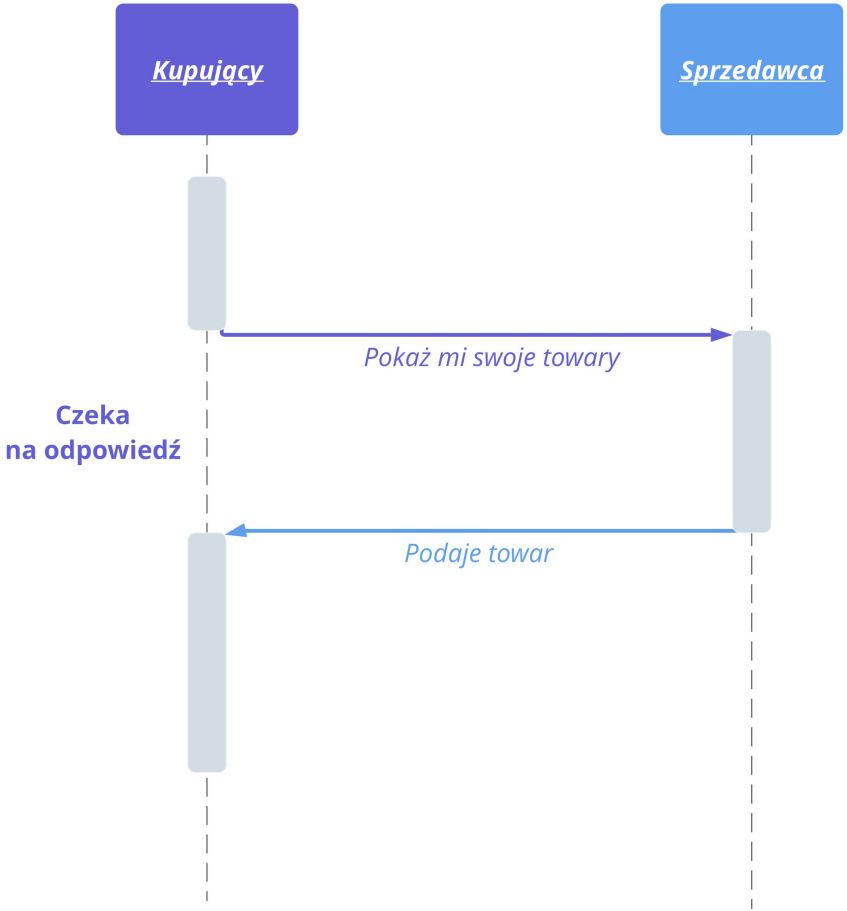
Przykłady do uzupełnienia:

https://upel.agh.edu.pl/mod/resource/view.php?id=263815

Reaktywne strumienie ReactiveX

Michał Idzik, Wydział Informatyki AGH
miidzik@agh.edu.pl

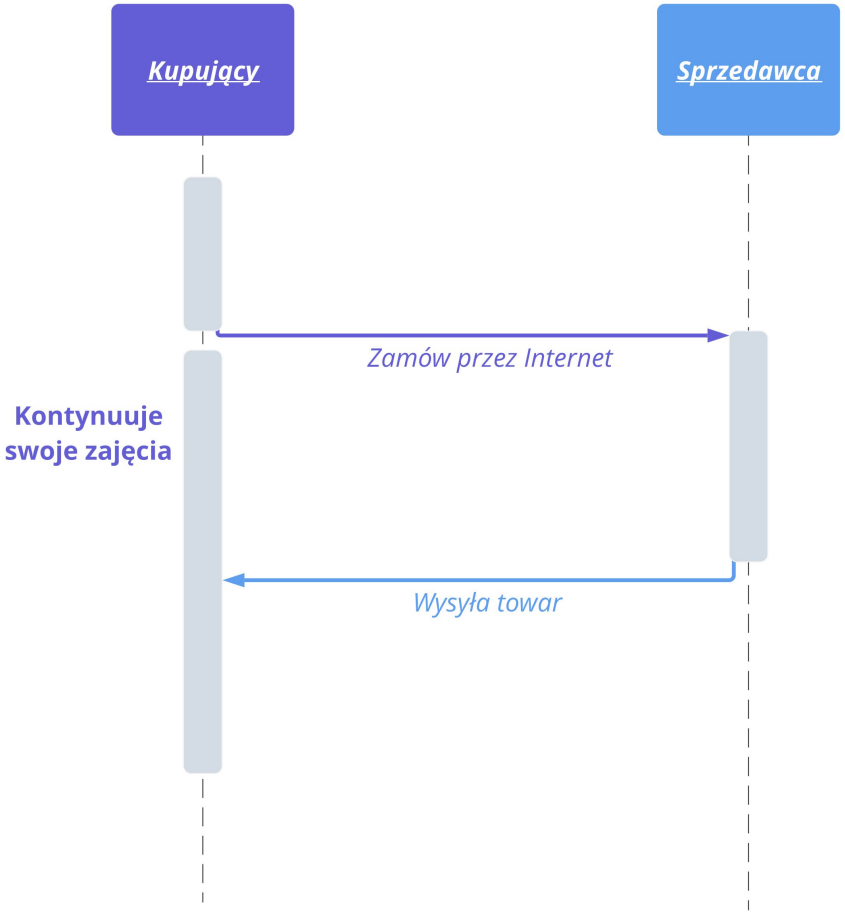


Operacja **synchroniczna**

Wywołanie operacji **blokuje** działanie programu.

Dopóki operacja nie zakończy się sukcesem i/lub nie zwróci wyniku, dalsze wywołanie programu **nie jest możliwe**.

Klasyczne wywołanie funkcji lub procedury w wielu językach programowania.



Operacja **asynchroniczna**

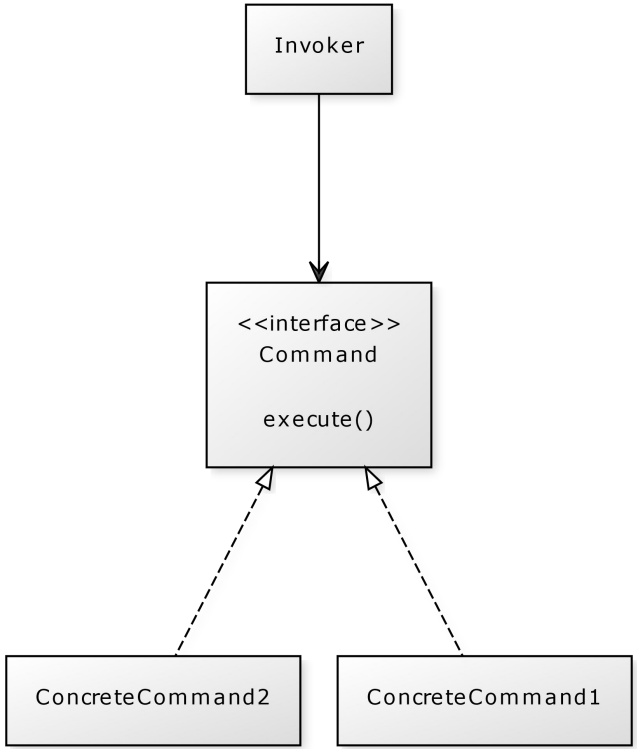
Wywołanie operacji **nie blokuje** działania programu.

Jeśli operacja zwraca rezultat, w momencie jej wywołania dostajemy jedynie **obietnicę**, że wkrótce pojawi się tam wynik.

Jeśli w danym momencie potrzebujemy wyniku operacji, możemy zablokować działanie programu i **poczekać** na rezultat.

Jak modelować asynchroniczne operacje obiektowo?

Podejście klasyczne: **Wątki**

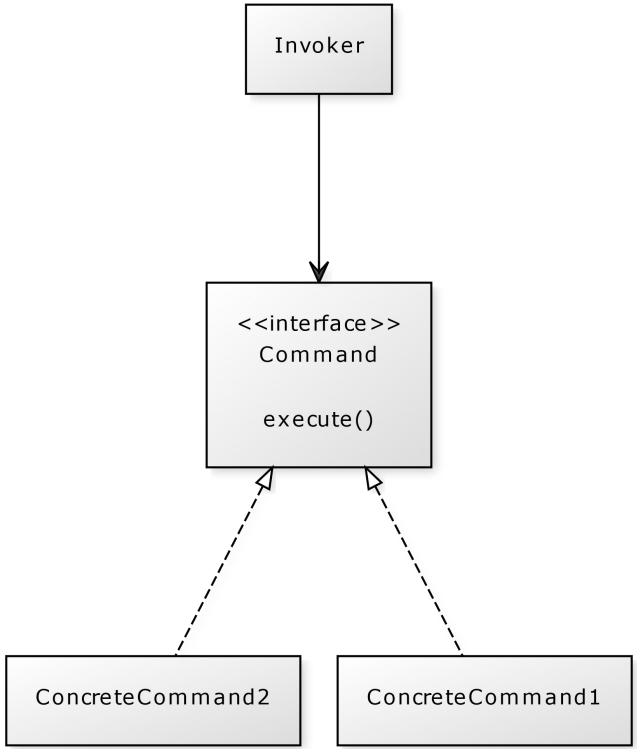


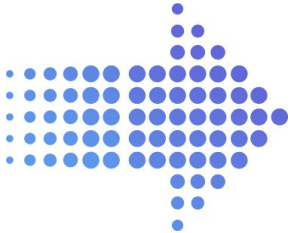
Wzorzec **Command**

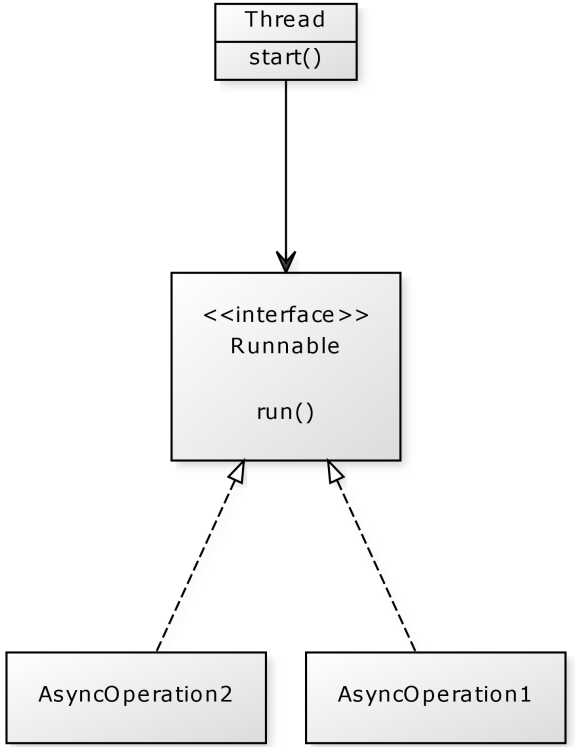
Opakowujemy dowolną logikę w klasę komendy. W ten sposób reprezentujemy funkcje jako klasy.

Wszystkie komendy mają wspólny interfejs (metoda *execute()*), wykonujący komendę nie musi znać jej przeznaczenia.

Java: **Runnable**







Zarządzanie wątkami jest trudne

- Sekcje krytyczne (współbieżny dostęp do zasobów)
- Konieczność synchronizacji i ręcznego zarządzania cyklem życia wątku
- Możliwość wystąpienia deadlocka lub zagłodzenia wątku
- ...i inne klasyczne problemy współbieżności, które trzeba samodzielnie rozwiązywać
- Wniosek: Interfejs do obsługi wątków jest niewystarczający?

Java 5: **Callable, Future**

- Część problemów można zredukować zmieniając filozofię obsługi asynchronicznych operacji

- Interfejs *Callable* w przeciwieństwie do *Runnable* może zwracać wartość □ wątek może kończyć się jakimś rezultatem

- Możliwość rzucenia wyjątku

```
public interface Runnable {
    public void run();
}
```

```
public interface Callable<V> {
          V call() throws Exception;
}
```

Java 5: **Callable, Future**

- Do obsługi *Callable* potrzeba dodatkowego narzędzia: *ExecutorService*
- *ExecutorService* operuje na puli wątków i automatycznie nimi zarządza
- Uruchomienie operacji *Callable* tworzy obiekt *Future*, program wywołuje się dalej
- Gdy potrzebujemy rezultatu, wykonujemy **blokująco** *get()* na obiekcie *Future*

```
Callable<Integer> asyncTask = () -> {
  // Do some calculations
  return 1234;
};

ExecutorService pool = Executors.newFixedThreadPool(4);

Future<Integer> resultFuture = pool.submit(asyncTask);

// Do some fun stuff here as long as you don't need the
result

// Block until result is ready
Integer result = resultFuture.get();
```

Jak modelować asynchroniczne operacje obiektowo?

Podejście funkcyjne:
Programowanie reaktywne

Java 8: **CompletableFuture**

- Co jeśli chcielibyśmy przetwarzać wyniki *Future* **nieblokująco**?
- *CompletableFuture* opakowuje wszystkie dotychczasowych mechanizmów w funkcyjne API
- Operatory do przetwarzania i łączenia obiektów *Future* i prostsze zarządzanie wątkami
- To już *prawie* współczesne programowanie reaktywne!

```
final CompletableFuture<Integer> resultFuture =
CompletableFuture.supplyAsync(() -> {
  // Do some calculations
  return 1234;
});

CompletableFuture<Void> anotherFuture = resultFuture
      .thenApply(x -> x * 2)
      .thenAccept(x -> System.out.println("My result is " +
x));

// Do some fun stuff here as long as you don't need the
result

// Block until task is completed
anotherFuture.get();
```

Zagadka: co wypisze program i kiedy wykonają się jego poszczególne części?

Programowanie reaktywne

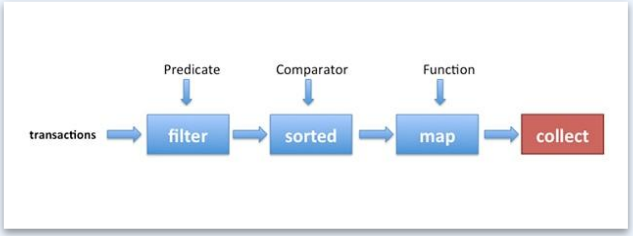
=

Asynchroniczne strumienie danych

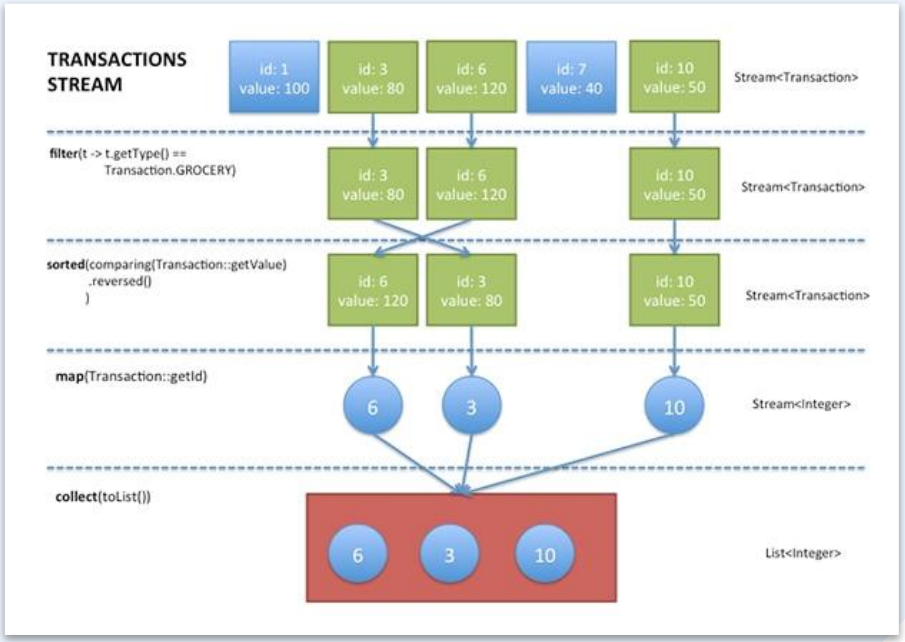
W programowaniu reaktywnym wszystko może być **strumieniem danych**

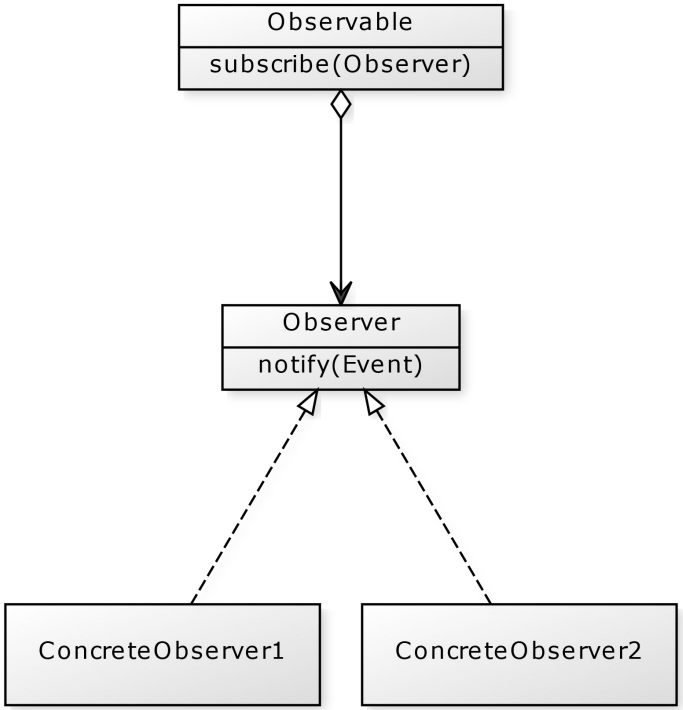
...strumieniem danych?

Java 8: **Stream**



- Podobnie jak kolekcja reprezentuje wiele obiektów tego samego typu.
- Udostępnia operatory przetwarzające kolejne elementy strumienia.
- Operatory mogą być łączone w łańcuch przetwarzania.
- Cały łańcuch wywołuje się dopiero po wykonaniu funkcji agregującej (np. *collect*).
- W przeciwieństwie do kolekcji nie musi trzymać wszystkich elementów w pamięci.





Wzorzec **Observer**

- *Observer* otrzymuje notyfikacje o zmianie stanu *Observable*.
- *Observable* przechowuje wszystkich swoich *Obserwerów*.
- Luźne powiązanie między obiektami: *Observable* nie wie o implementacjach swoich obserwatorów.


Iterator
Stream
Observer
Completable Future
Command
Callable
?

1 API do programowania reaktywnego

Bogata biblioteka operatorów do asynchronicznego przetwarzania obserwowalnych strumieni danych.

2 Stworzone w odpowiedzi na współczesne wyzwania

Koncept oparty o bibliotekę *Reactive Streams* **Microsoftu** (C#), rozpowszechniony dzięki inicjatywie firmy **Netflix**, która stworzyła projekt **RxJava.**

3 Wsparcie dla wielu języków programowania

Na bazie popularności **RxJavy** powstały wersje RX dla innych języków JVM (**RxScala**, **RxKotlin**, **RxClojure**), ale także inne, m. in. **RxJS**, **RxPY**, **Rx.NET**, **RxCpp**

4 Nowa filozofia programowania

Projekt *ReactiveX* przyczynił się do powstania wielu podobnych narzędzi, a nawet wpłynął na kształt API Javy 9. Wyznaczono nawet zbiór zasad programowania reaktywnego:
Reactive Manifesto

Reactive Manifesto

RESPONSIVE

System odpowiada tak szybko jak to możliwe. Jest to kluczowe dla jego użytkowników, ale także do sprawnego wykrywania sytuacji wyjątkowych.

MESSAGE DRIVEN

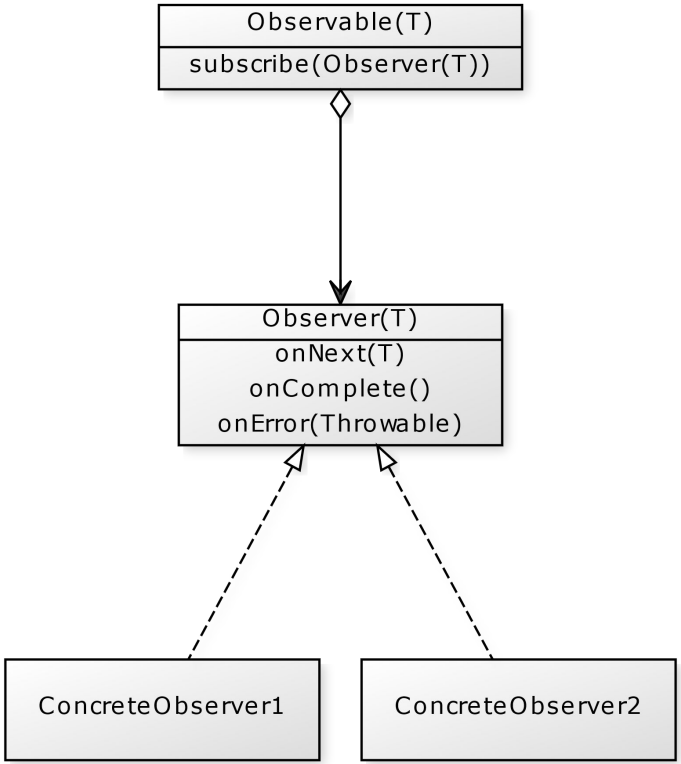
Sterowanie systemem oparte jest o asynchroniczne przekazywanie wiadomości i błędów między komponentami, zapewniając ich izolację i umożliwiając kontrolę obciążenia (*backpressure*).

RESILIENT

System pozostaje responsywny także w obliczu wystąpienia błędu. Błędy są powiązane z komponentami aplikacji działającymi w izolacji. Błąd w jednym komponencie nie przerywa pracy pozostałych.

ELASTIC

System pozostaje responsywny niezależnie od obciążenia. Dostarcza skalowalnych algorytmów, które potrafią replikować komponenty i rozdzielać między nie pracę.



Model **ReactiveX**

- *Observer* otrzymuje **kolejne elementy** typu **T**, **emitowane** przez *Observable*.
- Emisja kończy się gdy zostaje wywołana metoda **onComplete()** lub **onError(Throwable)**
- Połączenie wzorców **Observer** i **Iterator**

RxJava: **Jak to działa?**

```
Observable<Integer> source = Observable.just(0,1,2,3,4);
```

Observable<T>

Może emitować dowolną kolekcję elementów typu **T.**

```
source.subscribe(number -> System.out.println(2*number))
```

Observer<T>

Może być obiektem typu ***Consumer<T>*** realizującym **onNext(T).**

Tworzenie *Observable*

Just

Tworzy strumień na podstawie gotowych obiektów podanych explicite.

Empty/Never

Tworzy pusty strumień, który od razu się kończy.

From*

Przekształca obiekt na strumień zachowując jego specyfikę. Np. **fromIterable** utworzy strumień z kolekcji elementów, a **fromCallable** w momencie subskrypcji wywoła dostarczony obiekt Callable i zwróci w strumieniu jego wynik.

Create

Tworzy strumień według podanego przepisu. Daje największe możliwości kontroli, ale wymaga ostrożności – deweloper sam w odpowiednich momentach wywołuje na Observerze *onNext()*, *onComplete()* i *onError().*

Tworzenie *Observable*

Interval

Tworzy strumień wysyłający regularnie (co zadany kwant czasu) informację liczbową odpowiadającą liczbie wyemitowanych elementów. Może być wykorzystany do dalszych przekształceń.

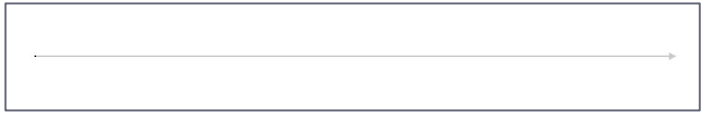
Timer

Tworzy strumień emitujący element po upłynięciu zadanego czasu. Może być wykorzystany do dalszych przekształceń.

RxJava: **obsługa błędów**


```
Observable<Integer> source = Observable.create(observer -> {
    for (int i = 1; i < 6; i++) {
        if (i != 3) {
            observer.onNext(i);
        } else {
            observer.onError(new Exception());
        }
    }
    observer.onComplete();
});
```

```
source.subscribe(number -> System.out.println(number),
                 error -> error.printStackTrace());
```

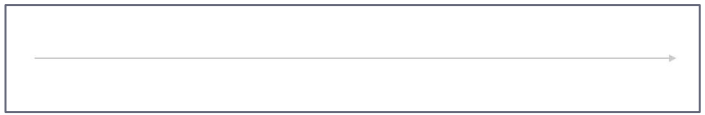


Observable

Może emitować elementy w dowolny sposób. Uwaga: operator **create() nie przerywa emisji** po wywołaniu **onError()**!

Observer

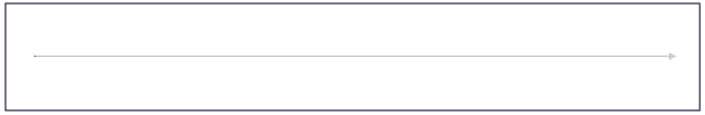
Może definiować również handler do obsługi błędów. **Przerywa subskrypcję** po otrzymaniu błędu.



RxJava: obsługa błędów

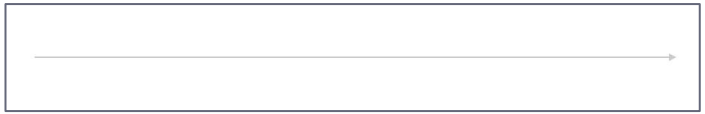

```
Observable<Integer> source = Observable.create(observer -> {
    for (int i = 1; i < 6; i++) {
        if (observer.isDisposed()) {
            break;
        }
        if (i != 3) {
            observer.onNext(i);
        } else {
            observer.onError(new Exception());
        }
    }
    observer.onComplete();
});
```

```
source.subscribe(number -> System.out.println(number),
                 error -> error.printStackTrace());
```



Observable

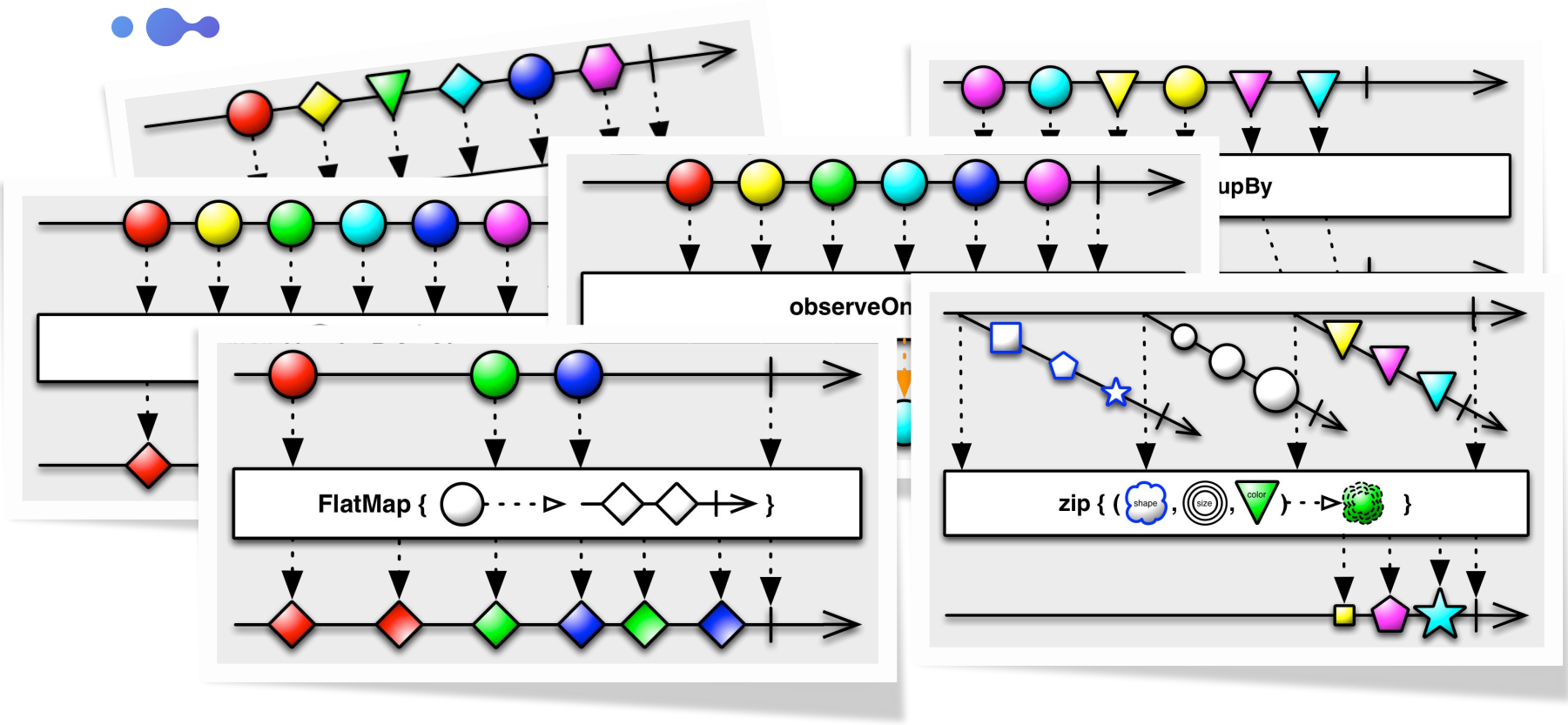
Może sprawdzić czy subskrypcja została zakończona aby nie emitować dalej elementów.



Operatory RX

- Rx dostarcza olbrzymiej biblioteki operatorów do przetwarzania strumienia danych.
- Operatory służą do przekształcania danych strumieniu, sterowania przepływem czy nawet kompozycji/rozdzielania wielu strumieni.
- Strumień operatorów można ułożyć w łańcuch wywołań, którego obsługa przypomina budowanie rurociągu.
- Zasada **Don't Break The Chain!** – staramy się nie wychodzić „poza strumień" i dokładać kolejne operatory zamiast rozpakowywać dane. Pomaga w tym operator *compose()*.

RxMarbles



RxJava: złożenie operatorów

Przykłady 5 - 6

```
Observable.just(0,1,2,3,4)
```

```
.filter(number -> number % 2 == 1)
```

```
.map(number -> "a" + number)
```

```
.subscribe(text->
System.out.println(text));
```

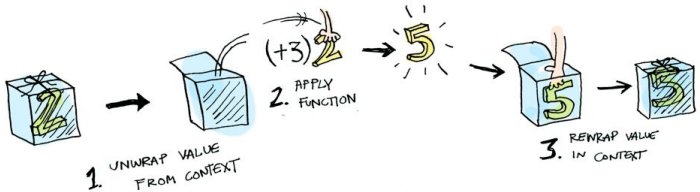
Złożenie operatorów: jak to działa?

Funktory

```
import java.util.function.Function;

interface Functor<T> {
  <R> Functor<R> map(Function<T, R> f);
}
```

Operacja funktora zawsze zwraca „nowe pudełko”. Funktory są więc *immutable*!



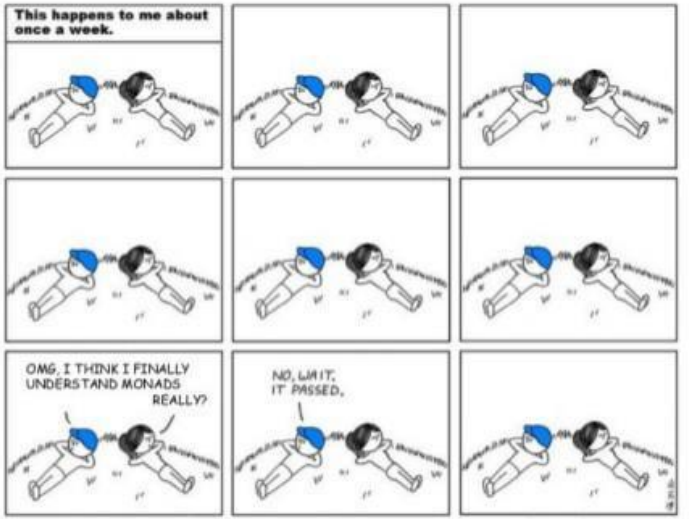
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Wniosek: ***Observable*** **jest funktorem!**

Złożenie operatorów: jak to działa?

Monady

https://blog.toggl.com/kill-dragon-comic/



https://medium.com/@nitinpatel_20236/what-does-the-phrase-monadic-bind-mean-a2184f34b2e3

Złożenie operatorów: jak to działa?

Monady

```
interface Monad<T,M extends Monad<?,?>> extends Functor<T,M> {
  M flatMap(Function<T,M> f);
}
```

Monada wypłaszcza dane za pomocą operatora FlatMap.
Spełniony warunek: **`m(x).flatMap(f) = f(x)`**

Złożenie operatorów: jak to działa?

Monady

Przykład: **Optional** (spoiler: w Javie też są monady!)

```
Person person = personMap.get("Name");
if (person != null) {
  Adress address = person.getAddress();
  if (address != null) {
    City city = address.getCity();
    if (city != null) {
      process(city)
    }
  }
}
```

```
Optional<Person> person = personMap.get("Name");
if (person.isPresent()) {
  Optional<Adress> address = person.getAddress();
  if (address.isPresent()) {
    Optional<City> city = address.getCity();
    if (city.isPresent()) {
      process(city)
    }
  }
}
```

```
personMap.find("Name")
  .flatMap(Person::getAddress)
  .flatMap(Address::getCity)
  .ifPresent(ThisClass::process);
```

Złożenie operatorów: jak to działa?

Monady

Optional (Java)

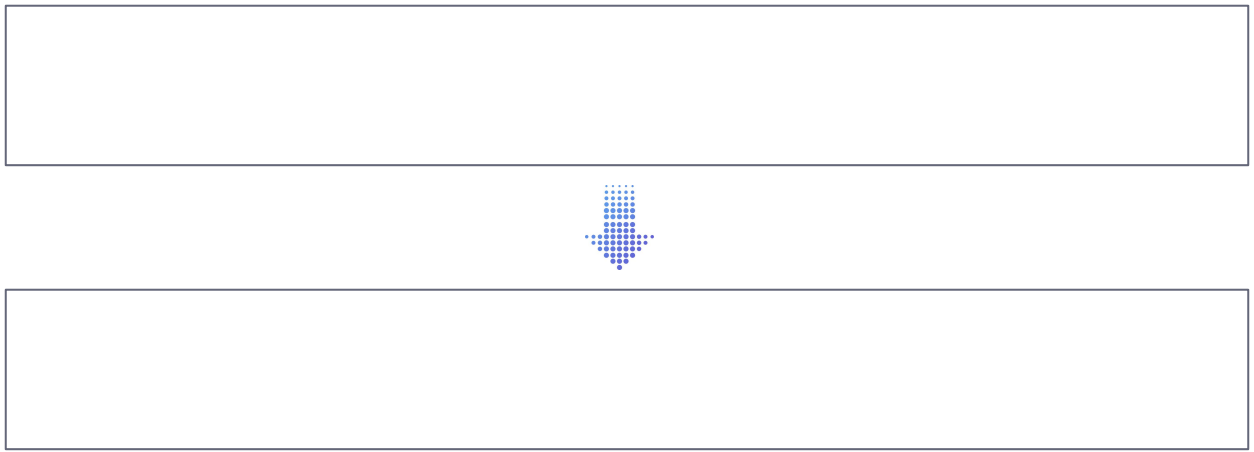
```
personMap.find("Name")
  .flatMap(Person::getAddress)
  .flatMap(Address::getCity)
  .ifPresent(ThisClass::process);
```

Null safe (Kotlin)

```
personMap.find("Name")
  ?.address
  ?.city
  ?.let { process(it) }
```

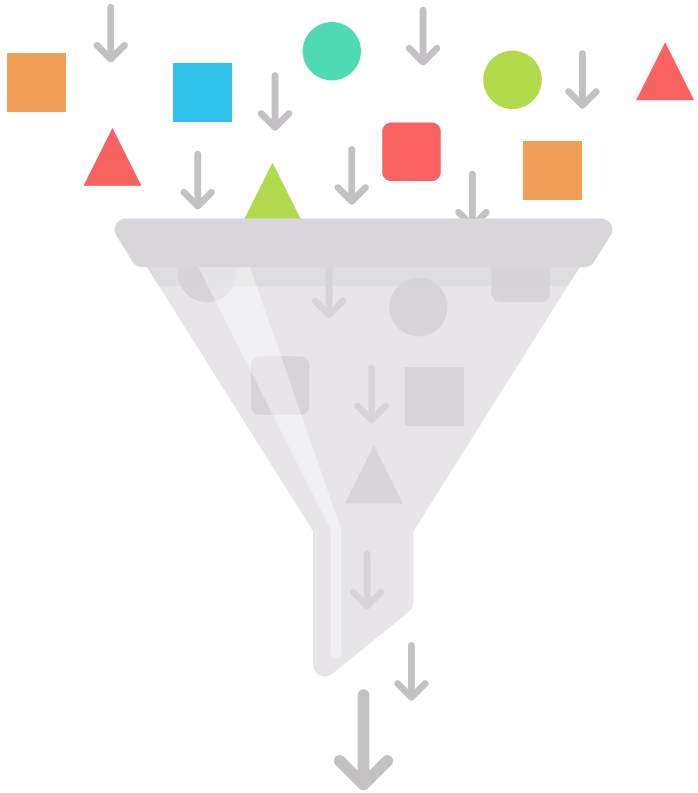
RxJava: **FlatMap**

```
Observable.just(1,2,3,4)
          .flatMap(x -> Observable.fromIterable(Collections.nCopies(x, x)))
          .subscribe(System.out::println);
```



Wniosek: ***Observable*** **jest także monadą!**

Łączenie strumieni RX



- Obiekty wielu strumieni można łączyć statycznymi operatorami **Merge()** oraz **Concat()**
- Często stosuje się do tego celu również operatory **FlatMap()**, **ConcatMap()** nie przerywając łańcucha wywołań.
- Operatory typu Concat()/ConcatMap() zachowują kolejność: najpierw wywołują pierwszy strumień, potem drugi, itp.
- Operatory Merge()/FlatMap() nie gwarantują kolejności, uruchamiając subskrypcję na wszystkich strumieniach jednocześnie.
  W aplikacji jednowątkowej działają identycznie jak Concat(), w przypadku strumieni działających w różnych wątkach uzyskujemy efekt równoległego przetwarzania.

RxJava: łączenie strumieni

Przykłady 8, 11, 12

o1:

o2:

```
Observable<Integer> o3 = Observable.merge(o1, o2);
```

o3:

Wielowątkowość w RxJava: **Schedulers**

computation()

Uruchamia strumień na jednym z wątków z dostępnej uniwersalnej puli. Pula jest zoptymalizowana pod obliczenia więc ma rozmiar równy liczbie dostępnych rdzeni.

io()

Zoptymalizowany pod operacje I/O. Uruchamia strumień w nowym wątku, ale wykorzystuje już wcześniej utworzone wątki, które akurat się zwolniły.

newThread()

Uruchamia strumień na nowym dedykowanym wątku. Uwaga: może prowadzić do niekontrolowanego wzrostu liczby wątków w aplikacji.

from(Executor)

Deleguje uruchamianie strumienia do klasycznego Executora, który sam zarządza cyklem życia wątków.

Wielowątkowość w RxJava: **subscribeOn**

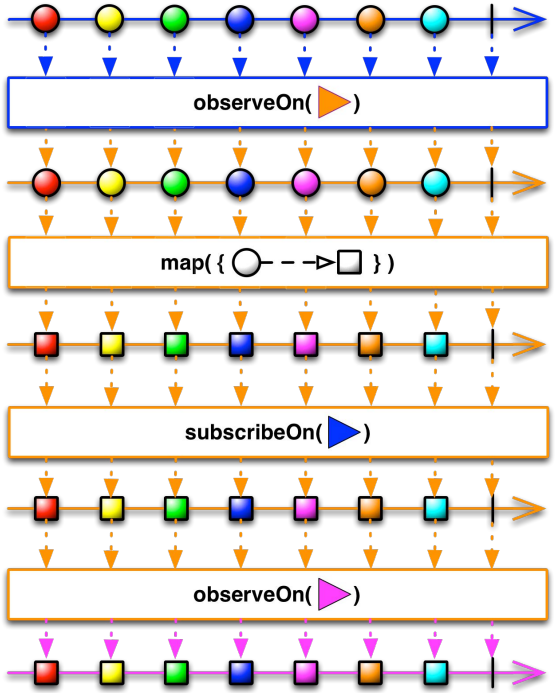
- **Konfiguracja bezpośrednio w łańcuchu operacji RxJava**
- Wywołanie **subscribeOn()** sprawia, że **zarówno** emisja obiektów, jak i ich obsługa przez subskrybentów odbywa się na danym **Schedulerze**!
- Umiejscowienie operatora subscribeOn() w łańcuchu wywołań nie ma znaczenia. Jeśli zdefiniujemy wiele wywołań subscribeOn() w jednym łańcuchu tylko pierwsze zostanie wzięte pod uwagę.

Uwaga: ustawienie *Schedulera* wpływa na **cały strumień** i nie sprawia, że jego elementy przetwarzane są równolegle! Tego typu efekt można osiągnąć używając operatorów **merge()** lub **flatMap()**.

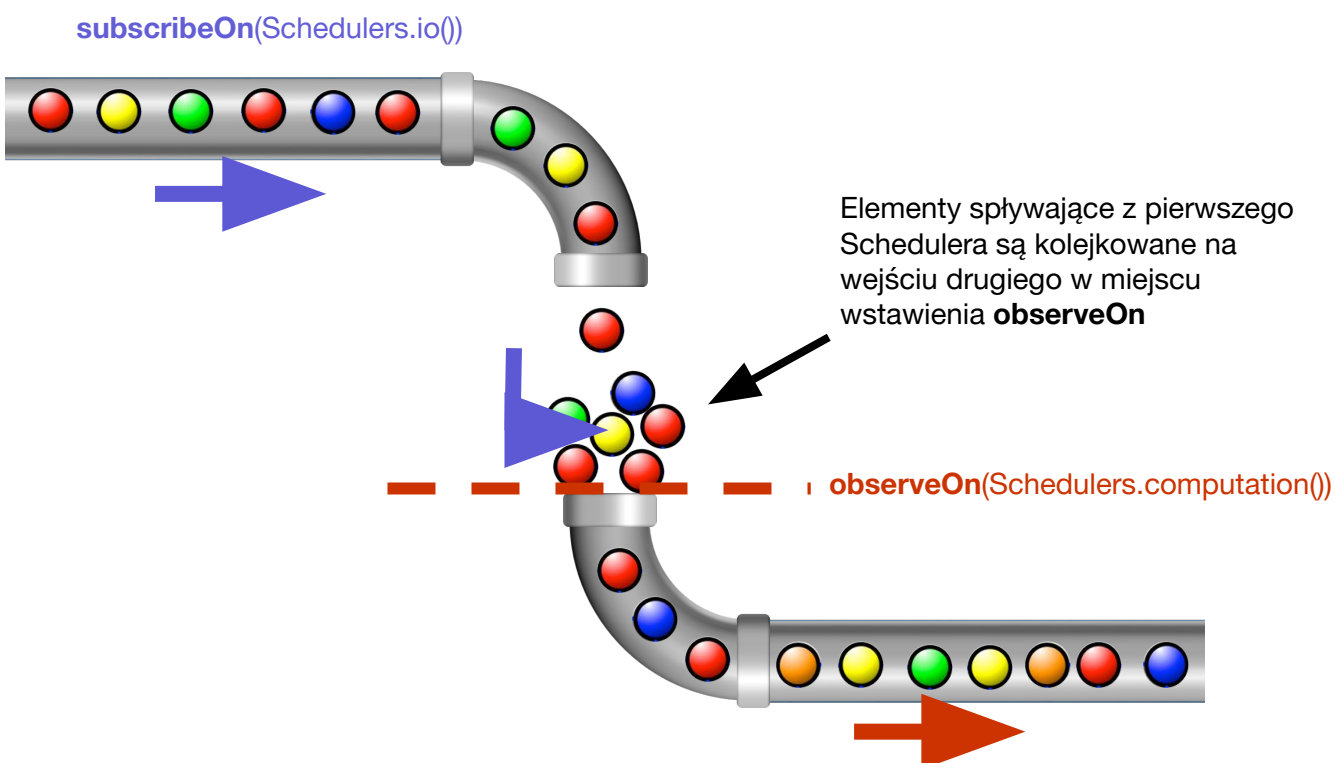
```
Observable.just(0,1,2,3,4)
.map(number -> "a" + number)
.subscribeOn(Schedulers.io())
.subscribe(text-> System.out.println(text));
```

Wielowątkowość w RxJava: observeOn

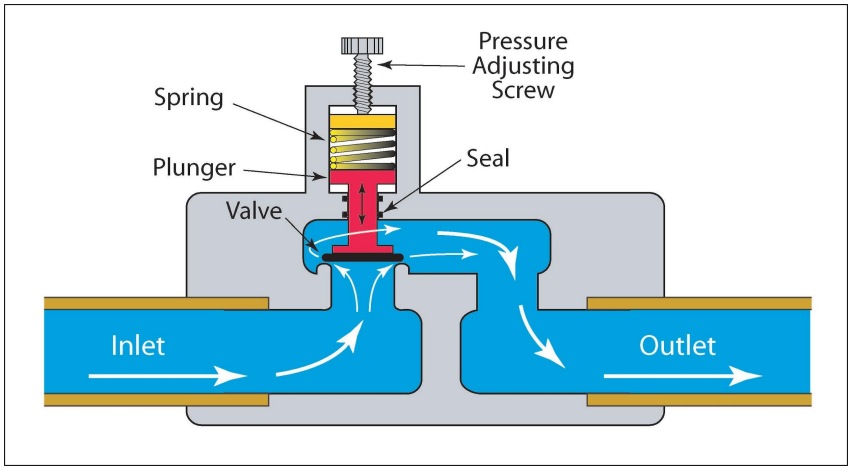
- Zmienia **Scheduler**, na którym mają się wykonać operatory w strumieniu poniżej (*downstream*). Dotyczy to także potencjalnych subskrybentów.
- W przeciwieństwie do **subscribeOn()** miejsce, w którym umieszczamy **observeOn()** w strumieniu ma kluczowe znaczenie.
- Możemy zdefiniować wiele wywołań observeOn() w jednym strumieniu. Każde z nich przełącza Scheduler.
- Każde wywołanie observeOn() tworzy w danym miejscu kolejkę, do której lądują elementy z górnej części strumienia (*upstream*).



Wielowątkowość w RxJava: **observeOn**



Wielowątkowość w RxJava: **Backpressure**



https://techblog.ctgclean.com/2012/04/valves-backpressure-regulating-valves/

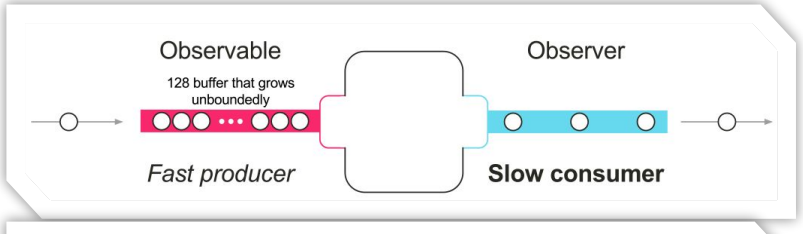
Jeśli Observable emituje dane szybciej niż Observer jest w stanie je przetwarzać, pojawiają się problemy z pamięciowym obciążeniem, które mogą doprowadzić do spadku responsywności systemu.

Aby temu zaradzić, RX wprowadza mechanizm **Backpressure**, który umożliwia:

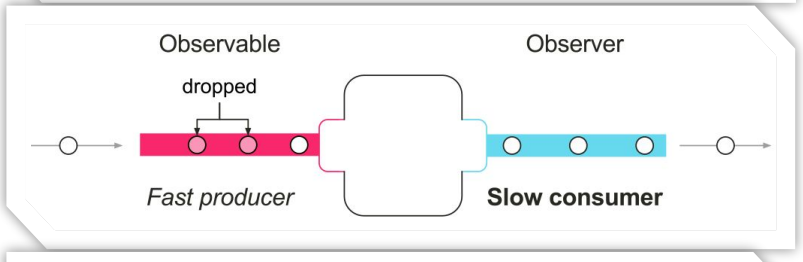
- sterowanie Observablem z poziomu Observera przez tzw. ***reactive pull***
- zastosowanie jednej z strategii radzenia sobie z obciążeniem gdy *reactive pull* nie jest wspierany przez Observable.

Wielowątkowość w RxJava: **Backpressure**

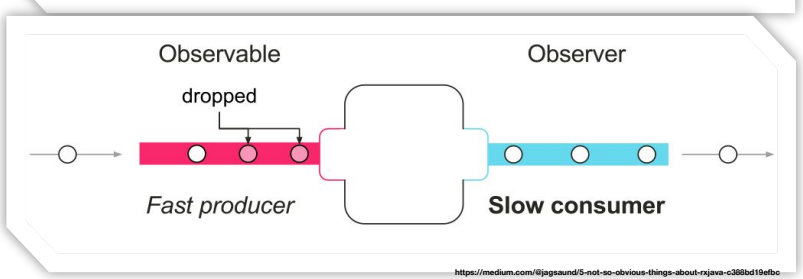
onBackpressureBuffer()



onBackpressureDrop()



onBackpressureLatest()



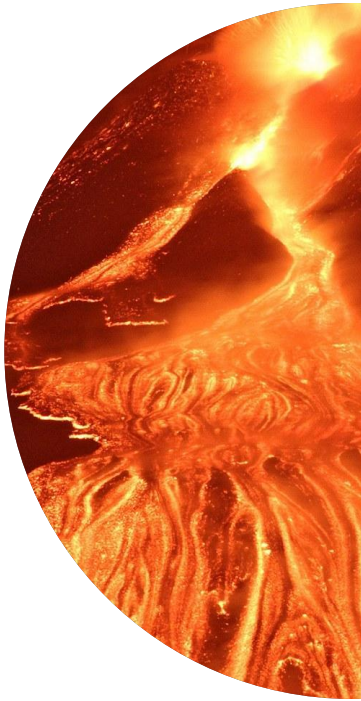
Zimne vs gorące Observable

Zimne Observable (*Cold Observables*) uruchamiają emisję od nowa dla każdego kolejnego subskrybenta. Można w ten sposób wielokrotnie używać raz zdefiniowanych strumieni. Domyślnie każdy Observable jest zimny (wszystkie dotychczasowe przykłady).

Gorące Observable (*Hot Observables*) mogą emitować dane do wielu subskrybentów jednocześnie. Każdy nowy subskrybent dołącza w określonym momencie emisji i nie otrzymuje elementów, które zostały już wcześniej wyemitowane. Jest to odpowiednik klasycznej realizacji wzorca Observer (jeden Obserable notyfikujący wielu Observerów).

Aby przekształcić zimne Observable na ciepłe, wystarczy użyć metody *publish()*.

Patrz też **RX Subjects**



RxJava2: **Nowe typy Observable**

Flowable<T>

Działa jak *Observable*, ale dodatkowo wspiera strategie *Backpressure* (w RxJava2 „klasyczne” *Observable* nie oferują tego mechanizmu).

Single<T>

Zwraca dokładnie jeden element typu **T** (lub kończy się błędem).

Completable

Nie zwraca żadnych wyników, ma jedynie początek i koniec (lub rzuca błędem). Analogia: *Runnable* / metod void.

Maybe<T>

Zwraca dokładnie jeden element **T** lub nic (lub kończy się błędem). Połączenie *Single* i *Completable*. Analogia: *Optional*

Dlaczego programowanie reaktywne?

Współbieżność łatwa w realizacji

Asynchroniczne operacje w złożonych problemach mogą być trudne w realizacji na wątkach (sekcje krytyczne, synchronizacja). Wzorce programowana wymuszają bezpieczne pisanie programów.

Bogaty zestaw operatorów

Wiele skomplikowanych operacji związanych z przepływem sterowania programu można wyrazić pojedynczym operatorem RX.

Czytelny, funkcyjny kod

Strumienie danych porządkują kod dzieląc logikę na wywołania operatorów, które znajdują się na tym samym poziomie abstrakcji. Każdy operator ma jasno sprecyzowane wejście i wyjście.

Wbudowany mechanizm *backpressure*

Dzięki strumieniom RX możemy nie tylko modelować logikę aplikacji, ale też zarządzać obciążeniem subskrybentów.

Dlaczego programowanie reaktywne?

Kompozycja strumieni

Strumienie można przetwarzać, ale również łączyć ze sobą, rozdzielać czy wypłaszczać. Daje to nieograniczone możliwości operowania na danych różnego typu w różnych ilościach.

Asynchroniczna obsługa błędów

Wyjątki w świecie asynchronicznych operacji nie mogą być przechwytywane w tradycyjny sposób (try-catch). RX dostarcza własne mechanizmy, które pozwalają reagować na błędy w podobny sposób jak na prawidłowe dane.

Trudne w debugowaniu

Opóźnione wywołanie i szerokie stosowanie funkcyjnych elementów (wyrażenia lambda) utrudnia sterowanie debugerem i odnajdywanie miejsc, w których faktycznie leży problem.

Duży próg wejścia dla nowych deweloperów

RX wymaga zmiany podejścia i myślenia o projektowaniu programów. Dla programistów przyzwyczajonych do imperatywnego wyrażania rozwiązań może wydawać się to początkowo trudne.

Kto używa **ReactiveX?**

RxJava + Retrofit

```
interface MyService {
  @GET("/user")
  Observable<User> getUser();
}
```

Alternatywy dla ReactiveX?

Project Reactor

SPRING

Akka Streams

NA BAZIE AKKA ACTORS

Kotlin Coroutines

OFICJALNA BIBLIOTEKA JĘZYKA KOTLIN

“Jedno API by wszystkie połączyć”

1. **Java 9 Flow API** – zbiór interfejsów (bez implementacji!) realizujących *Reactive Manifesto* na bazie dotychczasowych doświadczeń.

2. Napisanie własnej implementacji jest bardzo trudne, ale istnieje kilka gotowych (**RxJava**, **Akka Streams**, **Spring Reactor**).

3. Możliwość używania wielu implementacji w jednym programie naprzemiennie (np. kombinacja RxJava – Reactor).

Warto poczytać

- RxJava w pigułce (tutorial):
  https://www.infoq.com/articles/rxjava-by-example/
- Wielowątkowość w RxJava:
  http://tomstechnicalblog.blogspot.com/2016/02/rxjava-understanding-observeon-and.html
- Więcej o gorących/zimnych Observables:
  https://github.com/Froussios/Intro-To-RxJava/blob/master/Part%203%20-%20Taming%20the%20sequence/6.%20Hot%20and%20Cold%20observables.md
- Case Study Netflixa:
  https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a
- Monady w Javie:
  https://dzone.com/articles/whats-wrong-java-8-part-iv
  https://dzone.com/articles/functor-and-monad-examples-in-plain-java
- Szczegółowy tutorial do Optionali w Javie:
  https://www.baeldung.com/java-optional
- Kotlin Coroutines (tutorial):
  https://proandroiddev.com/kotlin-coroutines-channels-csp-android-db441400965f

Pytania? Potrzebna pomoc?

miidzik@agh.edu.pl
(albo MS Teams)