

Spis treści

1. Czym jest String w Javie?.....	2
2. Dlaczego String jest niezmienny?.....	2
3. Co to jest String Pool?.....	2
4. Jak porównywać dwa obiekty typu String?.....	3
5. Jaka jest różnica pomiędzy klasą String, StringBuffer oraz StringBuilder?.....	4
6. Jakie są metody inicjowania obiektów klasy String?.....	4
7. Czym różni się String od innych pochodnych klas?.....	4
8. String, StringBuffer i StringBuilder – która z tych trzech klas typu final?.....	5
9. Czym jest „String Intern”?.....	5
10. Czy w obiekcie klasy String można przechowywać dane wrażliwe? Dlaczego?....	5
11. Czym jest Kolekcja (Collection)?.....	5
12. Czym jest framework Kolekcji (Collections Framework)?.....	6
13. Jakie są zalety frameworka Kolekcji?.....	6
14. Jaka jest różnica pomiędzy Collection (Kolekcją) oraz Collections (Kolekcjami)?	6
15. Jakie są najważniejsze główne interfejsy zawierające się we frameworku kolekcji?.....	6
16. Dlaczego interfejs Map znajduje się we frameworku kolekcji, ale nie implementuje interfejsu Collection?.....	8
17. Czym jest Iterator?.....	8
18. Jaka jest zasadnicza różnica pomiędzy Kolejką (Queue) a stertą (Stack)?.....	8
19. Jaka jest różnica między LinkedList i ArrayList?.....	9
20. Jaka jest różnica pomiędzy HashMap i Hashtable?.....	10
21. Jaka jest różnica pomiędzy interfejsami Iterator oraz ListIterator?.....	10
22. Jaka jest różnica pomiędzy HashSet a TreeSet?.....	10
23. Na jakiej zasadzie działa HashMap?.....	11
24. Jaka jest różnica pomiędzy Comparable a Comparator?.....	14
25. Czym jest JVM?.....	15
26. Jaka jest różnica pomiędzy JDK oraz JRE?.....	15
27. Co rozumiesz przez pojęcie klasa oraz obiekt?.....	15
28. Czym jest polimorfizm?.....	15
29. Czym jest interfejs w Javie i jakie mają zastosowanie?.....	16
30. Czym jest klasa abstrakcyjna?.....	17
31. Jaka jest różnica pomiędzy interfejsem a klasą abstrakcyjną?.....	17
32. Jaka jest różnica pomiędzy wyjątkiem sprawdzonym oraz niesprawdzonym (checked/uncheckedException)?.....	18
33. Jakie typy danych występują w Javie?.....	18
34. Skoro istnieją typy prymitywne (np. boolean) i wiemy, że ich nazwy rozpoczynają się małą literą to czym w takim razie są klasy typu Boolean, Byte, Float?.....	18
35. Jakie nowe funkcjonalności zostały wprowadzone w Javie 8 i Javie 9?.....	18
36. Czym jest Serializacja (Serialization) i Deserializacja (Deserialization)?.....	19
37. Czym jest proces Garbage Collection (Zbierania odpadów)?.....	19
38. Jaka jest różnica pomiędzy porównywaniem obiektów przy pomocy metody .equals() oraz ==?.....	19
39. Jak działa adnotacja @Transactional w Springu?.....	19
40. Jak działa stream w JAVA?.....	26
41. Jak działa JIT? Czy ma jakieś wady?.....	45
42. Do czego służy adnotacja @PostConstruct?.....	45
43. Wymień sposoby wstrzykiwania zależności. Jakiego sposobu używasz i	

dlaczego?.....	46
44. Co to jest hermetyzacja?.....	46
45. Jakie rzeczy weszły od Java 11?.....	47
46. Mikroserwisy vs Monolity.....	47
47. Co wiesz o SOLID, IoC, Dependency Injection? • Literka “S”: video.....	48
48. Dlaczego BigDecimal jest lepszy od double do przechowywania wartości pieniężnych? Dlaczego double nie jest precyzyjny?.....	49
49. Do czego służy BindingResult w Springu?.....	49
50. Wymień poziomy izolacji transakcji i wyjaśnij jak działa jeden z nich?.....	50
51. Jakie właściwości ma adnotacja @Transactional? Co można ustawić?.....	50
52. Jak walidować przychodzące requesty REST w Springu? Jaki błąd wystąpi gdy podamy stringa zamiast inta?.....	51
53. Jak byś zabezpieczył api restowe?.....	51
54. Jak przechowywać hasła w bazie danych? Jak działa Bcrypt?.....	51
55. Jakie znasz zalety / wady stream api w Java?.....	52
56. Jak działa protokół HTTP? Jakie są metody?.....	52
57. Jak wygląda model pamięci w Java? Jakie znasz Garbage Collectory? Jak mają zastosowanie?.....	53
58. Jakie znasz typy ataków webowych? Wymień i objaśnij jeden z nich.....	54
59. Czym jest niezmiennosc (immutability) obiektów?.....	55

<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).

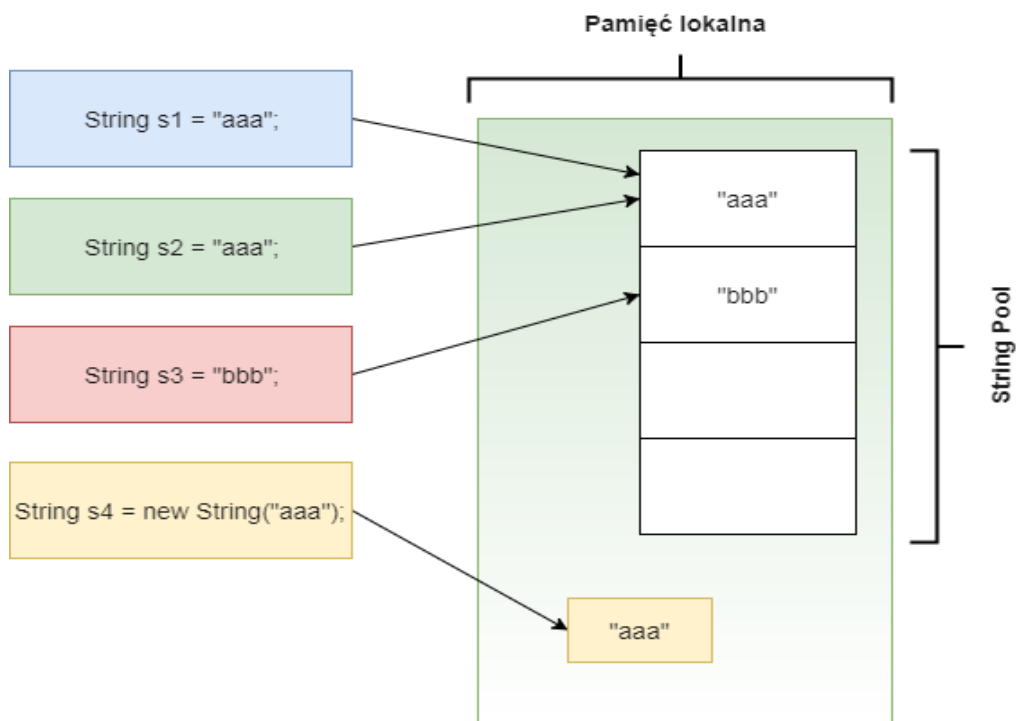
2. Dlaczego String jest niezmienny?

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 tę 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. Łądują tam wszystkie obiekty typu String, które zostały zainicjowane przy użyciu cudzysłowu – obiekty, które zostały zainicjowane przy pomocy słowa new łądują 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 String?

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(s3): " + 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(s3): 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 przechowywany. 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 słowa 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 tego 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. Słowo 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 jakkolwiek 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 słowa „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 więc 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 wydajność 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.

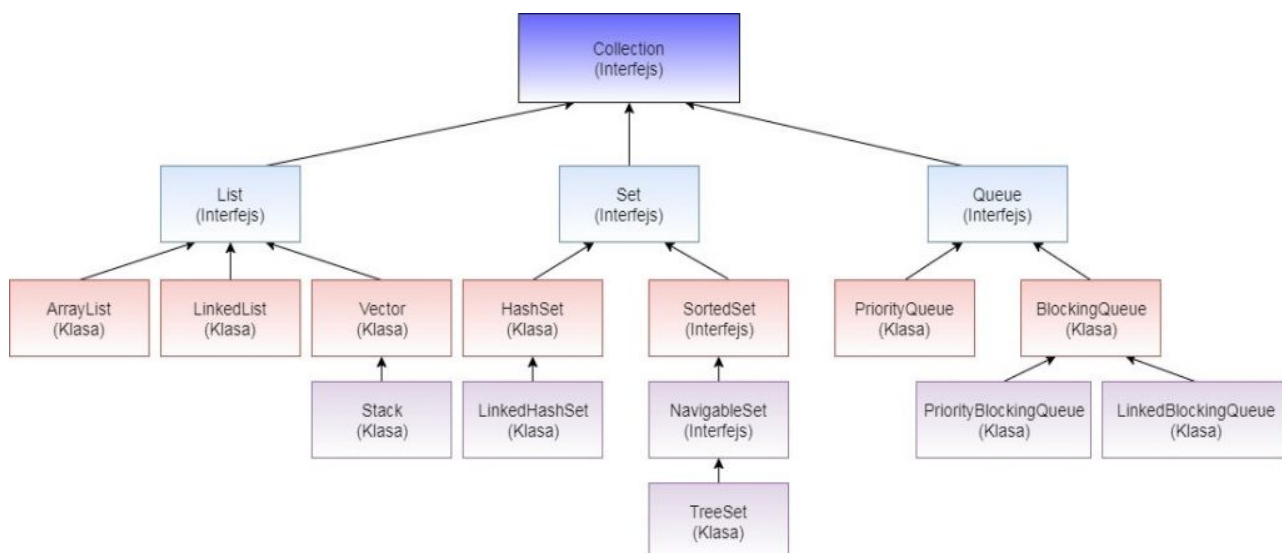
14. Jaka jest różnica pomiędzy Collection (Kolekcją) oraz Collections (Kolekcjami)?

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.

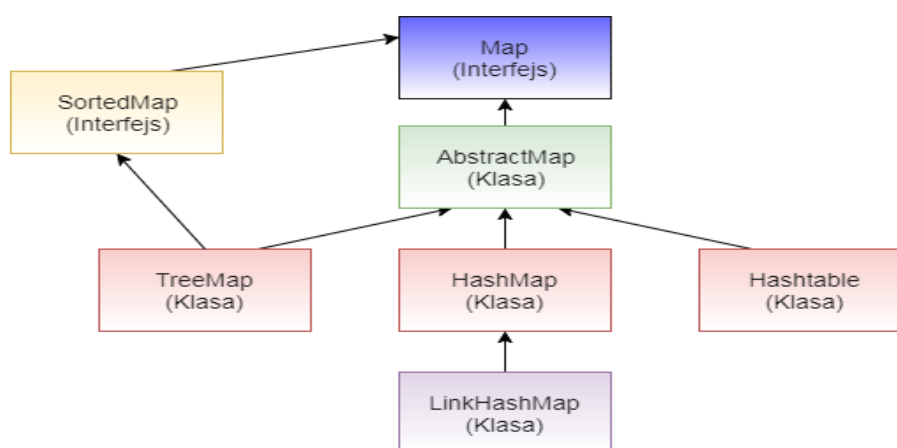
15. Jakie są najważniejsze główne interfejsy zawierające się we frameworku kolekcji?

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 interfejsu 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:

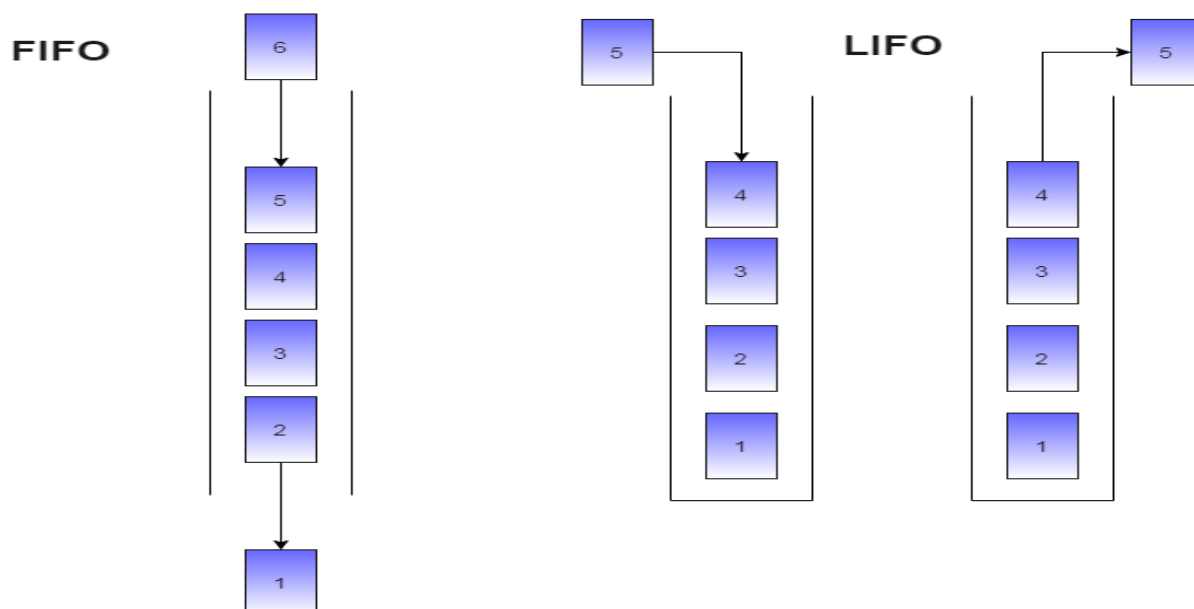
Mapy (Map) są to obiekty które mapują klucze do wartości. Mapa nie może posiadać dwóch takich samych kluczy – może natomiast posiadać dwie takie same wartości (pod warunkiem że mają różne klucze). Podstawowe implementacje interfejsu Map to HashMap, TreeMap oraz LinkedHashMap.

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)?

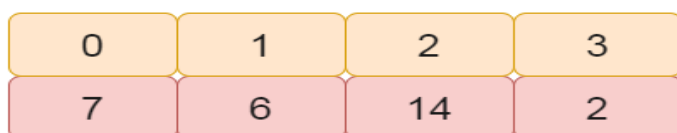
Raczej rzadko zadawane pytanie (z powodu rzadkiego użytkowania sterty) – aczkolwiek warto znać na nie odpowiedź. Otóż jak mówiliśmy wcześniej kolejki zazwyczaj korzystają z FIFO (first in, first out) natomiast sterty działają w oparciu o LIFO (last in, first out) tzn. że ostatni element który dostał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



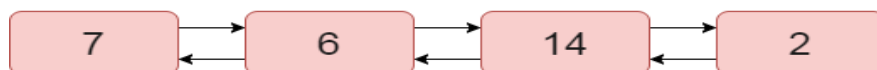
19. Jaka jest różnica między LinkedList i ArrayList?

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:

ArrayList



DoubleLinkedList



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 niesynchronizowane 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 przystosowany 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-wartość. 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 (kubelku) 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 `hashCode()` oraz `equals()`.

W sytuacji, 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 tę 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 tę 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 niewystąpienia 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 wartość.

- Klucze HashMap

Kluczem `hashmapy` może być każdy obiekt, który ma odpowiednio zaimplementowane metody `hashCode()` i `equals()`. Najlepiej, gdy obiekt klucza jest obiektem niezmiennym (`immutable`). W przypadku, gdy klucze mapy nie są niezmiennie, 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 niezmiennie (`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 `hashCode()` 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

```
1. import java.util.Objects;
2.
3. public final class MyKey {
4.     private final String myName;
5.     private final int myAge;
6.
7.     public MyKey(String myName, int myAge) {
8.         this.myName = myName;
9.         this.myAge = myAge;
10.    }
11.
12.    public String getMyName() {
13.        return myName;
14.    }
15.
16.    public int getMyAge() {
17.        return myAge;
18.    }
19.
20.    @Override
21.    public boolean equals(Object o) {
22.        if (this == o) {
23.            return true;
24.        }
25.        if (o == null || getClass() != o.getClass()) {
26.            return false;
27.        }
28.        MyKey myKey = (MyKey) o;
29.        return myAge == myKey.myAge &&
30.            Objects.equals(myName, myKey.myName);
31.    }
32.
33.    @Override
34.    public int hashCode() {
35.        return Objects.hash(myName, myAge);
36.    }
37. }
```

użycie:

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

zmienione, więc klasa spełnia warunki niezmienności. Zostały także zaimplementowane metody equals() i hashCode() (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.

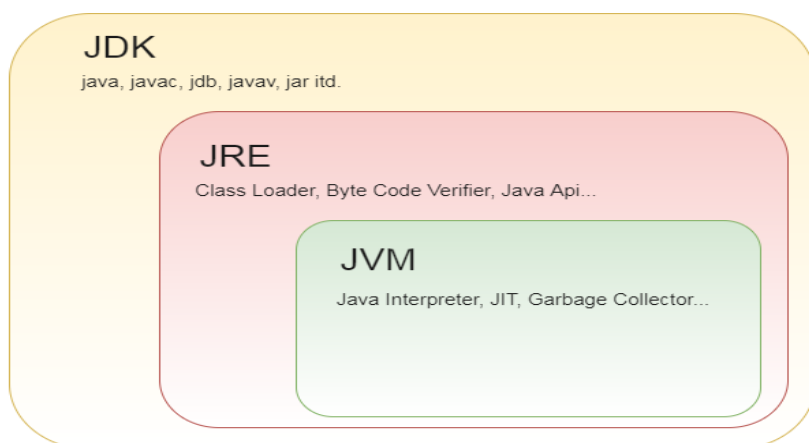
	Comparable	Comparator
Logika sortowania	Logika musi być zawarta w tej samej klasie którego obiekt ma być sortowany. Stąd może pojawiać się również nazwa naturalnego porządku(Natural Ordering)	Logika sortowania zawarta jest w osobnej klasie - dzięki czemu możemy pisać różne algorytmy sortowania w zależności 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 <i>compareTo(Object o1)</i> która porównuje ten obiekt z obiektem <i>o1</i> i zwraca odpowiadającą wartość. (>0 - ten obiekt jest większy niż <i>o1</i> , 0 - obiekty są równe, <0 ten obiekt jest mniejszy niż <i>o1</i>)	Metoda <i>compare(Object o1, Object o2)</i> która porównuje wybrane parametry obiektów <i>o1</i> i <i>o2</i> zwracając wartości odpowiadające ich porównaniu. (>0 - $o1 > o2$, 0 $o1 == o2$, <0 $o1 < o2$)
Jak użyć	<i>Collections.sort(List)</i> - obiekty będą posortowane według metody <i>CompareTo</i> .	<i>Collections.sort(List, Comparator)</i> - obiekty będą posortowane według metody <i>Compare</i> w klasie <i>Comparator</i> .

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?

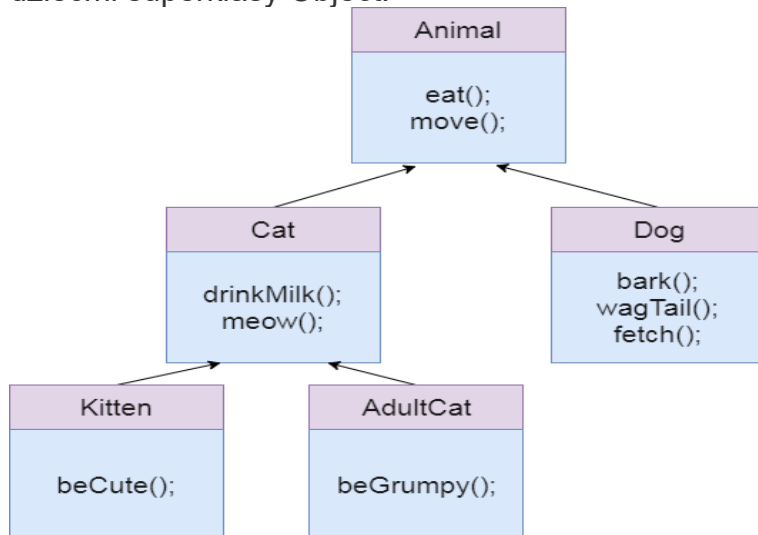
Klasa jest swojego rodzaju schematem na podstawie, którego powstają obiekty. W klasie definiujemy pola zawierające dane, oraz metody wykonywane przez obiekt.

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 słowa 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 literą. 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 typy 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 konwersję danych prymitywnych do klas opakowujących (np. int do Integer), 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:

- lambda Expressions - () -> {}
- method reference - Object::toString
- try-with-resource feature
- interface default methods
- annotations in many places: local variables, constructors 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 strumień 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 (usuwanie normalnym) GarbageCollectorwie, których obiektów zajmujących pamięć ma się pozbyć. Następnie, aby poprawić wydajność można również zastosować kompaktowanie pamięci, przesuując 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 więc 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 nieznanne.

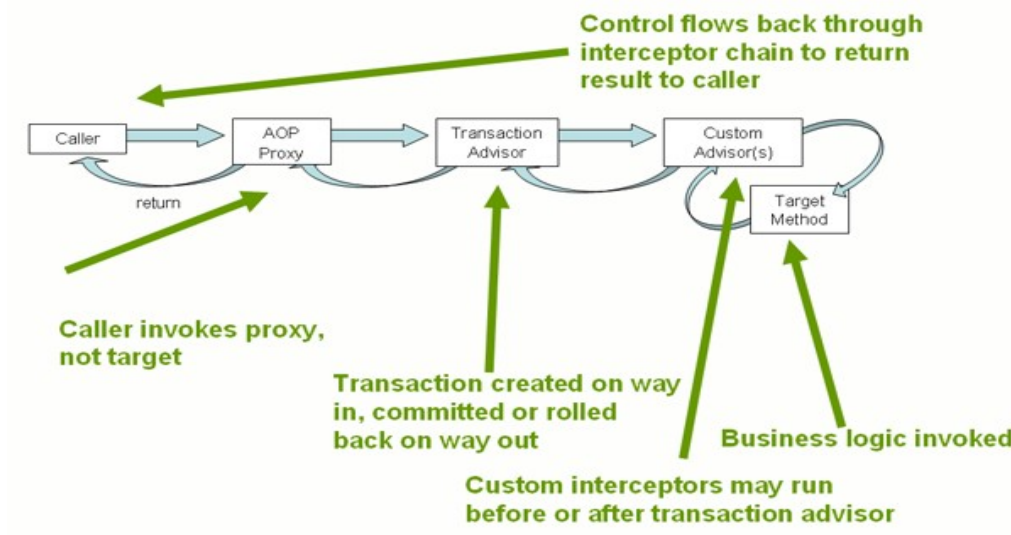
- 1. @Transactional w Springu

Transakcje w springu można uruchomić używając adnotacji @Transactional na metodzie lub klasie. Kiedy metoda oznaczona jako transakcyjna zostaje wywołana, Spring przechwytyje 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?

```
1 @Override
2 @Transactional
3 public void create(Task task) throws Exception {
4     em.persist(task);
5     // logika biznesowa która rzuca wyjątek
6     throw new Exception();
7 }
```

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 zajrzemy 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).

```
1 if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.rollbackOn(ex)) {  
2     try {  
3         txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());  
4     }  
5     ...  
6 }
```

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.

```
1 ---  
2 if (this.rollbackRules != null) {  
3     for (RollbackRuleAttribute rule : this.rollbackRules) {  
4         int depth = rule.getDepth(ex);  
5         if (depth >= 0 && depth < deepest) {  
6             deepest = depth;  
7             winner = rule;  
8         }  
9     }  
10 }  
11 ---
```

Używając adnotacji @Transactional możemy do niej przekazać różne parametry, między innymi parametr rollbackFor, w którym definiujemy 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.

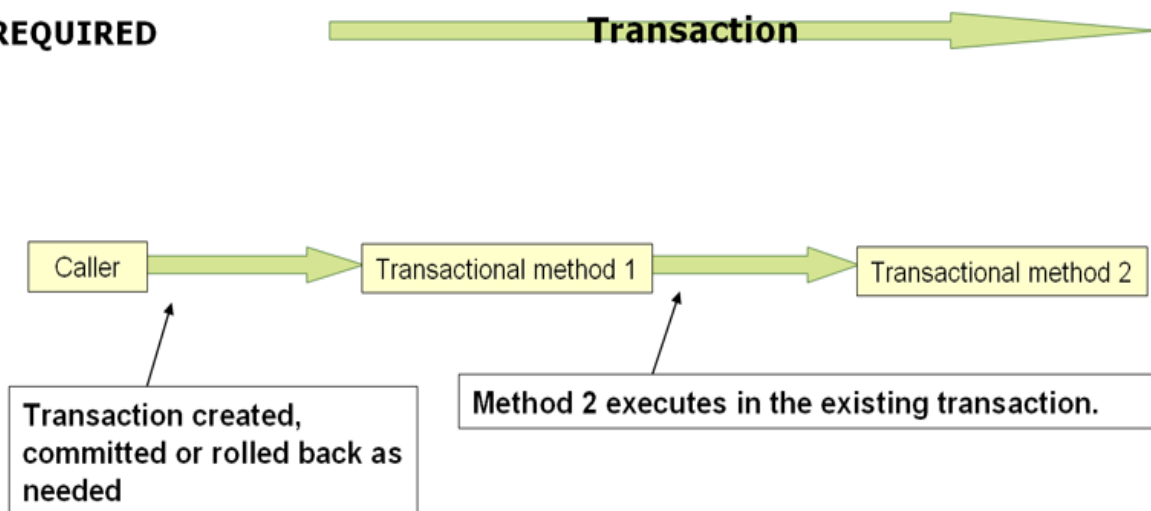
No to spójrzmy jakie typy propagacji są dostępne na adnotacji @Transactional

```
1 public enum Propagation {
2     REQUIRED(TransactionDefinition.PROPAGATION_REQUIRED),
3     SUPPORTS(TransactionDefinition.PROPAGATION_SUPPORTS),
4     MANDATORY(TransactionDefinition.PROPAGATION_MANDATORY),
5     REQUIRES_NEW(TransactionDefinition.PROPAGATION_REQUIRES_NEW),
6     NOT_SUPPORTED(TransactionDefinition.PROPAGATION_NOT_SUPPORTED),
7     NEVER(TransactionDefinition.PROPAGATION_NEVER),
8     NESTED(TransactionDefinition.PROPAGATION_NESTED);
9     ...
10 }
```

- 3.1. REQUIRED

Jest to 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.

REQUIRED



```
1 // nie stosuj w taki sposób
2 @Transactional(propagation = Propagation.REQUIRED)
3 public void requiredExample(String user) {
4     // ...
5 }
6
7 // preferowane użycie - Propagation.REQUIRED jest ustawiany domyślnie
8 @Transactional
9 public void requiredExample(String user) {
10    // ...
11 }
```

```
1 if (isExistingTransaction()) {
2     if (isValidExistingTransaction()) {
3         validateExistingAndThrowExceptionIfNotValid();
4     }
5     return existing;
6 }
7 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.

```
1 @Transactional(propagation = Propagation.SUPPORTS)
2 public void supportsExample(String user) {
3     // ...
4 }
```

Pseudokod:

```
1 if (isExistingTransaction()) {
2     if (isValidExistingTransaction()) {
3         validateExistingAndThrowExceptionIfNotValid();
4     }
5     return existing;
6 }
7 return emptyTransaction;
```

- 3.3. MANDATORY

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

```
1 @Transactional(propagation = Propagation.MANDATORY)
2 public void mandatoryExample(String user) {
3     // ...
4 }
```

Pseudokod:

```
1 if (isExistingTransaction()) {
2     if (isValidExistingTransaction()) {
3         validateExistingAndThrowExceptionIfNotValid();
4     }
5     return existing;
6 }
7 throw IllegalStateException;
```

- 3.4. REQUIRES_NEW

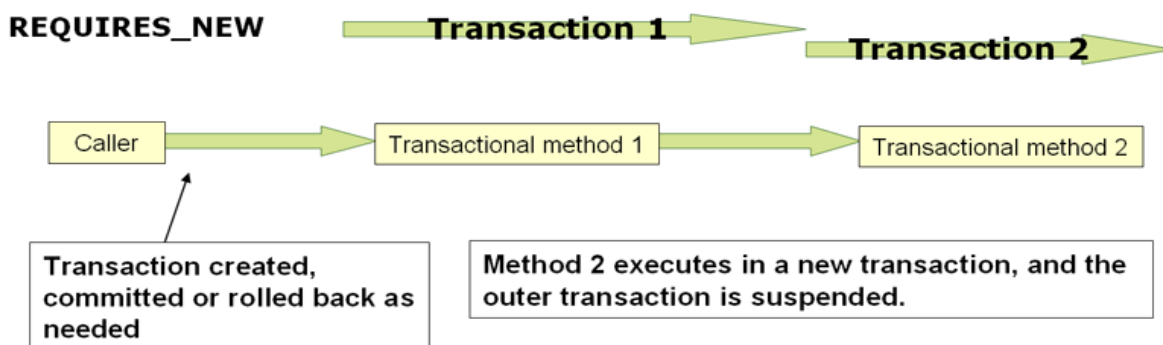
Zawsze powstaje nowa transakcja. W przypadku, gdy wcześniej jakaś transakcja była już otwarta, zostaje ona wstrzymana.

```
1 @Transactional(propagation = Propagation.REQUIRES_NEW)
2 public void requiresNewExample(String user) {
3     // ...
4 }
```

Pseudokod:

```
1 if (isExistingTransaction()) {
2     suspend(existing);
3     try {
4         return createNewTransaction();
5     } catch (exception) {
6         resumeAfterBeginException();
7         throw exception;
8     }
9 }
10 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.

```
1 @Transactional(propagation = Propagation.NOT_SUPPORTED)
2 public void notSupportedExample(String user) {
3     // ...
4 }
```

Pseudokod:

```
1 if (isExistingTransaction()) {
2     suspend(existing);
3 }
4 return continueWithBusinessLogic();
```


- 3.6. NEVER

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

```
1 @Transactional(propagation = Propagation.NEVER)
2 public void neverExample(String user) {
3     // ...
4 }
```

Pseudokod:

```
1 if (isExistingTransaction()) {
2     throw IllegalStateException;
3 }
4 return emptyTransaction;
```

- 3.7. NESTED

Sprawdzone 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.

```
1 @Transactional(propagation = Propagation.NESTED)
2 public void nestedExample(String user) {
3     // ...
4 }
```

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/>

Strumień 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 strumienia, 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 jakiegokolwiek operacji typu terminal dla strumienia spowoduje wyjątek `IllegalStateException`, 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 jedno 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. konkatencji).

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** („Employees 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. **Stream<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 „odповідź” ze strumienia danych. Redukcje są operacjami typu **terminal** – „redukuja” 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 Stream 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 pierwszym 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 **`null`**em 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 jakiegokolwiek 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 rzuty 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 wykorzystujemy 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 konsoli może po każdorazowym uruchomieniu się różnić z racji tego iż przydzielanie wątków z puli jest niedeterministyczne (o problemach z `parallel stream` przeczytasz <https://dzone.com/articles/think-twice-using-java-8>).

41. Java Rekord (Records) – Praktyczne wykorzystanie

W Javie 14 został wprowadzony obiekt typu `record`, zgodnie z JEP 359 . Nowy obiekt ten wnosi do języka wiele praktycznych funkcjonalności, które mogą znacznie ułatwić pracę programistom. `Record` w Javie 14 automatycznie generuje kod dla często używanych operacji, takich jak `hashCode`, `equals`, i `toString`. Dodatkowo, rekordy oferują prywatne i finalne pola, jednocześnie umożliwiając definiowanie publicznych pól, które są automatycznie generowane przez kompilator Javy.

Dzięki wprowadzeniu rekordów, programiści mogą znacząco skrócić i uprościć kod, zwłaszcza w przypadku klas reprezentujących dane, które często wymagają implementacji metod `equals`, `hashCode` i `toString`. Rekordy pozwalają na bardziej zwarte i czytelne rozwiązania, co przekłada się na zwiększenie produktywności i jakości kodu. Domyślnie posiadamy metody, umożliwiające odczyt pól, odpowiednik getterów.

Oczywiście, to dobre pytanie: jak wykorzystać rekordy w praktyce?

W poprzedniej części wpisu omówiłem koncepcję rekordów w Javie 14 i jakie korzyści niesie ze sobą ich wprowadzenie. Teraz skupmy się na praktycznym zastosowaniu

rekordów, zwłaszcza w kontekście architektury hexagonalnej.

W architekturze hexagonalnej (wpis o niej), często mamy do czynienia z obiektami, które reprezentują domenę problemową. Domena ta jest kluczowym elementem naszej aplikacji i często opiera się na danych, które są niezmiennie. W takich przypadkach obiekty typu record idealnie wpasowują się w nasze potrzeby. Więcej o architekturze w tym wpisie.

Dlaczego zatem rekordy są odpowiednie dla warstwy domeny w architekturze hexagonalnej? Przede wszystkim dlatego, że rekordy są klasami niemutowalnymi, co oznacza, że ich stan nie może być zmieniany po ich utworzeniu. Jest to kluczowe w przypadku reprezentowania danych domenowych, ponieważ chcemy uniknąć nieoczekiwanych zmian w tych danych.

Oczywiście, istnieją sytuacje, w których rekordy nie są odpowiednie, na przykład gdy jednym z pól rekordu jest obiekt lub kolekcja, ponieważ rekordy same w sobie nie zapewniają pełnej ochrony przed modyfikacją tych obiektów. Jednak dla prostych przypadków, w których obiekt rekordu zawiera jedynie typy takie jak long czy String, rekordy są idealnym rozwiązaniem.

Przykład implementacji

```
record User(Long userId, String firstName, String lastName, Integer age) {  
}
```

Na pierwszy rzut oka można zauważyć różnicę w deklaracji pól rekordu w porównaniu do klas. W przypadku rekordu, pola są zadeklarowane w nawiasach (), a nie w ciele klasy. Jednak to nie jedyna różnica.

Domyślnie, rekord już generuje konstruktor, który zawiera wszystkie pola zadeklarowane w rekordzie. Oznacza to, że nie musimy ręcznie definiować konstruktora ani implementować getterów dla pól rekordu. Rekord automatycznie dostarcza te funkcjonalności, co sprawia, że nasz kod jest bardziej zwężły i czytelny.

Konstruktor w rekordzie ?

W rekordach generowany jest tzw. kompaktowy konstruktor, który umożliwia inicjalizację wszystkich pól rekordu. Jest to wygodne i zwarte rozwiązanie, czasami możemy potrzebować przeprowadzić dodatkowe operacje w trakcie tworzenia obiektu. Dlatego w ramach rekordu możemy definiować własne konstruktory. Jednak istnieje pewne ograniczenie: pierwszą instrukcją w takim konstruktorze musi być wywołanie konstruktora kompaktowego rekordu.


```

record User(Long userId, String firstName, String lastName, Integer age) {

    public User(String firstName, String lastName, Integer age) {
        this(null, firstName, lastName, age); // Calling the compact constructor
        // Now we can add custom operations, e.g., validation
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}

```

W powyższym przykładzie mamy rekord "User", który ma kompaktowy konstruktor, ale również definiujemy własny konstruktor. Pierwszą instrukcją w własnym konstruktorze jest wywołanie konstruktora kompaktowego rekordu za pomocą `this(...)`. Dzięki temu możemy wykonać dodatkowe operacje, takie jak walidacja danych, przed utworzeniem obiektu rekordu. Jest to przydatne rozwiązanie, które daje nam elastyczność w projektowaniu naszych rekordów

Nadpisanie konstruktora kompaktowego

```

record User(Long userId, String firstName, String lastName, Integer age) {
    // The compact constructor will be automatically generated

    /**
     * Constructor for the User record that overrides the compact constructor.
     *
     * @param firstName First name of the user.
     * @param lastName Last name of the user.
     * @param age Age of the user.
     * @throws IllegalArgumentException If the age is negative.
     */
    public User(String firstName, String lastName, Integer age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}

```

W tym przypadku nadpisujemy kompaktowy konstruktor, a jedyną rzeczą, którą robimy, to sprawdzamy warunek dotyczący wieku i rzucając wyjątek, jeśli jest ujemny. Dzięki temu możemy skoncentrować się na walidacji lub innych operacjach bez potrzeby ponownego przypisywania wartości do pól rekordu. To przydatne podejście, które pozwala na bardziej czytelny i elastyczny kod.

Metody w rekordach

Oczywiście, możesz dodać metody do rekordu, które wykonują różne operacje na danych rekordu.

```
public record User(Long userId, String firstName, String lastName, Integer age) {

    public static final long ADULT_AGE = 18;

    /**
     * Constructor for the User record with age validation.
     *
     * @param firstName First name of the user.
     * @param lastName Last name of the user.
     * @param age Age of the user.
     * @throws IllegalArgumentException If the age is negative.
     */
    public User(String firstName, String lastName, Integer age) {
        this(null, firstName, lastName, age); // Calling the compact constructor
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }

    /**
     * Method to retrieve the full name of the user.
     *
     * @return Full name of the user.
     */
    public String getFullName() {
        return firstName + " " + lastName;
    }

    /**
     * Method to check if the user is an adult.
     *
     * @return True if the user is 18 years or older, false otherwise.
     */
    public boolean isAdult() {
        return age >= ADULT_AGE ;
    }

    /**
     * Method to compare two users based on their age.
     *
     */
}
```

```

    * @param other Another user to compare to.
    * @return 0 if the ages are the same, a negative number if this user is younger than the
    other, a positive number otherwise.
    */
    public int compareAge(User other) {
        return this.age.compareTo(other.age);
    }
}

```

W powyższym kodzie dodano trzy metody: `getFullName()`, która zwraca pełne imię i nazwisko użytkownika, `isAdult()`, która sprawdza, czy użytkownik jest pełnoletni, i `compareAge()`, która porównuje wiek dwóch użytkowników. Dzięki takim metodom możesz wykonywać bardziej złożone operacje na danych rekordu.

Możesz dodać stałą wartość do rekordu, ale nie jako pole, ponieważ pola rekordu są niezmiennicze (`final`) i muszą być zainicjowane w konstruktorze. Możesz jednak dodać pole statyczne do samej klasy, w której znajduje się rekord, jak to:

```
public static final long ADULT_AGE = 18;
```

Inicjalizacja i działanie na rekordach

```

public class Main {
    public static void main(String[] args) {
        // Initializing a record object
        User user1 = new User(1L, "John", "Doe", 25);

        // Using accessors
        System.out.println("User's first name: " + user1.firstName());
        System.out.println("User's last name: " + user1.lastName());
        System.out.println("User's age: " + user1.age());

        // Using methods
        System.out.println("Full name of the user: " + user1.getFullName());
        System.out.println("Is the user an adult? " + user1.isAdult());

        // Another object initialization
        User user2 = new User("Alice", "Smith", 17);

        System.out.println("\nUser's first name: " + user2.firstName());
        System.out.println("User's last name: " + user2.lastName());
        System.out.println("User's age: " + user2.age());
        System.out.println("Full name of the user: " + user2.getFullName());
        System.out.println("Is the user an adult? " + user2.isAdult());
    }
}

```

W tym przykładzie inicjalizujemy dwa obiekty rekordu "User", a następnie wykorzystujemy metody pól dostępowych, takie jak `firstName()`, `lastName()`, i `age()` które są domyślne dla

Recordów, do pobrania danych użytkowników. Dodatkowo, korzystamy z bardziej naszych metod, takich jak `getFullName()` i `isAdult()`, które wykonują różne operacje na danych rekordu. To pokazuje, jak łatwo i wygodnie można zarządzać danymi w rekordach i wykorzystywać ich metody.

Podsumowanie

Rekordy w Javie są specjalnym rodzajem klas wprowadzonym w Java 14 (JEP 359), które służą do reprezentacji danych w sposób zwięzły i niezmienny. Głównym celem rekordów jest ułatwienie tworzenia klas, które przechowują dane i zapewniają dostęp do tych danych za pomocą automatycznie generowanych metod. Oto kilka kluczowych cech rekordów:

Kompaktowy konstruktor: Rekordy automatycznie generują konstruktor, który umożliwia inicjalizację wszystkich pól rekordu. To znacząco upraszcza tworzenie obiektów rekordu.

Pola dostępne: Rekordy automatycznie generują metody dostępne (getter) do pól, które pozwalają na odczyt wartości pól. Pola rekordu są zazwyczaj oznaczone jako `final`, co oznacza, że nie można ich zmieniać po utworzeniu obiektu rekordu.

Niezmiennosc: Rekordy są klasami niemutowalnymi, co oznacza, że po utworzeniu obiektu rekordu nie można zmieniać jego stanu. Jeśli chcesz utworzyć nowy obiekt rekordu z innymi danymi, musisz stworzyć nowy obiekt.

Automatyczna metoda `equals()`: Rekordy automatycznie generują metodę `equals()`, która porównuje obiekty rekordu na podstawie ich pól. Dwa obiekty rekordu są uważane za równe, jeśli ich pola są równe.

Automatyczna metoda `hashCode()`: Rekordy generują także metodę `hashCode()`, która oblicza skrócony kod hash na podstawie pól rekordu. To ułatwia wykorzystanie obiektów rekordu w kolekcjach, takich jak `HashSet` czy `HashMap`.

Automatyczna metoda `toString()`: Rekordy generują domyślną implementację metody `toString()`, która zwraca reprezentację tekstową obiektu rekordu, opartą na jego polach.

Rekordy są przydatne do reprezentowania danych, takich jak modele domenowe, dane użytkowników itp. Dzięki nim kod staje się bardziej zwięzły, bardziej czytelny i mniej podatny na błędy.

41. Jak działa JIT? Czy ma jakieś wady?

Program napisany z wykorzystaniem języka Java jest kompilowany do kodu bajtowego, który jest zrozumiały dla maszyny wirtualnej Javy. JVM wykonując program interpretuje kolejne instrukcje kodu bajtowego konwertując je na natywne instrukcje maszyny, na której jest uruchomiona. Ponieważ interpretacja jest wolna w porównaniu do wykonywania kodu natywnego twórcy wprowadzili do środowiska uruchomieniowego JITa.

JIT (Just-In-Time compiler) jest komponentem maszyny wirtualnej Javy, który poprawia wydajność aplikacji stosując różnego rodzaju optymalizacje np. null check elimination, branch prediction, zagnieżdżanie metod (method inlining), eliminacja martwego kodu (dead code elimination), kompilacja do kodu natywnego i wiele innych.

Dla każdej z metod JVM zlicza ilość jej wywołań. Kiedy ilość wywołań danej metody osiągnie pewien próg (domyślnie 2000) metoda a w zasadzie jej bytecode jest przekazywany do kompilatora JIT, który jest odpowiedzialny za wykonanie na niej zestawu faz/optymalizacji, o których wspomniałem wcześniej w tym na końcu kompilacji do kodu natywnego. Skompilowany kod natywny jest zapisywany w przestrzeni procesu JVM w tzw. code cache.

JIT po starcie aplikacji zaczyna ją profilować analizując nasz kod i zbierając różnego rodzaju statystyki. Ma to wpływ na wydajność naszej aplikacji gdyż JVM przeznacza sporo zasobów na ten proces szczególnie na początku gdy aplikacja działa w trybie w pełni interpretowanym bez żadnych optymalizacji co może być jedną z wad JIT'a. Proces po starcie nazywamy "rozgrzewaniem" (warm up). Warto wziąć ten proces pod uwagę gdy badamy naszą aplikację pod kątem wydajnościowym i pamiętać, że wyniki mogą się różnić od tych, które zaobserwujemy na środowisku produkcyjnym.

42. Do czego służy adnotacja @PostConstruct?

Adnotacja @PostConstruct pochodzi z specyfikacji JSR 250 (Java Specification Request). Spring implementuje tą specyfikację tym samym zapewniając działanie tej adnotacji.

Adnotację @PostConstruct umieszczamy nad metodą klasy, która jest beanem tj. komponentem zarządzalnym przez Springa. Spring wywoła tą metodę jako callback po poprawnym zainicjalizowaniu naszego beana tj. po poprawnym wstrzyknięciu zależności dla tego komponentu. Wspomniany callback może być dobrym miejscem na różnego rodzaju post inicjalizację.

Kontrakt/Specyfikacja zapewnia, że ta metoda zostanie wywołana tylko raz w trakcie cyklu życia tego beana. Jeżeli chodzi o sygnaturę metody powinna być typu void, niestatyczna, bezparametrowa o dowolnym poziomie widoczności (public, protected, package private or private).

Warto zaznaczyć, że używanie @PostConstruct ma sens tylko wtedy gdy używamy wstrzykiwania typu field injection lub setter injection. Jeżeli używamy wstrzykiwania przez konstruktor czyli rekomendowanego przeze mnie podejścia inicjalizację możemy wykonać w konstruktorze. Dzięki temu łatwiej jest przetestować taki komponent, który możemy stworzyć poprawnie tylko i wyłącznie z wykorzystaniem konstruktora bez względu na to

czy robimy to w testach czy w trakcie stawiania całego kontekstu springowego bez bawienia się w mechanizmy refleksji.

W praktyce o adnotacji `@PostConstruct` powinniśmy zapomnieć na rzecz wstrzykiwania przez konstruktor ewentualnie zostawiając ją sobie jako alternatywę dla kodu legacy.

43. Wymień sposoby wstrzykiwania zależności. Jakiego sposobu używasz i dlaczego?

Wstrzykiwanie zależności (Dependency Injection) jest wzorcem projektowym, który polega na przekazywaniu do obiektu gotowych instancji obiektów (zależności), z których ten obiekt się składa tak aby zapewnić swoją funkcjonalność w przeciwieństwie do tworzenia ich samodzielnie. Podejście to ma wiele zalet m.in. obiekt, który dostaje zależności z zewnątrz (np. z kontenera springowego) nie jest uzależniony od konkretnej implementacji a więc łatwo można przekonfigurować aplikację lub podmienić implementację w testach.

Wyróżniamy następujące sposoby wstrzykiwania zależności:

field injection

setter injection

constructor injection

Preferuję ostatnie podejście czyli wstrzykiwanie przez konstruktor. Podejście to pozwala w łatwy sposób na skonfigurowanie obiektu pod testy bez wykorzystywania mechanizmów refleksji. Mamy również pewność, że obiekt stworzony przez new zawsze zwróci nam poprawny i spójny obiekt gotowy do działania w przeciwieństwie do field injection / setter injection gdzie łatwo jest się „nadsiać” na `NullPointerException`. Konstruktor pełni także funkcję dokumentacyjną klasy gdyż wiemy dokładnie co jest potrzebne aby poprawnie skonfigurować obiekt czego nie ma np. w przypadku setter injection gdzie musielibyśmy się zastanawiać jakich setterów wystarczy użyć aby obiekt był w spójnym stanie. Wadą używania setterów jest również to, że łamiemy zasadę hermetyzacji przez co łatwo w niekontrolowany sposób popsuć stan obiektu.

Kolejnym argumentem na rzecz constructor injection jest to, że mamy kontrolę nad ilością zależności. Przykładowo dodanie 5 zależności do konstruktora już może zniechęcać - szybko widzimy jak ich liczba się rozrasta więc możemy dzięki temu szybciej podjąć decyzję w stronę refactoringu i stworzeniu osobnej klasy odpowiedzialnej za daną podfunkcjonalność. W przypadku field injection dodanie kolejnej zależności jest banalnie proste i przyjemne, liczba zależności szybko rośnie przez co klasa może z czasem być trudna w utrzymaniu.

44. Co to jest hermetyzacja?

Hermetyzacja (lub enkapsulacja) polega na ukrywaniu implementacji czyli składowych lub metod obiektu, z których się składa. Hermetyzacja nadaje obiektowi charakter czarnej skrzynki co jest kluczowe dla koncepcji wielokrotnego użycia kodu jak i jego niezawodności. Oznacza to, że sposób przechowywania danych w klasie może się diametralnie zmienić, ale dopóki udostępnia ona te same metody do manipulacji danymi żaden obiekt nie zostanie tym dotknięty. Jeżeli stan obiektu zmieni się mimo, że nie wywołano na jego rzecz żadnej metody oznacza to, że została złamana zasada hermetyzacji.

45. Jakie rzeczy weszły od Java 11?

Pozwolę sobie wymienić tylko niektóre nowe funkcjonalności. Po komplet odsyłam do tego artykułu.

nowe metody w klasie String: `isBlank`, `lines`, `strip`, `stripLeading`, `stripTrailing`, `repeat`

nowe statyczne metody w klasie Files: `readString`, `writeString`

nowy wydajniejszy HTTP client z wsparciem HTTP/1.1, HTTP/1.2, Web Socket

7. Jak napisałbyś metodę kontrolera do usunięcia wielu użytkowników?

Jednym ze sposobów na rozwiązanie tego zadania może być przekazanie listy identyfikatorów do usunięcia w następujący sposób:

```
@DeleteMapping("/users/{userIds}")
public void deleteUsers(@PathVariable List<Long> userIds) {...}
```

46. Mikroserwisy vs Monolity

Mikroserwisy są sposobem na projektowanie aplikacji jako niezależnie wdrażanych usług w przeciwieństwie do monolitów, których wszystkie funkcjonalności są wdrażane i uruchamiane w jednym procesie aplikacyjnym.

Pojedynczy mikroserwis jest zazwyczaj odpowiedzialny za jakąś pojedynczą funkcjonalność/obszar, który trzeba dostarczyć dla biznesu.

W monolicie realizacja konkretnego przypadku biznesowego jest wykonywana przez zbiór obiektów/serwisów komunikujących się pomiędzy sobą, żyjących w obrębie tego samego procesu. Natomiast w architekturze mikroserwisowej na jeden proces biznesowy może być zaangażowanych wiele mikroserwisów komunikujących się pomiędzy sobą używając takich technologii jak np. HTTP REST, JMS, gRPC.

Architektura w oparciu o mikroserwisy ma wiele zalet szczególnie w przypadku dużych złożonych systemów m.in.:

skalowalność

Poszczególne komponenty mogą być skalowalne niezależnie od pozostałych. Jest to niesamowita zaleta. Możemy mieć komponent, który intensywnie wykorzystuje CPU np. jest odpowiedzialny za przetwarzanie obrazów w związku z tym wdrażamy taki mikroserwis na instancje zoptymalizowane pod kątem CPU np. Amazon EC2 Compute Optimized Instances. Inny moduł może być bazą danych in-memory więc wdrożymy go na instancje zoptymalizowane pod pamięć np. Amazon EC2 Memory Optimized instances. W przypadku monolitu nie mamy takich możliwości, musimy pójść na kompromis jeżeli chodzi o hardware. W miarę jak monolit rośnie musimy skalować wertykalnie czyli dokładać zasobów sprzętowych tj. CPU, RAM a jeżeli chcemy aby aplikacja obsłużyła większy ruch skalując horyzontalnie skalujemy cały duży monolit mimo, że np. część usług jest wykorzystywana rzadko i tego nie wymaga co jest jednocześnie trudne i kosztowne.

niezawodność

Kolejny problem w przypadku monolitu jest taki, że wszystkie moduły naszej aplikacji są

częścią jednego procesu aplikacji tzn., że problem powstały w jednym module np. wyciek pamięci może wpłynąć na działanie całego procesu co w efekcie może skutkować położeniem całej aplikacji i niedostępnością całego systemu. Mikroserwisy nie mają tego problemu ze względu na to, że są to niezależnie wdrażane komponenty uruchamiane jako niezależne procesy w systemie operacyjnym.

dowolność technologii (polyglot)

Każdy element może być tworzony w technologii najlepiej dostosowanej do funkcjonalności jaką ma realizować zgodnie z "use the right tool for the right job" np. z wykorzystaniem technologii Java i Spring wystawimy usługi restowe, Pythona użyjemy do Machine Learning a C/C++ do napisania aplikacji typu video-streaming.

modularność

Małe komponenty pozwalają na łatwe wdrożenie i rozbudowę bez konieczności ingerencji w całą aplikację. W monolicie jedna mała zmiana mogła powodować redeploy całego systemu, który jak możemy się spodziewać był długim i żmudnym procesem budowania, wykonywania się testów jednostkowych i integracyjnych oraz innych kroków co mogło skutkować także niedostępnością całego systemu przez jakiś czas. W mikroserwisach małe zmiany są prostsze do zrealizowania i wdrożenia. Dodatkowo łatwo podzielić takie zmiany pomiędzy zespołami developerskimi w przeciwieństwie do monolitu gdzie praca wielu osób nad jednym olbrzymim komponentem jest ciężka i trudna w utrzymaniu.

Jak ze wszystkim oczywiście architektura w oparciu o mikroserwisy ma również wady i należy przemyśleć czy będzie miała zastosowanie w naszym systemie m.in:

złożoność

Ze względu na to, że aplikacja składa się z wielu serwisów komunikujących się między sobą musimy zapewnić aby komunikacja pomiędzy nimi zachodziła w bezpieczny sposób dodatkowo dochodzi nam również obsługa różnych błędów głównie sieciowych związanych z naszą infrastrukturą. W przypadku monolitu nie mieliśmy tego problemu ponieważ wszystkie funkcjonalności były w środku więc na wywołanie lokalne metody nie musieliśmy się specjalnie zabezpieczać.

Testowanie w środowisku rozproszonym, zarządzanie infrastrukturą również jest wyzwaniem i narzutem na złożoność.

zapewnienie spójności danych

W przypadku monolitu zapewnienie spójności było relatywnie proste gdyż mieliśmy lokalną transakcję ACID, która zapewniała nam spójność i atomowość na poziomie bazy danych. W przypadku architektury mikroserwisowej gdzie mamy do czynienia z systemem rozproszonym nie jest tak łatwo i w zależności od potrzeb musimy posiłkować się wzorcami typu Saga Pattern i Eventual Consistency.

47. Co wiesz o SOLID, IoC, Dependency Injection?

- Literka "S": [video](#)
- Literka "O": [video](#)
- Literka "L": [video](#)
- Literka "I": [video](#)
- Literka "D": [video](#)

SOLID opisuje pięć podstawowych zasad projektowania w programowaniu obiektowym.

S - Single Responsibility Principle

Każda klasa powinna być skoncentrowana tylko na jednym zadaniu, mieć tylko jeden powód do zmiany.

O - Open/Closed Principle

Klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje. Przy nowych wymaganiach kod nie powinien być modyfikowany ale dodawany nowy, który rozszerza funkcjonalność.

L - Liskov Substitution Principle

Klasa pochodna powinna być możliwa do użycia w miejsce klasy nadrzędnej zachowując się w taki sam sposób bez modyfikacji. Klasa pochodna nie ma wpływu na zachowanie się klasy nadrzędnej.

I - Interface Segregation Principle

Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.

D - Dependency Inversion Principle

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji. Kod powinien zależeć od rzeczy „stabilnych” tj. abstrakcji, które się nie zmieniają.

IoC (Inversion of Control) jest wzorcem architektury polegającym na odwróceniu sterowania programem i zrzuconiu tego na framework np. Spring. Implementacją tego wzorca może być np. Dependency Injection lub Aspect Oriented Programming.

48. Dlaczego BigDecimal jest lepszy od double do przechowywania wartości pieniężnych? Dlaczego double nie jest precyzyjny?

Typ double nie jest precyzyjny w kontekście obliczeń finansowych ponieważ obliczenia wykonywane są w systemie binarnym zgodnie z standardem IEEE 754 w przeciwieństwie do BigDecimal, którego obliczenia działają na systemie dziesiętnym. W związku z tym nie jesteśmy w stanie wyrazić dokładnie pewnych wartości w systemie binarnym a jedynie pewną aproksymację tej wartości np. 0.09999999999999998 zamiast 0.1. Jeżeli chodzi o wartości pieniężne chcemy podczas obliczeń dokładnie odwzorować rzeczywistość tj. nie chcemy sytuacji, w której np. na koncie bankowym klienta jest kwota mniejsza niż to by wynikało z dokonanych transakcji. Z pomocą przychodzi nam typ BigDecimal w języku Java, który pozwala nam kontrolować precyzję i zapewnić dokładność obliczeń.

49. Do czego służy BindingResult w Springu?

BindingResult jest interfejsem, który definiuje w jaki sposób zbieramy wyniki walidacji obiektu, który walidujemy. Domyślnie spring rzuci wyjątkiem jeżeli dane, które walidujemy nie są poprawne. Dzięki BindingResult możemy mieć dostęp do wyniku walidacji przeprowadzonej przez framework i ręcznie zdecydować co dalej np:

```
@PostMapping("/register")
```

```

public ResponseEntity<?> register(@Valid @RequestBody RegisterRequest request,
BindingResult, result) {
if (result.hasErrors()) {
return ResponseEntity.badRequest().body(result.toString());
}
registerService.register(request);
return ResponseEntity.ok().build();
}

```

50. Wymień poziomy izolacji transakcji i wyjaśnij jak działa jeden z nich?

<https://www.baeldung.com/spring-transactional-propagation-isolation>

Wyróżniamy następujące poziomy izolacji transakcji:

READ_UNCOMMITTED
 READ_COMMITTED
 REPEATABLE_READ
 SERIALIZABLE

READ_COMMITTED - transakcja z ustawionym takim trybem izolacji będzie „widziała” tylko zacommitowane zmiany co oznacza, że ponowne odczyty mogą zwracać różne wyniki z uwagi na inne transakcje, które wykonują się w tym samym czasie. Aby zapewnić idempotentność na poziomie odczytu (pojedynczego rekordu) transakcji tj. aby każdy kolejny odczyt zwracał ten sam wynik możemy zwiększyć poziom izolacji na **REPEATABLE_READ**. Warto zaznaczyć, że mimo ustawienia **REPEATABLE_READ** może wystąpić tzw. phantom read, w którym pomiędzy odczytami w jednej transakcji inne transakcje dodają/usuwają rekordy w tym samym czasie co prowadzi do różnych rezultatów na poziomie odczytu. Jeżeli chcemy uodpornić się na phantom read możemy ustawić jeszcze bardziej restrykcyjny poziom izolacji tj. **SERIALIZABLE** jednak z pewnością wpłynie to negatywnie na wydajność.

51. Jakie właściwości ma adnotacja @Transactional? Co można ustawić?

Adnotacja @Transactional posiada m.in. następujące właściwości:

isolation

poziom izolacji transakcji; domyślnie przyjmuje wartość Isolation.DEFAULT czyli poziom izolacji będzie domyślny, zdeterminowany przez bazę danych, która jest „pod maską”

rollbackFor

określa występowanie jakich wyjątków powinno powodować wycofanie transakcji; domyślnie transakcja zostanie zrollbackowana gdy wystąpi jeden z wyjątków typu Error lub RuntimeException, checked exceptions domyślnie nie powodują wycofania transakcji

noRollbackFor

określa występowanie jakich wyjątków nie powinno powodować wycofania transakcji

propagation

typ propagacji domyślnie Propagation.REQUIRED

timeout

timeout dla transakcji w sekundach, domyślnie timeout po stronie aplikacji jest wyłączony

readOnly

flaga określająca czy transakcja jest tylko do odczytu

transactionManager

nazwa beana odpowiedzialnego za zarządzanie transakcjami (możemy mieć wiele transaction managerów w jednej aplikacji)

52. Jak walidować przychodzące requesty REST w Springu? Jaki błąd wystąpi gdy podamy stringa zamiast inta?

W aplikacji spring boot'owej gdy używamy zależności spring-boot-starter-web domyślnym formatem serializacji/deserializacji modelu transportowego jest json a biblioteka do tego to jackson. Gdy podamy łańcuch znaków zamiast typu integer powstanie błąd formatu czyli tak naprawdę błąd deserializacji. Jackson zwróci błąd parsowania, objawi się to wyjątkiem typu `InvalidFormatException`.

Walidację modelu transportowego możemy wykonać na różne sposoby. Warto zaznaczyć, że walidację możemy podzielić na 3 poziomy:

walidacja formatu

Sprawdzamy czy komunikat poprawnie się parsuje. Zwykle nie musimy ingerować w tą można powiedzieć niskopoziomą walidację. Biblioteka, której używamy powinna być za to odpowiedzialna tj. w przykładzie wyżej.

walidacja struktury

Sprawdzamy czy już przeparsowany komunikat jest zgodny z naszymi oczekiwaniami np. czy adres email jest poprawny, hasło jest w odpowiednim formacie itd. Nie bierzemy pod uwagę aktualnego stanu systemu.

walidacja spójności

Sprawdzamy czy komunikat/komenda może być poprawnie wykonana biorąc pod uwagę aktualny stan systemu. Przykładowo nie możemy zarejestrować 2x użytkownika na ten sam email.

Jeżeli chodzi o walidację struktury możemy posłużyć się biblioteką hibernate-validator (implementacja standardu JSR 380). W Spring boot aby dodać wsparcie dla tego typu walidacji możemy wykorzystać spring-boot-starter-validation. Spring wspiera adnotację `@NotNull`, `@NotBlank`, `@Min`, `@Max` itd. w związku z tym obok request body w kontrolerach restowych możemy użyć adnotacji `@Valid` aby dany model był automatycznie zwalidowany.

Możemy również ręcznie walidować komunikaty w odpowiednio przygotowanych do tego klasach jeżeli np. mamy jakieś specyficzne wymagania lub po prostu chcemy mieć pełną kontrolę i preferujemy taki sposób na walidację.

53. Jak byś zabezpieczył api restowe?

Api restowe możemy zabezpieczyć na wiele sposobów. Jednym z nich może być autoryzacja z wykorzystaniem tokena JWT (JSON Web Token). Natomiast cała komunikacja powinna być zabezpieczona z wykorzystaniem protokołu TLS.

54. Jak przechowywać hasła w bazie danych? Jak działa Bcrypt?

Rekomendowanym podejściem do bezpiecznego przechowywania haseł w bazie danych jest użycie jednokierunkowej funkcji hashującej z solą. Jednym z algorytmów implementujących to podejście jest bcrypt. Bcrypt jest funkcją hashującą dedykowaną do

przechowywania haseł w bezpieczny sposób opartą o szyfr Blowish. Bcrypt na podstawie hasła lub innego dowolnego ciągu zwróci nam 60 znakowy hash hasła. Budowa wyjściowego skrótu wygląda następująco:

`$2[wersja]${koszt}${sól}[hash]`

gdzie:

wersja - identyfikator algorytmu, domyślnie 'a' czyli bcrypt

koszt - liczba z przedziału 04-99 określająca tzw. work factor algorytmu, domyślnie 12 czyli $2^{12} = 4096$ rund, każde zwiększenie współczynnika o jeden zwiększa dwukrotnie czas obliczeń

sól - losowy ciąg o wielkości 22 znaków

hash - hash o wielkości 31 znaków

Przykład hasha dla hasła 'java':

`$2a$12$1gUpfSyUB.OaegFGSe/3rOrLiuZCGauzK2nAqVWaBCA7vRWfLAC2a`

Bcrypt jest uznawany za skuteczny i bezpieczny algorytm. Sól pełni rolę mitygacji przed atakami z wykorzystaniem tęczowych tablic a odpowiednio ustawiony „work factor” algorytmu powoduje, że czas łamania hasła znacząco się wydłuża z uwagi na złożoność obliczeniową, która rośnie wykładniczo w związku z tym łamanie metodą siłową przestaje się opłacać.

Więcej na temat bezpieczeństwa haseł możemy przeczytać tutaj.

55. Jakie znasz zalety / wady stream api w Java?

Zalety:

- zwiezłość i czytelność szczególnie w połączeniu z referencjami do metod
- elastyczność, możemy bardzo łatwo zmieniać kolejność kroków w algorytmie
- dzięki `parallelStream()` możemy w prosty sposób zrównoleglić operacje na danych

Wady:

- utrudnia debugowanie, chociaż IDE coraz lepiej sobie z tym radzą
- wyrażenia (lambda) w `map/filter` itd. nie mogą rzucać tzw. checked exceptions co może być problematyczne np. jak przetwarzamy I/O

56. Jak działa protokół HTTP? Jakie są metody?

HTTP (Hypertext Transfer Protocol) jest protokołem warstwy aplikacji, który definiuje komunikację pomiędzy przeglądarką a serwerem HTTP. W tym modelu serwer HTTP jest odpowiedzialny za obsługę żądań pochodzących od klienta - przeważnie przeglądarki internetowej. Klient jest inicjatorem połączenia a więc wszelkie zasoby, skrypty, obrazki są przesyłane przez serwer HTTP do klienta w ramach odpowiedzi na żądanie.

Wspomniałem o tym, że klientem jest przeważnie przeglądarka internetowa natomiast w praktyce mogą to być również inne aplikacje działające w imieniu użytkownika np. urządzenie mobilne albo inny system, który się z nami integruje. Rodzaj „narzędzia”, które komunikuje się z serwerem jest zdefiniowany przez nagłówek User-Agent.

Warto wspomnieć, że protokół HTTP jest protokołem bezstanowym co oznacza, że każde żądanie do serwera jest interpretowane oddzielnie i nie ma pomiędzy nimi logicznego powiązania. Połączenie TCP, na bazie którego opiera się ten protokół zwykle jest zamykane i nie jest utrzymywane chyba, że ze względów wydajnościowych chcemy utrzymać połączenie możemy „powiedzieć” o tym serwerowi używając nagłówka Connection: keep-alive (HTTP Persistent Connection).

Komunikaty (żądania i odpowiedzi) w protokole HTTP zbudowane są z nagłówków czyli metadanych oraz payload'u (lub body), który zawiera odpowiedź, o którą klient żądał, może to być np. strona www. Każde żądanie zawiera również tzw. metodę HTTP, która definiuje akcję jaką chcemy wykonać w ramach danego zasobu.

Wyróżniamy następujące podstawowe metody HTTP:

GET - pobranie lub odczytanie zasobu, żądanie tego typu nie powinno zmieniać stanu na serwerze w kontekście danego zasobu.

HEAD - żądanie podobne do GET z takim wyjątkiem, że nie zwraca body. Użyteczne gdy chcemy sprawdzić tylko metadane/nagłówki zasobu bez narzutu na transfer np. możemy sprawdzić czy dany plik istnieje i jaki ma rozmiar bez jego pobierania.

POST - wysłanie danych na serwer, zwykle gdy wypełniamy jakiś formularz korzystamy z tej metody aby przesłać wypełnione dane na serwer po czym serwer tworzy nowy zasób.

PUT - wysyłanie danych na serwer w celu aktualizacji istniejącego zasobu.

DELETE - usunięcie istniejącego zasobu.

PATCH - zmiana jakiejś części istniejącego zasobu np. zmiana hasła użytkownika.

Więcej o protokole HTTP można przeczytać tutaj.

57. Jak wygląda model pamięci w Java? Jakie znasz Garbage Collectory? Jakie mają zastosowanie?

Maszyna wirtualna Javy jest zwykłym procesem w systemie operacyjnym, który tj. każdy inny proces potrzebuje pamięci do funkcjonowania. Aby programy napisane w Java funkcjonowały w poprawny sposób wirtualna maszyna dzieli pamięć na różne obszary, które mają odpowiednie zastosowanie:

heap (sterta)

Pamięć wykorzystywana do przechowywania obiektów, które tworzymy w naszej aplikacji. Utworzone obiekty żyją dopóki ich używamy tj. mamy wskazanie do nich przez referencje. Kiedy na dany obiekt nie wskazuje już żadna referencja obiekt jest niszczone przez algorytm Garbage Collectora, który jest odpowiedzialny za zwalnianie pamięci przez obiekty, które już nie są używane.

Heap również możemy podzielić na 2 obszary tzw. **Young Generation** (który dzieli się również na Eden, Survivor0 i Survivor1) i **Old Generation**. W Java istnieje tzw. hipoteza

generacyjna, która mówi, że tzw. „młode obiekty” żyją krócej niż stare tj. istnieje większe prawdopodobieństwo, że obiekt, który przeżył kilka lub kilkanaście cykli GC będzie żył dalej niż obiekt, który dopiero co został stworzony. Dlatego też obiekty, które „swoje przeżyły” przerzucane są do Old Generation. Podział ten pozwala na dostosowanie odpowiednich algorytmów Garbage Collectora do danego obszaru pamięci np. dla przestrzeni Young Generation będziemy chcieli częściej uruchomić skanowanie GC niż dla Old Generation.

non-heap

Natywna pamięć maszyny wirtualnej. W tej przestrzeni zapisywane są obiekty potrzebne maszynie wirtualnej do poprawnego funkcjonowania. Wyróżniamy m.in. takie obszary jak:

- **metaspace**

Zawiera m.in. kod naszej aplikacji, constant poole, dane pól i metod

- **code cache**

Przestrzeń m.in. do zapamiętywania kodu natywnego przez JITa, o którym wspomniałem wcześniej

- **stos dla wątków**

Garbage collector w JVM używane są do odśmiecania pamięci czyli zwalniania nieużywanych obiektów. Dzięki temu nie musimy ręcznie zarządzać pamięcią tj. w językach C/C++, możemy zrzucić tę odpowiedzialność na maszynę wirtualną w tym przypadku na konkretny algorytm Garbage Collectora. Wyróżniamy m.in. następujące algorytmy GC:

Serial

Parallel

Concurrent Mark Sweep

G1

Domyślnie od Javy 9 używany jest algorytm G1. Mamy możliwość zmiany algorytmu przekazując odpowiednie parametry do JVM podczas uruchamiania aplikacji np:

```
java -XX:+UseParallelGC -jar Application.java
```

58. Jakie znasz typy ataków webowych? Wymień i objaśnij jeden z nich.

Możemy wyróżnić m.in. takie ataki jak XSS (Cross-site scripting), SQLI (SQL Injection), Path Traversal, DDoS (Distributed Denial of Service), SSRF (Server-Side Request Forgery).

Atak SQLI polega na zmanipulowaniu wejścia (np. nazwy użytkownika lub frazy wyszukiwania) w taki sposób aby silnik bazodanowy podczas parsowania tego zapytania zinterpretował je inaczej niż twórca aplikacji założył tj. zmieniając to zapytanie. Atakujący mając kontrolę nad zapytaniem zyskuje dostęp do danych w bazie danych, do których nie powinien mieć dostępu.

Aby nasza aplikacja nie była podatna na ten typ ataku powinniśmy zastosować tzw. prepared statement. Baza danych wtedy parsuje samo zapytanie oddzielnie kolejno wstrzykując podane wartości w odpowiednie miejsca w zapytaniu co nie tylko poprawia bezpieczeństwo ale też wydajność gdyż zapytanie nie musi być wielokrotnie parsowane

przez silnik bazodanowy.

59. Czym jest niezmiennosc (immutability) obiektów?

Dziś skupimy się na niezmienności, niezmiennalności, niemutowalności (i jeszcze pewnie kilka innych przymiotników z nie), ehh wolę po prostu immutability obiektów. Trend ten jest dosyć dobrze znany programistom języków funkcyjnych, ale wraz z upływem czasu podobną technikę stosuje się w coraz większej ilości języków.

Co to znaczy, że obiekt jest immutable?

Niezmienny obiekt to taki, który raz zainicjalizowany nie zmieni swojego stanu. O co dokładnie chodzi? Mamy gwarancję, że wszystkie dane (pola) w konkretnym obiekcie nie ulegną zmianie. Żeby zmodyfikować jakieś wartości, trzeba utworzyć całkowicie nowy obiekt.

Czy znasz jakiś niemutowalny obiekt?

W większości języków programowania idealnym przykładem obiektu, który nie zmienia się w czasie jest String. Tak, tak, za każdym razem kiedy używamy operatora += albo dodajemy jakieś znaki, tak na prawdę w środku tworzony jest całkowicie nowy obiekt. Dlatego też, jeśli budujesz nowy obiekt typu String z kilku kawałków (np. ładując coś z pliku) warto używać StringBuildera.

Po co tego używać?

Zakładam, że wiesz co to programowanie obiektowe, popatrz więc na obiekty w otaczającym nas świecie. Czy możesz skategoryzować obiekty na mutable oraz immutable? Poniżej kilka przykładów:

Mutable:

bak w samochodzie – możemy zmienić jego stan wlewając więcej paliwa

pióro – stan atramentu ciągle się zmienia

telefon – doładowujemy i rozładowujemy baterie

Immutable:

kolor – raz określony kolor, nie może zostać zmieniony, to co raz określiliśmy jako niebieskie już takie pozostanie

biurko – przedmiot codziennego użytku, już raz wyprodukowany, ma określoną szerokość, wysokość, materiał etc.

czas – konkretna data, już raz ustalona, jest niezmiennalna, tak jak chwila w której piszę ten post jest unikalna

Mam nadzieję, że dzięki przykładom, lepiej czujesz o co chodzi z niezmiennalnością obiektów. Zapewne przy nauce programowania obiektowego, cały otaczający Cię świat jest dla Ciebie „zmienialny”, ale po dłuższej chwili namysłu znajdziesz rzeczy, których stanu nie da lub nie powinno się modyfikować.

Jakie benefity daje tworzenie obiektów typu immutable?

Stan zmiennych referencyjnych. Załóżmy, że tworzymy metodę o nazwie ConvertXmlToJson i przekazujemy do niej jakiś obiekt dokumentu XML. Załóżmy taki

fragment kodu:

```
public Json ConvertXMLToJson(XML document)
{
    //Tutaj dzieje się logika konwersji
}
```

wygląda niewinnie, jednak typ który przekazujemy jest referencyjny – to znaczy, że jeśli przypadkowo zmodyfikujemy cokolwiek w obiekcie document w metodzie konwersji, to wpływ naszej zmiany będzie widoczny na zewnątrz metody. Być może, będziemy posługiwać się obiektem XML później i co wtedy? Niestety nie będziemy pewni co do stanu obiektu, co może spowodować późniejsze problemy. Gdybyśmy przygotowali klasę XML, tak aby nie można było modyfikować jej stanu problem „niepewności” by zniknął.

Bezpieczeństwo w wielowątkowości

To troszkę bardziej rozbudowany i skomplikowany temat. Zarówno w językach takich jak Java i C# naszą aplikację możemy zrównoleglić za pomocą wątków (część kodu, np. pobieranie pliku zostaje przetworzone przez oddzielną jednostkę obliczeniową w komputerze, dzięki czemu użytkownik ma poczucie, że dana operacja jest wykonywana w tle – temat wątków jest bardzo obszerny i zastosowałem tutaj bardzo duże uproszczenie, jeśli chcesz wiedzieć więcej to obejrzyj początek tego filmu, w którym tłumaczę trochę bardziej o co chodzi: [link](#)). W przypadku używania kilku wątków mamy dostęp do zmiennych w obrębie całej aplikacji, wyobraź sobie że w jednym momencie jeden wątek zapisuje jakieś parametry do obiektu, a drugi w tym samym czasie próbuje je odczytać. Jaka będzie wartość odczytanej zmiennej? Nie wiadomo – i takiej sytuacji należy unikać w programowaniu jak ognia. W tym momencie pojawiają się obiekty typu immutable.

Pewność kluczy przy korzystaniu z HashSet i HashMap

Jeśli korzystasz z HashMap lub HashSet-ów to klucze identyfikujące konkretne obiekty zostają utworzone za pomocą **hashCode**. W momencie kiedy zmodyfikujemy jakieś pole danego obiektu i spróbujemy użyć metody Contains nie znajdziemy obiektu w naszej kolekcji, ponieważ wartość klucza zapisanego w kolekcji jest inna niż wartość hashCode po modyfikacji obiektu. Ten krótki kod pozwoli na przeanalizowanie tej sytuacji:

```
public class StringHandler{
    private String string;

    public StringHandler(String s) {
        this.string = s;
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o == null || !(o instanceof StringHandler))
            return false;
    }
}
```



```

    else {
        final StringHandler other = (StringHandler) o;
        if (string == null)
            return (other.string == null);
        else
            return string.equals(other.string);
    }
}

public int hashCode() {
    return (string != null ? string.hashCode() : 0);
}

public String toString() {
    return string;
}
}

public static void main(String[] args) {
    StringHandler sh = new StringHandler("ala");
    HashSet testSet = new HashSet();
    testSet.add(sh);
    sh.setString("ma kota");
    System.out.println(testSet.contains(sh)); // FALSE
    System.out.println(testSet.size()); // 1
}
}

```

Dodajemy obiekt do HashSet, a następnie modyfikujemy jedno z pól obiektu za pomocą settera. Co się dzieje, gdy wywołamy metodę contains? Dostaniemy wartość **false**, w przypadku obiektów immutable taka sytuacja nie miałaby miejsca 😊

Jak przygotować naszą klasę, aby uniemożliwić zmianę stanu jej obiektów?

Aby zabezpieczyć się przed zmienialnością obiektów pamiętaj:

- Parametry potrzebne do utworzenia obiektu przekazuj w konstruktorze, najlepiej zastosować tu wzorzec budowniczy: [link](#)
- Kiedy tworzysz nowy obiekt, twórz kopię zmiennych, które mogą zmienić swój stan w trakcie działania programu
- Postaraj się uniemożliwić modyfikację publicznych pól za pomocą takich słów kluczowych jak `readonly` lub `final`
- Jeśli zwracasz jakieś zmienne typy to należy przekazywać ich kopię np. za pomocą metody `Clone`, [uważaj na shallow copy oraz deep copy](#)
- Jeśli nie przewidujemy, by można było dziedziczyć po danej klasie, warto ją zapieczętować (uniemożliwić) przed dziedziczeniem, za pomocą takich słów kluczowych jak `sealed` lub `final`
- Najlepiej, gdyby klasa, którą tworzymy dziedziczyła po typie `immutable`

Obiekty niemutowalne plusy i minusy

Już wiesz czym są obiekty immutable, więc czas na krótkie podsumowanie tej techniki:

Zalety:

- pewność stanu raz zainicjalizowanych obiektów

- zmniejszenie problemu z wielowątkowością (wyeliminowanie takich przypadków jak **wyścig** lub **zakleszczenie**)
- redukcja pomyłek podczas operacji na zmiennych referencyjnych
- poprawne działanie metod typu Contains
- dokładniejsze odzwierciedlenie świata obiektowego

Wady:

- dodatkowy narzut pamięciowy oraz czasowy związany z tworzeniem kopii obiektów
- w przypadku chęci zmodyfikowania pola obiektu, należy utworzyć całkowicie nowy obiekt z zaktualizowanymi wartościami
- zmiana podejścia do typów niemutowalnych, może zająć chwilę czasu

To wszystko na dziś, dzięki za przeczytanie i powodzenia podczas kodzenia!