**Klasy abstrakcyjne:**

- nie mogą być inicjowane słowem new, mogą być tylko dziedziczone przez inne klasy i w ten sposób użyte

- jeśli klasa posiada co najmniej jedną metodę abstrakcyjną to klasa też musi być abstrakcyjna

- stosować gdy potrzeba zaimplementować jakieś wspólne działanie dla wszystkich klas dziedziczących

- klasy dziedziczące muszą implementować metody abstrakcyjne

- ponieważ dziedziczenie jest ograniczone tylko do jednej klasy to czasem lepiej jest stosować interfejsy.

**Klasy abstrakcyjne vs interfejsy:**

- klasę można dziedziczyć (tylko jedną) a interfejsy implementować (wiele)

- interfejs ma tylko metody abstrakcyjne (public abstract), klasa abstrakcyjna może posiadać metody które nie są abstrakcyjne i mogą też być private.

- interfejsy są lepsze do definicji typów natomiast klasy abstrakcyjne reużycia i utrzymywania.

- dodając nową metodę do interfejsu która nie jest default musimy dodać ją do wszystkich klas implementujących bo inaczej otrzymamy błąd kompilatora. Dodając nową metodę nieabstrakcyjną do klasy abstrakcyjnej nie trzeba jej dodawać w klasach dziedziczących.

**String konstruktor vs nadanie wartości:**

Ogólnie jest co najmniej jedna sytuacja kiedy lepiej użyć konstruktora Stringa zamiast zwykłej implementacji. Kiedy tworzymy nowego stringa który jest substringiem poprzedniego. Następująca sytuacja:

String s = "0123456789012345678901234567890123456789";

String s2 = s.substring(0, 1);

s = null;

W tej sytuacji string ‘s’ nie zostanie usunięty przez garbage collector ponieważ tak deklarowane stringi nigdy nie są usuwane przez VM. Żeby pierwszy string został usunięty należy użyć konstruktora i napisać kod w następujący sposób.:

String s2 = new String(s.substring(0, 1));

W ten sposób string pierwszy zostanie usunięty i powstanie nowy string o nazwie s2.

**Final na zmiennej:**

Dla typów prymitywnych oznacza że raz nadana wartość nie może być zmieniona. Dla typów obiektowych oznacza tyle że adres przechowywyany przez zmienną nie może się zmienić natomiast sam obiekt już tak (np. można zmieniać wartości pól takiego obiektu).

**Final na klasach:**

Oznacza że klasa nie może być dziedziczona (przykładowe klasy to String oraz System). Metody w klasie final automatycznie też są final.

**Final na metodach:**

Metoda nie może być napisana lub przysłoniona.

**Pamięć w JVM:**



Podzielona jest na nową i starą. Nowa to eden oraz S0 i S1 (oznaczają surviver memory). Gdy eden się przepełnia nowymi obiektami rusza GC i obiekty które przetrwają trafiają do jednej z pamięci przetrwania S0 lub S1. Obiekty które przetrwają kilka cykli GC trafiają do starej pamięci (old memory). Czyszczenie obiektów z old memory zajmuje dużo czasu dlatego warto dbać aby takich obiektów było mało.

W pamięci perm są metadane opisujące klasy użyte w aplikacji. Są tam też biblioteki środowiska java.

**Java pamięć heap:**

Używana na etapie runtime i w niej się odbywa zarządzanie obiektami (dodawanie usuwanie). W tej pamięci również są klasy JRE (klasy runtimu). W tej pamięci powstają obiekty. Po przepełnieniu rzuca OutOfMemmoryError. Pamięć runtimeu.

**Java pamięć stack:**

Wykorzystywana podczas obsługi wątków (podczas wykonywania wątku). Zawiera informacje o wywoływanych metodach i zawsze działa na zasadzie LIFO. Gdy jest wywoływana jakaś metoda w pamięci stack powstaje blok który posiada informacje o zmiennych lokalnych metody oraz referencjach do obiektów z pamięci heap. Pop przepełnieniu rzuca StackOverFolowError.

**Heap vs Stack:**

- stack jest szybszy niż heap ponieważ jest prostszy i używa LIFO. Jak tylko metoda kończy działanie zwalniana jest pamięć stack

- do elementów pamięci heap można się odwoływać globalnie natomiast do elementów pamięci stack może odwoływać się tylko jeden wątek

- rzucają inne błędy po przepełnieniu

- heap jest używany przez wszystkie elementy aplikacji natomiast stack tylko przez jeden wątek

- stack zawiera tylko prymitywne zmienne i referencje do obiektów, heap zawiera prawdziwe obiekty

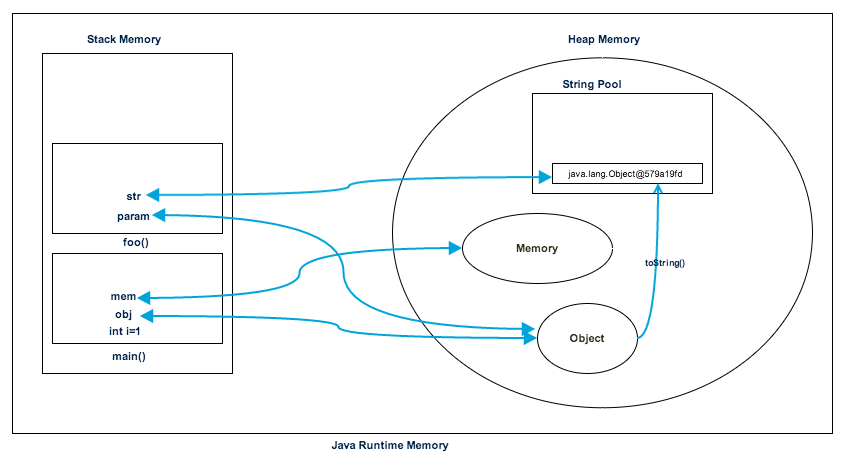
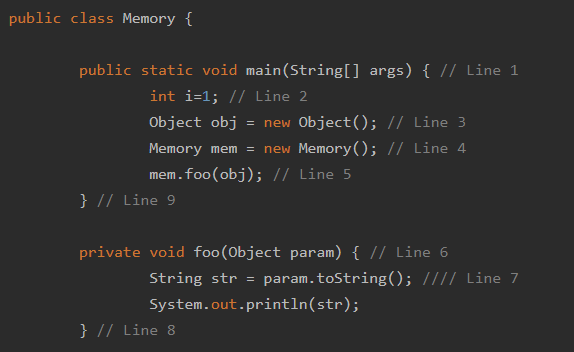
- pamięć stack żyje krótko natomiast heap przez cały okres życia aplikacji

**String Pool:**

Obszar pamięci ze wszystkimi stringami które powstały bez użycia konstruktora. Znajduje się on w pamięci heap.

**Zależność pomiędzy heap a stack:**

Stack odwołuje się do obiektów heap przez referencje do nich. Poniżej program + wykres zależności między satck i heap.



**Kompozycja (has-a):**

Polega to na tym że jedna klasa posiada obiekt innej klasy jako pole, zamiast rozszerzać tą klasę. Lepsze aby kod pozostawał użyteczny ponieważ zmiany w jednej klasie nie muszą mieć wpływu na drugą. Pozwala kontrolować widoczność obiektów dla innych klas (specyfikatory dostępu). Pozwala to również inicjalizować obiekt na etapie runtime przez co nie zaśmiecamy pamięci. Dobrą praktyką jest korzystanie z kompozycji zamiast dziedziczenia (is-a).

**Dlaczego has-a (kompozycja) zamiast is-a (dziedziczenie):**

- kompozycja pozwala posiadać metody w klasach o takich samych nazwach bez strachu przed problemami z kompilacją ponieważ kompilator zawsze będzie wiedział której implementacji użyć na podstawie obiektu wywołującego

- dziedziczenie odsłania wszystkie metody klasy nadrzędnej. Korzystając z kompozycji możemy kontrolować dostęp do pól i metod przy pomocy specyfikatorów dostępu.

- łatwiej testować bo testujemy tylko użyte metody gdy korzystamy z kompozycji. Używając dziedziczenia trzeba testować wszystkie metody klasy super.

**Spring WebFlux:**

Od spring 5 – to implementacja pozwalająca na równoległe przetwarzanie danych w przeciwieństwie do spring mvc, które działa synchronicznie. Równoległe przetwarzanie danych w połączeniu z brakiem blokad (blokowanie programu związane z przepływem informacji) to tzw. reaktywne programowanie. WebFlux jest oparte na projekcie Reactor Project. Są dwa sposoby implementacji, przy pomocy adnotacji (bardzo podobne do zwykłego spring mvc), oraz poprzez konfigurowanie routingów.

**Synchronized:**

Słowo kluczowe które pozwala na zarządzanie dostępem do np. metody przez wątki. Używając tego słowa zapewniamy że tylko jeden wątek w danej chwili będzie mógł skorzystać z danego fragmentu programu a pozostałe wątki będą oczekiwać na swoją kolej.

**Synchronized na metodach instacyjnych:**

Powoduje że dana metoda może być wykonywana przez jeden wątek w danej chwili, pozostałe wątki oczekują. Każda instancja klasy posiada swoją metodę synchronizowaną. Oznacza to że jeżeli mamy dwa różne obiekty tej samej klasy to każdy z nich posiada swoją metodę synchronizowaną a więc wątki działające na tych różnych obiektach nie będą się blokować. Sytuacja jest inna gdy dwa wątki odwołają się do tego samego obiektu i tej samej metody, wtedy muszą czekać.

public static synchronized void add(int value)

**Synchronized na metodach statycznych:**

W tym przypadku metoda jest synchronizowana dla obiektu klasy. Ponieważ istnieje tylko jeden obiekt klasy w JVM to metodę synchronizowaną statyczną klasy może w danej chwili używać tylko jeden wątek.

public static synchronized void add(int value)

**Synchronized na blokach:**

Nie zawsze cała metoda musi być synchronizowana. W takiej sytuacji można korzystać z synchronizowania bloków. Przykład kodu:

public void add(int value){

synchronized(this){

this.count += value;

}

}

Do bloku trzeba podać obiekt którego ma dotyczyć synchronizacja. Obiekt ten nazywany jest monitorem. W tym przypadku this wskazuje na aktualny obiekt korzystający z metody. W przypadku metody statycznej należało by podać nazwę klasy np. Testowa.clas ale również tam może być podany dowolny obiekt. Kod będzie synchronizowany na obiekcie monitora.

**Synchronized podsumowanie:**

- może być używane tylko na metodach i blokach kodu.

- synchronized nie można używać na klasach.

- na konstruktorze nie można używać synchronized – nie ma to sensu ponieważ tylko wątek który tworzy obiekt ma dostęp do konstruktora wiec nie ma szans na konflikty.

- na polach final nie można stosować synchronized ponieważ nie mogą one być modyfikowane po utworzeniu i inicjalizacji. Zresztą kompilator rzuca błędy

- nie stosować zagnieżdżeń w synchronized!! – może to doprowadzić do deadlocka – czyli dwa lub więcej wątki są zablokowane oczekując na siebie w nieskończoność.

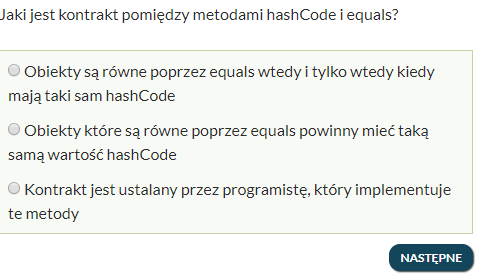
**Czy klasa dziedzicząca po abstrakcyjnej ma konstruktor domyślny:**

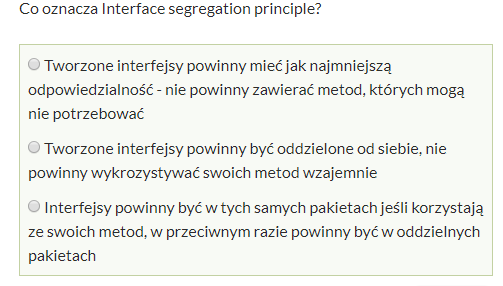
- tak jeśli klasa abstrakcyjna posiada taki konstruktor.

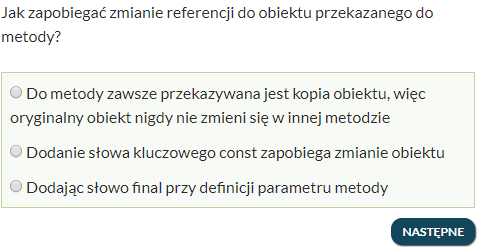
- klasa dziedzicząca po abstrakcyjnej może mieć tylko takie konstruktory jakie ma klasa abstrakcyjna (JEŻELI KLASA ABSTRAKCYJNA NIE MA KONSTRUKTORA DOMYŚLENGO)

- klasa dziedzicząca po abstrakcyjnej może mieć inne konstruktory jeżeli abstrakcyjna posiada konstruktor domyślny

- pamiętać trzeba że konstruktora domyślnego nie ma jeżeli zdefiniujemy jakikolwiek konstruktor jawnie.







Linki do notatek;

<http://www.java67.com/2012/09/what-is-difference-between-interface-abstract-class-java.html>

<https://javarevisited.blogspot.com/2012/06/20-design-pattern-and-software-design.html> - pytania do wzorców