Model pamięci

Spis treści

Jak jest uruchamiany program? . . . 1
- Proces . . . 1
- Wątek . . . 1
- Concurrency (współbieżność) . . . 2
- Parallelism (równoległość) . . . 2
- Po co jakieś procesy, wątki, parallelism i concurrent? . . . 2

Stack (stos) . . . 2
- Stack frame (ramka) . . . 3

Heap (sterta) . . . 4

Podsumowując . . . 5

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

Jak jest uruchamiany program?

Czyli co dzieje się pod spodem jak odpalimy program? Zanim jednak to podsumujemy, chcę przypomnieć czym jest stos (stack) a czym jest sterta (heap).

Ale zanim... Któryś raz pojawi się stwierdzenie `wątek` i cały czas nie jest wyjaśniane ☺. Dlatego, że staram się ten temat poruszyć bardzo powierzchownie, bo jest on złożony i nie jest na teraz.

Zanim przejdziemy dalej, wyjaśnijmy parę terminów.

Proces

Proces – mówi się, że jest egzemplarzem wykonywanego programu. Gdy odpalamy naszą aplikację, może ona uruchomić jeden proces w naszym systemie operacyjnym, ale może też uruchomić takich procesów wiele. W systemach operacyjnych każdy proces ma swoje unikalne PID (process identifier), który jest jednoznacznym identyfikatorem procesu. Gdy uruchamiamy program, system operacyjny takiemu procesowi przydziela zasoby, m.in. pamięć RAM, czas procesora.

Wątek

W ramach jednego **procesu** możemy stworzyć jeden lub więcej **wątków**, które będą wykonywały jakieś części programu. Wątki współdzielą zasoby zarezerwowane dla procesu, oprócz czasu procesora. Ten jest przydzielany indywidualnie do każdego wątku.

Wiedząc już o tym, należy dodać, że jak uruchamiamy aplikację w Javie przez wywołanie metody `main()`, automatycznie tworzymy jeden wątek i na nim operujemy. Wątki możemy też tworzyć sami i

wykonywać pewne czynności współbieżnie, o tym, jak to zrobić i jakie pojawiają się wtedy problemy, dowiemy się później.

Zatem zapamiętaj, że dopóki nie powiem wyraźnie, że jest inaczej, to cały czas będziemy pracowali na jednym wątku.

Chciałbym jeszcze wyjaśnić, jeżeli zaczniemy rozmawiać o kilku wątkach i ich procesowaniu, pojawią nam się 2 terminy (oczywiście pojawi się później więcej ☺), które często są ze sobą mylone:

Concurrency (współbieżność)

Concurrency - oznacza, że aplikacja jest w stanie robić postępy w więcej niż jednym zadaniu jednocześnie. Czy faktycznie jest tak, że dzieje się to w dokładnie tym samym momencie, zależy od tego, ile rdzeni ma procesor na maszynie, gdzie program jest uruchamiany. Ale sama współbieżność oznacza, że więcej niż jedno zadanie jest realizowane w tym samym czasie, w trakcie działania naszej aplikacji.

Wypadałoby podać jakiś przykład z życia. Jesteś w stanie jednocześnie jeść i mówić? Dokładnie w tym samym czasie? Mama zawsze mówiła, żeby tak nie robić, bo się udławisz ☺. Można powiedzieć, że jesteś w stanie realizować na raz konsumowanie posiłku i prowadzenie konwersacji, ale dokładnie w tym samym czasie przełykać posiłku i wymawiać słowa nie możemy. Czyli musisz przełknąć, potem możesz coś powiedzieć, potem możesz przełknąć, potem możesz coś powiedzieć. Nie da się tego zrobić jednocześnie. Realizujemy jakiś proces/zadanie (czyli jedzenie i rozmawianie) w tym samym czasie i idą one do przodu (robimy w nich progress), czyli jedzenia jest coraz mniej i zmierzamy w tej rozmowie do końca, ale jednocześnie nie jesteśmy w stanie tego robić.

Zatem wykonujemy kilka zadań w tym samym czasie, ale niekoniecznie są one wykonywane równocześnie.

Parallelism (równoległość)

Parallelism - oznacza, że aplikacja jest w stanie podzielić to, co ma zostać wykonane na mniejsze fragmenty i faktycznie wykonać je równolegle. Aby tak mogło się stać, wymagane jest posiadanie więcej niż jednego CPU, wtedy każde z nich jest w stanie wykonywać jakieś zadanie jednocześnie.

Znowu przykład z życia, to by mogło zadziałać, jakbyśmy mieli obok siebie 2 usta. Jedno by jadło, a drugie rozmawiało w tym samym czasie. Tutaj akurat mogłoby się to dziać w tym samym czasie.

Po co jakieś procesy, wątki, parallelism i concurrent?

Po co w ogóle o tym piszę? Niżej przy wyjaśnianiu stosu, pojawiają się terminy o współbieżności i zachowaniu stosu w odniesieniu do kilku wątków. Nie będziemy tworzyć na razie w naszych aplikacjach kilku wątków, ale warto wiedzieć, co dzieje się z pamięcią, jeżeli wątków jest więcej niż jeden i aplikacja może wykonywać się współbieżnie.

Stack (stos)

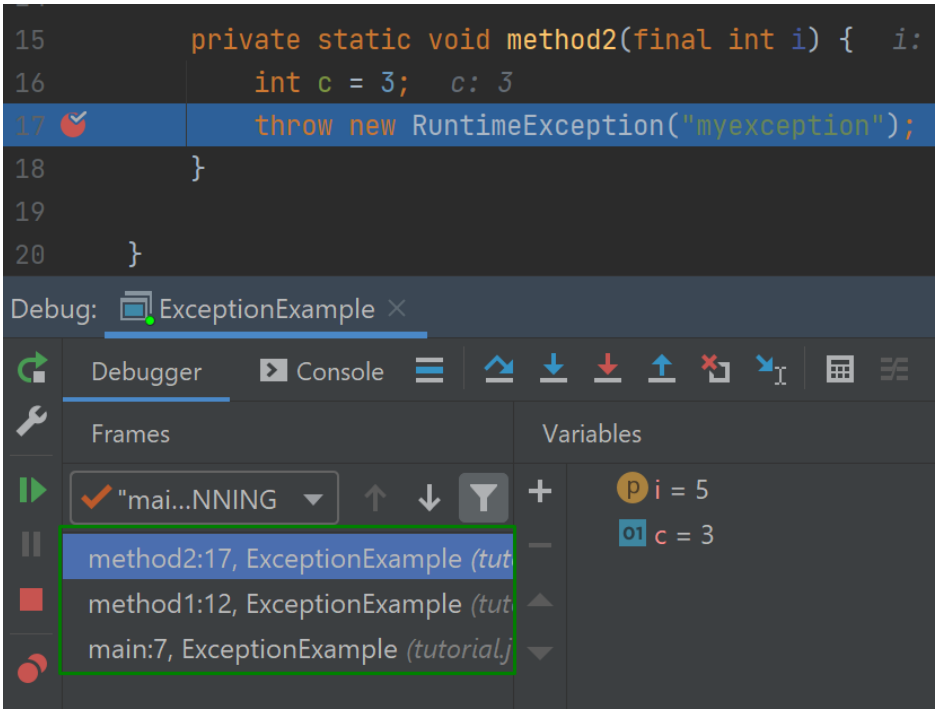
Struktura danych, w której przechowywane w pamięci są zmienne (referencje). Główne aspekty **stosu**:

- służy do wykonania wątku - jak chcemy uruchomić wątek, potrzebny jest nam stack (stos),
- zawiera wartości typów prymitywnych oraz referencje do obiektów znajdujących się na stercie (heap),
- z racji, że jest to stos - zachowanie struktury to `LIFO` - *last-in-first-out*,
- za każdym razem gdy wywołujemy nową metodę w kodzie, tworzony jest nowy element na szczycie stosu zawierający wartości specyficzne dla tej metody - prymitywy i referencje,
- kiedy metoda zakończy działanie - jej ramka stosu (czyli fragment stosu utworzony na potrzeby wywołania tej metody) jest czyszczona, wywołania wracają do metody wywołującej, dane stają się dostępne dla następnej metody,
- rośnie i kurczy się w miarę wywoływania i zwracania nowych metod,
- zmienne wewnątrz stosu istnieją tak długo, jak długo działa metoda, która je utworzyła. Gdy metoda kończy swoje działanie, jej ramka stosu jest czyszczona,
- nie ma potrzeby uruchamiania `GarbageCollection`, ponieważ stos jest automatycznie przydzielany i zwalniany za każdym razem, gdy metoda kończy swoje działanie,
- w przypadku braku dostępnego miejsca, dostępnej pamięci - wyrzucany jest błąd `java.lang.stackOverflowError`,
- w porównaniu do sterty, ten obszar pamięci jest szybki,
- gdy zaczynamy w aplikacji tworzyć wiele wątków - każdy wątek ma swój stos.

Stack frame (ramka)

Wprowadźmy takie pojęcia jak call stack (stos wywołań) - w informatyce, call stack jest strukturą przechowującą informacje o aktywnych wywołaniach programu.

Pamiętasz StackTrace z wyjątków? **StackTrace** jest kopią **call stacka**, która została zapisana w momencie, gdy w programie wystąpił błąd. Call stack jest natomiast obecnym stosem wywołań w programie, w momencie, gdy ten faktycznie jest uruchomiony. Czyli, gdy mamy uruchomiony debugger, to w lewym dolnym rogu, patrząc na stos wywołań poszczególnych metod, patrzymy na call stack:



Obraz 1. Call stack (stos wywołań)

A jeżeli w tym momencie zostałby wyrzucony błąd i żadna metoda by go nie obsłużyła, zobaczylibyśmy taki stackTrace:

```
Exception in thread "main" java.lang.RuntimeException: myexception
    at tutorial.zajavka.ExceptionExample.method2(ExceptionExample.java:17)
    at tutorial.zajavka.ExceptionExample.method1(ExceptionExample.java:12)
    at tutorial.zajavka.ExceptionExample.main(ExceptionExample.java:7)
```

Zwróć uwagę, że widoczne są te same nazwy metod oraz te same numery linijek.

Call stack składa się ze stack frames (ramek stosu). Kolejne ramki są tworzone za każdym razem, gdy wywoływana jest nowa metoda programu. Ramki mają różne rozmiary, które są zależne od parametrów metody, ilości zmiennych. Podczas wywołania metody, kod może mieć dostęp tylko do wartości dostępnych w konkretnym stack frame, nie może mieć dostępu do zmiennych z innego stack frame, bo to by znaczyło, że ma dostęp do zmiennych z innej metody. stack frame może być wizualizowany, jako najnowszy klocek na stosie (zobacz jeszcze raz obrazek wyżej z Intellij).

Heap (sterta)

Sterta jest używana do przydzielania pamięci dla obiektów Java i klas JRE w czasie wykonywania programu. Główne aspekty sterty:

- ten obszar pamięci jest dalej podzielony na części, zwane pokoleniami (nie będziemy teraz schodzić w to głębiej):
  - Young generation - tutaj powstają i alokowane są wszystkie nowe obiekty. Gdy nie ma tu już miejsca, uruchamiane są drobne GC (Garbage Collection) (minor GC - czyszczenie Young generation)
  - Old lub Tenured Generation - miejsce do przechowywania "ocalałych" - kiedy obiekty są

tworzone w młodym pokoleniu i co jakiś czas są czyszczone, może być tak, że nasz obiekt przeżyje kilka takich czyszczeń. Liczony jest wtedy jego wiek, na zasadzie, ile cykli czyszczenia przetrwał. Jeżeli przekroczy ustalony próg, obiekt jest przenoszony do starej generacji,

    - Permanent Generation - ta część składa się z metadanych JVM dla uruchomieniowych klas i metod aplikacji,

- sterta jest zorganizowana przy użyciu złożonych technik zarządzania pamięcią polegających na zarządzaniu obszarami Young Generation, Old lub Tenured Generation i Permanent Generation. Przypomnę, że Java sprząta pamięć za nas,
- w przypadku braku dostępnej pamięci, którą można zarezerwować, aby stworzyć obiekt - wyrzucany jest błąd `java.lang.OutOfMemoryError`,
- sterta jest stosunkowo wolniejsza niż stack,
- ten obszar pamięci nie jest automatycznie zwalniany, gdy np. jakaś metoda zakończy swoje działanie, tak jak ma to miejsce w przypadku stosu. GC jest potrzebne, aby zwolnić trochę miejsca, usuwając nieużywane obiekty,
- w przypadku stosu, jeżeli tworzyliśmy nowy wątek w aplikacji, tworzony był nowy stos, dedykowany dla tego wątku. sterta jest jedna, wiele wątków może jednocześnie uzyskać dostęp do tego samego obiektu, co może prowadzić do wielu nieprzewidzianych błędów, ale o tym dowiemy się w przyszłości.

Podsumowując

Czyli jeszcze raz.

Zmienne prymitywne są przechowywane na stosie. Referencje do obiektów są przechowywane na stosie pod warunkiem, że jest to referencja o zakresie lokalnym, czyli obiekt jest stworzony w metodzie i przypisany do referencji w zakresie lokalnym metody. Jeżeli tworzymy obiekty, to sam obiekt jest przetrzymywany na stercie, referencja do niego jest przechowywana na stosie. Chyba że referencja do naszego obiektu jest zdefiniowana jako pole w klasie, wtedy jest ona również przechowywana na stercie.