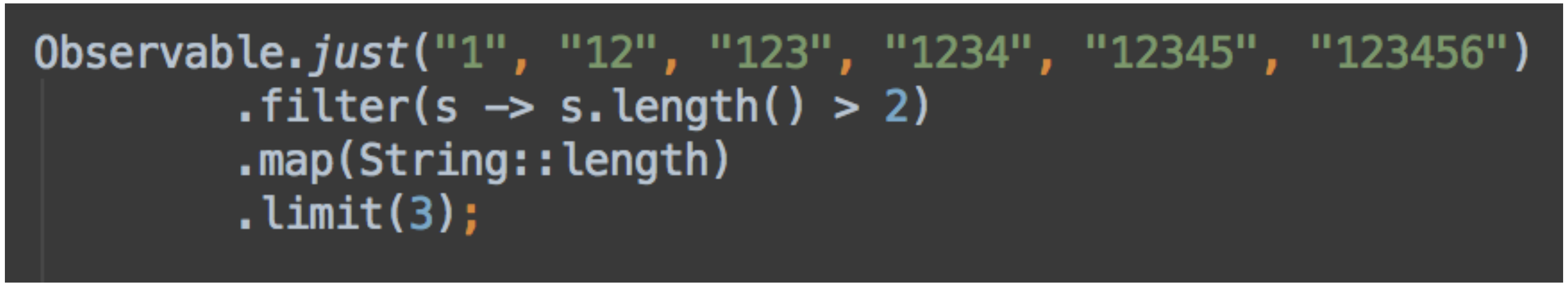
```
Observable.just("one", "two", "three", "four", "five")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(/* an Observer */);
```

Operators

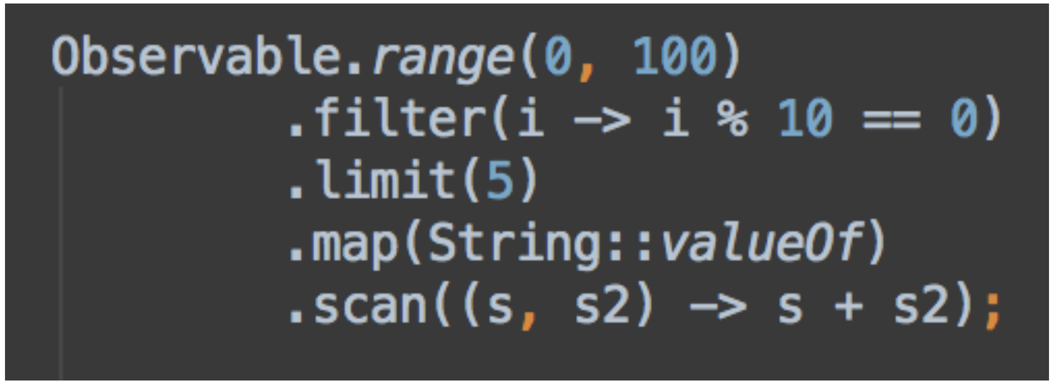
- `Aggregate`
- **`All`**
- **`Amb`**
- `and_`
- **`And`**
- `Any`
- `apply`
- `as_blocking`
- `asObservable`
- `AssertEqual`
- `asyncAction`
- `asyncFunc`
- **`Average`**
- `averageDouble`
- `averageFloat`
- `averageInteger`
- `averageLong`
- `blocking`
- **`Buffer`**
- `bufferWithCount`
- `bufferWithTime`
- `bufferWithTimeOrCount`
- `byLine`
- `cache`
- `case`
- `Cast`
- **`Catch`**
- `catchError`
- `catchException`
- `collect`
- `collect` (RxScala version of **`Filter`**)
- **`CombineLatest`**
- `combineLatestWith`
- **`Concat`**
- `concat_all`
- `concatMap`
- `concatMapObserver`
- `concatMapTo`
- `concatAll`
- `concatWith`
- **`Connect`**
- `connect_forever`
- `cons`
- **`Contains`**
- `controlled`
- **`Count`**
- `countLong`
- **`Create`**
- `cycle`
- **`Debounce`**
- `decode`
- **`DefaultIfEmpty`**
- **`Defer`**
- `deferFuture`
- **`Delay`**
- `delaySubscription`
- `delayWithSelector`
- **`Dematerialize`**
- **`Distinct`**
- `distinctKey`
- `distinctUntilChanged`
- `distinctUntilKeyChanged`
- **`Do`**
- `doAction`
- `doOnCompleted`
- `doOnEach`
- `doOnError`
- `doOnRequest`
- `doOnSubscribe`
- `doOnTerminate`
- `doOnUnsubscribe`
- `doseq`
- `doWhile`
- `drop`
- `dropRight`
- `dropUntil`
- `dropWhile`
- **`ElementAt`**
- `ElementAtOrDefault`
- **`Empty`**
- `emptyObservable`
- `empty?`
- `encode`
- `ensures`
- `error`
- `every`
- `exclusive`
- `exists`
- `expand`
- `failWith`
- **`Filter`**
- `filterNot`
- `Finally`
- `finallyAction`
- `finallyDo`
- `find`
- `findIndex`
- **`First`**
- `FirstOrDefault`
- `firstOrElse`
- **`FlatMap`**
- `flatMapFirst`
- `flatMapIterable`
- `flatMapIterableWith`
- `flatMapLatest`
- `flatMapObserver`
- `flatMapWith`
- `flatMapWithMaxConcurrent`
- `flat_map_with_index`
- `flatten`
- `flattenDelayError`
- `foldl`
- `foldLeft`
- `for`
- `forall`
- `ForEach`
- `forEachFuture`
- `forIn`
- `forkJoin`
- **`From`**
- `fromAction`
- `fromArray`
- `FromAsyncPattern`
- `fromCallable`
- `fromCallback`
- `FromEvent`
- `FromEventPattern`
- `fromFunc0`
- `from_future`
- `from_iterable`
- `fromIterator`
- `from_list`
- `fromNodeCallback`
- `fromPromise`
- `fromRunnable`
- `Generate`

. . .

Using operators



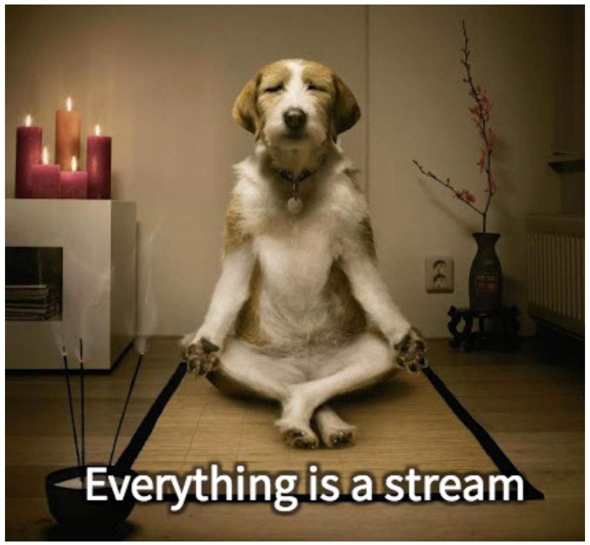
```java
Observable.just("1", "12", "123", "1234", "12345", "123456")
        .filter(s -> s.length() > 2)
        .map(String::length)
        .limit(3);
```



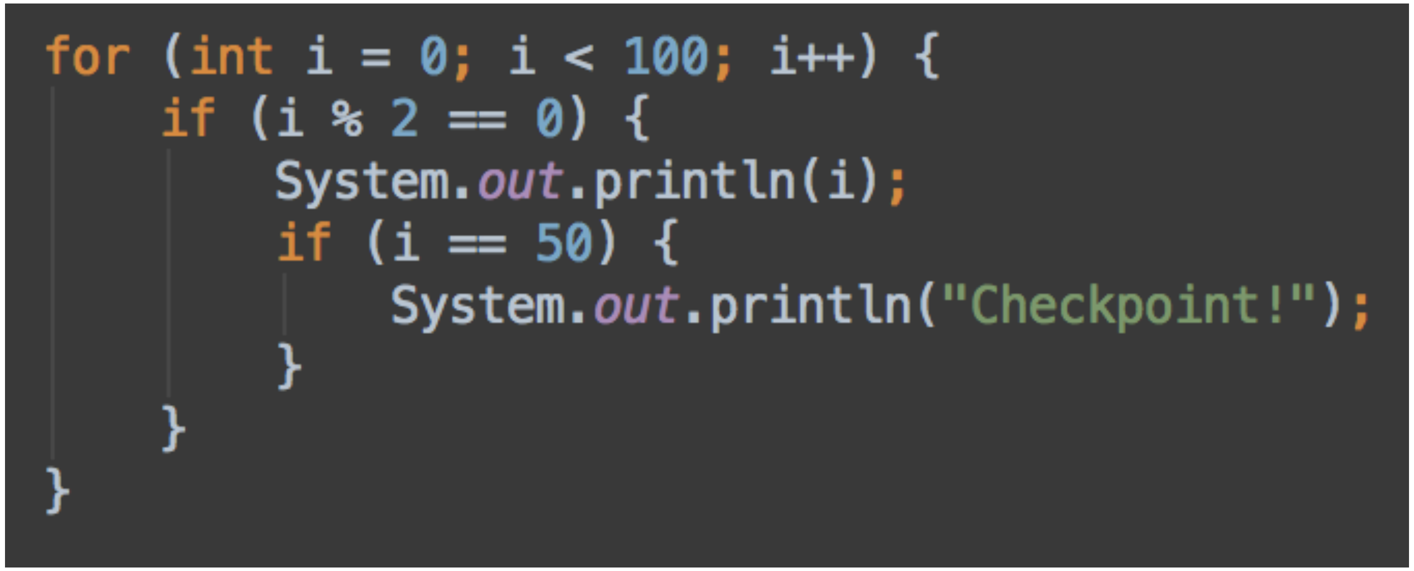
```java
Observable.range(0, 100)
        .filter(i -> i % 10 == 0)
        .limit(5)
        .map(String::valueOf)
        .scan((s, s2) -> s + s2);
```

Streams

Anything can be a stream (variables, user inputs, properties, data structures, etc.)

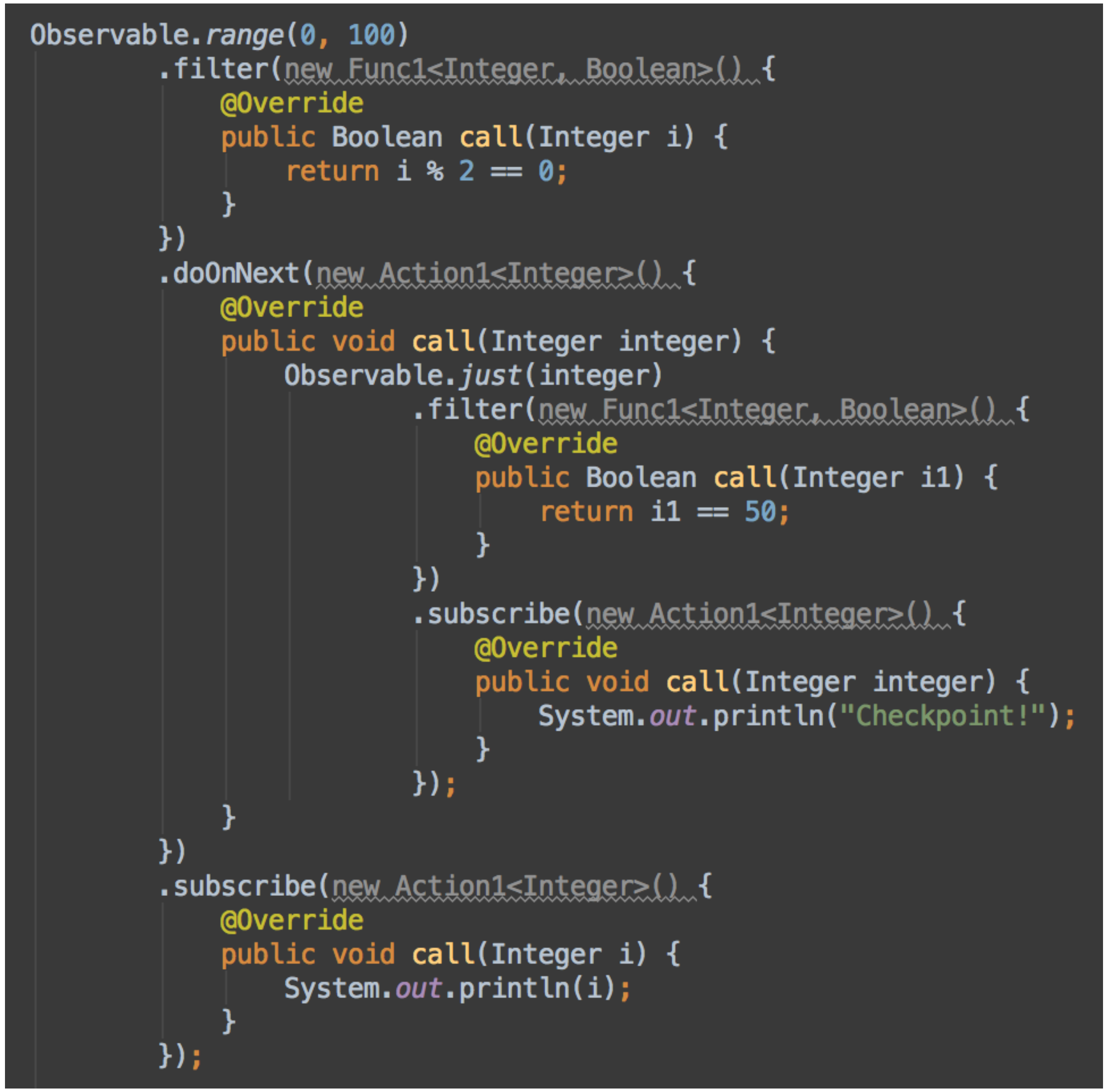


Thinking in streams (1)



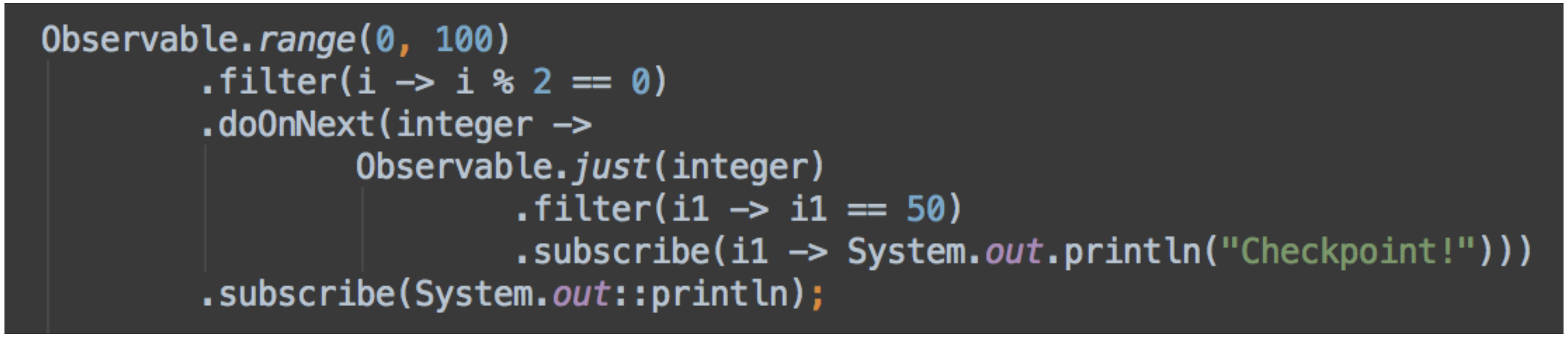
```java
for (int i = 0; i < 100; i++) {
    if (i % 2 == 0) {
        System.out.println(i);
        if (i == 50) {
            System.out.println("Checkpoint!");
        }
    }
}
```

Thinking in streams (1)



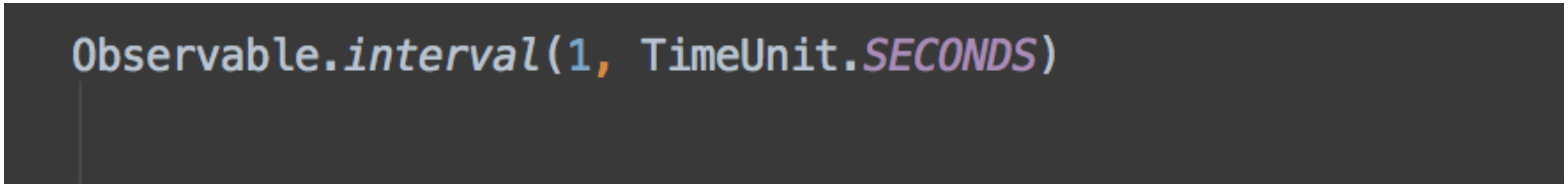
```
Observable.range(0, 100)
        .filter(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer i) {
                return i % 2 == 0;
            }
        })
        .doOnNext(new Action1<Integer>() {
            @Override
            public void call(Integer integer) {
                Observable.just(integer)
                        .filter(new Func1<Integer, Boolean>() {
                            @Override
                            public Boolean call(Integer i1) {
                                return i1 == 50;
                            }
                        })
                        .subscribe(new Action1<Integer>() {
                            @Override
                            public void call(Integer integer) {
                                System.out.println("Checkpoint!");
                            }
                        });
            }
        })
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer i) {
                System.out.println(i);
            }
        });
```

Thinking in streams (1)



```
Observable.range(0, 100)
        .filter(i -> i % 2 == 0)
        .doOnNext(integer ->
                Observable.just(integer)
                        .filter(i1 -> i1 == 50)
                        .subscribe(i1 -> System.out.println("Checkpoint!")))
        .subscribe(System.out::println);
```

Thinking in streams (2)



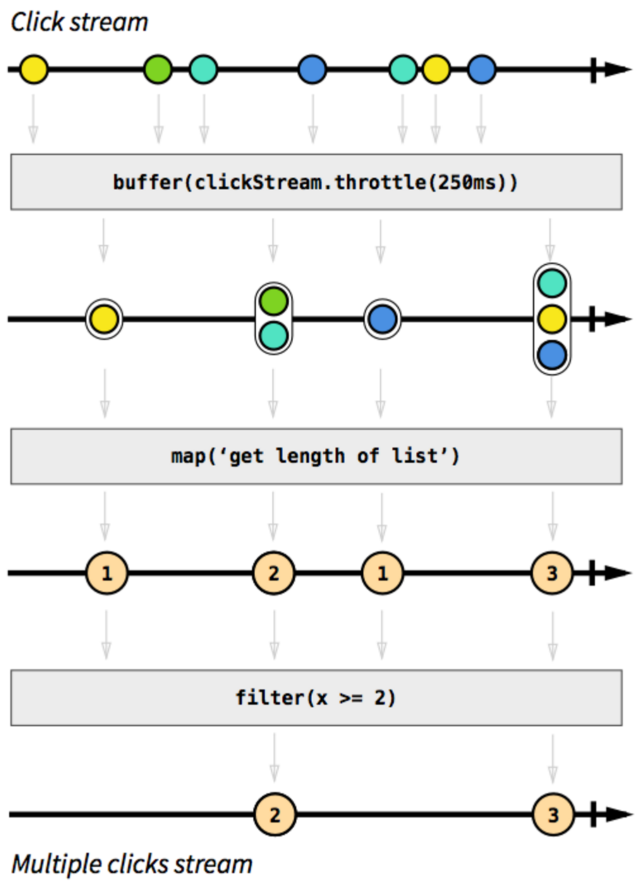
```
Observable.interval(1, TimeUnit.SECONDS)
```

Practice (1)

We want to have a stream of multiple-click events.

Practice (1)

We want to have a stream of multiple-click events.


Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream