JAVA4 Gdańsk

JVM / GC / CLI

7 - 8.09.2017

O mnie

Maciej Rzepiński
maciej.rzepinski@gmail.com
http://**mrzepinski.pl**
Slack: @**maciej.rzepinski**

O Tobie

- Imię i nazwisko?
- * Twoje doświadczenie z IT?
- * Dlaczego IT i dlaczego JAVA?
- * Czego oczekujesz od kursu?
- * Co chcesz robić po kursie?
- * Firma IT “marzenie”?

* Dla chętnych.

JAVA?

https://go.java

Inne języki?

JVM = WORA

- Akronim dla
  Java Virtual Machine
- aplikacja napisana w C++
- “Write Once, Run Anywhere” – WORA
- JAVA bytecode

JVM
języki

- Java
- Scala
- Clojure
- Groovy
- AspectJ
- Kotlin
- ...

JVM - implementacje

Powstają na bazie specyfikacji języka - http://docs.oracle.com/javase/specs/jvms/se8/html/

- Oracle HotSpot (oficjalna)
- OpenJDK
- IBM J9
- Azul Zing
- ...

Oracle HotSpot JVM

elementy

- środowisko uruchomieniowe (JRE)
- kompilator kodu (JIT)
- interpreter bytecode
- “odśmiecacz / zarządca pamięci” (GC - Garbage Collector)
- narzędzia JDK
- debugging
- Hot Swap (tylko dla metod)

Oracle HotSpot JVM

zadania

- parsowanie argumentów z linii komend
- zarządzanie cyklem życia VM
- ładowanie klas
- interpretacja kodu bajtowego
- przechwytywanie wyjątków
- synchronizacja i zarządzanie wątkami
- ustawienie zmiennych środowiskowych (CLASSPATH)
- wybranie głównej klasy do wykonania

Argumenty linii poleceń

- standardowe: -server, -classpath (-cp)
- niestandardowe: -Xms128m, -Xmx512m
- programistyczne: -XX:+AggressiveOpts

Moje przykłady kodu / zadania

https://bitbucket.org/mrzepinski/java4gdajvm
LUB
https://goo.gl/2BuQHy

Twój kod

1. Załóż konto w jednym z poniższych serwisów:
   - Github - https://github.com
   - Bitbucket - https://bitbucket.org
2. Utwórz repozytorium Git i nowy projekt w Intellij na bazie linka do utworzonego repozytorium.
3. Wyślij informację o utworzonym repozytorium (link) do prowadzącego zajęcia ;)
4. Wykorzystaj stworzone repozytorium do zapisywania postępów swojej pracy nad kodem.

Zadania #1

javac / javap / java

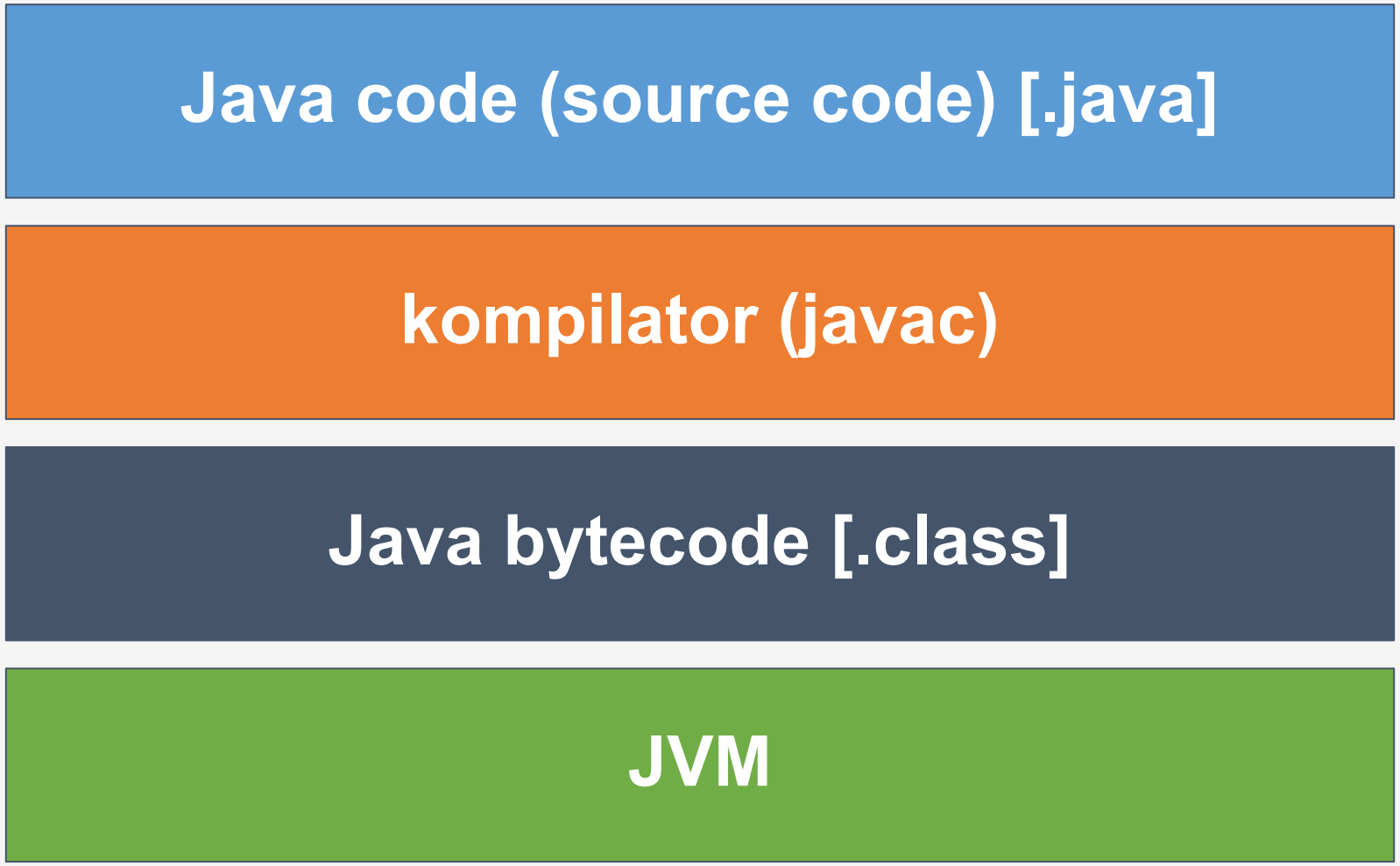
Zadania #1

javac / java

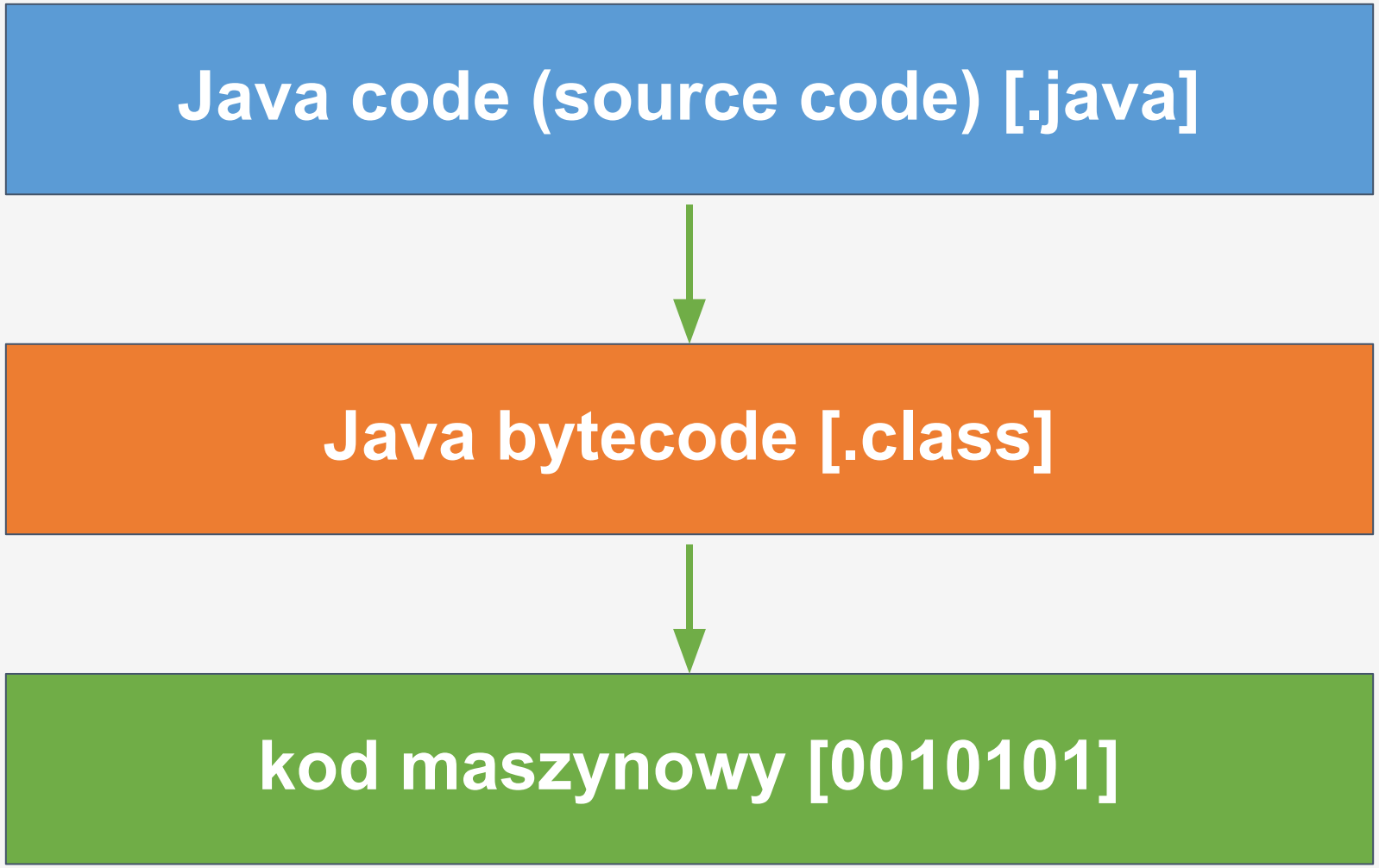
1. Napisz klasę z metodą main, która będzie przyjmować argument [N] typu int od użytkownika i wyświetli napis “Hello World!” N razy.
2. Wykorzystaj polecenia **javac** oraz **java** do kompilacji i uruchomienia swojej klasy.
3. Wykorzystaj polecenie **javap** do dekompilacji klasy.
4. Postaraj się poprawić metodę main tak, by podanie argumentu było opcjonalne, a napis “Hello World!” wyświetlił się co najmniej raz.
5. * Rozszerz metodę main tak, by przyjmowała ona polecenia od użytkownika w sposób ciągły i kończyła pracę dopiero po podaniu przez użytkownika polecenia “EXIT”. Możesz spróbować napisać coś więcej niż “Hello World!”, ale **pamiętaj o powyższym założeniu**.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

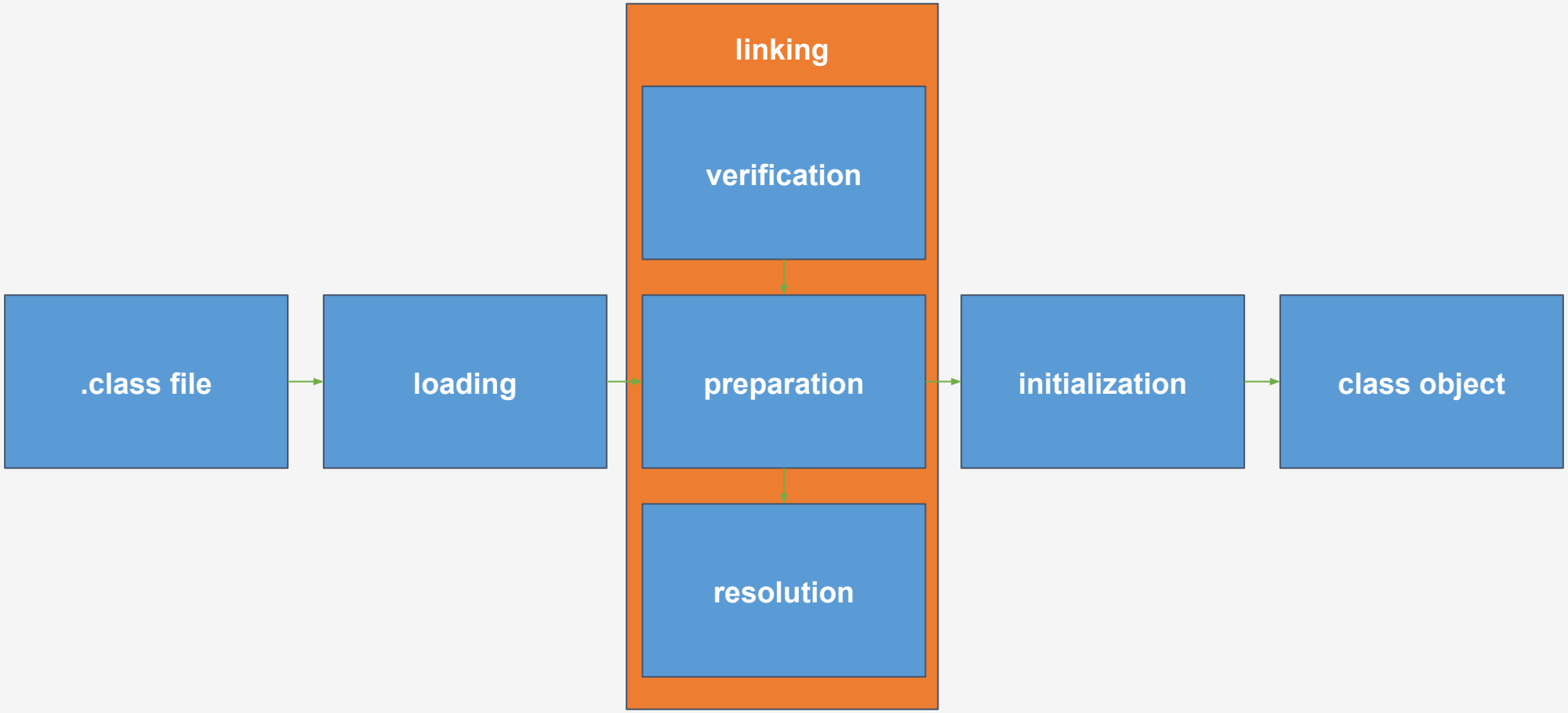
Java code → JVM



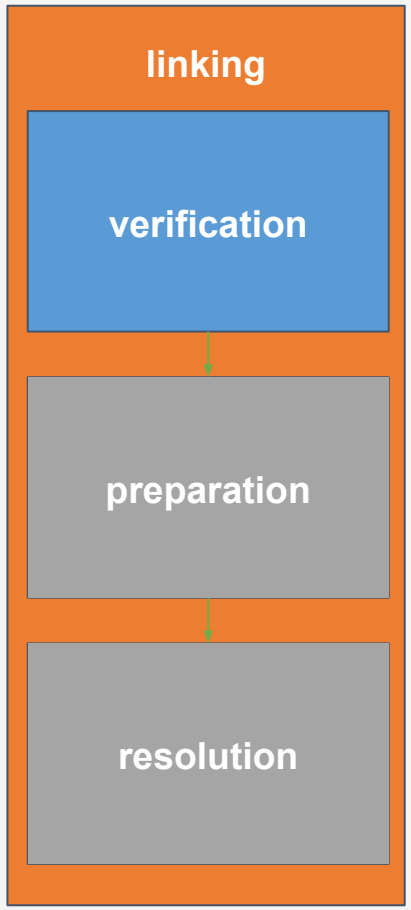
Java code → kod maszynowy



Ładowanie klasy
(ClassLoader)

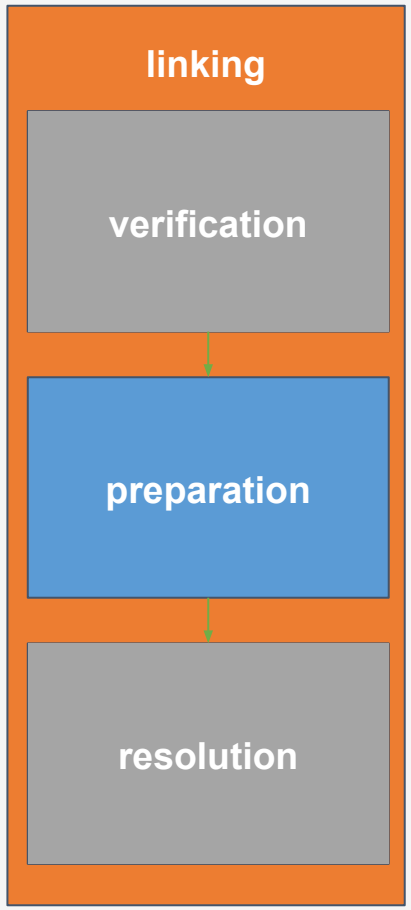


Ładowanie klasy
(ClassLoader)



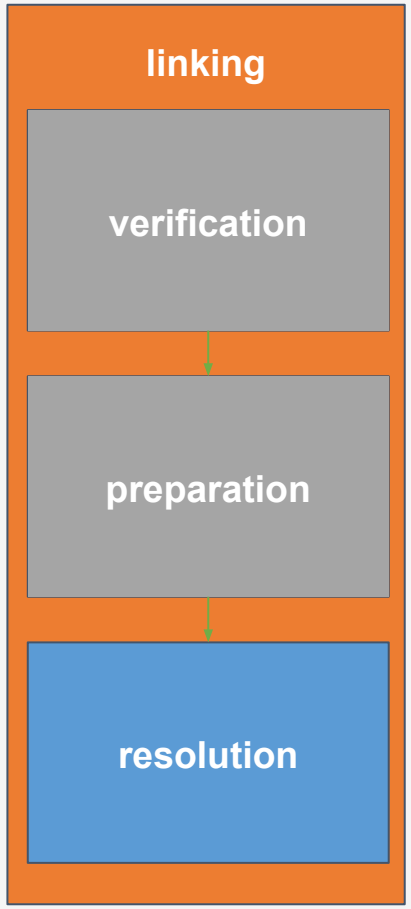
- klasa / metoda nie nadpisuje **final**
- klasa używa właściwych **metod dostępu** (np. private)
- metody mają prawidłową **liczbę parametrów**
- **bytecode** nie manipuluje stosem w złośliwy sposób
- zmienne są **zainicjalizowane przed odczytem**
- zmienne mają **prawidłowe typy**

Ładowanie klasy
(ClassLoader)



- wszystkie wymagane zależności są załadowane do **Constant Pool Table**
- inicjalizacja **pól statycznych** (tworzenie / domyślna wartość)

Ładowanie klasy
(ClassLoader)



- każdy typ, do którego odnosi się nasza klasa, jest **gotowy do wykorzystania** (a nie tylko załadowany)
- obiekt klasy może być **stworzony**

JDK vs JRE vs JVM

JIT - Just In Time compiler

- bytecode jest interpretowany = jego wykonanie jest wolne
- podczas pracy programu następuje kompilacja bytecodu do kodu maszynowego w tzw. “locie”
- możliwa jest ciągła optymalizacja
- “gorące metody / punkty” są optymalizowane po osiągnięciu progu wywołań - zamiana na kod maszynowy (Oracle HotSpot = gorące punkty)

JIT - Just In Time compiler

sposoby działania

- 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)

= ROZGRZEWANIE APLIKACJI

Zadania #2

JAR / JavaDoc

Zadania #2

tworzenie archiwum JAR / praca z JavaDoc

1. Bazując na kodzie aplikacji z zadania pierwszego, stwórz archiwum JAR i sprawdź jego działanie.
2. Napisz prostą grę logiczną / tekstową (odgadywanie słów, mini RPG, przechodzenie labiryntu, Sudoku, Hanoi, test wiedzy itp.) Wykorzystaj programowanie obiektowe do zbudowania logiki swojej gry. Klasa z metodą main ma służyć wyłącznie do uruchomienia gry i przyjmowania poleceń gracza.
3. Używając znaczników JavaDoc (https://binfalse.de/2015/10/05/javadoc-cheats-sheet/), napisz komentarze dokumentujące kod Twojej gry.
4. Wykorzystaj narzędzie javadoc do wygenerowania dokumentacji.
5. * Wyślij swoją grę do prowadzącego zajęcia / kolegi / koleżanki w formie archiwum JAR z krótkim opisem gry oraz wygenerowaną dokumentacją kodu.
6. * Napisz testy symulujące grającego użytkownika, jednocześnie sprawdzające przypadki brzegowe w Twojej grze.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Kończymy na dzisiaj!
Do zobaczenia jutro ;)

Piątek, piąteczek, piątunio ;)

Szybka powtórka

- JVM
- JDK vs JRE vs JDK
- Java code -> JVM
- JIT compiler

Zadania #3

VisualVM

Zadania #3

VisualVM

1. Wykorzystując narzędzie VisualVM, przetestuj działanie klasy MemoryManagement.
2. Zrefaktoruj kod klasy MemoryManagement tak, by przyjmował on parametry wejściowe z linii komend.
3. Stwórz archiwum JAR.
4. Podaj takie parametry wejściowe, które spowodują, że zabraknie pamięci (obserwuj działanie w VisualVM) - w konsoli IntelliJ powinien pojawić się wyjątek: java.lang.OutOfMemoryError
5. * Ustaw maszynę wirtualną Javy tak, by wykorzystywała niewielką ilość zasobów pamięci (-Xms, -Xmx). Przetestuj działanie aplikacji w takich warunkach.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Garbage Collector

- “odśmiecacz” / zarządca pamięci
- Java zarządza się “sama” w przeciwieństwie np. do języka C
- gdy obiekt nie jest już potrzebny, GC rozpoznaje taką sytuację i zwalnia zajmowaną przez niego pamięć

Garbage Collector

plusy

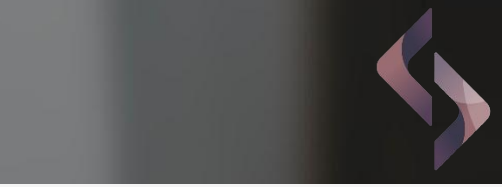
\+ gotowe algorytmy, które działają w tle i efektywnie zwalniają nieużywaną pamięć

= deweloper nie musi zaprzątać sobie głowy czyszczeniem pamięci

Garbage Collector

minusy

- uruchomienie GC powoduje pauzę w działaniu aplikacji - “Stop the world”
- nie jesteśmy w stanie wpłynąć oraz przewidzieć kiedy GC zostanie uruchomiony (mimo System.gc() oraz Runtime.gc())
- ! algorytmy GC działają tak jak zostały zaprojektowane, a nasz kod i tak może powodować wycieki pamięci



Garbage Collector

algorytmy

skalarny

- zliczanie referencji
- problem przy referencjach cyklicznych

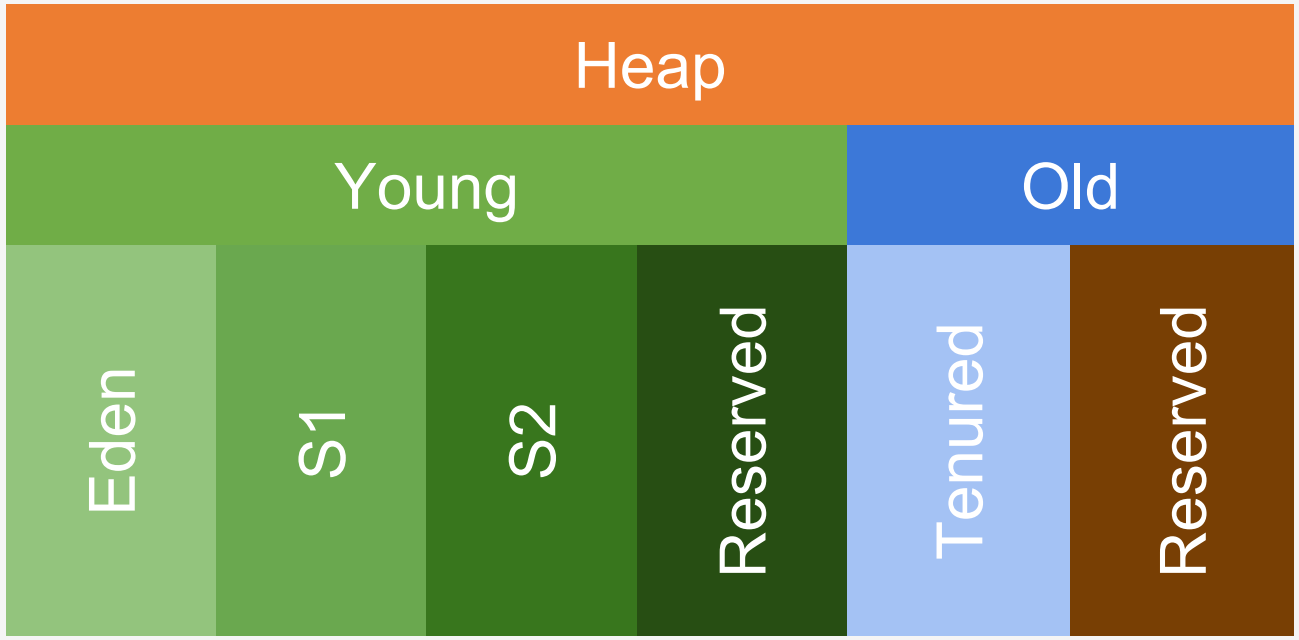
wektorowy

- określanie korzenia
- oznaczanie żywych obiektów
- usuwanie martwych obiektów

Rodzaje pamięci JVM

- heap (sterta)
  podlega GC, zawiera tworzone obiekty
- off-heap (pamięć natywna)
  nie podlega GC, zawiera wewnętrzne struktury JVM umożliwiające jej funkcjonowanie

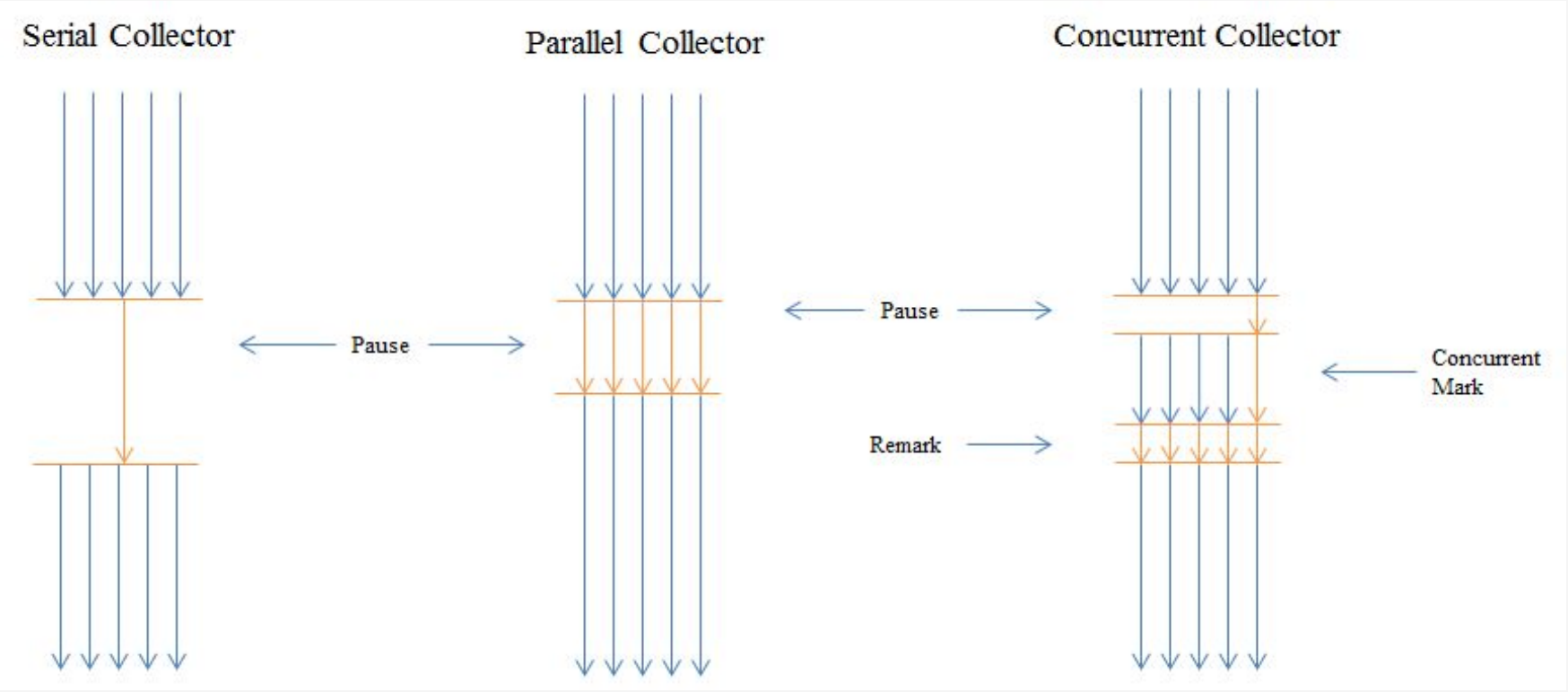
Heap (sterta)
(infant mortality) = “śmiertelność niemowląt”



- świadomy podział na generacje młodą (young) i starą (old / tenured)
- stosowanie różnych strategii
- young dzieli się na kilka przestrzeni:
  - eden
  - survivor
    - S1
    - S2
- dodatkowa pamięć zarezerwowana ze względu na dynamikę obszarów
- old praktycznie nie odwołuje się do young
- kopiowanie zapobiega defragmentacji (zwłaszcza, gdy powstaje dużo nowych obiektów)

Garbage Collector

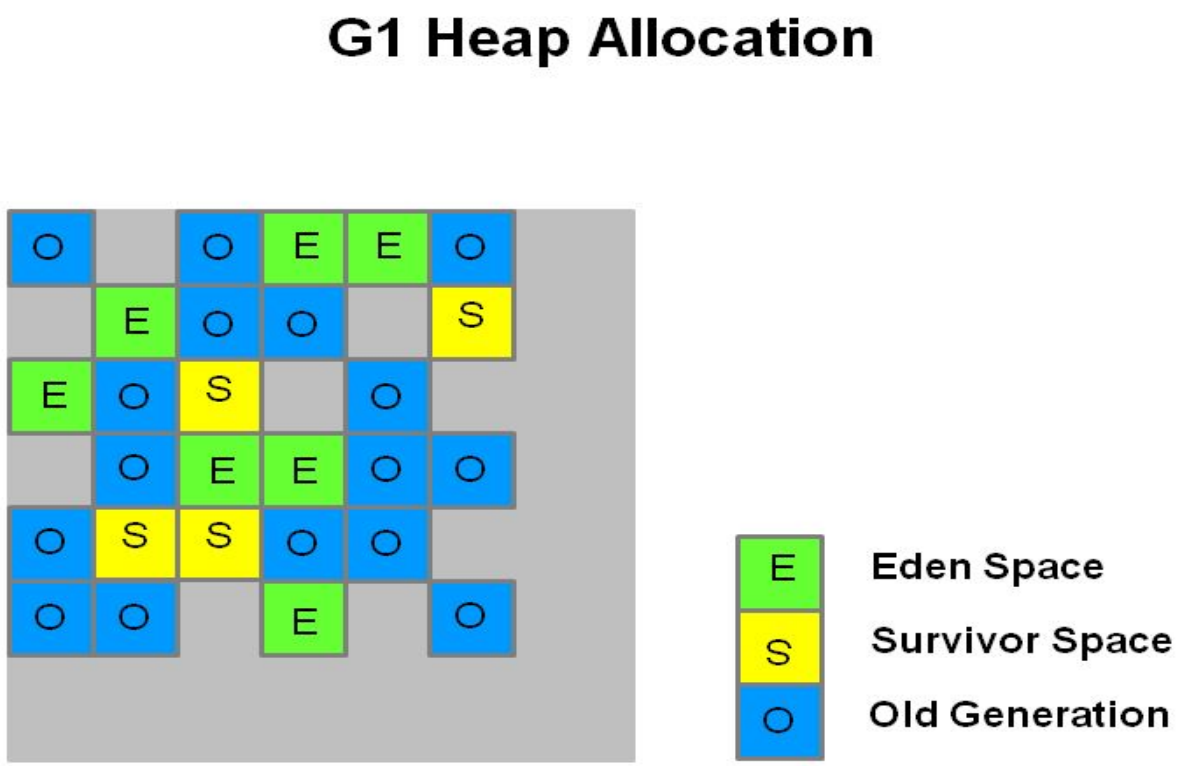
rodzaje



- Serial
- Parallel (Throughput GC)
- CSM (Concurrent Mark-Sweep)

Garbage Collector

rodzaje



- G1 (Garbage First)

Garbage Collector

rodzaje

- **Serial**
  - sekwencyjny
  - “Stop the world”
- **Parallel (Throughput GC)**
  - domyślny
  - wielowątkowa ulepszona wersja serial
- **CSM (Concurrent Mark-Sweep)**
  - concurrent - **równoległe**, mark - **oznaczanie**, sweep - **obiektów do usunięcia**
  - zaprojektowany do zminimalizowania “Stop the world” - działa cały czas razem z aplikacją
- **G1 (Garbage First)**
  - wprowadzony w JDK7
  - domyślny w Java9
  - najpierw zwalnia prawie puste duże obszary pamięci (stąd nazwa)
  - podział na małe obszary 1 - 32 MB

Zadania #4

VisualVM

Zadania #4

VisualVM

1. Wykorzystując grę stworzoną na poprzednich zajęciach, napisz kod symulujący działanie użytkownika.
2. Stwórz archiwum JAR.
3. Uruchom grę z niewielkimi zasobami pamięci.
4. Wykorzystując narzędzie VisualVM, sprawdź działanie swojej aplikacji pod kątem wykorzystania pamięci oraz czasu użycia procesora.
5. * Zoptymalizuj swoją grę - w tle cały czas korzystaj z VisualVM i testuj działanie aplikacji.
6. * Do kodu swojej gry dodaj “memory leak” i znajdź go w VisualVM.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Podsumowanie

- JVM
- JDK vs JRE vs JDK
- Java code -> JVM
- JIT compiler
- podział pamięci
- GC
- narzędzia CLI

Pytania?

ANKIETY!

KONIEC

Dziękuję ;)