[Tutaj wpisz tytuł wpisu]

Chciałem wam dzisiaj trochę poopowiadać o Jave 8.

Na początku chciałbym przeprosić wszystkich purytów językowych, gdyż będe mieszał trochę polski i angielski.

Wersja 8 Javy została opublikowana 18 marca tego roku i przyniosła ona sporo nowości, szczególnie jeżeli porównany ją do tego, co nowego zaoferowała nam Java 7.

Mamy tutaj skróconą listę najciekawszych właściwości które pojawiły się w nowej wersji Javy:

- lambda expressions – czyli anonimowe funkcje

- method references – czyli inny sposób na zapisanie wyrażen lambda

- Stream API – do operacji na kolekcjach

- default methods – czyli metody wraz z implementacją w interfejsie

- klasa Optional czyli alternatywa dla nulla i sposób ochrony przed NullPointerExceptionem

- oraz nowe API do operowania na dacie i czasie

O pierwszych 4 tematach będę chciał szerzej powiedzieć podczas tej prezentacji

Zacznijmy od terminu Behaviour parametrization. Załóżmy że mamy metodę, i chcielibyśmy przekazać do niej fragment kodu jako jej argument i ewentualnie wywołać go później.

Zobaczmy to na przykładzie.

Załóżmy że mamy napisać fragment kodu odpowiedzialny za selekcje jabłek ze względu na kolor. Piszemy metodę filterApplesByColour. Następnie chcemy napisać metode która filtruje jabłka ze względu na wage.

Możemy oczywiście napisać oddzielną metodę filterApplesByWeight, ale bardzo szybko zauważymy że mamy spora duplikacje kodu, te metody różnią się tylko jedną linijką. Wpadamy na pomysł, że jako parametr przekazemy fragment kodu który będzie odpowiedzialny za filtrowanie – czyli to jest właśnie behaviour parametrization.

Zobaczmy jak wyglądało by to w Javie 7.

Zdefiniujemy sobie interfejs ApplePredicate, z jedną metodą test zwracającą wartość typu boolean. Od razu możemy zdefiniować sobie klasy implementujące ten interfejs.

Nasza nowa metoda filterApples jako 2 parametr przyjmuje obiekt typu ApplePredicate i wywołuje metodę test.

A tutaj widzimy wywołania tej funkcji: wykorzystująć klasę anonimową, oraz 2 wcześniej zdefiniowane klasy.

Musieliśmy wykonać stosunkowo dużo żmudnej pracy żeby osiągnąć nasz cel. Spójrzmy na wywołanie tej metody z klasą anonimową. Tak naprawdę istotna jest tylko jedna linika, wszystko poza nią jedynie zaciemnia kod.

Java 8 pozwala nam na dużo czytelniejszy i zwięźlejszy zapis przy użyciu wyrażeń lambda, a przedstawiony zapis mógłby zostać jeszcze skrócony, gdyż typ Apple jest zbędny.

Zobaczmy jeszcze jeden przykład, gdzie klasy anonimowe są często wykorzystywane: implementacja komparatora. Moim zdaniem, zapis z lambdami jest czytelniejszy, pozbyliśmy się sporo nikomu nie potrzebnego kodu.

Widzieliśmy już przykład wyrażeń lambda, teraz trochę teorii.

Co to takiego ta lambda – można powiedzieć że jest to anonimowa funkcja, która może być przypisywana do zmiennych, przekazywana jako argument metody, zwracana z funkcji, która to funkcja nie ma nazwy, ale ma ciało funkcji, parametry, zwracany typ, i listę wyjątkow które może rzucać.

Lambda nie pozwala na nic, co nie było możliwe wcześniej, ale dzięki nim nie musimy już dłużej pisać niezgrabnego kodu używająć klas anonimowych aby skorzystać z behaviour parametrization w naszym kodzie.

Składnia lambdy:

Na początku mamy parametry lambdy. Jeżeli jest tylko 1 parametr możemy pominąć nawiasy. Co więcej możemy także pominąć typy ponieważ kompilator może się ich domyślić

Następnie mamy strzałkę.

A za strzałką ciało funkcji.

Ciało funkcji może być zapisane w klamrach lub bez. Jeżeli ciało naszej lambdy zawiera jedno wyrażenie to nawiasy klamrowe i instrukcję return możemy pominąć, w przeciwnym przypadku musimy użyć klamr.

Lambdy możemy używać wszędzie tam gdzie typem jest funtional interface.

Functional interface jest to interfejs który specyfikuje dokładnie jedną abstrakcyjną metode.

Lambda pozwala na implementacje abstrakcyjnej metody interfejsu inline i jest ona traktowana jako instancja interfejsu funkcyjnego.

Stworzona została adnotacja @FunctionalInterface, za jej pomocą możemy oznaczyć że naszą intencją było stworzenie interfejsu funkcyjngo.

Jeżeli oznaczymy interfejs tą adnotacją, a nie będzie on spełniał definicji interfejsu funkcyjnego, wtedy kompilator zasygnalizuje nam błąd. Może się to zdarzyć jeżeli np. interfejs będzie miał 2 abstrakcyjne metody.

Adnotacja ta nie jest obowiązkowa, czyli mogą istnieć interfejsy funkcyjne bez niej, służy ona celom dokumentacyjnym i szybkiemu sygnalizowaniu błędów kompilacji.

Jak pamiętacie w naszym przykładzie stworzyliśmy interfejs ApplePredicate. Dobrą wiadomością jest to, że tak naprawdę nie potrzebujemy go, gdyż podobny interfejs Predicate już istnieje w pakiecie java.util.function. Naliczyłem w tym pakiecie 46 funkcyjnych interfejsów, warto część z nich znać gdyż są one dosyć intensywnie wykorzystywane np. w Streams API.

Zobaczmy kilka najczęściej spotykanych.

Wcześniej już wspomniany Predicate, który ma metodę test zwracającą wartość typu boolean dla danego obiektu – możemy go wykorzystać właśnie to operacji typu filtrowanie.

Następny interfejs to Consumer, którego metoda accept przyjmuje obiekt i nic nie zwraca. Możemy ją wykorzystać np. do wypisania czegoś na konsoli.

Interfejs Function parametryzowany 2 typami T i R, metoda apply przekształca obiekt typu T w R.

Wyobraźmy sobie że mamy listę obiektów typu Apple, i chcemy zmapować ją na listę kolorów, w tym celu użyjemy tego interfejsu.

Istnieją także wersje tych interfejsów dla typów prymitywnych. Powodem ich stworzenia była wydajność, dzięki nim możemy uniknąć niepotrzebnego pakowania i rozpakowywania typów prymitywnych.

Przykładowo dla inta mamy IntPredicate, IntConsumer, IntFunction, IntToLongFunction, IntToDoubleFunction, ToIntFunction.

Teraz przejdźmy do Method references. Wrócmy do naszego przykładu z selekcjonowaniem jabłek. Załózmy że klasa apple ma metodę isOrganic mówiącą czy dane jabłko było wytwarzane ekologicznie.

Mamy tutaj przykład filtrowania ekologicznych jabłek przy pomocy lambdy i method reference

Method reference możemy używać tam, gdzie lambda składała się tylko z wywołania jednej metody. Jest to syntatic sugar, pozwala nam na napisanie jeszcze zwięźlejszego kodu.

Czasami zapis wykorzystujący method references może być bardziej czytelny niż lambda

Zobaczmy jeszcze jeden przykład: sortowanie jabłek ze względu na ich wagę.

W przypadu method reference wykorzystujemy jeszcze metodę comparing z interfejsu Comparator, ale zwróccie uwagę jak czytając od razu wiemy co ten kod robi: sortuj porównując wagę jabłek – nie musimy się ani sekundy zastanawiać za co ten kod jest odpowiedzialny.

Method references możemy używać w 3 przypadkach:

1. Kiedy odnosimy się do statycznej metody, zamiast pisać (a) -> integer.parseInt(a) możemy napisać Integer::parseInt
2. Jeżeli chcemy wywołać zwykła metodę danej klasy, zamiast pisać a-> a.length() możemy napisać String::length. Gdyby było więcej parametrów, to byłyby one przekazane jako parametry tej metody
3. Trzeci przykład pokazuje wywołanie metody na konkretnym obiekcie. Mamy obiekt abc typu ThreadLocal, zamiast pisać x-> abc.set(x), możemy napisać abc::set

Ta sama notacji możemy użyć jeżeli chcemy wywołać konstruktor

Mamy tutaj przykład konstruktora 0 i 1 argumentowego. Zakładamy że klasa apple ma takie konstruktory, a w przypadku 1 arg, jedynym argumentem jest int

**Kolejny temat to Streams API**

Collections API jest jednym z najczęściej używanych API, ale nie jest ono pozbawione wad.

Wiele operacji businessowych wymaga operacji podobnych do tych jakie spotykamy w bazach danych, grupowanie, wyszukiwanie jakiegoś elementu w kolekcji, projekcja.

W Javie sprowadza się to do żmudnego pisania każdego kroku algorytmu, natomiast bazy danych pozwalają nam zdefiniować takie operacje deklaratywnie, mówimy co chcemy otrzymać, a nie jak chcemy żeby to było osiągnięte, w ogóle się nad tym nie musimy zastanawiać.

Drugą kwestią są operacje na dużych zbiorach. Jeżeli chcemy poprawić wydajność, czasami musimy napisać kod wielowątkowy. Ale pisanie takiego kodu jest znaczne trudniejsze w porównianiu do kodu jednowątkowego, nie mówiąc już o jego debugowaniu.

I między innymi z tych dwóch powodów pojawiło się Streams API.

Po pierwsze to API pozwala nam na operowaniu na kolekcji w sposób deklaratywny.

Po drugie, istnieje możliwość wykonywania operacji na streamie współbieżnie, bez konieczności pisania dodatkowego kodu zajmującego się tworzeniem wątków, lokowaniem itd.

Na początku spójrzmy na przykład, który mam nadzieje zainteresuje was Stream API i pokaże jego możliwości. XXX

Załóżmy że naszym zadaniem jest zwrócenie nazw dań które są niskoklaoryczne, i posortowane ze względu na kalorie.

Widzimy na początku zmienną pomocniczą lowCaloricDishes, następnie filtrowanie dan z menu które są niskokaloryczne, po czym sortujemy tę kolekcje.

W ostatniej już pętli z każdego obiektu typu Dish wyciągamy nazwę dania. Sporo kodu jak na tak stosunkowo proste zadanie.

Zobaczmy jakby to wygląłdało w Javie 8.

Na liście dishes wywołujemy metodę stream() która zwraca nam obiekt Stream, następnie filtrujemy potrawy który mają mniej niż 400 kcal, sortujemy porównując kalorie, z każdego dania wyciągamy tylko nazwę i wynik zapisujemy do listy

Jeżeli chcemy wykonać to współbieżnie, to zmienia się tylko metoda tworzenia Streamu, zamiast wywoływać metodę stream() wywołujemy parallelStream()

Nasze zadania zapisaliśmy teraz w sposób deklaratywny, przez co jest ono znacznie bardziej zwięzłe i czytelne.

Rozwiązanie komponujemy poprzez dodawanie kolejnych kroków w przetwarzaniu strumienia.

No i wydajność: bardzo łatwo jest wykonać to wielowątkowo, kod wygląda niemal identycznie, czyli niskim kosztem możemu uzyskać sporą poprawę wydajności.

Przejdźmy na chwile do teorii, co to jest ten Stream?

Stream jest to sekwencja elementów pochodząca z jakiegoś źródla która pozwala na wykonywanie operacji agregujących

Rozbijmy tę definicje na kawałki:

- sekwencja elementów: mamy tu podobieństwo do kolekcji, ale kolekcje są skupione na przechowywaniu danych, natomiast Stream pozwala nam wykonywać operacje na tej sekwencji danych

- Żrodło: stream konsumuje dane dostarczone np. z kolekcji, tablicy czy też zasobów typu plik

- operacje agregujące: stream dostarcza operacji znanych wszystkim z SQLa i częstych w programowaniu funkcyjnym: filter, map, reduce, find, match, sorted.

Kolejnym elementem który odróżnia stream od kolekcji jest sposób iteracji.

W kolekcjach wykorzystujemy zewnętrzną iterację: tu sam użytkownik jest odpowiedzialny za przemierzenie całej kolekcji – dlatego też musimy pisać pętle np. for

W Streamach jest wykorzystywana wewnętrzna iteracja, to stream wykonuje iteracje za nas, i jest odpowiedzialny za przechowywanie wynikowego streamu.

Korzysć z tego jest taka, że biblioteka może zoptymalizować w jaki sposób odbywa się przechodzenie po streamie, i dostosować do sprzętu na jakim jest to wykonywane.

Operacje na streamie można podzielić na 2 grupy: operacje pośrednie i operacjie końcowe.

Na przedstawionym przykładzie, operacjami pośrednimi są filter (filtrowanie), map (z danego dania wyciągamy tylko jego nazwę), i limit (ograniczamy liczbę wyników do maksimum 3.

Operacje pośrednie charakteryzują się tym że mogą być ze sobą łączone, gdyż zwracają one Stream. Dodatkowo żadne operacje nie są wykonywane do momentu wywołania operacji końcowej – czyli są one leniwe, a dzięki temu mogą być zaimplementowane optymalnie.

Drugą grupą są operacje końcowe, na tym przykładzie jest to metoda collect powodująca że tworzona jest końcowa lista. Dopiero wykonanie tej operacji powoduje wykonanie wszystkich pośrednich operacji, a po nim strumień jest zamykany. Dla danego strumienia może być wykonana tylko jedna operacja koncowa.

A teraz chciałbym wam przedstawić jedną z najciekawszych właściwości stramów: laziness i short-circuiting – czyli coś co ma duży wpływ na wydajność, gdyż pozwala nam na uniknięcie niepotrzebnych obliczeń.

Mamy listę dań, teraz macie 3 sekundy żeby je zapamiętać ☺

Mamy następujące operacje wykonane na streamie: filtrujemy te dania, które mają więcej niż 300 kalorii, interesuje nas tylko nazwa tych dań, dlatego wykonujemy mapowanie, i chcemy tylko pierwsze 3 elementy, do tego wykorzystujemy metodę limit. Jak widzicie mamy tutaj jeszcze wypisywanie na konsoli podczas operacji filter i map, żeby zrozumieć co kiedy się wykonuje.

I teraz na wyjściu tego programu pojawi się coś takiego:  
Widzimy, że zostało to wykonane optymalnie, żadne zbędne operacje nie miały miejsca właśnie, dzięki leniwej naturze streamów. Pomimo tego że mamy wiele dan które mają więcej niż 300 kalorii, to operacje filtrowania i mapowania została wykonana tylko dla pierwszych trzech. Jest to spowodowanie przez metodę limit i technikę short-circuiting.

Short circuiting zapewne znacie z sposobu obliczania wartości logicznej gdy używamy operatora AND , jeżeli pierwszy argument zwraca false, to nie musimy już ewaluować dalszej części wyrażenia. I tutaj dzieje się podobnie.

Po drugie, pomimo tego że operacje filter i map są to 2 odmienne operacje, zostały one zmergowane i są wykonywane wspólnie podczas jednego przebiegu.

Na następnym slajdzie mamy przedstawioną niepełną listę metod dostpenych w klasie stream.

Nie bede szczegółowo omawiał wszystkich tych metod, jedynie wymienie:  
filter, distinct, skip, limit, map, sorted, anyMatch, noneMatch, allMatch, findAny, findFirst, forEach, collect, reduce, count.

Oprócz ogólnej klasy stream Java 8 wprowadziła także 3 specjalizowane klasy streamów dla typów prymitywnych: IntStream, DoubleStream i LongStream.

Załóżmy że chcielibyśmy zsumować kalorie wszystkich dań w menu. Dla zwykłego streamu użylibyśmy metody reduce, co może nie być do końca czytelene. Zamiast tego możemy użyć metody mapToInt która zwróci IntStream i wywołać po prostu metodę sum.

Inne specjalizowane metody to min, max, average.

Specjalizowane streamy mają dodatkowo metody range i rangeClosed, które pozwlają nam na tworzenie streamu zawierającego jakiś zakres liczb. Przykładowo ten kod wygeneruje nam stream zawierający wszystkie parzyste liczby w zakresie od 1 do 100.

Stream można stworzyć na wiele sposobów:  
z kolekcji: co już widzieliśmy – wywołujemy metodę stream()

Ze zbioru wartosci podanych explicite: Stream.of

Z tablicy wykorzystujemy metodę klasy Arrays.stream

Z pliku:

Java NIO API dosyć intensywnie wykorzystuje Stream API, np. Files.lines zwraca strumień wierszy z pliku

I w końcu streamy możemy generować z funkcji, do czego służą 2 metody: Stream.iterate i Stream.generate. Obie te metody tworzą strumienie nieskończone, które nie mają ograniczonej wielkości, dlatego rozsądnie jest użyć metody limit! Metody iterate używamy gdy chcemy wygenerować sekwencje liczb, i kolejna wartośc zależy od poprzedniej. W metodzie generate tworzone wartości są nie zależne, to znaczy nie zależą od poprzednio wygenerowanych wartości.

Kolejna nowośc w Javie 8 to defaultowe metody w interfejsie.

Od tej wersji jest możliwe deklarowanie metod w interfejsie wraz z implementacją

- mogą to być metody statyczne

- lub defaultowe metody – metody niestatyczne

Mamy tu przykład takiego interfejsu z zdefiniowaną metodą statczną i metodą defaultową.

Jak widzimy metoda defaultowa poprzedzona jest słowem kluczowym default

Może rodzić się pytanie: czym w takim razie różni się inerfejs od klasy abstrakcyjnej. Przecież teraz i klasa abstrakcyjna i interfejs może posiadać metody abstrakcyjne i metody z implementacją.

Należy jednak pamiętać że klasa może rozszerzać tylko jedną klasę abstrakcyjną, i implementować wiele interfejsów.

Z drugiej strony klasa abstrakcyjna może mieć pola instancyjne, natomiast w interfejsie nie jest to możliwe.

Kiedy mogą przydać się metody domyślne?

Są one użyteczne gdy chcemy rozwijać bibliotekę, a przy tym zachować wsteczną kompatybilność. Przykładem może być metoda sort dodana do interfejsu List. To może być przydatne również w naszym przypadku hybris, czasami chciałoby się dodać jakąś metodę do interfejsu, ale nie możemy, przez to że musimy być wstecznie kompatybilni. W Javie 8 łatwo moglibyśmy dodawać nowe metody i jednocześnie zachowywać tą kompatybilność.

2. Dosyć często metody domyślne występują w interfejsach funkcyjnych – np. Predicate.and() – pamiętamy że interfejsy funkcyjne mogą mieć tylko 1 metodę abstrakcyjną

3. Opcjonalne metody. Przykładem może być metoda remove w interfejsie Iterator. Wcześniej często się zdarzało że developerzy nie implementowali tej metody, pozostawiali ją pustą, co skutkowało zbędnym kodem, który mógł tylko zaciemniać całośc. Dzięki temu że w interfejsie jest ona zaznaczona jako defaultowa, nie musimy jej implementować kiedy nie chcemy.

4. Wreszcie w javie mamy możliwość dziedziczenie zachowania z wielu miejsc (klas, interfejsów).

Teoretycznie jest możliwość, że klasa odziedziczy tą samą metodę z wielu interfejsów/klasy. Dlatego też zdefiniowano 3 reguły, mówiące o tym która metoda jest ostatecznie brana pod uwagę.

1. Klasa ma zawsze pierwszeństwo. Metoda zdefiniowana w klasie lub superklasie ma wyższy priorytet niż jakakolwiek domyślna metoda pochodząca z interfejsu
2. Metoda z bardziej specifycznego interfejsu ma większy priorytet. Jeżeli B rozszerza A, B jest bardziej specyficzny niż A
3. Jeżeli występuje ciągle niejednoznaczność, klasa musi zaimplementować metodę domyślną, I ewentualnie wywołać metodę z danego interfejsu explicitly,

Tutaj mamy właśnie przedstawioną taką sytuacje, gdzie mamy interfejsy A i B oba mają domyślne metodę hello. W klasie C która implementuje oba te interfejsu musimy nadpisać tą metodę i wskazujemy ze chcielibyśmy użyć metody z interfejsu B.

Bibliografia:

2 książki które mogę polecić:

Java 8 in Action – z niej pochodzi większość dzisiejszych przykładów, w porównaniu do następnej książki jest bardziej szczegółowa.

Functional Programming in Java której autorem jest Venkant Subramaniam

Na stronie oracla możemy znaleźć sporo filmów, kompletną dokumentacje oczywiście.

Na javacodegeeks jest całkiem spory tutorial przedstawiający nowości z Javy 8

Jakieś pytania?

Dziękuje za uwagę!