

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

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).

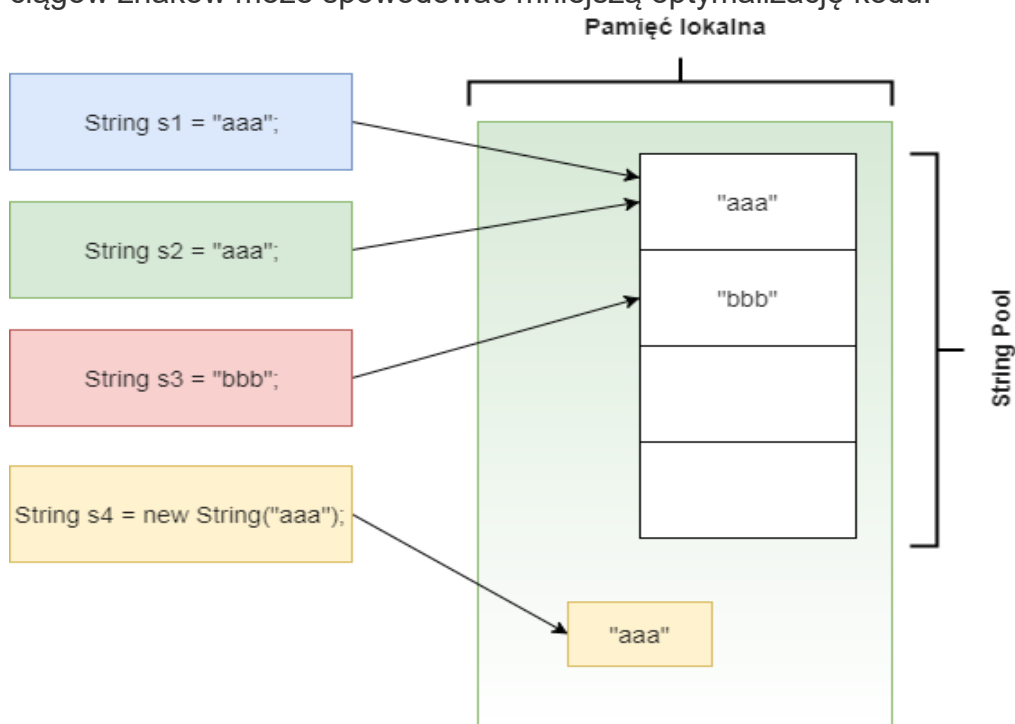
Dlaczego String jest niezmienny?

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.

Co to jest String Pool?

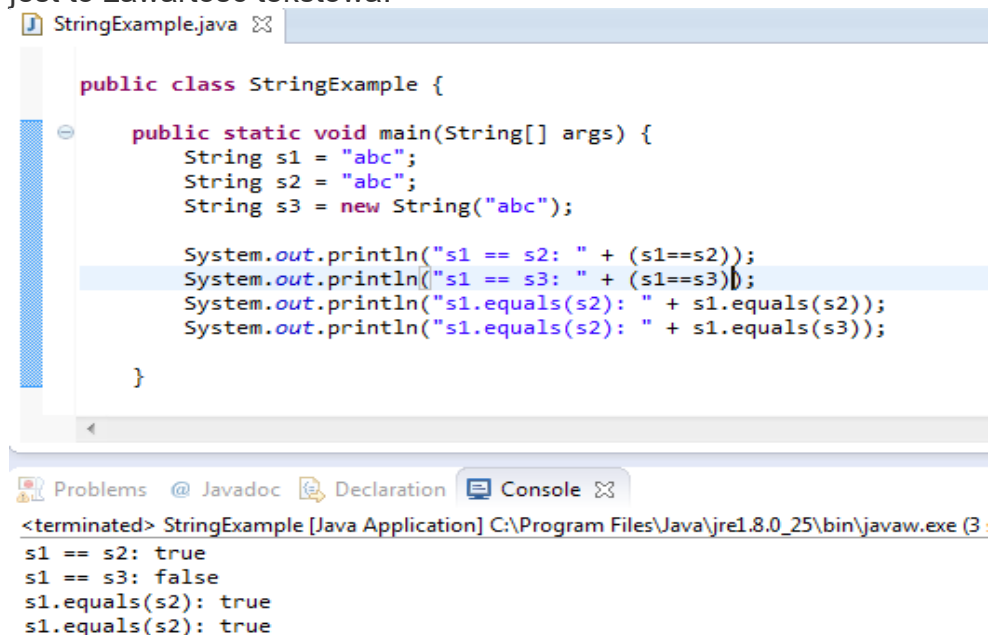
String pool jest specjalnym miejscem w pamięci, w którym przechowywana jest pula wszystkich Stringów. Ładują tam wszystkie obiekty typu String, które zostały zainicjowane przy użyciu cudzysłowu – obiekty, które zostały zainicjowane przy pomocy słowa new ładują w pamięci lokalnej, poza pulą ciągów znaków. 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`.

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
```

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

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.

Czym różni się String od innych pochodnych klas?

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

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

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!

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.

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 łąduje podobnie jak w poprzednim przykładzie w puli, natomiast drugi poprzez użycie słowa „new” tworzony jest w pamięci lokalnej.

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.

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`

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.

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.

Jakie są najważniejsze główne interfejsy zawierające się we frameworku kolekcji?

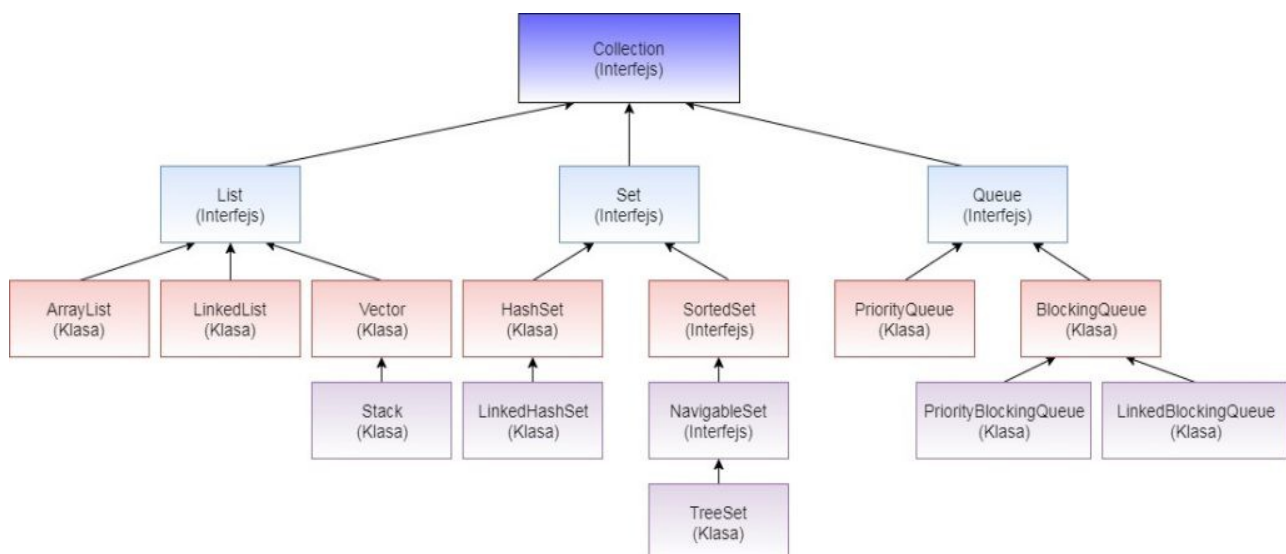
Najważniejsze z nich to:

Collection

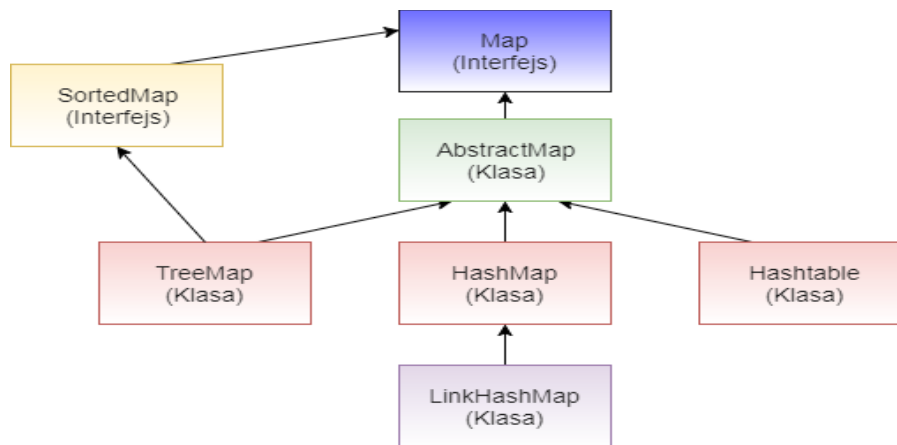
Set

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.

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:

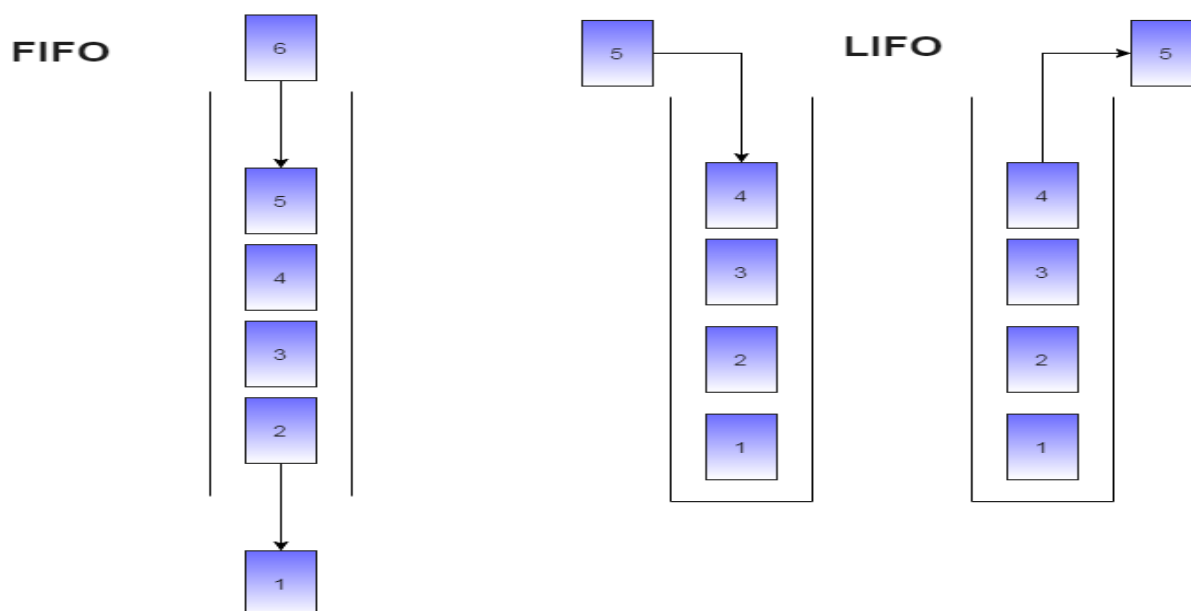
Mapy (Map) są to obiekty które mapują klucze do wartości. Mapa nie może posiadać dwóch takich samych kluczy – może natomiast posiadać dwie takie same wartości (pod warunkiem że mają różne klucze). Podstawowe implementacje interfejsu Map to HashMap, TreeMap oraz LinkedHashMap.

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.

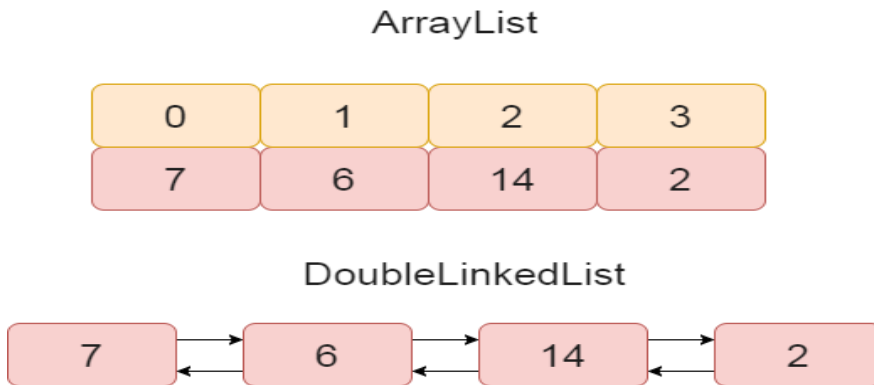
Jaka jest zasadnicza różnica pomiędzy Kolejką (Queue) a stertą (Stack)?

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



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 podzucamy rysunek:



Rys. 4 Różnice pomiędzy ArrayList a Double LinkedList

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

- odczyt elementu – ponieważ ArrayList działa na indeksach jest znacznie szybszy niż LinkedList, potrzebna jest mniejsza ilość operacji aby odczytać wartość pod danym indeksem. Aby odszukać daną wartość w LinkedList należy przeiterować przez całą listę, co z pewnością jest znacznie bardziej czasochłonne.
- usuwanie elementu – w tej sytuacji korzystniej zachowuje się LinkedList. Spowodowane jest to tym że po usunięciu danego elementu zmianie podlegają tylko odnośniki do kolejnego (bądź poprzedniego) elementu w dwóch elementach obok usuwanego. W przypadku ArrayList musimy zmienić położenie wszystkich elementów znajdujących się za usuwanym elementem – powoduje to znacznie większy koszt w porównaniu do LinkedList.
- dodawanie elementów – dodawanie elementów w LinkedList jest łatwe i szybkie w porównaniu do ArrayList gdyż nie istnieje ryzyko powiększenia tablicy i kopiowania zawartości do nowej tablicy. Co to oznacza? Jeżeli dodana wartość sprawia że tablica na której opiera się ArrayList staje się pełna, program musi stworzyć nową, większą tablicę i przekopiować tam wszystkie dane z poprzedniej tablicy. Dodatkowo ArrayList musi również zmienić indeksy poszczególnych wartości jeżeli element nie jest dodany na końcu listy.
- pamięć – LinkedList potrzebuje więcej pamięci gdyż w odróżnieniu od ArrayList (w którym każdy indeks przechowuje jakieś dane) musimy przechowywać informacje o adresach i danych poprzedniego oraz następnego elementu.

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 przystosowana do pracy z aplikacjami wielowątkowymi, a oprócz tego oferuje również większą szybkość działania.

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.

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 ($\log(n)$).

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 `hashCode`ów. Mówiąc inaczej obiekty mogą mieć zgodny `hashCode` 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 niezienne, 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 niezienne (`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 $O(\log n)$.

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 <code>compareTo(Object o1)</code> która porównuje ten obiekt z obiektem <code>o1</code> i zwraca odpowiadającą wartość. (>0 - ten obiekt jest większy niż <code>o1</code> , 0 - obiekty są równe, <0 ten obiekt jest mniejszy niż <code>o1</code>)	Metoda <code>compare(Object o1, Object o2)</code> która porównuje wybrane parametry obiektów <code>o1</code> i <code>o2</code> zwracając wartości odpowiadające ich porównaniu. (>0 - <code>o1 > o2</code> , 0 <code>o1 == o2</code> , <0 <code>o1 < o2</code>)
Jak użyć	<code>Collections.sort(List)</code> - obiekty będą posortowane według metody <code>CompareTo</code> .	<code>Collections.sort(List, Comparator)</code> - obiekty będą posortowane według metody <code>Compare</code> w klasie <code>Comparator</code> .