Jakub Kubryński

Co każdy programista Java powinien wiedzieć o JVM

Dla znacznej liczby programistów Javy jednym z podstawowych elementów związanych bezpośrednio z codzienną pracą jest Wirtualna Maszyna Javy (Java Virtual Machine, w skrócie JVM). Jednak pomimo oczywistości jej wpływu na rozwiązywane przez nas zagadnienia, wyjątkowo często jest ona pomijana i traktowana jedynie jako „czarne pudełko", do którego wnętrza nie warto zaglądać. I właśnie otwarciu tego czarnego pudełka, połączonego z omówieniem jego zawartości, poświęcony jest ten artykuł.

Już wiele wieków temu ludzie nauczyli się wszystkie rzeczy i zjawiska, których nie potrafili zrozumieć, tłumaczyć magią. Zadziwiająco wiele z tych przyzwyczajeń zostało nam do dziś. Sir Arthur Charles Clarke powiedział, że każda „dostatecznie zaawansowana technologia nie różni się od magii." Podobnie także my, programiści, podchodzimy do technologii, z którymi pracujemy na co dzień. Jeżeli są zbyt złożone, zamykamy je szczelnie w pudełeczku i traktujemy jak magię, której przecież i tak nikt nie zrozumie. Mówiąc językiem technicznym, przykrywamy warstwą abstrakcji i udajemy, że złożoność jest jedynie wyimaginowanym tworem wciskanym nam przez autorów książek. Niestety, jak wiemy z prawa „cieknących abstrakcji" sformułwanego przez Joela Spolsky'ego, każda abstrakcja w pewnym momencie zaczyna cieknąć i wymagać od nas znajomości zarówno jej wnętrza, jak i zasad, którymi kierowali się jej autorzy.

CO TO JEST JVM?

Czym zatem jest JVM? Mówiąc wprost, jest to nic więcej niż aplikacja napisana w większości w języku C++. I tyle. Oczywiście jest to aplikacja bardzo złożona, implementująca liczne wyszukane algorytmy czy optymalizacje, jednak w dalszym ciągu jest to „tylko" aplikacja. Kod źródłowy, który każdy z nas może pobrać, przeanalizować, skompilować i uruchomić. Jak każdą inną napisaną przez programistów aplikację.

Czym natomiast w takim razie jest Java? W dużym skrócie idea, która przyświecała autorom Javy, to „napisz raz, uruchamiaj gdziekolwiek" (Write Once, Run Anywhere – WORA). Żeby to osiągnąć, potrzebowali nowego tworu żyjącego poziom wyżej niż kod maszynowy, dzięki czemu nie byłby on zależny od konkretnej platformy, na której docelowo miałby być uruchamiany. Ten kod, nazywany inaczej kodem bajtowym Javy (Java bytecode), stanowi właśnie reprezentację naszej aplikacji, która po procesie kompilacji (zamiana kodu źródłowego na kod bajtowy) uruchamiana jest w Wirtualnej Maszynie. Co warte zaznaczenia JVM w żaden sposób nie jest zainteresowana językiem źródłowym, z którego powstał byte code, dzięki czemu ta sama maszyna może jednocześnie uruchamiać aplikacje napisane w Javie, Scali, Groovy, Kotlinie czy jakimkolwiek innym języku kompilowanym do kodu bajtowego Javy.

Pełna specyfikacja Wirtualnej Maszyny Javy, definiująca wszystko, począwszy od jej architektury, aż do sposobów implementacji poszczególnych elementów, została zebrana w dokument nazwany „The Java® Virtual Machine Specification", którego aktualną wersję można znaleźć pod adresem http://docs.oracle.com/javase/specs/jvms/se8/html/.

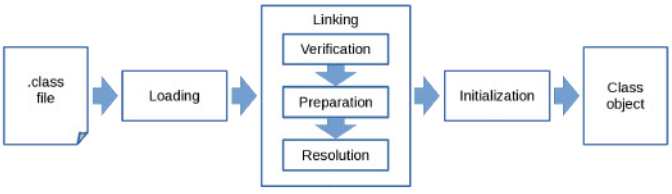
Podobnej różnorodności jak w przypadku języków możemy także doświadczyć w zakresie implementacji JVM. Najpopularniejszy Oracle HotSpot nie jest bynajmniej jedynym dostępnym produktem spełniającym wymagania specyfikacji. Z innych popularnych maszyn możemy wymienić także OpenJDK, IBM J9 czy też Azul Zing.

CYKL WYKONANIA PROGRAMU

Przeniesiemy się teraz do momentu, gdy mamy już przygotowany artefakt, który chcemy uruchomić (jego utworzenie wykracza poza zakres tego artykułu). Pierwszym krokiem jego uruchomienia jest etap startu maszyny wirtualnej (JVM Spec 5.2. Java Virtual Machine Startup). Polega on na odnalezieniu głównej klasy w celu uruchomienia statycznej metody `main(String[])`. Aby jednak uruchomienie metody było możliwe, klasę należy najpierw załadować. Odpowiada za to etap ładowania (Loading – JVM Spec 5.3. Creation and Loading). Głównym aktorem tego działania jest ClassLoader (nie podejmuję się przetłumaczenia tego zwrotu na polski), którego zadaniem jest odnalezienie binarnej reprezentacji klasy na classpath'ie, a następnie jej wczytanie do odpowiedniego segmentu pamięci – utworzenie obiektu Class, który jednak nie jest jeszcze gotową do użycia klasą. Po tym, jak klasa już znalazła się w pamięci, następuje trzyetapowy proces tak zwanego łączenia (JVM Spec 5.4. Linking). Pierwszy krok to przeprowadzenie weryfikacji (verification) definicji klasy, na które składa się sprawdzenie, czy klasa lub interfejs są poprawne strukturalnie, wszystkie zależności są załadowane, a także szereg dodatkowych kontroli jak na przykład, czy:

- nie nadpisujemy finalnych klas lub metod,
- metody „szanują" poziomy dostępu (nie próbujemy na przykład wołać prywatnych metod innych klas),
- metody mają prawidłową liczbę i typy parametrów,
- bytecode nie próbuje w złośliwy sposób manipulować stosem,
- zmienne są zainicjowane przed próbą odczytu,
- zmienne mają wartości prawidłowego typu.

Następnie przechodzimy do fazy przygotowania (preparation), podczas której tworzone oraz inicjowane są wszystkie pola statyczne. Ostatnim krokiem łączenia jest rozwiązywanie (resolution). Zapewnia ono, że każdy typ, do którego odnosi się nasza klasa, jest gotowy do wykorzystania (a nie tylko załadowany). Teraz pozostaje nam już tylko utworzyć obiekt naszej klasy, który powstaje w procesie inicjalizacji (JVM Spec 5.5. Initialization), po którego zakończeniu obiekt Class jest już pełnoprawną klasą gotową do użycia.



Rysunek 1. Proces ładowania klas

Jak wiemy z doświadczenia, nic nie trwa wiecznie, a co za tym idzie także wystartowana na początku tego akapitu Wirtualna Maszyna musi zakończyć swój żywot. Dzieje się to w momencie, kiedy jakikolwiek wątek zawoła metodę `Runtime.exit()`, lub zostanie ona „zabita" przez czynniki zewnętrzne. Jako ciekawostkę warto przytoczyć tu, iż Wirtualna Maszyna potrafi popełnić „samobójstwo". Żeby ją do tego nakłonić, wystarczy wykorzystać flagę `-XX:SelfDestructTimer=10` ustawioną na odpowiednią (w tym przypadku 10) liczbę minut.

JUST-IN-TIME COMPILER

Teraz wrócimy do momentu, w którym otrzymaliśmy gotową do użycia główną klasę naszej aplikacji. Aby rozpocząć pracę, należy uruchomić metodę `main`, która z reguły powoduje swoistą kaskadę tworzenia nowych klas i wywołań metod. Jak już wiemy, ciała metod są transformowane przez kompilator do postaci kodu bajtowego, który następnie jest uruchamiany przez Maszynę Wirtualną. Mówiąc precyzyjniej, jest on wykonywany przez interpreter, który w odróżnieniu od bytecode'u jest już zależny od konkretnej architektury systemowej, na której uruchamiamy naszą aplikację. Niestety takie podejście poza zagwarantowaniem przenaszalności aplikacji ma dość poważną wadę – interpretacja jest znacznie wolniejsza niż wykonywanie kodu natywnego, czyli skompilowanego do postaci binarnej odpowiedniej architektury. To jest też źródło często powtarzanego mitu, że „Java jest wolna", który za chwilkę bez najmniejszych skrupułów obalimy. Powiedzieliśmy sobie, że fakt uruchamiania kodu w Maszynie Wirtualnej pozwala nam uniezależnić kompilację od natywnego kodu. Jest to prawda, jednak to tylko wierzchołek góry lodowej, bo sam fakt uruchamiania kodu w kontrolowanym środowisku otwiera przed nami ogromne możliwości, jak choćby jego ciągła optymalizacja. I dokładnie ten mechanizm wykorzystuje w stosunku do naszych aplikacji JVM. Nieprzerwanie zbierane są statystyki wykonania naszych metod i w momencie, w którym osiągną odpowiedni próg wywołań (czyli warto się nimi zainteresować, bo mogą mieć istotny wpływ na wydajność), rozpocznie się ich optymalizacja. Te metody nazywamy inaczej „gorącymi metodami" lub „gorącymi punktami". Jak można się domyślić, to ostatnie określenie jest genezą nazwy najpopularniejszego JVM – Oracle HotSpot. Dla uproszczenia przyjmujemy, że dla wszystkich maszyn serwerowych (maszyna serwerowa w rozumieniu HotSpota to każdy komputer mający co najmniej 2 rdzenie i 2 GB pamięci operacyjnej) ten próg to 10000 wywołań danej metody. Elementem JVM odpowiedzialnym za te działania jest kompilator Just-In-Time, czyli popularny JIT. Co się zatem dzieje z metodą, która stała się obiektem zainteresowania JITa? Wachlarz optymalizacji, które wykonuje JVM, jest bardzo szeroki i obejmuje kilkadziesiąt pozycji, natomiast ograniczając się do tych najpopularniejszych, warto wymienić:

- zagnieżdżanie metod (method inlining),
- eliminacja martwego kodu (dead code elimination),
- kompilacja do kodu natywnego,
- rozwijanie pętli (loop unrolling),
- grupowanie blokad (lock coarsening),
- eliminacja blokad (lock elision),
- ostrzenie typów (type sharpening).

Pokrótce omówimy trzy pierwsze, co pozwoli nam lepiej zrozumieć sposób działania optymalizatora.

Zagnieżdżanie metod polega na zastąpieniu wywołania metody jej ciałem, co demonstruje poniższy przykład, w którym przeprowadzimy tę operację w stosunku do metody `getValueFromSupplier`:

Listing 1. Zagnieżdżanie metod

```
public String getValueFromSupplier(Supplier<String> supplier) {
  return supplier.get();
}
public String businessMethod(String param) {
  Supplier<String> stringSupplier = new StringSupplier("my" + param);
  return getValueFromSupplier(stringSupplier);
}

// zostanie zamieniona na

public String businessMethod(String param) {
  Supplier<String> stringSupplier = new StringSupplier("my" + param);
  return stringSupplier.get();
}
```

Jak widzimy, JIT stwierdził, że metoda `getValueFromSupplier()` jest na tyle często wołana (tutaj próg jest wyjątkowo niski i domyślnie wynosi 250 wywołań) i na tyle mała, że można ją niejako skopiować w miejsce jej pierwotnego wywołania. Zagnieżdżanie metod często jest nazywane „matką wszystkich optymalizacji", ponieważ łącząc ze sobą rozproszoną logikę, zyskujemy dużo lepszy „przegląd pola", a co za tym idzie możemy pozwolić sobie na zastosowanie dodatkowych optymalizacji. Aby unaocznić ten mechanizm, po raz kolejny odwołam się do przykładu. Wyobraźmy sobie, że metoda `getValueFromSupplier()` wygląda następująco:

Listing 2. Rozbudowa logiki

```
public String getValueFromSupplier(Supplier<String> supplier) {
  if (supplier == null) {
    throw new IllegalArgumentException("Supplier cannot be null");
  }
  return supplier.get();
}
```

Jest to implementacja jak najbardziej prawidłowa i nie patrząc na kontekst jej wywołania, niewiele możemy w niej zmienić. Sytuacja ulega jednak radykalnej zmianie, gdy zagnieździmy (inline) metodę w miejscu jej wywołania:

Listing 3. Metoda po zagnieżdżeniu

```
public String businessMethod(String param) {
  Supplier<String> supplier = new StringSupplier("my" + param);
  if (supplier == null) {
    throw new IllegalArgumentException("Supplier cannot be null");
  }
  return supplier.get();
}
```

Po wykonaniu tej operacji stało się jasne, że sprawdzanie, czy `supplier` nie wskazuje na pusty obiekt, nie ma sensu, bo warunek ten będzie spełniony zawsze. Skoro my to widzimy, to możemy oczekiwać, że do takiego samego wniosku dojdzie JIT. I jest to prawidłowe założenie, bo właśnie tak się stanie. Jeżeli jakiś kod nigdy nie będzie wykonany, lub też jego wykonanie nie będzie miało żadnego wpływu na działanie naszego programu, to blok ten zostanie uznany za kod martwy (dead code) i w efekcie usunięty. Czyli nasza metoda zmieni się następująco:

Listing 4. Eliminacja martwego kodu

```
public String businessMethod(String param) {
  Supplier<String> supplier = new StringSupplier("my" + param);
  return supplier.get();
}
```

Ostatnią optymalizacją, którą omówimy, jest kompilacja do kodu natywnego. Jest to optymalizacja, która daje niesamowity przyrost wydajności. Polega (jak zresztą nazwa wskazuje) na konwersji kodu bajtowego, który uruchamiany jest w interpreterze na natywny kod maszynowy, którego wykonywanie jest nawet 20-krotnie szybsze. Co warte zauważenia konwersja ta nie jest permanentna i jeżeli JIT na podstawie obserwacji stwierdzi, że jeżeli decyzja o wykonaniu kompilacji była błędna, może kod natywny wyrzucić i powrócić do interpretacji. Głównie dzieje się tak dlatego, iż JIT uznaje, że warto spróbować zastosować dodatkowe optymalizacje, bądź zmieniły się okoliczności zewnętrzne (na przykład załadowano dodatkowe klasy) i wykonane wcześniej optymalizacje należy wycofać.

Jak widać, JIT wykonuje ogromną ilość pracy, szczególnie w okresie zaraz po starcie aplikacji, kiedy zaczyna ona pracować w stanie całkowicie interpretowanym i bez optymalizacji (z wyłączniem tych wykonanych przez kompilator `javac`). Ten okres nazywamy „rozgrzewaniem systemu". Świadomość jego istnienia jest bardzo ważna, ponieważ JVM przeznacza wyjątkowo dużo zasobów na optymalizację i kompilację naszej aplikacji. W efekcie jej profil wydajnościowy może być zupełnie inny niż po zakończeniu rozgrzewania. Nie należy o tym zapominać, gdyż jakiekolwiek obserwacje wydajnościowe wykonane w tym momencie nie mają sensu, ponieważ w praktyce obserwujemy „inną wersję" naszej aplikacji, niż ta, która docelowo będzie obsługiwała nasz produkcyjny ruch.

KOMPILACJA POZIOMOWA (TIERED COMPILATION)

Pewnie wielu z nas zadało sobie kiedyś pytanie: skoro kod natywny jest dużo wydajniejszy niż interpretowany, to czemu nie przetransformować do niego całej aplikacji? Powodów jest wiele, natomiast dwa najważniesze to brak możliwości dalszej optymalizacji kodu, który nie jest już bezpośrednio kontrolowany przez JVM, drugim powodem jest natomiast koszt samej kompilacji. To m.in. właśnie ze względu na tę drugą przyczynę cały czas trwają prace rozwojowe wokół kompilatora JIT. Rozróżnić możemy dwie zasadnicze implementacje kompilatora:

- kliencką – C1
- serwerową – C2

Jeżeli pamiętacie flagi `-server` i `-client` dostępne w JVM, to w głównej mierze służyły one do wyboru określonej implementacji kompilatora JIT. W dużym uproszczeniu można powiedzieć, iż wersja kliencka działa szybciej niż serwerowa, natomiast produkowany przez nią kod natywny jest wolniejszy. Mieliśmy zatem do wyboru dostać szybciej słabszą optymalizację lub trochę później – lepszą. Ktoś natomiast zadał sobie pytanie, czy nie dałoby się tych dwóch wersji połączyć w jedną i mieć „zarówno rybki, jak i akwarium". Tak narodziła się idea kompilacji poziomowej. Standardowo poziomy były dwa:

- kod interpretowany
- kod natywny (wynik kompilacji JIT)

Przy włączeniu kompiilacji poziomowej (domyślne w JDK 8 lub we wcześniejszych wersjach po użyciu flagi `-XX:+TieredCompilation`) tych poziomów robi się więcej:

0: kod interpretowany
1: prosty kod natywny C1 (kliencki)
2: ograniczony kod natywny C1
3: pełny kod natywny C1
4: kod natywny C2 (serwerowy)

Standardowy cykl życia wygląda następująco: rozpoczynamy oczywiście od poziomu 0, czyli kodu intepretowanego. Po odpowiedniej liczbie wywołań (domyślnie 2000) JIT wykona kompilację do poziomu 3, używając kompilatora klienckiego. Jeżeli natomiast metoda dalej będzie wykorzystywana często, po przekroczeniu kolejnego progu wywołań (domyślnie 15000) przeprowadzi ponowną kompilację, tym razem przy wykorzystaniu kompilatora serwerowego. Dzięki temu udało nam się zintegrować zalety obu kompilatorów i „upiec dwie pieczenie przy jednym ogniu".

Jeżeli teraz połączymy wszystkie opisane w tym artykule cechy kompilatora JIT, widzimy, że wydajność Javy (i innych języków kompilowanych do jej kodu bajtowego) to problematyka wyjątkowo złożona. Z jednej strony zaczynamy od prostej interpretacji, która nie może nawet konkurować z wydajnością aplikacji natywnych. Z drugiej jednak fakt zastosowania środowiska uruchomieniowego w postaci Wirtualnej Maszyny pozwala nam na zastosowanie optymalizacji, które nie są i nigdy nie będą dostępne na etapie kompilacji statycznej, co może nawet doprowadzić do sytuacji, w której nasze aplikacje będą szybsze niż ich natywne odpowiedniki.

OBSZARY PAMIĘCI

Kwestia pamięci w Javie to temat bardzo często pomijany. Ograniczamy się tutaj z reguły do wykorzystania flagi `XMX` (maksymalny heap) i koniecznie ustawionej na taką samą wartość `XMS` (startowa wielkość heap). Pierwsza wartość stanowi panaceum na `OutOfMemoryError` pojawiający się w logach aplikacji lub na konsoli naszych narzędzi developerskich. Niestety zapominamy, że przyczyn pojawienia się tego wyjątku może być kilka, co wynika z dość dużej złożoności struktury pamięci JVM. Co warte zaznaczenia druga praktyka to natomiast jeden z podstawowych mitów dotyczących wydajności Javy, który jednak co prawda nadal wpływa na wydajność, jednak w zdecydowanie negatywny sposób.

Zobaczmy teraz, w jaki sposób HotSpot JVM dzieli pamięć i jakie są role poszczególnych obszarów. Cała pamięć, jaką wykorzystuje Wirtualna Maszyna, dzieli się na kilku poziomach. Podstawowy podział to rozróżnienie na stertę (heap) i pamięć natywną (off-heap). W dużym uproszczeniu możemy powiedzieć, że podstawowa różnica między nimi jest taka, że pierwszy obszar leży w kręgu zainteresowania odśmiecacza pamięci (garbage collector), natomiast drugi już nie. Drugie uproszczone rozróżnienie, jakie możemy przyjąć, dotyczy obiektów (czyli naszych podstawowych jednostek, składających się na aplikację). Obiekty żyją właśnie na stercie (heap), natomiast poza nim lokowane są głównie byty potrzebne do wsparcia działania samej Wirtualnej Maszyny i reprezentacji jej struktur wewnętrznych. Teraz zajmiemy się szerzej każdym z dwóch podstawowych bloków.

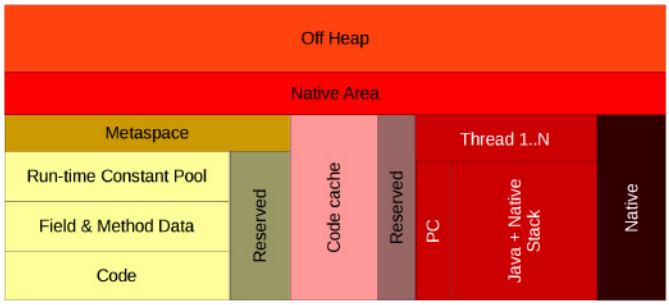
STERTA (HEAP)

Rysunek 2. Struktura sterty

Jak już powiedzieliśmy, sterta jest głównym przedmiotem zainteresowania programistów, jako że to właśnie tutaj żyją wszystkie tworzone przez nas obiekty. To właśnie sposób wykorzystania sterty ma największy wpływ na wydajność i responsywność naszej aplikacji. Heap nie jest przestrzenią ciągłą i podlega dalszemu podziałowi na dwie generacje – młodą (young) i starą (old lub tenured). Taki podział przestrzeni wynika jednoznacznie z tzw. hipotezy generacyjności, czasem określanej także jako „śmiertelność niemowląt" (infant mortality). W ogólnym zarysie hipoteza ta mówi o tym, że prawdopodobieństwo zguby młodych (niedawno utworzonych) obiektów jest dużo wyższe niż obiektów, które już jakiś czas przeżyły. Trafnym porównaniem mogą być tutaj żółwie. Największe prawdopodobieństwo unicestwienia mają one zaraz po wykluciu się z jaj podczas podróży przez plażę do morza. Jeżeli natomiast uda im się dotrzeć do wody, szansa przeżycia gwałtownie wzrasta, a długość życia sięga kilkudziesięciu lat. Bardzo zbliżonym sposobem rozumowania kierowali się autorzy Javy, co właśnie ma odzwierciedlenie w układzie pamięci Wirtualnej Maszyny. Takie podejście pozwala na stosowanie różnych strategii w zależności od generacji obiektu. To jednak nie jest jeszcze koniec podziałów sterty. O ile stara generacja jest już przestrzenią ciągłą, to w młodej możemy wyróżnić kilka podprzestrzeni. Pierwszą z nich jest eden. Tworzone przez nas obiekty alokowane są początkowo właśnie tutaj i pozostają w tym segmencie do momentu, aż zostanie uruchomiony garbage collector. Wtedy, o ile obiekty jeszcze żyją, zostaną przeniesione do kolejnej przestrzeni młodego pokolenia, czyli tak zwanej przestrzeni prze-

trwalnikowej (survivor). Jak zapewne zauważyliście na załączonym obrazku, przestrzeń ta jest powielona, co wynika z algorytmów odśmiecacza pamięci zastosowanych w JVM – nie są one jednak tematem tego artykułu. Warte zaznaczenia jest natomiast to, iż właśnie w tych przestrzeniach przebywają obiekty aż do momentu, w którym garbage collector uzna je za warte promocji do starszej generacji. Wtedy trafią właśnie do przestrzeni old (określanej także jako tenured) i tak pozostaną aż do końca swych dni. Jak pewnie także zdążyliście zauważyć, w każdej z dwóch dostępnych generacji mamy fragmenty pamięci, które są zarezerwowane, jednak nie wchodzą w skład wykorzystywanych obszarów. Powodem tego jest dynamika struktury pamięci, która w celu osiągnięcia maksymalnej wydajności nieustannie zwiększa lub zmniejsza wielkości poszczególnych segmentów.

PAMIĘĆ NATYWNA (OFF HEAP)



Rysunek 3. Struktura pamięci natywnej

O ile sterta jest fragmentem pamięci bliższym (w sposób bardziej lub mniej świadomy) programistom, o tyle pozostałe obszary wykorzystuje już głównie JVM. Oczywiście są w Javie dostępne techniki pozwalające nam na bezpośrednie zarządzanie pamięcią natywną, jednak wykraczają one zdecydowanie poza interesujący nas w tym artykule zakres. Struktury natywne poznamy na postawie najnowszej obecnie specyfikacji, czyli Javy 8.

Cały obszar off-heap możemy podzielić na dwie grupy. Pierwszą będą stanowiły obszary współdzielone, a drugą obszary autonomiczne dla każdego wątku działającego w naszej maszynie. Jedną z podstawowych zmian, jakie zaszły w stosunku do poprzedniej specyfikacji JVM, jest zastąpienie przestrzeni PermGen przez Metaspace. O ile sama rola została zbliżona, o tyle sposób implementacji zmienił się diametralnie. Przestrzeń Metaspace (inaczej określana jako Method Area) możemy traktować jako swoisty „magazyn" obiektów wewnętrznych Wirtualnej Maszyny. Klasy, metody, pola wykorzystywane na każdym kroku są serwowane właśnie z tej przestrzeni, a dokładnie we fragmencie opisanym na schemacie jako „Field & Method Data". Natomiast wszelkie literały znakowe, stałe czy referencje także stanowiące zawartość Metaspace są umiejscowione w segmencie „Run-time Constant Pool", który jest run-time'owym odzwierciedleniem tablicy stałych (The Constant Pool) zapisywanych przez kompilator w plikach class. Ostatnia grupa znajdująca się w Metaspace to kody klas, składowane tam przez JVM. Kolejnym istotnym segmentem off-heap jest code cache. Jest to przestrzeń kluczowa z punktu widzenia omówionego wcześniej kompilatora JIT. To właśnie tam przechowywane są optymalizowane i kompilowane kody. Ostatni współdzielony obszar stanowi pamięć natywna, która może być przy pomocy pewnych zaawansowanych technik wykorzystywana także wprost przez programistę. Jeżeli w naszej aplikacji występują na przykład wyjątkowo duże tablice liczb lub znaków, możemy samodzielnie zarządzać nimi właśnie w zakresie tej przestrzeni, co pozwoli zdjąć ciężar z garbage collectora, dla którego pamięć natywna leży daleko poza zakresem zainteresowań.

Pozostaje nam do omówienia jeszcze pamięć wykorzystywana przez poszczególne wątki żyjące w Maszynie Wirtualnej. Każdy z nich wykorzystuje dwa segmenty: program counter i stos (stack). Pierwszy z nich zawiera referencję do aktualnie wykonywanej instrukcji (opcode), wskazującej na odpowiedni fragment przestrzeni Method Area. Drugi to znany nam wszystkim stos. Jest to nic innego jak zwykła lista LIFO (Last In First Out) zawierająca ramki (frame) wywołań. Każda ramka zawiera między innymi tablicę zmiennych lokalnych oraz wartość zwracaną. I na tym kończy się opis struktury pamięci wykorzystywanej przez JVM.

PODSUMOWANIE

Złożoność tematu budowy i zasad funkcjonowania JVM jest wyjątkowo duża, co w sposób oczywisty nie pozwala na jej głębokie omówienie na zaledwie kilku stronach artykułu. Mam natomiast nadzieję, że lektura tego tekstu w jakimś stopniu przybliżyła i usystematyzowała wiedzę w zakresie architektury i podstawowych procesów zachodzących w Wirtualnej Maszynie Javy. Ja ze swojej strony mogę natomiast zapewnić, że poświęcenie kilku chwil na pozyskanie wiedzy z tego obszaru zdecydowanie w pozytywny sposób odbije się na jakości przyszłej pracy z systemami uruchamianymi w ramach JVM.

W kolejnym artykule z tego cyklu omówimy jeden z najważnieszych mechanizmów stanowiących o sile Javy – czyli automatyczne zarządzanie pamięcią, i przeanalizujemy dostępne algorytmy garbage collectora.

Jakub Kubryński jk@codearte.io

Od ponad 11 lat zawodowo zajmuje się oprogramowaniem. Częsty prelegent na branżowych konferencjach takich jak GeeCON, 33rd Degree, JDD, 4Developers czy Confitura, której jest organizatorem, Współzałożyciel startupu DevSKiller.com dostarczającego innowacyjną platformę oceny kompetencji programistów. Związany z software house Codearte. Trener w firmie Bottega, gdzie prowadzi szkolenia m.in.z wydajności, monitorowania i skalowalnej architektury systemów IT.