**RX JAVA OPERATORS**

All the operators are categorized depending on the kind of work it do. Some operators are used to Create Observables. The operators like **create, just, fromArray, range** creates an Observable.

Some operators such as **debounce, filter, skip, last** are used to filter the data emitted by an Observable. The operators like **buffer, map, flatMap, switchMap, compose** creates an Observable by transform the data emitted by another Observable.

**take(n)** acts exactly opposite to skip. It takes first N emissions of an Observable.

**take(4)** takes first 4 emissions i.e **1, 2, 3, 4** and skips the remaining.

**takeLast(n)** emits last N items from an Observable.

**takeLast(4)** takes last 4 emissions i.e **7, 8, 9, 10** and skips the remaining.

**take** operator has three main variants. Let’s look at each of them.

**take**Until**()** emits items until a second observable doesn’t start emitting values. Alternatively, it can be used for conditions as well. operator is like a do-while loop. It first prints the statement before checking for the condition for the next iteration.

**takeWhile()** emits items as long as a condition is true. It ignores the remaining.

**Distinct** operator filters out items emitted by an Observable by avoiding duplicate items in the list.

Using **distinct()**, emission of duplicates can be avoided.

**toList** and **toSortedList** are used to convert the emission into another observable that is of the type list.  
**toSortedList** sorts the emission in ascending order.  
We can also set our own comparator to sort.

**skip() is somewhat the opposite of take.** **skip is the ideal operator to use when avoiding null values.**

**skipLast()** ignored the last n values

**skipWhile()** ignores all the values until a specific condition is met. It emits all the remainder of values.

**skipUntil()** ignores all the values until another observable starts emitting. We’ll look at this later

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); Observable<Integer> skipObservable = Observable.from(numbers); skipObservable.skip(3).subscribe(System.out::println); //prints 4 and 5 skipObservable.skipLast(3).subscribe(System.out::println); //prints 1 and 2 skipObservable.skipWhile(i-> i<3).subscribe(System.out::println); prints 3 4 and 5

### startWith() appends the given element at the start of the emission.

**Reduce()** operator acts as an accumulator. It adds a the next value to previously added values.  
Finally prints the accumulated value for the subscriber. operator is useful for calculating the sum, appending strings etc.  
The following code finds the length of the concatenated strings.

**Scan()** operator unlike reduce, prints the accumulator value incrementally

### **all()** operator checks whether each value meets the condition. It returns a true/false.

**Contains()** It checks if the value mentioned exists in the Observable lot.

**elementAt()** It prints the value present at the given index among the list of emitted values.

**zip** is used to pair each emission from each of the observables. Each observable would wait for the others to emit the current value and then each of the values is available for you in a function.  
zip is useful to concatenate different types

 Here an Observable is created using **fromArray()** operator which emits the numbers from **1 to 20**.

Integer[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

                11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

        Observable.fromArray(numbers)

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .subscribe(new DisposableObserver<Integer>() {

                    @Override

                    public void onNext(Integer integer) {

                        Log.d(TAG, "Number: " + integer);

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.d(TAG, "All numbers emitted!");

                    }

                });

 using **range(1, 20)** operator as below.

Observable.range(1, 20)

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .subscribe(new DisposableObserver<Integer>() {

                    @Override

                    public void onNext(Integer integer) {

                        Log.e(TAG, "Number: " + integer);

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.e(TAG, "All numbers emitted!");

                    }

                });

**Chaining operators**

Sometimes the desired data stream can’t achieved using a single operator. In that case you can use multiple operators together. When multiple operators are used, the operators takes the result from the previous operator.

Let’s take same example of emitting numbers from **1 to 20**. But in this case we want to **filter out the even numbers** along with we want to **append a string** at the end of each number.

* **range():** Range operator generates the numbers from 1 to 20
* **filter():** Filters the numbers by applying a condition onto each number
* **map():** Map transform the data from Integer to String by appending the string at the end
* In the operator chain, **filter()** will be executed first and **map()** takes the result from filter and performs it’s job

Observable.range(1, 20)

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .filter(new Predicate<Integer>() {

                    @Override

                    public boolean test(Integer integer) throws Exception {

                        return integer % 2 == 0;

                    }

                })

                .map(new Function<Integer, String>() {

                    @Override

                    public String apply(Integer integer) throws Exception {

                        return integer + " is even number";

                    }

                })

                .subscribe(new Observer<String>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                    }

                    @Override

                    public void onNext(String s) {

                        Log.d(TAG, "onNext: " + s);

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.d(TAG, "All numbers emitted!");

                    }

                });

**Max()** operator finds the maximum valued item in the Observable sequence and emits that value.

The below example emits the max value of an integer series.

Integer[] numbers = {5, 101, 404, 22, 3, 1024, 65};

Observable<Integer> observable = Observable.from(numbers);

MathObservable

     .max(observable)

     .subscribe(new Subscriber<Integer>() {

            @Override

            public void onCompleted() {

            }

            @Override

            public void onError(Throwable e) {

            }

            @Override

            public void onNext(Integer integer) {

                Log.d(TAG, "Max value: " + integer);

            }

        });

**Min()** operator emits the minimum valued item in the Observable data set.

Integer[] numbers = {5, 101, 404, 22, 3, 1024, 65};

Observable<Integer> observable = Observable.from(numbers);

MathObservable

      .min(observable)

      .subscribe(new Subscriber<Integer>() {

            @Override

            public void onCompleted() {

            }

            @Override

            public void onError(Throwable e) {

            }

            @Override

            public void onNext(Integer integer) {

                Log.d(TAG, "Min value: " + integer);

            }

        });

**Sum**():Calculates the sum of all the items emitted by an Observable and emits only the **Sum** value. In the below example, **sumInteger()** is used to calculate the sum of Integers. Likewise, we have **sumFloat()**, **sumDouble()** and **sumLong()** available to calculate sum of other primitive datatypes.

Integer[] numbers = {5, 101, 404, 22, 3, 1024, 65};

Observable<Integer> observable = Observable.from(numbers);

MathObservable

         .sumInteger(observable)

         .subscribe(new Subscriber<Integer>() {

                    @Override

                    public void onCompleted() {

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onNext(Integer integer) {

                        Log.d(TAG, "Min value: " + integer);

                    }

                });

**Average**  : Calculates the average of all the items emitted by an Observable and emits only the **Average** value.

The below example calculates the average value of integers using **averageInteger()** method. To calculate average of other datatypes, **averageFloat()**, **averageDouble()** and **averageLong()** are available.

MathObservable

          .averageInteger(observable)

          .subscribe(new Subscriber<Integer>() {

                    @Override

                    public void onCompleted() {

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onNext(Integer integer) {

                        Log.d(TAG, "Average: " + integer);

                    }

                });

**Count()** : Counts number of items emitted by an Observable and emits only the count value.

 we have an Observable that emits both **Male** and **Female** users. We can count **number of Male users** using count() operator as shown.

**filter()** filters the items by gender by applying **user.getGender().equalsIgnoreCase(“male”)** on each emitted item.

getUsersObservable()

                .filter(new Predicate<User>() {

                    @Override

                    public boolean test(User user) throws Exception {

                        return user.getGender().equalsIgnoreCase("male");

                    }

                })

                .count()

                .subscribeWith(new SingleObserver<Long>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                    }

                    @Override

                    public void onSuccess(Long count) {

                        Log.d(TAG, "Male users count: " + count);

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                });

private Observable<User> getUsersObservable() {

        String[] maleUsers = new String[]{"Mark", "John", "Trump", "Obama"};

        String[] femaleUsers = new String[]{"Lucy", "Scarlett", "April"};

        final List<User> users = new ArrayList<>();

        for (String name : maleUsers) {

            User user = new User();

            user.setName(name);

            user.setGender("male");

            users.add(user);

        }

        for (String name : femaleUsers) {

            User user = new User();

            user.setName(name);

            user.setGender("female");

            users.add(user);

        }

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        for (User user : users) {

                            if (!emitter.isDisposed()) {

                                emitter.onNext(user);

                            }

                        }

                        if (!emitter.isDisposed()) {

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

**Reduce()** :Reduce applies a function on each item and emits the final result. First, it applies a function to first item, takes the result and feeds back to same function on second item. This process continuous until the last emission. Once all the items are over, it emits the final result.

Below we have an Observable that emits numbers from 1 to 10. The **reduce()** operator calculates the sum of all the numbers and emits the final result.

Observable

        .range(1, 10)

        .reduce(new BiFunction<Integer, Integer, Integer>() {

            @Override

            public Integer apply(Integer number, Integer sum) throws Exception {

                return sum + number;

            }

        })

        .subscribe(new MaybeObserver<Integer>() {

            @Override

            public void onSubscribe(Disposable d) {

                disposable = d;

            }

            @Override

            public void onSuccess(Integer integer) {

                Log.e(TAG, "Sum of numbers from 1 - 10 is: " + integer);

            }

            @Override

            public void onError(Throwable e) {

                Log.e(TAG, "onError: " + e.getMessage());

            }

            @Override

            public void onComplete() {

                Log.e(TAG, "onComplete");

            }

        });

**Map** operator transform each item emitted by an Observable and emits the modified item.

public class MapOperatorActivity extends AppCompatActivity {

    private static final String TAG = MapOperatorActivity.class.getSimpleName();

    private Disposable disposable;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_map\_operator);

        getUsersObservable()

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .map(new Function<User, User>() {

                    @Override

                    public User apply(User user) throws Exception {

                        // modifying user object by adding email address

                        // turning user name to uppercase

                        user.setEmail(String.format("%s@rxjava.wtf", user.getName()));

                        user.setName(user.getName().toUpperCase());

                        return user;

                    }

                })

                .subscribe(new Observer<User>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                        disposable = d;

                    }

                    @Override

                    public void onNext(User user) {

                        Log.e(TAG, "onNext: " + user.getName() + ", " + user.getGender() + ", " + user.getAddress().getAddress());

                    }

                    @Override

                    public void onError(Throwable e) {

                        Log.e(TAG, "onError: " + e.getMessage());

                    }

                    @Override

                    public void onComplete() {

                        Log.e(TAG, "All users emitted!");

                    }

                });

    }

    /\*\*

     \* Assume this method is making a network call and fetching Users

     \* an Observable that emits list of users

     \* each User has name and email, but missing email id

     \*/

    private Observable<User> getUsersObservable() {

        String[] names = new String[]{"mark", "john", "trump", "obama"};

        final List<User> users = new ArrayList<>();

        for (String name : names) {

            User user = new User();

            user.setName(name);

            user.setGender("male");

            users.add(user);

        }

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        for (User user : users) {

                            if (!emitter.isDisposed()) {

                                emitter.onNext(user);

                            }

                        }

                        if (!emitter.isDisposed()) {

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        disposable.dispose();

    }

}

**SwithMap()** on the other hand is completely a different operator from FlatMap and ConcatMap. SwitchMap always return the latest Observable and emits the items from it.

public class SwitchMapOperatorActivity extends AppCompatActivity {

    private static final String TAG = SwitchMapOperatorActivity.class.getSimpleName();

    private Disposable disposable;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_switch\_map\_operator);

Observable<Integer> integerObservable =

                Observable.fromArray(new Integer[]{1, 2, 3, 4, 5, 6});

        // it always emits 6 as it un-subscribes the before observer

        integerObservable

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .switchMap(new Function<Integer, ObservableSource<Integer>>() {

                    @Override

                    public ObservableSource<Integer> apply(Integer integer) throws Exception {

                        return Observable.just(integer).delay(1, TimeUnit.SECONDS);

                    }

                })

                .subscribe(new Observer<Integer>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                        Log.d(TAG, "onSubscribe");

                        disposable = d;

                    }

                    @Override

                    public void onNext(Integer integer) {

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

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.d(TAG, "All users emitted!");

                    }

                });

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        disposable.dispose();

    }

}

[Buffer](http://reactivex.io/documentation/operators/buffer.html) gathers items emitted by an Observable into batches and emit the batch instead of emitting one item at a time.

Below, we have an Observable that emits integers from 1-9. When **buffer(3)** is used, it emits **3 integers** at a time.

Observable<Integer> integerObservable = Observable.just(1, 2, 3, 4,

        5, 6, 7, 8, 9);

integerObservable.subscribeOn(Schedulers.io())

        .observeOn(AndroidSchedulers.mainThread())

        .buffer(3)

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

            @Override

            public void onSubscribe(Disposable d) {

            }

            @Override

            public void onNext(List<Integer> integers) {

                Log.d(TAG, "onNext");

                for (Integer integer : integers) {

                    Log.d(TAG, "Item: " + integer);

                }

            }

            @Override

            public void onError(Throwable e) {

            }

            @Override

            public void onComplete() {

                Log.d(TAG, "All items emitted!");

            }

        });

**flatmap()**

The flatMap operator help you to transform one event to another Observable (or transform an event to zero, one, or more events).

It's a perfect operator when you want to call another method which return an Observable

public class FlatMapActivity extends AppCompatActivity {

    private static final String TAG = FlatMapActivity.class.getSimpleName();

    private Disposable disposable;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_flat\_map);

        getUsersObservable()

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .flatMap(new Function<User, Observable<User>>() {

                    @Override

                    public Observable<User> apply(User user) throws Exception {

                        // getting each user address by making another network call

                        return getAddressObservable(user);

                    }

                })

                .subscribe(new Observer<User>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                        Log.e(TAG, "onSubscribe");

                        disposable = d;

                    }

                    @Override

                    public void onNext(User user) {

                        Log.e(TAG, "onNext: " + user.getName() + ", " + user.getGender() + ", " + user.getAddress().getAddress());

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.e(TAG, "All users emitted!");

                    }

                });

    }

    /\*\*

     \* Assume this as a network call

     \* returns Users with address filed added

     \*/

    private Observable<User> getAddressObservable(final User user) {

        final String[] addresses = new String[]{

                "1600 Amphitheatre Parkway, Mountain View, CA 94043",

                "2300 Traverwood Dr. Ann Arbor, MI 48105",

                "500 W 2nd St Suite 2900 Austin, TX 78701",

                "355 Main Street Cambridge, MA 02142"

        };

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        Address address = new Address();

                        address.setAddress(addresses[new Random().nextInt(2) + 0]);

                        if (!emitter.isDisposed()) {

                            user.setAddress(address);

                            // Generate network latency of random duration

                            int sleepTime = new Random().nextInt(1000) + 500;

                            Thread.sleep(sleepTime);

                            emitter.onNext(user);

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

    /\*\*

     \* Assume this is a network call to fetch users

     \* returns Users with name and gender but missing address

     \*/

    private Observable<User> getUsersObservable() {

        String[] maleUsers = new String[]{"Mark", "John", "Trump", "Obama"};

        final List<User> users = new ArrayList<>();

        for (String name : maleUsers) {

            User user = new User();

            user.setName(name);

            user.setGender("male");

            users.add(user);

        }

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        for (User user : users) {

                            if (!emitter.isDisposed()) {

                                emitter.onNext(user);

                            }

                        }

                        if (!emitter.isDisposed()) {

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        disposable.dispose();

    }

}

**ConcatMap()**

Now consider the same example of FlatMap but replacing the operator with ConcatMap. Technically the both operators produces the same output but the sequence the data emitted changes.

* **ConcatMap()** maintains the order of items and waits for the current Observable to complete its job before emitting the next one.
* ConcatMap is more suitable when you want to maintain the order of execution.

public class ConcatMapOperatorActivity extends AppCompatActivity {

    private static final String TAG = ConcatMapOperatorActivity.class.getSimpleName();

    private Disposable disposable;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_concat\_map);

        getUsersObservable()

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .concatMap(new Function<User, Observable<User>>() {

                    @Override

                    public Observable<User> apply(User user) throws Exception {

                        // getting each user address by making another network call

                        return getAddressObservable(user);

                    }

                })

                .subscribe(new Observer<User>() {

                    @Override

                    public void onSubscribe(Disposable d) {

                        Log.e(TAG, "onSubscribe");

                        disposable = d;

                    }

                    @Override

                    public void onNext(User user) {

                        Log.e(TAG, "onNext: " + user.getName() + ", " + user.getGender() + ", " + user.getAddress().getAddress());

                    }

                    @Override

                    public void onError(Throwable e) {

                    }

                    @Override

                    public void onComplete() {

                        Log.e(TAG, "All users emitted!");

                    }

                });

    }

    /\*\*

     \* Assume this as a network call

     \* returns Users with address filed added

     \*/

    private Observable<User> getAddressObservable(final User user) {

        final String[] addresses = new String[]{

                "1600 Amphitheatre Parkway, Mountain View, CA 94043",

                "2300 Traverwood Dr. Ann Arbor, MI 48105",

                "500 W 2nd St Suite 2900 Austin, TX 78701",

                "355 Main Street Cambridge, MA 02142"

        };

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        Address address = new Address();

                        address.setAddress(addresses[new Random().nextInt(2) + 0]);

                        if (!emitter.isDisposed()) {

                            user.setAddress(address);

                            // Generate network latency of random duration

                            int sleepTime = new Random().nextInt(1000) + 500;

                            Thread.sleep(sleepTime);

                            emitter.onNext(user);

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

    /\*\*

     \* Assume this is a network call to fetch users

     \* returns Users with name and gender but missing address

     \*/

    private Observable<User> getUsersObservable() {

        String[] maleUsers = new String[]{"Mark", "John", "Trump", "Obama"};

        final List<User> users = new ArrayList<>();

        for (String name : maleUsers) {

            User user = new User();

            user.setName(name);

            user.setGender("male");

            users.add(user);

        }

        return Observable

                .create(new ObservableOnSubscribe<User>() {

                    @Override

                    public void subscribe(ObservableEmitter<User> emitter) throws Exception {

                        for (User user : users) {

                            if (!emitter.isDisposed()) {

                                emitter.onNext(user);

                            }

                        }

                        if (!emitter.isDisposed()) {

                            emitter.onComplete();

                        }

                    }

                }).subscribeOn(Schedulers.io());

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        disposable.dispose();

    }

}

[Debounce](http://reactivex.io/documentation/operators/debounce.html) operators emits items only when a specified timespan is passed. This operator is very useful when the Observable is rapidly emitting items but you are only interested in receiving them in timely manner.

public class DebounceOperatorActivity extends AppCompatActivity {

    private static final String TAG = DebounceOperatorActivity.class.getSimpleName();

    private CompositeDisposable disposable = new CompositeDisposable();

    private Unbinder unbinder;

    @BindView(R.id.input\_search)

    EditText inputSearch;

    @BindView(R.id.txt\_search\_string)

    TextView txtSearchString;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_debounce\_operator);

        unbinder = ButterKnife.bind(this);

        disposable.add(

                RxTextView.textChangeEvents(inputSearch)

                        .skipInitialValue()

                        .debounce(300, TimeUnit.MILLISECONDS)

                        .subscribeOn(Schedulers.io())

                        .observeOn(AndroidSchedulers.mainThread())

                        .subscribeWith(searchQuery()));

        txtSearchString.setText("Search query will be accumulated every 300 milli sec");

    }

    private DisposableObserver<TextViewTextChangeEvent> searchQuery() {

        return new DisposableObserver<TextViewTextChangeEvent>() {

            @Override

            public void onNext(TextViewTextChangeEvent textViewTextChangeEvent) {

                Log.d(TAG, "search string: " + textViewTextChangeEvent.text().toString());

                txtSearchString.setText("Query: " + textViewTextChangeEvent.text().toString());

            }

            @Override

            public void onError(Throwable e) {

            }

            @Override

            public void onComplete() {

            }

        };

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        unbinder.unbind();

        disposable.clear();

    }

}

**Delay()** operator delays the start of emission of values from the Observable by a certain time.

Observable.just(1,2,3,4,5,6) .delay(5, TimeUnit.SECONDS) .subscribe(System.out::println, System.out::println, () -> System.out.print("OnComplete")); Thread.sleep(4000);