You can implement your own Observable operators. This page shows you how.

If your operator is designed to *originate* an Observable, rather than to transform or react to a source Observable, use the `create()` method rather than trying to implement `Observable` manually. Otherwise, you can create a custom operator by following the instructions on this page.

If your operator is designed to act on the individual items emitted by a source Observable, follow the instructions under *Sequence Operators* below. If your operator is designed to transform the source Observable as a whole (for instance, by applying a particular set of existing RxJava operators to it) follow the instructions under *Transformational Operators* below.

(**Note:** in Xtend, a Groovy-like language, you can implement your operators as *extension methods* and can thereby chain them directly without using the methods described on this page. See RxJava and Xtend for details.)

Sequence Operators

The following example shows how you can use the `lift()` operator to chain your custom operator (in this example: `myOperator`) alongside standard RxJava operators like `ofType` and `map`:

```
fooObservable = barObservable.ofType(Integer).map({it*2}).lift(new myOperator<T>
()).map({"transformed by myOperator: " + it});
```

The following section shows how you form the scaffolding of your operator so that it will work correctly with `lift()`.

Implementing Your Operator

Define your operator as a public class that implements the `Operator` interface, like so:

```
public class myOperator<T> implements Operator<T> {
  public myOperator( /* any necessary params here */ ) {
    /* any necessary initialization here */
  }

  @Override
  public Subscriber<? super T> call(final Subscriber<? super T> s) {
    return new Subscriber<t>(s) {
      @Override
      public void onCompleted() {
        /* add your own onCompleted behavior here, or just pass the completed
notification through: */
        if(!s.isUnsubscribed()) {
          s.onCompleted();
        }
      }

      @Override
      public void onError(Throwable t) {
        /* add your own onError behavior here, or just pass the error notification
through: */
```

```
        if(!s.isUnsubscribed()) {
          s.onError(t);
        }
      }

      @Override
      public void onNext(T item) {
        /* this example performs some sort of operation on each incoming item and
emits the results */
        if(!s.isUnsubscribed()) {
          transformedItem = myOperatorTransformOperation(item);
          s.onNext(transformedItem);
        }
      }
    };
  }
}
```

Transformational Operators

The following example shows how you can use the `compose()` operator to chain your custom operator (in this example, an operator called `myTransformer` that transforms an Observable that emits Integers into one that emits Strings) alongside standard RxJava operators like `ofType` and `map`:

```
fooObservable = barObservable.ofType(Integer).map({it*2}).compose(new
myTransformer<Integer,String>()).map({"transformed by myOperator: " + it});
```

The following section shows how you form the scaffolding of your operator so that it will work correctly with `compose()`.

Implementing Your Transformer

Define your transforming function as a public class that implements the `Transformer` interface, like so:

```
public class myTransformer<Integer,String> implements Transformer<Integer,String> {
  public myTransformer( /* any necessary params here */ ) {
    /* any necessary initialization here */
  }

  @Override
  public Observable<String> call(Observable<Integer> source) {
    /*
     * this simple example Transformer applies map() to the source Observable
     * in order to transform the "source" observable from one that emits
     * integers to one that emits string representations of those integers.
     */
    return source.map( new Func1<Integer,String>() {
      @Override
      public String call(Integer t1) {
```

```
            return String.valueOf(t1);
        }
    } );
  }
}
```

See also

- "Don't break the chain: use RxJava's compose() operator" by Dan Lew

Other Considerations

- Your sequence operator may want to check its Subscriber's `isUnsubscribed()` status before it emits any item to (or sends any notification to) the Subscriber. There's no need to waste time generating items that no Subscriber is interested in seeing.
- Take care that your sequence operator obeys the core tenets of the Observable contract:
  - It may call a Subscriber's `onNext()` method any number of times, but these calls must be non-overlapping.
  - It may call either a Subscriber's `onCompleted()` or `onError()` method, but not both, exactly once, and it may not subsequently call a Subscriber's `onNext()` method.
  - If you are unable to guarantee that your operator conforms to the above two tenets, you can add the `serialize()` operator to it, which will force the correct behavior.
- Keep an eye on Issue #1962 — there are plans to create a test scaffold that you can use to write tests which verify that your new operator conforms to the Observable contract.
- Do not block within your operator.
- When possible, you should compose new operators by combining existing operators, rather than implementing them with new code. RxJava itself does this with some of its standard operators, for example:
  - `first()` is defined as `take(1).single()`
  - `ignoreElements()` is defined as `filter(alwaysFalse())`
  - `reduce(a)` is defined as `scan(a).last()`
- If your operator uses functions or lambdas that are passed in as parameters (predicates, for instance), note that these may be sources of exceptions, and be prepared to catch these and notify subscribers via `onError()` calls.
  - Some exceptions are considered "fatal" and for them there's no point in trying to call `onError()` because that will either be futile or will just compound the problem. You can use the `Exceptions.throwIfFatal(throwable)` method to filter out such fatal exceptions and rethrow them rather than try to notify about them.
- In general, notify subscribers of error conditions immediately, rather than making an effort to emit more items first.
- Be aware that "`null`" is a valid item that may be emitted by an Observable. A frequent source of bugs is to test some variable meant to hold an emitted item against `null` as a substitute for testing whether or not an item was emitted. An emission of "`null`" is still an emission and is not the same as not emitting anything.
- It can be tricky to make your operator behave well in *backpressure* scenarios. See Advanced RxJava, a blog from Dávid Karnok, for a good discussion of the factors at play and how to deal with them.