ReactiveX – programowanie reaktywne w niemalże każdym języku i systemie

Programowanie reaktywne, razem z funkcyjnym, już od lat jest zapowiadane jako paradygmat przyszłości. O tym, na ile jest ono praktyczne, decyduje jednak głównie wsparcie od strony biblioteki lub języka programowania. W ostatnich latach powstało kilka bibliotek, które wspierają programowanie reaktywne. Pośród nich, szczególnie w świecie Javy i JavaScript, wyróżnia się ruch ReactiveX. Zapewnia on rozbudowane i funkcjonalne biblioteki z tymi samymi narzędziami dla wielu różnych języków i platform.

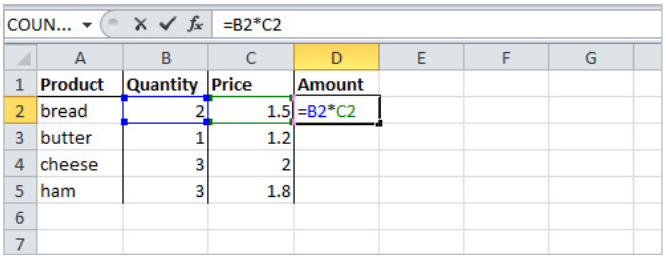
Sam ruch ReactiveX koncepcyjnie nie jest nowością. Może się on wręcz wydawać pewnym déjà vu osobom, które mają doświadczenia z Akka, Rabbit czy Reaktor. Bibliotek, które konkurują o wprowadzenie reaktywności do Javy, jest kilka. Na szczęście powstała także inicjatywa, by wznieść się w Javie ponad te różnice. Mam na myśli Reactive Streams, która działa, by wprowadzić do Javy 9 strumienie, które mogłyby współpracować z tymi bibliotekami. Pokazuje nam to, jak duże jest zainteresowanie wprowadzeniem reaktywności do popularnych języków programowania. Jednocześnie fakt, że biblioteki te działają w większości niemalże identycznie, pokazuje, jak dojrzała jest to technologia. Pojawia się jednak pytanie: Skoro jest tak wiele bibliotek, to czemu akurat skupiać się na RxJava? Głównymi powodami, dlaczego uważam, że to właśnie jej należy poświęcić ten artykuł, jest jej uniwersalność i charakter. Jest ona używana w wielu językach (ma wsparcie dla 13 popularnych języków, w tym dla JavaScript, Java, C#, PHP, Ruby, C++, Python i Swift). Społeczność ReactiveX ma ambicje, by wspierać wszystkie popularne platformy i języki programowania. Jest ku temu na dobrej drodze. Nie jest także zależna od frameworka (jak Akka czy Spring Cloud Stream). Także sposób, w jaki RxJava jest napisana, świadczy o jej dojrzałości. Posiada dużą liczbę metod, a przy tym jest na tyle intuicyjna, że większość osób, które korzystały z innej biblioteki do programowania reaktywnego, nie powinna mieć trudności z jej zrozumieniem.

W tym artykule zaprezentuję bibliotekę ReactiveX i pokażę, co może ona wnieść do projektu. Skupię się przy tym na przykładach konkretnych zastosowań. Zacznę od minimalistycznego wstępu do programowania reaktywnego, mającego pomóc czytelnikowi wejść w temat.

Zdecydowałem się prezentować przykłady w Kotlinie, ponieważ jest on bardzo podobny do Javy, ale bardziej zwięzły i czytelny. Same metody i funkcje pochodzą jednak z RxJava, ponieważ Kotlin jest z Java w pełni kompatybilny. W zasadzie biblioteka RxKotlin dostarcza tylko dodatkowe metody, charakterystyczne dla Kotlina (nie będziemy z nich korzystali). Warto jednak zauważyć, że w innych językach obsługiwanych przez ReactiveX kod ten będzie wyglądał niemal identycznie. Funkcje są wszędzie takie same i różnica tkwi głównie w elementach składowych języka.

PROGRAMOWANIE REAKTYWNE

Programowanie reaktywne jako paradygmat jest bardzo szeroką koncepcją. W świecie komputera najbardziej znaną jej implementacją są komórki w programie Microsoft Excel. Definiujemy w nich formuły, które obliczają wartość na podstawie innych komórek. Gdy jednak któraś z wartości tych komórek ulegnie zmianie, to zmieniają się wartości wszystkich komórek od niej zależnych. Jeśli więc komórka zależy od innych, to będzie reagowała zmianą swojego stanu na zmianę stanu dowolnej komórki, od której zależy. Nie wiem, jak realnie działa to w Excelu, ale moim zdaniem najbardziej intuicyjną implementacją byłoby ustawienie po zmianie formuły obserwatorów (observer lub listener) na komórkach, od których wartość zależy. Tak by dowolna ich zmiana powodowała ponowne obliczenie wartości komórek, na które wpływają. Jak widać, programowanie reaktywne jest silnie powiązane ze wzorcem obserwatora. Na końcu artykułu przedstawiony jest kod definiujący zachowanie podobne do komórek programu Microsoft Excel.



Rysunek 1. Komórki w programie Microsoft Excel (źródło: http://www.excel-easy.com/functions/cell-references.html)

Programowanie reakcyjne jest w pewien sposób odwróceniem kierunku przepływu danych. Propagacja zmian zaczyna się wtedy, kiedy zmieniony zostanie obiekt, od którego zależą inne. Dotyczy także wyłącznie tych, które od zmienionych obiektów zależą.

W praktyce programistycznej programowanie reaktywne wygląda różnie dla różnych języków programowania i bibliotek. Mimo wielu różnic nieodłącznymi jego elementami są:

- Strumienie danych (ang. *dataflow*) – jest to sposób programowania, gdzie definiujemy kolejne kroki przetwarzania danych. W kolejnych krokach przyjmuje się jakieś dane i zamienia się je na inne dane. Poszczególne kroki powinny być bezstanowe. Często do przetwarzania danych wykorzystuje się uniwersalne metody, które jako argument przyjmują funkcję lambda. Typowym przykładem jest strumieniowe przetwarzanie list.
- Propagacja zmian – wcześniej przedstawiona na przykładzie komórek Excela. Polega zwykle na tym, że to obiekty definiu-

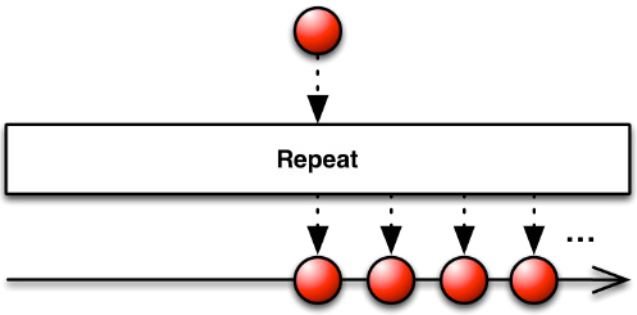
ją, jak obliczany jest ich stan i od czego zależy. W ten sposób zmiana jest propagowana.

Warto zauważyć na tym etapie, że programowanie reaktywne dobrze zgrywa się z wielowątkowością. Zarówno strumienie danych, jak i propagacja zmian nadają się do uruchomienia na wielu wątkach. Był to istotny powód popularyzacji paradygmatu reaktywnego.

W praktyce programistycznej częściej używa się strumieni danych i to wokół nich skupione są funkcjonalności ReactiveX. Tworzony jest obiekt, który jest w stanie wyemitować zdarzenie, czyli nic innego jak jakieś dane, które mogą być przekazane dalej. Dla przykładu, możemy utworzyć obiekt, który w odpowiedzi na kliknięcie przycisku wyemituje zdarzenie.

Żeby jednoznacznie nazywać tego typu obiekt, będę używał w tym artykule nazwy `Observable`. Jest to odpowiednik nazwy `Publisher` z nomenklatury Javy 9 (Reactive Streams). Wysłane zdarzenie powinno zostać obsłużone. Wcześniej jednak można na nim wykonywać liczne operacje strumieniowe.

Poszczególne operatory w świece ReactiveX przyjęło się dokumentować dodatkowo za pomocą tzw. Marble Diagrams. Jeden z nich przedstawiony został na Rysunku 2. Metody są tu reprezentowane jako bloki, gdzie nad nimi i pod nimi znajduje się przykładowa oś czasu z zaznaczonymi zdarzeniami. W dalszej części przedstawione będą inne Marble Diagrams, by ułatwić zrozumienie bardziej skomplikowanych funkcji.



Rysunek 2. Marble Diagram metody repeat. Zamienia ona Observable (którego jest metodą) na takie, które po odebraniu zdarzenia emituje je wielokrotnie co pewien odstęp czasu (źródło: www.reactivex.io)

Przejście do programowania reaktywnego dostarcza szereg korzyści:

1. Obsługa przepływu zawarta jest w jednym miejscu, dzięki czemu dobrze widać, co się dzieje w aplikacji. Dodatkowo biblioteki takie jak ReactiveX posiadają szereg funkcji związanych z definiowaniem tego strumienia, dzięki którym możemy tworzyć bardziej rozbudowaną logikę (na przykład oczekiwać na kilka odpowiedzi HTTP i wyemitować zdarzenie dopiero po otrzymaniu ostatniej).
2. Wynosimy obsługę wątku, na którym działa kod wyżej, do obsługi strumienia. Wielowątkowości staje się więc bardzo prosta.
3. Tworzone bloki są naturalnie oddzielone od tego, jak oddziałują z resztą kodu. Dzięki temu częściej nadają się do ponownego użycia oraz są łatwiejsze do analizy.

Zdania te mogą brzmieć abstrakcyjnie, ale w dalszej części postaram się przybliżyć te korzyści poprzez konkretne przykłady.

PRZETWARZANIE STRUMIENIOWE

Zanim przejdę do bardziej praktycznych przykładów działania biblioteki RxJava (i innych z ReactiveX), chciałbym tytułem wstępu pokazać kilka prostszych przykładów jej działania. Przetwarzanie strumieniowe w RxJava wygląda bardzo podobnie do wprowadzonego w Javie 8 strumieniowego przetwarzania kolekcji. Przykładowe przetwarzanie strumieniowe w RxJava przedstawione zostało w Listingu 1. W Listingu 2 pozostawione zostało natomiast analogiczne przetwarzanie zrealizowane przy pomocy narzędzi przetwarzania strumieniowego wprowadzonych w Javie 8.

Listing 1. Przykładowe przetwarzanie strumienia zdarzeń w RxJava, które wygląda podobnie do przetwarzania kolekcji znanego z Kotlina, Scali lub Javy 8

```
Observable.range(1, 10) // 1
  .map { it * 2 } // 2
  .sumInteger() // 3
  .subscribe { println(it) } // 4
// sum = 110
```

1. Funkcja `range` generuje zdarzenia kolejno dla liczb z podanego w argumencie zakresu.
2. Metoda `map` mapuje każde zdarzenie na inne.
3. Metoda czeka na ostatnie zdarzenie i wtedy liczy sumę dla wszystkich otrzymanych zdarzeń.
4. Metoda `subscribe` wykonuje podaną w argumencie operację dla każdego odebranego zdarzenia. Tutaj jest to wyświetlenie sumy.

Listing 2. Działania analogiczne do Listingu 1, ale wykonywane na strumieniach Java 8

```
long sum = IntStream.range(0, 10)
  .asLongStream()
  .map((i) -> i*2)
  .sum();
System.out.println(sum);
```

Istotną różnicę między przetwarzaniem kolekcji a zdarzeń możemy zaobserwować, gdy wprowadzimy opóźnienie między elementami. W Listingu 3 zaprezentowany został przykład, w którym, dzięki wprowadzeniu opóźnienia przez funkcję `delay`, otrzymamy efekt odliczania.

Listing 3. Przykład kodu, który co sekundę wywołuje funkcję showCountDownNumber z kolejnymi liczbami od 10 do 1

```
Observable.from(10 downTo 1) // 1
  .concatMap { s -> Observable.just(s).delay(1, SECONDS) } // 2
  .subscribe { num -> showCountDownNumber(num) }
```

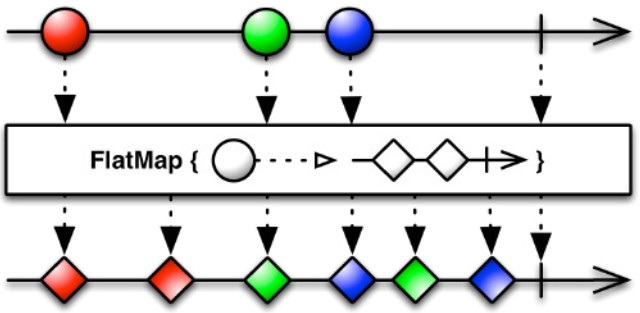
1. Funkcja `from` tworzy kolejne zdarzenia dla kolejnych elementów kolekcji.
2. Metoda `delay` wprowadza opóźnienie przed zdarzeniem, a metoda `concatMap` zmusza eventy, by każdy zaczekał na swojego poprzednika.

Widać tutaj, że w strumieniach ReactiveX kluczowy jest czas nadejścia zdarzenia. Każdy z kroków przetwarza zdarzenie zaraz po jego otrzymaniu. Część zawarta w `concatMap` przetrzymuje je jednak i puszcza po kolei, a do tego metoda `delay` wprowadza opóźnienie.

Spójrzmy na bardziej praktyczny przykład. Załóżmy, że dostaliśmy zadanie, by na kliknięcie przycisku wysyłane było zapytanie o jakiś element, po czym byłby on wyświetlony w określonym polu. W większości projektów realizuje się tego typu zadanie poprzez zagnieżdżenie callbacków. Przykładowy kod znajduje się w Listingu 4.

Listing 4. Klasyczna realizacja ciągu zdarzeń zrealizowana przez zagnieżdżenie callbacków. Funkcje thread i uiThread zmieniają wątek zagnieżdżonego wywołania odpowiednio na nowy wątek oraz na wątek główny

```
addOnPageLoadedListener {
  button.setOnClickListener {
    thread {
      loadElement() { elem ->
        uiThread {
          showElement(elem)
        }
      }
    }
  }
}
```



Rysunek 3. Działanie funkcji flatMap na Marble Diagram (źródło: http://reactivex.io/)

Już na tak małym przykładzie widać, że wielokrotne zagnieżdżanie nie jest dobre. Największy problem występuje zwykle w JavaScript (zwłaszcza w Node.js), gdzie programiści wielokrotne zagnieżdżenia nazywają kolokwialnie „piekłem callbacków" (ang. *callback hell*). Nie bez powodu – zmniejszają one czytelność kodu i utrudniają wprowadzanie zmian. Wydzielenie każdego poziomu do osobnej funkcji niewiele zmienia. Takie funkcje nie nadawały się raczej do ponownego użycia, ponieważ zawierałyby w sobie dalszą część przepływu aplikacji. Także obserwacja tego, co każda z nich robi, wymagać będzie schodzenia w dół do coraz bardziej szczegółowych funkcji. Ten sposób działania nie stanowi więc realnej pomocy dla programisty.

ReactiveX daje jednak alternatywę. Zamiast zagnieżdżać wywołania, stworzymy `Observable` i ustawimy je w strumień. Każdy z nich uruchamiany będzie w odpowiedzi na zdarzenie wysłane przez poprzedni. Definicja takiego strumienia w RxJava przedstawiona jest w Listingu 5.

Listing 5. Realizacja ciągu zależnych zdarzeń w RxJava. Oczekiwanie na załadowanie strony, nasłuchiwanie na naciśnięcie przycisku, ładowanie elementu z API oraz docelowo wyświetlenie go przy pomocy funkcji showElement

```
onPageLoadedObservable()
  .flatMap { onClickedObservable(button) }
  .observeOn(Schedulers.newThread())
  .flatMap { loadElementObservable() }
  .observeOn(Schedulers.io())
  .subscribe(
    { elem -> showElement(elem) }, // onNext
    { error -> logError(error) } // onError
  )
```

W przykładzie wykorzystana została funkcja `flatMap`. Przyjmuje ona jako parametr funkcję lambda, która zwraca nowy obiekt `Observable`. Dla każdego otrzymanego zdarzenia funkcja `flatMap` wywołuje funkcję z argumentu i tworzy nowego `Observable`. Obiekt `Observable`, który jest zwracany przez `flatMap`, będzie emitował wszystkie zdarzenia z utworzonych przez `flatMap` obiektów `Observable`. Zostało to przedstawione na Rysunku 3. W większości przypadków mamy do czynienia z sytuacją, gdy pierwszy `Observable` generuje tylko jedno zdarzenie (jak w Listingu 2). Wtedy dodawanie kolejnych `Observable` przez `flatMap` tworzy klasyczne przetwarzanie strumieniowe. Później zobaczymy jednak, że funkcja `flatMap` ma znacznie większe możliwości.

Łatwo zauważyć, że pojawiły się jednak tutaj elementy, których wcześniej nie było. Przede wszystkim widzimy dodanie obsługi błędów. Dowolny błąd rzucony przez jednego z obserwatorów wpłynie właśnie do `onError`. Zobaczyliśmy także zmianę wątku obserwacji. Wielowątkowość jest silnie powiązana z wzorcem reaktywnym. W ReactiveX wystarczy jedno polecenie, by każde działanie w strumieniu wykonywane było w osobnym wątku. Widać ją w Listingu 5, gdzie funkcja `observeOn` zmienia wątek wywołania odpowiednio na nowy wątek lub na wątek główny.

Pomyślmy o nieco bardziej skomplikowanym przykładzie. Załóżmy, że dostaliśmy zadanie, by pobrać listę użytkowników z API, wyciągnąć ich id, dla każdego z nich pobrać listę zakupów i wyświetlić ją na ekranie. Więcej – przełożony zażądał, by każda pobrana lista natychmiast została wyświetlana, nie czekając na resztę. Na koniec wyświetlimy informację o tym, że wszystkie listy są już wyświetlone. Brzmi jak koszmar? Nie w ReactiveX. Tutaj obsługa tej koncepcji będzie wyglądała jak w Listingu 6.

Listing 6. Przykład strumieniowego przetwarza dla bardziej skomplikowanego przypadku

```
getUsersObservable() // 1
  .map { u: User -> u.id } // 2
  .observeOn(Schedulers.newThread()) // 3
  .flatMap { id -> getShoppingListObservable(id) } // 4
  .observeOn(schedulers.io()) // 5
  .subscribe(
    { elem -> addListToView(elem) }, // onNext, 6
    { e -> logError(e) }, // onError, 7
    { informAboutListsFilled() } // onComplete, 8
  )
```

1. Tworzymy `Observable`, który wysyła zapytanie do API o listę użytkowników.
2. Mapujemy otrzymanego użytkownika na jego id.
3. Zmieniamy wątek obserwacji na nowy.
4. Dla każdego otrzymanego zdarzenia tworzymy `Observable`, które pobierze listę zakupów. Zdarzenia zawierają id użytkownika, ponieważ zostały zmapowane w 2.
5. Zmieniamy wątek obserwacji na główny, by móc zmieniać elementy widoku (wymóg Androida).
6. Ustawiamy funkcję lambda, które zostanie wywołana po otrzymaniu poprawnego zdarzenia zawierającego element listy zakupów.
7. Ustawiamy funkcję lambda, która zostanie wywołana po otrzymaniu błędu. Taka sytuacja ma miejsce, gdy dowolny z pośrednich `Observable` zwróci błąd
8. Ustawiamy funkcję lambda, która zostanie wywołana po zamknięciu strumienia. Stanie się tak, gdy otrzymamy ostatnią odpowiedź zarówno na zapytanie o użytkowników, jak i o listy zakupów.

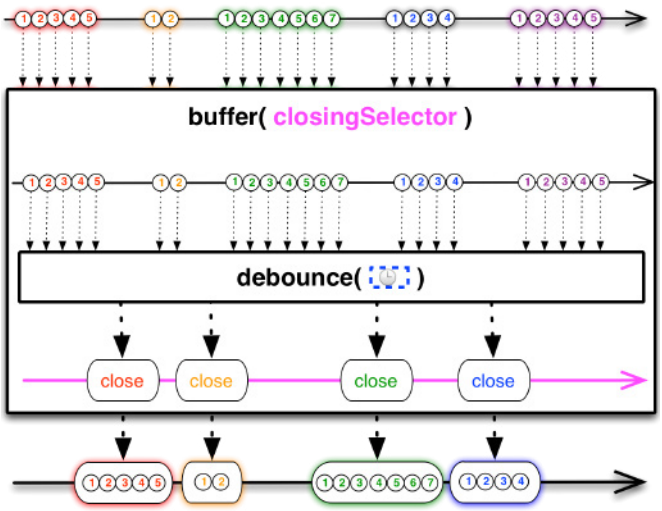
W przykładzie wciąż jednak używamy wyłącznie dwóch metod dostarczanych przez RxJava: `flatMap` i `map`. Jeśli zajrzymy do dokumentacji, to zobaczymy, że tak naprawdę są dziesiątki funkcji, które pozwalają na strumieniowe przetwarzanie obiektów `Observable`. Istnieje szereg metod, które pozwalają na ich łączenie, filtrowanie, agregację i wiele innych operacji. Zobaczmy inny przykład – chcemy zrobić obsługę dwukliku danego elementu. To nie jest

takie proste, ponieważ jeśli użytkownik naciśnie przycisk bardzo szybko 6 razy, to chcielibyśmy interpretować to jako 1 wieloklik, a nie kilka. Implementacja będzie wyglądała następująco:

Listing 7. Definicja strumienia, który będzie wysyłał zdarzenia po wielokliku (przynajmniej dwukliku)

```
val doubleClickStream = clicksStream
  .buffer(clicksStream.debounce (1, SECOND))
  .filter { it.size > 2 }
```

Jest to bardziej skomplikowane przetwarzanie zdarzeń, ponieważ łączy dwie funkcje: `buffer` i `debounce`. `buffer` grupuje zdarzenia w listę. Jest ono sterowane zdarzeniami wysyłanymi przez element obserwowalny z argumentu (tutaj `debounce`). `debounce` emituje sygnał po określonym czasie od ostatniego zdarzenia, po którym nie nastąpiło inne. W tym przypadku po 1 sekundzie. Tutaj zdarzenia emitowane przez `debounce` oznaczają ucięcie nasłuchiwania dla metody `buffer`. Zostało to przedstawione na Rysunku 4.



Rysunek 4. Jak działa połączenie buffer i debounce służące do wychwycenia wieloklików (źródło: www.github.com/ReactiveX/RxJava/wiki/Backpressure)

Wygląda to bardzo dobrze, ale co tak naprawdę kryje się pod tymi wszystkimi funkcjami, takimi jak `onClickObservable` czy `getUsersObservable`?

TWORZENIE OBIEKTÓW TYPU OBSERVABLE

Większość tworzonych `Observable` jest w pewnym stopniu odpowiednikiem funkcji z callbackiem. Przykładowa taka funkcja przedstawiona jest w Listingu 8. Tworzenie analogicznego do niej `Observable` zostało natomiast przedstawione w Listingu 9. Jak widać – funkcje te niewiele się różnią. To jest jednak prosty przypadek, gdzie generowane jest tylko jedno zdarzenie `onNext`.

Listing 8. Przykład funkcji wykorzystującej callback

```
fun makeLongOperation(callback: (Data)->Unit) {
  thread {
    val d: Data = longOperation()
    callback(d)
  }
}
```

Listing 9. Przykład funkcji tworzącej Observable, analogiczny do funkcji z callbackiem z Listingu 5

```
fun makeLongOperationObservable() = Observable.create<Data> {
subscriber ->
  val d = longOperation()
  subscriber.onNext(d)
}
```

`Observable` mogą generować wiele zdarzeń `onNext`. Zostało to przedstawione w Listingu 10. Tam generowane jest zdarzenie dla każdej z kolejnych liczb. Bardziej praktyczny przypadek tworzenia obiektu obserwowalnego możemy zobaczyć w Listingu 11, gdzie tworzymy obiekt generujący zdarzenie `onNext` za każdym razem, gdy naciśnięty zostanie przycisk podany w argumencie. Poza zdarzeniem `onNext` wygenerowane mogą zostać jeszcze:

- `onComplete` – zdarzenie generowane, gdy zakończeniu ulega obserwacja.
- `onError` – zdarzenie generowane, gdy wystąpi jakiś błąd.

Obydwa te zdarzenia standardowo zamykają strumień. Przykład `Observable`, który reaguje (wysyła `onNext`) na naciśnięcie przycisku, ale może zostać zamknięty przy pomocy innego przycisku, pokazany został w Listingu 12.

Listing 10. Tworzenie Observable, który emituje wiele zdarzeń i kończy działanie

```
fun counter(countTo: Int, millisBetween: Int) = Observable.
create<Int> { subscriber ->
  for (i in 0 until countTo) {
    subscriber.onNext(i)
    Thread.sleep(millisBetween)
  }
}
```

Listing 11. Tworzenie Observable, który wysyła zdarzenia za każdym razem, gdy naciśnięty zostanie przycisk

```
fun onClicked(view: View) = Observable.create<View> {
subscriber ->
  view.setOnClickListener { v -> subscriber.onNext(v) }
}
```

Listing 12. Tworzenie Observable, który wysyła zdarzenia za każdym razem, gdy naciśnięty zostanie przycisk, ale może zostać zamknięty poprzez naciśnięcie drugiego przycisku

```
fun onClickClosable(eventButton: View, closeObservableButton:
View) = Observable.create<View> { subscriber ->
  eventButton.setOnClickListener { v -> subscriber.onNext(v) }
  closeObservableButton.setOnClickListener { v -> subscriber.
  onComplete() }
}
```

Przedstawiona tutaj funkcja `create` to tylko jeden ze sposobów tworzenia `Observable`, ponieważ ReactiveX dostarcza szereg innych metod, dzięki którym możemy je tworzyć z list, wartości, przedziałów czasowych, funkcji i wielu innych. Zobaczyliśmy już w Listingu 3 zastosowanie funkcji `from`, która z przedziału liczb (`IntRange`) wygenerowała zdarzenia zawierające kolejne liczby.

W Androidzie rzadko kiedy definiuje się samodzielnie `Observable`, ponieważ ReactiveX ma dodatkowe wsparcie dla Androida (RxBinding), które dostarcza szereg funkcji tworzących różnego rodzaju `Observable` dla elementów widoku. Podobne wsparcie ReactiveX ma dla Netty i Cocoa. W Androidzie jest znacznie więcej bibliotek, które silnie wspierają RxJava, takich jak Retrofit (najpo-

pularniejsza aktualnie biblioteka do operacji sieciowych w Androidzie) czy Nucleus (popularna biblioteka wspomagająca MVP). Dlatego bardzo rzadko kiedy sami musimy tworzyć `Observable`.

PROPAGACJA ZMIAN

Wspomniana na wstępie propagacja zmian też jest obecna w ReactiveX, ale można ją zrealizować na wiele różnych sposobów. Zazwyczaj uwzględniają one obiekt typu `Subject` (temat). Jest to obiekt, który jest jednocześnie `Observable` oraz pozwala na przyjmowanie zdarzeń (poprzez `onNext`). Przykład jego działania przedstawiony został w Listingu 13.

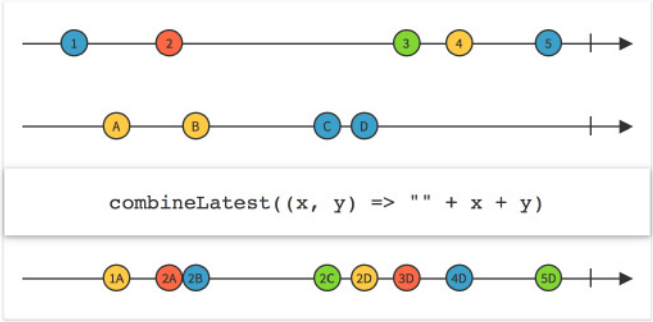
Listing 13. Przykład działania PublishSubject. Do obiektu tego typu w dowolnym momencie można podpiąć kolejnego nasłuchującego oraz można przy jego użyciu wyemitować zdarzenie, które dotrze do wszystkich nasłuchujących

```
val s: Subject<String, String> = BehaviorSubject()
s.onNext("Przyszedł A")
s.subscribe { println("Jestem A, słyszę: $it") }
s.onNext("Przyszedł B")
// Jestem A, słyszę: Przyszedł B
s.subscribe { println("Jestem B, słyszę: $it") }
s.onNext("Przyszedł C")
// Jestem A, słyszę: Przyszedł C
// Jestem B, słyszę: Przyszedł C
s.subscribe { println("Jestem C, słyszę: $it") }
s.onNext("Przyszedł D")
// Jestem A, słyszę: Przyszedł D
// Jestem B, słyszę: Przyszedł D
// Jestem C, słyszę: Przyszedł D
```

Najprostszym sposobem na implementację propagacji zmian jest właśnie zaimplementowanie obiektów jako `Subject` i uzależnienie ich stanu od zdarzeń pochodzących z obiektów, które o tym stanie powinny decydować. Obiekty typu `Subject` potrafią także przechowywać stan. Przykład, naśladujący komórki znane z programu Microsoft Excel, przedstawiony został w Listingu 14. Wykorzystana została tam funkcja `combineLatest`, która emituje zdarzenie w odpowiedzi na zdarzenie z dowolnego `Observable` przekazanego przez argument. Marble Diagram funkcji `combineLatest` przedstawiony został na Rysunku 5. Implementacja, przedstawiona w przykładzie z Listingu 14, jest typowym przypadkiem propagacji zmian.

Listing 14. Przykład działania BehaviorSubject. Działa on podobnie jak PublishSubject, ale przechowuje wartość i umożliwia do niej dostęp w dowolnym momencie. Tutaj przykład naśladuje komórki znane z programu Microsoft Excel

```
val a = BehaviorSubject.create(0)
val b = BehaviorSubject.create(0)
val c = BehaviorSubject.create(0)
val d = BehaviorSubject.create(0)

a.subscribe { i -> println("a: $i") } // a: 0
b.subscribe { i -> println("b: $i") } // b: 0
c.subscribe { i -> println("c: $i") } // c: 0
d.subscribe { i -> println("d: $i") } // d: 0

// b = a * 2
a.subscribe { i -> b.onNext(i * 2) }
// b: 0

// c = a * b + d
Observable.combineLatest(a, b, d, { a, b, d -> Triple(a, b, d)
})
  .subscribe { (a, b, d) -> c.onNext(a * b + d) }
// c: 0

// a <= 3
a.onNext(3)
// a: 3 b: 6 c: 0 c: 18

// d <= 4
d.onNext(4)
// d: 4 c: 22
```



Rysunek 5. Marble Diagram funkcji combineLatest (źródło: http://reactivex.io/documentation/operators/combinelatest.html)

PODSUMOWANIE

Programowanie reaktywne nie tylko wygląda obiecująco, ale także sprawdza się w praktyce. ReactiveX pozwala na tworzenie strumieni zamiast struktur callbacków. W ich obrębie udostępnia szeroką gamę funkcji do manipulowania tymi strumieniami. Dzięki wsparciu od strony ReactiveX łatwo jest tworzyć `Observable` oraz można je uzyskiwać poprzez łączenie kilku prostszych. Jest także wiele bibliotek, które tworzą `Observable`. Jest to w dużym stopniu zasługa tego, że funkcje dostarczające `Observable` są oddzielone od reszty kodu i łatwe do ponownego użycia.

W Androidzie RxJava jest właściwie standardem wymienianym często jako dobra praktyka. Zwłaszcza popularne jest użycie tej biblioteki do zapytań sieciowych w połączeniu z biblioteką Retrofit. Jest ona wykorzystywana przez takich gigantów jak Netflix (który chwali się jej zastosowaniem do błyskawicznego ładowania list na stronie), GitHub, Trello, Treehouse i wiele innych.

Pozwala na rozwiązanie wielu problemów oraz wprowadzenie konceptu programowania reaktywnego do języków, które nie posiadają dla niego wbudowanego wsparcia. Sprawdza się nie tylko dla aplikacji komunikujących się z użytkownikiem, ale także dla aplikacji serwerowych. Pozwala, na przykład, na proste wprowadzenie wielowątkowości do projektu, dzięki czemu lepiej są wykorzystywane zasoby urządzenia. Osobiście polecam ją na podstawie ponad roku doświadczeń z intensywnego wykorzystywania jej w projektach.

Dziękuję Kamilowi Karwowskiemu za cenne komentarze przy pisaniu tego artykułu.

MARCIN MOSKAŁA
marcin.moskala@docplanner.com
Programista Kotlin i Groovy na platformę Android pracujący dla ZnanegoLekarza, autor bloga KotlinDev.pl oraz współzałożyciel Nootro (www.nootro.pl). Pasjonat samorozwoju oraz tworzenia pięknego i prostego kodu.