



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 8 - PROGRAMARE REACTIVĂ

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- Programare funcțională și asincronă (3)
- **Programare reactivă** (1)
- Servicii web REST (2)
- Quarkus Framework (4)

CUPRINS CURS

- Introducere în programarea reactivă
- Observable și Subscriber
- Operatori de bază
- Combinarea mai multor Observable
- Multicasting, replaying și caching
- Paralelism și concurență
- Flowables și Backpressure
- Bibliografie

INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

- Este un stil de programare focalizat pe **reacția asincronă la schimbări**, care pot fi **date sau evenimente**
- **Programarea reactivă:** lucrul și analiza pe surse de date care se modifică în timp real (precum feed-uri Twitter sau valoarea acțiunilor la bursă)
- Având **posibilitatea de a compune evenimente și date ca stream-uri** acestea pot fi **agregate, filtrate, sparte și transformate**
- În cele mai multe cazuri se face imperativ. Un **callback** este un exemplu de **programare reactivă executată imperativ**
- **Programarea reactivă** este influențată de programarea funcțională și **folosește o abordare declarativă** pentru a evita problemele codului imperativ
- Acet stil de programare se numește **programare reactivă funcțională (FTP)**

INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

- **Programarea reactivă** este recomandată în scenarii precum:
 - **Procesarea unor evenimente** precum mișcările și clicurile mouse-ului, apăsarea tastelor, semnal GPS care se schimbă cu deplasarea dispozitivului, semnale de la dispozitive de tip giroscop, etc.
 - **Răspunsul și procesarea evenimentelor I/O** de la rețea sau de pe disc, care sunt nativ asincrone
 - **Manipularea evenimentelor și datelor pe care nu le putem controla** (log-uri produse pe diferite sisteme, cele menționate anterior etc.)
 - **Abordarea reactivă** se recomandă atunci când este **necesară combinarea evenimentelor sau răspunsurilor asincrone** produse de diferite stream-uri, între care există o logică de interacțiune

INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

- RxJava este o implementare concretă a principiilor programării reactive influențată de programarea funcțională
- Versiunea curentă este RxJava 2
- Tipul de bază cu care vom lucra **Observable<T>** care împinge elemente (emisii) printr-o serie de operatori până ce ajung la **Observer** care consumă acele elemente
- În exemplu avem un **Observable<String>** care împinge obiecte de tip String
- **Observable.just()** emite un set de elemente
- Este nevoie ca un **Observer** să subscrive la **Observable** pentru ca acesta să emită

INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> myStrings =
            Observable.just("A", "B", "C", "D", "E");
        myStrings.subscribe(System.out::println);
    }
}
```

Rezultat:
A
B
C
D
E

- **Observable<String>** împinge fiecare obiect de tip String la **Observer-ul nostru** care este o expresie lambda în acest caz
- Putem utiliza **operatori** între **Observable** și **Observer** care **transformă sau manipulează** elementele
- **Operatorii returnează** un nou **Observable** dar care reflectă transformarea (la fel ca stream-urile)

INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

- Putem folosi `map()` pentru a converti `String`-ul în lungimea sa, care apoi va fi împins unui `Observer`

```
package atm.paradigms;
import io.reactivex.Observable;

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

Rezultat:

5
4
5
5
7

- Diferența dintre stream-uri și `Observable` este că **primul trage** (pull) evenimentele din secvență iar al **doilea le împinge** (push)
- `Observable` poate împinge evenimente



INTRODUCERE ÎN PROGRAMAREA REACTIVĂ

- **Observable.interval()** va împinge numere consecutive la un interval specificat

```
package atm.paradigms;
import java.util.concurrent.TimeUnit;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<Long> secondIntervals =
            Observable.interval(1, TimeUnit.SECONDS);
        secondIntervals.subscribe(System.out::println);
        // hold main thread for 5 s
        sleep(6000);
    }

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

Rezultat:

0
1
2
3
4
5



OBSERVABLE ȘI SUBSCRIBER

- **Observable**<T> împinge elemente (numite emisii) de tip T printr-o serie de operatori până ce ajunge în final la un **Observer** care le consumă
- Elemente sunt trimise secvențial **Observer-ului**
- **Observable** trece 3 tipuri de evenimente:
- **onNext()**: trece fiecare element unul câte unul de la Observable sursă la Observer
- **onComplete()**: comunică un eveniment de finalizare **Observer-ului** anunțând că nu se vor mai avea loc invocări **onNext()**
- **onError()**: comunică **Observer-ului** eroarea pe care acesta o tratează. Dacă nu invocă **retry()**, Observable nu mai trimitе alte emisii



OBSERVABLE ȘI SUBSCRIBER

OBSERVABLE.CREATE()

- Metoda statică **Observable.create()** permite **crearea unui Observable** printr-un lambda
- Prin invocarea metodei **onNext()** pe emitter **trimitem emisia pe lanț**. La finalizare invocăm **onComplete()**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            emitter.onNext("Alpha");
            emitter.onNext("Beta");
            emitter.onNext("Gamma");
            emitter.onNext("Delta");
            emitter.onNext("Epsilon");
            emitter.onComplete();
        });
        source.subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

Rezultat:

RECEIVED: Alpha

RECEIVED: Beta

RECEIVED: Gamma

RECEIVED: Delta

RECEIVED: Epsilon



OBSERVABLE ȘI SUBSCRIBER

OBSERVABLE.CREATE()

- Modificăm codul anterior pentru a **prinde erorile și a le emite pe lanț cu onError()**. Eroarea este tratată la Observer

```
package atm.paradigms;
import io.reactivex.Observable;
public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            try {
                emitter.onNext("Alpha");
                emitter.onNext("Beta");
                emitter.onNext("Gamma");
                emitter.onNext("Delta");
                emitter.onNext("Epsilon");
                emitter.onComplete();
            } catch (Throwable e) {
                emitter.onError(e);
            }
        });
        source.subscribe(s -> System.out.println("RECEIVED: " + s), Throwable::printStackTrace);
    }
}
```

OBSERVABLE ȘI SUBSCRIBER

OBSERVABLE.CREATE()

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.create(emitter -> {
            try {
                emitter.onNext("Alpha");
                emitter.onNext("Beta");
                emitter.onNext("Gamma");
                emitter.onNext("Delta");
                emitter.onNext("Epsilon");
                emitter.onComplete();
            } catch (Throwable e) { emitter.onError(e); }
        });
        source.map(String::length) // convert to int
        .filter(i -> i > 4)      // filter up from 5
        .subscribe(System.out::println,
                  Throwable::printStackTrace);
    }
}
```

Rezultat:

5
5
5
7

Se aplică operatorii **map()** și **filter()** pe **pipeline** pentru a converti Stringurile în numere întregi și a le elimina pe cele mai mici ca 5.



OBSERVABLE ȘI SUBSCRIBER

OBSERVABLE.JUST()

- Pentru **surse comune** se poate folosi **Observable.just()**, care poate **primi până la 10 elemente**. Invocă automat **onNext()** pe fiecare element și **onComplete()** la final

```
package atm.paradigms;
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) // convert to int
            .filter(i -> i > 4)      // filter up from 5
            .subscribe(System.out::println,
                    Throwable::printStackTrace);
    }
}
```

OBSERVABLE ȘI SUBSCRIBER

OBSERVABLE.FROMITERABLE()

- Putem emite elemente dintr-un tip iterabil precum o listă folosind `Observable.fromIterable()`

```
package atm.paradigms;
import java.util.Arrays;
import java.util.List;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        List<String> items = Arrays.asList("Alpha", "Beta", "Gamma", "Delta",
        "Epsilon");
        Observable<String> source = Observable.fromIterable(items);
        source.map(String::length) // convert to int
        .filter(i -> i > 4)      // filter up from 5
        .subscribe(System.out::println,
                  Throwable::printStackTrace);
    }
}
```

OBSERVABLE ȘI SUBSCRIBER

INTERFAȚA OBSERVER

```
package io.reactivex;  
  
import io.reactivex.disposables.Disposable;  
  
public interface Observer<T> {  
  
    void onSubscribe(Disposable d);  
  
    void onNext(T value);  
  
    void onError(Throwable e);  
  
    void onComplete();  
  
}
```

- Metodele **onNext()**, **onComplete()**, și **onError()** definesc tipul **Observer**
- Fiecare **Observable returnat** de un operator este **intern un Observer** care **primește asincron, transformă și dă mai departe emisiile către următorul Observer din lanț**

OBSERVABLE ȘI SUBSCRIBER

INTERFAȚA OBSERVER

- Implementarea interfeței **Observer** ca o clasă anonimă

```
Observable<String> source = Observable.just("Alpha", "Beta", "Gamma", "Delta",
                                             "Epsilon");
Observer<Integer> myObserver = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        // do nothing with Disposable, disregard for now
    }
    @Override
    public void onNext(Integer value) {
        System.out.println("RECEIVED: " + value);
    }
    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }
    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
};
source.map(String::length) // convert to int
    .filter(i -> i > 4) // filter up from 5
    .subscribe(myObserver);
```

Implementarea unui **Observer** ca o clasă anonimă în scopul demonstrării metodelor **asincrone onNext(), onComplete(), onError()**.



OBSERVABLE ȘI SUBSCRIBER

INTERFAȚA OBSERVER

- Metoda `subscribe()` are versiuni supraîncărcate (cu 3, 2 sau 1 parametri) care acceptă expresii lambda:

```
Consumer<Integer> onNext = i -> System.out.println("RECEIVED: „ + i);
```

```
Action onComplete = () -> System.out.println("Done!");
```

```
Consumer<Throwable> onError = Throwable::printStackTrace;
```

```
package atm.paradigms;
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) // convert to int
            .filter(i -> i > 4) // filter up from 5
            .subscribe(System.out::println,
                    Throwable::printStackTrace,
                    () -> System.out.println("Done!"));
    }
}
```

Rezultat:

5

5

5

7

Done!

În metoda `subscribe()` parametrii corespunzători metodelor `onNext()`, `onComplete()`, `onError()` sunt implementate prin expresii lambda.

OBSERVABLE ȘI SUBSCRIBER

COLD OBSERVABLES

- **Cold Observables retransmit emisiile pentru fiecare Observer, asigurându-se că toți Observer-i primesc toate datele.** Exemple `Observable.just()` și `Observable.fromIterable()`

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
        //first observer
        source.subscribe(s ->
            System.out.println("Observer 1 Received:" + s));
        //second observer
        source.subscribe(s ->
            System.out.println("Observer 2 Received:" + s));
    }
}
```

Rezultat:

```
Observer 1 Received:Alpha
Observer 1 Received:Beta
Observer 1 Received:Gamma
Observer 1 Received:Delta
Observer 1 Received:Epsilon
Observer 2 Received:Alpha
Observer 2 Received:Beta
Observer 2 Received:Gamma
Observer 2 Received:Delta
Observer 2 Received:Epsilon
```

OBSERVABLE ȘI SUBSCRIBER

HOT OBSERVABLES

- Hot Observables reprezintă evenimente mai curând decât seturi de date finite
- Transmite aceleasi emisii la toți Observer-i în același timp. Un Observer care a subscris mai târziu va pierde unele emisii
- **ConnectableObservable** transformă orice fel de Observable în **Hot Observable** prin invocarea metodei **publish()**. Emisiile încep la **connect()** - *multicasting*

```
package atm.paradigms;
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;

public class Launcher {
    public static void main(String[] args) {
        ConnectableObservable<String> source =
            Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon").publish();
        source.subscribe(s -> System.out.println("Observer 1: " + s));
        source.map(String::length)
            .subscribe(i -> System.out.println("Observer 2: " + i));
        source.connect(); //Fire!
    }
}
```

Rezultat:

Observer 1: Alpha
Observer 2: 5
Observer 1: Beta
Observer 2: 4
Observer 1: Gamma
Observer 2: 5
Observer 1: Delta
Observer 2: 5
Observer 1: Epsilon
Observer 2: 7

OBSERVABLE ȘI SUBSCRIBER

ALTE SURSE OBSERVABLE

- ***Observable.range()*** - pentru a emite **intervale de numere întregi**, primul argument este **numărul de început** iar al doilea este **numărul de emisii**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.range(5, 10)
            .subscribe(s -> System.out.print(s + " "));
    }
}
```

- **Rezultat:** 5 6 7 8 9 10 11 12 13 14
- ***Observable.interval()*** – produce emisii **în timp consecutiv** începând cu 0 la intervalul de timp specificat



OBSERVABLE ȘI SUBSCRIBER

SINGLE ȘI MAYBE

- Există **cazuri speciale de Observable** setate pentru **una sau nicio emisie**
- Pot fi **creați în mod similar Observable** dar pot fi **returnați și de anumiți operatori**
- **Single** – este un **Observable<T>** care **emite doar un singur element**

```
package atm.paradigms;
import io.reactivex.Observable;
import io.reactivex.Single;

public class Launcher {
    public static void main(String[] args) {
        Single.just("Hello")
            .map(String::length).subscribe(System.out::println,
                                         Throwable::printStackTrace);

        Observable<String> source = Observable.just("Alpha", "Beta", "Gamma");
        source.first("Nil") // returns a Single
            .subscribe(System.out::println);
    }
}
```

Rezultat:
5
Alpha

OBSERVABLE ȘI SUBSCRIBER

SINGLE ȘI MAYBE

- **Maybe** – poate transmite o emisie sau niciuna. **Maybe Observer** asociat are metoda **onSuccess()** în loc de **onNext()**

```
package atm.paradigms;
import io.reactivex.Maybe;

public class Launcher {
    public static void main(String[] args) {
        Maybe<Integer> presentSource = Maybe.just(100);
        presentSource.subscribe(s ->
            System.out.println("Process 1 received: " + s),
            Throwable::printStackTrace,
            () -> System.out.println("Process 1 done!"));
        // no emission
        Maybe<Integer> emptySource = Maybe.empty();
        emptySource.subscribe(s ->
            System.out.println("Process 2 received: " + s),
            Throwable::printStackTrace,
            () -> System.out.println("Process 2 done!"));
    }
}
```

Rezultat:

Process 1 received: 100

Process 2 done!



OPERATORI DE BAZĂ DE SUPRIMARE

- **filter()** – Acceptă **Predicate<T>** pentru o sursă **Observable<T>** dată. Poartă numele și de **operator de suprimare** deoarece **suprimă emisiile care nu satisfac un criteriu dat**

```
package atm.paradigms;
import io.reactivex.Observable;

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

- *Rezultat:*
RECEIVED: Beta
RECEIVED: Epsilon



OPERATORI DE BAZĂ DE SUPRIMARE

- **take()** – are 2 variante: una care cheamă **onComplete()** după un număr dat de emisii și alta care ia emisii pentru o durată de timp apoi cheamă **onComplete()**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
            .take(3)
            .subscribe(s -> System.out.println("RECEIVED: " + s));
    }
}
```

Rezultat:

RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma



OPERATORI DE BAZĂ DE SUPRIMARE

```
package atm.paradigms;
import java.util.concurrent.TimeUnit;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.interval(300, TimeUnit.MILLISECONDS)
            .take(2, TimeUnit.SECONDS)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
        sleep(5000);
    }

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

Operatorul `take()` acceptă emisii pentru 2s apoi cheamă `onComplete()`.

Rezultat:

RECEIVED: 0
RECEIVED: 1
RECEIVED: 2
RECEIVED: 3
RECEIVED: 4
RECEIVED: 5



OPERATORI DE BAZĂ DE SUPRIMARE

- **skip()** – opusul lui **take()**, ignoră un număr dat de emisii și apoi le emite pe cele care urmează

```
ackage atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.range(1, 100)
            .skip(95)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
    }
}
```

Rezultat:

RECEIVED: 96
RECEIVED: 97
RECEIVED: 98
RECEIVED: 99
RECEIVED: 100



OPERATORI DE BAZĂ DE SUPRIMARE

Operator	Descriere
takeWhile()	În emisii câte vreme o condiție este adevărată
<code>Observable.range(1,100).takeWhile(i -> i < 5) .subscribe(i -> System.out.println("RECEIVED: " + i));</code>	
skipWhile()	Elimină emisii câtă vreme o condiție este adevărată
<code>Observable.range(1,100).skipWhile(i -> i <= 95) .subscribe(i -> System.out.println("RECEIVED: " + i));</code>	
distinct()	Trece emisii unice și suprimă duplicatele
<code>Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon").distinct(String::length) .subscribe(i -> System.out.println("RECEIVED: " + i));</code>	
elementAt()	Trece o emisie specifică pe baza indicelui
<code>Observable.just("Alpha", "Beta", "Zeta", "Eta", "Gamma", "Delta").elementAt(3) .subscribe(i -> System.out.println("RECEIVED: " + i));</code>	



OPERATORI DE BAZĂ DE TRANSFORMARE

- **map()** – fiind dat un *Observable<T>* operatorul **transformă o emisie T într-o emisie R folosind lambda de tip *Function<T, R>***

```
package atm.paradigms;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        DateTimeFormatter dtf =
            DateTimeFormatter.ofPattern("M/d/yyyy");
        Observable.just("1/3/2016", "5/9/2016", "10/12/2016")
            .map(s -> LocalDate.parse(s, dtf))
            .subscribe(i -> System.out.println("RECEIVED: " + i));
    }
}
```

Rezultat:
RECEIVED: 2016-01-03
RECEIVED: 2016-05-09
RECEIVED: 2016-10-12

Transformă **String** în **LocalDate**
folosind un **DateTimeFormatter** în
metoda statică **LocalDate.parse()**.



OPERATORI DE BAZĂ DE TRANSFORMARE

Operator	Descriere
cast()	Transformă tipul fiecărei emisii
<code>Observable<Object> items = Observable.just("Alpha", "Beta", "Gamma") .map(s -> (Object) s);</code>	
startWith()	Permite introducerea unei emisii de tip T înaintea celorlalte
<code>menu.startWith("COFFEE SHOP MENU") .subscribe(System.out::println);</code>	
defaultIfEmpty()	Se poate specifica o emisie dacă nu există nici una
<code>items.filter(s -> s.startsWith("Z")) .defaultIfEmpty("None") .subscribe(System.out::println);</code>	
switchIfEmpty()	Specifică o altă sursă Observable dacă cea originală este goală
<code>Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon") .filter(s -> s.startsWith("Z")) .switchIfEmpty(Observable.just("Zeta", "Eta", "Theta")) .subscribe(i -> System.out.println("RECEIVED: " + i), e -> System.out.println("RECEIVED ERROR: " + e));</code>	



OPERATORI DE BAZĂ DE TRANSFORMARE

- **sorted()** – sortează un *Observable<T>* finit folosind interfața *Comparable<T>*.
- **Intern colectează emisiile apoi le emite în ordine sortată**

```
package atm.paradigms;
import java.util.Comparator;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just(6, 2, 5, 7, 1, 4, 9, 8, 3)
            .sorted(Comparator.reverseOrder())
            .subscribe(System.out::println);
    }
}
```

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just("Alpha", "Beta",
                        "Gamma", "Delta", "Epsilon")
            .sorted((x, y) ->
                    Integer.compare(x.length(), y.length()))
            .subscribe(System.out::println);
    }
}
```



OPERATORI DE BAZĂ DE TRANSFORMARE

Operator	Descriere
delay()	Retîne emisiile și le întârzie cu o perioadă specificată de timp
	<pre>Observable.just("Alpha", "Beta", "Gamma" , "Delta", "Epsilon").delay(3, TimeUnit.SECONDS) .subscribe(s -> System.out.println("Received: " + s));</pre>
repeat()	Repetă emisiile după onComplete() de un număr precizat de ori
	<pre>Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon") .repeat(2).subscribe(s -> System.out.println("Received: " + s));</pre>
scan()	Adună fiecare emisie la un acumulator pe care îl emite incremental
	<pre>Observable.just(5, 3, 7, 10, 2, 14).scan((accumulator, next) -> accumulator + next) .subscribe(s -> System.out.println("Received: " + s));</pre>



OPERATORI DE BAZĂ DE REDUCERE

- **la o serie de emisii și o consolidează într-o singură emisie**
- **count() – numără emisiile și trimită valoarea**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
            .count()
            .subscribe(s -> System.out.println("Received: " + s));
    }
}
```



OPERATORI DE BAZĂ DE REDUCERE

Operator	Descriere
reduce()	Similar cu scan() dar emite acumularea finală pe onComplete() <pre>Observable.just(5, 3, 7, 10, 2, 14).reduce((total, next) -> total + next) .subscribe(s -> System.out.println("Received: " + s));</pre>
all()	Verifică dacă toate emisiile respectă o anumită condiție <pre>Observable.just(5, 3, 7, 11, 2, 14).all(i -> i < 10) .subscribe(s -> System.out.println("Received: " + s));</pre>
any()	Verifică dacă cel puțin o emisie respectă o anumită condiție <pre>Observable.just("2016-01-01", "2016-05-02", "2016-09-12", "2016-04-03") .map(LocalDate::parse).any(dt -> dt.getMonthValue() >= 6) .subscribe(s -> System.out.println("Received: " + s));</pre>



OPERATORI DE BAZĂ DE COLECTARE

- Operatorii de colectare acumulează toate emisiile într-o colecție precum o listă sau map pe care apoi o emit într-o singură emisie
- **toList()** – colectează emisiile primite într-o listă apoi transmite *List<T>* într-o singură emisie. Implicit folosește *ArrayList*

```
package atm.paradigms;
import io.reactivex.Observable;

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

- Rezultat: Received: [Alpha, Beta, Gamma, Delta, Epsilon]



OPERATORI DE BAZĂ DE COLECTARE

- ***toSortedList()*** – la fel ca ***toList()*** doar că elementele sunt sortate în ordine naturală
- La fel ca ***sorted()*** poate primi un **Comparator** ca argument

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just(6, 2, 5, 7, 1, 4, 9, 8, 3)
            .toSortedList()
            .subscribe(s -> System.out.println("Received: " + s));
    }
}
```

- **Rezultat:**
Received: [1, 2, 3, 4, 5, 6, 7, 8, 9]



OPERATORI DE BAZĂ DE COLECTARE

- ***toMap()*** – colectează emisiile în *Map<K, T>* unde K este **cheia derivată de lambda de tip Function<T, K>**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just("Alpha", "Beta", "Gamma", "Delta",
                        "Epsilon")
            .toMap(s -> s.charAt(0))
            .subscribe(s -> System.out.println("Received: " + s));
    }
}
```

- **Rezultat:**
Received: {A=Alpha, B=Beta, D=Delta, E=Epsilon, G=Gamma}



OPERATORI DE BAZĂ DE COLECTARE

- **collect()** – se folosește pentru a **specifica tipuri diferite colecții** neacoperite de metodele anterioare

```
package atm.paradigms;
import java.util.HashSet;
import io.reactivex.Observable;

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

- **Rezultat:**
Received: [Gamma, Delta, Alpha, Epsilon, Beta]

COMBINAREA MAI MULTOR OBSERVABLE MERGING

- Programarea reactivă permite **combinarea mai multor Observable** sursă și **consolidarea acestora într-un singur Observable**
- **Observable.merge()** – ia **2 sau mai multe surse Observable<T>** și le **combină într-un singur Observable<T>** la care **Observer subscrive**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source1 = Observable.just("Alpha", "Beta",
                "Gamma", "Delta", "Epsilon");
        Observable<String> source2 = Observable.just("Zeta", "Eta", "Theta");
        Observable.merge(source1, source2)
            .subscribe(i -> System.out.println("RECEIVED: " + i));
    }
}
```

Rezultat:

RECEIVED: Alpha
RECEIVED: Beta
RECEIVED: Gamma
RECEIVED: Delta
RECEIVED: Epsilon
RECEIVED: Zeta
RECEIVED: Eta
RECEIVED: Theta

COMBINAREA MAI MULTOR OBSERVABLE MERGING

```
public static void main(String[] args) {
    // emit every second
    Observable<String> source1 = Observable.interval(1,
                                                       TimeUnit.SECONDS)
        .map(l -> l + 1) // emit elapsed seconds
        .map(l -> "Source1: " + l + " seconds");
    // emit every 300 milliseconds
    Observable<String> source2 = Observable.interval(300,
                                                       TimeUnit.MILLISECONDS)
        .map(l -> (l + 1) * 300) // emit elapsed milliseconds
        .map(l -> "Source2: " + l + " milliseconds");
    Observable.merge(source1, source2)
        .subscribe(System.out::println);
    // keep alive for 3 seconds
    sleep(3000);
}
```

Rezulta:

Source2: 300 milliseconds
Source2: 600 milliseconds
Source2: 900 milliseconds
Source1: 1 seconds
Source2: 1200 milliseconds
Source2: 1500 milliseconds
Source2: 1800 milliseconds
Source1: 2 seconds
Source2: 2100 milliseconds
Source2: 2400 milliseconds
Source2: 2700 milliseconds
Source2: 3000 milliseconds



COMBINAREA MAI MULTOR OBSERVABLE MERGING

[Cuprins](#)

Metodă	Descriere
mergeWith()	Varianta operator a metodei merge() <code>source1.mergeWith(source2)</code> <code>.subscribe(i -> System.out.println("RECEIVED: " + i));</code>
mergeArray()	Varianta care primește argumente variabile (varargs) <code>Observable.mergeArray(source1, source2, source3, source4, source5)</code>
flatMap()	Mapează o emisie la mai multe emisii

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.just("521934/2342/FOXTROT",
"21962/12112/78886/TANGO", "283242/4542/WHISKEY/2348562");
        source.flatMap(s -> Observable.fromArray(s.split("/")))
            .filter(s -> s.matches("[0-9]+")) // use regex to filter integers
            .map(Integer::valueOf)
            .subscribe(System.out::println);
    }
}
```

Operatorul **flatMap()** transformă `Observable<String[]>` în `Observable<String>` și emite cuvintele individuale. Funcționalitate este similară **flatMap()** din Stream API.



COMBINAREA MAI MULTOR OBSERVABLE

CONCATENATION

- Este similar cu *merging*, diferența este că va **trimite emisiile fiecărui Observable secvențial** în ordinea precizată
- **Ordinea** în care se produc emisiile **este garantată**

Metodă	Descriere
concat()	Trimite emisiile secvențial. Nu trece la următoarea sursă până nu o termină pe prima
Observable.concat(source1, source2) .subscribe(i -> System.out.println("RECEIVED: " + i));	
concatWith()	Realizează același lucru dar în mod operator
source1.concatWith(source2) .subscribe(i -> System.out.println("RECEIVED: " + i));	
concatMap()	Similar cu flatMap() dar procesarea emisiilor se face secvențial pe fiecare sursă
source.concatMap(s -> Observable.fromArray(s.split("")))) .subscribe(System.out::println);	



COMBINAREA MAI MULTOR OBSERVABLE

ZIPPING

- Permite **crearea de emisii prin combinarea emisiilor de la 2 surse**
- Observable.** Emisiile pot fi de tipuri diferite

```
package atm.paradigms;
import java.time.LocalTime;
import java.util.concurrent.TimeUnit;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> strings = Observable.just("Alpha", "Beta",
                "Gamma", "Delta", "Epsilon");
        Observable<Long> seconds = Observable.interval(1, TimeUnit.SECONDS);
        Observable.zip(strings, seconds, (s, i) -> i + " " + s)
            .subscribe(s -> System.out.println("Received " + s +
                    " at " + LocalTime.now()));
        sleep(6000);
    }
    public static void sleep(long millis) {
        // ..
    }
}
```

```
Received 0 Alpha at 13:06:07.982958500
Received 1 Beta at 13:06:08.957260600
Received 2 Gamma at 13:06:09.957572200
Received 3 Delta at 13:06:10.954893300
Received 4 Epsilon at 13:06:11.963933600
```



COMBINAREA MAI MULTOR OBSERVABLE GROUPING

- Se pot grupa emisiile după o cheie în Observable separați
- Va emite un tip special de Observable numit **GroupedObservable<K,T>**

```
package atm.paradigms;
import io.reactivex.Observable;
import io.reactivex.observables.GroupedObservable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.just("Alpha", "Beta",
                                                       "Gamma", "Delta", "Epsilon");
        Observable<GroupedObservable<Integer, String>> byLengths =
                source.groupBy(s -> s.length());
        byLengths.flatMapSingle(grp -> grp.toList())
                  .subscribe(System.out::println);
    }
}
```

Rezultat:
[Beta]
[Alpha, Gamma, Delta]
[Epsilon]

Operatorul **flatMapSingle()** extrage
valorile din **GroupedObservable** ca liste
și emite listele ca o singură emisie.

COMBINAREA MAI MULTOR OBSERVABLE GROUPING

```
package atm.paradigms;
import io.reactivex.Observable;
import io.reactivex.observables.GroupedObservable;

public class Launcher {
    public static void main(String[] args) {
        Observable<String> source = Observable.just("Alpha", "Beta",
                                                       "Gamma", "Delta", "Epsilon");
        Observable<GroupedObservable<Integer, String>> byLengths = source.groupBy(s -> s.length());
        byLengths.flatMapSingle(grp -> grp.reduce("", (x, y) ->
                                                       x.equals("") ? y : x + ", " + y)
                               .map(s -> grp.getKey() + ": " + s))
        .subscribe(System.out::println);
    }
}
```

Rezultat:

4: Beta
5: Alpha, Gamma, Delta
7: Epsilon

Se folosește un *pipeline* pe **GroupedObservable** pentru a concatena liste de valori cu **reduce()** și a adăuga cheia cu **map()** și metoda **getKey()**.

MULTICASTING, REPLAYING ȘI CACHING

- Am discutat anterior cum **cold Observables** care **va regenera emisiile** pentru fiecare **Observer** subscris
- De asemenea am văzut cum putem crea **hot Observables** folosind **ConnectableObservable**. Cu metodele **publish()** și **connect()** emisiile sunt **trimise simultan** pentru fiecare Observer subscris
- Ideea de stream consolidat poartă numele de **multicasting**
- **Multicasting-ul** poate fi **influențat de operatori**
- În exemplu următor vom folosi **Observable.range()** și vom mapea emisiile la un întreg aleator.
- Pentru ca **multicasting-ul** să funcționeze, **Observer-i** ar trebui să primească aceleași **numere**

MULTICASTING, REPLAYING ȘI CACHING

```
package atm.paradigms;
import java.util.concurrent.ThreadLocalRandom;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<Integer> threeRandoms = Observable.range(1, 3)
            .map(i -> randomInt());
        threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
        threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
    }

    public static int randomInt() {
        return ThreadLocalRandom.current().nextInt(100000);
    }
}
```

Rezultat:

```
Observer 1: 27758
Observer 1: 79993
Observer 1: 6138
Observer 2: 66911
Observer 2: 78873
Observer 2: 83609
```

- Un **cold Observable** va genera o secvențe de emisii pentru fiecare **Observer**.
Fiecare stream va avea propriul map și rezultatele vor fi diferite



MULTICASTING, REPLAYING ȘI CACHING

- Următoarea idee ar fi să folosim **ConnectableObservable**.

```
package atm.paradigms;
import java.util.concurrent.ThreadLocalRandom;
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;

public class Launcher {
    public static void main(String[] args) {
        ConnectableObservable<Integer> threeInts =
                Observable.range(1, 3).publish();
        Observable<Integer> threeRandoms = threeInts.map(i -> randomInt());
        threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
        threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
        threeInts.connect();
    }

    public static int randomInt() {
        return ThreadLocalRandom.current().nextInt(100000);
    }
}
```

Rezultat:

Observer 1: 62919
 Observer 2: 65227
 Observer 1: 93786
 Observer 2: 48706
 Observer 1: 41677
 Observer 2: 28743

Observăm că se obțin valori diferite.
 În cazul **multicasting**-ului ne așteptăm ca aceleși emisii să fie trimise la toți subscriber-i.



MULTICASTING, REPLAYING ȘI CACHING

- Nici acum nu am obținut rezultatul dorit, deoarece **multicasting** are loc **înainte de map**. Fiecare **Observer** va primi un **stream** pe care **aplicăm map separat**
- Soluția este să se aplice **map** **înainte de publish**

```
package atm.paradigms;
import java.util.concurrent.ThreadLocalRandom;
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;

public class Launcher {
    public static void main(String[] args) {
        ConnectableObservable<Integer> threeRandoms =
            Observable.range(1, 3).map(i -> randomInt()).publish();
        threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
        threeRandoms.subscribe(i -> System.out.println("Observer 2: " + i));
        threeRandoms.connect();
    }
    public static int randomInt() {
        return ThreadLocalRandom.current().nextInt(100000);
    }
}
```

Rezultat:

Observer 1: 77512
Observer 2: 77512
Observer 1: 4948
Observer 2: 4948
Observer 1: 94795
Observer 2: 94795



MULTICASTING, REPLAYING ȘI CACHING

- Multicasting-ul se recomandă în prevenirea redundanței. Atunci când mai mulți Observer-i efectuează operațiuni comune vor folosi un singur stream consolidat
- Exemplu: Un **Observer tipărește numerele aleatoare** iar un **altul calculează suma**

```
package atm.paradigms;
import java.util.concurrent.ThreadLocalRandom;
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;

public class Launcher {
    public static void main(String[] args) {
        ConnectableObservable<Integer> threeRandoms = Observable.range(1, 3)
                .map(i -> randomInt()).publish();
        threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
        threeRandoms.reduce(0, (total, next) -> total + next)
                .subscribe(i -> System.out.println("Observer 2: " + i));
        threeRandoms.connect();
    }
    // ...
}
```

Rezultat:

Observer 1: 20754
Observer 1: 51428
Observer 1: 64894
Observer 2: 137076



MULTICASTING, REPLAYING ȘI CACHING

- Există **varianta de conectare automată folosind `autoConnect()`**. Există riscul ca Observer-ii să nu primească toate emisiile

```
package atm.paradigms;
import java.util.concurrent.ThreadLocalRandom;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<Integer> threeRandoms = Observable.range(1, 3)
            .map(i -> randomInt()).publish().autoConnect(2);
        threeRandoms.subscribe(i -> System.out.println("Observer 1: " + i));
        threeRandoms.reduce(0, (total, next) -> total + next)
            .subscribe(i -> System.out.println("Observer 2: " + i));
    }

    public static int randomInt() {
        return ThreadLocalRandom.current().nextInt(100000);
    }
}
```

Rezultat:

Observer 1: 49234
Observer 1: 36525
Observer 1: 92033
Observer 2: 177792



MULTICASTING, REPLAYING ȘI CACHING

- Operatorul **replay()** constituie o modalitate de a păstra emisiile anterioare și a le reemite când apare un nou Observer
- Returnează un **ConnectableObservable** care va transmite emisiile curente în mod multicast dar și pe cele anterioare. Emisiile anterioare sunt trimise imediat la conectarea unui nou Observer

```
public static void main(String[] args) {  
    Observable<Long> seconds =  
        Observable.interval(1, TimeUnit.SECONDS)  
            .replay()  
            .autoConnect();  
    // Observer 1  
    seconds.subscribe(l -> System.out.println("Observer 1: " + l));  
    sleep(3000);  
    // Observer 2  
    seconds.subscribe(l -> System.out.println("Observer 2: " + l));  
    sleep(3000);  
}
```

Rezultat:

```
Observer 1: 0  
Observer 1: 1  
Observer 1: 2  
Observer 2: 0  
Observer 2: 1  
Observer 2: 2  
Observer 1: 3  
Observer 2: 3  
Observer 1: 4  
Observer 2: 4  
Observer 1: 5  
Observer 2: 5
```



MULTICASTING, REPLAYING ȘI CACHING

- Dacă sursa este infinită sau dacă nu contează decât ultimele emisii, `replay()` poate primi ca argument `bufferSize`, care limitează numărul de emisii
- Există metode supraîncărcate pentru `replay()` care permit specificarea unui interval de timp în care emisiile sunt păstrate

```
public static void main(String[] args) {
    Observable<Long> seconds =
        Observable.interval(300, TimeUnit.MILLISECONDS)
            .map(l -> (l + 1) * 300) // map to elapsed milliseconds
            .replay(1, TimeUnit.SECONDS)
            .autoConnect();
    // Observer 1
    seconds.subscribe(l -> System.out.println("Observer 1: " + l));
    sleep(2000);
    // Observer 2
    seconds.subscribe(l -> System.out.println("Observer 2: " + l));
    sleep(1000);
}
```

Rezultat:

Observer 1: 300
Observer 1: 600
Observer 1: 900
Observer 1: 1200
Observer 1: 1500
Observer 1: 1800
Observer 2: 1500
Observer 2: 1800
Observer 1: 2100
Observer 2: 2100
Observer 1: 2400
Observer 2: 2400
...



MULTICASTING, REPLAYING ȘI CACHING

- Dacă se dorește **păstrarea pe termen nedefinit a emisiilor** se folosește operatorul **cache()**
- Acesta **ține valorile pe termen nelimitat**. Este **de evitat pentru surse Observable infinite**

```
package atm.paradigms;
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable<Integer> cachedRollingTotals =
            Observable.just(6, 2, 5, 7, 1, 4, 9, 8, 3)
                .scan(0, (total, next) -> total + next)
                .cache();
        cachedRollingTotals.subscribe(System.out::println);
    }
}
```

Rezultat:

0
6
8
13
20
21
25
34
42
45



PARALELISM ȘI CONCURENȚĂ

SCHEDULERS

- **Schedulers** – pentru un Observer dat **va furniza un thread** din thread pool **care va trimite emisiile**. După invocarea `onComplete()` **thread-ul revine în thread pool**
- Invocarea `Schedulers.computation()` menține **numărul thread-uri** în acord cu **nucleele procesorului** (mai există și altele)

```
Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
.subscribeOn(Schedulers.computation());
```

- **subscribeOn()** – stabilește **tipul de Scheduler** ce va fi folosit pe lanț precum și modul de utilizare a thread-urilor
- Având mai mulți **Observer-i** pe un singur **Observable** cu **subscribeOn()** fiecare **va fi executat în propriul thread** (sau își va aștepta rândul)



PARALELISM ȘI CONCURENȚĂ SCHEDULERS

- Operatorul **subscribeOn()** instruiește sursele Observable ce Scheduler să folosească pentru a trimite emisiile și se va aplica pe tot lanțul până la Observer
- **observeOn()** interceptează emisiile într-un punct din lanțul de Observable și comută la un alt Scheduler de la acel punct încolo

```
public static void main(String[] args) {  
    // Happens on IO Scheduler  
    Observable.just("WHISKEY/27653/TANGO", "6555/BRAVO",  
        "232352/5675675/FOXTROT")  
        .subscribeOn(Schedulers.io())  
        .flatMap(s -> Observable.fromArray(s.split("/")))  
    // Happens on Computation Scheduler  
    .observeOn(Schedulers.computation())  
    .filter(s -> s.matches("[0-9]+"))  
    .map(Integer::valueOf)  
    .reduce((total, next) -> total + next)  
    .subscribe(i -> System.out.println("Received " + i  
        + " on thread "  
        + Thread.currentThread().getName()));  
    sleep(1000);  
}
```

Rezultat:

Received 5942235 on thread
RxComputationThreadPool-1



FLOWABLES ȘI BACKPRESSURE

- Există situații în care **sursa produce mai rapid emisii decât operatorii din aval pot procesa**
- Este **recomandabil** ca proactiv să facem **sursa să pornească încet** apoi să **agreeze un ritm** cu operațiile din aval. Procesul poartă numele de **backpressure sau flow control**
- **Se activează** prin folosirea **Flowable** în loc de **Observable**

```
package atm.paradigms;

public final class MyItem {
    final int id;

    public MyItem(int id) {
        this.id = id;
        System.out.println("Constructing MyItem " + id);
    }
}
```

Folosim un **POJO imutabil** pentru emisii. Variabila de instanță **id** se initializează doar prin constructor. Acesta tipărește un mesaj ce marchează atribuirea unei valori variabilei.

FLOWABLES ȘI BACKPRESSURE

```
package atm.paradigms;
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;

public class Launcher {
    public static void main(String[] args) {
        // Happens on IO Scheduler
        Observable.range(1, 999_999_999)
            .map(MyItem::new)
            .observeOn(Schedulers.io())
            .subscribe(myItem -> {
                sleep(50);
                System.out.println("Received MyItem " + myItem.id);
            });
        sleep(Integer.MAX_VALUE);
    }

    public static void sleep(int millis) {
        // . .
    }
}
```

Rezultat:

...

Constructing MyItem 47359

Constructing MyItem 47360

Constructing MyItem 47361

Received MyItem 179

Constructing MyItem 47362

Constructing MyItem 47363

Constructing MyItem 47364

...

Emisiile sunt produse mult mai rapid decât pot fi consumate



FLOWABLES ȘI BACKPRESSURE

- *Flowable este varianta cu backpressure care reduce pasul conform operațiilor din aval*

```
package atm.paradigms;
import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;

public class Launcher {
    public static void main(String[] args) {
        // Happens on IO Scheduler
        Flowable.range(1, 999_999_999)
            .map(MyItem::new)
            .observeOn(Schedulers.io())
            .subscribe(myItem -> {
                sleep(50);
                System.out.println("Received MyItem " + myItem.id);
            });
        sleep(Integer.MAX_VALUE);
    }
    public static void sleep(int millis) {
        // ...
    }
}
```

Rezultat:

- - -

Received MyItem 95
Received MyItem 96
Constructing MyItem 129
Constructing MyItem 130
Constructing MyItem 131
Constructing MyItem 152
...
Constructing MyItem 222
Constructing MyItem 223
Constructing MyItem 224
Received MyItem 97
Received MyItem 98
Received MyItem 99
Received MyItem 100
...



FLOWABLES ȘI BACKPRESSURE

- **Flowable** folosește **Subscriber** în loc de **Observer**
- **Flowable.range()** construiește **128 emisii cu instanțe MyItem**
- **observeOn()** – trece 96 emisii **Subscriber-ului** pentru procesare. Când sunt gata mai trimit 96
- Lanțul **Flowable** are ca scop să aibă 96 de emisii pregătite pentru a fi transmise
- Acest mecanism poartă numele de **backpressure** și introduce o dinamică de **pull**, în afara celei de **push** prezentate, cu **scopul de a limita frecvența** cu care emite sursa
- **Flowable** se comportă la fel cu **Observable** pentru aproape toți operatorii
învătați până acum

BIBLIOGRAFIE

- **Learning RxJava: Build concurrent, maintainable, and responsive Java in less time**, Thomas Nield, Packt Publishing, 2017
- **Reactive Programming with RxJava**, Tomasz Nurkiewicz and Ben Christensen, O'Reilly, 2017
- **Design Patterns and Best Practices in Java**, Kamalmeet Singh, Adrian Ianculescu, Lucian-Paul Torje, Packt Publishing, 2018