

We make IT



JCommerce

Reactor i programowanie reaktywne w Javie

Agenda

- Kilka słów o mnie
- Programowanie reaktywne
- Reactive Streams
- Reactor – ogólnie
- Mono i Flux
- Operatory
- Obsługa błędów
- Współbieżność
- Testowanie
- Backpressure
- Debugowanie
- Kontekst
- Budowa własnej aplikacji
- Pytania i dyskusja

Kilka słów o mnie

Kilka słów o mnie



Marcin Chrost
Senior Java Developer / Technical Leader
[mchrost82\(at\)gmail.com](mailto:mchrost82@gmail.com)
[marcin.chrost\(at\)jcommerce.pl](mailto:marcin.chrost(at)jcommerce.pl)



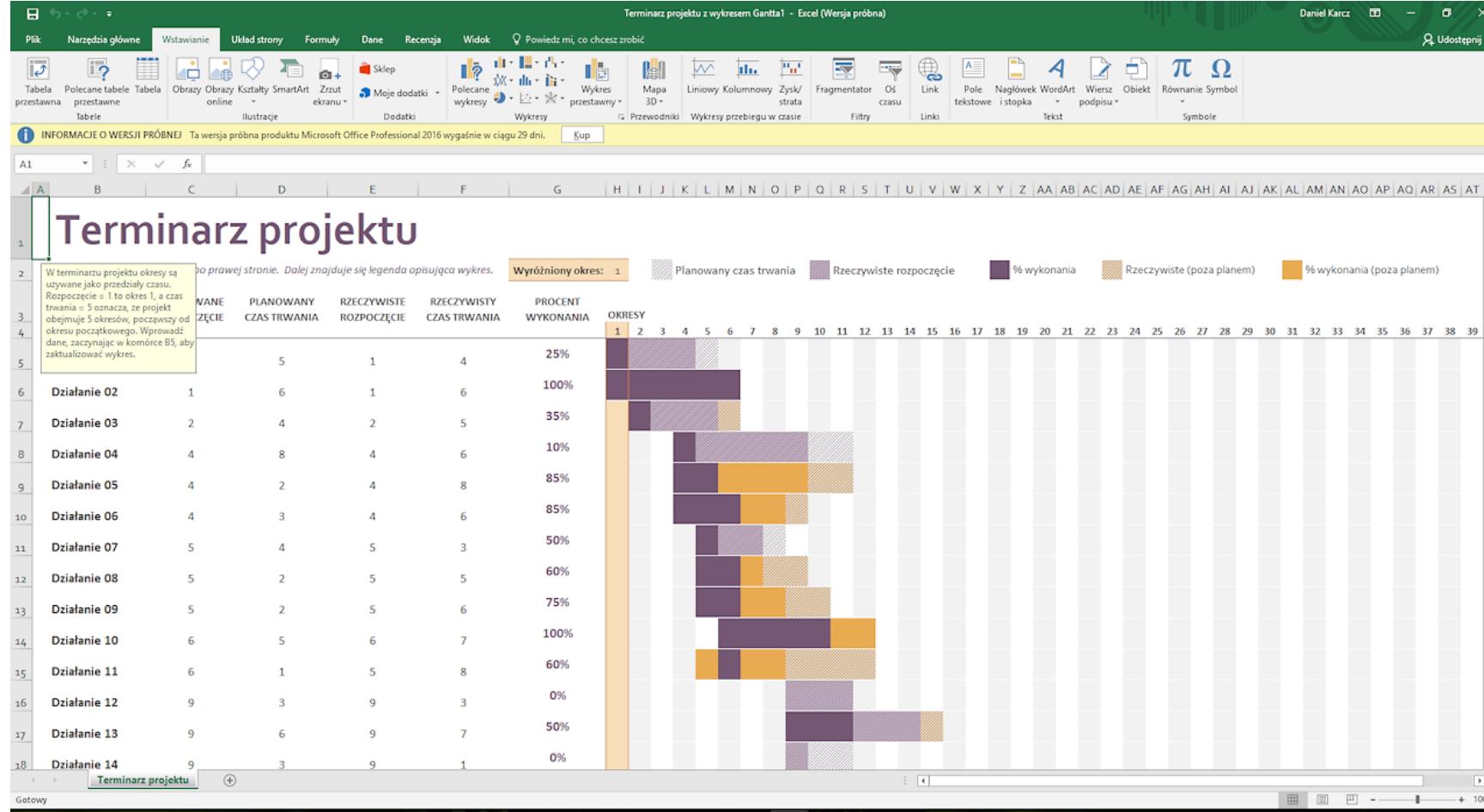
Programowanie reaktywne

Definicja

- Paradygmat programowania zorientowany na przepływ danych i propagację zmian
- Ale co to właściwie znaczy ? 😊

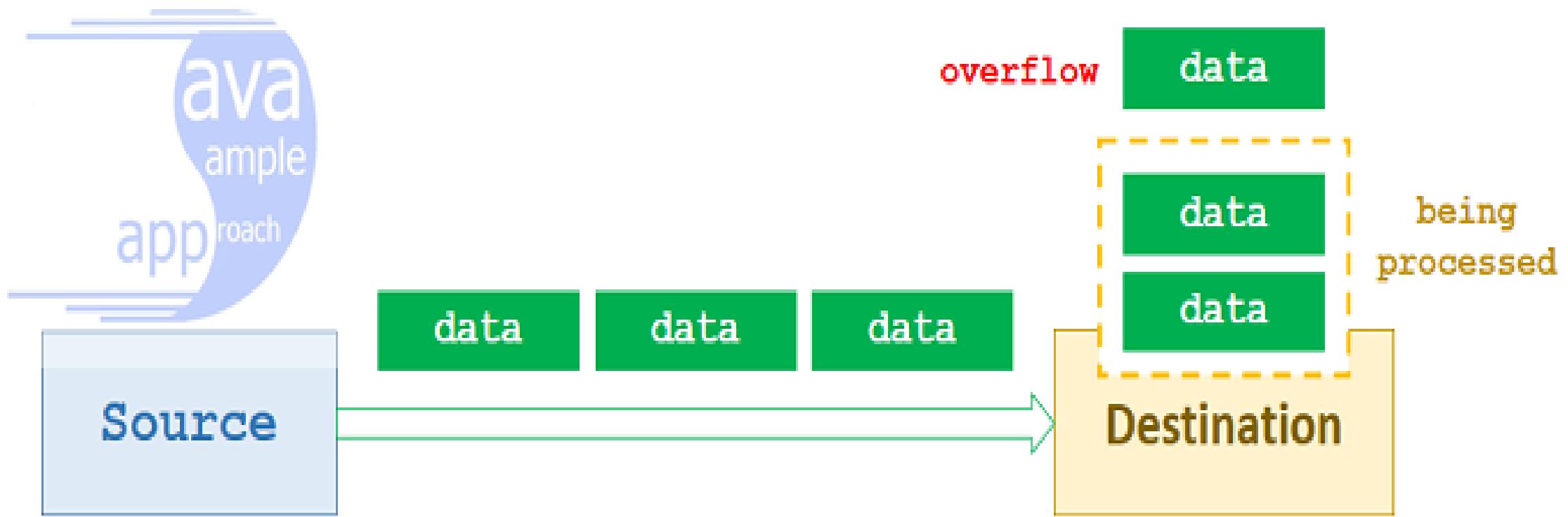


Najprostszy przykład aplikacji reaktywnej ☺



Observable & observer

- RX = Observable + Observer + Schedulers
- RX – Reactive eXtensions
- Observable – w największym skrócie strumienie danych
- Observer – odbiorcy danych produkowanych przez Observable



Przecież to zwykły Publisher / Subscriber !



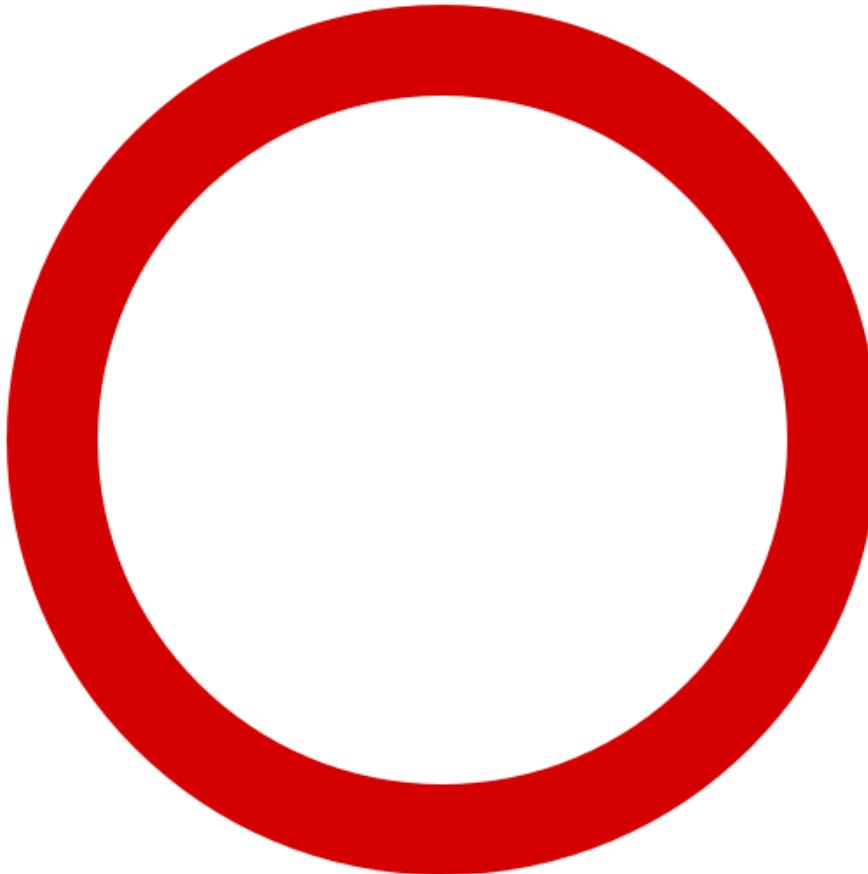
Prawie jednak robi różnicę...



Trzy fundamenty reaktywności

- Samo nie płynie
- Żadnego blokowania
- Backpressure

Samo nie płynie



Żadnego blokowania



A jeśli coś jednak blokuje ?



Wątki & schedulery

- Bez uwzględnienia aspektu wielowątkowości całość wygląda jak klasyczny publisher / subscriber
- Podstawowa zasada – wszystkie operacje MUSZĄ być nieblokujące !!!
- Jeśli coś jednak blokuje (np. sieć / baza) – za pomocą schedulera przenosimy to do oddzielnego wątku / wątków tak aby nie „zanieczyszczać” głównych roboczych wątków programu
- Przy takim założeniu można operować na małej ilości wątków i ciągle przełączać kontekst – czas się nie marnuje.

Backpressure



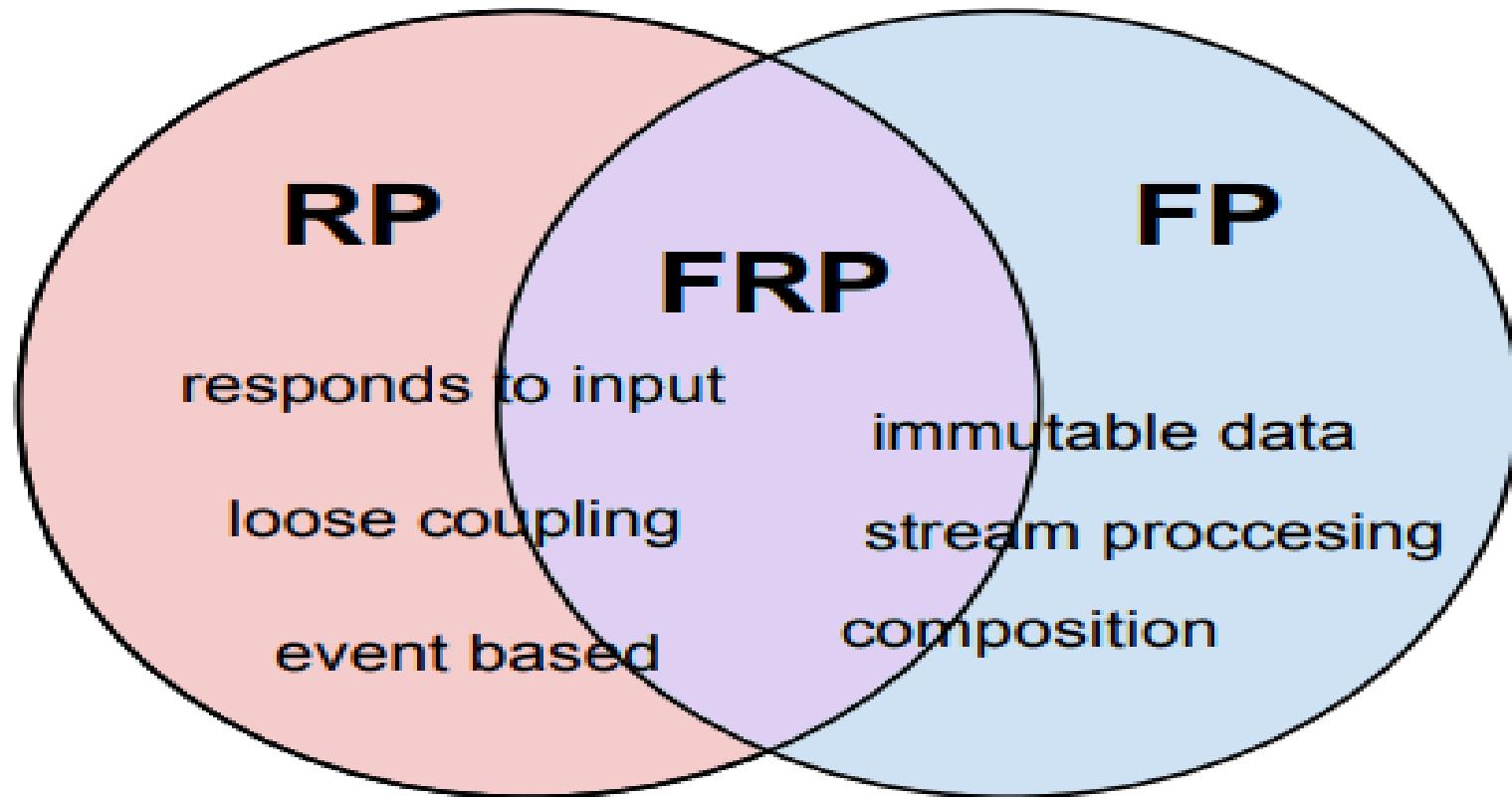
Backpressure

- Jest to możliwość (przynajmniej częściowego) sterowania szybkością pushowania danych przez Observable
- Zastosowanie – szybki producent, wolny odbiorca



Programowanie reaktywne - funkcyjne

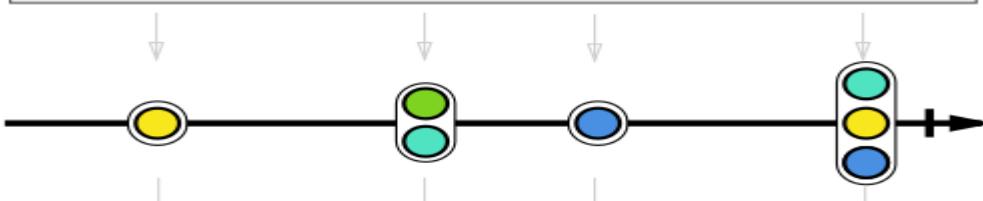
- Dane ze strumienia są przetwarzane poprzez łączenie ze sobą bogatego zestawu operatorów
- Każdy operator jest tak naprawdę nowym Observable, które podłącza się do istniejącego Observable jako Observer 😊. Istniejące Observable pozostaje niezmodyfikowane.
- Jest to rozszerzenie paradymatu programowania reaktywnego – można programować reaktywnie bez podejścia funkcyjnego, ale takie podejście ułatwia sprawę.



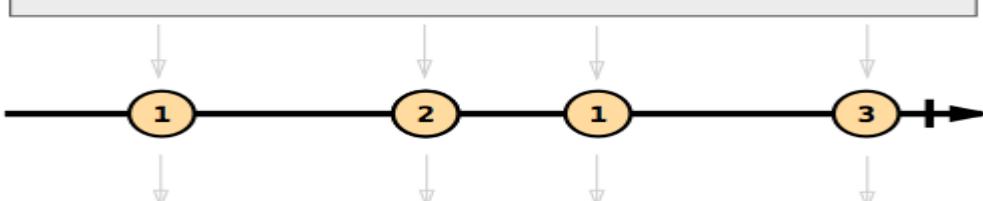
Click stream



```
buffer(clickStream.throttle(250ms))
```



```
map('get length of list')
```



```
filter(x >= 2)
```



Multiple clicks stream

Reactive Streams

Reactive Streams

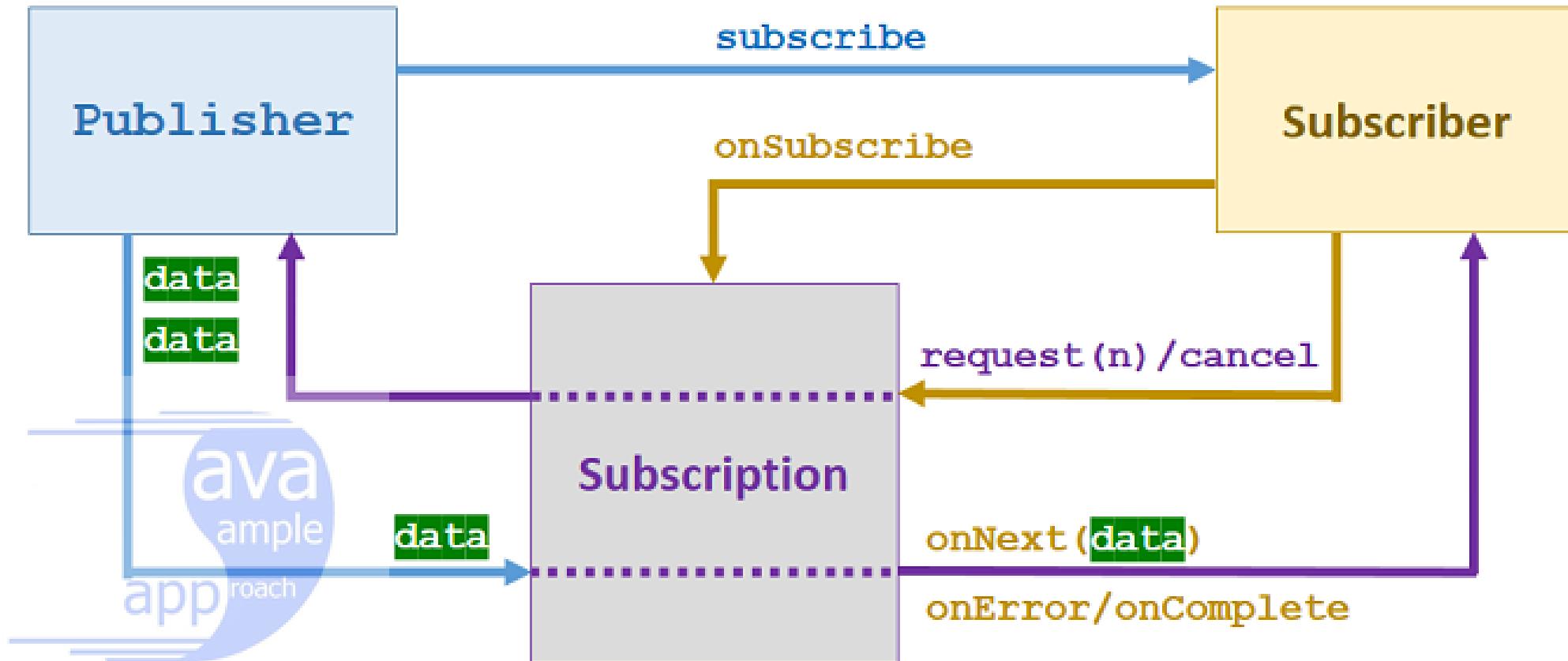
- <https://github.com/reactive-streams/reactive-streams-jvm>
- Próba stworzenia standardu programowania reaktywnego dla JVM
- Stworzone przez konsorcjum firm: m.in. Netflix, Pivotal (Spring)
- Dostarcza:
 - Bazowe Java API (w obecnej chwili 4 interfejsy)
 - Specyfikację
 - TCK (Test Compliance Kit)
- Java 9 inkorporowała Reactive Streams API (klasa `java.util.concurrent.Flow`)

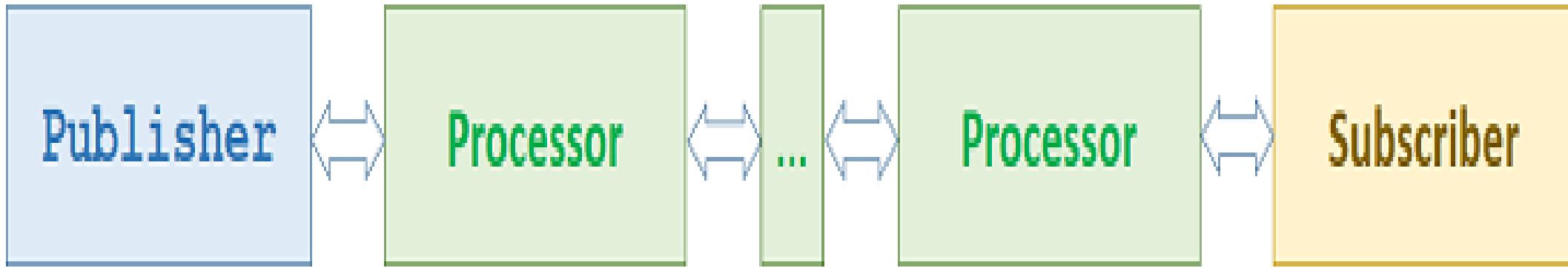
Reactive Streams – API (1)

- Publisher<T> - źródło danych, odpowiednik Observable
 - subscribe(Subscriber<? super T> s)
- Subscriber<T> - odbiorca danych, odpowiednik Observer
 - onSubscribe(Subscription s)
 - onNext(T t)
 - onComplete()
 - onError(Throwable t)

Reactive Streams – API (2)

- Subscription - uchwyt do subskrypcji
 - request (long n)
 - cancel ()
- Processor<T, R> - opcjonalny element pośredni między Observable a Observer, do transformacji danych (będący jednocześnie Publisher<R> i Subscriber<T>)





Reactive streams – dualność

Synchronous / Pull (Iterator)	Asynchronous / Push (Observable)
<code>T next()</code>	<code>onNext(T)</code>
<code>throws Exception e</code>	<code>onError(Throwable t)</code>
<code>returns</code>	<code>onComplete()</code>

Reactive Streams - implementacje

- Rx Java 2
- Reactor
- Akka Streams
- Robienie własnej implementacji nie jest ani łatwe ani zalecane 😊
- Poszczególne implementacje są w podstawowej mierze współpracujące (ze względu na akceptowanie strumieni w postaci interfejsów Reactive Streams)

Reaktor – ogólnie

Reactor - ogólnie

- W pełni nieblokujące środowisko programowania reaktywnego dla JVM
- Jedna z implementacji Reactive Streams
- W pełni wykorzystuje możliwości Javy w wersji 8 (CompletableFuture, Stream, Duration oraz interfejsy funkcyjne)
- Obsługuje backpressure, a także zapewnia jego realizację dla protokołów HTTP, TCP, UDP
- Dostępne również wsparcie dla Kotlina
- Reaktywna podstawa dla frameworka Spring 5

Reactor – przydatne linki

- <http://projectreactor.io> – strona projektu
- <http://projectreactor.io/docs/core/release/reference> – dokumentacja
- <https://github.com/reactor/lite-rx-api-hands-on> - zestaw prostych ćwiczeń pozwalających na zapoznanie się ze Reactorem
- <https://tech.io/playgrounds/929/reactive-programming-with-reactor-3> - j.w.
ale w formie sandboxa w przeglądarce

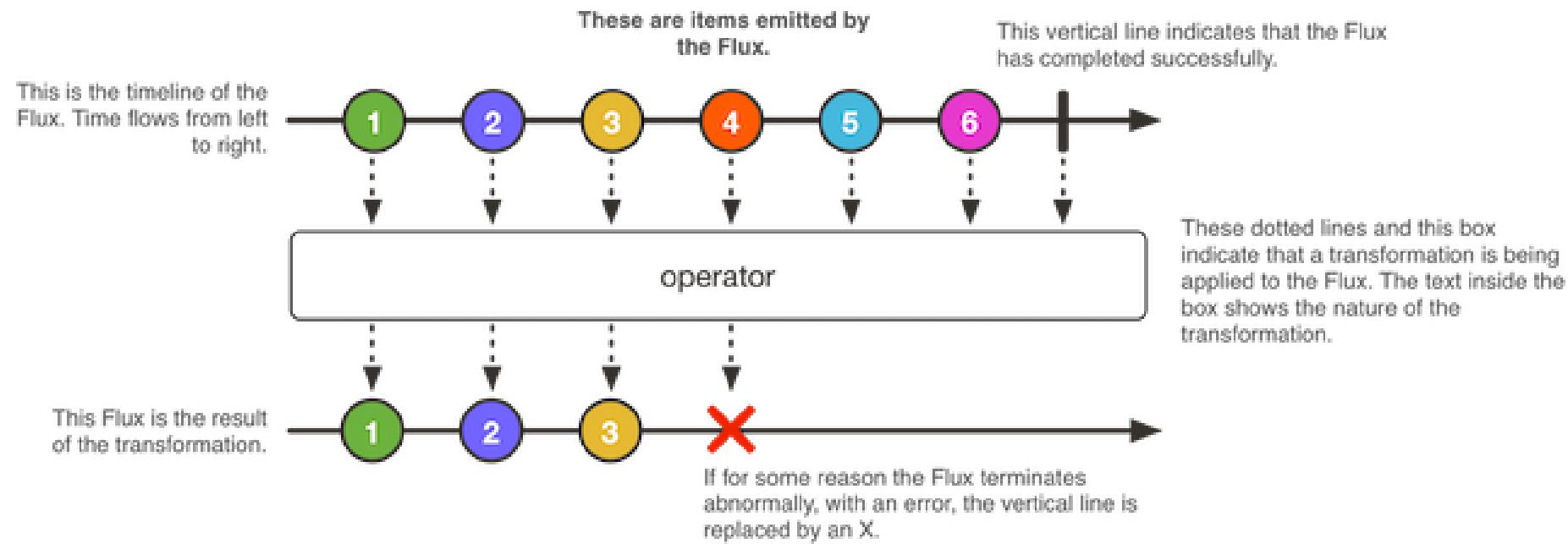
Mono i Flux

Mono i Flux

- Podstawowe klasy Reactor
- Reprezentują Observable (asynchroniczny strumień danych)
- W związku z powyższym implementują Publisher<T>
- Elementy strumienia NIE mogą przyjmować wartości null !

Flux

- Reprezentuje asynchroniczną sekwencję 0..N elementów
- Emitemuje dowolną ilość elementów (dopuszczalne jest też zero)
- Następnie może lecz nie musi zgłosić sygnał końca LUB błąd (ale nie oba naraz):
- `onNext x 0..N [onError | onComplete]`



Flux - tworzenie

- Najprostszym sposobem stworzenia strumienia typu Flux jest użycie jednej z wielu statycznych metod oferowanych przez klasę:
- Przykładowe metody:
- Flux#empty() – tworzy pusty strumień
- Flux#just(T... data) – tworzy strumień z gotowych danych
- Flux#fromIterable(Iterable<? extends T> i) – tworzy strumień na podstawie Iterable
- Flux#error(Throwable t) – tworzy strumień zgłaszający błąd
- Flux#interval(Duration d) – tworzy nieskończony strumień emitujący elementy ze stałą częstotliwością

Flux – tworzenie programowe

- Strumień typu Flux można też stworzyć programowo. Takie podejście jest niezbędne wtedy, kiedy nie mamy możliwości przekazania od razu całej listy wartości jakie ma zwrócić strumień albo jest to po prostu niewygodne.
- Najprostszą metodą do programowego tworzenia strumienia jest Flux#generate(). Pozwala ona na tworzenie synchronicznego strumienia element po elemencie.
- Flux#generate() jako parametr przyjmuje funkcję przyjmującą SynchronousSink. Ta funkcja będzie zwołana w momencie gdy Reactor zgłosi zapotrzebowanie na kolejny element strumienia, który tworzymy za pomocą otrzymanego SynchronousSink możemy wysłać kolejny element (metoda next()), dać znać że strumień się skończył lub też zgłosić błąd.
- Ważna uwaga: metodę next() na przekazanym SynchronousSink można zwołać najwyżej raz.
- W praktyce używa się rozszerzonej wersji metody Flux#generate pozwalającej przekazywać kontekst pomiędzy poszczególnymi wywołaniami.

Flux - tworzenie programowe

```
Flux<String> flux = Flux.generate(() -> 'A', (character, sink) -> {  
    sink.next(Character.toString(character));  
    if ('Z' == character) {  
        sink.complete();  
    }  
    return (char) (character + 1);  
} );
```

Flux – tworzenie programowe

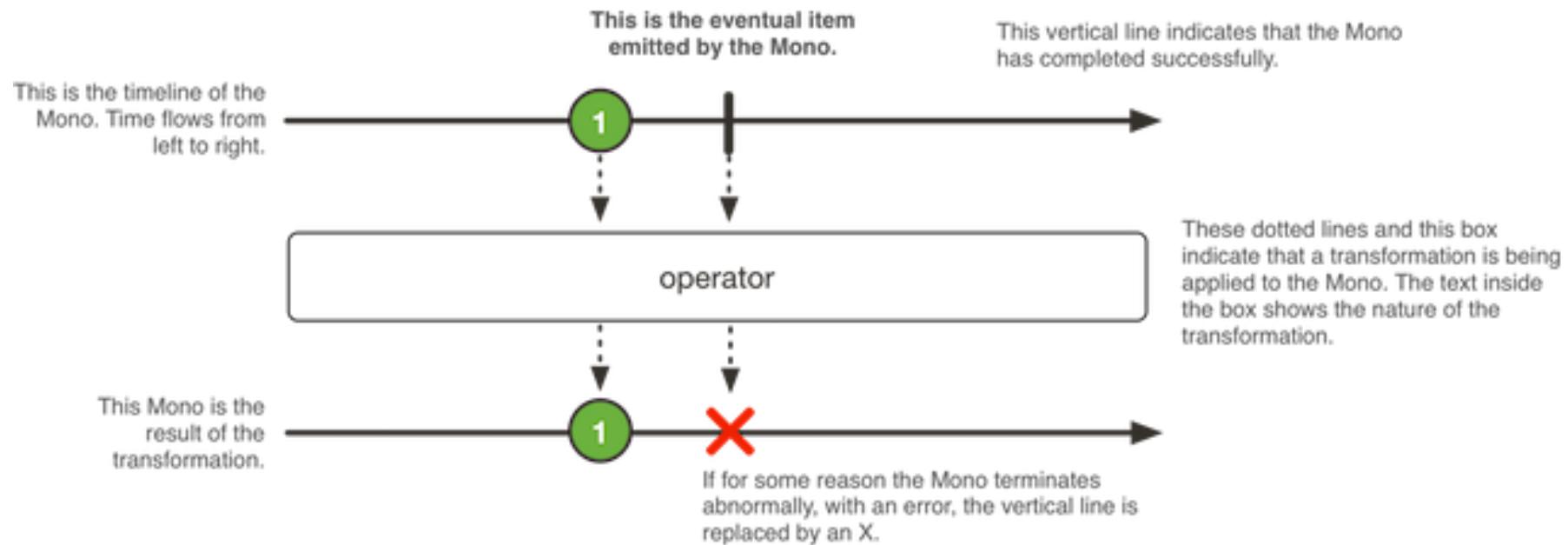
- Bardziej złożoną metodą pozwalającą programowo stworzyć strumień typu Flux jest Flux#create. Za jej pomocą możemy tworzyć strumienie zarówno synchroniczne jak i asynchroniczne, wysyłając jednocześnie wiele elementów naraz
- Idea działania tej metody jest bardzo podobna do Flux#generate - ale tutaj do dyspozycji zamiast SynchronousSink otrzymujemy FluxSink
- FluxSink w przeciwieństwie do SynchronousSink pozwala na:
 - zwołanie next () dowolną ilość razy
 - zwołanie next () z dowolnego wątku
 - zwołanie next () jednocześnie z kilku wątków
- Dzięki takiemu podejściu Flux#create pozwala nam podłączyć się do istniejących event handlerów i zamienić je na pełnoprawny Flux.

Flux - tworzenie programowe

```
Flux<String> flux = Flux.create(sink -> {
    new Thread(() -> {
        IntStream.rangeClosed('A', 'Z')
            .forEach(i -> sink.next(Character.toString((char)i)));
        sink.complete();
    }).start();
}) ;
```

Mono

- Działa analogicznie jak Flux, ale z jedną zasadniczą różnicą – może wyemitować maksymalnie jeden element:
- [onNext x] [onError | onComplete]



Mono - tworzenie

- Podobnie jak w przypadku Flux, również klasa Mono oferuje kilka statycznych metod pozwalających utworzyć strumień typu Mono:
 - Mono#empty () – tworzy pusty strumień
 - Mono#just (T data) – tworzy strumień z jednego elementu
 - Mono#error (Throwable t) – tworzy strumień zgłaszający błąd

Mono / Flux - dualność

Liczebność	Synchronicznie	Asynchronicznie
0	<code>void</code>	<code>Mono<Void></code>
1	<code>T</code>	<code>Mono<T></code>
N	<code>Iterable<T></code>	<code>Flux<T></code>

Ćwiczenia - informacje wstępne

- Proszę sklonować repozytorium:

<https://github.com/chrosciu/lite-rx-api-hands-on>

- Przełączyć się na branch chrosciu-tasks
- Zimportować sklonowany projekt do IntelliJ / Eclipse
- Uruchomić testy w katalogu src/main/tests. Powinny się skompilować ale nie przejść.
- Na warsztatach będziemy po kolej starać się napisać / naprawić kod aby testy przechodziły.
- Rozwiązania są na branchu chrosciu-complete (ale proszę korzystać tylko w ostateczności ☺)
- W razie problemów można alternatywnie wykorzystać playground pod adresem:
<https://tech.io/playgrounds/929/reactive-programming-with-reactor-3>

Ćwiczenie ☺ (1,2,12)

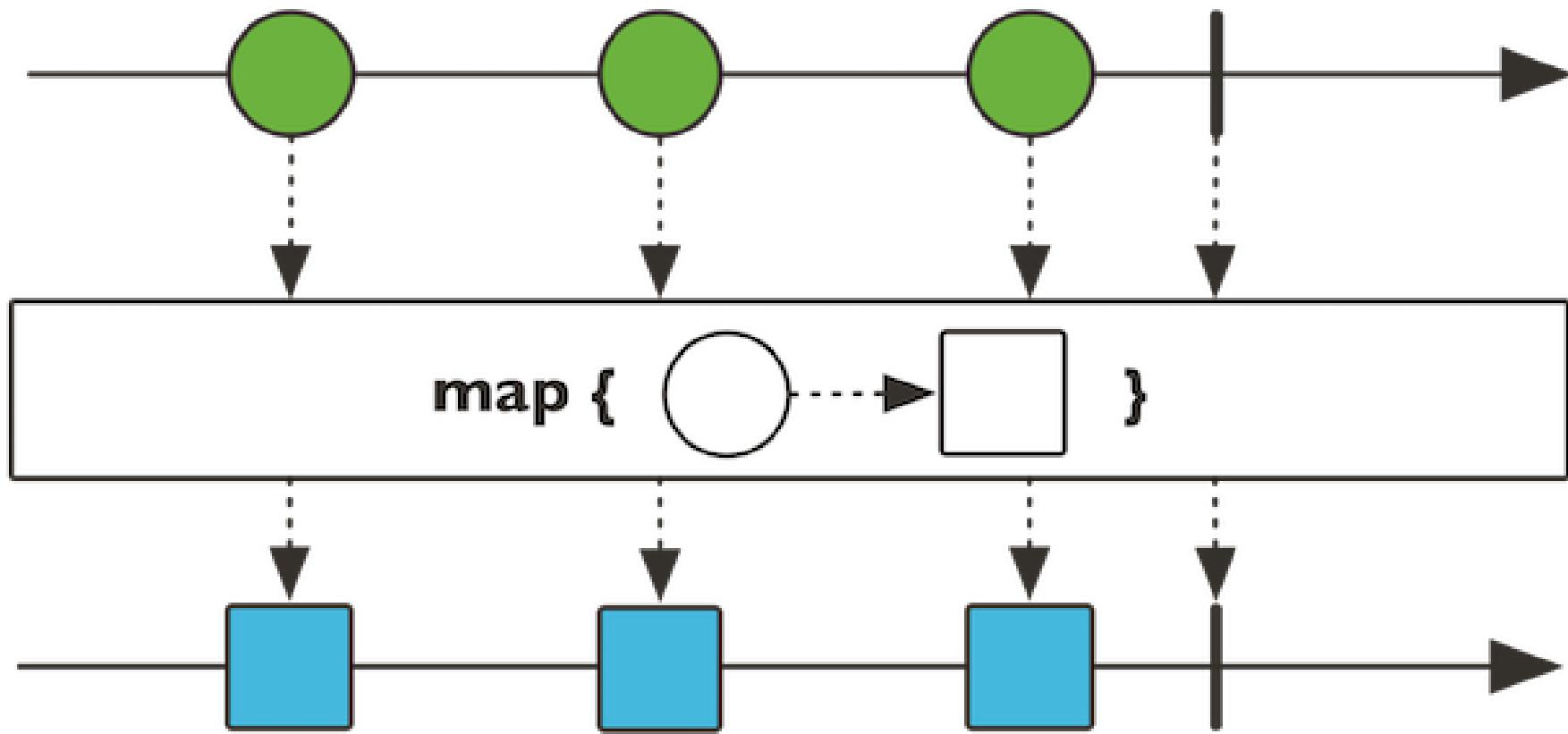


www.shutterstock.com - 153891284

Operatory

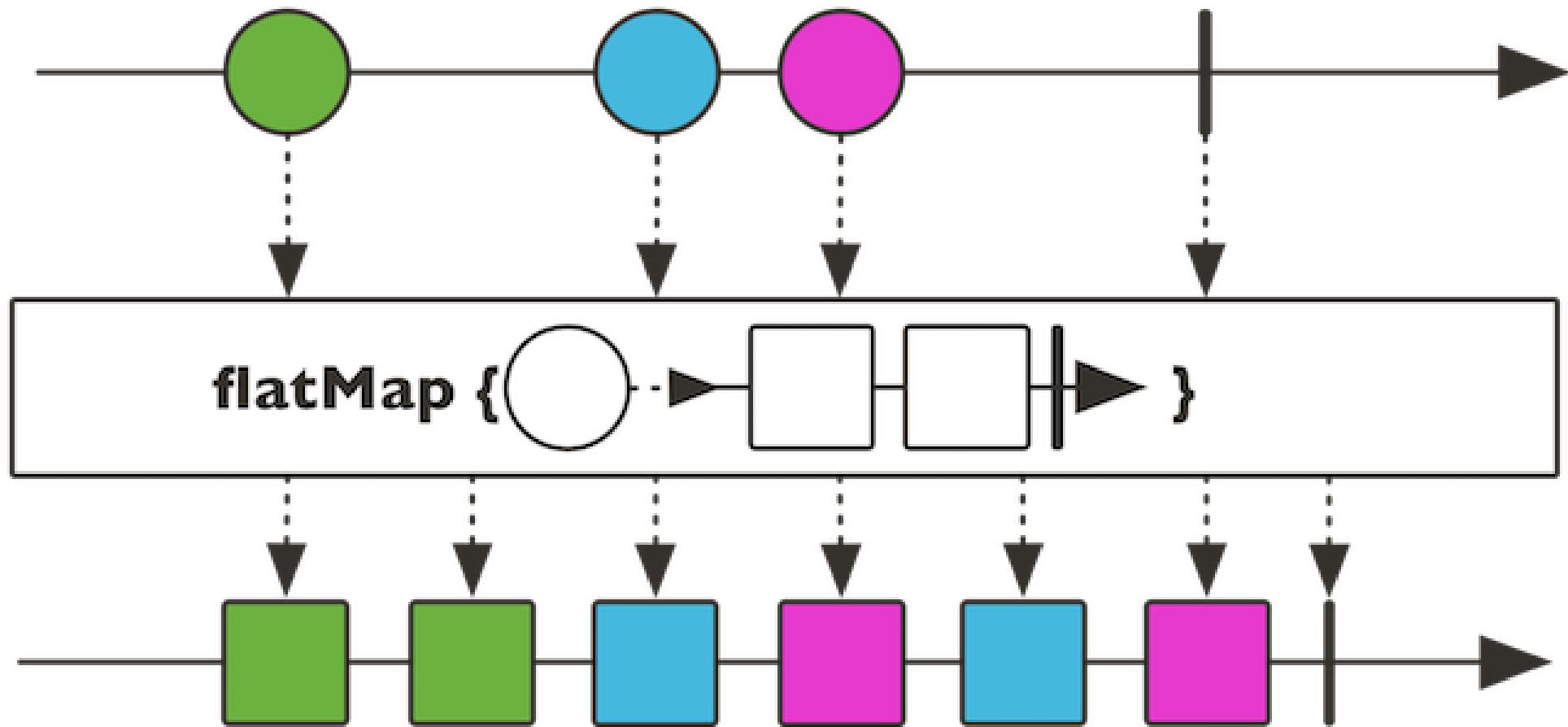
Operatory – podstawowe transformacje

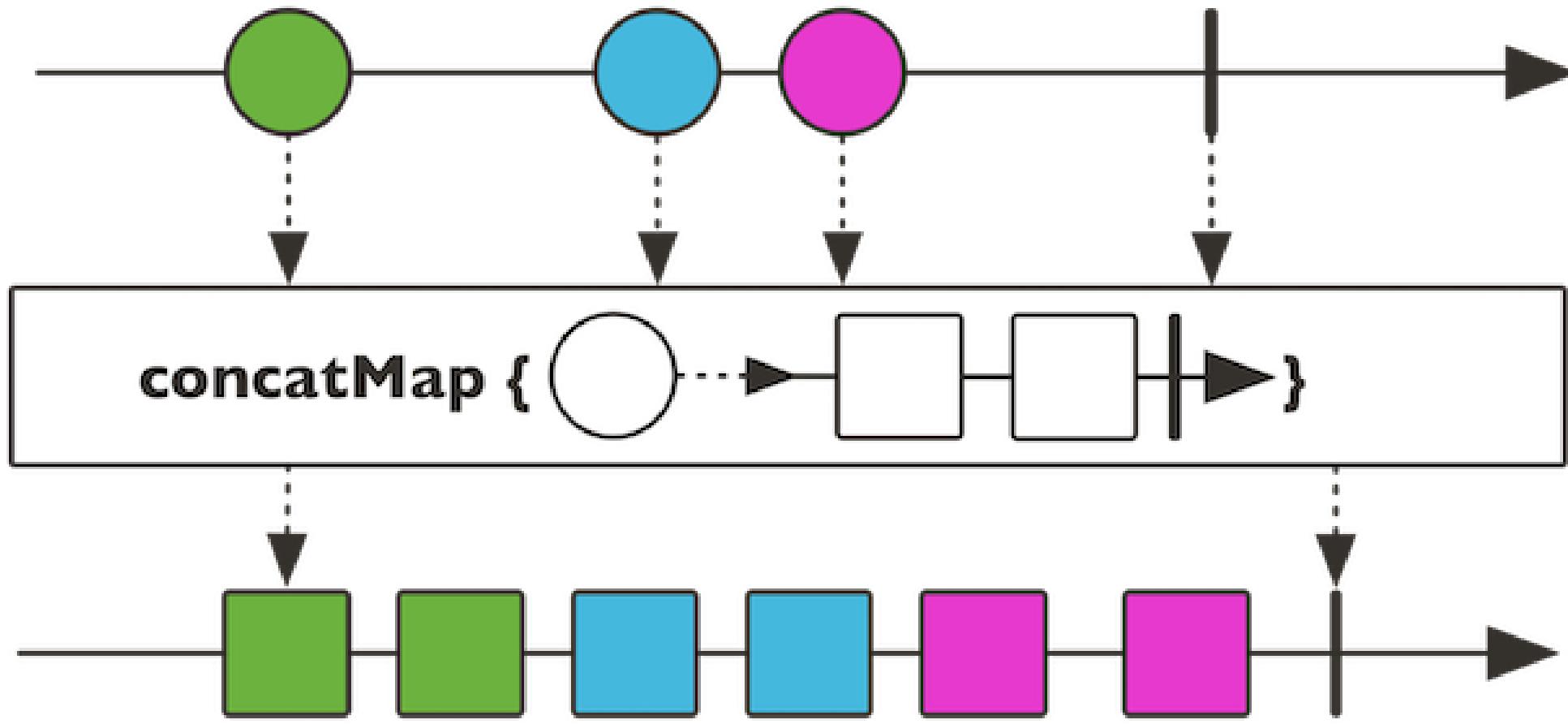
- Działanie operatorów najprościej rozrysować za pomocą „marble diagrams” – często jeden rysunek wart jest tysiąca słów
- Najprostszym operatorem jest `map ()` – przepuszcza każdy element strumienia przez podaną funkcję i zwraca strumień zawierający przetworzone obiekty



Operatory – podstawowe transformacje

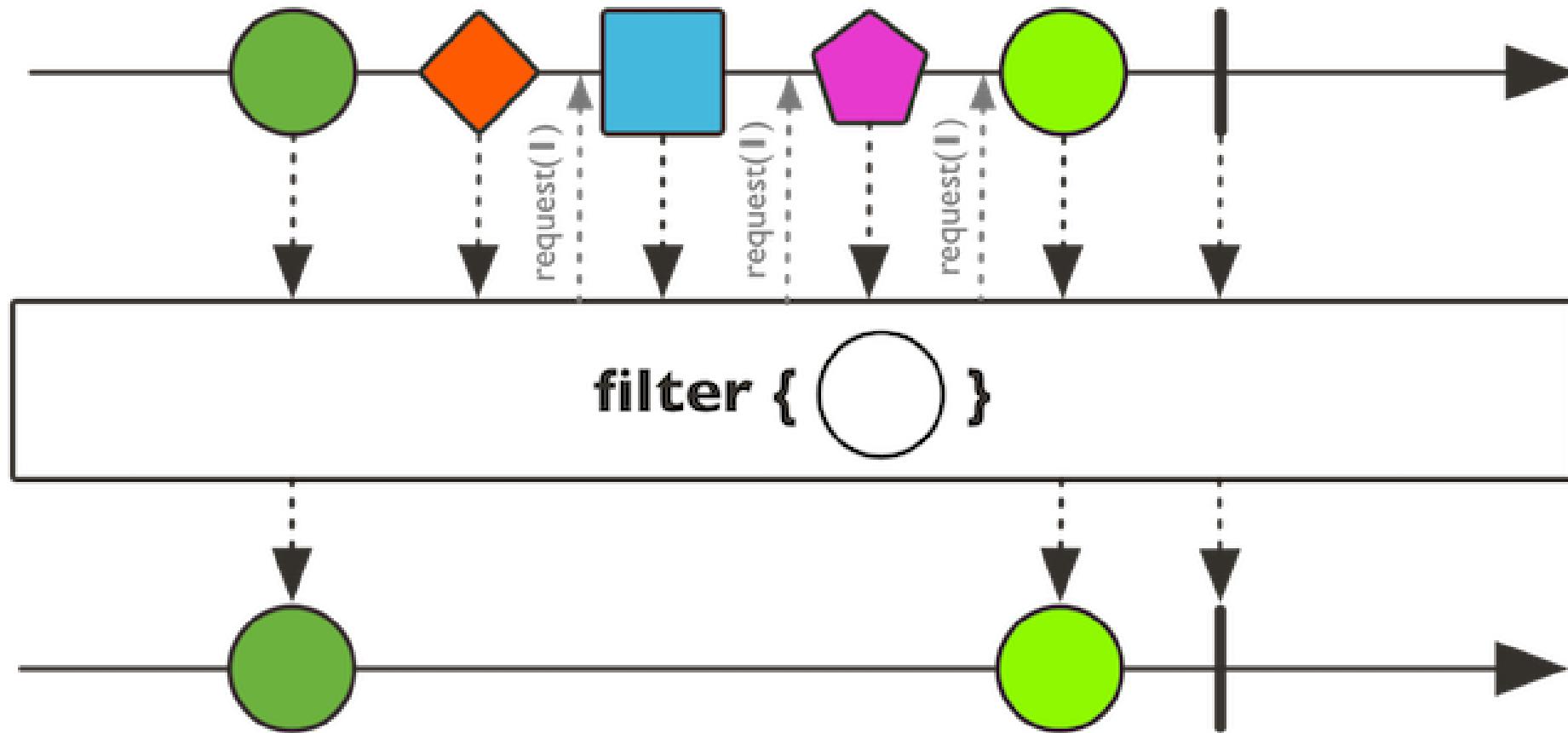
- Bardziej złożonym operatorem jest flatMap()
- Używamy go wtedy gdy funkcja transformująca zwraca nie element lecz strumień elementów (np. jeśli musimy coś wykonać asynchronicznie).
- Zwykły map() wróciłby nam wtedy strumień strumieni, co byłoby ciężkie w użyciu. flatMap() subskrybuje się do wszystkich tych strumieni i łączy je w jeden.
- Zauważmy że wynikowe elementy mogą się przeplatać. Aby tego uniknąć zamiast flatMap() można użyć concatMap()

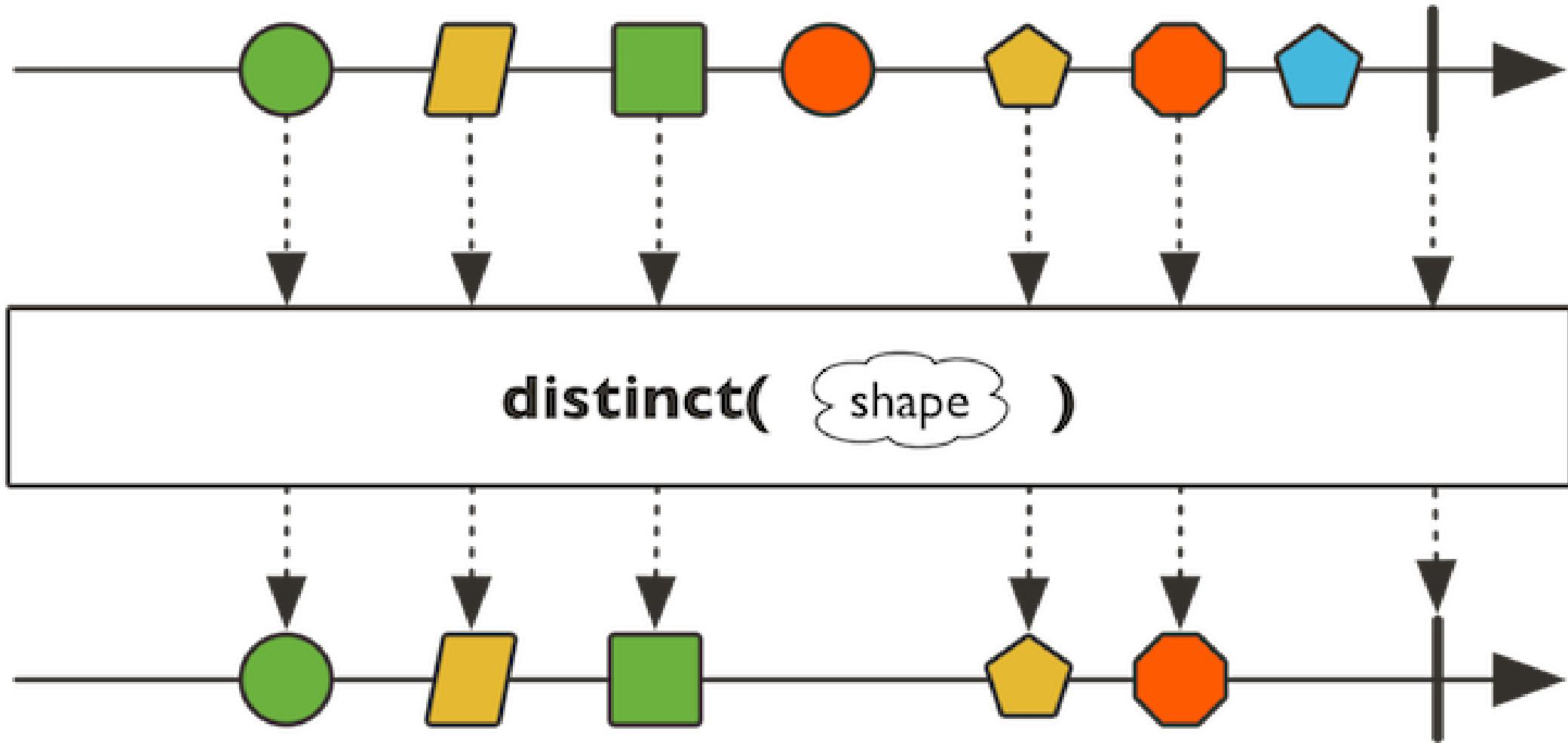


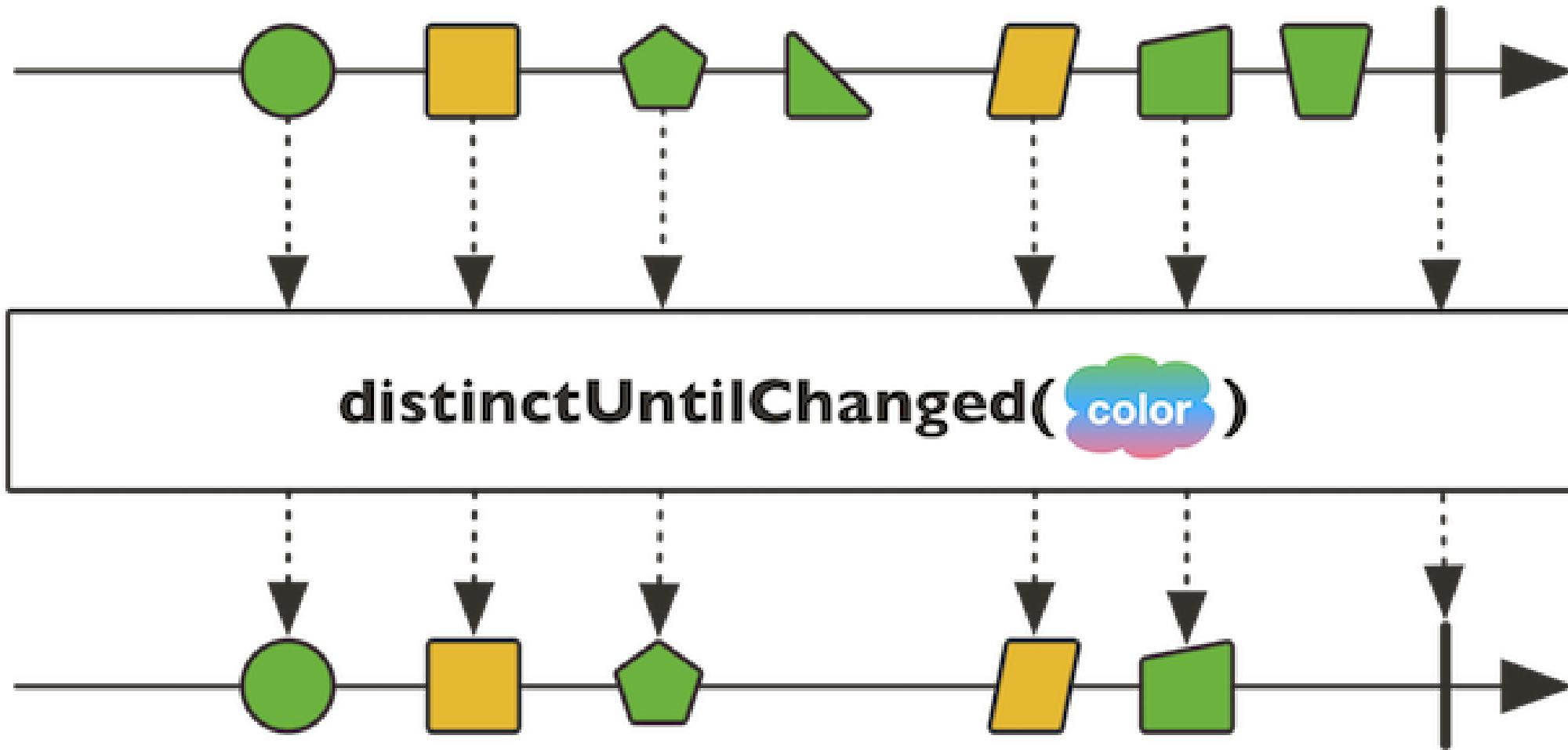


Operatory – filtrowanie elementów

- Podstawowym operatorem pozwalającym na filtrowanie elementów strumienia jest `filter()`. Operator ten przepuszcza każdy element strumienia przez `Predicate` przekazany jako argument operatora i „wycina” te elementy dla których `Predicate` zwraca `false` (w takiej sytuacji zaciągany jest kolejny element, aż do momentu gdy któryś spełni warunek)
- Operator `distinct()` „wycina” duplikujące się elementy ze strumienia. Jego podstawowa wersja bazuje na funkcji `equals()` dla elementów, dlatego jest mało przydatna. Na szczęście istnieje możliwość przekazania własnej funkcji służącej do sprawdzenia czy dwa obiekty są sobie równoważne.
- Odmianą `distinct()` jest `distinctUntilChanged()`, która „wycina” duplikujące się elementy, dopóki w sekwencji nie wystąpi inny element. Również tu można przekazać własną funkcję porównującą elementy



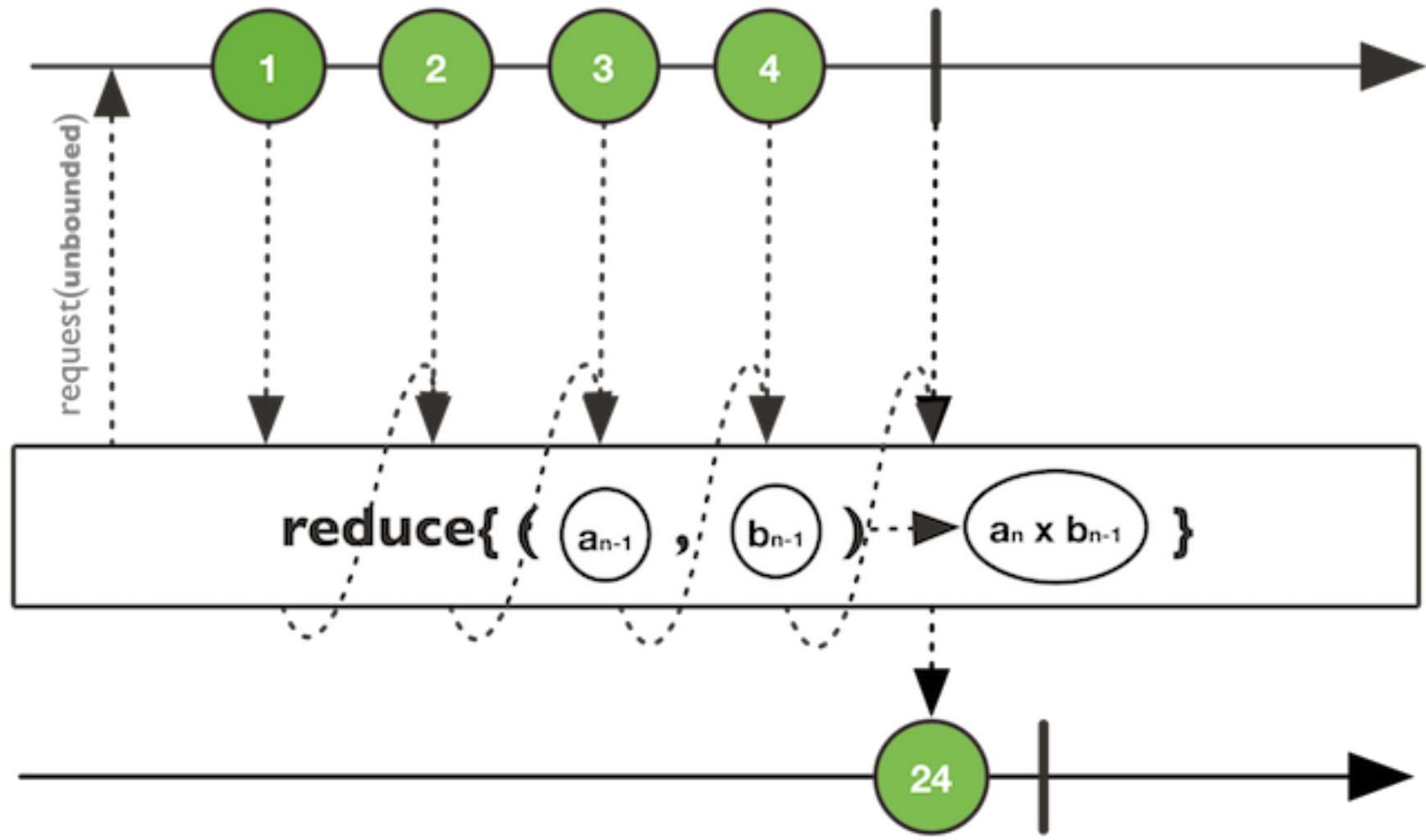


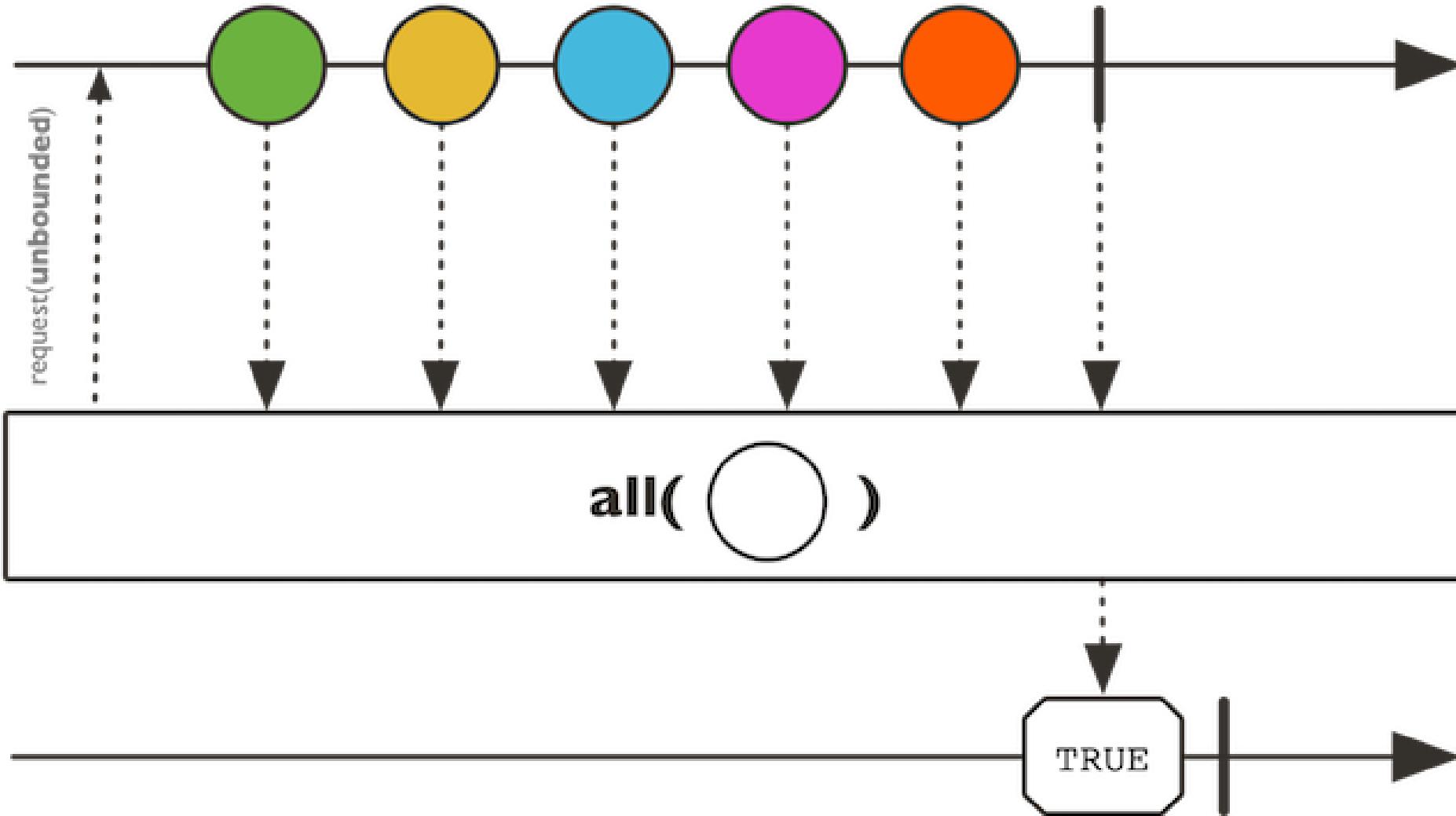


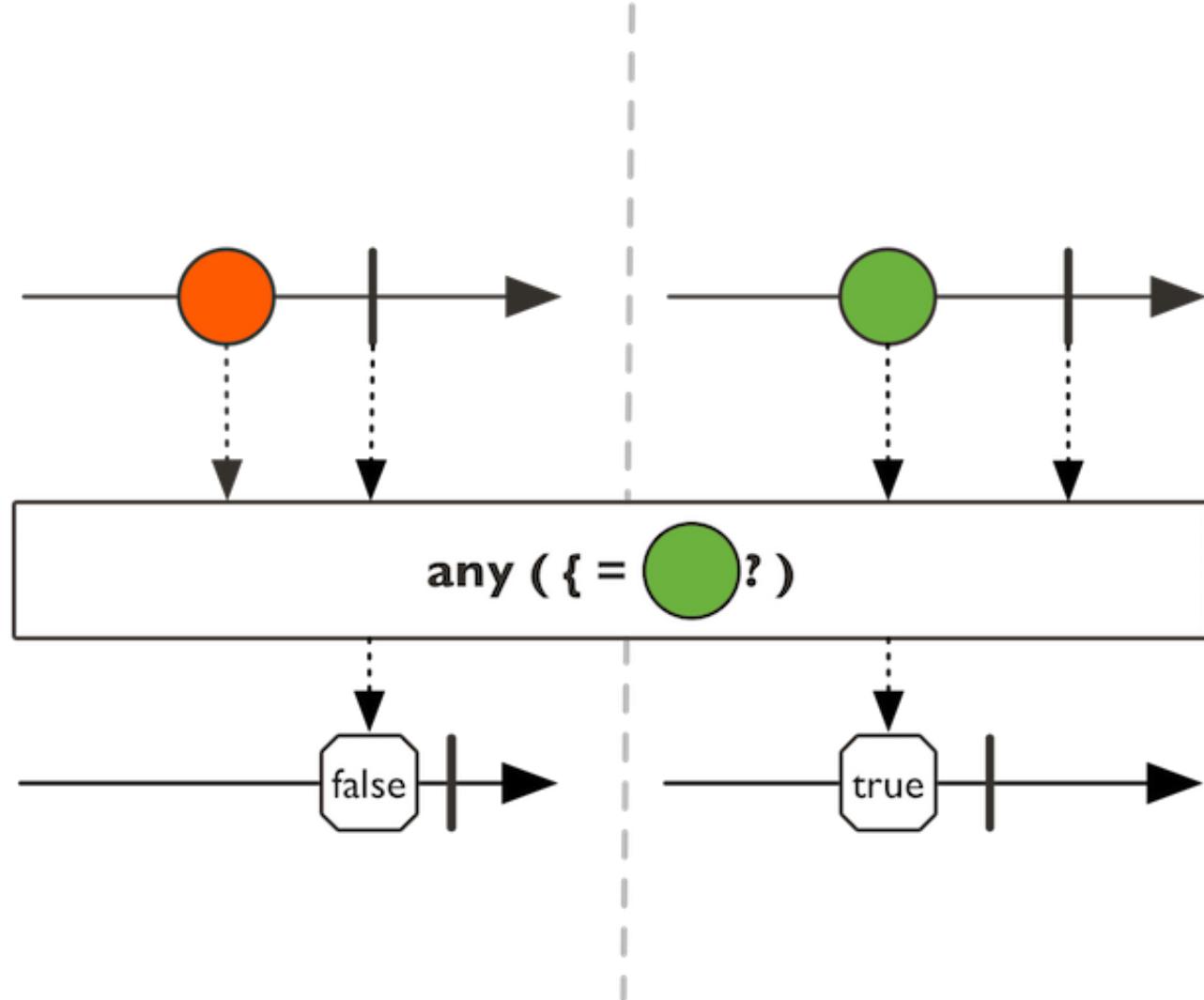
`distinctUntilChanged(<color>)`

Operatory – kombinowanie elementów

- Operator `reduce()` służy do „kombinowania” elementów strumienia Flux w jedną końcową wartość. Jako argument przyjmuje funkcję, która łączy dwa elementy w jeden. Klasycznym zastosowaniem takiego operatora jest np. sumowanie elementów.
- Dostępna jest także wersja z dodatkowym argumentem – inicjalną wartością używaną jako argument przy „kombinowaniu” z pierwszym elementem
- Operator `all()` sprawdza czy wszystkie elementy strumienia spełniają warunek przekazany jako argument operatora (`Predicate`). Używany jest mechanizm *short-circuit* – pierwszy element który nie przechodzi testu, kończy działanie operatora
- Operator `any()` działa podobnie do `all()` ale sprawdza czy chociaż jeden element strumienia spełnia podany warunek
- Wszystkie operatory „kombinujące” zwracają Mono (co w sumie jest logiczne)





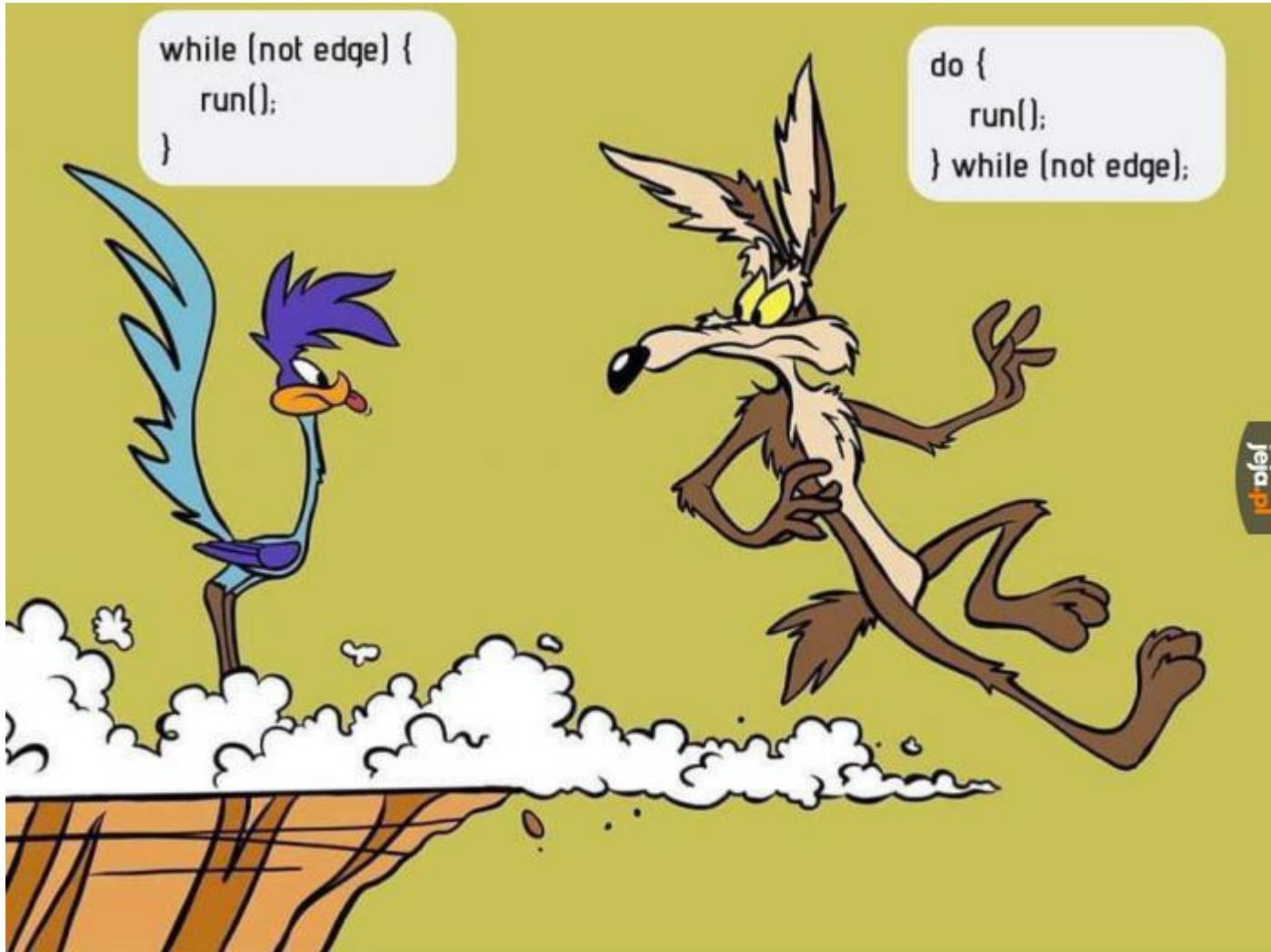


Ćwiczenie ☺ (4,13,15)

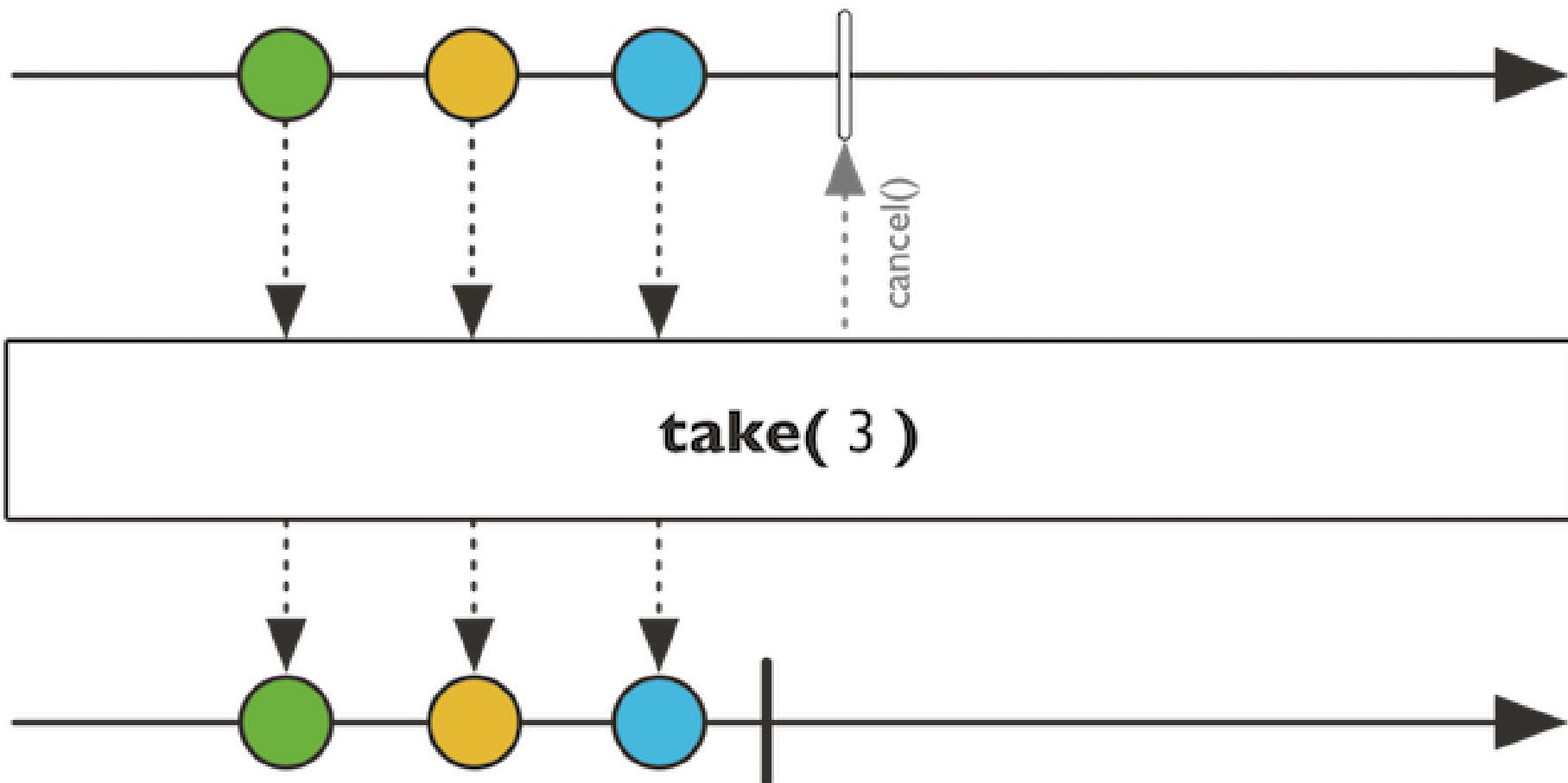


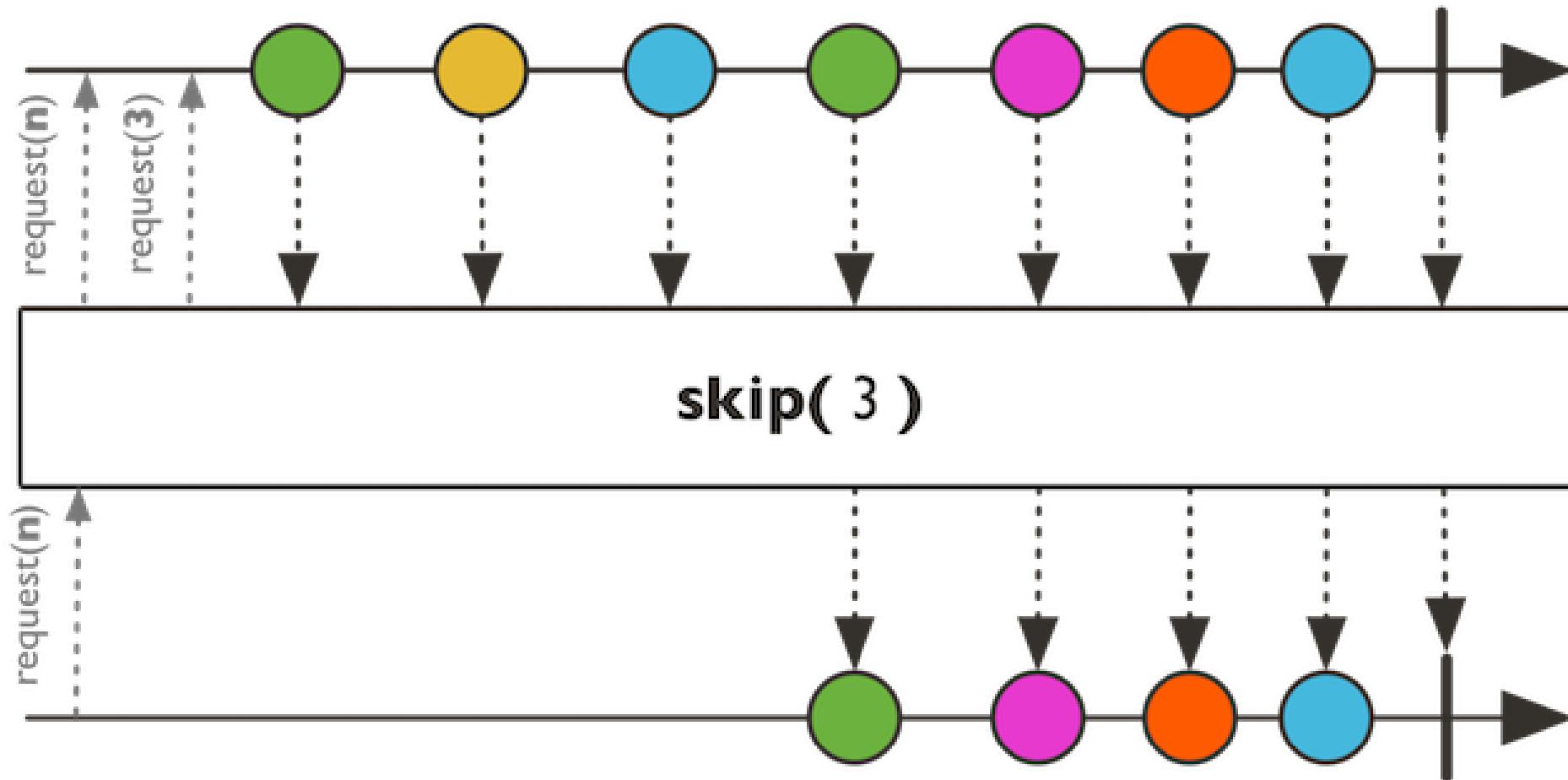
Operatory – obcinanie strumienia

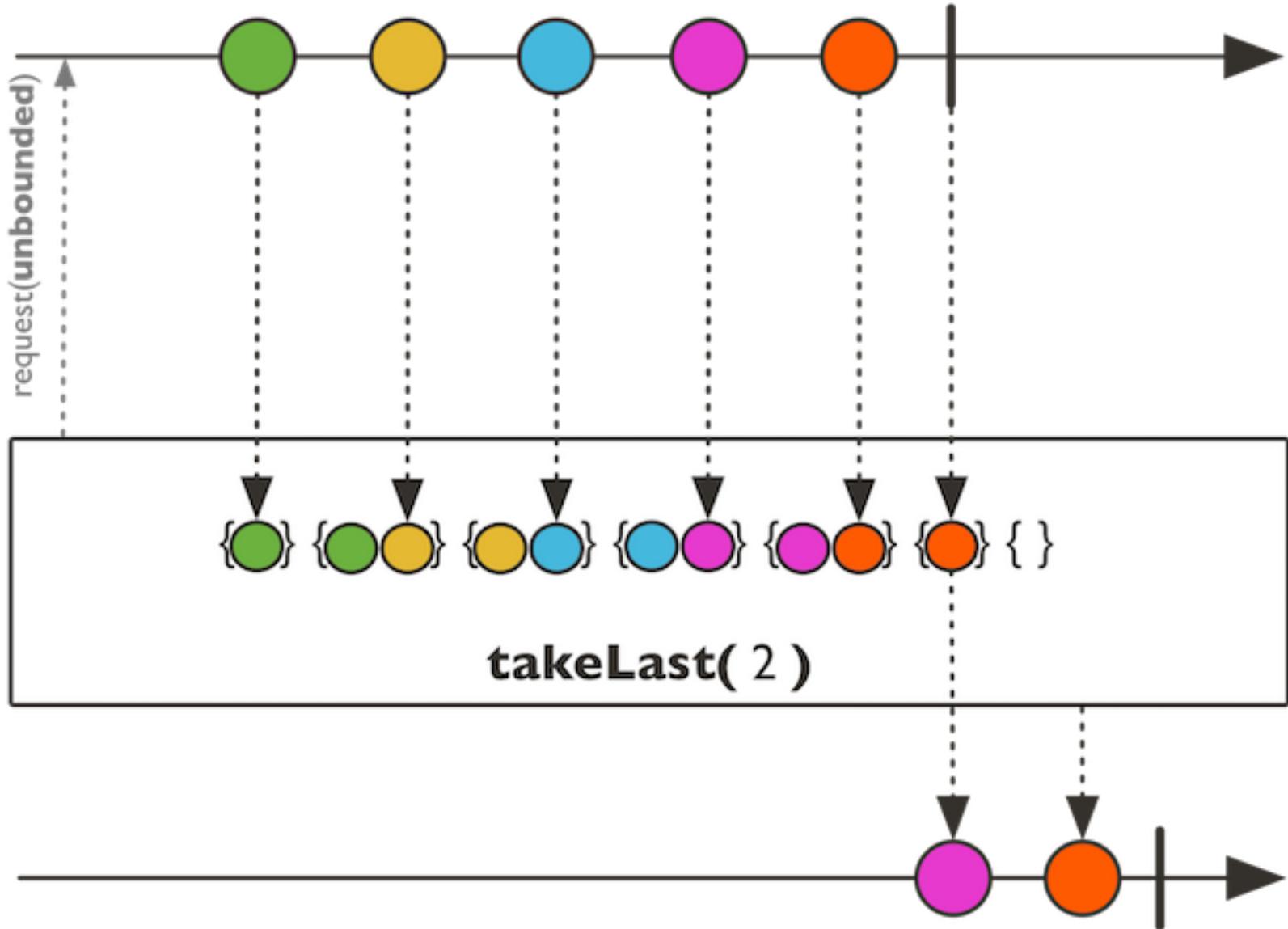
- Operator `take(n)` zwraca strumień, składający się tylko z `n` początkowych elementów strumienia
- Operator `skip(n)` działa odwrotnie – zwraca strumień nie zawierający `n` początkowych elementów strumienia
- Istnieją także operatory `takeLast()` i `skipLast()` – działają podobnie do `take()` i `skip()` ale dotyczą elementów końcowych
- Operator `takeWhile()` zwraca strumień składający się z tych elementów początkowych, które spełniają warunek przekazany jako argument. Wystąpienie pierwszego elementu nie spełniającego tego warunku powoduje zakończenie strumienia wynikowego.
- Na podobnych zasadach do operatora `takeWhile()` działają operatory `takeUntil()`, `skipWhile()` i `skipUntil()` (UWAGA: warto zwrócić uwagę na to czy element „graniczny” jest przekazywany do strumienia wynikowego)

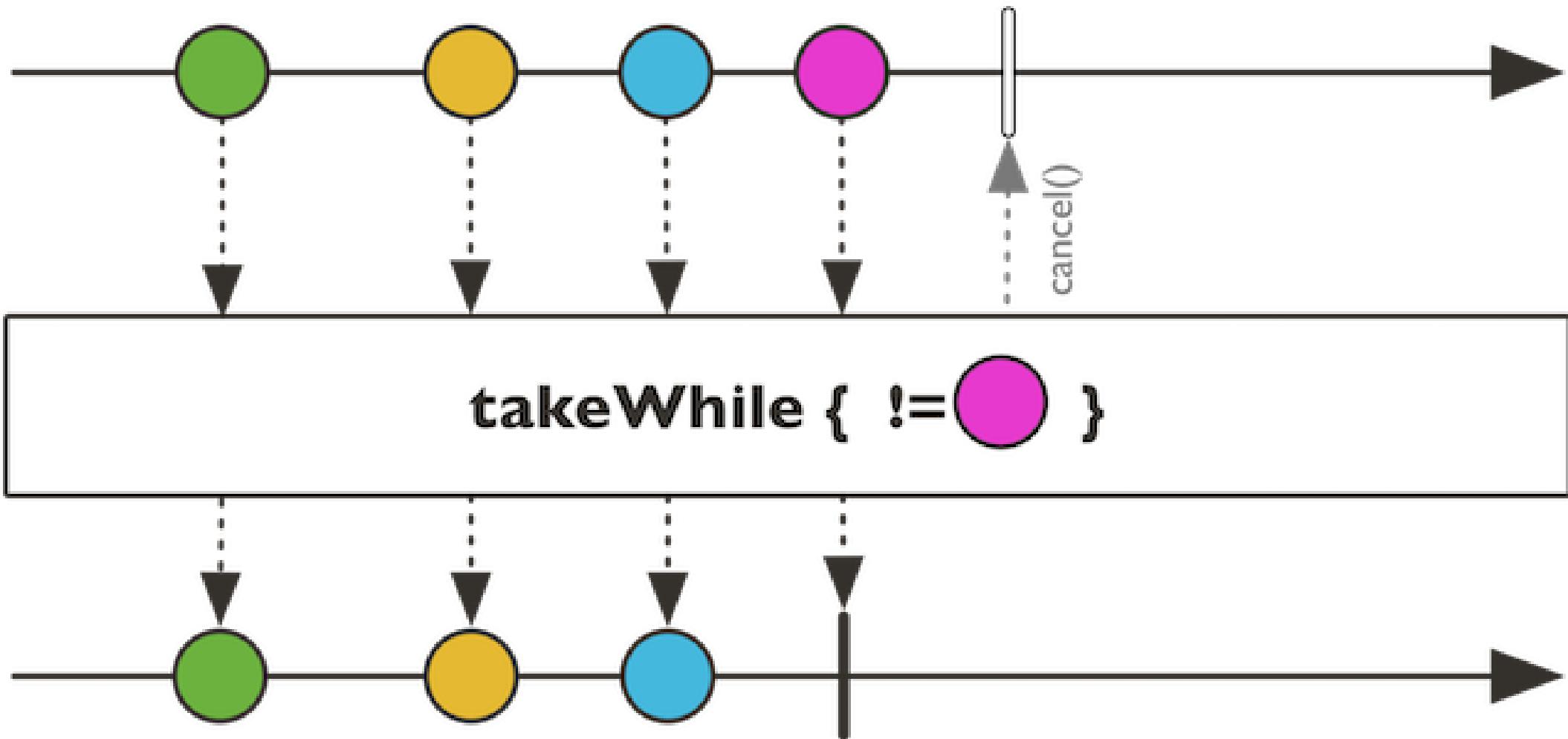


jeja.pl



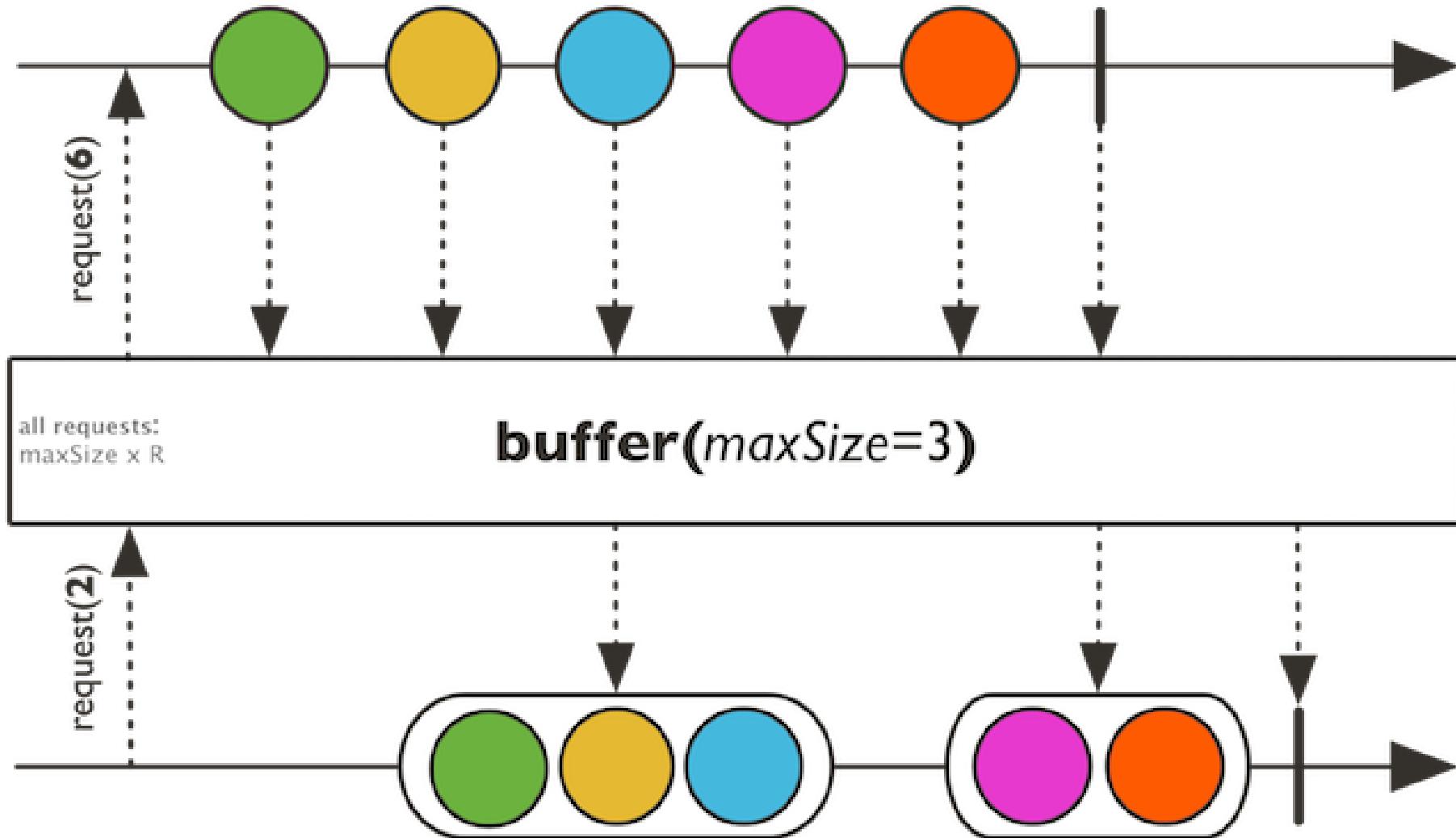


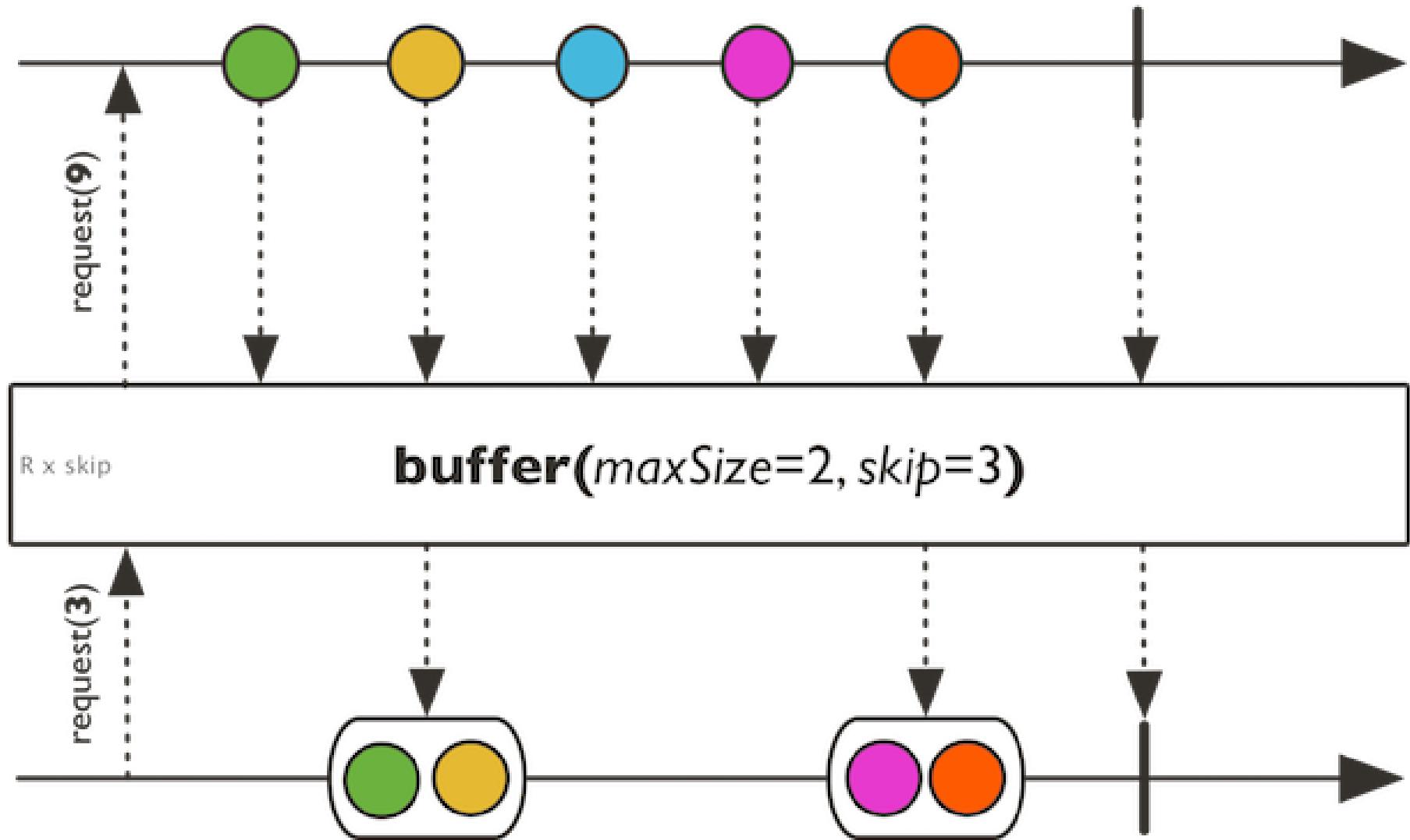


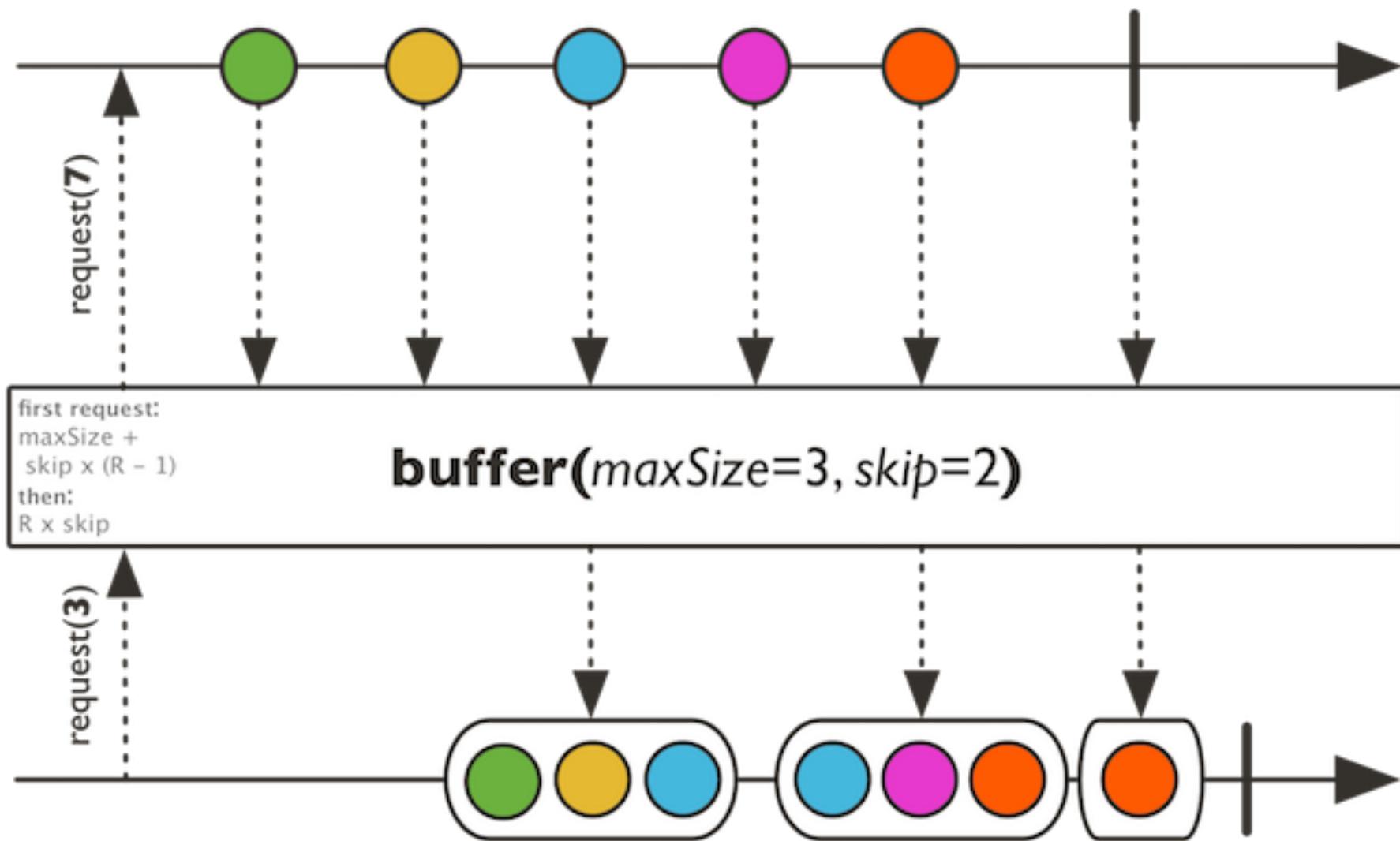


Operatory – grupowanie elementów

- Operator `buffer(n)` grupuje elementy strumienia w „paczki” po `n` elementów i zwraca strumień emitujący te „paczki” (w rzeczywistości obiekty typu `List`)
- Odmiana tego operatora – `buffer(n, m)` – również zwraca paczki po `n` elementów – ale dodatkowo paczki będą tworzone co `m` elementów.
- Operator `bufferWhile()` grupuje elementy strumienia w następujący sposób: każda paczka tworzona jest z nieprzerwanego ciągu elementów spełniających warunek przekazany jako argument. Napotkanie elementu niespełniającego tego warunku zamyka aktualną paczkę. Elementy nie spełniające warunku NIE trafiają do paczek i są tracone.
- Operator `bufferUntil()` działa nieco podobnie do `bufferWhile()`, z dwoma ważnymi różnicami:
 - Paczka jest zamykana w momencie gdy element SPEŁNIA dany warunek
 - Do paczek trafiają wszystkie elementy – żaden nie jest tracony







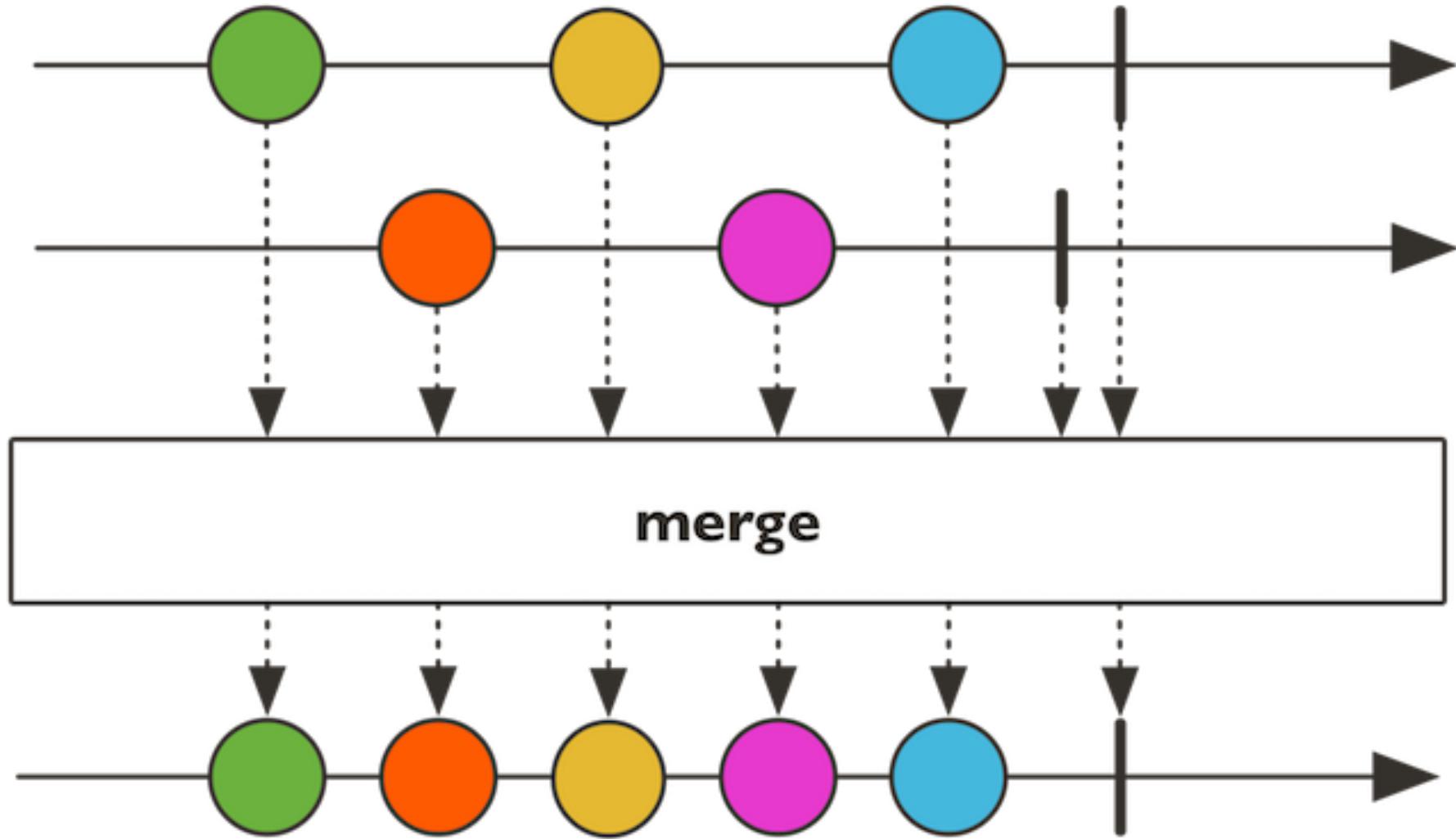
Ćwiczenie ☺ (14,16)

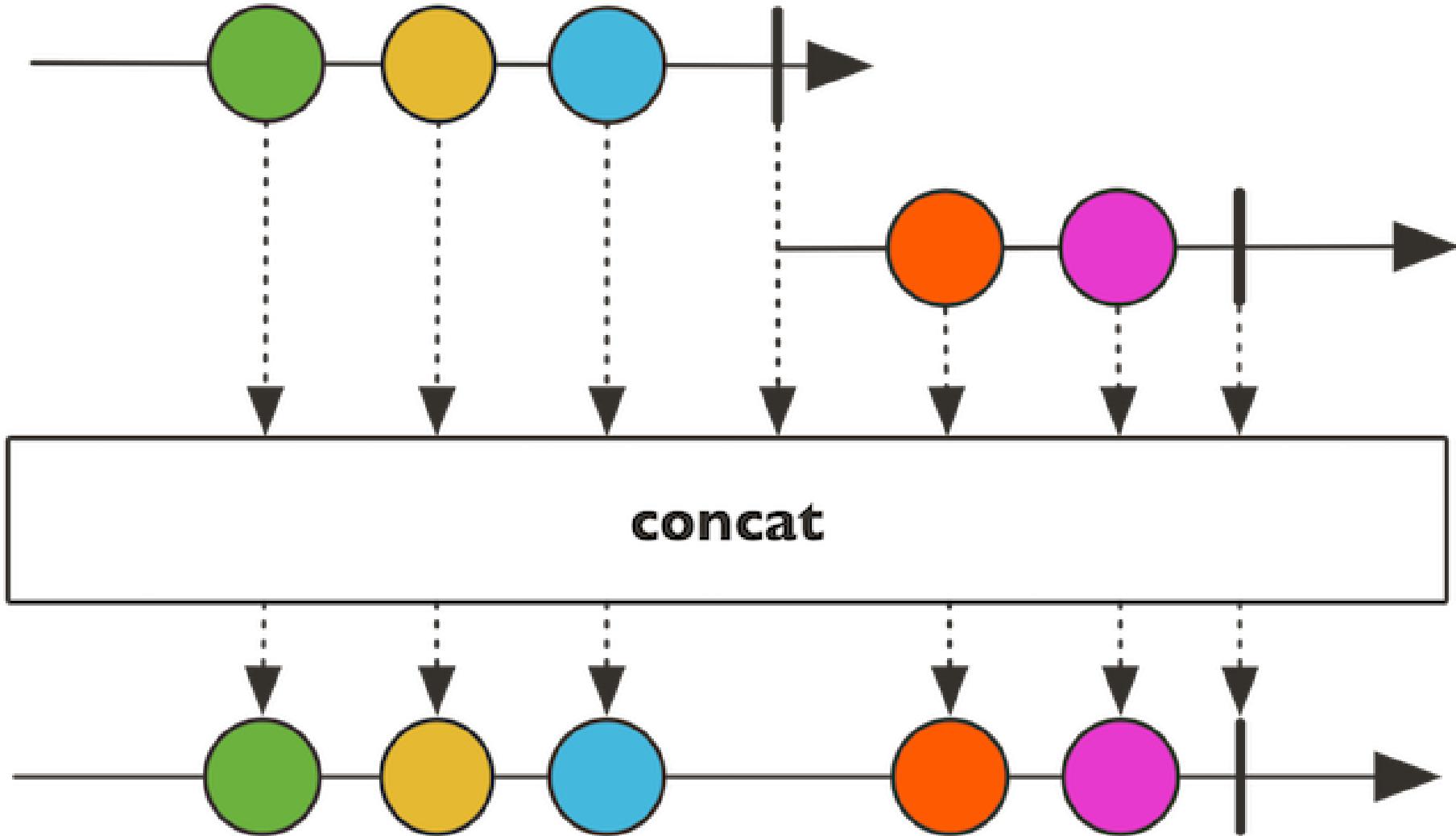


www.shutterstock.com - 153891284

Operatory – łączenie strumieni

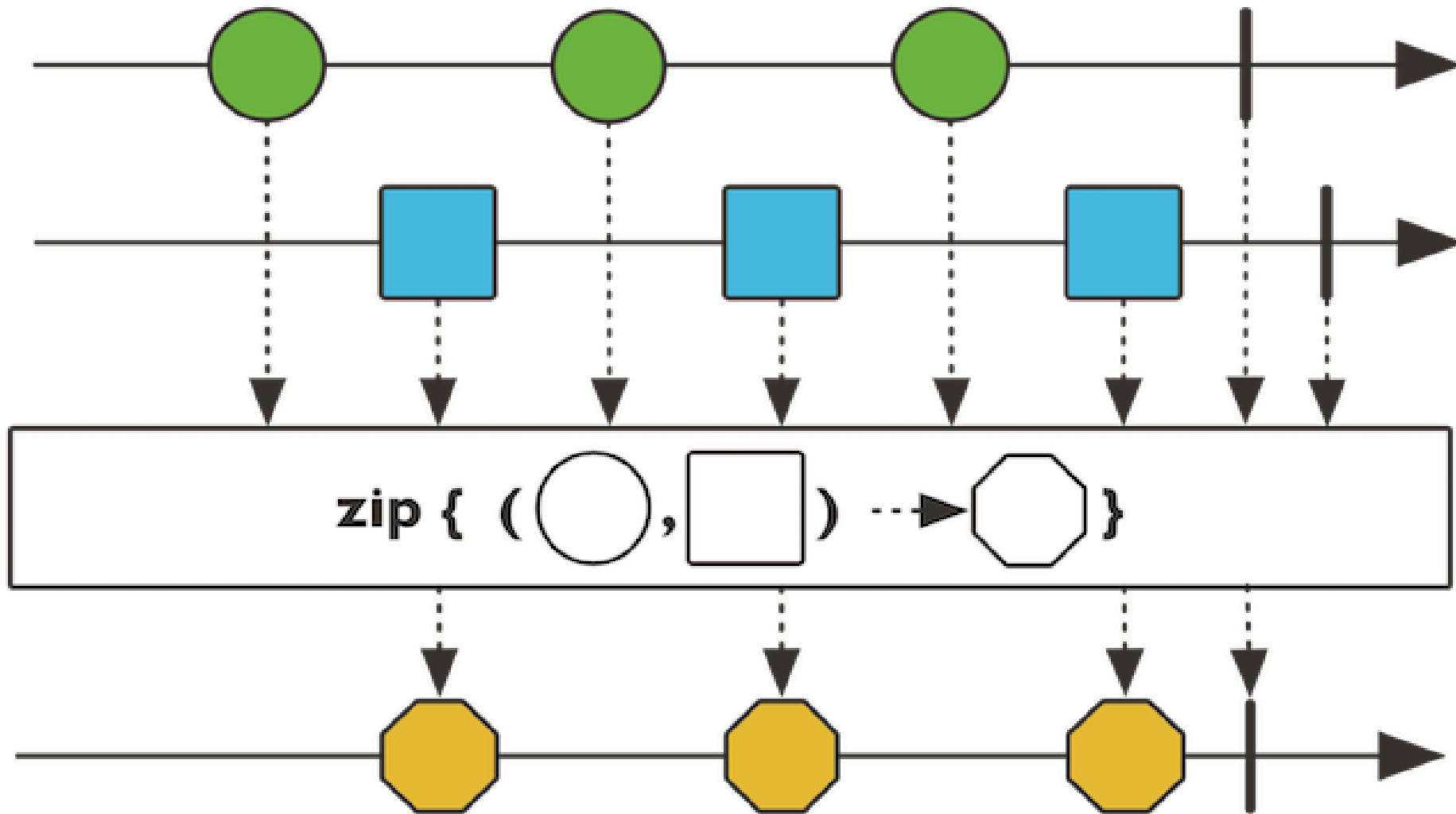
- Podstawowymi operatorami służącymi do łączenia strumieni są `mergeWith()` oraz `concatWith()`
- Obydwa operatory łączą dwa strumienie tworząc w wyniku strumień zawierający elementy obydwu
- `mergeWith()` wysyła do wynikowego strumienia elementy w kolejności jakiej pojawiają się w czasie, pozwalając na przeplatanie elementów z dwóch strumieni
- `concatWith()` najpierw pobiera wszystkie elementy tego strumienia, którego element zgłosił się wcześniej a dopiero potem wszystkie z drugiego
- Istnieją też wersje statyczne tych operatorów – `merge()` i `concat()` pozwalające na połączenie N strumieni naraz
- Operatorów `merge()` i `concat()` można też użyć dla Mono, ale zwróconą wartością będzie Flux (co w sumie jest logiczne)





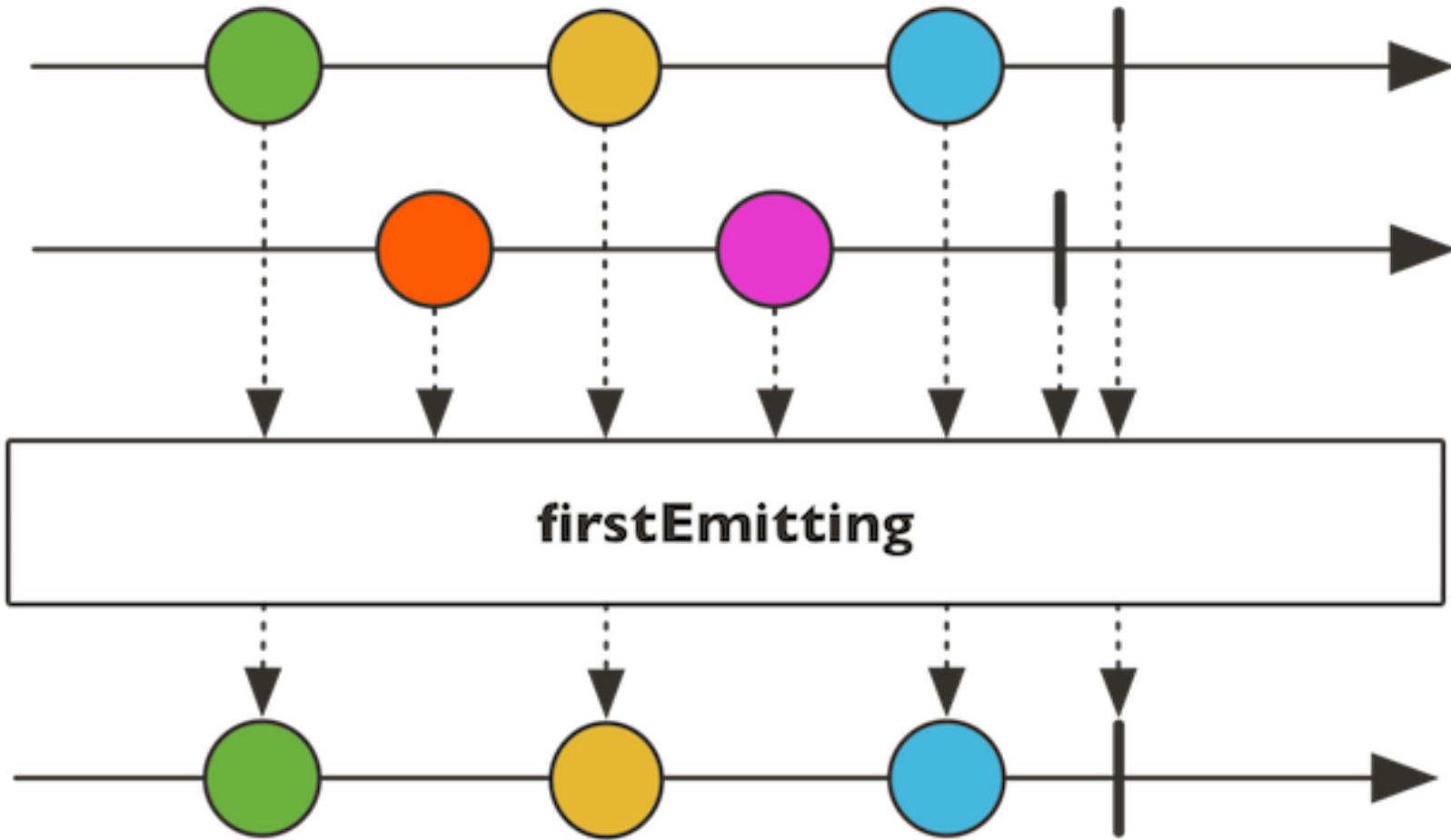
Operatory – łączenie strumieni

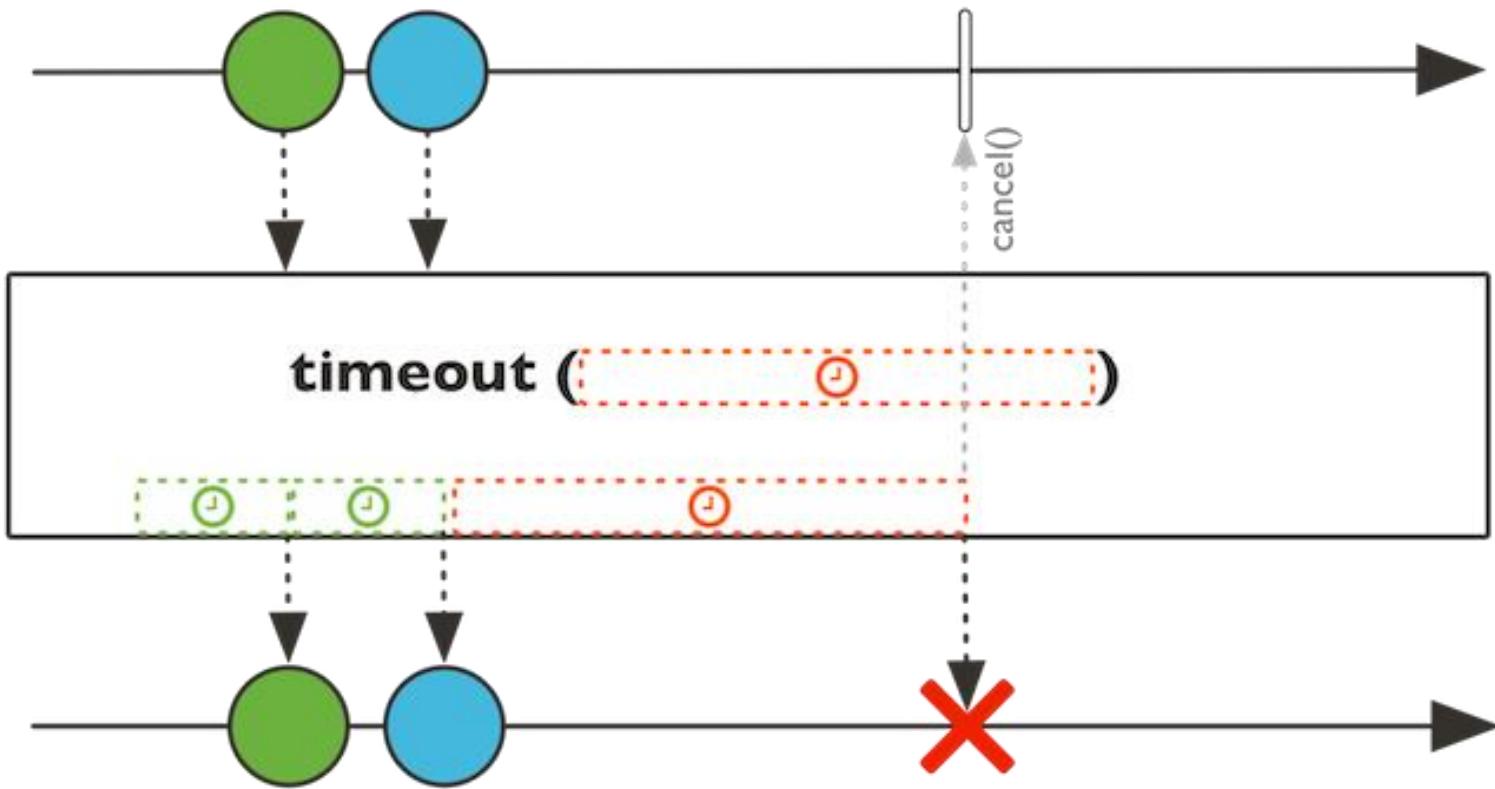
- Bardziej zaawansowanym operatorem pozwalającym na połączenie dwóch strumieni jest `zipWith()`
- `zipWith()` pobiera po jednym elemencie z każdego strumienia wejściowego, przepuszcza je przez funkcję agregującą i zwróconą wartość wypycha do docelowego strumienia
- Cykl powtarza się dla kolejnych elementów
- Istnieje także statyczna wersja metody – `zip()` pozwalająca na połączenie N strumieni naraz



Operatory – zależności czasowe

- Istnieje mnóstwo operatorów pozwalających na powiązanie strumienia z zależnościami czasowymi. Tutaj omówimy tylko dwa najbardziej podstawowe
- Operator `first()` przyjmuje N strumieni Flux i zwraca ten w którym jako pierwszym pojawi się **jakikolwiek** sygnał (istnieje też wersja dla Mono)
- Operator `timeout()` przyjmuje strumień Flux i zmienia jego zachowanie tak, że w przypadku gdy kolejne elementy nie są publikowane odpowiednio szybko (maksymalny odstęp jest przekazywany jako parametr) – następuje rzucenie błędu (`TimeoutException`)





Ćwiczenie ☺ (5,8)



www.shutterstock.com - 153891284

Operatory – tworzenie własnych

- W przypadku gdy w kilku miejscach aplikacji stosujemy ten sam ciąg kilku operatorów, Reactor pozwala nam na enkapsulację takiego ciągu jako nowego operatora. Takie podejście pozwala na zastosowanie metodologii DRY.
- Nowy operator tworzymy jako implementację interfejsu `Function<Flux<T>, Flux<T>>` W implementacji definiujemy w jaki sposób chcemy przekształcić wejściowy strumień w wyjściowy
- Stworzone przekształcenie możemy wykorzystać następnie za pomocą dwóch operatorów - `transform()` lub `compose()`.
- Różnica między nimi polega na tym iż `transform()` uruchamia nasze przekształcenie tylko raz, niezależnie od ilości subskrybentów. W przypadku `compose()` przekształcenie uruchamiane jest dla każdego subskrybenta.

Ćwiczenie ☺ (18)



www.shutterstock.com - 153891284

Operatory - podsumowanie

- Pokazane operatory to tylko wierzchołek góry lodowej ☺
- Bardzo dobry przewodnik po operatorach (w 90% przypadków wystarcza) znajduje się w dodatku do dokumentacji Reactor:
<http://projectreactor.io/docs/core/release/reference/docs/index.html#which-operator>
- W pozostałych 10% najlepiej posłużyć się dokumentacją do klas Flux i Mono:
<https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html> oraz
<https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html>

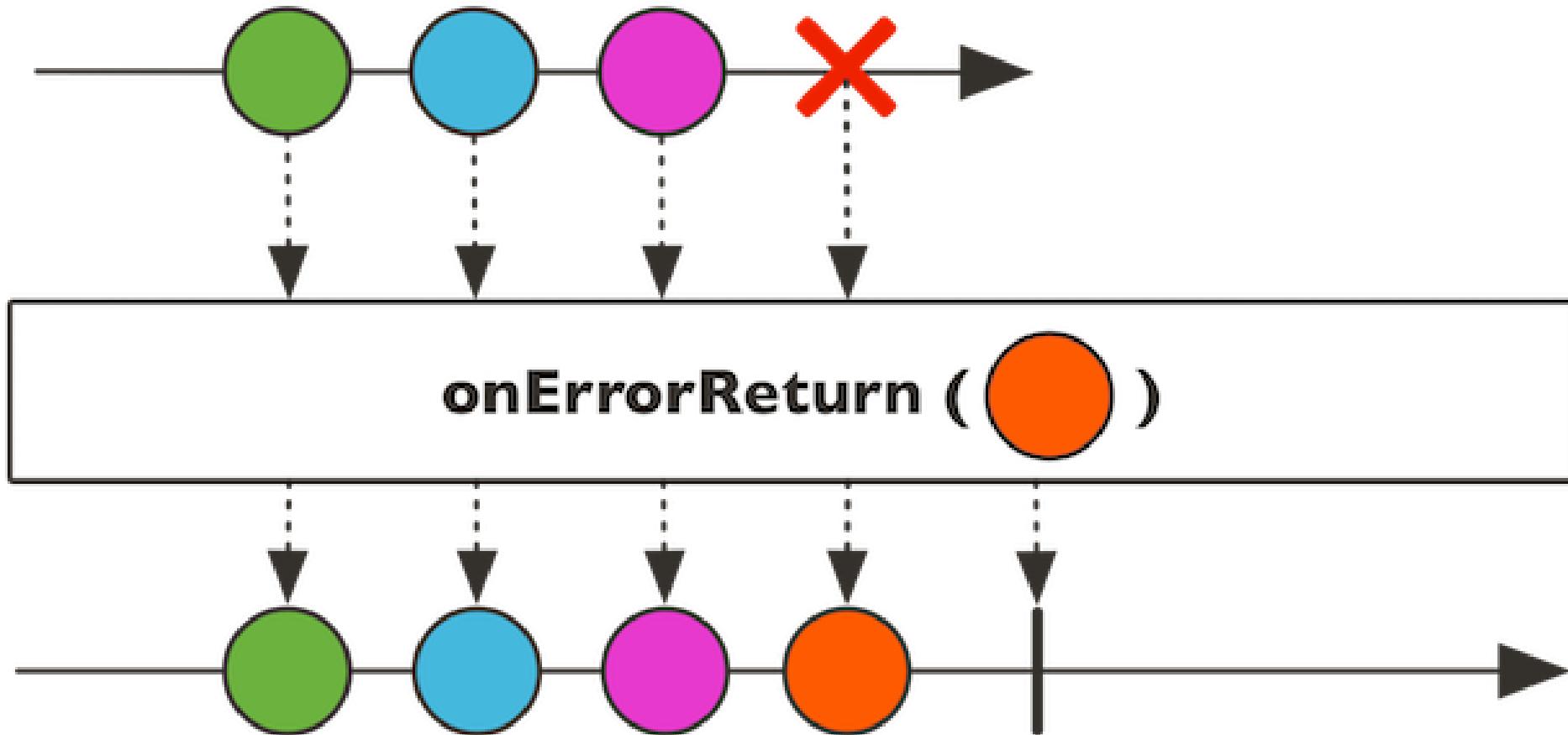
Obsługa błędów

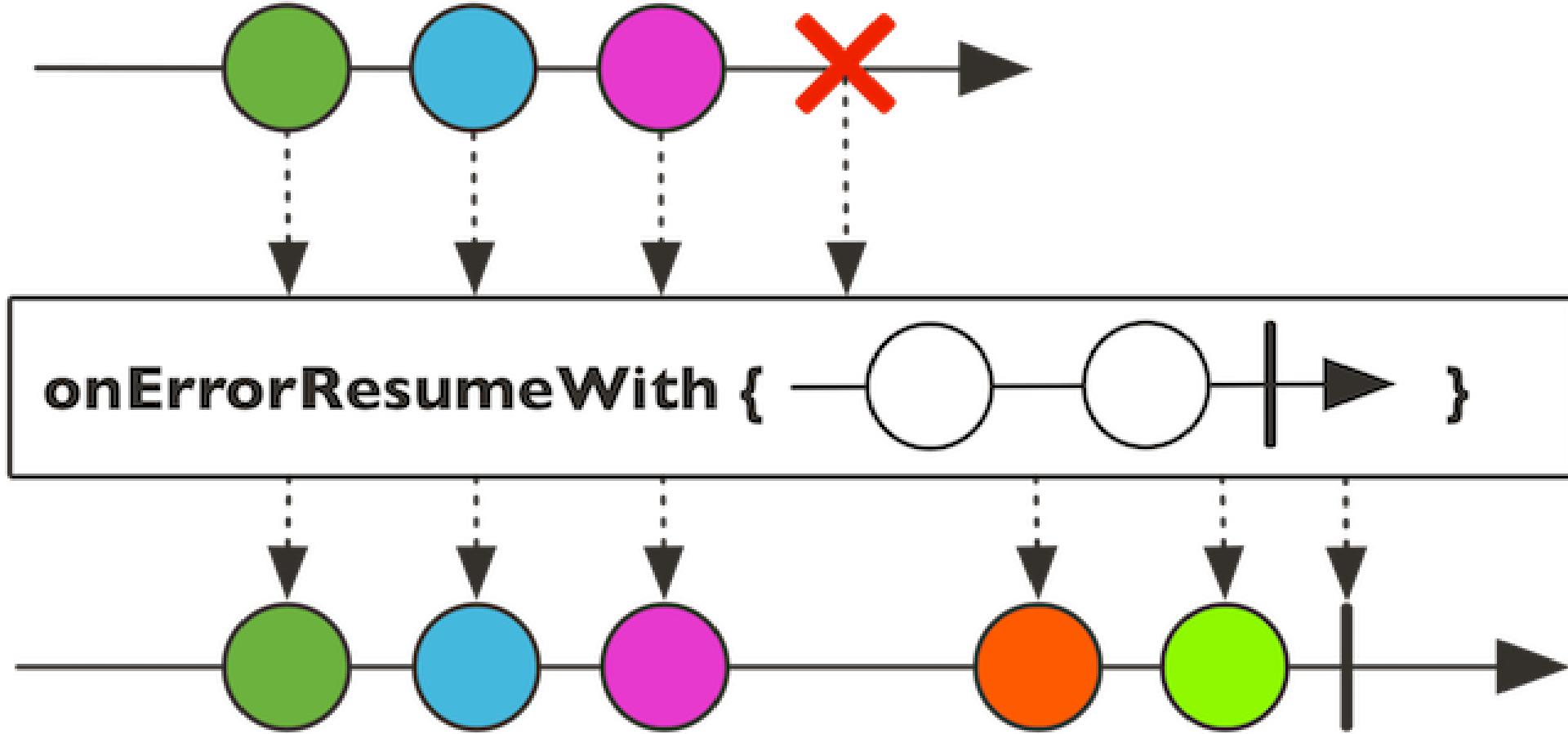
Czasami kończy się tak...

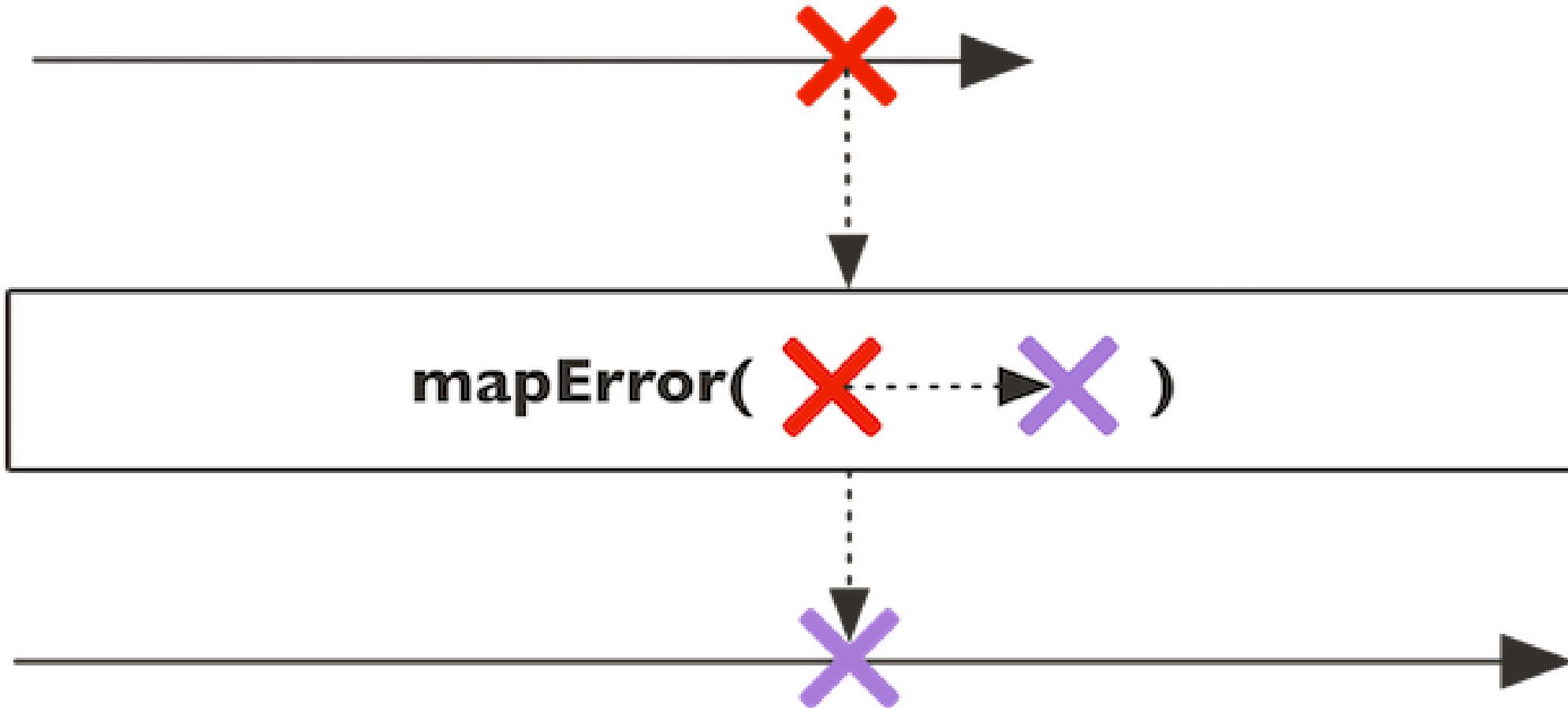


Obsługa błędów

- Każdy zdarzenie `onError()` występujące w strumieniu jest zdarzeniem terminującym (wynika to ze specyfikacji Reactive Streams).
- Błędy takie można obsłużyć na kilka sposobów:
- Skonstruować nowy strumień zwracający element zamiast błędu (operator `onErrorReturn()`)
- Skonstruować nowy strumień zwracający strumień zamiast błędu (operator `onErrorResume()`)
- Rzucić inny wyjątek (można wykorzystać operator `onErrorMap()`)







Obsługa błędów

- W przypadku wyjątków sprawdzalnych przydaje się dwie statyczne metody klasy `Exceptions` - `propagate()` i `unwrap()`. Pierwsza „pakuje” wyjątek sprawdzalny w niesprawdzalny, który potem można „odpakować” za pomocą drugiej
- Nieliczne klasy wyjątków (np. `OutOfMemoryError`) są tzw. wyjątkami fatalnymi – nie są propagowane jako zdarzenie `onError()` tylko od razu wyrzucane. Wynika to z faktu że po wystąpieniu takiego wyjątku Reactor nie jest już w stanie kontynuować pracy.

Ćwiczenie ☺ (7)



www.shutterstock.com - 153891284

Współbieżność



Współbieżność

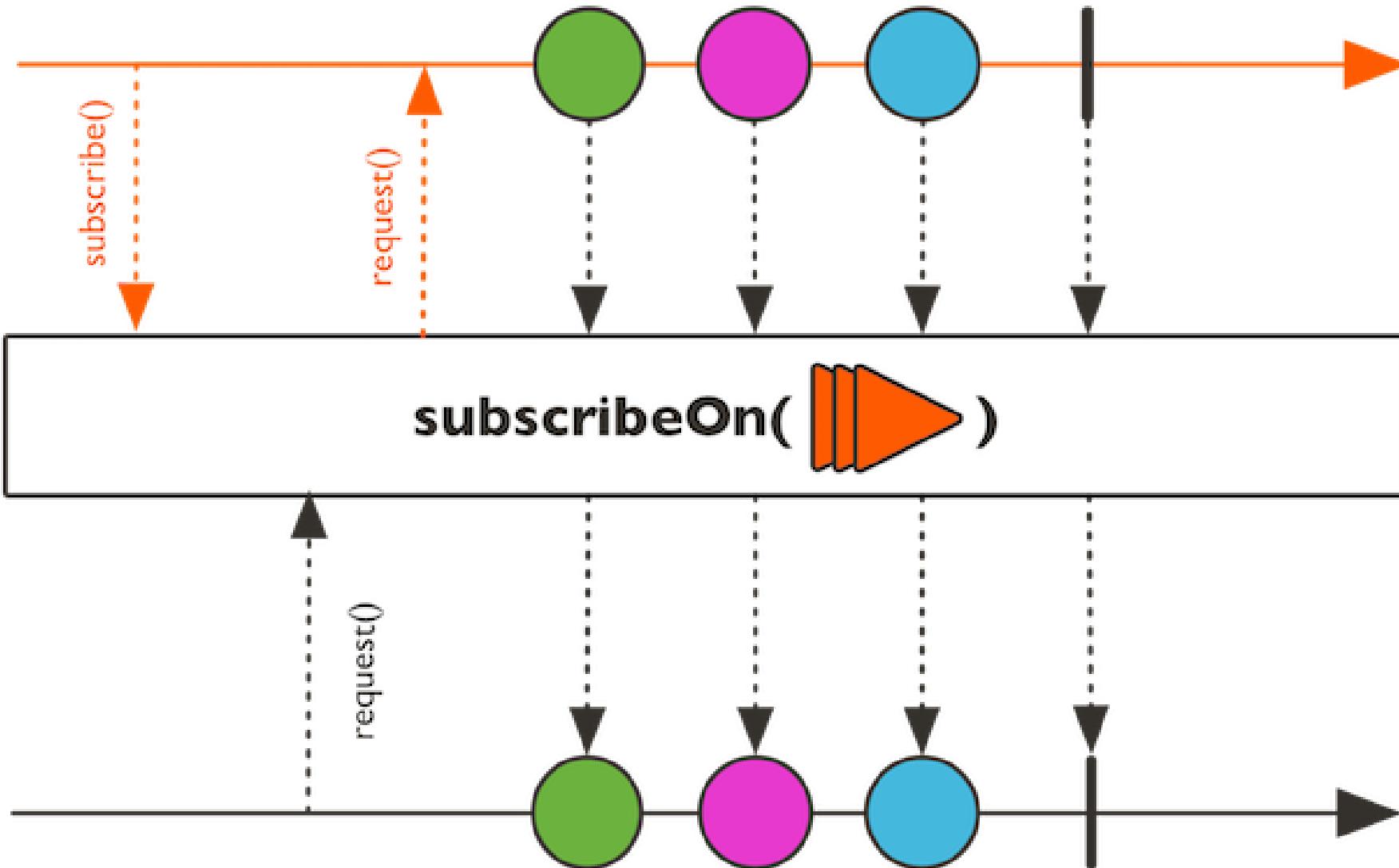
- Reactor sam z siebie nie narzuca w jakim modelu współbieżności będą wykonywały się poszczególne operacje (i czy w ogóle współbieżność będzie użyta).
- To w jakim wątku wykona się konkretna operacja jest określone przez przyporządkowany do niej scheduler.
- Scheduler jest obiektem implementującym interfejs Scheduler dostarczany przez Reactor. Można w dużym skrócie powiedzieć że jest to odpowiednik interfejsu ExecutorService z Javy

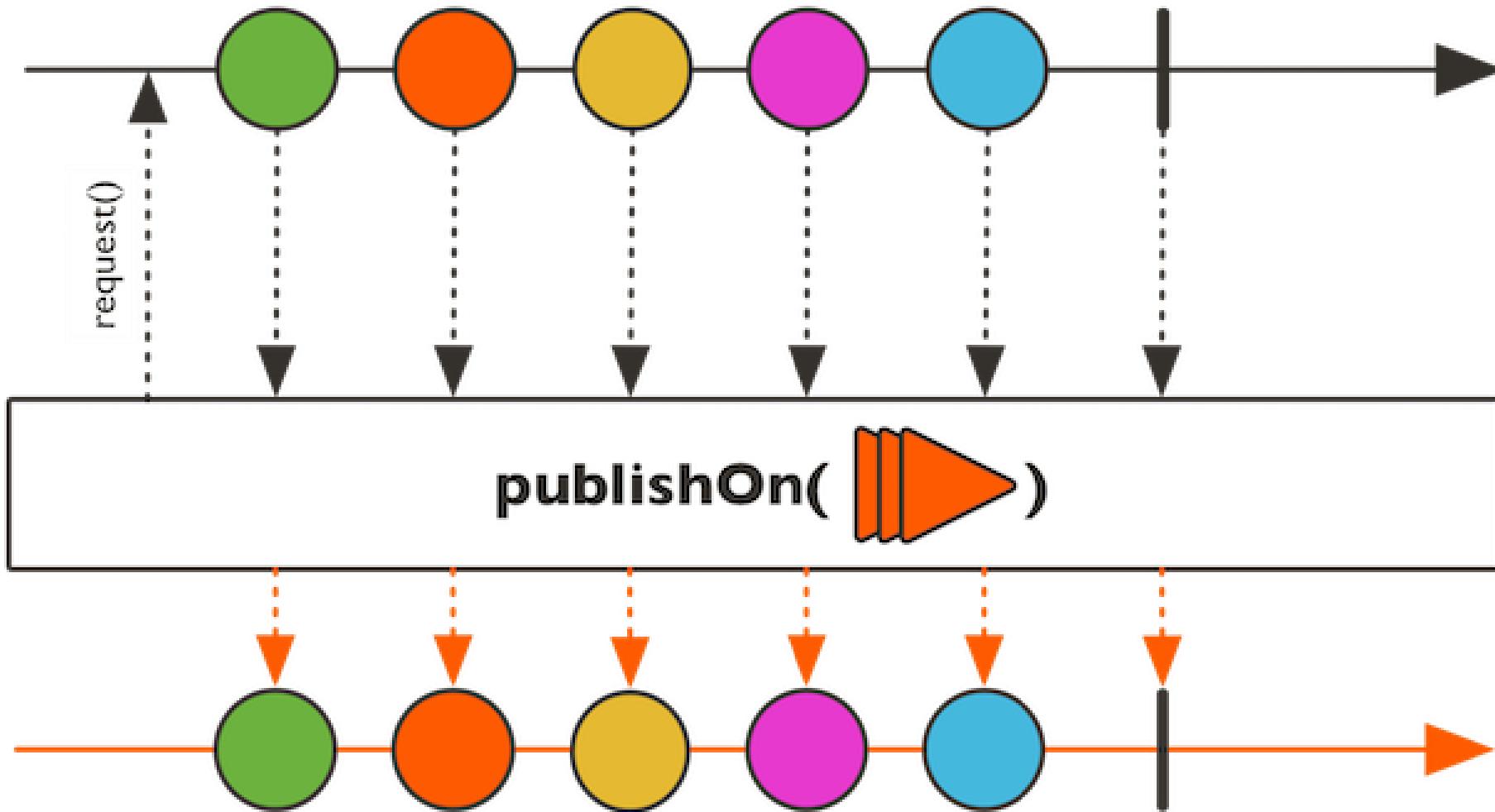
Schedulery

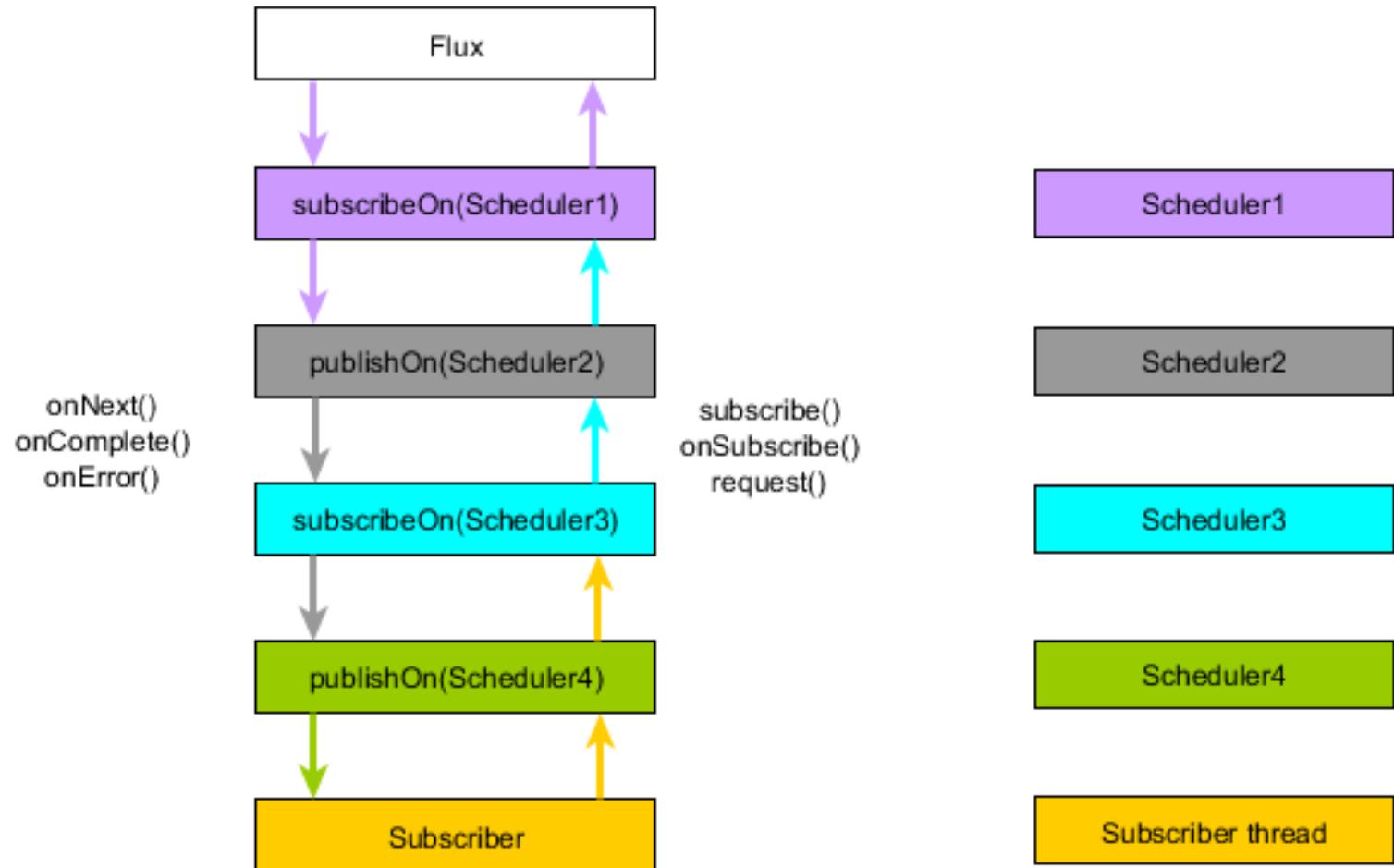
- Reactor dostarcza kilka gotowych implementacji interfejsu Scheduler. Są one dostępne poprzez metody klasy pomocniczej Schedulers. Najbardziej podstawowe to:
 - immediate () – wykonuje operacje w aktualnym wątku. Stosowany domyślnie przez większość operatorów
 - single () – wykonuje wszystkie operacje w pojedynczym reużywalnym wątku
 - newSingle () – dla każdej nowej operacji tworzy nowy wątek
 - elastic () – tworzy elastyczną pulę wątków. Idealny do stosowania w sytuacji kiedy zachodzi potrzeba wykonania blokującej operacji (o tym później)
 - parallel () – sztywna pula wątków o ilości odpowiadającej ilości CPU w systemie

Operatory zmiany schedulera

- Do zmiany obiektu Scheduler który zostanie użyty podczas operacji na strumieniu służą dwa operatory: publishOn () i subscribeOn () . Oba jako argument przyjmują żądany Scheduler
- Aby łatwiej zrozumieć ich działanie, należy mieć na uwadze że każde wpięcie operatora w łańcuchu tak naprawdę tworzy nowy strumień subskrybujący się do poprzedniego.
- Operator subscribeOn () powoduje że podany Scheduler zostanie zastosowany do operacji subscribe () oraz request () w całym łańcuchu powyżej miejsca jego wpnięcia – oraz do wszystkich sygnałów (next/complete/error)
- Operator publishOn () powoduje że podany Scheduler zostanie zastosowany do sygnałów (next/complete/error) poniżej miejsca jego wpnięcia







Kod reaktywny w kodzie blokującym

- Jak już zostało wspomniane w idealnym reaktywnym świecie wszystkie operacje muszą być nieblokujące
- Ponieważ jednak świat nie jest idealny ☺ i trzeba dostosować się do istniejących bibliotek / kodu, to musi być możliwość przejścia między kodem reaktywnym a kodem blokującym.
- Łatwiejsze do zrozumienia jest przejście w stronę Reactive → Blocking. Sytuacja taka zachodzi gdy musimy pobrać wszystkie dane (lub przynajmniej ich część) ze strumienia aby przekazać je dalej do kodu nieobsługującego reaktywności.
- Najbardziej popularne metody służące do blokowania to:
 - Dla Mono – `block()`
 - Dla Flux – `blockFirst()`, `blockLast()`, `toStream()`, `toIterable()`

Kod blokujący w kodzie reaktywnym

- Nieco trudniejsze jest przejście w stronę Blocking → Reactive – innymi słowy gdy mamy kod w większości reaktywny, ale są kawałki typowo blokujące. Można tu generalnie wyróżnić dwa przypadki:
- Pierwszy przypadek zachodzi gdy chcemy zbudować strumień a proces zbierania danych jest blokujący (np. odczyt z bazy). Wówczas najlepiej zastosować operator `subscribeOn()` – tak aby przenieść pobieranie elementów do oddzielnego wątku / puli wątków
- Czasem konieczne też może okazać się zastosowanie operatora `defer()` – opóźnia on budowanie strumienia do momentu aż faktycznie ktoś go zasubskrybuje
- Drugi przypadek zachodzi gdy musimy wykonać blokującą operację dla każdego elementu wyemitowanego przez strumień. Tu z kolei dobrze sprawdzi się operator `publishOn()`

Ćwiczenie ☺ (10, 11)



www.shutterstock.com - 153891284

Testowanie

WAŻNE, ŻE DZIAŁA



memy.pl

We make IT



JCommerce

Testowanie

- W obecnych czasach testowanie jednostkowe to już podstawa – nie należy pisać kodu jeżeli nie mamy możliwości przetestowania go
- Testowanie jednostkowe w programowaniu reaktywnym jest trudniejsze ze względu na to że kod wykonuje się nieliniowo; dodatkowo mogą wejść w grę zależności czasowe
- Aby ułatwić pisanie testów jednostkowych Reactor udostępnia nam kilka klas pomocniczych: StepVerifier, TestPublisher i PublisherProbe
- Zajmiemy się tylko klasą StepVerifier jako najprostszą w użyciu a jednocześnie pozwalającą przetestować podstawowe przypadki użycia

Testowanie – StepVerifier

- Klasy StepVerifier używamy wtedy gdy mamy jakiś strumień Flux / Mono i chcemy sprawdzić co zacznie emitować gdy się do niego zasubskrybujemy. Taki scenariusz występuje najczęściej
- Obiekt klasy StepVerifier tworzymy używając buildera, który z kolei tworzymy poprzez statyczną metodę create(). Jej argumentem jest Flux / Mono do którego chcemy się zasubskrybować.
- Można też utworzyć StepVerifier z wykorzystaniem wirtualnego zegara (przydatne przy testowaniu zależności czasowych; normalny test trwałby wtedy zbyt długo). Zamiast create() używamy wtedy metody withVirtualTime()

Testowanie – StepVerifier

- Po utworzeniu buildera możemy:
 - dołożyć do niego asercje sprawdzające to co wyemituje badany strumień np: `assertNext()`, `assertNextCount()`, `assertComplete()`, `assertError()`
 - skonsumować jeden lub więcej elementów ze strumienia bez sprawdzania co w nich jest np: `consumeNext()`
 - odczekać jakiś czas nic nie robiąc – `thenAwait()` lub zweryfikować że przez ten czas nic się nie zdarzyło – `expectNoEvent()`
- Powyższe metody można zwracają builder więc można je łączyć w jeden łańcuch
- Builder musimy zakończyć metodą `verify()`. Dopiero ona powoduje faktyczną subskrypcję do strumienia i wykonanie wszystkich asercji / konsumpcji zawartych w builderze

Ćwiczenie ☺ (3)



www.shutterstock.com - 153891284

Backpressure

Backpressure

- Jak już było wcześniej wspomniane Reactor posiada wsparcie dla backpressure – mechanizmu kontrolującego szybkość wysyłania elementów do subskrybenta strumienia
- Każdy obiekt typu Subscription posiada metodę `request(long n)` – jej zwołanie oznacza że z jednej strony prosimy o dane, a z drugiej pozwalamy strumieniowi wysłać maksymalnie `n` elementów do subskrybenta.
- Wyjątkiem od tej reguły jest sytuacja gdy jako `n` podawana jest wartość `Long.MAX_VALUE` – wtedy strumień może założyć że szybkość wysyłania nie jest niczym ograniczona (tzw. nieograniczony subskrybent)
- Bez zwołania metody `request()` strumień nie wyśle NIC
- Jeśli przy zwołaniu metody `subscribe()` nie przekażemy obiektu subskrybenta Reactor użyje subskrybenta domyślnego – który zachowuje się jak nieograniczony

Backpressure

- Backpressure możemy także zastosować podczas testowania za pomocą klasy StepVerifier:
- Metoda `create()` tej klasy może przyjąć dodatkowy parametr typu `long` – oznacza on ile inicjalnie elementów chcemy pociągnąć ze strumienia po zasubskrybowaniu. Domyślnie przyjmowany jest `Long.MAX_VALUE` – czyli brak ograniczeń
- Do buildera można podłączyć także metodę `thenRequest(long n)` – oznacza ona wtedy że chcemy pobrać kolejne `n` elementów
- Należy pamiętać o wpięciu metody `thenCancel()` w sytuacji gdy celowo nie ciągniemy wszystkich elementów ze strumienia – w przeciwnym razie test zakończy się wyjątkiem

Ćwiczenie ☺ (6.1,6.2)



www.shutterstock.com - 153891284

Debugowanie

Standardowy callstack Reaktora przy błędzie...

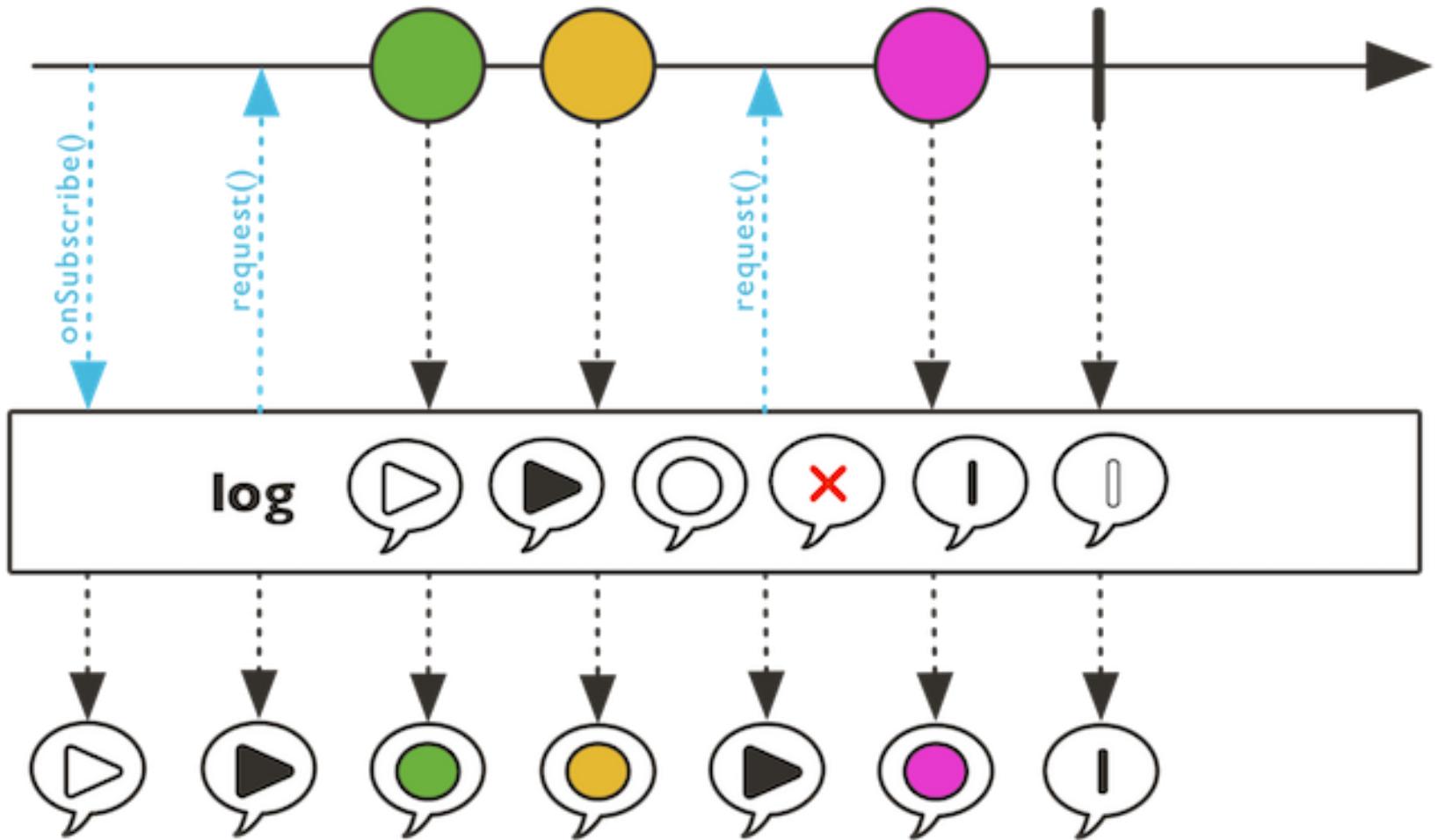


Debugowanie

- Analiza tego co dzieje się w ramach przetwarzania strumieni może być trudna, ze względu na to iż:
 - Kod wykonuje się nieliniowo (w ramach pojawiania się kolejnych elementów)
 - Stos wywołania zawiera praktycznie same odwołania do wewnętrznych klas i metod biblioteki Reactor co jest mało czytelne
- Powyższe problemy stają się szczególnie bolesne w momencie gdy wystąpi jakiś błąd i trzeba go namierzyć
- Reactor oferuje trzy mechanizmy ułatwiające debugowanie – operator `log()`, operatory `doOn...` i tryb debugowania

Operator log()

- Operator log() służy do logowania tego co dzieje się w strumieniu
- Używa się go tak jak każdego innego operatora, jego zachowanie jest „przezroczyste”
- Operator podpina się do mechanizmu logowania aplikacji (domyślnie przez SLF4J) i zrzuca do logów takie zdarzenia ze strumienia jak:
 - onNext(), onComplete(), onError()
 - onSubscribe(), onSubscriptionCancel()
 - request()



Operatory doOn...()

- Operatory doOn...() umożliwiają zdefiniowane akcji jaka ma zostać wykonana w momencie gdy zajdzie jakieś zdarzenie w strumieniu
- Możemy wykorzystać następujące operatory:
 - doOnSubscribe(), doOnRequest(), doOnCancel()
 - doOnNext(), doOnComplete(), doOnError(), doOnEach()
 - doOnTerminate(), doAfterTerminate()
- W ramach zdefiniowanej akcji możemy wykonać dowolną czynność, ale warunek jest taki że nie może ona :
 - Trwać dugo
 - Blokować
- Akcję możemy wykorzystać np. na wypisanie czegoś do logów, jeśli domyślne działanie operatora `log()` nam nie odpowiada

Tryb debugowania

- W przypadku włączenia trybu debugowania Reactor dokonuje instrumentacji kodu w czasie składania łańcucha operatorów co pozwala na stworzenie czytelniejszego stosu wywołania
- Tryb debugowania włącza się poprzez zwołanie następującej metody `Hooks.onOperatorDebug()` (sam temat haków Reactor wykracza poza temat prezentacji)
- Metoda ta musi być zwołana zanim strumień zostanie poskładany, a więc najlepszym dla niej miejscem jest moment startu aplikacji
- Instrumentacja kodu jest bardzo kosztowną opcją i powinna być stosowana wyłącznie jeśli inne metody debugowania zawiodą.

Ćwiczenie ☺ (6.3,6.4)



Kontekst

Kontekst (1)

- Ze względu na to, iż Reactor może wykonywać operację w dowolnym wątku jaki akurat będzie dostępny oraz brak gwarancji że dany ciąg operacji dla danego subskrybenta będzie wykonany w ramach tego samego wątku – nie można użyć ThreadLocal do przechowywania kontekstu subskrybenta. By to rozwiązać można zastosować kilka sposobów
- Najprostsze podejście polega na rozszerzeniu danych biznesowych o dane kontekstowe i przekazywanie takiego typu w strumieniu. Taki sposób działa, niemniej jest mało elegancki (zanieczyszczanie domeny)
- Z tego powodu w wersji 3.1 Reactora wprowadzono możliwość ustawiania kontekstu subskrybenta i wykorzystywania go w ramach operacji na danym strumieniu

Kontekst (2)

- Typem reprezentującym kontekst jest Context - bardzo podobny do `HashMap<Object, Object>`
- Ustawiany jest za pomocą operatora `subscriberContext()`. Operator ten wymaga podania funkcji, która na podstawie istniejącego kontekstu zbuduje nowy, użyty powyżej miejsc subskrypcji
- Referencję do niego można otrzymać za pomocą statycznej funkcji `Mono.subscriberContext()` (zwraca `Mono<Context>` dlatego potrzebny będzie też operator `flatMap()`)
- Kontekst jest niezmienniczy (`immutable`). Wstawienie wartości do kontekstu np. za pomocą metody `put()` zawsze zwraca nową instancję.

Ćwiczenie (19)



www.shutterstock.com - 153891284

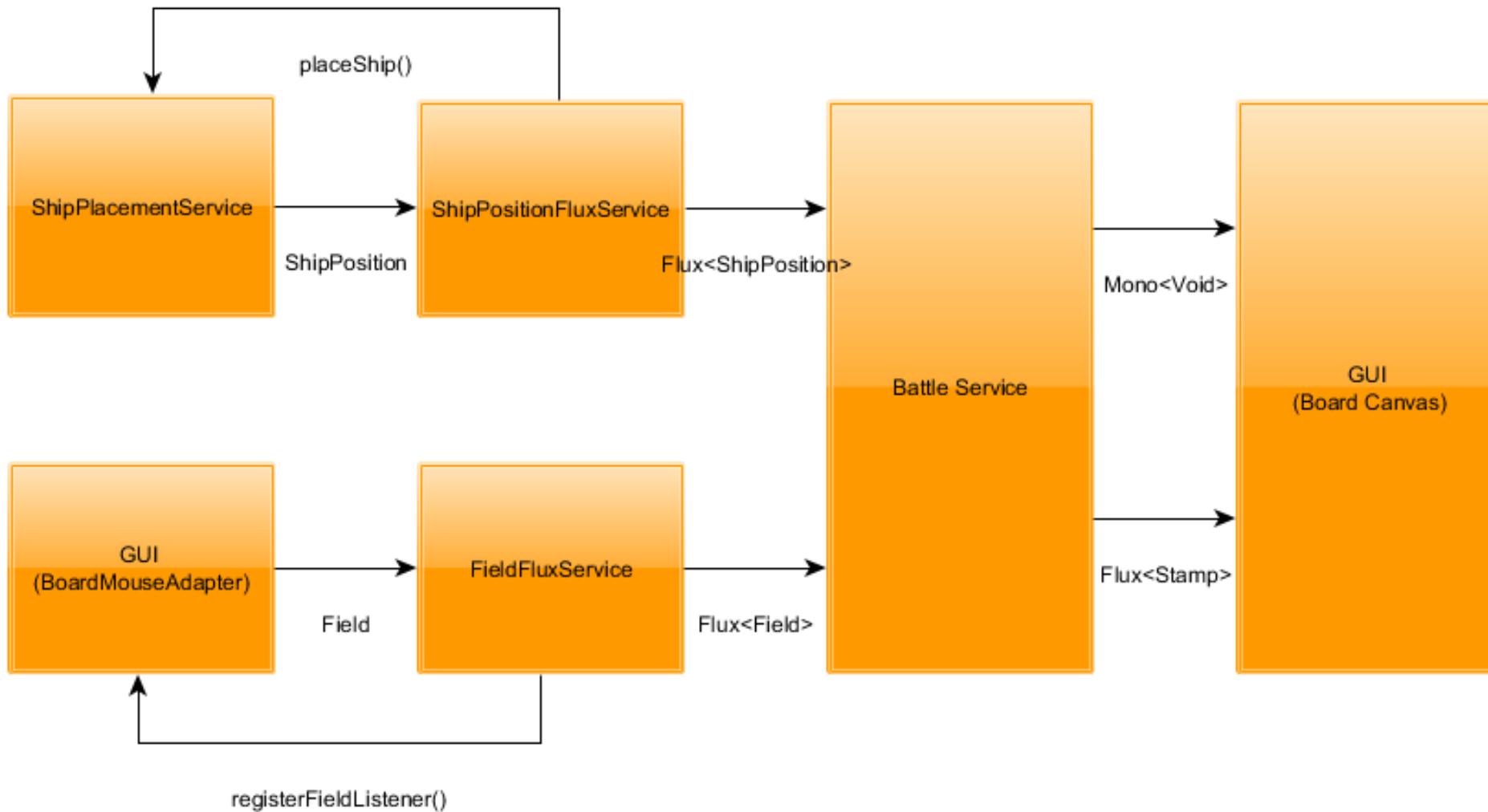
Budowa własnej aplikacji

Let's play the game ☺

	1	2	3	4	5	6	7	8	9	10
A		■						■		
B										
C			■■■							
D										
E				■				■		
F							■			■
G					■					
H										
I						■■				■
J			■■■							

Gra w okręty

- Nie ma lepszej praktyki niż pisanie gier ☺ Spróbujemy stworzyć jednoosobową grę w okręty w wersji reaktywnej
- Proszę sklonować repozytorium:
<https://github.com/chrosciu/rx-battleships>
- Z punktu widzenia reaktywności w grze występują dwa strumienie zdarzeń:
 - Statki wraz z ich położeniem (generowane przez algorytm losujący)
 - Strzały oddawane przez użytkownika (generowane przez MouseListener)
- Zadanie polega na takim stworzeniu strumieni, oraz ich podłączeniu do GUI tak aby powstała pełnoprawna gra w statki dla jednej osoby
- Dla ułatwienia zadania serwisy, które trzeba zaimplementować mają już wstrzyknięte wszystkie niezbędne zależności



Gra w okręty - zadania (1)

- Zaimplementować interfejs ShipPositionFluxService w klasie ShipPositionFluxServiceImpl tak aby powstał Flux<ShipPosition> zwracający położenie kolejno umieszczanych na planszy statków. W tym celu wykorzystać interfejs ShipPlacementService oraz listę Constants.SHIP_SIZES.

UWAGA: Operację umieszczania statku należy umieścić w oddzielnym wątku (ze względu na jej kosztowność)

- Zaimplementować metodę getShipsReadyMono() z interfejsu BattleService (w klasie BattleServiceImpl), tak aby powstało Mono<Void> sygnalizujące sukces w momencie gdy uda się rozmieścić wszystkie statki

Jeśli wszystko się powiedzie to gra wyświetli okno dialogowe informujące o tym że udało się rozmieścić statki i można rozpoczęć strzelanie.

Gra w okręty - zadania (2)

- Zaimplementować interfejs FieldFluxService w klasie FieldFluxServiceImpl tak aby powstał Flux<Field> zwracający pola w które oddawane są kolejne strzały przez użytkownika. W tym celu wykorzystać interfejs BoardMouseAdapter (zarejestrować tam FluxListener)
- Zaimplementować metodę getStampFlux () z interfejsu BattleService (w klasie BattleServiceImpl), tak aby powstał Flux<Stamp> zwracający wyniki strzałów (obiekty klasy Stamp oznaczające co GUI ma wyświetlić na planszy). Powstały Flux ma zgłosić sygnał końca, gdy wszystkie okręty zostaną zatopione

Jeśli wszystko się powiedzie to można uruchomić grę.

Gra w okręty – wskazówki do zadań

- Ostatnie zadanie jest już nieco bardziej złożone, gdyż wymaga zaimplementowania logiki przeliczającej czyste strzały na ich wyniki. Najlepiej tak zaimplementować klasę BattleServiceImpl aby trzymała listę statków jakie zostały położone na planszy a następnie dla każdego strzału sprawdzić czy spowodował jakieś skutki dla któregoś ze statków. Do tego celu można wykorzystać publiczne metody klasy Ship.
- Proszę pamiętać o tym, iż jeśli okręt zostanie zatopiony, to wówczas trzeba wysłać Stamp dla wszystkich pól wchodzących w jego skład (tak aby GUI oznaczyło cały okręt jako zatopiony a nie tylko pole, które było trafione jako ostatnie)
- Gra posiada licznik czasu, tak więc można rywalizować w kategorii szybkości zatopienia okrętów

Gra w okręty - zadania zaawansowane

- Zmodyfikować metodę `getStampFlux()` z interfejsu `BattleService` (w klasie `BattleServiceImpl`), tak aby powstawały Flux zwracał błędy gdy:
 - Minie za dużo czasu między poszczególnymi strzałami (użytkownik za dugo się namyśla)
 - Nie uda się zatopić wszystkich okrętów po ustalonej liczbie strzałów (jaką ilość przyjąć to już zostawiam Wam ☺)
- Zmniejszyć rozmiar planszy np. do 8x8. Co wtedy dzieje się z procesem umieszczania okrętów (podpowiedź - czasami zawisa) ? Czy można jakoś z tego wybrnąć ?

Pytania i dyskusja



Ankieta po warsztacie

<https://bit.ly/2BU4wKq>

Organizator

sages

Partner Strategiczny



Partnerzy



FINDWISE
SEARCH DRIVEN SOLUTIONS

e-point

 **GEPOL**

PRAGMATISTS

{j}DD



**HACK
YEAH**



SALEOMEGA.pl
Baza ekskluzywnego Konferencyjnego...
www.saleomega.pl

 **CLUSTER
COWORK**

 **COWORKING**
www.koscielnau.pl

 **Spotkania-IT**

We make IT  **JCommerce**



Dziękuję za uwagę!