

# KONKURENTNÉ PROGRAMOVANIE

Cvičenie 11 : Reaktívne programovanie 2: Operátory

# Generovanie dát do prúdu Flux (výber)

- ❑ `Flux<String> prúd1 = Flux.just("Hello","world");`
  - ❑ Generovanie statických vymenovaných hodnôt
- ❑ `Flux<Integer> prúd2 = Flux.fromArray(new Integer[]{1, 2, 3});`
- ❑ `Flux<Integer> prúd3 = Flux.fromIterable(Arrays.asList(1, 2, 3));`
- ❑ `Flux<Integer> prúd4 = Flux.range(2020, 5);`
  - ❑ Pošle postupne 5 hodnôt: 2020, 2021, 2022, 2023, 2024
- ❑ `Flux<String> empty = Flux.empty();`
  - ❑ Posiela iba informáciu complete o uzavretí prúdu
- ❑ `Flux<String> never = Flux.never();`
  - ❑ Neposiela absolútne nič
- ❑ `Flux<Integer> prúd5 = Flux.create( sink -> {...})`
  - ❑ `sink.next(x)` posielame ďalšiu hodnotu
  - ❑ `sink.complete()` uzatvára prúd
  - ❑ `sink.error(e)` pošle výnimku
  - ❑ + ďalšie metódy, napr. na odchytenie udalostí
- ❑ `Flux<Integer> prúd6 = Flux.generate( () -> init_state, (state, sink) -> {...})`
  - ❑ Druhá metóda sa volá pri každej požiadavke na ďalšiu hodnotu

# Generovanie dát do prúdu Mono (výber)

- `Mono<String> prúd1 = Mono.just("Hello");`
  - Generovanie statickej hodnoty
- `Mono<String> empty = Mono.justOrEmpty(null);`
  - Ak je null uzavrie prúd, inak generuje hodnotu
- `Mono<String> empty2 = Mono.justOrEmpty(Optional.empty());`
  - Ak `optional.isPresent()`, generuje hodnotu, inak uzavrie prúd
- `Mono<String> prúd2 = Mono.fromCallable(Callable);`
  - Spustí úlohu, keď sa subscribne prúd a odošle výsledok
- `Mono<String> prúd3 = Mono.fromFuture(CompletableFuture);`
  - Úloha spojená s týmto Future je už spustená pred tým a keď vráti hodnotu, odošle sa do prúdu
  - podobne aj `fromCompletionStage(CompletionStage)`
- `Mono<Void> prúd4 = fromRunnable(Runnable)`
  - Keď úloha dobehne, uzavrie prúd
- `Mono<String> err = Mono.error(new RuntimeException("mono error"));`
- `Mono<String> prúd4 = Mono.defer(  
 () -> (podmienka ? Mono.just("x") : Mono.error(new Exception("neplatí podmienka"))));`
  - Vnútorná funkcia vracia Mono, no tá sa spustí, až keď nastane subscribe
  - V tomto príklade sa podmienka overuje až v čase, keď sa urobí subscribe

# Spustenie prúdu: subscribe()

- Metódy spustiteľné na inštancii Mono alebo Flux – na poslednom operátore v kanáli:
  - **subscribe()**
    - Spustíme prúd s požiadavkou na Long.MAX\_VALUE dátových objektov, nespracujeme žiadne prichádzajúce dáta, výnimky ani ukončenie prúdu
  - **subscribe(Consumer<T> dataConsumer)**
    - Spustíme prúd s požiadavkou na Long.MAX\_VALUE dátových objektov, dáta spracujeme v dataConsumer-i, nespracujeme výnimky a ukončenie prúdu
  - **subscribe(Consumer<T> dataConsumer, Consumer<Throwable> errorConsumer)**
    - Spustíme prúd s požiadavkou na Long.MAX\_VALUE dátových objektov, dáta spracujeme v dataConsumer-i, výnimky spracujeme v errorConsumer-i, ignorujeme ukončenie prúdu
  - **subscribe(Consumer<T> dataConsumer, Consumer<Throwable> errorConsumer, Runnable completeConsumer)**
  - **subscribe(Subscriber<T> subscriber)**
    - Spustíme prúd, spracovanie udalostí aj požiadavky na dátové objekty prenechávame na subscriber
    - Najbezpečnejšie je rozšíriť BaseSubscriber<T>, aby sme nezabudli implementovať všetku réžiu požiadaviek a spracovania dát.

# BaseSubscriber

- Môžeme prekryť iba tie metódy, ktoré chceme

```
BaseSubscriber<XXX> subscriber = new BaseSubscriber() {  
    protected void hookOnSubscribe(Subscription s) { }  
  
    protected void hookOnNext(XXX value) { }  
  
    protected void hookOnError(Throwable throwable) { }  
  
    protected void hookOnComplete() { }  
  
    protected void hookOnCancel() { }  
}  
  
nejakýFlux.subscribe(subscriber);
```

Ak požadujeme N dátových objektov zavoláme `s.request(N)`. Prúd môžeme ukončiť cez `subscription.cancel()`

# Iné generátory dát

- knižnice pre MongoDB, Cassandra, Redis, Couchbase, relačné DB (R2DBC)
  - ▣ zasielanie hodnôt zo select-ov
  - ▣ získanie odozvy ukončenia insert-ov, update-ov a ďalších operácií
- WebFlux
  - ▣ reaktívne REST API cez eventsQueue
    - Vlákna nespia, kým dôjdu dáta z pezipistentnej vrstvy
  - ▣ reaktívny WebSocket
- WebClient
  - ▣ reaktívny HTTP klient
- Processory (Sinks)
  - ▣ Keď generovanie dát je závislé od externých udalostí

# Príklad 1

- Stiahnite si z gitu poslednú verziu
- 1. Využite klienta WebFlux na získanie a vypísanie JSONu Star wars filmov získaného z adresy <https://swapi.dev/api/films>

# Prúdové operátory

- Zmena prichádzajúcich dátových objektov
- Odchytávanie udalosti uprostred prúdu
  - ▣ Prichádzajúce dáta a výnimky sú preposielané ďalej
- Delenie a spájanie viacerých prúdov
- Synchronne vrátenie dát
- Práca s časom



# Mapovanie elementov

- `map(x -> zmen(x))`
  - ▣ Vezme dátový objekt a na základe neho vráti zmenený, alebo úplne iný objekt
- `timestamp()`
  - ▣ Vezme objekt `x` a vráti `Tuple.of(long_timestamp, x)`
- `index()`
  - ▣ prevezme `i`-ty objekt `x` z prúdu a vráti `Tuple.of(i, x)`

# Príklad 2

---

- Vypíšte zoznam Star wars filmov

# Filtrovanie elementov

- `filter(x -> podmienka(x))`
  - ▣ Vezme dátový objekt `x`
    - Ak podmienka vráti `true`, pošle ho ďalej, inak ho zahodí
- `ignoreElements()`
  - ▣ Zahodí všetky objekty, pustí len chyby a `complete`
- `takeLast()`
  - ▣ Pustí iba posledný objekt pred udalosťou `complete`
- `elementAt(n)`
  - ▣ Pustí iba `n`-tý objekt prúdu a zavrie prúd
  - ▣ pošle `IndexOutOfBoundsException`, ak prúd je kratší
- `single()`
  - ▣ Ak príde jediný objekt, pustí ho
  - ▣ Ak nepríde objekt a prúd sa uzavrie, pošle `NoSuchElementException`
  - ▣ Ak príde druhý objekt, pošle `IndexOutOfBoundsException`

# Filtrovanie elementov 2

- `take(n)`
  - ▣ Pustí maximálne  $n$  objektov, potom zavrie prúd
- `skip(n)`
  - ▣ Prvých  $n$  objektov zahodí, ostatné objekty púšťa ďalej
- `takeLast(n)`
  - ▣ Po zatvorení vstupného prúdu pošle posledných  $n$  objektov
- `skipLast(n)`
  - ▣ Po prijatí  $(n+1)$ . objektu posiela prvý, po  $(n+2)$ . posiela druhý, ... tak, aby posledných  $n$  neposlal
- `take(Duration)`
  - ▣ Púšťa objekty pokiaľ neubehne daný časový interval od otvorenia prúdu, potom zavrie prúd
- `skip(Duration)`
  - ▣ Počas daného časového intervalu od otvorenia prúdu zahodí všetko a potom už objekty púšťa

# Filtrovanie elementov 3

- `takeUntil(x -> podmienka(x))`
  - ▣ Pokial' platí podmienka, objekty púšť'a, ak vnútorná funkcia vráti `false`, zavrie prúd
- `skipUntil(x -> podmienka(x))`
  - ▣ Pokial' neplatí podmienka, objekty nepúšť'a, po prvom `true` už púšť'a všetky
- `takeUntilOther(Publisher)`
  - ▣ Púšť'a objekty, pokiaľ nepríde prvý objekt z druhého prúdu, potom zavrie svoj prúd
- `skipUntilOther(Publisher)`
  - ▣ Zahadzuje objekty pokiaľ nepríde prvý objekt z druhého prúdu, potom už objekty púšť'a

# Filtrovanie elementov

- `distinct()`
  - ▣ Pošle ďalej iba elementy, ktoré sa v prúde ešte nevyskytli (pozor na pamäť)
  - ▣ `1,1,2,2,2,3,1,2,2,2,3,3,4 -> 1,2,3,4`
- `distinctUntilChanged()`
  - ▣ Z každej podpostupnosti rovnakých objektov pošle iba prvý
  - ▣ `1,1,2,2,2,3,1,2,2,2,3,3,4 -> 1,2,3,1,2,3,4`

# Analýza prúdu

- Všetky tieto metódy vracajú Mono
- `count()`
  - ▣ pošle počet objektov v predchádzajúcom prúde po jeho zatvorení
- `all(x -> podmienka(x))`
  - ▣ Pošle true, ak všetky prvky splnili podmienku
  - ▣ Po prvom prvku, čo ju nespĺnil, zatvorí vstupný prúd a pošle false
- `any(x -> podmienka(x))`
  - ▣ Pošle false, ak všetky prvky nespĺnili podmienku
  - ▣ Po prvom prvku, čo splnil podmienku, zatvorí vstupný prúd a pošle true
- `hasElements()`
  - ▣ Pošle true, po prijatí prvého objektu a zatvorí vstupný prúd
  - ▣ Ak sa vstupný prúd zavrie bez prvku, posiela sa false
- `hasElement(e)`
  - ▣ Pošle true, ak príde rovnaký objekt a zatvorí vstupný prúd
  - ▣ Ak sa prku nedočká a vstupný prúd sa zavrie, posiela sa false

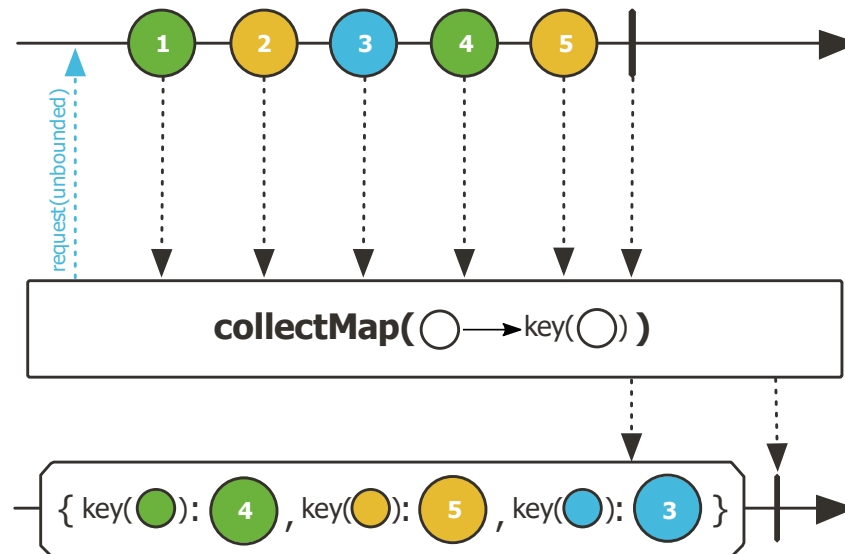
# Kombinovanie vstupných hodnôt

- `sort()` a `sort(comparator)`
  - ▣ Po uzavretí vstupného prúdu, generuje nový výstupný prúd, kde sa posielajú prijaté hodnoty už usporiadané
- `reduce(init_acc, (acc, elem) -> new_acc)`
  - ▣ Pre každý `element` zo vstupného prúdu zavolá druhú funkciu, ktorá vracia nový medzivýsledok `acc`
  - ▣ Prvý parameter je iníciaľna hodnota medzivýsledku `acc`
  - ▣ Výsledkom je `Mono` s poslednou vrátenou hodnotou medzivýsledku
- `reduce((acc, elem) -> new_acc)`
  - ▣ iníciaľna hodnota medzivýsledku je prvý `element`, vnútorná funkcia sa spúšťa až pre druhý `element` vstupného prúdu
- `scan(init_acc, (acc, elem) -> new_acc)` a `scan((acc, elem) -> new_acc)`
  - ▣ Ako `reduce`, len výsledkom je `Flux` so všetkými medzivýsledkami



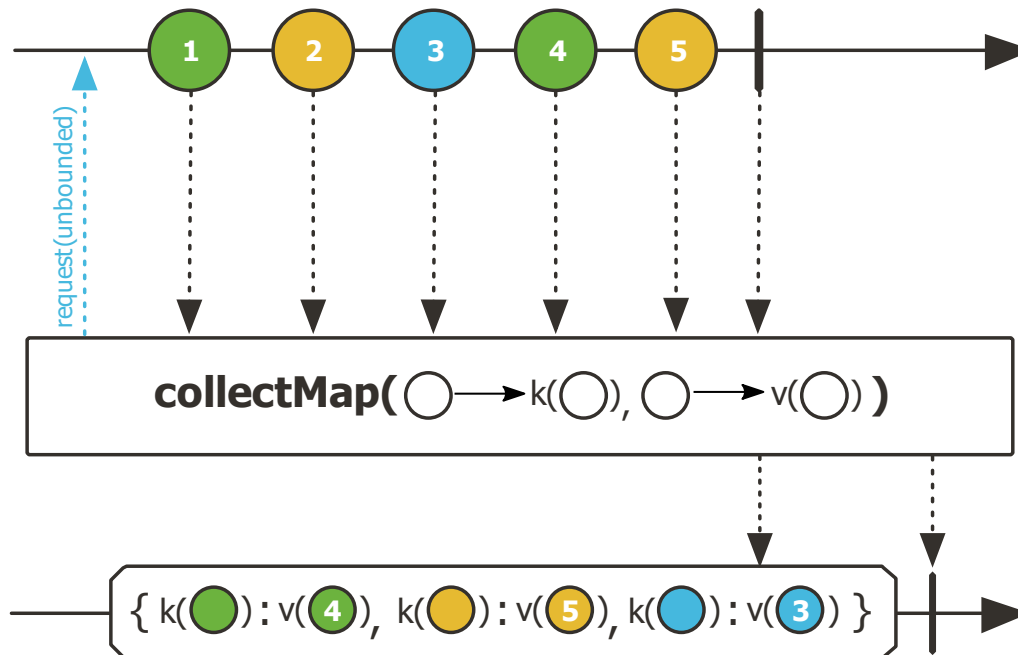
# Vyzbieranie hodnôt do kolekcie

- Meníme Flux na Mono
  - ▣ Keď sa Flux zavrie, operátor posiela jediný objekt – kolekciu s prijatými hodnotami do výstupného prúdu
- `collectMap(x ->vrát'Kľúč(x))`
  - ▣ Výsledná mapa priradí každému kľúču posledný objekt, pre ktorý sme tento kľúč vrátili



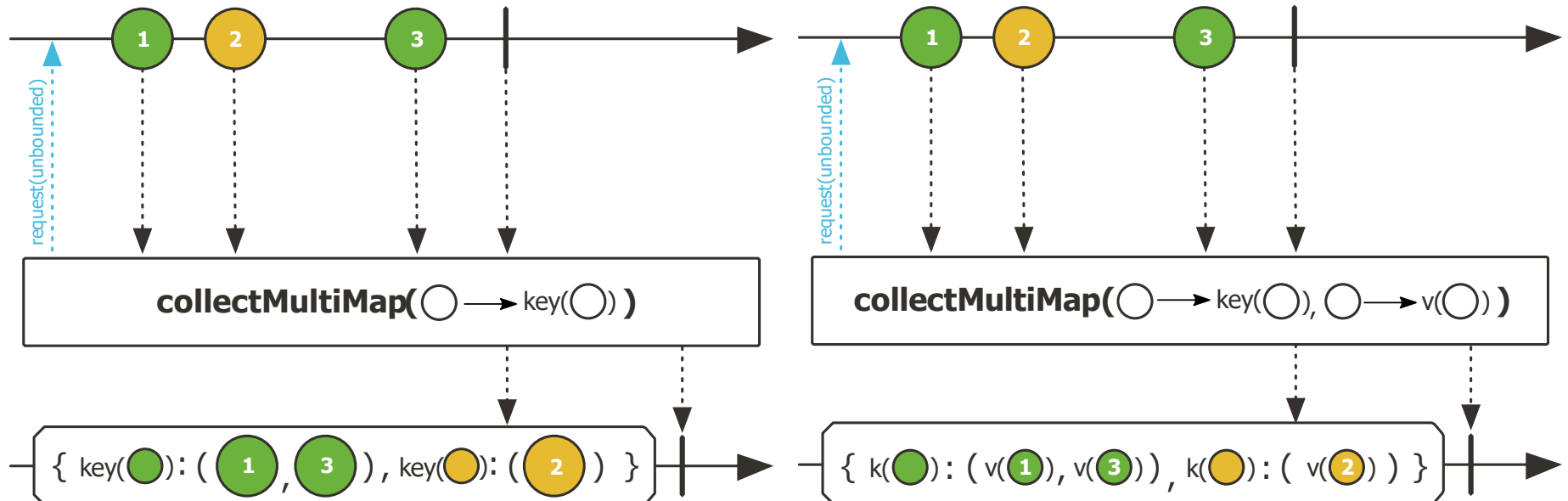
# Vyzbieranie hodnôt do kolekcie 2

- `collectMap(x ->vrát'Kl'úč(x), x->vrát'Hodnotu(x))`
  - ▣ Výsledná mapa priradí každému kľúču poslednú hodnotu z objektu, pre ktorý sme tento kľúč vrátili



# Vybieranie hodnôt do kolekcie 3

- `collectMultiMap(x ->vrát'Kľúč(x))`
- `collectMultiMap(x ->vrát'Kľúč(x), x->vrát'Hodnotu(x))`
  - ▣ Výsledná mapa priradí každému kľúču kolekciu hodnôt z objektov, pre ktoré sme tento kľúč vrátili



# Vyzbieranie hodnôt do kolekcie 4

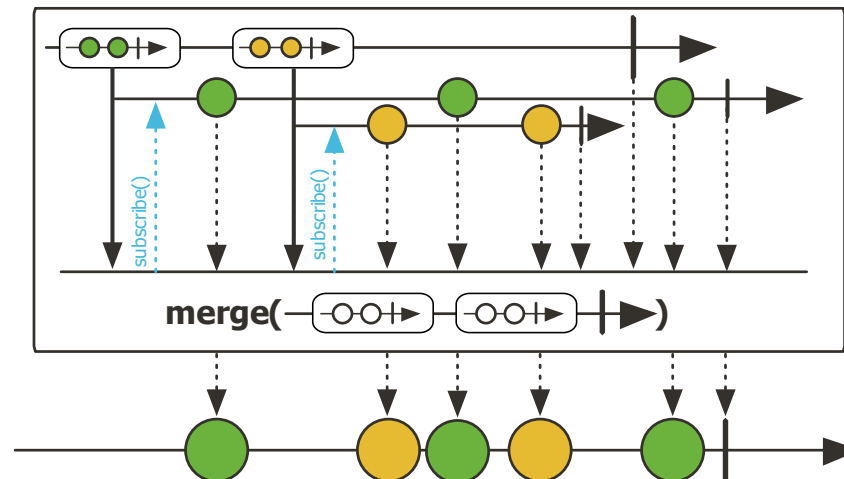
- `collectList()`
  - ▣ Pošle hodnoty ako zoznam
- `collectSortedList()` a `collectSortedList(Comparator)`
  - ▣ Pošle usporiadaný zoznam prijatých hodnôt
- `collect(Collector)`
  - ▣ Použije na vytvorenie kolekcie ľubovoľný kolektor implementujúci `java.util.stream.Collector`
- `collect(() -> init_acc, (acc, value) -> {})`
  - ▣ Pošle object **acc**, keď sa vstupný prúd zavrie
  - ▣ Predpokladá sa, že v druhej funkcii pridávame **value** do **acc**

# Napájanie prúdov

- Tieto operátory ignorujú dáta zo vstupných prúdov, len čakajú, kedy sa zatvoria, aby spustili ďalšie
  - ▣ `Flux.then(Mono)`
  - ▣ `Flux.thenMany(Publisher)`
  - ▣ `Flux.thenEmpty(Publisher)`
  - ▣ `Mono.then(Mono)`
  - ▣ `Mono.thenMany(Publisher)`
  - ▣ `Mono.thenEmpty(Publisher)`
  - ▣ `Mono.thenReturn(value)`
    - Vráti `Mono` s danou hodnotou

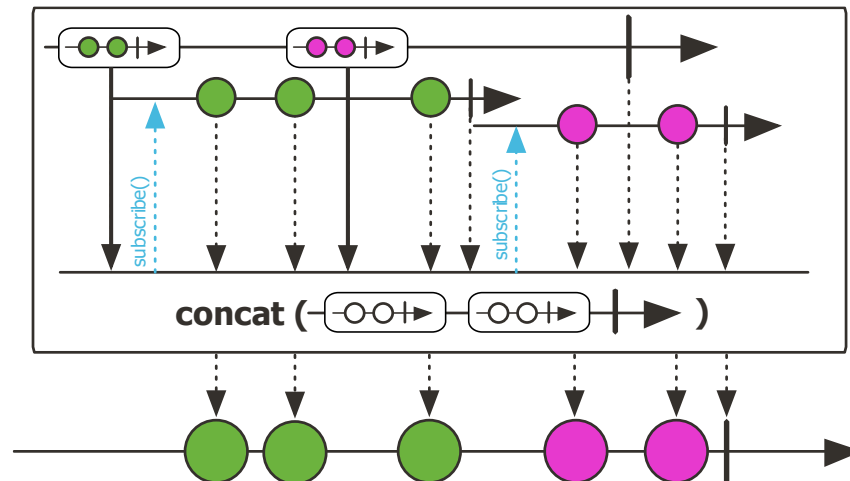
# Spájanie prúdov

- `Flux.merge(listPublisherov)` a `Flux.merge(publisher1, publisher2, ...)`
  - ▣ Spraví `subscribe` na každý kanál, zavrie, keď všetky vstupné zavrú
  - ▣ Čo posielajú vnútorné prúdy, hneď preposielam do výstupného prúdu
- `Flux.merge(PublisherPublisherov)`
  - ▣ Urobím `subscribe` na každý kanál, hneď keď príde
  - ▣ Čo posielajú vnútorné prúdy, hneď preposielam do výstupného prúdu



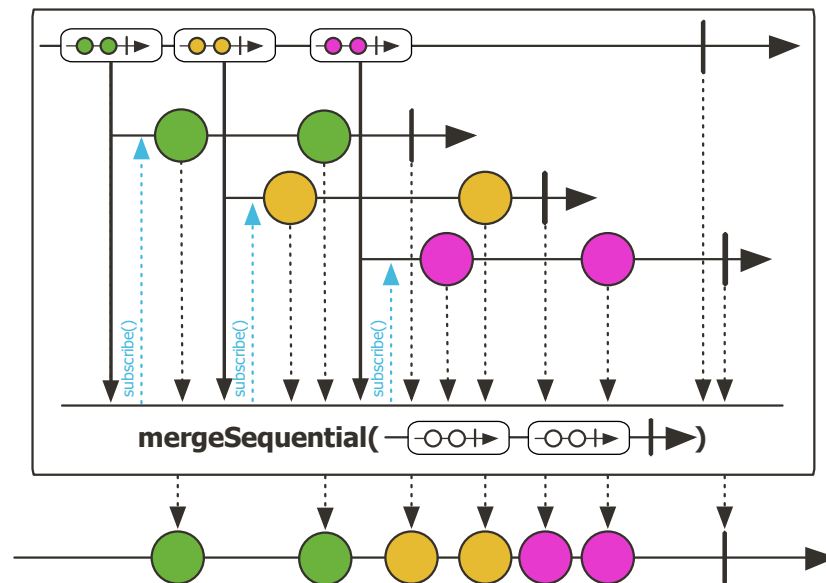
# Spájanie prúdov

- `Flux.concat(listPublisherov)` a `Flux.concat(publisher1, publisher2, ...)`
  - ▣ Postupne robí subscribe na každý kanál, keď sa  $i$ -ty uzavrie, spustí  $(i+1)$ . kanál
  - ▣ Čo posielajú vnútorné prúdy, hneď preposielam do výstupného prúdu
- `Flux.concat(PublisherPublisherov)`
  - ▣ Keď pritečie  $i$ -ty kanál, urobím na ňom subscribe a preposielam dáta z neho na výstup. Keď prúd  $i$ -teho kanála uzavrie, tak urobím subscribe na  $(i+1)$ . kanáli čo priteklo po ňom, alebo naňho "počkám".



# Spájanie prúdov

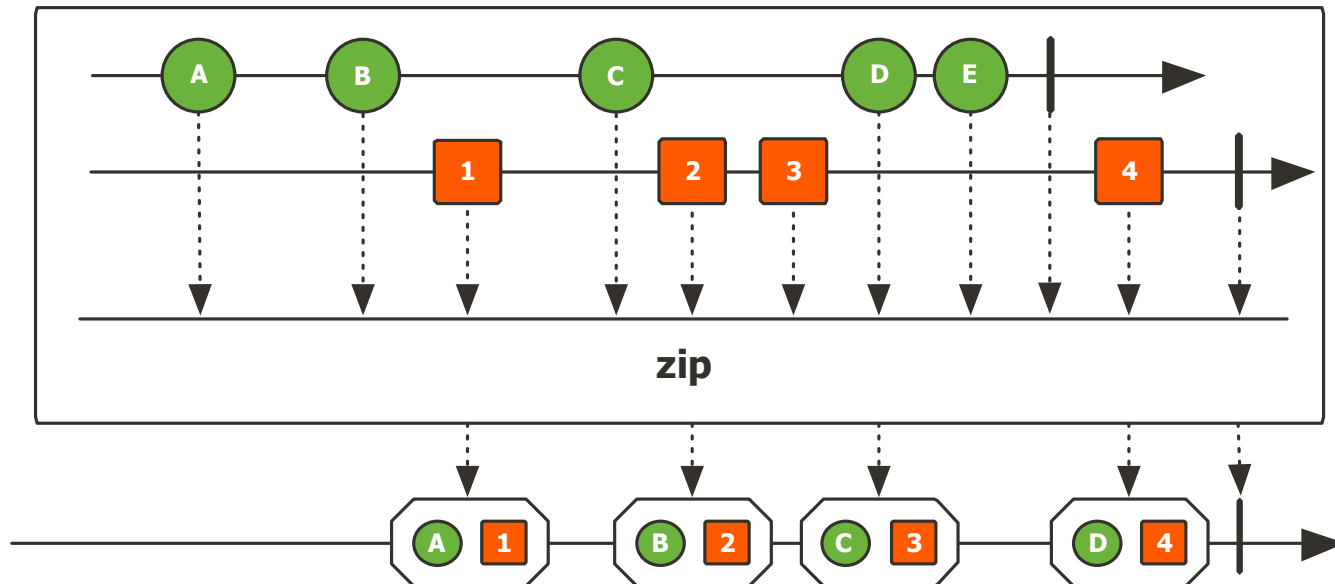
- `Flux.mergeSequential(listPublisherov)` a `Flux.mergeSequential(publisher1, publisher2, ...)`
  - ▣ Spraví `subscribe` na každý kanál, zavrie, keď všetky vstupné zavrú
  - ▣ Dáta z *i*-teho prúdu pošle až keď (*i*-1). prúd zavrie
- `Flux.mergeSequential(PublisherPublisherov)`
  - ▣ Urobím `subscribe` na každý kanál, hneď keď príde
  - ▣ Dáta z *i*-teho prúdu pošle až keď (*i*-1). prúd zavrie





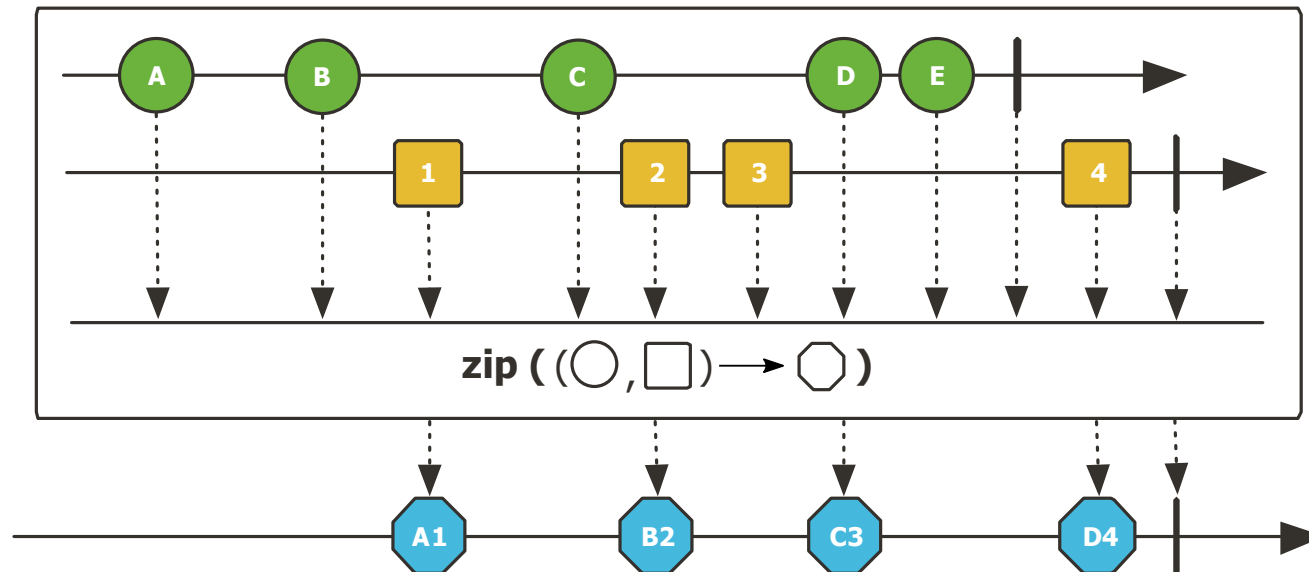
# Spájanie prúdov

- Flux.zip(publisher1, publisher2, ...)
  - ▣ Spraví subscribe na každý kanál, keď prídú i-te objekty zo vstupných prúdov, pošle na výstup objekt typu Tuple



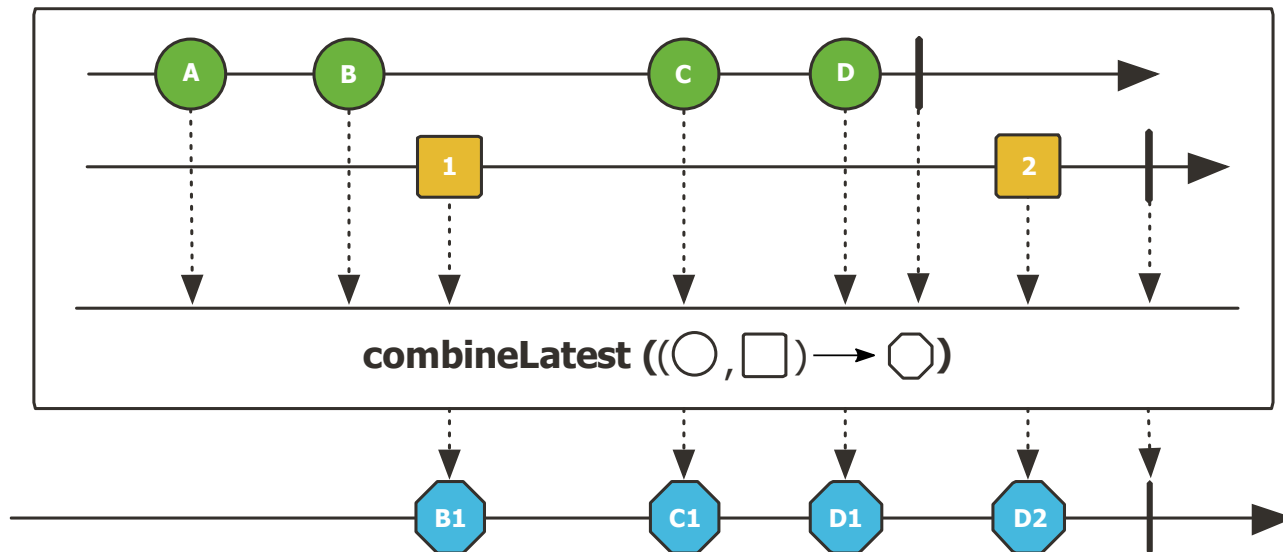
# Spájanie prúdov

- `Flux.zip(listPublisherov, ([v1,v2,..]) -> v)`
  - ▣ Spraví subscribe na každý kanál, keď prídú i-te objekty zo vstupných prúdov, pošle na výstup výsledok vnútornej funkcie



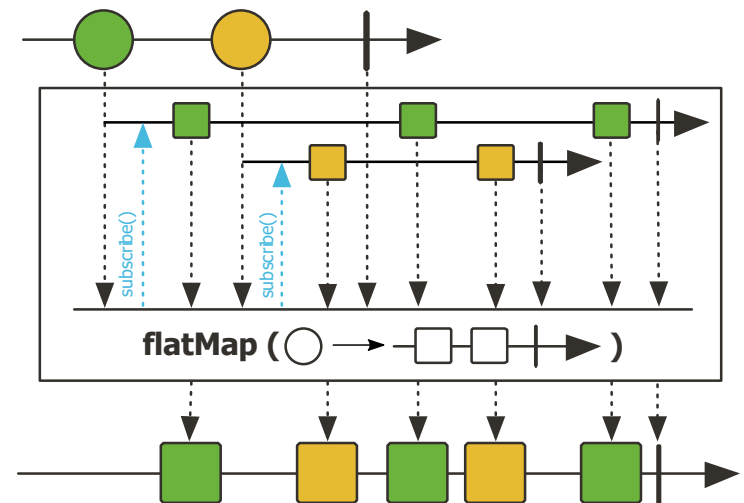
# Spájanie prúdov

- `Flux.combineLatest(listPublisherov, ([v1,v2,..]) -> v)` a
- `Flux.combineLatest(publisher1, publisher2,.., ([v1,v2,..]) -> v)`
  - ▣ Princíp je rovnaký ako pri `zip`, ale posielajú sa n-tice z posledných hodnôt, hneď ako niektorý zo vstupov pošle dáta



# Mapovanie cez prúdy

- Veľmi silný nástroj
- Vieme zaradiť do kanálu viaceré dlhotrvajúce operácie
  - ▣ Čítanie z internetu, databázy, ...
- Napr. ak vstup je prúd používateľských udalostí v GUI, po ktorých potrebujem komunikovať so serverom
  - ▣ Výsledky komunikácie môžem spracovať v mnou určenom poradí
- `flatMap(value -> Publisher)`
  - ▣ S *i*-tou vstupnou hodnotou sa vytvorí *i*-ty vnútorný kanál
  - ▣ *i*-ty vnútorný kanál sa **hned'** `subscribe`-ne
  - ▣ Na hlavný výstupný kanál sa posielajú hodnoty zasielané z vnútorných kanálov



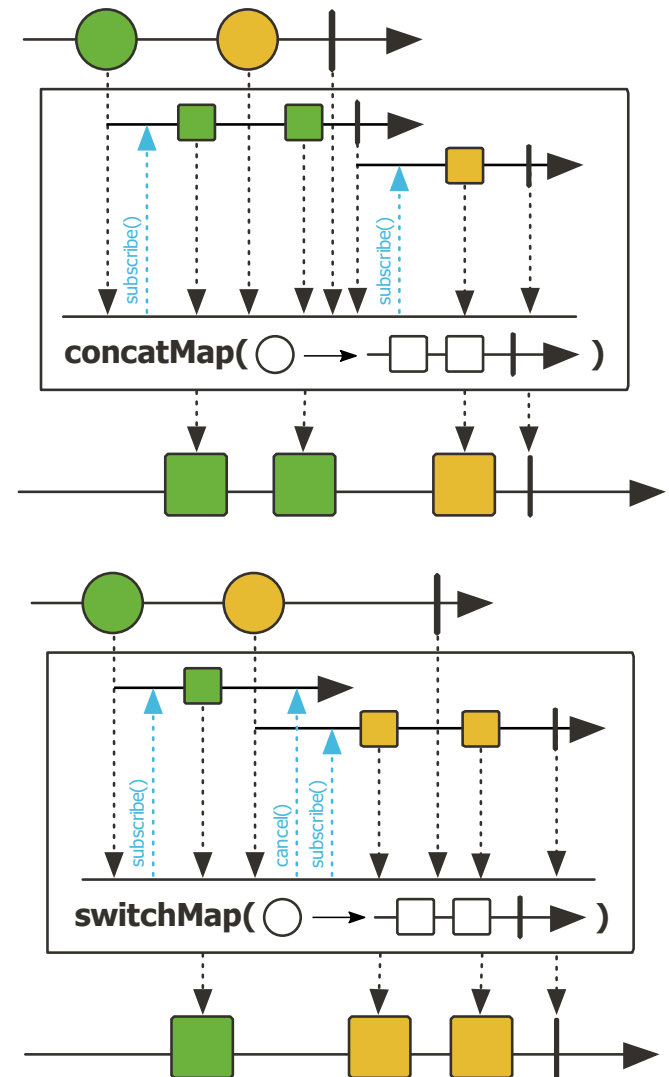
# Príklad 3 a 4

- Vytvorte prúd filmov
  - ▣ Vypíšte čísla epozód Star wars filmov spolu s ich názvami
- Vytvorte prúd postáv z filmu Impérium vracia úder
  - ▣ Vypíšte mená postáv z tohto filmu

# Mapovanie cez prúdy

- `concatMap(value -> Publisher)`
  - ▣ S  $i$ -tou vstupnou hodnotou sa vytvorí  $i$ -ty vnútorný kanál cez danú funkciu
  - ▣  $i$ -ty vnútorný kanál sa subscribe-ne, keď  $(i-1)$ -vý vnútorný kanál zavrie
  - ▣ Na hlavný výstupný kanál sa posielajú hodnoty zasielané z vnútorných kanálov

- `switchMap(value -> Publisher)`
  - ▣ S  $i$ -tou vstupnou hodnotou sa vytvorí  $i$ -ty vnútorný kanál a zruší  $(i-1)$ -vý
  - ▣  $i$ -ty vnútorný kanál potom subscribe-ne
  - ▣ Na hlavný výstupný kanál sa posielajú hodnoty zasielané z vnútorných kanálov



# Delenie prúdov

- `groupBy(value ->keyMapper(value))`
  - ▣ Vracia prúd kanálov
  - ▣ Každý z odoslaných kanálov ešte treba zvlášť subscribe-núť
    - alebo zasa spojiť a subscribe-núť spojenie
  - ▣ Každý kanál, ktorý sa pošle do výstupného prúdu je typu `GroupedFlux`, ktorý má oproti bežnému `Flux` navyše getter na kľúč
  - ▣ Hodnoty zo vstupu si vyrátajú kľúč `K`,
    - Hodnota je preposlaná do výstupného prúdu s kľúčom `K`
    - Ak taký prúd ešte nebol zaslaný vytvorí sa nový, priradí sa mu kľúč `K`, pošle sa doňho daná hodnota
    - Hodnoty sa reálne pošlú do prúdu kanála `K`, až keď je kanál `K` spustený
- `groupBy(value ->keyMapper(value), value->valueMapper(value))`
  - ▣ To isté akurát do výstupných prúdov sa posielajú nie pôvodné hodnoty ale to čo vráti `valueMapper()`

```
private String divisibleBy(long value) {  
    List<Integer> divisors = Arrays.asList(2,3,5,7,11,13);  
    for(Integer divisor: divisors) {  
        if (value % divisor == 0) {  
            return "Divisible by " + divisor;  
        }  
    }  
    return "Others";  
}
```

```
FibonacciFlux.generateFib().take(20).groupBy(value ->divisibleBy(value))  
    .subscribe(stream -> {  
        stream.collectList().subscribe(value -> {  
            System.out.println("key: " + stream.key());  
            System.out.println("data: " + value);  
        });  
    });
```

```
key: Others  
data: [1, 1, 89, 233, 1597, 4181]  
key: Divisible by 13  
data: [13, 377]  
key: Divisible by 3  
data: [3, 21, 987]  
key: Divisible by 2  
data: [0, 2, 8, 34, 144, 610, 2584]  
key: Divisible by 5  
data: [5, 55]
```

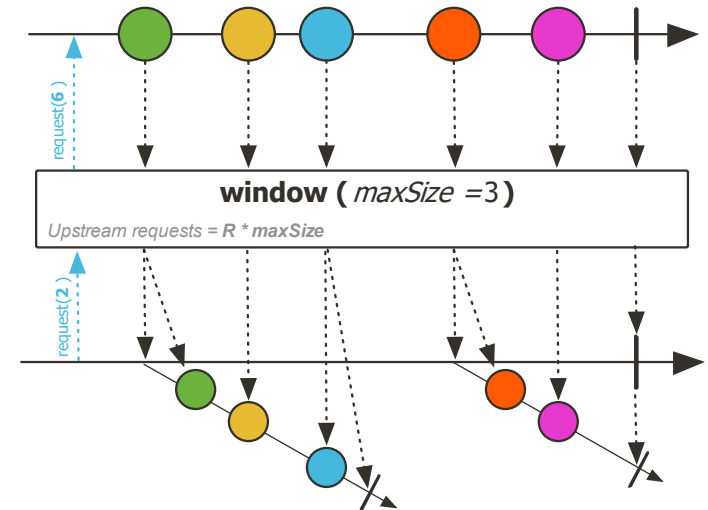


# Dávkové spracovanie

- `buffer()`
  - Po skončení vstupného prúdu vracia Flux, v ktorom pošle jeden List prijatých objektov
  - `a,b,c,d,e,f,g -> [a,b,c,d,e,f,g]`
- `buffer(maxsize)`
  - Po prijatí ďalších `maxsize` prvkov pošle do výstupného prúdu List s týmito prvkami. Ak sa vstupný prúd zavrie, pošle do výstupného prúdu List so zvyšnými prvkami, ktoré ešte neposlal a zavrie
  - `buffer(3)` pre `a,b,c,d,e,f,g -> [a,b,c], [d,e,f], [g]`
- `buffer(maxsize, skipsize)`
  - `Skipsize` hovorí po koľkých vstupných prvkoch začína nový List
  - `buffer(2,3)` pre `a,b,c,d,e,f,g -> [a,b], [d,e], [g]`
  - `buffer(3,2)` pre `a,b,c,d,e,f,g -> [a,b,c], [c,d,e], [e,f,g], [g]`
- `bufferUntil(value -> podmienka)`
  - Ukončí List, keď platí podmienka
  - `bufferUntil(value -> value%2 ==0)` pre `1,2,3,5,6,7,9 -> [1,2], [3,5,6], [7,9]`
- `bufferUntil(value -> podmienka, true)`
  - Začne nový List, keď platí podmienka
  - `bufferUntil(value -> value%2 ==0, true)` pre `1,2,3,5,6,7,9 -> [1], [2,3,5], [6,7,9]`

# Delenie prúdov 2

- `window(maxSize)`
  - ▣ Posiela prúd prúdov
  - ▣ Po prijatí ďalších `maxsize` prvkov vytvorí nový kanál, do ktorého bude preposielať následné prvky
  - ▣ Každý z odoslaných kanálov ešte treba zvlášť `subscribe-núť`
- Analogicky k variantám `buffer` metód fungujú aj
  - ▣ `buffer(maxsize, skipsize)`
  - ▣ `bufferUntil(value -> podmienka)`
  - ▣ `bufferUntil(value -> podmienka, true)`



# Odchytávanie udalosti uprostred prúdu

- **doOnSubscribe**( subscription -> {} )
  - ▣ Na konci kanála bol spustený subscribe
- **doOnRequest**( count -> {} )
  - ▣ Od príjemcu sú požadované dáta
- **doOnNext**( value -> {} )
  - ▣ Cez prúd prítiekol dátový objekt, posielam ho ďalej
- **doOnError**( error -> {} )
  - ▣ Cez prúd prítiekla výnimka, posielam ju ďalej
- **doOnComplete**( () -> {} )
  - ▣ Cez prúd už nepôjdu žiadne dáta, zatvára sa
- **doOnCancel**( () -> {} )
  - ▣ Príjemca už nechce žiadne dáta, posielame info odosielateľovi
- **doOnEach**( signal -> {} )
  - ▣ Nastal subscribe, next, error alebo completion
- **doOnTerminate**( () -> {} )
  - ▣ Nastala chyba alebo completion
- **doFinally**( signalType -> {} )
  - ▣ Nastala chyba, completion alebo cancel
- **doOnDiscard**( Class<T> typZahodeneho, x -> {} )
  - ▣ Ak predchádzajúci operátor podporuje volanie listenera na zahodenie prvku, vieme sa k nemu dostať

# Transformácia na blokované štruktúry

- `blockFirst()`
  - Spraví `subscribe`, zapýta prvok a zaspí. Keď prvok pritečie vráti ho, zavrie prúd. Ak prúd zavrie bez prvku, vráti `null`.
- `blockFirst(timeout)`
  - Zaspí na maximálne `timeout`. Keď prvok pritečie vráti ho, keď nastane `timeout` vyhodí výnimku `RuntimeException`
- `blockLast()`
  - `Subscribe`-ne kanál a zaspí, kým sa prúd nezavrie. Potom sa zobudí a vráti posledný dátový objekt z prúdu, alebo v prípade prázdneho prúdu vráti `null`.
- `blockLast(timeout)`
  - Zaspí na maximálne `timeout`. Keď sa prúd do `timeout`-u nezavrie, vyhodí výnimku `RuntimeException`
- `tolterable()`
  - Automaticky sa spraví `subscribe`
  - S každým volaním `iterator.next()` sa urobí `request(1)` na zaslanie jedného prvku z prúdu. Pokiaľ tento prvok príde cez prúd volajúce vlákno zaspí.
- `tolterable(n)`
  - S každým `n`-tým volaním `iterator.next()` sa urobí `request(n)` na zaslanie maximálne `n` prvkov z prúdu. Pokiaľ chceme `next()` a z prúdu nič nechodí volajúce vlákno zaspí.
- `toStream()` a `toStream(n)`
  - Vyrobit stream z Java Stream API, na pozadí používa `tolterable()`

# Práca s časom (výber)

- `interval(period)`
  - ▣ posiela postupne prirodzené čísla s prestávkami `period`
- `delayElements(delay)`
  - ▣ Po odoslaní každého prvku neposiela ďalší minimálne daný `delay` interval
- `delaySequence(delay)`
  - ▣ Posunie celý prúd v čase – každý prvok je preposlaný presne o `delay` neskôr
- `delaySubscription(delay)`
- `buffer(časZbierania)` a `buffer(časZbierania, novýBufferKaždých)` a `buffer(časZbierania, timer)` a `buffer(maxBufferSize, maxTime)`
  - ▣ Spája dáta, ktoré prišli za ďalší časový interval
- ... podobne s `window`

# Práca s časom (výber)

- `sample(duration)`
  - ▣ Po každom `duration` čase pošle poslednú v ňom prijatú hodnotu, alebo nič
- `timeout(duration)`
  - ▣ Ak cez prúd od naposledy zaslanej hodnoty ubehol daný čas, uzavrie prúd
- `timeout(duration, Publisher)`
  - ▣ Ak cez prúd od naposledy zaslanej hodnoty ubehol daný čas, uzavrie vstupný prúd a namiesto neho urobí `subscribe` na druhý parameter a začne posielat' jeho hodnoty

