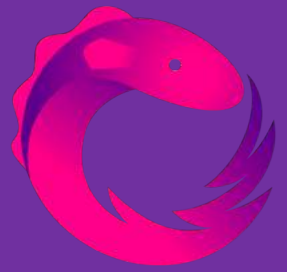
RxJava - 리액티브 프로그래밍 기초 스터디

6주차 – 디버깅과 예외 처리

김대윤



디버깅

```
Observable<Integer> observable = Observable.create();
```

Upstream과 Downstream이 동일한 문장으로 이루어져 있음.

=> 예외처리 어떻게?

Observable

- onNext() : 데이터 발행할 때
- onError() : 중간에 에러가 발생할 때
- onComplete() : 모든 데이터의 발행을 마쳤을 때

doOnNext(), doOnComplete(), doOnError() : Observable의 알림 이벤트 함수들

```
String[] orgs = {"1", "3", "5"};
Observable<String> source = Observable.fromArray(orgs);

source.doOnNext(data-> Log.d( tag: "onNex()",data))
        .doOnComplete(()->Log.d( tag: "onComplete()",  msg: ""))
        .doOnError(e -> Log.e( tag: "onError()",e.getMessage()))
        .subscribe();
```

```
09-08 17:18:22.558 4221-4221/jeongari.com.rxjava2 D/onNext(): 1
    3
    5
09-08 17:18:22.558 4221-4221/jeongari.com.rxjava2 D/onComplete(): onComplete()
```


```
Integer[] divider = {10, 5, 0};

Observable.fromArray(divider)
        .map(div -> 1000/div)
        .doOnNext(data -> Log.d( tag: "onNext()", data.toString()))
        .doOnComplete(() -> Log.d( tag: "onComplete", msg: "onComplete()"))
        .doOnError(e -> Log.e( tag: "onError()", e.getMessage()))
        .subscribe();
```

```
09-08 17:18:22.560 4221-4221/jeongari.com.rxjava2 D/onNext(): 100
    200
09-08 17:18:22.560 4221-4221/jeongari.com.rxjava2 E/onError(): divide by zero
09-08 17:18:22.562 4221-4221/jeongari.com.rxjava2 W/System.err: io.reactivex.exceptions.OnErrorNotImplementedException: divide by zero
```

이런 onError() 이벤트에 대한 처리를 개발자가 해주어야 한다!

onNext() : 데이터 발행할 때

onError() : 중간에 에러가 발생할 때

onComplete() : 모든 데이터의 발행을 마쳤을 때

처리를 한번에 하고 싶다면? doOnEach()

doOnEach( ) — register an action to take whenever an Observable emits an item

```
public final Observable<T> doOnEach(
  final Consumer<? super Notification<T>> onNotification)
public final Observable<T> doOnEach(final Observer<? super T> observer)
```

객체를 받아와서
이벤트 별로 구별하여 처리

```
String[] data = {"ONE", "TWO", "THREE"};
Observable<String> source = Observable.fromArray(data);

source.doOnEach(noti -> {
    if (noti.isOnNext()) Log.d("onNext()", noti.getValue());
    if (noti.isOnComplete()) Log.d("onComplete()");
    if (noti.isOnError()) Log.e("onError()", noti.getError().getMessage());
})

  .subscribe(System.out::println);
```

Notification<T> 객체의 함수들

```
String[] orgs = {"1", "3", "5"};
Observable<String> source = Observable.fromArray(orgs);

source.doOnEach(new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
        // doOnEach()에서는 onSubscribe() 함수가 호출되지 않습니다.
    }

    @Override
    public void onNext(String value) {
        Log.d("onNext()", value);
    }

    @Override
    public void onError(Throwable e) {
        Log.e("onError()", e.getMessage());
    }

    @Override
    public void onComplete() {
        Log.d("onComplete()");
    }
});

  .subscribe(Log::i);
```

Notification<T> 객체를 통해 처리하는 것이 일반적이나

Observer 인터페이스를 통해 처리할 수 도 있음!

보시다시피 간결하지 못하다는 단점이.. 소곤소곤...

onNext() : 데이터 발행할 때

onError() : 중간에 에러가 발생할 때

onComplete() : 모든 데이터의 발행을 마쳤을 때

얘네 말고도 사용하는 알림 이벤트들이 있어요!

doOnSubscribe() Observable을 구독했을 때

구독의 결과로 나오는 Disposable 객체

```
public final Observable<T> doOnSubscribe(Consumer<? super Disposable> onSubscribe)
```

doOnDispose() Observable을 구독을 해지했을 때

```
public final Observable<T> doOnDispose(Action onDispose)
```

구독 해지 시에 할 액션, 스레드 안전하게 동작해야한다!

```
String[] orgs = {"1", "3", "5", "2", "6"};
Observable<String> source = Observable.fromArray(orgs)
    .zipWith(Observable.interval(100L, TimeUnit.MILLISECONDS), (a, b) -> a)
    .doOnSubscribe(d -> Log.d("onSubscribe()"))
    .doOnDispose(() -> Log.d("onDispose()"));
 Disposable d = source.subscribe(Log::i);
```

Interval() 과 합성된 zipWith()
=> 계산 스케줄러에서 동작

```
CommonUtils.sleep(200);
d.dispose();
CommonUtils.sleep(300);
```

메인 스레드에서 200ms 후에 Observable 구독 해지

```
main ¦ debug = onSubscribe()
RxComputationThreadPool-1 ¦ value = 1
RxComputationThreadPool-1 ¦ value = 3
main ¦ debug = onDispose()
```

doOnSubscribe() Observable을 구독했을 때

```
public final Observable<T> doOnSubscribe(Consumer<? super Disposable> onSubscribe)
```

doOnDispose() Observable을 구독을 해지했을 때

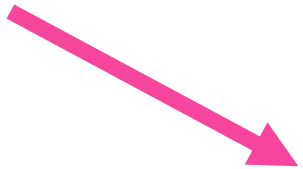
```
public final Observable<T> doOnDispose(Action onDispose)
```

처리를 한번에 하고 싶다면? doOnLifeCycle()

doOnSubscribe()

doOnDispose()

```
String[] orgs = {"1", "3", "5", "2", "6"};
Observable<String> source = Observable.fromArray(orgs)
    .zipWith(Observable.interval(100L, TimeUnit.MILLISECONDS), (a, b) -> a)
    .doOnSubscribe(d -> Log.d("onSubscribe()"))
    .doOnDispose(() -> Log.d("onDispose()"));
 Disposable d = source.subscribe(Log::i);

 CommonUtils.sleep(200);
 d.dispose();
 CommonUtils.sleep(300);
```



doOnLifeCycle()

```
String[] orgs = {"1", "3", "5", "2", "6"};
Observable<String> source = Observable.fromArray(orgs)
    .zipWith(Observable.interval(100L, TimeUnit.MILLISECONDS), (a, b) -> a)
    .doOnLifecycle(
      d -> Log.d("onSubscribe()"), () -> Log.d("onDispose()"));
Disposable d = source.subscribe(Log::i);

CommonUtils.sleep(200);
d.dispose();
CommonUtils.sleep(300);
```

onError() : 중간에 에러가 발생할 때

onComplete() : 모든 데이터의 발행을 마쳤을 때

의 호출 직전에 호출되는

doOnTerminate()

```
String[] orgs = {"1", "3", "5"};
Observable<String> source = Observable.fromArray(orgs);

source.doOnTerminate(() -> Log.d("onTerminate()"))
   .doOnComplete(() -> Log.d("onComplete()"))
   .doOnError(e -> Log.e("onError()", e.getMessage()))
   .subscribe(Log::i);
```

```
main ¦ value = 1
main ¦ value = 3
main ¦ value = 5
main ¦ debug = onTerminate()
main ¦ debug = onComplete()
```

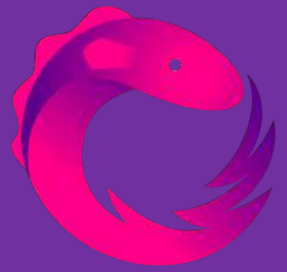
넘 복잡 하구 많잖아 (깊은 빡침...)

정리 해봤음 !

onNext 이벤트
onError 이벤트
onComplete 이벤트
→ doOnEach( )

onSubscribe 이벤트
onDispose 이벤트
→ doOnLifecycle( )

doOnEach( ), doOnTerminate( ), doOnLifecycle( ) → doFinally( )



예외 처리

RxJava에서는 에러 또한 데이터라고 간주할 수 있다!

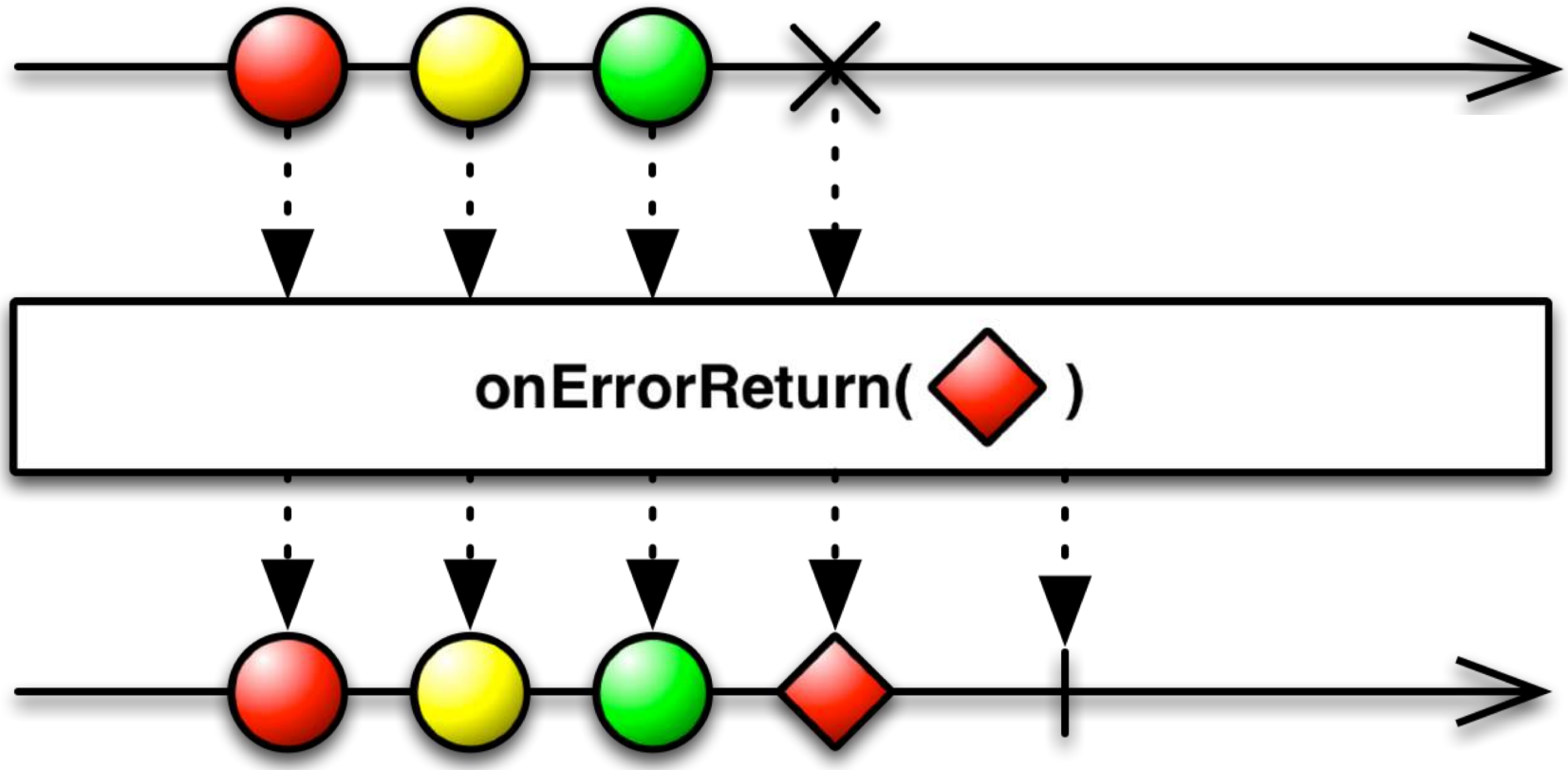
발행 데이터 1
발행 데이터 2
발행 데이터 3
...

에러가 있다면? 에러 로그로 대체!

onErrorReturn()

** OOM 같은 긴박한 상황 빼고는 onError 이벤트를 발생시키지 않는다.

데이터 흐름이 끊겨 버리기 때문!


onErrorReturn( )

```
String[] grades = {"70", "88", "$100", "93", "83"};  // $100이 에러 데이터

Observable<Integer> source = Observable.fromArray(grades)
  .map(data -> Integer.parseInt(data))
  .onErrorReturn(e -> {
      if(e instanceof NumberFormatException) {
          e.printStackTrace();
      }
      return -1;
  });

source.subscribe(data -> {
    if (data < 0) {
        Log.e("Wrong Data found!!");

        return;
    }

    Log.i("Grade is " + data);
});
```

NumberFormatException에 대한 처리가 필요!

** onErrorReturn() vs onError 이벤트 처리 비교

```
String[] grades = {"70", "88", "$100", "93", "83"};  // $100이 에러 데이터

Observable<Integer> source = Observable.fromArray(grades)
  .map(data -> Integer.parseInt(data))
  .onErrorReturn(e -> {
      if(e instanceof NumberFormatException) {
          e.printStackTrace();
      }
      return -1;
  });

source.subscribe(data -> {
    if (data < 0) {
        Log.e("Wrong Data found!!");

        return;
    }

    Log.i("Grade is " + data);
});
```

선언적 처리 가능!

onError 이벤트 사용시 선언 처리 불가

```
e -> {
    if(e instanceof NumberFormatException) {
        e.printStackTrace();
    }
    Log.e("Wrong Data found!!");
}
```

** onErrorReturn() vs onError 이벤트 처리 비교

```
String[] grades = {"70", "88", "$100", "93", "83"};  // $100이 에러 데이터

Observable<Integer> source = Observable.fromArray(grades)
  .map(data -> Integer.parseInt(data))
  .onErrorReturn(e -> {
      if(e instanceof NumberFormatException) {
          e.printStackTrace();
      }
      return -1;
  });
```

Observable이 에러 가능성을 미리 명시해주면

```
source.subscribe(data -> {
    if (data < 0) {
        Log.e("Wrong Data found!!");

        return;
    }

    Log.i("Grade is " + data);
});
```

구독자가 선언된 예외 처리 사항(onErrorReturn)을 보고

그에 맞는 처리가 가능하다!

```
String[] grades = {"70", "88", "$100", "93", "83"};  // $100이 에러 데이터

Observable<Integer> source = Observable.fromArray(grades)
   .map(data -> Integer.parseInt(data))
   .onErrorReturn(e -> {
       if(e instanceof NumberFormatException) {
           e.printStackTrace();
       }
       return -1;
   });

source.subscribe(data -> {
    if (data < 0) {
        Log.e("Wrong Data found!!");

        return;
    }

    Log.i("Grade is " + data);
});
```

onErrorReturnItem()

```
.onErrorReturnItem(-1);
```

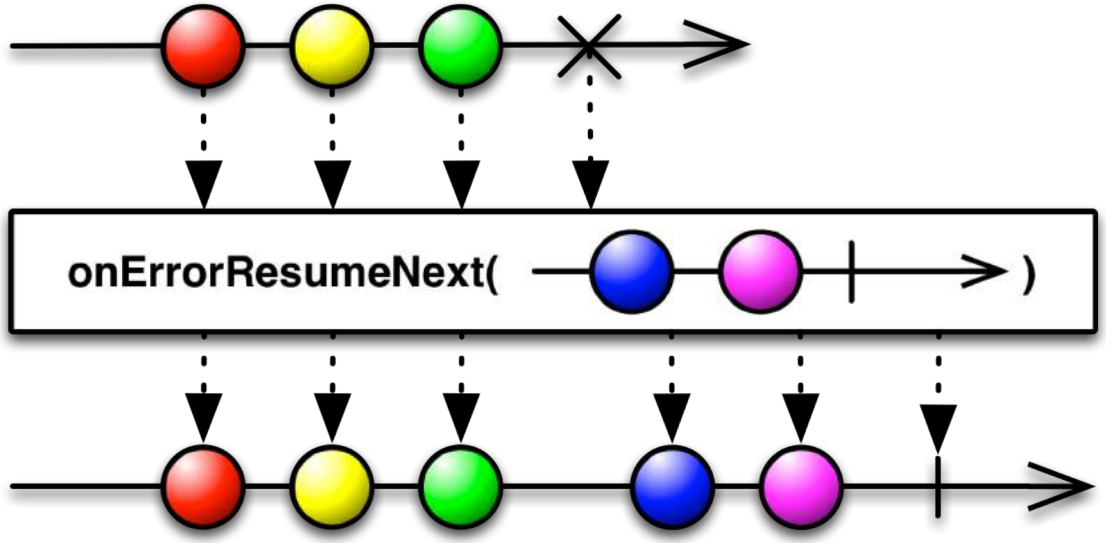
onErrorReturn()

onErrorReturnItem()

에러가 발생한 시점에서 데이터를 특정 값으로 대체하는 함수

에러를 뿜을 경우, 다른 Observable로 갈아타고 싶다면?

onErrorResumeNext()



```
String[] salesData = {"100", "200", "A300"}; // A300은 에러 데이터.
Observable<Integer> onParseError = Observable.defer(() -> {
    Log.d("send email to administrator");
    return Observable.just(-1);
}).subscribeOn(Schedulers.io());  // IO 스케줄러에서 실행됨.

Observable<Integer> source = Observable.fromArray(salesData)
  .map(Integer::parseInt)
  .onErrorResumeNext(onParseError);

source.subscribe(data -> {
    if (data < 0) {
        Log.e("Wrong Data found!!");
        return;
    }

    Log.i("Sales data : " + data);
});
```

관리자에게 이메일 보낸 후, -1 리턴!

1) 에러가 발생 하면

2) 관리자에게 이메일을 보내는 Observable인 onParseError를 구독

```
main ¦ value = Sales data : 100
main ¦ value = Sales data : 200
RxCachedThreadScheduler-1 ¦ debug = send email to administrator
RxCachedThreadScheduler-1 ¦ error = Wrong Data found!!
```

예외 처리의 다른 방법 - 재시도

retry()

함수 원형

예제에서 사용한 오버로딩 형태

```
public final Observable<T> retry()
public final Observable<T> retry(
  BiPredicate<? super Integer, ? super Throwable> predicate)
public final Observable<T> retry(long times)
public final Observable<T> retry(long times, Predicate<? super Throwable> predicate)
public final Observable<T> retry(Predicate<? super Throwable> predicate)
```

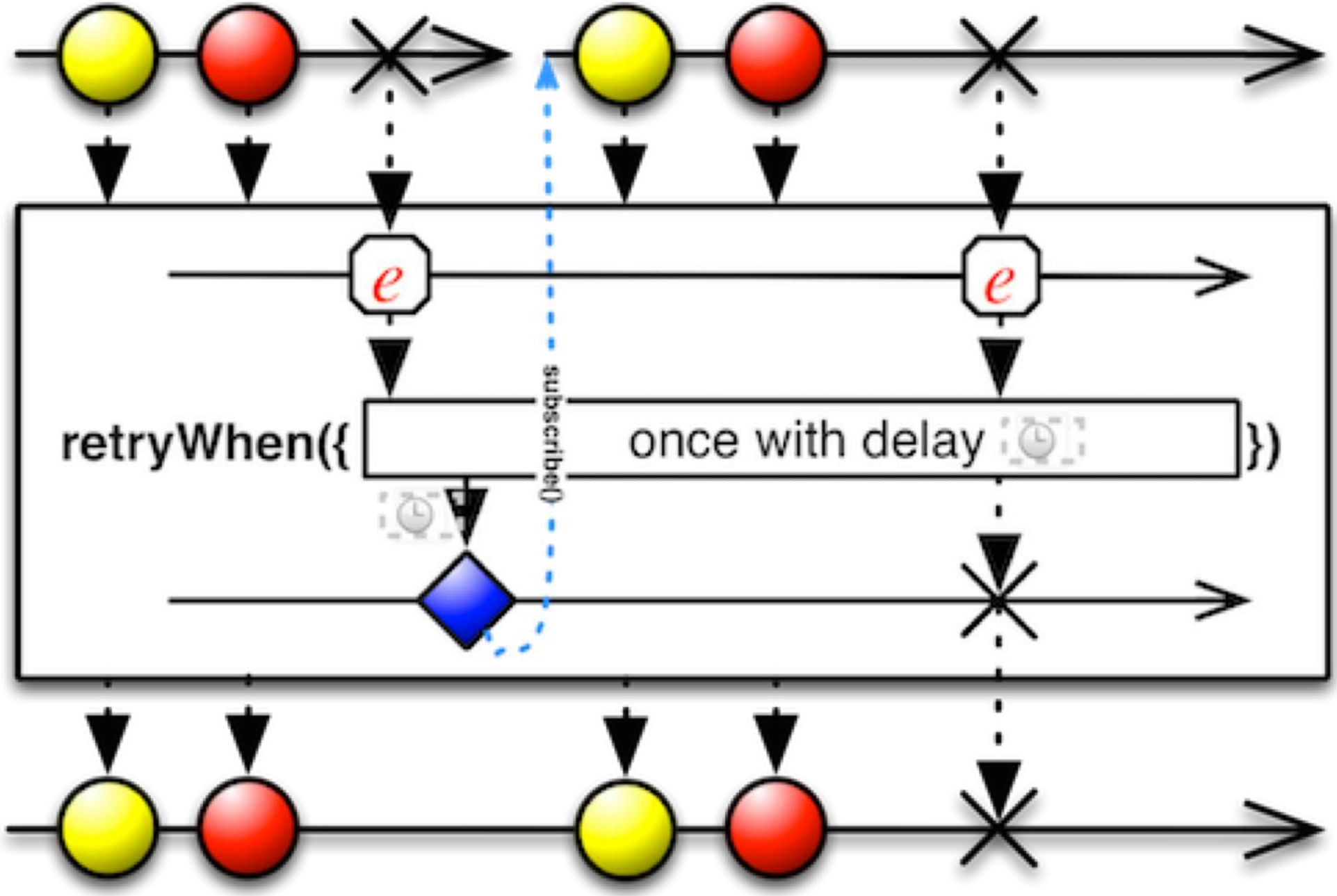
```
final int RETRY_MAX = 5;
final int RETRY_DELAY = 1000;

CommonUtils.exampleStart();

  String url = "https://api.github.com/zen";
  Observable<String> source = Observable.just(url)
    .map(OkHttpHelper::getT)
    .retry((retryCnt, e) -> {
        Log.e("retryCnt = " + retryCnt);
        CommonUtils.sleep(RETRY_DELAY);

        return retryCnt < RETRY_MAX ? true: false;
    })
    .onErrorReturn(e -> CommonUtils.ERROR_CODE);

  source.subscribe(data -> Log.it("result : " + data));
```

Github API의 호출을 재시도 하는 예제

```
main | 610 | error = api.github.com
main | error = retryCnt = 1
main | 1612 | error = api.github.com
main | error = retryCnt = 2
main | 2613 | error = api.github.com
main | error = retryCnt = 3
main | 3614 | error = api.github.com
main | error = retryCnt = 4
main | 4616 | error = api.github.com
main | error = retryCnt = 5
main | 5617 | value = result : -500
```

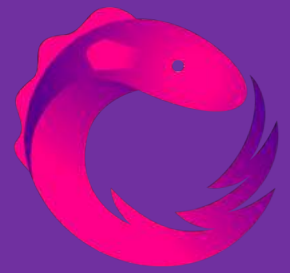
retryUntil() 특정 조건(중단할 조건)이 충족될 때 까지 계속 시도

retryWhen() 재시도 조건을 동적으로 설정 해야할 때


e
e
retryWhen({
once with delay
})
subscribe()

```
Observable.create((ObservableEmitter<String> emitter) -> {
    emitter.onError(new RuntimeException("always fails"));
}).retryWhen(attempts -> {
    return attempts.zipWith(Observable.range(1, 3), (n, i) -> i)
      .flatMap(i -> {
            Log.d("delay retry by " + i + " seconds");
            return Observable.timer(i, TimeUnit.SECONDS);
        }
     );
 }).blockingForEach(Log::d);
```

```
subscribing
delay retry by 1 second(s)
subscribing
delay retry by 2 second(s)
subscribing
delay retry by 3 second(s)
subscribing
```



흐름 제어와 Flowable 클래스

누가 설명 해주나요?