Spis treści

https://blog.it-leaders.pl/rozmowa-kwalifikacyjna-javy-zaden-problem-cz-string/

1. Czym jest String w Javie?

String jest to klasa, która została zdefiniowana w pakiecie java.lang. Warto też zauważyć, iż nie jest to prymitywny typ danych (prymitywy zaczynają się z małej litery). Jest niezmienny oraz (zazwyczaj) przechowywany w puli ciągów znaków (String pool).

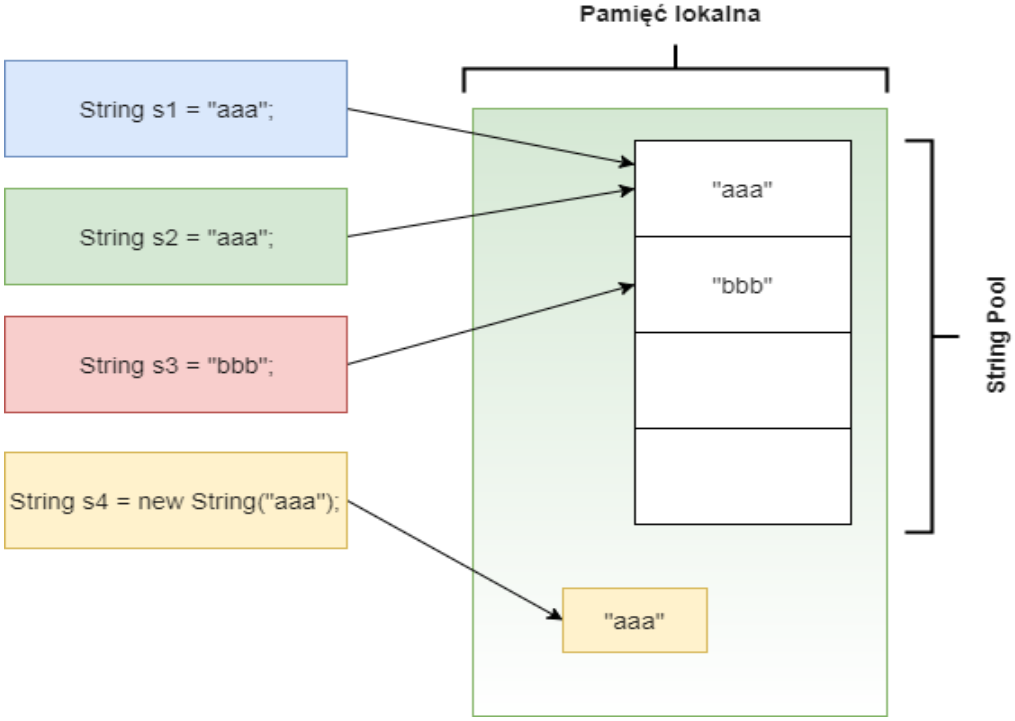
2. Dlaczego String jest niezmienny?

Po pierwsze, jest to spowodowane:

- Bezpieczeństwem – jeżeli w aplikacji występują połączenia (przedstawiane za pomocą Stringa), dane użytkowników (nazwy użytkowników, hasła), połączenia z bazami danych, to w przypadku gdy String mógłby być modyfikowany łatwość zmiany tych parametrów stanowiłaby zagrożenie bezpieczeństwa.
- Synchronizacja (w programowaniu wielowątkowym) – skoro String jest niezmienny to automatycznie rozwiązuje to problem z synchronizacją.
- Zarządzanie Pamięcią – kompilator widzi kiedy dwa obiekty typu String posiadają tą samą wartość. Dzięki temu możliwa jest optymalizacja – wystarczy jeden obiekt aby reprezentować dwie zmienne.

3. Co to jest String Pool?

String pool jest specjalnym miejscem w pamięci, w którym przechowywana jest pula wszystkich Stringów. Lądują tam wszystkie obiekty typu String, które zostały zainicjowane przy użyciu cudzysłowu – obiekty, które zostały zainicjowane przy pomocy słówka new lądują w pamięci lokalnej, poza pulą ciągów znaków. Warto pamiętać, że taka inicjalizacja ciągów znaków może spowodować mniejszą optymalizację kodu.



Rys, 1 Wizualizacja puli. Warto zauważyć, że s1==s2 daje true, s1 ==s4 zwraca false, natomiast s1.equals(s4) zwraca true.

4. Jak porównywać dwa obiekty typu Spring?

To pytanie bezpośrednio nawiązuje do pytania trzeciego. Otóż, jeżeli chcemy porównać zawartość obiektu (tj. ciąg znaków) powinniśmy używać metody, w którą zaopatrzony jest ten obiekt – s1.equals(). Jeżeli do porównywania używamy dwóch znaków równości to możemy spodziewać się niejednoznacznym wyników. Znak „==” sprawdza czy obiekty są identyczne, więc jeżeli inny obiekt będzie miał tą samą zawartość tekstową, ale przykładowo inny hashcode, to takie porównanie zwróci nam wynik „false”.
Funkcja .equals() porównuje to co jest w niej zapisane do porównywania – w przypadku Stringa jest to zawartość tekstowa.

StringExample.java

```
public class StringExample {

    public static void main(String[] args) {
        String s1 = "abc";
        String s2 = "abc";
        String s3 = new String("abc");

        System.out.println("s1 == s2: " + (s1==s2));
        System.out.println("s1 == s3: " + (s1==s3));
        System.out.println("s1.equals(s2): " + s1.equals(s2));
        System.out.println("s1.equals(s2): " + s1.equals(s3));

    }
```

Problems @ Javadoc Declaration Console

```
<terminated> StringExample [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (3
s1 == s2: true
s1 == s3: false
s1.equals(s2): true
s1.equals(s2): true
```

5. Jaka jest różnica pomiędzy klasą String, StringBuffer oraz StringBuilder?

Podczas manipulacji Stringiem w czasie wykonywania programu za każdym razem gdy go zmieniamy tworzymy obiekt. Aby więc nie marnować pamięci na ciągłe tworzenie nowych ciągów znaków java udostępnia nam dwie klasy, które pozwalają na manipulację ciągami znaków, bez zużywania zbędnej pamięci – czyli StringBuffer oraz StringBuilder. Różnice są zawarte w tabelce poniżej:

	String	StringBuffer	StringBuilder
Czy niezmienny?	Tak	Nie	Nie
Przechowywanie	String pool (ew. pamięć lokalna)	Pamięć lokalna	Pamięć Lokalna
Tworzenie	słówkiem new, oraz String s1 = "abc"	Słówkiem new	Słówkiem new
Bezpieczny wielowątkowo?	Tak	Tak	Nie
Wydajność	Niższa	Niższa	Wyższa

6. Jakie są metody inicjowania obiektów klasy String?

Istnieją dwie metody inicjowania obiektów klasy String:

```
public class StringExample {

    public static void main(String[] args) {

        String s1 = "abc";              //Metoda 1
        String s2 = new String("abc"); //Metoda 2

    }

}
```

Ważne jest, iż metody te nie są sobie równoważne (przynajmniej nie w każdym aspekcie). Metoda pierwsza powoduje, iż utworzony String trafia do puli obiektów klasy String (String pool), natomiast metoda druga lokuje obiekt w pamięci lokalnej. Dzięki metodzie pierwszej w przypadku powtórzenia danego ciągu znaków kompilator odwoła się do obiektów klasy String zawartych w puli – czego skutkiem będzie wyższa wydajność programu.

7. Czym różni się String od innych pochodnych klas?

String w odróżnieniu od prymitywów (primitive classes) jest pochodną klasą (derived class).

Po pierwsze – String posiada swoją specjalną pulę, w którym jest on przechowywane. Spowodowane jest to tym, iż jest on jedną z najczęściej wykorzystywanych klas (na dodatek bez możliwości modyfikacji) i dzięki temu możliwy jest wzrost wydajności programów. Ponadto z powodu powszechnego występowania, String posiada również możliwość deklarowania obiektów w sposób niedostępny dla innych klas – bez słówka kluczowego „new”, przy pomocy tzw. „String literals” (mała uwaga – wyjątkiem są klasy typu Wrapper np. Long, Integer czy Float). Jest to również preferowany sposób, dzięki temu nasze obiekty lądują w puli obiektów typu String. Ostatnią rzeczą, która jest wyjątkowa w tej klasie, jest operator „+” umożliwiający łatwe łączenie teog typu obiektów.

8. String, StringBuffer i StringBuilder – która z tych trzech klas typu final?

To pytanie jest pułapką – wszystkie z tych klas są zadeklarowane jako final. Warto pamiętać, że zadeklarowanie klasy jako final nie stanowi o jej zmienności. Słówko final stanowi o tym, że klasa nie może być dziedziczona przez inne klasy!

9. Czym jest „String Intern”?

To pytanie odnosi się do puli obiektów typu String (String Pool). „String Intern” jest to obiekt składowany w puli. Użycie metody .intern() na ciągu obiektów String spowoduje, że będziemy mogli być pewni, iż wszystkie obiekty o tej samej zawartości mają jedno miejsce w pamięci w puli.

10. Czy w obiekcie klasy String można przechowywać dane wrażliwe? Dlaczego?

W obiektach klasy String nie powinniśmy przechowywać danych wrażliwych np. haseł. Spowodowane jest to tym, iż String jest niezmienny (immutable) a więc nie ma sposobu aby pozbyć się zawartości danego obiektu. Dodatkowo przez to, że obiekty typu String są przechowywane w puli (String Pool) mają one trochę dłuższą żywotność niż obiekty w pamięci lokalnej (heap), co stanowi ryzyko, iż jakikolwiek użytkownik, który ma dostęp do

pamięci Javy może mieć dostęp do tych danych. Hasła i inne wrażliwe dane powinny być przechowywane w tablicy znaków – char[].

String s1 = „abc”;

String s2 = „abc”;

W tym przypadku tworzony jest jeden obiekt. Pierwszy obiekt s1 tworzony jest w pamięci lokalnej w puli obiektów typu String, natomiast drugi obiekt s2, poprzez referencję będzie odwoływał się do tej samej wartości.

String s1 = „abc”;

String s2 = new String(„abc”);

W drugim przypadku tworzone są dwa obiekty. Pierwszy z nich ląduje podobnie jak w poprzednim przykładzie w puli, natomiast drugi poprzez użycie słówka „new” tworzony jest w pamięci lokalnej.

https://blog.it-leaders.pl/rozmowa-kwalifikacyjna-javy-zaden-problem-cz-ii-kolekcje/

11. Czym jest Kolekcja (Collection)?

Kolekcją jest pojedynczy obiekt który pozwala na przechowywanie w nim wielu elementów. Kolekcje wiec są swojego rodzaju „kontenerem” do przechowywania innych obiektów w Javie. W odróżnieniu od tablicy kolekcje są bardziej zaawansowane oraz elastyczne – podczas gdy tablice oferują przechowywanie określonej ilości danych, kolekcje przechowują dane dynamiczne – pozwalają na dodawanie oraz usuwanie obiektów wedle Twojego życzenia.

12. Czym jest framework Kolekcji (Collections Framework)?

Framework kolekcji jest to zbiór struktur danych oraz algorytmów służących ułatwieniu organizacji danych dla programistów – tak aby sami nie musieli tworzyć odpowiadających im klas. W frameworku kolekcji zaimplementowane są najpopularniejsze struktury danych które są najczęściej wykorzystywane w programowaniu. Oferują one szereg metod służących przeprowadzaniu operacji na zbiorach (dodawanie, przeszukiwanie, usuwanie etc.) oraz doskonale prowadzoną dokumentację, dzięki czemu łatwo jest z nich korzystać. Najważniejsze Interfejsy definiowane przez framework to:

- Collection
- List
- Set
- SortedSet
- Map
- SortedMap

13. Jakie są zalety frameworka Kolekcji?

Przede wszystkim poprawiają szybkość działania oraz jakość pisanego przez nas oprogramowania – framework ten jest zbudowany aby dostarczyć nam wysoką wydajnośc i jakość. Kolejnym plusem jest zmniejszenie czasu potrzebnego na pisanie

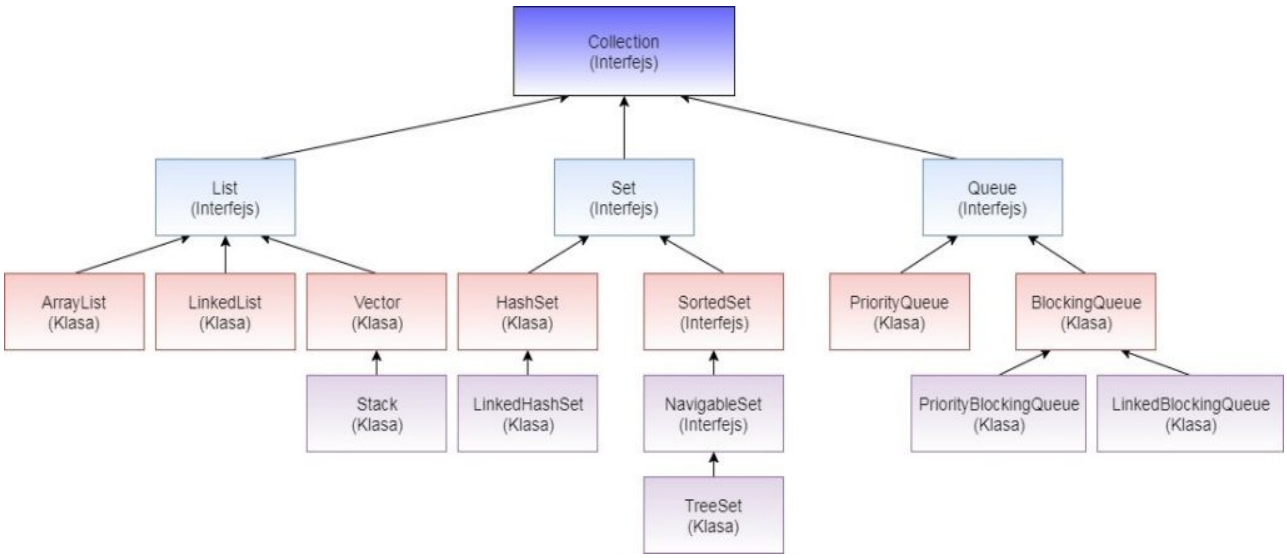
oprogramowania – nie musimy się skupiać na tworzeniu kontenerów samemu, możemy skupić się czysto na logice biznesowej. Ostatnim plusem jest to że framework jest wbudowany w JDK (Java Development Kit). Dzięki temu kod napisany z wykorzystaniem framework'u może być ponownie wykorzystany w innych aplikacjach oraz bibliotekach.

14. Jaka jest różnica pomiędzy Collection (Kolekcją) oraz Collections (Kolekcjami)?

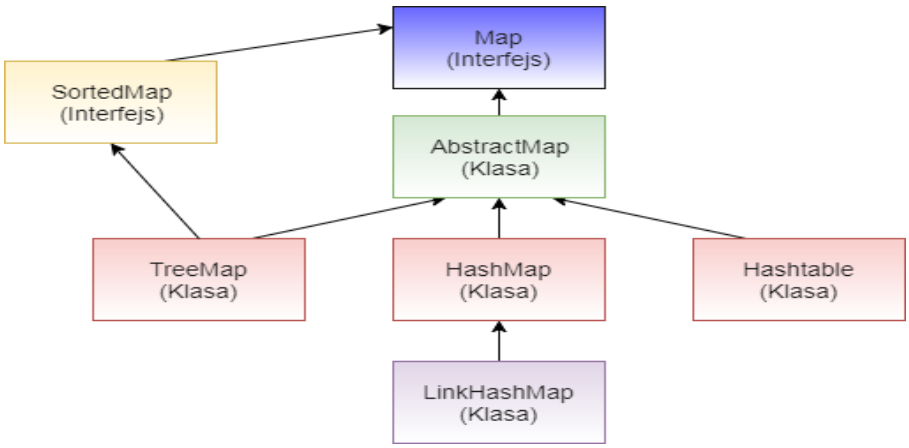
Collection jest to interfejs Frameworku Kolekcji Javy, natomiast Collections jest to klasa użyteczna która posiada statyczne metody potrzebne do dokonywania operacji na obiektach typu Collection. Przykładowo – Collections zawiera metodę dzięki której odnajdziemy konkretny element, lub posortujemy nasz zbiór. Czym jest interfejs Collection doskonale widać na rysunku w kolejnym pytaniu.

15. Jakie są najważniejsze główne interfejsy zawierające się we frameworku kolekcji?

Najważniejsze z nich to:
Collection
Set
Queue
Map



Rys. 1 Interfejsy frameworku kolekcji, które implementują interfejs Collection



Rys. 2 Implementacja interfejsu Map w kolekcjach

Po pierwsze zwróćmy uwagę na sposób formułowania pytania – nie mamy opowiedzieć o tym jakie rodzaje implementacji występują we frameworku kolekcji ale mamy opowiedzieć o poszczególnych implementacjach interfejsu Collection. Posiłkując się rysunkiem numer jeden widzimy że interfejsy które bezpośrednio implementują interfejs Collection to:

- List
- Set
- Queue

I tak:

Lista (List) jest to uporządkowany zbiór obiektów w którym obiekty mogą występować więcej niż jeden raz. Przykładowo lista może przykładowo zawierać następujące elementy {1,2,3,4,5,1,2,3,8,11} i poprzez odwołanie się do indeksu nr. 3 zwróci nam element 4 (Uwaga matlabowcy – listy rozpoczynają się od zerowego indeksu!). Lista jest podobna do tablicy, przy czym lista potrafi dynamicznie zmieniać swoją długość. Pozwala ona również na dostęp do każdego elementu poprzez podanie jego indeksu. Najczęściej używane implementacje interfejsu List to ArrayList oraz LinkedList. Lista posiada również metody które pozwalają dodać, usunąć oraz zamienić element na konkretnym miejscu.

Zbiór (Set) – implementacje interfejsu Set nie mogą posiadać duplikujących się elementów. I tak przykładowo wykorzystując zbiór z poprzedniego tłumaczenia Listy oraz dodając go do obiektu za pomocą metody addAll() (Dodającej wszystkie elementy po kolei) nasz zbiór będzie zawierał wszystkie te elementy które się nie powtarzają, czyli {1,2,3,4,5,8,11}. Jeżeli chcemy wypisać wartości znajdujące się w implementacji Set musimy wykorzystać Iterator bądź specjalną składnię pętli for (Enhanced For Loop).

Kolejka (Queue) – kolejka pozwala na przechowywanie elementów w kolejności do przetworzenia. Jest uporządkowana ale elementy możemy dodawać tylko na „końcu" kolejki, natomiast usuwać tylko na „początku kolejki. Kolejki zazwyczaj (choć nie zawsze) ustawiają elementy według FIFO (first in, first out – tzn. ten który pierwszy wchodzi, pierwszy wychodzi. Wyjątkiem może być tutaj klasa PriorityQueue które porządkują elementy w zależności od dostarczonego obiektu klasy Comparator, bądź naturalnego kolejkowania.

16. Dlaczego interfejs Map znajduje się we frameworku kolekcji, ale nie implementuje interfejsu Collection?

To pytanie jest podchwytliwe – obecność we frameworku kolekcji nijak ma się do samego interfesju Collection. Mapy nie implementują tego interfejsu ponieważ różnią się od pozostałych elementów. Implementacje interfejsu Collection zakładają elementy o jednej wartości – natomiast mapy przyjmują elementy w parach klucz/wartość. Z tego powodu niektóre funkcje zawarte w interfejsie Collection nie byłyby kompatybilne ze strukturą map.

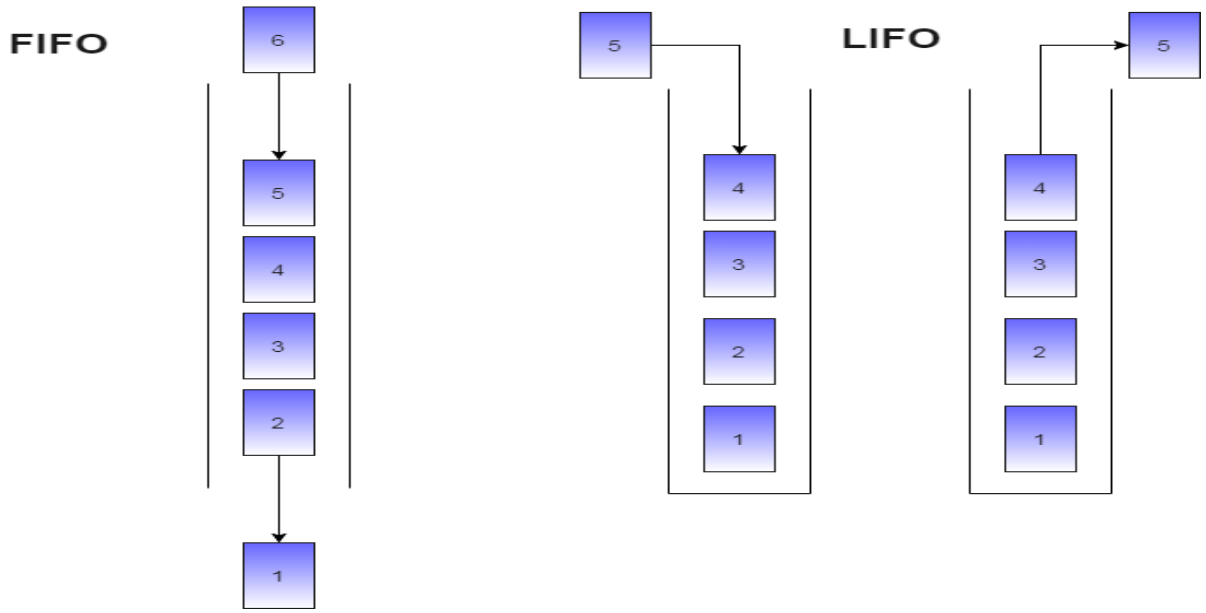
I krótka definicja mapy:
Mapy (Map) są to obiekty które mapują klucze do wartości. Mapa nie może posiadać dwóch takich samych kluczy – może natomiast posiadać dwie takie same wartości (pod warunkiem że mają różne klucze). Podstawowe implementacje interfejsu Map to HashMap, TreeMap oraz LinkedHashMap.

17. Czym jest Iterator?

Iterator jest to interfejs który udostępnia metody potrzebne do iterowania poprzez każdą z kolekcji. Dzięki Iteratorowi możemy przeprowadzać takie operacje jak odczyt oraz usunięcie.

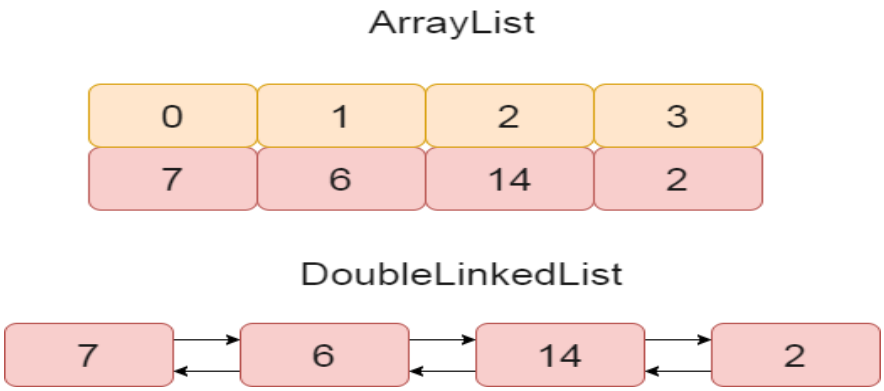
18. Jaka jest zasadnicza różnica pomiędzy Kolejką (Queue) a stertą (Stack)?

Raczej rzadko zadawane pytanie (z powodu rzadkiego użytkowania stert) – aczkolwiek warto znać na nie odpowiedź. Otóż jak mówiliśmy wcześniej kolejki zazwyczaj korzystają z FIFO (first in, first out) natomiast sterty działają w oparciu o LIFO (last in, first out) tzn. że ostatni element który dostał się do kolejki będzie pierwszym który się z niej wydostanie.



19. Jaka jest różnica między LinkedList i ArrayList?

Pierwszą i najważniejszą różnicą pomiędzy tymi dwoma rodzajami listy jest to że (jak sama nazwa wskazuje) *ArrayList* jest implementowany przy pomocy dynamicznie zmieniającej swoje rozmiary tablicy, natomiast *LinkedList* jest implementowany przy pomocy podwójnie łączonej listy (*doubly LinkedList*). Co to tak właściwie oznacza? Aby ułatwić podrzucamy rysunek:



Rys. 4 Różnice pomiędzy ArrayList a Double LinkedList

Jak widać ArrayList przechowuje informacje o tym pod jakim indeksem znajduje się dana wartość, natomiast Double LinkedList przechowuje informacje o następnym i poprzedzającym elemencie. Wynikają z tego następujące różnice:

– odczyt elementu – ponieważ ArrayList działa na indeksach jest znacznie szybszy niż LinkedList, potrzebna jest mniejsza ilość operacji aby odczytać wartość pod danym indeksem. Aby odszukać daną wartość w LinkedList należy przeiterować przez całą listę, co z pewnością jest znacznie bardziej czasochłonne.

– usuwanie elementu – w tej sytuacji korzystniej zachowuje się LinkedList. Spowodowane jest to tym że po usunięciu danego elementu zmianie podlegają tylko odnośniki do kolejnego (bądź poprzedniego) elementu w dwóch elementach obok usuwanego. W przypadku ArrayList musimy zmienić położenie wszystkich elementów znajdujących się za usuwanym elementem – powoduje to znacznie większy koszt w porównaniu do LinkedList.

– dodawanie elementów – dodawanie elementów w LinkedList jest łatwe i szybkie w porównaniu do ArrayList gdyż nie istnieje ryzyko powiększenia tablicy i kopiowania zawartości do nowej tablicy. Co to oznacza? Jeżeli dodana wartość sprawia że tablica na której opiera się ArrayList staje się pełna, program musi stworzyć nową, większą tablicę i przekopiować tam wszystkie dane z poprzedniej tablicy. Dodatkowo ArrayList musi również zmienić indeksy poszczególnych wartości jeżeli element nie jest dodany na końcu listy.

– pamięć – LinkedList potrzebuje więcej pamięci gdyż w odróżnieniu od ArrayList (w którym każdy indeks przechowuje jakieś dane) musimy przechowywać informacje o adresach i danych poprzedniego oraz następnego elementu.

20. Jaka jest różnica pomiędzy HashMap i Hashtable?

Istnieje kilka różnic:

– Hashtable jest zsynchronizowane (synchronized) natomiast HashMap nie jest. Oznacza to że HashMap funkcjonuje lepiej w aplikacjach które nie są wielowątkowe gdyż niezsynchronizowane obiekty mają wyższą wydajność niż obiekty zsynchronizowane.

– Hashtable nie zezwala na używanie wartości null jako klucza i wartości, natomiast HashMap zezwala na używanie jednego klucza o wartości null, oraz nielimitowanej ilości

wartości null.

– HashMap do Iteracji poprzez wartości obiektów wykorzystuje Iterator, natomiast Hashtable enumerator.

– HashMap jest dużo szybszy niż Hashtable.

Warto zwrócić uwagę że Hashtable może być również zastąpiony przy pomocy ConcurrentHashMap, który również jest przystosowana do pracy z aplikacjami wielowątkowymi, a oprócz tego oferuje również większą szybkość działania.

21. Jaka jest różnica pomiędzy interfejsami Iterator oraz ListIterator?

ListIterator pozwala na przeszukiwanie listy w obie strony (podczas gdy Iterator przeszukuje listę tylko w jedną stronę). ListIterator może być używany tylko do List. Ponadto ListIterator pozwala również na dodawanie elementów, oraz zmianę ich wartości – podczas gdy zwykły Iterator umożliwia jedynie usuwanie elementów.

22. Jaka jest różnica pomiędzy HashSet a TreeSet?

Występuje kilka różnic pomiędzy tymi strukturami:

– HashSet zachowuje elementy w losowej kolejności (ta klasa nie jest w stanie zagwarantować nam stałej kolejności) podczas gdy TreeSet gwarantuje że przechowywane wartości będą posortowane w kolejności naturalnej bądź ułożone według naszych specyfikacji zawartych w konstruktorze.

– HashSet może przechowywać obiekt o wartości null, podczas gdy TreeSet nie może.

– Hashset oferuje stałą szybkość działania (O(1)) podczas gdy szybkość działania TreeSet jest wyliczana logarytmicznie (log(n)).

23. Na jakiej zasadzie działa HashMap?

https://nullpointerexception.pl/pytania-rekrutacyjne-jak-dziala-hashmapa-w-javie/

Trudne i ważne pytanie – przed każdą rozmową warto sobie je powtórzyć 😉

Żeby odpowiedzieć sensownie na to pytanie, trzeba zacząć od definicji HashMapy.

HashMapa to struktura danych, która pozwala przechowywać dane typu klucz-wartości. W większości przypadków pozwala pobierać i dodawać je w stałym czasie O(1) oraz działa ona na bazie hashowania.

- ***Co to znaczy, że działa na bazie hashowania?***

W mapie mamy dostępne metody put() i get(). Gdy wywołujemy metodę put(), musimy podać klucz i wartość. Mapa wywołuje metodę hashcode() na obiekcie klucza, a następnie używa własnej funkcji hashującej, by określić, w którym bucket (kubełku) zostanie umieszczona wartość reprezentowana przez Map.Entry. Co ważne, w Map.Entry mapa przechowuje zarówno klucz jak i wartość.

Gdy próbujemy odczytać z hashmapy wartość, odbywa się podobny process. Wywołujemy metodę get(), podając jako parametr klucz. Hashmapa wywołuje metodę hashcode dla klucza i używa funkcji hashującej, żeby określić, w którym buckecie znajduje się dana wartość. Jeśli odnajduje wartość w tym buckecie, to jest ona zwracana, jeśli nie zwracany jest null.

W tym momencie rekruterzy dopytują zwykle: „Co się dzieje, gdy dwa klucze mają ten sam hascode i czy jest to dopuszczalne?" lub „Co może być kluczem i jakie warunki musi spełniać klucz mapy?". Oczywiście możesz też sam omówić wszystkie aspekty hashmapy, ale zwykle pojawiają się tego typu dodatkowe pytania.

- ***Kolizje w hashmapie***

Z odpowiedzią na pierwsze pytanie wiąże się zagadnienie kolizji w hashmapie i to jak hashmapa sobie z nimi radzi. Także bezpośrednio z tym związane jest to, że klasa klucza musi mieć zaimplementowaną metodę hascode() oraz equals().

W sytuacja, kiedy dla dwóch obiektów klucza przy wywołaniu metody hashcode() zwracana jest ta sama wartość, mamy do czynienia z kolizją. Hashmapa dodatkowo wywołuje dla takich kluczy metodę equals(). Jeśli metoda equals() dla dwóch kluczy zwróci false, to znaczy, że są to dwa różne klucze. Wtedy mapa umieszcza dwie wartości w tym samym buckecie. Kolejne obiekty dla tych samych kluczy są umieszczane w LinkedList, co może prowadzić do degradacji wydajności – dlatego ważne jest, by dobrze zaimplementować metody hashcode() i equals(). Natomiast, jeśli metoda equals() zwróci true, to oznacza, że są to te same klucze i stara wartość jest zastępowana nową.

- ***Skąd możemy mieć taką pewność, że te klucze są takie same?***

Wynika to z kontraktu pomiędzy metodami equals() i hashcode().

Kontrakt pomiędzy metodami equals(Object object) i hashcode():

- Kiedykolwiek metoda hashcode() jest wywołana na tym samym obiekcie więcej niż raz, musi za każdym razem zwrócić tą samą wartość (int) hashcode niezależnie od metody equals(Object).
- Jeśli dwa obiekty są równe zgodnie z metodą equals(Object), wtedy każde wywołanie metody hashcode() dla tych obiektów musi zwrócić tą samą wartość (hashcode'y są równe).
- Jeśli dwa obiekty nie są równe zgodnie z metodą equals(Object), nie muszą zwracać różnych hashcodów. Mówiąc inaczej obiekty mogą mieć zgodny hascode i być nie równe zgodnie z metodą equals(Object) (equals zwróci false).

- Jak odczytujemy wartości w przypadku kolizji?

Dzieje się to w analogiczny sposób, jak w przypadku niewystąpienie kolizji. W tym wypadku, dodatkowo po znalezieniu odpowiedniego bucketa, jest iterowana linked lista i na każdym elemencie (sprawdzany jest klucz w Map.Entry) jest wywoływana metoda equals(). Gdy metoda ta zwróci true, zwracana jest jest wartość.

- ***Klucze HashMap***

Kluczem hashmapy może być każdy obiekt, który ma odpowiednio zaimplementowane metody hascode() i equals(). Najlepiej, gdy obiekt klucza jest obiektem niezmiennym (immutable). W przypadku, gdy klucze mapy nie są niezmienne, może to prowadzić do nieprzewidywalnych rezultatów. Wyobraź sobie, że zapisujesz jakiś obiekt pod jakimś kluczem, nadal masz referencję do obiektu tego klucza, po chwili zmieniasz go i zmienia się też jego hashcode. W innym miejscu programu próbujesz pobrać z mapy pożądaną wartość. Już nie masz referencji do obiektu klucza, więc tworzysz go na nowo. I niestety, mapa zwraca Ci null, mimo że wartość, która cię interesuje jest ciągle w mapie (nie wiem, czy to jest do końca dla Ciebie jasne, jeśli nie – daj znać w komentarzu).

Dlatego obiekty takich klas jak String, czy Integer (i inne wrappery prymitywów) są najczęściej wykorzystywanymi kluczami w Hashmapach. Są niezmienne (immutable) i ich klasy są final co znaczy, że nie mogą być rozszerzane (nie można po nich dziedziczyć).

Oczywiście nie ma żadnego problemu, żeby stworzyć swoją własną klasę dla klucza. Wystarczy, że taka klasa będzie miała odpowiednio zaimplementowane metody equals() i hascode() oraz będzie niezmienna (nie jest to konieczne, ale jest to dobrą praktyką).

Poniżej przykładowa implementacja klasy, która może być wykorzystana jako klucz w hashmapie. Pola klasy są inicjalizowane przez konstruktor, nie mogą być w inny sposób

```java
import java.util.Objects;

public final class MyKey {
    private final String myName;
    private final int myAge;

    public MyKey(String myName, int myAge) {
        this.myName = myName;
        this.myAge = myAge;
    }

    public String getMyName() {
        return myName;
    }

    public int getMyAge() {
        return myAge;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        MyKey myKey = (MyKey) o;
        return myAge == myKey.myAge &&
                Objects.equals(myName, myKey.myName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(myName, myAge);
    }
}
```

użycie:

```java
Map<MyKey, String> map = new HashMap<>();
map.put(new MyKey("Jan Kowalski", 22), "Jakaś wartość");
```

zmienione, więc klasa spełnia warunki niezmienności. Zostały także zaimplementowane metody equals() i hascode() (Tutaj implementacja została wygenerowana za pomocą środowiska Intellij Idea):

- Optymalizacja w Javie 8

Warto także wspomnieć o jednej optymalizacji, która została wprowadzona w Javie 8 i dotyczy ona kolizji. Normalnie w przypadku kolizji tworzona jest LinkedLista, co w najgorszym przypadku może degradować wydajność pobierania elementów z HashMapy do O(n) (gdzie normalnie jest to O(1)). Żeby poprawić tę sytuację, architekci Javy postanowili zamienić LinkedListę na drzewo binarne. Dzieje się to przy odpowiedniej wielkości listy (TREEIFY_THRESHOLD = 8). Wtedy wydajność pobierania elementu w najgorszym wypadku będzie O(log n).

24. Jaka jest różnica pomiędzy Comparable a Comparator?

Comparable – wszystkie klasy które mają być posortowane przy użyciu tego interfejsu muszą implementować ten interfejs. Wraz z tym interfejsem otrzymywana jest metoda compareTo(Object) która musi zostać zaimplementowana.

Comparator – klasy które mają być posortowane nie muszą implementować tego interfejsu. Inna klasa może implementować ten interfejs aby posortować dane obiekty. Wykorzystując Comparator możemy tworzyć różne algorytmy sortowania opierające się na różnych atrybutach.

	Comparable	Comparator
Logika sortowania	Logika musi być zawarta w tej samej klasie którego obiekt ma być sortowany. Stąd może pojawiać się również nazwa naturalnego porządku(Natural Ordering)	Logika sortowania zawarta jest w osobnej klasie - dzięki czemu możemy pisać różne algorytmy sortowania w zależności od tego według jakich parametrów chcemy sortować.
Implementacja	Klasa która ma podlegać sortowaniu musi zawierać ten interfejs.	Klasy które mają podlegać sortowaniu nie muszą zawierać tego interfejsu. Muszą go natomiast implementować klasy które będą odpowiedzialne za sortowanie naszych obiektów.
Metoda Sortowania	Metoda *compareTo(Object o1) która porównuje* ***ten*** *obiekt z obiektem o1 i zwraca odpowiadającą wartość. (>0 -* ***ten*** *obiekt jest większy niż o1, 0 - obiekty są równe, <0* ***ten*** *obiekt jest mniejszy niż o1*	Metoda *compare(Object o1, Object o2) która porównuje wybrane parametry obiektów o1 i o2 zwracając wartości odpowiadające ich porównaniu. (>0 - o1 > o2, 0 o1==o2, <0 o1<o2)*
Jak użyć	*Collections.sort(List)* - obiekty będą posortowane według metody *CompareTo.*	*Collections(List, Comparator)* - obiekty będą posortowane według metody *Compare* w klasie *Comparator.*

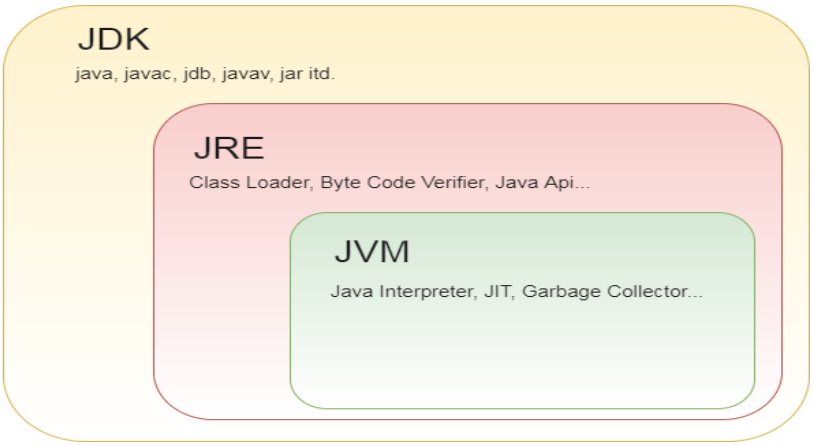
https://blog.it-leaders.pl/rozmowa-kwalifikacyjna-javy-zaden-problem-cz-iii-core/

25. Czym jest JVM?

JVM jest to nic innego jak wirtualna maszyna Javy (Java Virtual Machine). Jest to swojego rodzaju procesor wykonujący skompilowany kod języka Java. Każdy plik źródłowy Javy jest skompilowany w kod bajtowy, który jest wykonywany właśnie poprzez JVM. Głównym założeniem Javy jest akronim WORA (Write Once, Run Anywhere,) tłumaczony, jako „napisz raz, uruchamiaj gdziekolwiek. Aby osiągnąć ten cel potrzebna była właśnie wirtualna maszyna, która działała o poziom wyżej niż kod maszynowy – dzięki temu kodu nie trzeba było kompilować do różnych wersji danego programu dla różnych korzystających z niego platform.

26. Jaka jest różnica pomiędzy JDK oraz JRE?

JRE (Java Runtime Environment) jest wirtualną maszyną Javy, w której Twoje programy są uruchamiane. JDK stanowi natomiast skrót od Java Development Kit, zawarte są w nim wszystkie komponenty potrzebne do tworzenia aplikacji w języku Java. Zawarte są w nim chociażby takie rzeczy jak JRE, kompilator oraz narzędzia takie jak JavaDoc, Debugger.



Rys. 1 Różnica pomiędzy JRE, JDK oraz JVM

27. Co rozumiesz przez pojęcie klasa oraz obiekt?

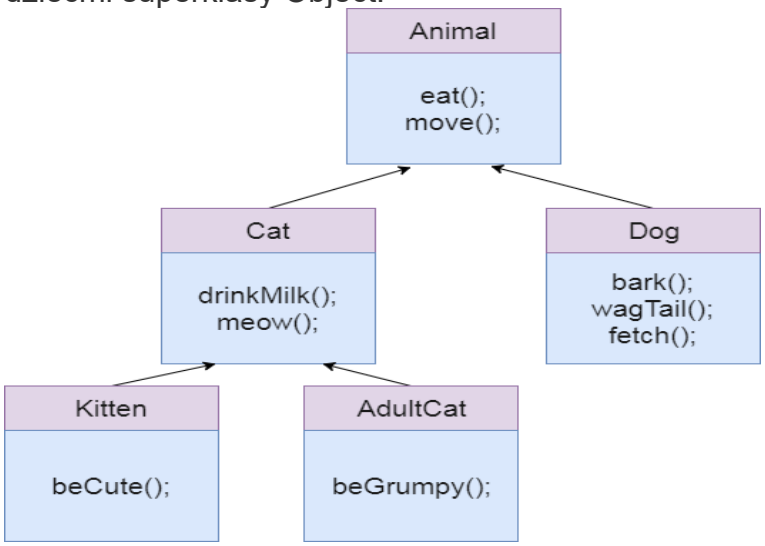
Klasa jest swojego rodzaju schematem na podstawie, którego powstają obiekty. W klasie definiujemy pola zawierające dane, oraz metody wykonywane przez obiekt.

Obiekt jest natomiast konkretną jednostką, która zawiera stany zapisane w polach, oraz zachowania definiowane przez metody. Metody odpowiadają za operacje wykonywane na wewnętrznych stanach obiektu, oraz za komunikację pomiędzy dwoma różnymi obiektami. Obie jednostki są podstawowymi elementami programowania obiektowego.

28. Czym jest polimorfizm?

Nazwa ta wywodzi się z greckiego i oznacza wiele form (z angielskiego Poly = wiele, Morph = zmiana lub forma). Oznacza to tyle, iż obiekt ma możliwość przyjmowania wielu form. Każdy obiekt, który jest w stanie pozytywnie przejść więcej niż jeden test IS-A (z ang. IS-A – jest – angielski odpowiednik lepiej tłumaczy według nas istotę polimorfizmu)

jest polimorficzny. W Javie wszystkie obiekty są polimorficzne, ponieważ wszystkie są dziećmi superklasy Object.



Rys. 2 Jak widać na rysunku klasa „Kitten” (kociak) zdaje test Is-A (Kitten IS-A Cat, Kitten IS-A Animal), a więc jest polimorficzny. Chcemy tylko dodać – wiemy że kotom nie powinno się podawać mleka

29. Czym jest interfejs w Javie i jakie mają zastosowanie?

Interfejs jest to typ referencyjny w Javie. Zawiera pewne podobieństwo do klasy, ponieważ definiowane są w nim zmienne (będące domyślnie finalne oraz statyczne) i metody (ich typ zwracanej zmiennej, nazwę oraz argumenty przyjmowane przez metody), aczkolwiek nie zawiera ich konkretnej implementacji. Klasy mogą implementować dowolną ilość interfejsów (w odróżnieniu dziedziczenia po innych klasach), dziedzicząc wszystkie ich metody, oraz implementując ich funkcjonalność. W ósmej odsłonie Javy uległo to jednak trochę zmianie – metody typu domyślne (default) oraz statyczne mogą zawierać konkretną funkcjonalność w interfejsie. Warto również zaznaczyć, że interfejsy mogą być rozszerzane przez inne interfejsy. Nie istnieje tutaj ograniczenie jak w klasie gdzie metody mogą być dziedziczone poprzez tylko jedną klasę – Interfejsy mogą dziedziczyć po nieograniczonej ilości innych interfejsów.

```
public interface MainAntagonist extends Genius, Villian {

    static void provideTooLongSpeech(String protagonistName) {
        System.out.println("Mr. " + protagonistName + "now you can't run from me!...");
    }

    Integer countNumberOfGoons();

    void useSuperWeapon();

}
```

Rys. 3 Przykładowa klasa interfejsu – jak widać, w odróżnieniu od klas, interfejs może rozszerzać kilka interfejsów. Ponadto może również zawierać metody statyczne które posiadają już funkcjonalność (Java 8).

30. Czym jest klasa abstrakcyjna?

Klasa abstrakcyjna jest to klasa, która została zdefiniowana poprzez słówko kluczowe abstract. Może ona zawierać zwykłe metody bądź metody abstrakcyjne (w klasach abstrakcyjnych brakuje konkretnej funkcjonalności). Nie można utworzyć instancji klasy abstrakcyjnej – inne klasy mogą natomiast dziedziczyć funkcjonalność po tych klasach. Jeżeli klasa nie zaimplementuje funkcjonalności wszystkich metod zadeklarowanych jako abstrakcyjne, musi być również zadeklarowana jako klasa abstrakcyjna.

```
import java.math.BigDecimal;

public abstract class Protagonist extends Hero {

    public void introduceYourself(String firstName, String secondName) {
        System.out.println("My name is " + secondName + ", "+ firstName + " " + secondName)
    }

    public abstract void tellSadBackstory(String sadEvent);

    public abstract BigDecimal countEnemiesDefeated();

}
```

Rys. 4 Przykładowa klasa abstrakcyjna. Jak widać może ona zawierać pewną funkcjonalność w swoich metodach (tych które nie są zadeklarowane abstrakcyjne). W odróżnieniu od interfejsu, może dziedziczyć jedynie po jednej klasie.

```
public abstract class Hero {

    public void heAlwaysWins() {
        System.out.println("I Always win!");
    }

}
```

Rys. 5 Klasa abstrakcyjna, po której dziedziczy klasa abstrakcyjna Protagonist.

31. Jaka jest różnica pomiędzy interfejsem a klasą abstrakcyjną?

Po pierwsze – interfejsy mogą zawierać tylko deklaracje danych metod – nie mogą zawierać w sobie konkretnie zaimplementowanych metod (poza metodami statycznymi oraz domyślnymi). Klasy abstrakcyjne mogą natomiast zawierać konkretną implementację danych metod, dzięki czemu mogą definiować podstawową funkcjonalność.

Kolejną ważną różnicą jest to, że interfejs może rozszerzać mnogą ilość interfejsów, oraz że klasy mogą implementować również mnogą ilość interfejsów. Rozszerzając klasę abstrakcyjną klasa może uczestniczyć jedynie w jednej hierarchii, natomiast używając interfejsów klasa może uczestniczyć w wielu typach hierarchii.

Po trzecie, interfejsy wymagają od użytkownika implementacji wszystkich metod w nich zawartych (co może okazać się dosyć męczące). Klasy abstrakcyjne nie wymagają tego od użytkownika, mogą to również ułatwić umożliwiając podstawową implementację danej funkcjonalności.

32. Jaka jest różnica pomiędzy wyjątkiem sprawdzonym oraz niesprawdzonym (checked/uncheckedException)?

Sprawdzony wyjątek to wyjątek, który powstaje przy kompilacji aplikacji. Programista musi to uwzględnić w swoim kodzie wykonać jedną z dwóch czynności – stwierdzić jak poradzić sobie z danym wyjątkiem, bądź podać go o poziom wyżej przy użyciu słówka kluczowego throws. Wyjątek niesprawdzony nie jest wykrywany podczas kompilacji, jego przyczyną są najczęściej błędy logiczne w kodzie (np. dzielenie przez zero, bądź wskazanie na wartość null).

33. Jakie typy danych występują w Javie?

Typy danych można podzielić przede wszystkim na dwa główne typy – prymitywne oraz złożone (nie-prymitywne). Prymitywne typy danych zawsze zaczynają się małą literą (większość IDE nadaje im również specjalny kolor, podobnie jak słówkom kluczowym. Nazwy typów złożonych powinny rozpoczynać się dużą litera. Do typów prymitywnych należą: **boolean, char, byte, short, int, long, float**.
Natomiast do typów złożonych należą wszystkie typy danych stworzone przez programistów, przykładowo String, Array, List…

34. Skoro istnieją typu prymitywne (np. boolean) i wiemy, że ich nazwy rozpoczynają się małą literą to czym w takim razie są klasy typu Boolean, Byte, Float?

Są to tak zwane klasy opakowujące (wrapperclasses). Pozwalają nam one na opakowanie prymitywnych typów danych. Dzięki temu jesteśmy w stanie używać prymitywnych typów danych (opakowanych we wrappery) np. w listach lub mapach. Występuje tu również zjawisko autoboxingu oraz unboxingu – pierwsze z nich opisuje automatyczną konwersje danych prymitywnych do klas opakowujących (np. int do Integera), drugie natomiast opisuje tą samą operację, ale w odwrotną stronę.

35. Jakie nowe funkcjonalności zostały wprowadzone w Javie 8 i Javie 9?

https://reflectoring.io/java-release-notes/
https://github.com/mzerek/playingWithJava -> examples

Java8:
- lambra Expressions - () -> {}
- method reference - Object::toString
- try-with-resource feature
- interface default methods
- annotations in mamy places: local variables, costructors calls, generic, type casting, collections

Java9:
- module
- diamond operator inside the inner anonymous class
- private method in interface

36. Czym jest Serializacja (Serialization) i Deserializacja (Deserialization)?

Jest to proces, w którym obiekt jest konwertowany na sekwencje bitów. Obiekt, który został poddany serializacji, może zostać zapisany w postaci pliku na dysk twardy, lub przesyłany poprzez strumienie danych. Jeżeli proces ten odbywa się w stronę odwrotną (tzn. z pliku do obiektu) obiekt poddawany jest wtedy procesowi deserializacji. Aby była możliwa serializacja, klasa obiektu musi implementować interfejs Serializable.

37. Czym jest proces Garbage Collection (Zbierania odpadów)?

Jest to proces przeprowadzany przez program GarbageCollector (zbieracz odpadów), który wykonywany jest w wirtualnej maszynie javy (JVM). Proces ten pozbywa się nieużywanych przez aplikację obiektów. Dzięki temu uwalniana jest pamięć, która może być wykorzystana do zapisywania nowych obiektów. Pierwszym podejmowanym w trakcie działania procesu jest oznaczanie czy dana pamięć jest używana czy nie – dzięki temu w następnym procesie (usuwaniu normalnym) GarbageCollectorwie, których obiektów zajmujących pamięć ma się pozbyć. Następnie, aby poprawić wydajność można również zastosować kompaktowanie pamięci, przesuwając pozostałe używane obiekty obok siebie, dzięki czemu dostęp do pamięci jest prostszy oraz szybszy.

38. Jaka jest różnica pomiędzy porównywaniem obiektów przy pomocy metody .equals() oraz ==?

W Javie operator dwóch znaków „=” używany jest, aby sprawdzić czy dwa obiekty odwołują się do tego samego miejsca w pamięci. Metoda .equals() używana jest natomiast do porównywania obiektów poprzez ustalone kryteria (metoda jest zawarta w klasie Object a wiec implementują je wszystkie klasy, ponadto może zostać przez nas nadpisana w celu porównywania konkretnych pól danych obiektów). Dlatego jeżeli chcemy sprawdzić czy dwa obiekty typu String mają taką samą wartość powinniśmy używać metody .equals(). W przypadku, gdy chcemy sprawdzić czy dwa obiekty odwołują się do tego samego miejsca w pamięci (rzadkie przypadki – najczęściej == używane jest do porównywania prymitywów) używamy operatora ==.

https://www.bdabek.pl/jak-dziala-adnotacja-transactional-w-springu/
39. Jak działa adnotacja @Transactional w Springu?

Użycie adnotacji @Transactional stało się na tyle powszechne, że często nie wiemy, po co to robimy. Transakcje w Springu są dostarczone domyślnie i jako użytkownicy frameworka nie martwimy się jak działają pod spodem. Zdarza się jednak, że niewiedza działa na naszą niekorzyść i nie inaczej jest w tym przypadku. Transakcje są czymś niewidocznym. Niby wiemy jak to działa, lub powinno działać, ale mimo wszystko gdzieś tam pod spodem dzieje się coś nieznanego. W tym artykule przybliżymy i odkryjemy to nieznane.

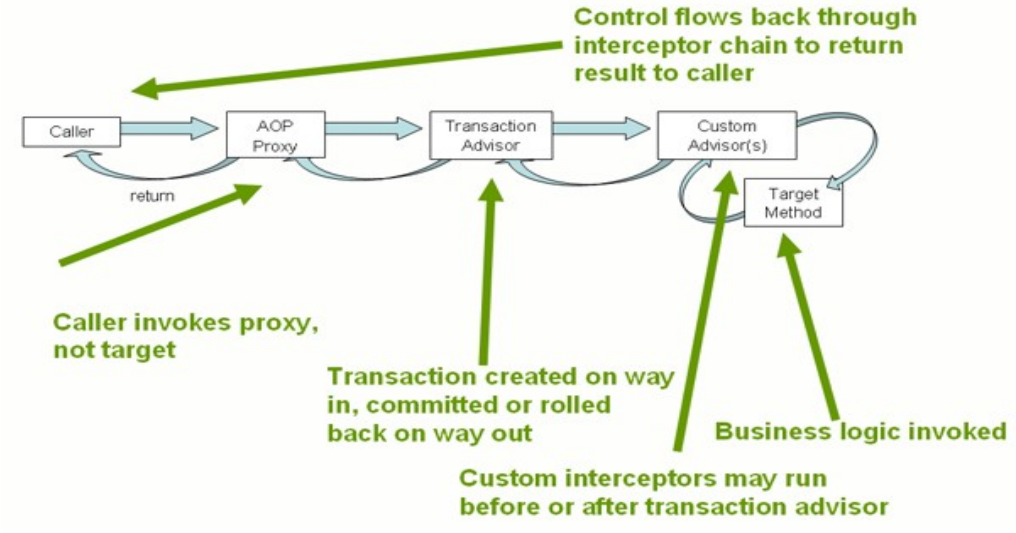
- 1. @Transactional w Springu

Transakcje w springu można uruchomić używając adnotacji @Transactional na metodzie lub klasie. Kiedy metoda oznaczona jako transakcyjna zostaje wywołana, Spring przechwytuje wywołanie i na naszą metodę nakłada proxy (*lub manipuluje naszym kodem bajtowym, jeżeli zmieniliśmy domyślny proxy mode dla springa). Proxy uruchamia TransactionInterceptor, który zarządza transakcją. a następnie w klasie

TransactionAspectSupport (klasa rodzic dla TransactionInterceptor), wywoływana jest docelowa metoda biznesowa. Żeby lepiej to zobrazować posłużę się obrazkiem z dokumentacji springa.

Warto wiedzieć, że samo nałożenie adnotacji @Transactional na metodę nie jest wystarczające. Oprócz tego należy spełnić jeszcze 3 warunki:

- klasa, w której znajduje się metoda musi być bean'em springowym,
- metoda, na której znajduje się adnotacja musi być publiczna (przy założeniu, że używasz domyślnego proxy mode),
- metoda oznaczona jako @Transactional musi być wołana z innego beana springowego.



- 2. Wyjątki w transakcjach

Wyobraź sobie taki scenariusz. Masz jakiś serwis, w którym zapisujesz zadanie do bazy danych. Twój serwis potrafi jednak wyrzucić jakiś checked exception.

Zapisze zadanie do bazy czy nie?

```
@Override
@Transactional
public void create(Task task) throws Exception {
    em.persist(task);
    // logika biznesowa która rzuca wyjątek
    throw new Exception();
}
```

Z jednej strony widać że poleciał błąd i naturalnym by się wydawało że powinien był wykonać się rollback. Jednak, jak możesz domyślać się z kontekstu – rollback nie został wykonany. Zamiast tego akcja została zacommitowana do bazy i pozwól, że wyjaśnię co się stało i czy da radę temu jakoś zaradzić. Na początku zacytuję dokumentację:

Any RuntimeException triggers rollback, and any checked Exception does not.

Dlaczego architekci Springa podjęli właśnie taką decyzję? Oczywiście jest to decyzja

projektowa – cytuję dalej dokumentację z innego miejsca:

While the Spring default behavior for declarative transaction management follows EJB convention(roll back is automatic only on unchecked exceptions), it is often useful to customize this behavior.

Jeżeli zajrzymy w kod aspektu transakcji znajdujący się w TransactionAspectSupport, który odpowiedzialny jest za przechwytywanie wyjątków, dojdziemy do metody completeTransactionAfterThrowing. Metoda ta ma w sobie sprawdzenie txInfo.transactionAttribute.rollbackOn(ex).

```
if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.rollbackOn(ex)) {
    try {
        txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
    }
    ...
}
```

Doprowadza nas to do klasy RuleBasedTransactionAttribute i jej metody rollbackOn. Nie będę przeklejał tutaj całej metody, ale skupię się na fragmencie, który pomoże nam zrozumieć w jaki sposób możemy rozszerzyć @Transactional o rollbackowanie wyjątków sprawdzanych. Zauważ, że jest tu robiona iteracja na this.rollbackRules. I to jest właśnie klucz.

```
---
if (this.rollbackRules != null) {
    for (RollbackRuleAttribute rule : this.rollbackRules) {
        int depth = rule.getDepth(ex);
        if (depth >= 0 && depth < deepest) {
            deepest = depth;
            winner = rule;
        }
    }
}
---
```

Używając adnotacji @Transactional możemy do niej przekazać różne parametry, między innymi parametr rollbackFor, w którym definijemy klasy dziedziczące po Throwable, jakie mają być brane pod uwagę przy rollbackowaniu. Oczywiście dotyczy się to wyjątków przechwytywanych (checked exceptions). Więcej info w dokumentacji – @Transactional Settings.

- 3. Propagacja transakcji

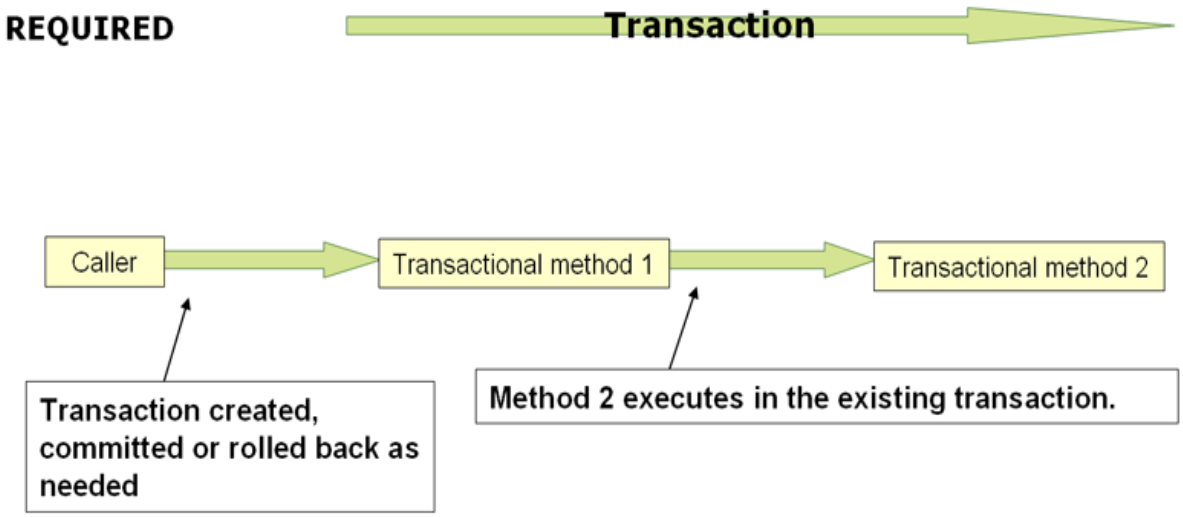
Jest ważne aby wiedzieć, jak będą zachowywały się transakcje w naszym systemie, gdy mamy kilka wywołań metod a na każdej z nich adnotację @Transactional. Dodatkowo, dosyć często, pytanie o to, co się dzieje z transakcją jest zadawane na rozmowach rekrutacyjnych.

No to spójrzmy jakie typy propagacji są dostępne na adnotacji @Transactional

```
public enum Propagation {
    REQUIRED(TransactionDefinition.PROPAGATION_REQUIRED),
    SUPPORTS(TransactionDefinition.PROPAGATION_SUPPORTS),
    MANDATORY(TransactionDefinition.PROPAGATION_MANDATORY),
    REQUIRES_NEW(TransactionDefinition.PROPAGATION_REQUIRES_NEW),
    NOT_SUPPORTED(TransactionDefinition.PROPAGATION_NOT_SUPPORTED),
    NEVER(TransactionDefinition.PROPAGATION_NEVER),
    NESTED(TransactionDefinition.PROPAGATION_NESTED);
    ...
}
```

- 3.1. REQUIRED

Jest do domyślny poziom propagacji. Spring wchodząc do metody oznaczonej adnotacją @Transactional sprawdza czy istnieje aktywna transakcja i jeżeli nie, to tworzy nową. W przypadku, gdy transakcja już istniała, logika metody jest aplikowana do istniejącej transakcji.




```
// nie stosuj w taki sposób
@Transactional(propagation = Propagation.REQUIRED)
public void requiredExample(String user) {
    // ...
}

// preferowane użycie - Propagation.REQUIRED jest ustawiany domyślnie
@Transactional
public void requiredExample(String user) {
    // ...
}
```

```
if (isExistingTransaction()) {
    if (isValidateExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
return createNewTransaction();
```

Pseudokod logiki aplikującej REQUIRED

- 3.2. SUPPORTS

Ten typ propagacji sprawdza czy istnieje transakcja i jeżeli tak, to jej używa. W przeciwnym razie metoda jest wykonywana bez użycia transakcji.

```java
@Transactional(propagation = Propagation.SUPPORTS)
public void supportsExample(String user) {
    // ...
}
```

Pseudokod:

```
if (isExistingTransaction()) {
    if (isValidateExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
return emptyTransaction;
```

- 3.3. MANDATORY

Szuka istniejącej transakcji i jej używa. W przypadku, gdy nie może znaleźć istniejącej transakcji wyrzuca wyjątek.

```java
@Transactional(propagation = Propagation.MANDATORY)
public void mandatoryExample(String user) {
    // ...
}
```

Pseudokod:

```
if (isExistingTransaction()) {
    if (isValidateExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
throw IllegalTransactionStateException;
```

- 3.4. REQUIRES_NEW

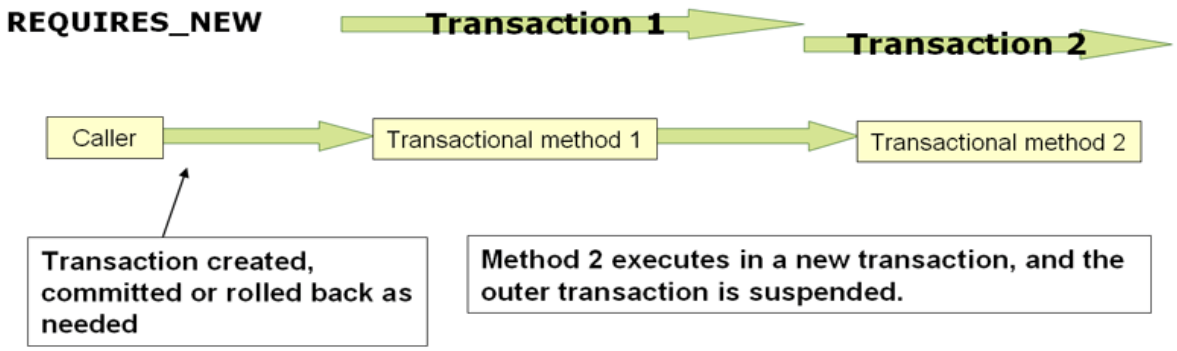
Zawsze powstaje nowa transakcja. W przypadku, gdy wcześniej jakaś transakcja była już otwarta, zostaje ona wstrzymana.

```java
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void requiresNewExample(String user) {
    // ...
}
```

Pseudokod:

```
if (isExistingTransaction()) {
    suspend(existing);
    try {
        return createNewTransaction();
    } catch (exception) {
        resumeAfterBeginException();
        throw exception;
    }
}
return createNewTransaction();
```

W taki sposób możesz to sobie zilustrować.



- 3.5. NOT_SUPPORTED

Sprawdza, czy istnieje transakcja i jeżeli znajduje istniejącą, to ją wstrzymuje. Logika biznesowa dalej wykonywana jest już bez użycia transakcji.

```java
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void notSupportedExample(String user) {
    // ...
}
```

Pseudokod:

```
if (isExistingTransaction()) {
    suspend(existing);
}
return continueWithBusinessLogic();
```

- 3.6. NEVER

Wykonuje metodę bez użycia transakcji. W przypadku gdy transakcja istniała, wyrzcany jest wyjątek.

```java
@Transactional(propagation = Propagation.NEVER)
public void neverExample(String user) {
    // ...
}
```

Pseudokod:

```
if (isExistingTransaction()) {
    throw IllegalTransactionStateException;
}
return emptyTransaction;
```

- 3.7. NESTED

Sprawdzane jest czy istnieje transakcja – jeżeli tak, to oznacza ją jako savepoint. Tzn. jeżeli logika biznesowa wyrzuci wyjątek to rollback przywróci nas do tego savepointa. Jeżeli transakcja nie istniała, to propagacja zachowuje się tak samo jak w przypadku REQUIRED.

```java
@Transactional(propagation = Propagation.NESTED)
public void nestedExample(String user) {
    // ...
}
```

Stety, niestety – opcja ta wymaga wsparcia savepointów i działa tylko dla połączeń JDBC. Zatem, jeżeli używasz hibernate'a to przy próbie wykorzystania tego poziomu propagacji dostaniesz wyjątek:

org.springframework.transaction.NestedTransactionNotSupportedException: JpaDialect does not support savepoints - check your JPA provider's capabilities

40. Jak działa stream w JAVA?

https://geek.justjoin.it/zastosowanie-stream-api-z-java-8-przyklady/

Strumienie jako takie nie przechowują danych i w tym sensie nie są żadną strukturą, ponadto też nie modyfikują/zmieniają źródła, na którym operują. Java 8 Stream API w znaczący sposób wykorzystuje/wspiera funkcyjny styl programowania przy wykonywaniu operacji na poszczególnych elementach strumienia np. w wypadku transformacji elementów kolekcji.

Strumień w kontekście Java API zawiera sekwencje elementów, i jak wyżej wspomniałem, umożliwia wykonanie różnych operacji na elementach strumienia. Poniżej utworzymy przykładowy strumień z kilkoma elementami i powiemy trochę więcej na temat tworzenia strumieni, ich charakterystyki oraz operacji na elementach strumienia. Pokażemy tym samym, co z taką sekwencją elementów w strumieniu możemy zrobić.

```
Stream<String> namesStream = Stream.of("John", "Marry", "George", "Paul", "Alice", "Ann");

    namesStream
        .filter(e -> e.startsWith("A"))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

Do utworzenia strumienia została użyta statyczna metoda **Stream.of**. Metoda ta posiada parametr **varargs** toteż możemy stworzyć w ten sposób strumień składający się z dowolnej liczby elementów. Warto podkreślić, że w **Java 8** dodano nową metodę **stream()** do interfejsu **Collection**, stąd też taki strumień można w następujący sposób wykonać:

```
List<String> namesList = Arrays.asList("John", "Marry", "George", "Paul", "Alice", "Ann");

    namesList
        .stream()
        .filter(e -> e.startsWith("A"))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

Oczywiście w obu wypadkach rezultat otrzymany w wyniku działania kodu będzie ten sam:

```
ALICE
ANN
```

Teraz kilka słów na temat typów operacji na elementach strumienia. Generalnie możemy je podzielić na operacje pośrednie i końcowe (**intermediate operations** oraz **terminal operations**), gdzie w dalszej części będę używał angielskich odpowiedników nazw typów operacji. Operacje typu **terminal** są operacjami, które w wyniku działania nic nie zwracają (**void**), bądź też wynik działania takiej operacji nie jest już strumieniem.

Operacje typu **intermediate** w wyniku działania nadal zwracają strumień w postaci sekwencji elementów, stąd też takie operacje można ze sobą łączyć bez używania średnika w kodzie. W naszym przykładzie **filter**, **map** oraz **sorted** są operacjami typu **intermediate** natomiast **forEach** jest operacją typu **terminal**.

Dodatkową oraz ważną cechą operacji typu **intermediate** jest **laziness** (tu też będę się trzymał angielskiego terminu, bo tłumaczenie na lenistwo nie najlepiej brzmi w tym kontekście). Żeby to wyjaśnić spójrzmy na poniższy przykład, gdzie nie pojawia się operacja typu **terminal**.

```
List<String> namesList = Arrays.asList("John", "Marry", "George", "Paul", "Alice", "Ann");

    namesList
        .stream()
        .filter(e -> {
            System.out.println("filter: " + e);
            return true;
        });
```

W wyniku działania powyższego kodu nic nie zostanie wydrukowane na konsoli, ponieważ operacje typu **intermediate** zostaną wykonane tylko wtedy, gdy pojawi się operacja typu **terminal forEach** jak na poniższym przykładzie:

```
List<String> namesList = Arrays.asList("John", "Marry", "George", "Paul", "Alice", "Ann");

    namesList
        .stream()
        .filter(e -> {
            System.out.println("filter: " + e);
            return true;
        })
        .forEach(e -> System.out.println("forEach: " + e));
```

Tym razem na konsoli pojawi się wynik, którego się spodziewamy:

filter: John
forEach: John
filter: Marry
forEach: Marry
filter: George
forEach: George
filter: Paul
forEach: Paul
filter: Alice
forEach: Alice
filter: Ann
forEach: Ann

Dzięki temu podejściu streamy mogą optymalizować ilość wywołań poszczególnych metod.

- Strumienie „specjalizowane”

Tak dla przypomnienia **map()** może nam dostarczyć nowy strumień po wykonaniu określonej operacji dla każdego elementu oryginalnego strumienia. Nowy strumień może być strumieniem innego typu w porównaniu z oryginalnym, by krócej to ująć metoda map() przekonwertuje obiekty w strumieniu z jednego typu na inny typ.

Warto też wspomnieć o metodzie **filter()**, która dostarcza nam „przefiltrowany” strumień

zawierający tylko elementy z oryginalnego strumienia, spełniające określony warunek, który został wyspecyfikowany przez **Predicate**.

Jak wynika z powyższego przykładu **Stream** (rozumiany jako obiekt) jest strumieniem (sekwencją) referencji do obiektów w naszym akurat wypadku typu **String**. W ogólności strumienie mogą być tworzone z różnego rodzaju źródeł danych, głównie z kolekcji – poprzez wspomnianą wyżej metodą **stream()** dodana do interfejsu **Collection**.

Poza strumieniami składającymi się z „regularnych" obiektów **Java 8** dostarcza specjalizowanych strumieni działających na prymitywnych typach danych typu **int**, **long** czy też **double**. Są to **IntStream**, **LongStream** oraz **DoubleStream**. Te specjalizowane strumienie okazują się przydatne, gdy mamy do czynienia z dużą ilością danych numerycznych.

Jeśli spojrzymy na API **Java 8** to okaże się, że zarówno **IntStream**, jak i **LongStream** oraz **DoubleStream** nie dziedziczą po interfejsie **Stream**, lecz po **BaseStream**, które jest też interfejsem nadrzędnym w hierarchii dziedziczenia w stosunku do **Stream**. Istotną konsekwencją tego faktu jest to iż nie wszystkie operacje/metody z **API** interfejsu **Stream** są wspierane przez implementacje dla **IntStream**, **LongStream** oraz **DoubleStream**.

Dla przykładu, gdy spojrzymy na **min()** oraz **max()** dla **Stream**, dowiemy się, że metody te jako parametru formalnego używają comparatora, zaś w wypadku strumieni specjalizowanych parametr tego typu się nie pojawia. Dodatkowo warto zauważyć iż wspomniane wyżej strumienie specjalizowane wspierają dodatkowo operacje agregujące typu terminal tj. sum() oraz average().

Przykładowo by utworzyć **IntStream** możemy użyć metody **mapToInt()** dla istniejącego strumienia.

```
List<String> strings = Arrays.asList("a1", "a2", "b3", "b4", "c5", "c6");

    strings
        .stream()
        .map(string -> string.substring(1))
        .mapToInt(Integer::parseInt)
        .average()
        .ifPresent(System.out::println); // 3.5
```

W powyższym przykładzie użyłem operacji agregującej **average()** typu terminal. Wynik jej działania to, jak łatwo wywnioskować po nazwie, średnia arytmetyczna wyliczona dla wszystkich elementów typu **int** w strumieniu.

Możemy też w tym celu użyć statycznej metody **of()** by stworzyć **IntStream**:

```
IntStream.of(1, 2, 3, 4, 5, 6)
```

lub też posłużyć statyczną metodą **range()**:

```
IntStream.range(1, 6)
        .forEach(System.out::println);

    // 1
    // 2
    // 3
    // 4
    // 5
```

Jak widzimy pierwsza wartość jest inclusive, zaś ostatnia exclusive.

- Ponowne użycie strumieni w Java 8

Strumienie w Java 8 nie mogą być ponownie użyte. Wywołanie jakiejkolwiek operacji typu terminal dla strumienia spowoduje wyjątek IllegalStateExcpetion, jak na poniższym przykładzie:

```
Stream<String> namesStream =
    Stream.of("John", "Marry", "George", "Paul", "Alice", "Ann");

Predicate<String>  hasName = name -> name.equals("Alice");

namesStream.anyMatch(hasName);  // ok
namesStream.noneMatch(hasName); // exception

// Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
// at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
// at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
// at bosch.com.article.StreamApiDemo.fourthDemo(StreamDemo.java:88)
// at bosch.com.article.StreamApiDemo.main(StreamDemo.java:488)
```

W powyższym przykładzie użyto dwóch operacji typu terminal **anyMatch()** oraz **noneMatch()**. Wywołanie pierwszej z nich (**anyMatch()**) spowodowało „**zamknięcie**” strumienia, a wywołanie kolejnej na „zamkniętym” strumieniu skutkowało wygenerowaniem wyjątku.

By takie ograniczenie w pewien sposób obejść po każdej operacji typu terminal wykonanej na strumieniu należałoby utworzyć nowy strumień. W naszym wypadku możemy skorzystać z interfejsu **Supplier**, który został wprowadzony wraz z **Java 8** (pakiet **java.util.function**). Interfejs ten reprezentuje funkcję, która nie używa żadnego argumentu, a wynikiem jej działania jest „wyprodukowanie” wartości typu T. Jedyną metodą zdefiniowaną dla tego interfejsu jest bezargumentowa metoda **get()** – interfejs ten wspiera programowanie funkcyjne.

Na poniższym przykładzie zastosowanie tego interfejsu dla naszych potrzeb:

```
Supplier<Stream<String>> namesStreamSupplier =
        () -> Stream.of("John", "Marry", "George", "Paul", "Alice", "Ann");

    Predicate<String>  hasName = name -> name.equals("Alice");

    namesStreamSupplier.get().anyMatch(hasName);  // ok
    namesStreamSupplier.get().noneMatch(hasName); // ok
```

Jak widzimy każdorazowe wywołanie metody **get()** dostarcza/produkuje nam „nowy” strumień i bez problemu na takim strumieniu możemy wykonać operację typu **terminal** bez obaw iż zostanie rzucony wyjątek.

- Zaawansowane operacje na strumieniach

Jeśli spojrzymy na **Stream API Javy 8** to zauważymy jak wiele operacji (metod) jest wspieranych przez ten interfejs. W powyższych przykładach zostały wykorzystane jedne z najważniejszych i chyba najczęściej stosowanych w postaci **map()** i **filter()**.

W poniższych przykładach będę korzystał do celów prezentacji z następującej definicji listy zatrudnionych. Dla przejrzystości kodu pominięte zostały settery i gettery w poniższym listingu.

```
public class Employee {
    private int id;
    private double salary;
    private String division;
    private DayJob dayJob;

    public Employee(int id, double salary, String division, DayJob dayJob) {
        this.id = id;
        this.salary = salary;
        this.division = division;
        this.dayJob = dayJob;
    }

    public enum DayJob {
        FULL_TIME("Full-time job"),
        PART_TIME("Part-time job");

        String dayJobDescription;

        DayJob(String dayJobDescription) {
            this.dayJobDescription = dayJobDescription;
        }
    }

    @Override
    public String toString() {
        final StringBuilder stringBuilder = new StringBuilder("Employee {");
        stringBuilder.append(" nid: ").append(this.id);
        stringBuilder.append(", nsalary: ").append(this.salary);
        stringBuilder.append(", ndivision: ").append(this.division);
        stringBuilder.append(", ndayJob: ").append(this.dayJob);
        stringBuilder.append("}");

        return stringBuilder.toString();
    }
...
}

================================================

List<Employee> employees =
                Arrays.asList(
                        new Employee(1, 2000d, "Risk Department", Employee.DayJob.FULL_TIME),
                        new Employee(2,2500d, "Scoring Department", Employee.DayJob.FULL_TIME),
                        new Employee(3,2600d, "Scoring Department", Employee.DayJob.FULL_TIME),
                        new Employee(4,2700d, "Credit Department", Employee.DayJob.FULL_TIME),
                        new Employee(5,2700d, "Credit Department", Employee.DayJob.PART_TIME)
                );
```

Poniżej przykład konwersji strumienia danego typu na strumień innego typu – tutaj przekonwertujemy strumień typu Employee (obiektów typu Employee) na strumień typu String zawierający nazwy poszczególnych departamentów, w których są zatrudnieni pracownicy:

```
employees
     .stream()
     .map(employee -> employee.getDivision())
     .forEach(System.out::println);

        // Risk Department
        // Scoring Department
        // Scoring Department
        // Credit Department
        // Credit Department
```

Poniższy listing filtruje oryginalny strumień pod kątem pracowników, których pensja jest większa niż 2600:

```
employees
     .stream()
     .filter(employee -> employee.getSalary() > 2600d)
     .forEach(employee -> System.out.println(employee.toString()));

         // Employee {
         //      id: 4,
         //      salary: 2700.0,
         //      division: Credit Department,
         //      dayJob: FULL_TIME}
         // Employee {
         //     id: 5,
         //     salary: 2700.0,
         //     division: Credit Department,
         //     dayJob: PART_TIME}
```

Przejdźmy do nieco bardziej zaawansowanych operacji na strumieniach.

- Collect

Z collect() zetknęliśmy się w jednym z poprzednich przykładów. Bardzo często, gdy skończymy „przetwarzanie” strumienia chcielibyśmy przyjrzeć elementom strumienia zwróconego w postaci innej struktury danych np. List, Set lub Map. Z pomocą przychodzi nam niezwykle użyteczna metoda collect(), która wykonuje szereg operacji („przepakowanie” elementów strumienia do innej struktury danych i wykonanie dodatkowych operacji na elementach np. konkatenacji).

Metoda collect() korzysta z Collector, na który to z kolei składają się cztery „składniki” w postaci supplier, accumulator, combiner i finisher (tu znów angielska nomenklatura). Brzmi to może mało zachęcająco, ale Java 8 dostarcza wielu wbudowanych kolektorów przez klasę Collectors, by ułatwić nam życie.

Na poniższym listingu chyba najbardziej powszechny przypadek użycia – dostajemy listę z „przefiltrowanymi” pracownikami pod kątem zarobków.

```
Predicate<Employee> salaryPredicate = employee -> employee.getSalary() > 2600d;

        List<Employee> filteredEmployeeList =
                employees
                      .stream()
                      .filter(salaryPredicate)
                      .collect(Collectors.toList());
```

Jak widzimy nie jest trudno stworzyć listę (**List**) pracowników z elementów strumienia

(przy okazji: uzyskane listy i sety są niemutowalne). Jak było to pokazane w jednym z wcześniejszych listingów, jeśli zamiast listy potrzebowalibyśmy kolekcji typu Set wystarczy, że użyjemy kolektora **Collectors.toSet()**.

Naturalnie nic nie stoi na przeszkodzie by dokonać transformacji elementów strumienia na mapę (**Map**). W tym wypadku należy określić jaki atrybut elementu strumienia ma być kluczem oraz czego chcemy użyć jako wartości. Musimy pamiętać, by klucze posiadały unikalne wartości w przeciwnym razie zostanie rzucony wyjątek **IllegalStateException**. Na poniższym listingu jako naturalnego kandydata na wartość klucza użyto **id** pracownika.

```
 Map<Integer, Employee> employeeMap =
                employees
                    .stream()
                    .limit(2)
                    .collect(Collectors.toMap(
                        Employee::getId,
                        Function.identity(),
                        (key1, key2) -> {throw new IllegalStateException(String.format("duplicate
key value found %s", key1));}
                    ));

        System.out.println(employeeMap);

         // {1=Employee {
         //     id: 1,
         //     salary: 2000.0,
         //     division: Risk Department,
         //     dayJob: FULL_TIME},
         //  2=Employee {
         //     id: 2,
         //     salary: 2500.0,
         //     division: Scoring Department,
         //     dayJob: FULL_TIME}}
```

Dzięki takiej transformacji możemy mieć wgląd na każdego zatrudnionego po jego id. Warto w tym miejscu wspomnieć, że jeśli chcemy jako wartości dla klucza użyć bieżącego elementu kolekcji to jako drugiej funkcji należy użyć **Function.identity()**. Dodatkowo ze względu na czytelność wyniku na konsoli rozmiar strumienia i tym samym mapy został ograniczony do dwóch elementów (operacja **limit**).

Collectors umożliwia nam też grupowanie elementów strumienia po wartości wybranego atrybutu. Poniższy listing grupuje nam pracowników po dywizji/departamencie, w którym są zatrudnieni. Naturalnie grupowania takiego moglibyśmy także dokonać ze względu na zarobki albo wymiar etatu.

```
Map<String, List<Employee>> employeesGroupedByDivision = employees
                .stream()
                .collect(Collectors.groupingBy(employee -> employee.getDivision()));

        employeesGroupedByDivision
                .forEach((division, workers) -> System.out.println(String.format("Division: %s %s",
division, workers)));

         // Division: Risk Department [Employee {
         //     id: 1,
         //     salary: 2000.0,
         //     division: Risk Department,
         //     dayJob: FULL_TIME}]
         // Division: Scoring Department [Employee {
         //     id: 2,
         //     salary: 2500.0,
         //     division: Scoring Department,
         //     dayJob: FULL_TIME}, Employee {
         //     id: 3,
         //     salary: 2600.0,
```

```
//     division: Scoring Department,
//     dayJob: FULL_TIME}]
// Division: Credit Department [Employee {
//     id: 4,
//     salary: 2700.0,
//     division: Credit Department,
//     dayJob: FULL_TIME}, Employee {
//     id: 5,
//     salary: 2700.0,
//     division: Credit Department,
//     dayJob: PART_TIME}]
```

Inną, mniej oczywistą możliwością zastosowania grupowania jest np. podzielenie kolekcji wejściowej (List lub Set) na szereg kolekcji o mniejszym rozmiarze składających z tych samych elementów co kolekcja wejściowa. Na poniższym przykładowym listingu użyto kolekcji składającej się z obiektów typu Integer dla prostoty przykładu.

```
final int chunkSize = 3;
final List<Integer> integers =
        Arrays.asList(2, 4, 6, 8, 10, 12, 14, 16);

AtomicInteger counter = new AtomicInteger(0);

Stream<List<Integer>> integerListStream =
        integers
            .stream()
            .collect(Collectors.groupingBy(integer -> counter.getAndIncrement()/chunkSize))
            .entrySet()
            .stream()
            .map(Map.Entry::getValue);

List<List<Integer>> chunkIntegersList =
        integerListStream
        .collect(Collectors.toList());

System.out.println(chunkIntegersList);
// [[2, 4, 6], [8, 10, 12], [14, 16]]
```

Jak widzimy wejściowa kolekcja **integers** została podzielona na mniejsze kolekcje, których rozmiar nie przekraczał wartości wskazywanej przez parametr **chunkSize**. W moim przypadku tego typu zabieg np. pozwolił na „obejście” limitu Oracle’a związanego z ilością elementów w klauzuli IN (ORA-01795: maximum number of expressions in a list is 1000).

Nie jest to jedyna z możliwości **Collectors**, możemy się pokusić na przykład o złożone statystyki dotyczącą zarobków poszczególnych pracowników – kolektor może nam zwrócić wbudowany obiekt z takimi statystykami jak na poniższym listingu.

```
DoubleSummaryStatistics salarySummarising =
                employees
                    .stream()
                    .collect(Collectors.summarizingDouble(employee -> employee.getSalary()));

        System.out.println(salarySummarising);
        //DoubleSummaryStatistics{count=5, sum=12500,000000, min=2000,000000, average=2500,000000,
max=2700,000000}
```

Oczywiście kolektor też może dostarczyć np. średniej zarobków, jeśli nie potrzebujemy statystyki w postaci wbudowanego obiektu. Wystarczy że użyjemy kolektora jak poniżej.

```
Double averageSalary =
                employees
                        .stream()
                        .collect(Collectors.averagingDouble(employee -> employee.getSalary()));

        System.out.println("Average salary: " + averageSalary);
        // Average salary: 2500.0
```

Kolejną funkcją kolektora, o której warto wspomnieć jest **joining()**. Na poniższym listingu zobaczmy jak **Collectors.joining()** działa.

```
String fullTimeEmployees =
                employees
                      .stream()
                      .filter(employee -> employee.getDayJob().equals(Employee.DayJob.FULL_TIME))
                      .map(employee -> Integer.toString(employee.getId()))
                      .collect(Collectors.joining(" , ", "Employees with id's: ", " work full-
time"));

        System.out.println(fullTimeEmployees);
        // Employees with id's: 1 , 2 , 3 , 4 work full-time
```

Jak widzimy **Collectors.joining()** łączy elementy strumienia w jeden łańcuch znakowy używając podanego przez nas delimitera oraz jako opcjonalnych parametrów **prefix** („Employess with id's: ”) oraz **suffix** („work full-time”). W powyższym przykładzie wykorzystaliśmy **Collectors.joining()** do wydrukowania identyfikatorów pracowników zatrudnionych na pełen etat.

- FlatMap

Jak wiemy strumień może składać się z obiektów o bardziej złożonej strukturze danych, przykładowo może to być strumień zdefiniowany jako **Stream<List<Integer>>**, gdzie tego typu strumień stworzyliśmy dzieląc kolekcję wejściowej (**List** lub **Set**) na szereg kolekcji o mniejszym rozmiarze z wykorzystaniem operacji grupowania (**Collectors.groupingBy()**).

Czasem jednak zachodzi potrzeba uproszczenia takiej struktury i przekształcenia takiego strumienia w np. **Steram<Integer>**. Innymi słowy zamiast strumienia w postaci […[2, 4, 6], [8, 10, 12], [14, 16]…] chcemy uzyskać strumień w postaci […2, 4, 6, 8, 10, 12, 14, 16…]. W tym wypadku **flatMap()** okazuje się przydatną operacją by taką strukturę uprościć i tym samym ułatwić dalsze operacje na elementach strumienia. Poniższy listing pokazuje przykładowe zastosowanie **flatMap()** , by uzyskać pożądany efekt.

```
final List<List<Integer>> slicedIntegers = Arrays.asList(
                Arrays.asList(2, 4, 6),
                Arrays.asList(8, 10, 12),
                Arrays.asList(14, 16)
        );

        final List<Integer> simpleIntegerList =
                slicedIntegers
                        .stream()
                        .flatMap(Collection::stream)
                        .collect(Collectors.toList());

        System.out.println("slicedIntegers: " + slicedIntegers);
        //slicedIntegers: [[2, 4, 6], [8, 10, 12], [14, 16]]

        System.out.println("simpleIntegerList: " + simpleIntegerList);
        //simpleIntegerList: [2, 4, 6, 8, 10, 12, 14, 16]
```

- Reduce

Operacje redukcji łączą wszystkie elementy strumienia w jeden wynikowy element, czyli zwracają „odpowiedź” ze strumienia danych. Redukcje są operacjami typu **terminal** – „redukują” strumień do wartości nie będącej strumieniem. Przykładami operacji redukcji są np. **findFirst()**, **min()**, **max()** – wszystkie zwracają jako wynik **Optional<T>** jako iż ten typ **Optional** jest znacznie lepszym sposobem wskazania iż zwrócona wartość jest pusta np. gdy dany strumień był pusty. W ogólności operacje redukcji możemy przedstawić jako:

T reduce(T identity, BinaryOperator<T> accumulator)

gdzie **identity** jest wartością początkową redukcji lub wartością domyślną, gdy w strumieniu nie ma żadnych elementów, **accumulator** zaś jest funkcją (**BiFunction**), która posiada dwa formalne parametry. Pierwszy to cząstkowy wynik redukcji oraz kolejny element strumienia. Tyle definicji. Na poniższym listingu zsumujemy zarobki wszystkich zatrudnionych na pełen etat używając **reduce()** na naszym strumieniu.

```
Double salariesSumForFullTimeEmployees =
                employees
                      .stream()
                      .filter(employee -> employee.getDayJob().equals(Employee.DayJob.FULL_TIME))
                      .map(Employee::getSalary)
                      .reduce(0.0, Double::sum);

        System.out.println("Salaries sum for full-time workers: " +
salariesSumForFullTimeEmployees);
        //Salaries sum for full-time workers: 9800.0
```

W powyższym przykładzie można było użyć specjalizowanego strumienia DoubleStream i wykonać operację DoubleStream.sum(), by osiągnąć dokładnie ten sam efekt.

W kolejnym przykładzie na poniższym listingu wykorzystamy operację redukcji z użyciem akumulatora (BiFunction), gdzie obydwa operandy są tego samego typu. Porównujemy zarobki osób zatrudnionych na pełny etat i zwracamy dane pracownika z najwyższą pensją.

```
employees
       .stream()
       .filter(employee -> employee.getDayJob().equals(Employee.DayJob.FULL_TIME))
       .reduce(((employee1, employee2) -> employee1.getSalary() > employee2.getSalary() ?
employee1 : employee2))
       .ifPresent(System.out::println);

             // Employee {
             //     id: 4,
             //     salary: 2700.0,
             //     division: Credit Department,
             //     dayJob: FULL_TIME
             // }
```

Dla zainteresowanych w Java 9 dodatkowo dla Stream API pojawiają się takeWhile, dropWhile oraz iterate, gdzie możemy użyć trzech argumentów. Więcej o omawianych rozszerzeniach w Steam API.

- Java 9 – Stream API

https://codecouple.pl/2017/10/06/java-9-stream-api/

1. takeWhile/dropWhile

Operacje na nieskończonych strumieniach są teraz ułatwione, w przykładzie poniżej zostaną wypisane tylko trzy napisy, natomiast program będzie się wykonywał dalej w nieskończoność:

```
Stream.iterate("", s -> s + "s")
        .filter(s->s.length() < 3)
        .forEach(this::log);
```

W nowym API dostajemy dwie metody:

- `takeWhile`
- `dropWhile`

Służą one do “obcinania” strumieni. W pierwszy przypadku zostaną wypisane tylko trzy napisy i strumień się zamknie. W drugim przypadku będą wyświetlane wszystkie napisy, których długość jest większa od trzy:

```
Stream.iterate("", s -> s + "s")
        .takeWhile(s->s.length() < 3)
        .forEach(this::log);
Stream.iterate("", s -> s + "s")
        .dropWhile(s->s.length() < 3)
        .forEach(this::log);
```

2. iterate

Przykład podobny do tego powyżej. Metoda iterate ma teraz swoją trójargumentową wersję. Chcemy wypisać 10 liczb po kolei:

```
Stream.iterate(0, i -> i < 10, i -> i + 1)
        .forEach(this::log);
```

No dobra, ale w **Javie 8** mogliśmy zrobić coś takiego:

```
Stream.iterate(0, i -> i + 1)
        .limit(10)
        .forEach(this::log);
```

Ale teraz wyobraźmy sobie, że mamy bardziej skomplikowany obiekt, na przykład wypisz mi wszystkie daty od początku roku do dziś:

```
Stream.iterate(startDate, date -> date.plusDays(1))
        .filter(date->date.isBefore(LocalDate.now()))
        .forEach(this::log); // To wypisze nam wszystkie daty do dziś, ale strumień będzie się dalej
"kręcił"

Stream.iterate(startDate, date -> date.plusDays(1))
        .peek(this::log)
        .allMatch(date->date.isBefore(LocalDate.now())); // Działający przykład, mniej intuicyjny
```

W **Javie 9** możemy użyć trójargumentowego **iterate**:

```
Stream.iterate(startDate,date -> date.isBefore(LocalDate.now()),  date -> date.plusDays(1))
        .forEach(this::log);
```

3. ofNullable

Kolejny przykład to metoda **ofNullable**. Działa ona tak samo jak **ofNullable** w **Optional** (nie musimy sprawdzać, czy element jest **nullem** i wstawiać **Stream.empty()**):

```
Map<String, Integer> map = new HashMap<>();
map.put("String", 1);
map.put("StringSecond", null);

//JAVA 8
List<Integer> collect = Stream.of("String", "StringSecond")
        .flatMap(element -> {
            Integer temp = map.get(element);
            return temp != null ? Stream.of(temp) : Stream.empty();
        })
```

```
        .collect(toList());

//JAVA 9
List<Integer> collection = Stream.of("String", "StringSecond")
        .flatMap(element -> Stream.ofNullable(map.get(element)))
        .collect(toList());
```

4. Stream z Optional

Ostatnim usprawnieniem jest dodanie metody, która pozwala z Optional'a stworzyć Stream. Bo tak naprawdę Optional to dwa elementy: null lub wartość. Od teraz nie trzeba wykonywać operacji sprawdzania, czy element istnieje tylko od razu można wykorzystać **flatMap**:

```
@Test
void streamFromOptional(){
    //JAVA 8
    Stream.of("string", "second")
            .map(this::getSomething)
            .filter(Optional::isPresent)
            .map(Optional::get)
            .forEach(this::log);

    //JAVA 9
    Stream.of("string", "second")
            .map(this::getSomething)
            .flatMap(Optional::stream)
            .forEach(this::log);
}

Optional<String> getSomething(String text){
    return text.equals("second") ? Optional.of(text)  : Optional.empty();
```

- Parallel Streams

W wypadku strumieni, które zawierają dużą ilość elementów do przetworzenia, operacje na strumieniu w celu zwiększenia wydajności mogą być zostać „zrównoleglone" – innymi słowy mogą być jednocześnie wykonywane przez kilka wątków. By taki efekt osiągnąć potrzebujemy przede wszystkim tzw. parallel stream'u. Strumień taki możemy uzyskać z jakiejkolwiek kolekcji używając metody Collection.parallelStream() w kontradykcji do sekwencyjnego strumienia danych utworzonego poprzez Collection.stream().

Założeniem oczywiście jest, że operacje wykonywane równolegle na strumieniu dadzą taki sam rezultat jakby były wykonywane sekwencyjnie. Odpowiedzialnością programisty jest by każda funkcja przekazana do operacji na strumieniu mogła być „bezpiecznie" wykonana z punktu widzenia wielowątkowości (spójność danych – race condition).

Jeśli wykonamy kod z poniższego listingu, który na pierwszy rzut oka wygląda na poprawny, to będziemy mieli do czynienia z race condition. Po każdym wykonaniu kodu nasza lista pensji, każdorazowo drukowana w ramach operacji forEach() będzie wyglądać inaczej po każdym uruchomieniu metody. W ramach ćwiczenia możemy zastąpić parallelStream() metodą stream() i zobaczyć jaki będzie miało to efekt, gdy będziemy poniższy kod uruchamiali wiele razy.

```
List<Employee> employees =
                Arrays.asList(
                        new Employee(1, 2000d, "Risk Department", Employee.DayJob.FULL_TIME),
                        new Employee(2,2500d, "Scoring Department", Employee.DayJob.FULL_TIME),
                        new Employee(3,2600d, "Scoring Department", Employee.DayJob.FULL_TIME),
                        new Employee(4,2700d, "Credit Department", Employee.DayJob.FULL_TIME),
                        new Employee(5,2700d, "Credit Department", Employee.DayJob.PART_TIME)
                );

final List<Double> salaries = Collections.synchronizedList(new ArrayList<>());

        employees
                .parallelStream()
                .forEach(employee -> {
                    if(employee.getSalary() > 2000d) {
                        salaries.add(employee.getSalary();
                        System.out.println(salaries);
                    }
                });

        // First run:
        // [2600.0]
        // [2600.0, 2700.0, 2500.0, 2700.0]
        // [2600.0, 2700.0]
        // [2600.0, 2700.0, 2500.0]

        // Second run:
        // [2600.0]
        // [2600.0, 2700.0, 2700.0, 2500.0]
        // [2600.0, 2700.0, 2700.0]
        // [2600.0, 2700.0]
```

W momencie gdyby kolekcja salaries nie była zsynchronizowana, to efektem działania byłby jedynie zrzucony wyjątek w postaci **ConcurrentModificationException** (jak na poniższym listingu) spowodowany modyfikacją kolekcji, w momencie gdy kolejny wątek próbuje trawersować tę kolekcję używając jej iteratora – **System.out.println(**salaries**)**.

```
final List<Double> salaries = new ArrayList<>();

        employees
                .parallelStream()
                .forEach(employee -> {
                    if(employee.getSalary() > 2000d) {
                        salaries.add(employee.getSalary());
                        System.out.println(salaries);
                    }
                });

Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
        at java.util.ArrayList$Itr.next(ArrayList.java:851)
        at java.util.AbstractCollection.toString(AbstractCollection.java:461)
        at java.lang.String.valueOf(String.java:2994)
        at java.io.PrintStream.println(PrintStream.java:821)
        at bosch.com.article.StreamApiDemo.lambda$seventeenthDemo$25(StreamApiDemo.java:9)
```

Powyższe przykłady pokazują iż w wypadku stosowania strumieni równoległych (przetwarzanych wielowątkowo) pojawiają się dodatkowe aspekty, na które musimy zwrócić uwagę – pierwsze co się rzuca w oczy to operacje wykonywane równolegle (w oddzielnych wątkach) nie modyfikują współdzielonych zasobów.

Przy okazji omawiania strumieni równoległych warto przyjrzeć się jak działa takie równoległe przetwarzanie. Strumienie te wykorzystują ForkJoinPool, w Java 8 jest to domyślna pula wątków, którą wykorzystuję się do realizacji zadań, które można podzielić. Być może bardziej namacalnie zobaczyć jak zrównoleglone przetwarzanie działa w

kontekście wielowątkowości/strumieni równoległych spójrzmy na poniższy listing.

```
employees
              .parallelStream()
              .filter(employee -> {
                  System.out.println(String.format("Filter person with id %s thread [%s]"
                          , employee.getId(), Thread.currentThread().getName()));
                  return true;
              })
              .map(employee -> {
                  System.out.println(String.format("Map person id %s thread [%s]"
                          , employee.getId(), Thread.currentThread().getName()));
                  return employee.getId();
              })
              .forEach(employeeId -> System.out.println(String.format("For each employee id %s
thread [%s]"
                      , employeeId, Thread.currentThread().getName())));

          // Filter person with id 4 thread [ForkJoinPool.commonPool-worker-4]
          // Filter person with id 2 thread [ForkJoinPool.commonPool-worker-1]
          // Filter person with id 3 thread [main]
          // Filter person with id 5 thread [ForkJoinPool.commonPool-worker-2]
          // Map person id 2 thread [ForkJoinPool.commonPool-worker-1]
          // Map person id 3 thread [main]
          // Map person id 5 thread [ForkJoinPool.commonPool-worker-2]
          // For each employee id 2 thread [ForkJoinPool.commonPool-worker-1]
          // For each employee id 5 thread [ForkJoinPool.commonPool-worker-2]
          // For each employee id 3 thread [main]
          // Filter person with id 1 thread [ForkJoinPool.commonPool-worker-3]
          // Map person id 4 thread [ForkJoinPool.commonPool-worker-4]
          // Map person id 1 thread [ForkJoinPool.commonPool-worker-3]
          // For each employee id 4 thread [ForkJoinPool.commonPool-worker-4]
          // For each employee id 1 thread [ForkJoinPool.commonPool-worker-3]
```

Patrząc na powyższy listing, a w zasadzie na to co pojawiło na konsoli widzimy, które wątki zostały użyte do wykonania operacji na strumieniu. Jak widzimy zostały wykorzystane 4 wątki z puli **ForkJoinPool**. To co zostało wydrukowane na konsole może po każdorazowym uruchomieniu się różnić z racji tego iż przydzielanie wątków z puli jest niedeterministyczne (o problemach z paralles stream przeczytasz *https://dzone.com/articles/think-twice-using-java-8*).