This page shows methods that create reactive sources, such as `Observable` s.

Outline

- `create`
- `defer`
- `empty`
- `error`
- `from`
- `generate`
- `interval`
- `just`
- `never`
- `range`
- `timer`

just

Available in: ✓ `Flowable` , ✓ `Observable` , ✓ `Maybe` , ✓ `Single` , ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/just.html

Constructs a reactive type by taking a pre-existing object and emitting that specific object to the downstream consumer upon subscription.

just example:

```
String greeting = "Hello world!";

Observable<String> observable = Observable.just(greeting);

observable.subscribe(item -> System.out.println(item));
```

There exist overloads with 2 to 9 arguments for convenience, which objects (with the same common type) will be emitted in the order they are specified.

```
Observable<Object> observable = Observable.just("1", "A", "3.2", "def");

  observable.subscribe(item -> System.out.print(item), error ->
error.printStackTrace(),
                () -> System.out.println());
```

From

Constructs a sequence from a pre-existing source or generator type.

Note: These static methods use the postfix naming convention (i.e., the argument type is repeated in the method name) to avoid overload resolution ambiguities.

ReactiveX documentation: http://reactivex.io/documentation/operators/from.html

fromIterable

Available in: ✓ `Flowable`, ✓ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Signals the items from a `java.lang.Iterable` source (such as `List`s, `Set`s or `Collection`s or custom `Iterable`s) and then completes the sequence.

fromIterable example:

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8));

Observable<Integer> observable = Observable.fromIterable(list);

observable.subscribe(item -> System.out.println(item), error ->
error.printStackTrace(),
      () -> System.out.println("Done"));
```

fromArray

Available in: ✓ `Flowable`, ✓ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

Signals the elements of the given array and then completes the sequence.

fromArray example:

```
Integer[] array = new Integer[10];
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}

Observable<Integer> observable = Observable.fromArray(array);

observable.subscribe(item -> System.out.println(item), error ->
error.printStackTrace(),
      () -> System.out.println("Done"));
```

Note: RxJava does not support primitive arrays, only (generic) reference arrays.

fromCallable

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

When a consumer subscribes, the given `java.util.concurrent.Callable` is invoked and its returned value (or thrown exception) is relayed to that consumer.

fromCallable example:

```
Callable<String> callable = () -> {
    System.out.println("Hello World!");
    return "Hello World!");
}

Observable<String> observable = Observable.fromCallable(callable);

observable.subscribe(item -> System.out.println(item), error ->
error.printStackTrace(),
    () -> System.out.println("Done"));
```

Remark: In `Completable`, the actual returned value is ignored and the `Completable` simply completes.

fromAction

Available in: ○ `Flowable`, ○ `Observable`, ✓ `Maybe`, ○ `Single`, ✓ `Completable`

When a consumer subscribes, the given `io.reactivex.function.Action` is invoked and the consumer completes or receives the exception the `Action` threw.

fromAction example:

```
Action action = () -> System.out.println("Hello World!");

Completable completable = Completable.fromAction(action);

completable.subscribe(() -> System.out.println("Done"), error ->
error.printStackTrace());
```

Note: the difference between `fromAction` and `fromRunnable` is that the `Action` interface allows throwing a checked exception while the `java.lang.Runnable` does not.

fromRunnable

Available in: ○ `Flowable`, ○ `Observable`, ✓ `Maybe`, ○ `Single`, ✓ `Completable`

When a consumer subscribes, the given `io.reactivex.function.Action` is invoked and the consumer completes or receives the exception the `Action` threw.

fromRunnable example:

```
Runnable runnable = () -> System.out.println("Hello World!");

Completable completable = Completable.fromRunnable(runnable);
```

```
completable.subscribe(() -> System.out.println("Done"), error ->
error.printStackTrace());
```

Note: the difference between `fromAction` and `fromRunnable` is that the `Action` interface allows throwing a checked exception while the `java.lang.Runnable` does not.

fromFuture

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

Given a pre-existing, already running or already completed `java.util.concurrent.Future`, wait for the `Future` to complete normally or with an exception in a blocking fashion and relay the produced value or exception to the consumers.

fromFuture example:

```
ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

Future<String> future = executor.schedule(() -> "Hello world!", 1,
TimeUnit.SECONDS);

Observable<String> observable = Observable.fromFuture(future);

observable.subscribe(
    item -> System.out.println(item),
    error -> error.printStackTrace(),
    () -> System.out.println("Done"));

executor.shutdown();
```

from{reactive type}

Wraps or converts another reactive type to the target reactive type.

The following combinations are available in the various reactive types with the following signature pattern: `targetType.from{sourceType}()`

Available in:

targetType \ sourceType	Publisher	Observable	Maybe	Single	Completable
Flowable	✓				
Observable	✓				
Maybe					

				✓	✓
Single	✓	✓			
Completable	✓	✓	✓	✓	

*Note: not all possible conversion is implemented via the `from{reactive type}` method families. Check out the `to{reactive type}` method families for further conversion possibilities.

from{reactive type} example:

```
Flux<Integer> reactorFlux = Flux.fromCompletionStage(CompletableFuture.
<Integer>completedFuture(1));

Observable<Integer> observable = Observable.fromPublisher(reactorFlux);

observable.subscribe(
    item -> System.out.println(item),
    error -> error.printStackTrace(),
    () -> System.out.println("Done"));
```

generate

Available in: ✓ `Flowable`, ✓ `Observable`, ○ `Maybe`, ○ `Single`, ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/create.html

Creates a cold, synchronous and stateful generator of values.

generate example:

```
int startValue = 1;
int incrementValue = 1;
Flowable<Integer> flowable = Flowable.generate(() -> startValue, (s, emitter) -> {
    int nextValue = s + incrementValue;
    emitter.onNext(nextValue);
    return nextValue;
});
flowable.subscribe(value -> System.out.println(value));
```

create

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/create.html

Construct a **safe** reactive type instance which when subscribed to by a consumer, runs an user-provided function and provides a type-specific `Emitter` for this function to generate the signal(s) the designated business logic requires. This method allows bridging the non-reactive, usually listener/callback-style world, with the reactive world.

create example:

```
ScheduledExecutorService executor = Executors.newSingleThreadedScheduledExecutor();

ObservableOnSubscribe<String> handler = emitter -> {

     Future<Object> future = executor.schedule(() -> {
          emitter.onNext("Hello");
          emitter.onNext("World");
          emitter.onComplete();
          return null;
     }, 1, TimeUnit.SECONDS);

     emitter.setCancellable(() -> future.cancel(false));
};

Observable<String> observable = Observable.create(handler);

observable.subscribe(item -> System.out.println(item), error ->
error.printStackTrace(),
     () -> System.out.println("Done"));

Thread.sleep(2000);
executor.shutdown();
```

Note: `Flowable.create()` must also specify the backpressure behavior to be applied when the user-provided function generates more items than the downstream consumer has requested.

defer

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/defer.html

Calls an user-provided `java.util.concurrent.Callable` when a consumer subscribes to the reactive type so that the `Callable` can generate the actual reactive instance to relay signals from towards the consumer. `defer` allows:

- associating a per-consumer state with such generated reactive instances,
- allows executing side-effects before an actual/generated reactive instance gets subscribed to,

- turn hot sources (i.e., `Subject` s and `Processor` s) into cold sources by basically making those hot sources not exist until a consumer subscribes.

defer example:

```
Observable<Long> observable = Observable.defer(() -> {
    long time = System.currentTimeMillis();
    return Observable.just(time);
});

observable.subscribe(time -> System.out.println(time));

Thread.sleep(1000);

observable.subscribe(time -> System.out.println(time));
```

range

Available in: ✓ `Flowable` , ✓ `Observable` , ○ `Maybe` , ○ `Single` , ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/range.html

Generates a sequence of values to each individual consumer. The `range()` method generates `Integer` s, the `rangeLong()` generates `Long` s.

range example:

```
String greeting = "Hello World!";

Observable<Integer> indexes = Observable.range(0, greeting.length());

Observable<Character> characters = indexes
    .map(index -> greeting.charAt(index));

characters.subscribe(character -> System.out.print(character), error ->
error.printStackTrace(),
        () -> System.out.println());
```

interval

Available in: ✓ `Flowable` , ✓ `Observable` , ○ `Maybe` , ○ `Single` , ○ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/interval.html

Periodically generates an infinite, ever increasing numbers (of type `Long`). The `intervalRange` variant generates a limited amount of such numbers.

interval example:

```
Observable<Long> clock = Observable.interval(1, TimeUnit.SECONDS);

clock.subscribe(time -> {
    if (time % 2 == 0) {
        System.out.println("Tick");
    } else {
        System.out.println("Tock");
    }
});
```

timer

Available in: ✓ `Flowable` , ✓ `Observable` , ✓ `Maybe` , ✓ `Single` , ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/timer.html

After the specified time, this reactive source signals a single `0L` (then completes for `Flowable` and `Observable`).

timer example:

```
Observable<Long> eggTimer = Observable.timer(5, TimeUnit.MINUTES);

eggTimer.blockingSubscribe(v -> System.out.println("Egg is ready!"));
```

empty

Available in: ✓ `Flowable` , ✓ `Observable` , ✓ `Maybe` , ○ `Single` , ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/empty-never-throw.html

This type of source signals completion immediately upon subscription.

empty example:

```
Observable<String> empty = Observable.empty();

empty.subscribe(
    v -> System.out.println("This should never be printed!"),
    error -> System.out.println("Or this!"),
    () -> System.out.println("Done will be printed."));
```

never

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/empty-never-throw.html

This type of source does not signal any `onNext`, `onSuccess`, `onError` or `onComplete`. This type of reactive source is useful in testing or "disabling" certain sources in combinator operators.

never example:

```
Observable<String> never = Observable.never();

never.subscribe(
    v -> System.out.println("This should never be printed!"),
    error -> System.out.println("Or this!"),
    () -> System.out.println("This neither!"));
```

error

Available in: ✓ `Flowable`, ✓ `Observable`, ✓ `Maybe`, ✓ `Single`, ✓ `Completable`

ReactiveX documentation: http://reactivex.io/documentation/operators/empty-never-throw.html

Signal an error, either pre-existing or generated via a `java.util.concurrent.Callable`, to the consumer.

error example:

```
Observable<String> error = Observable.error(new IOException());

error.subscribe(
    v -> System.out.println("This should never be printed!"),
    e -> e.printStackTrace(),
    () -> System.out.println("This neither!"));
```

A typical use case is to conditionally map or suppress an exception in a chain utilizing `onErrorResumeNext`:

```
Observable<String> observable = Observable.fromCallable(() -> {
    if (Math.random() < 0.5) {
        throw new IOException();
    }
    throw new IllegalArgumentException();
});

Observable<String> result = observable.onErrorResumeNext(error -> {
    if (error instanceof IllegalArgumentException) {
        return Observable.empty();
    }
    return Observable.error(error);
});

for (int i = 0; i < 10; i++) {
    result.subscribe(
        v -> System.out.println("This should never be printed!"),
        error -> error.printStackTrace(),
        () -> System.out.println("Done"));
}
```