There are a variety of operators that you can use to react to or recover from `onError` notifications from reactive sources, such as `Observable`s. For example, you might:

1. swallow the error and switch over to a backup Observable to continue the sequence
2. swallow the error and emit a default item
3. swallow the error and immediately try to restart the failed Observable
4. swallow the error and try to restart the failed Observable after some back-off interval

Outline

- `doOnError`
- `onErrorComplete`
- `onErrorResumeNext`
- `onErrorReturn`
- `onErrorReturnItem`
- `onExceptionResumeNext`
- `retry`
- `retryUntil`
- `retryWhen`

doOnError

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

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

Instructs a reactive type to invoke the given `io.reactivex.functions.Consumer` when it encounters an error.

doOnError example

```
Observable.error(new IOException("Something went wrong"))
    .doOnError(error -> System.err.println("The error message is: " + error.getMessage()))
    .subscribe(
        x -> System.out.println("onNext should never be printed!"),
        Throwable::printStackTrace,
        () -> System.out.println("onComplete should never be printed!"));
```

onErrorComplete

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

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

Instructs a reactive type to swallow an error event and replace it by a completion event.

Optionally, a `io.reactivex.functions.Predicate` can be specified that gives more control over when an error event should be replaced by a completion event, and when not.

onErrorComplete example

```
Completable.fromAction(() -> {
    throw new IOException();
}).onErrorComplete(error -> {
    // Only ignore errors of type java.io.IOException.
    return error instanceof IOException;
}).subscribe(
    () -> System.out.println("IOException was ignored"),
    error -> System.err.println("onError should not be printed!"));
```

onErrorResumeNext

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

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

Instructs a reactive type to emit a sequence of items if it encounters an error.

onErrorResumeNext example

```
 Observable<Integer> numbers = Observable.generate(() -> 1, (state, emitter) -> {
    emitter.onNext(state);

    return state + 1;
});

numbers.scan(Math::multiplyExact)
    .onErrorResumeNext(Observable.empty())
    .subscribe(
        System.out::println,
```

```
        error -> System.err.println("onError should not be printed!"));

// prints:
// 1
// 2
// 6
// 24
// 120
// 720
// 5040
// 40320
// 362880
// 3628800
// 39916800
// 479001600
```

onErrorReturn

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, ⭕ `Completable`

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

Instructs a reactive type to emit the item returned by the specified `io.reactivex.functions.Function` when it encounters an error.

onErrorReturn example

```
Single.just("2A")
    .map(v -> Integer.parseInt(v, 10))
    .onErrorReturn(error -> {
        if (error instanceof NumberFormatException) return 0;
        else throw new IllegalArgumentException();
    })
    .subscribe(
        System.out::println,
        error -> System.err.println("onError should not be printed!"));

// prints 0
```

onErrorReturnItem

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ✅ `Single`, ⭕ `Completable`

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

Instructs a reactive type to emit a particular item when it encounters an error.

onErrorReturnItem example

```
Single.just("2A")
    .map(v -> Integer.parseInt(v, 10))
    .onErrorReturnItem(0)
    .subscribe(
        System.out::println,
        error -> System.err.println("onError should not be printed!"));

// prints 0
```

onExceptionResumeNext

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ⭕ `Single`, ⭕ `Completable`

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

Instructs a reactive type to continue emitting items after it encounters an `java.lang.Exception`. Unlike `onErrorResumeNext`, this one lets other types of `Throwable` continue through.

onExceptionResumeNext example

```
Observable<String> exception = Observable.<String>error(IOException::new)
    .onExceptionResumeNext(Observable.just("This value will be used to recover from the IOEx

Observable<String> error = Observable.<String>error(Error::new)
    .onExceptionResumeNext(Observable.just("This value will not be used"));

Observable.concat(exception, error)
    .subscribe(
        message -> System.out.println("onNext: " + message),
        err -> System.err.println("onError: " + err));

// prints:
// onNext: This value will be used to recover from the IOException
// onError: java.lang.Error
```

retry

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

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

Instructs a reactive type to resubscribe to the source reactive type if it encounters an error in the hopes that it will complete without error.

retry example

```
Observable<Long> source = Observable.interval(0, 1, TimeUnit.SECONDS)
    .flatMap(x -> {
        if (x >= 2) return Observable.error(new IOException("Something went wrong!"));
        else return Observable.just(x);
    });

source.retry((retryCount, error) -> retryCount < 3)
    .blockingSubscribe(
        x -> System.out.println("onNext: " + x),
        error -> System.err.println("onError: " + error.getMessage()));

// prints:
// onNext: 0
// onNext: 1
// onNext: 0
// onNext: 1
// onNext: 0
// onNext: 1
// onError: Something went wrong!
```

retryUntil

Available in: ✅ `Flowable`, ✅ `Observable`, ✅ `Maybe`, ⭕ `Single`, ⭕ `Completable`

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

Instructs a reactive type to resubscribe to the source reactive type if it encounters an error until the given `io.reactivex.functions.BooleanSupplier` returns `true`.

retryUntil example

```
LongAdder errorCounter = new LongAdder();
Observable<Long> source = Observable.interval(0, 1, TimeUnit.SECONDS)
    .flatMap(x -> {
        if (x >= 2) return Observable.error(new IOException("Something went wrong!"));
        else return Observable.just(x);
    })
    .doOnError((error) -> errorCounter.increment());

source.retryUntil(() -> errorCounter.intValue() >= 3)
    .blockingSubscribe(
        x -> System.out.println("onNext: " + x),
        error -> System.err.println("onError: " + error.getMessage()));

// prints:
// onNext: 0
// onNext: 1
// onNext: 0
// onNext: 1
// onNext: 0
// onNext: 1
// onError: Something went wrong!
```

retryWhen

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

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

Instructs a reactive type to pass any error to another `Observable` or `Flowable` to determine whether to resubscribe to the source.

retryWhen example

```
Observable<Long> source = Observable.interval(0, 1, TimeUnit.SECONDS)
    .flatMap(x -> {
        if (x >= 2) return Observable.error(new IOException("Something went wrong!"));
        else return Observable.just(x);
    });

source.retryWhen(errors -> {
    return errors.map(error -> 1)

    // Count the number of errors.
```

```java
    .scan(Math::addExact)

    .doOnNext(errorCount -> System.out.println("No. of errors: " + errorCount))

    // Limit the maximum number of retries.
    .takeWhile(errorCount -> errorCount < 3)

    // Signal resubscribe event after some delay.
    .flatMapSingle(errorCount -> Single.timer(errorCount, TimeUnit.SECONDS));
}).blockingSubscribe(
    x -> System.out.println("onNext: " + x),
    Throwable::printStackTrace,
    () -> System.out.println("onComplete"));

// prints:
// onNext: 0
// onNext: 1
// No. of errors: 1
// onNext: 0
// onNext: 1
// No. of errors: 2
// onNext: 0
// onNext: 1
// No. of errors: 3
// onComplete
```