---

p.43

주요 차이점은 스트림과 시퀀스는 아이템을 당겨오지만(pull) 옵저버블은 아이템을 푸시한다는 것이다.

⇒

주요 차이점은 스트림과 시퀀스는 아이템을 풀(pull)하고, 옵저버블은 아이템을 푸시한다는 것이다.

푸시를 기반으로 하는 반복은 당겨오기에 기반한 반복보다 훨씬 강력하다.

⇒

푸시를 기반으로 하는 반복은 풀에 기반한 반복보다 훨씬 강력하다.

---

p.44

구동 시에 계산 스레드에서 배출을 발생시키기 때문에 sleep() 메서드를 생성해야 한다.

⇒

구동 시에 계산 스레드(Computation Thread)에서 배출을 발생시키기 때문에 sleep() 메서드를 사용해야 한다.

---

p.50

옵저버블 팩토리 메서드

⇒

옵저버블을 생성하는 팩토리 메서드

옵저버에게 만드는 몇 가지 방법을 다뤄볼 예정이지만

⇒

옵저버블을 만드는 방법도 다룰 것이다.

---

p.51

옵저버는 다음 코드 스니펫과 같이 각 아이템을 인쇄한다.

⇒

다음 코드 스니펫에서 옵저버는 각 아이템을 출력한다.

---

p.57

```
import io.reactivex.Observable;

public class Launcher {
  public static void main(String[] args) {
    Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
    source.map(String::length).filter(i ⇒ i >= 5)
                .subscribe(s ⇒ System.out.println("RECEIVED: " + s));
  }
}
```

```
import io.reactivex.Observable;

public class Launcher {
  public static void main(String[] args) {
    Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
    source.map(String::length)
                .filter(i ⇒ i >= 5)
                .subscribe(s ⇒ System.out.println("RECEIVED: " + s));
  }
}
```

---

p.60

```
//do nothing with Disposable, disregard for now
⇒
//지금은 디스포저블을 사용하지 않는다.
```

---

p.68

JavaFX에서 Observable.create()를 사용해 ToggleButton의 selectedProperty() 연산자를 사용해 Observable<Boolean>을 만들 수 있다.

⇒

JavaFX에서 Observable.create()와 ToggleButton의 selectedProperty() 연산자를 사용해 Observable<Boolean>을 만들 수 있다.

---

p.76

지금 당장은 main() 메서드가 옵저버블을 시작하겠지만 그것이 끝나기를 기다리지는 않는다.

⇒

지금 당장은 main() 메서드가 옵저버블을 시작하겠지만, 메인 스레드는 옵저버블이 완료되는 것을 기다리지는 않는다.

---

p.80

RxJava 옵저버블은 Futures보다 강력하고 표현력이 뛰어나다. 그러나 Futures를 반환하는 기존 라이브러리가 있는 경우 Observable.future()를 통해 Observables로 쉽게 변환할 수 있다.

⇒

RxJava 옵저버블은 자바의 Futures보다 강력하고 표현력이 뛰어나다. 그러나 Futures를 반환하는 기존 라이브러리가 있는 경우 Observable.fromFuture()을 통해 옵저버블로 쉽게 변환할 수 있다.

---

p.81

옵저버에서 Done!을 인쇄한 onComplete의 호출이 곧바로 일어났다!

⇒

onComplete 이벤트를 수신한 옵저버는 Done!을 출력한다.

---

p.86

예를 들어, Observable.just()를 0으로 나누는 식으로 생성하려고 하면 옵저버까지 전달되지 않고 예외가 생긴다.

⇒

예를 들어, Observable.just()에서 값을 0으로 나누는 방식으로 옵저버블을 생성하려고 하면 옵저버까지 전달되지 않고 예외가 생긴다.

---

p.88

그것은 옵저버블과 똑같이 작동하지만, 단일 배출에 대해 작동하는 연산자로만 제한된다.

⇒

싱글은 옵저버블과 동일하게 작동하지만 하나의 배출을 대상으로 하는 연산자만 사용할 수 있다.

---

p.90

MaybeObserver는 더 일반적인 옵저버블과 유사하지만 onSuccess() 대신 onNext()가 호출된다.

⇒

MaybeObserver는 표준 옵저버와 유사하지만 onNext()는 onSuccess()를 대신 호출한다.

---

p.133

```
import io.reactivex.Observable;

public class Launcher {

  public static void main(String[] args) {
    Observable.range(1, 1000)
              .toList(1000)
              .subscribe(s ⇒ System.out.println("Received: " + s));
  }
}
```

```
import io.reactivex.Observable;
import java.util.concurrent.CopyOnWriteArrayList;

public class Launcher {

  public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
              .toList(CopyOnWriteArrayList::new)
              .subscribe(s ⇒ System.out.println("Received: " + s));
  }
}
```

p.144

```
import io.reactivex.Observable;

public class Launcher {

  public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
              .map(i ⇒ 10 / i)
              .onErrorReturnItem(-1)
              .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
              );
  }
}
```

```
import io.reactivex.Observable;

public class Launcher {

  public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
              .map(i ⇒ 10 / i)
               .onErrorResumeNext(Observable.empty())
              .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
              );
  }
}
```

p.145

```
import io.reactivex.Observable;

public class Launcher {

  public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
              .map(i ⇒ 10 / i)
              .onErrorReturn(e ⇒ −1)
              .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
              );
  }
}
```

```
import io.reactivex.Observable;

public class Launcher {

  public static void main(String[] args) {
    Observable.just(5, 2, 4, 0, 3, 2, 8)
              .map(i ⇒ 10 / i)
              .onErrorResumeNext((Throwable e) ⇒ Observable.just(−1).repeat(3))
              .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
              );
  }
}
```

p.146

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ {
                  try {
                    return 10 / i;
                  } catch (ArithmeticException e) {
                    return -1;
                  }
                })
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ 10 / i)
                .retry()
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e));
    }

}
```

p.147

```
import com.google.common.collect.ImmutableList;
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .collect(ImmutableList::builder, ImmutableList.Builder::add)
                .map(ImmutableList.Builder::build)
                .subscribe(s -> System.out.println("Received: " + s));
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i -> 10 / i)
                .retry(2)
                .subscribe(i -> System.out.println("RECEIVED: " + i),
                        e -> System.out.println("RECEIVED ERROR: " + e));
    }

}
```

p.149

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ 10 / i)
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .doOnNext(s ⇒ System.out.println("Processing: " + s))
                .map(String::length)
                .subscribe(i ⇒ System.out.println("Received: " + i));
    }
}
```

p.150

```
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {

        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ 10 / i)
                .onErrorReturnItem(-1)
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .doOnComplete(() ⇒ System.out.println("Source is done emitting!"))
                .map(String::length)
                .subscribe(i ⇒ System.out.println("Received: " + i));
    }
}
```

p.150~151

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ 10 / i)
                .onErrorReturn(e ⇒ −1)
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .doOnError(e ⇒ System.out.println("Source failed!"))
                .map(i ⇒ 10 / i)
                .doOnError(e ⇒ System.out.println("Division failed!"))
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e));
    }
}
```

p.152

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i -> {
                  try {
                    return 10 / i;
                  } catch (ArithmeticException e) {
                    return -1;
                  }
                })
                .subscribe(i -> System.out.println("RECEIVED: " + i),
                        e -> System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .doOnSubscribe(d -> System.out.println("Subscribing!"))
                .doOnDispose(() -> System.out.println("Disposing!"))
                .subscribe(i -> System.out.println("RECEIVED: " + i));
    }
}
```

p.153~154

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 2, 4, 0, 3, 2, 8)
                .map(i ⇒ 10 / i)
                .onErrorResumeNext(Observable.just(-1).repeat(3))
                .subscribe(i ⇒ System.out.println("RECEIVED: " + i),
                        e ⇒ System.out.println("RECEIVED ERROR: " + e)
                );
    }
}
```

```
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just(5, 3, 7, 10, 2, 14)
                .reduce((total, next) ⇒ total + next)
                .doOnSuccess(i ⇒ System.out.println("Emitting: " + i))
                .subscribe(i ⇒ System.out.println("Received: " + i));
    }
}
```

---

p.165

빈 문자열 인수 " "를 전달했다.
⇒
빈 문자열 인수 ""를 전달했다.

Split함수는 모든 문자를 포함하는 배열 String []을 반환하며,
⇒
Split함수는 모든 문자를 포함하는 배열 String[]을 반환하며,

---

p.222

```
import io.reactivex.subjects.ReplaySubject;
import io.reactivex.subjects.Subject;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject =
            ReplaySubject.create();
    subject.subscribe(s => System.out.println("Observer 1: " + s));
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.subscribe(s => System.out.println("Observer 2: " + s));
  }
}
```

```
import io.reactivex.subjects.BehaviorSubject;
import io.reactivex.subjects.Subject;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject = BehaviorSubject.create();
    subject.subscribe(s => System.out.println("Observer 1: " + s));

    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");

    subject.subscribe(s => System.out.println("Observer 2: " + s));
  }
}
```

p.223

```
import io.reactivex.subjects.AsyncSubject;
import io.reactivex.subjects.Subject;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject = AsyncSubject.create();
    subject.subscribe(s => System.out.println("Observer 1: " + s),
                      Throwable::printStackTrace,
                      () => System.out.println("Observer 1 done!")
);
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.subscribe(s => System.out.println("Observer 2: " + s),
                      Throwable::printStackTrace,
                      () => System.out.println("Observer 2 done!"));
  }
}
```

```
import io.reactivex.subjects.ReplaySubject;
import io.reactivex.subjects.Subject;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject =  ReplaySubject.create();
    subject.subscribe(s => System.out.println("Observer 1: " + s));
    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();
    subject.subscribe(s => System.out.println("Observer 2: " + s));
  }

}
```

p.224

```
import io.reactivex.Observable;
import io.reactivex.subjects.Subject;
import io.reactivex.subjects.UnicastSubject;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {

    Subject<String> subject = UnicastSubject.create();
    Observable.interval(300, TimeUnit.MILLISECONDS)
                .map(l ⇒ ((l + 1) * 300) + " milliseconds")
                .subscribe(subject);
    sleep(2000);
    subject.subscribe(s ⇒ System.out.println("Observer 1: " + s));

    sleep(2000);
  }

  public static void sleep(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```
import io.reactivex.subjects.AsyncSubject;
import io.reactivex.subjects.Subject;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject = AsyncSubject.create();

    subject.subscribe(s ⇒ System.out.println("Observer 1: " + s),
                      Throwable::printStackTrace,
                      () ⇒ System.out.println("Observer 1 done!")
    );

    subject.onNext("Alpha");
    subject.onNext("Beta");
    subject.onNext("Gamma");
    subject.onComplete();

    subject.subscribe(s ⇒ System.out.println("Observer 2: " + s),
                      Throwable::printStackTrace,
                      () ⇒ System.out.println("Observer 2 done!"));
  }
}
```

---

p.226

```
import io.reactivex.Observable;
import io.reactivex.subjects.Subject;
import io.reactivex.subjects.UnicastSubject;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject =
            UnicastSubject.create();
    Observable.interval(300, TimeUnit.MILLISECONDS)
              .map(l ⇒ ((l + 1) * 300) + " milliseconds")
              .subscribe(subject);

    sleep(2000);
    //여러 개의 옵저버를 지원하기 위해 멀티캐스팅
    Observable<String> multicast = subject.publish().autoConnect();

    //첫 번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 1: " + s));

    sleep(2000);

    //두 번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 2: " + s));
    sleep(1000);
  }

  public static void sleep(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```
import io.reactivex.Observable;
import io.reactivex.subjects.Subject;
import io.reactivex.subjects.UnicastSubject;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {

    Subject<String> subject = UnicastSubject.create();
    Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(l ⇒ ((l + 1) * 300) + " milliseconds")
            .subscribe(subject);
    sleep(2000);

    subject.subscribe(s ⇒ System.out.println("Observer 1: " + s));

    sleep(2000);
  }

  public static void sleep(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

---

p.228

```
import io.reactivex.Observable;
import io.reactivex.subjects.Subject;
import io.reactivex.subjects.UnicastSubject;
import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {

    Subject<String> subject =
              UnicastSubject.create();

    Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(l ⇒ ((l + 1) * 300) + " milliseconds")
            .subscribe(subject);

    sleep(2000);

    // 여러개의 옵저버를 지원하기 위해 멀티캐스팅
    Observable<String> multicast = subject.publish().autoConnect();

    //첫번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 1: " + s));

    sleep(2000);

    //두번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 2: " + s));

    sleep(1000);
  }

  public static void sleep(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```
import io.reactivex.Observable;
import io.reactivex.subjects.Subject;
import io.reactivex.subjects.UnicastSubject;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {
    Subject<String> subject = UnicastSubject.create();
    Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(l ⇒ ((l + 1) * 300) + " milliseconds")
            .subscribe(subject);

    sleep(2000);
    // 여러개의 옵저버를 지원하기 위해 멀티캐스팅
    Observable<String> multicast = subject.publish().autoConnect();

    //첫번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 1: " + s));

    sleep(2000);

    //두번째 옵저버를 등록
    multicast.subscribe(s ⇒ System.out.println("Observer 2: " + s));
    sleep(1000);
  }

  public static void sleep(long millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

---

p.247

SQL 작업을 수행할 수 있다.

⇒

SQL 작업을 수행할 때 Schedulers.io()를 사용할 수 있다.

---

p.250

ExecutorService는 프로그램을 무한으로 유지하기 때문에 애플리케이션이 유한한 경우 직접 해지 처리를 해야 한다.

⇒

ExecutorService는 프로그램을 무한으로 유지하기 때문에 애플리케이션을 종료하고 싶다면 직접 해지 처리를 해야 한다.

---

p.251

Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

⇒

//다음 세 가지 예제에서 subscribeOn은 모두 같은 효과를 낸다.
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")

---

p.254

Received 5 on thread RxComputationThreadPool-2
Received 4 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-2
Received 5 on thread RxComputationThreadPool-1
Received 7 on thread RxComputationThreadPool-2
Received 4 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1

Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 4 on thread RxComputationThreadPool-1
Received 4 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1
Received 5 on thread RxComputationThreadPool-1

---

p.256

```
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {
    Observable.interval(1, TimeUnit.SECONDS, Schedulers.newThread())
              .subscribe(i ⇒ System.out.println("Received " + i + " on thread "
+
                        Thread.currentThread().getName()));
    sleep(5000);
  }
  public static void sleep(int millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;

import java.net.URL;
import java.util.Scanner;

public class Launcher {
  public static void main(String[] args) {
    Observable.fromCallable(() ⇒
            getResponse("https://api.github.com/users/thomasnield/starred")
    ).subscribeOn(Schedulers.io())
     .subscribe(System.out::println);
    sleep(10000);
  }

  private static String getResponse(String path) {
    try {
      return new Scanner(new URL(path).openStream(),
              "UTF-8").useDelimiter("//A").next();
    } catch (Exception e) {
      return e.getMessage();
    }
  }
  public static void sleep(int millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

p.257

subscribeOn()의 뉘앙스

⇒

subscribeOn()의 주의사항

---

p.257~258

```
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;
public class Launcher {
  public static void main(String[] args) {
    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
              .subscribeOn(Schedulers.computation())
              .filter(s ⇒ s.length() == 5)
              .subscribeOn(Schedulers.io())
              .subscribe(i ⇒ System.out.println("Received " + i + " on thread "
+
                      Thread.currentThread().getName()));
    sleep(5000);
  }

  public static void sleep(int millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

```
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;

import java.util.concurrent.TimeUnit;

public class Launcher {
  public static void main(String[] args) {
    Observable.interval(1, TimeUnit.SECONDS, Schedulers.newThread())
              .subscribe(i ⇒ System.out.println("Received " + i + " on thread " +
                      Thread.currentThread().getName()));
    sleep(5000);
  }

  public static void sleep(int millis) {
    try {
      Thread.sleep(millis);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

---

p.318

제한 없이 대기열에 저장되기 때문에

⇒

제한 없이 버퍼에 저장되기 때문에

---

p.332

subscribe()에서 반환된 Disposable 타입의 dispose()가 호출되면 배출이 멈출 수 있으며 for 루프가 이를 검사한다.

⇒

subscribe()에서 반환된 Disposable 타입의 dispose()를 호출하면 이를 검사하고 for 루프에 의해 배출이 중지된다.

---

p.337

Observable<String>이 다섯 개 이상의 배출이 있는 경우(예를 들어, 1,000개에서 10,000개) flatMap이 적용된 플로어블을 옵저버블로 바꾸는 대신 Observable<String>을 플로어블로 전환하는 것이 좋다.

⇒

Observable<String>이 다섯 개 이상을 배출하는 경우(예를 들어 1,000개에서 10,000개) flatMap에서 플로어블을 옵저버블로 변환하는 대신 Observable<String>을 플로어블로 직접 변환하는 편이 낫다.

Long.MAX_VALUE의 배출이 필요하다.

⇒

Long.MAX_VALUE의 배출을 요청한다.

---

p.339

개발자를 위한 더 많은 오버로드 인자가 있다.

⇒

개발자를 위한 더 많은 오버로드 메서드가 존재한다.

자바독(JavaDocs)을 참고하면 된다.

⇒

자바독(JavaDocs)을 참고하라(http://reactivex.io/RxJava/javadoc/).

열거형을 설정해

⇒

열거형을 설정하면

---

p.345

플로어블을 사용하고 대부분 표준 팩토리 메서드 및 연산자는 자동으로 백프레셔를 처리해주는 것이 사실이다.
⇒
플로어블과 함께 대부분의 표준 팩토리 메서드 및 연산자는 자동으로 백프레셔를 처리해준다.

다소 타협한 방식으로
⇒
다소 절충된 방식으로

먼저 소스인 Iterable<T>를 Flowable.fromIterable()의 인자로 사용하는 것을 고려해보자.
⇒
먼저 소스인 Iterable<T>를 Flowable.fromIterable()에 사용하는 것을 고려해보자.

---

p.346

96개의 배출이 먼저 푸시된다.
⇒
앞서 생성된 96개 배출이 먼저 다운스트림으로 푸시된다.

---

p.349

Flowable.generator()
⇒
Flowable.generate()

Flowable.generator()를 사용하는 것이 낫다.
⇒
Flowable.create()대신 Flowable.generate()를 사용하는 것이 낫다.

---

p.350

커스텀 플로어블 소스를 만드는 경우
⇒
플로어블을 직접 생성하는 경우

---

p.352

두 개의 기타 옵저버블 구독을
⇒
두 개의 서로 다른 옵저버블 구독을

---

p.354

연산자에 그것을

⇒

연산자에 트랜스포머를

---

p.357

구현할 때

⇒

구현한 것처럼

---

p.454

작은 코드를

⇒

적은 코드를
