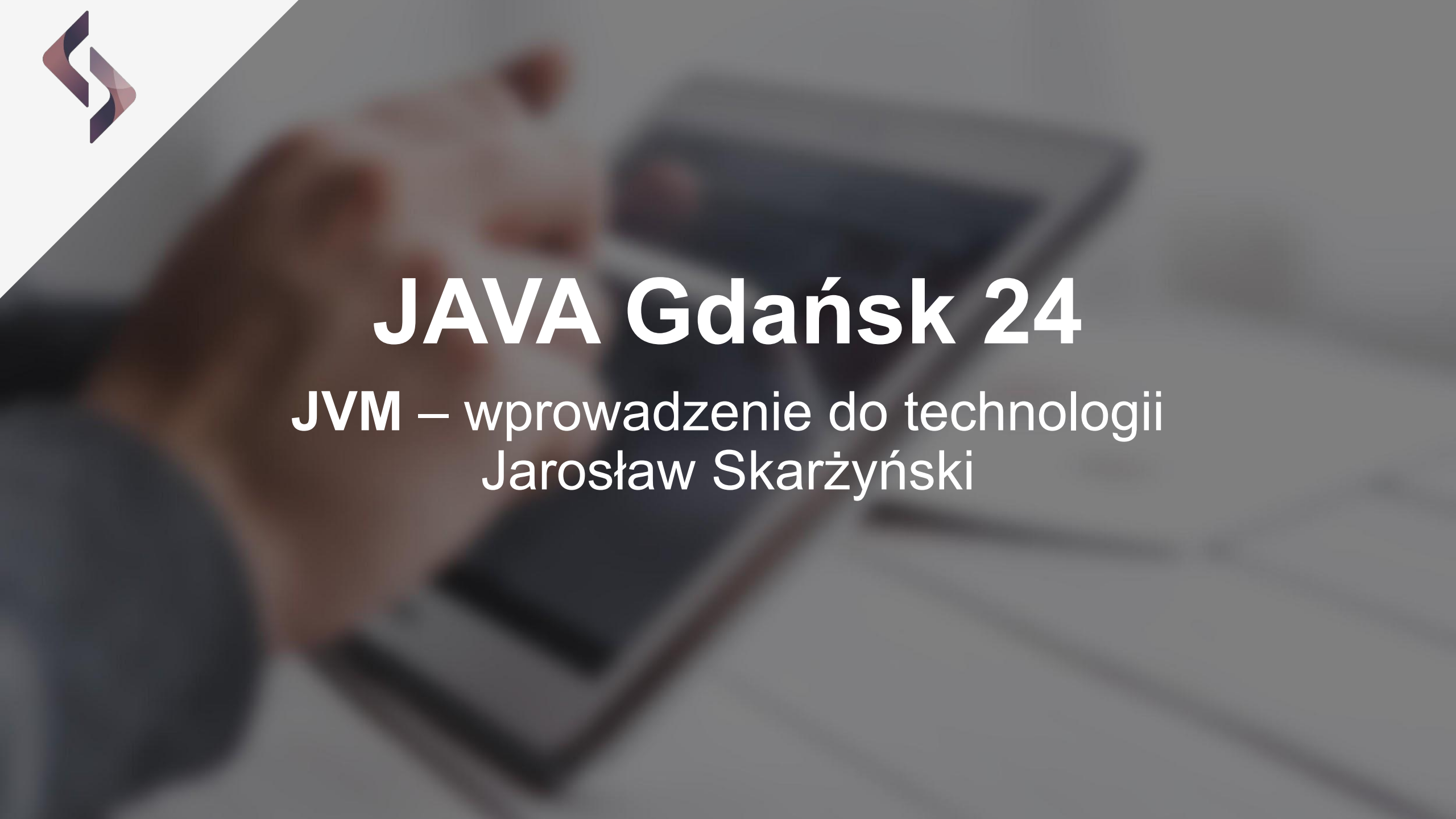


JAVA Gdańsk 24

JVM – wprowadzenie do technologii

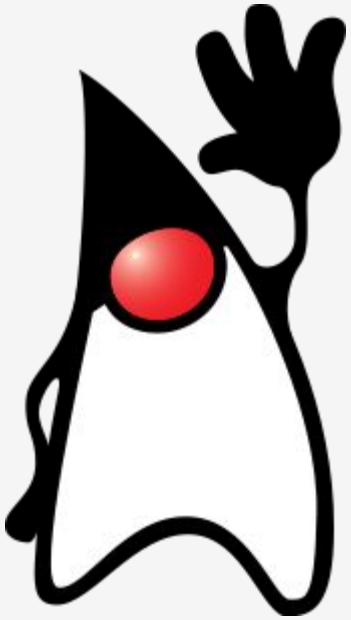
Jarosław Skarżyński

Przygotowania

1. (Opcjonalnie) Zainstaluj Notepad++ (https://notepad-plus-plus.org/download/) - zignoruj jeżeli już masz lub używasz innego edytora tekstowego
2. Otwórz okno poleceń swojego systemu operacyjnego, jak to zrobić możesz sprawdzić np. tutaj:https://tutorial.djangogirls.org/pl/intro_to_command_line/
3. Sprawdź czy Java jest prawidłowo zainstalowana, polecenie: **java -version**
4. Sprawdź czy działa polecenie: **javac -version** i czy wersja jest taka sama jak w pkt 3
5. Jeżeli polecenia powyżej nie działają sprawdź czy masz pobraną i zainstalowane **JDK** (a nie tylko **JRE!**) w wersji **8**.
6. Jeżeli nie masz **JDK** pobierz i zainstaluj. Pobrać można ze strony https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
7. Jeżeli nadal nie możesz wykonać poleceń z pkt 3 i/lub 4 wykonaj instrukcje poniżej:
   a. Wykonaj polecenia z linku żeby znaleźć zmienną systemową Path: https://www.java.com/pl/download/help/path.xml
   b. Dodaj do zmiennej systemowej Path nowy wpis [JAVA_HOME]\bin – gdzie [JAVA_HOME] to katalog z JDK, np.: C:\Program Files\Java\jdk1.8.0_172\bin
   c. Zamknij i otwórz ponownie okno poleceń systemu, wróć do pkt 3
8. Zainstaluj dodatkowe pluginy do programu **JVisualVM**:
   a. przejdź do katalogu [JAVA_HOME]\bin – gdzie [JAVA_HOME] to katalog z **JDK**
   b. znajdź i uruchom program **jvisualvm.exe**
   c. w menu górnym kliknij Tools -> Plugins
   d. przejdź do zakładki: Available Plugins i zaznacz dwa pluginy: **Visual GC** i **VisualVM - MBeans**
   e. kliknij przycisk: Install w lewym dolnym rogu
   f. zainstalowane pluginy powinny pojawić się w zakładce: Installed

Rozkład jazdy

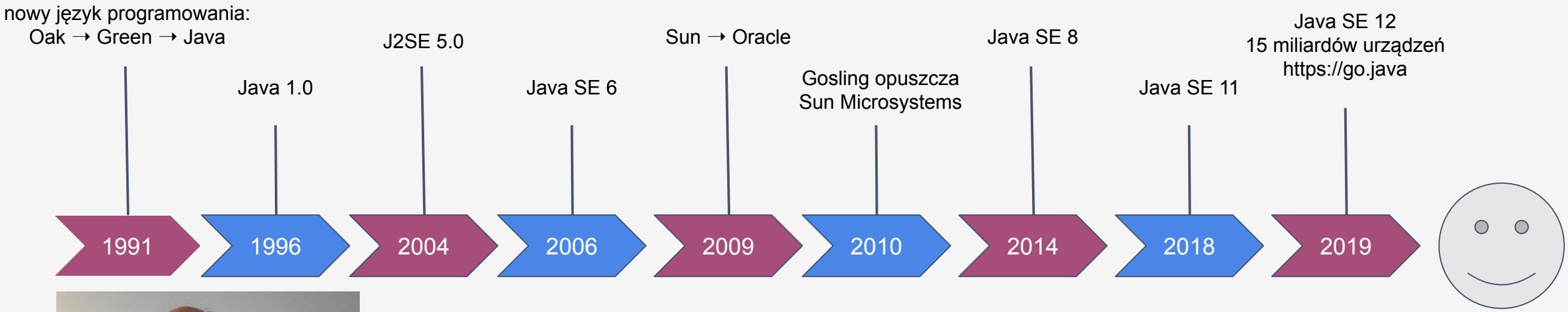
09:00 - trochę historii + podstawowe właściwości Javy
09:30 - **JVM** - classloader + execution engine
10:30 - przerwa krótka
10:40 - archiwum jar + javadocs
12:40 - przerwa długa
13:00 - konfiguracja aplikacji
14:00 - **VisualVm** + ćwiczenia
14:30 - przerwa krótka
14:40 - **VisualVm** + ćwiczenia cd.
15:00 - **JVM** - zarządzanie pamięcią



JAVA

trochę historii, podstawowe założenia

Rys historyczny



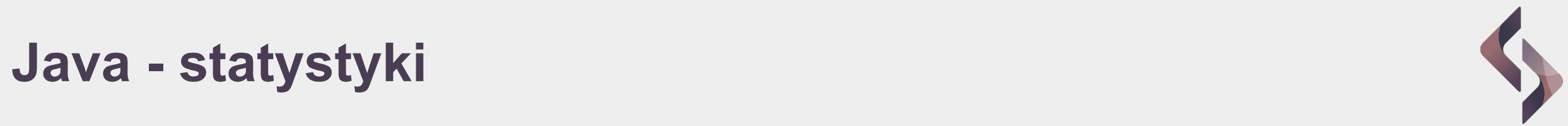
James Gosling
Father of Java

Sun microsystems

ORACLE®

http://oracle.com.edgesuite.net/timeline/java/

Java - statystyki



- 15 miliardów urządzeń
- 5 miliardów kart Java SmartCard w użyciu
- 3 miliardy telefonów komórkowych (Android!)
- 125 milionów urządzeń telewizyjnych
- komputery lokalne, tablety, drukarki, odtwarzacze Blu-ray
- roboty, automatyczne samochody, transport publiczny
- Twitter&Java, Netflix&Java
- komputery typu mainframe,
- nr 1 jako platforma programistyczna w chmurze

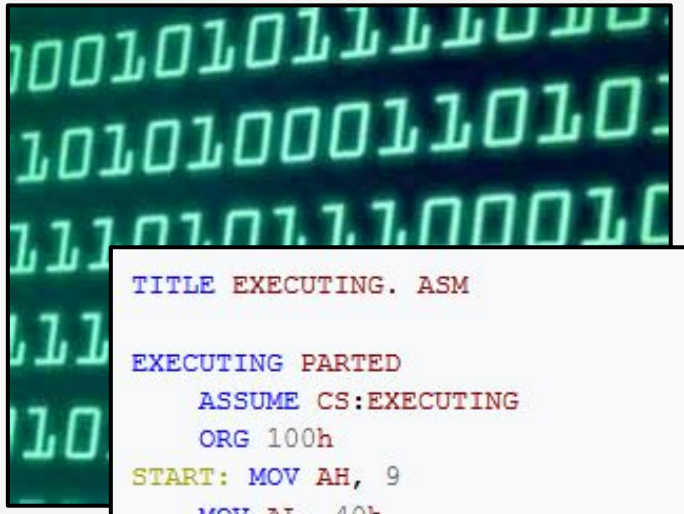
Źródło: https://www.java.com/pl/about
https://go.java

- 10 milionów programistów
- 5 milionów studentów
- najczęściej wybierany język przez programistów
- nr 1 jako platforma programistyczna

Podstawowe założenia

- **WORA** – **Write Once Run Anywhere** - wieloplatformowość
- sieciowość i obsługa programowania rozproszonego
- prosty, obiektowy język, łatwy do przyswojenia przez programistów
- niezawodność i bezpieczeństwo
- wydajność, wykorzystanie procesorów wielordzeniowych

Jak to „widzi” procesor/maszyna



język maszynowy - zestaw rozkazów procesora, w którym zapis programu wyrażony jest w postaci liczb binarnych stanowiących rozkazy oraz ich argumenty (pliki *.com i *.exe dla Windows). Język maszynowy jest nieprzenośny, ponieważ każda architektura procesora ma swój własny model programowy, a więc m.in. listę rozkazów maszynowych.

```
TITLE EXECUTING. ASM

EXECUTING PARTED
    ASSUME CS:EXECUTING
    ORG 100h
START: MOV AH, 9
    MOV AL, 40h
    MOV AH, 10d
    MOV CX, 11D
    INT 10h
    MOV AH, 4ch
    INT 21h
EXECUTING ENDS
END START ; / PLIK OTRZYM
```

język niskiego poziomu – język programowania umożliwiający zapis rozkazów maszynowych za pomocą stosunkowo prostych oznaczeń symbolicznych, np. języki asemblera (zwyczajowo asemblery) to rodzina języków programowania niskiego poziomu, których jedno polecenie odpowiada zasadniczo jednemu rozkazowi procesora. Kod nieprzenośny, zależy od architektury procesora. Asemblacja – generowanie kodu maszynowego

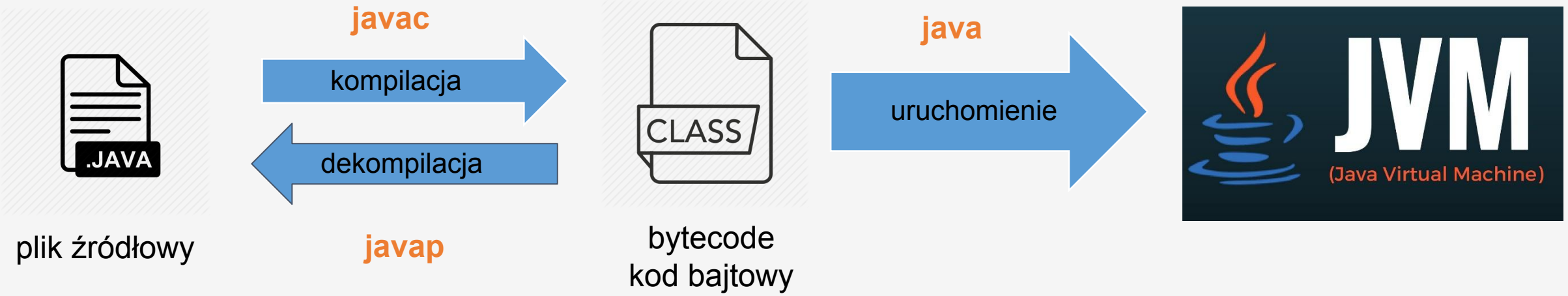
```
int x;
scanf("%d", &x);
switch(x)
{
  case 0:
  case 1:
    printf("jeden");
    break;
  case 2:
    printf("dwa");
    break;
  default:
    printf("coś innego");
    break;
}
```

język wysokiego poziomu – typ języka programowania, którego składnia i słowa kluczowe mają maksymalnie ułatwić rozumienie kodu programu dla człowieka, tym samym zwiększając poziom abstrakcji i dystansując się od sprzętowych niuansów. Przykłady: C(?), C++, Pascal, Java. Kod przenośny/nieprzenośny. Kompilacja – tłumaczenie kodu źródłowego na maszynowy.

źródło: https://pl.wikipedia.org

Java - język programowania wysokopoziomowego

- **kod źródłowy** - kod napisany przez programistę w plikach z rozszerzeniem ***.java*** (np.: HelloWorld.java).
  - jeden plik *.***java** = dokładnie jedna publiczna klasa
  - w pliku mogą znajdować się definicje innych (niepublicznych) klas
  - nazwa pliku = nazwa publicznej klasy zdefiniowanej w pliku (uwaga: wielkość liter ma znaczenie!)
- **bytecode** - kod bajtowy (plik z rozszerzeniem ***.class***)
  - powstaje podczas kompilacji kodu źródłowego
  - jest interpretowany i tłumaczony na konkretne rozkazy procesora przez **JVM**
- wirtualna maszyna Javy (ang. **Java Virtual Machine**, w skrócie **JVM**) wykonuje bytecode Javy poprzez interpretację / kompilację na kod maszynowy



Podstawowe komendy linii poleceń

- **javac <options> <source files>** – kompilacja kodu źródłowego, np.: **javac -d out Calculator.java CalculatorCli.java**
  Przykładowe opcje (**javac -help** - opis wszystkich opcji):
  - -cp <path> - lista katalogów (lub plików jar) gdzie znajdują się klasy zależne, np.: **-cp out;lib\math.jar;resources**
  - -d <directory> - nazwa katalogu gdzie ma zostać stworzony plik .class
  - -deprecation - wyświetlanie miejsc gdzie jest wykorzystywany przestarzały kod
  - -source <release> - z jaką wersją jest kompatybilny kod źródłowy
  - -target <release> - w jakiej wersji ma być generowany plik z kodem bajtowym
  - javac -version - wyświetla wersję kompilatora

- **java [-options] class [args...]** - uruchomienie programu, np.: **java -cp out CalculatorCli ADD 10 77**
  Przykładowe opcje (java -help - opis wszystkich opcji):
  - -classpath <path>, -cp <path> - gdzie należy szukać klas powiązanych z klasą uruchamianą
  - -D<name>=<value> - ustawienie zmiennych systemowych
  - -client, -server – wybór Java HotSpot Server VM lub Client VM
  - java -version - wyświetla wersję środowiska Java (w tym wersję VM)

- **javap <options> <classes>** - dekompilacja pliku *.class do postaci źródłowej, np.: **javap -p Calculator.class**
  Przykładowe opcje (javap -help - opis wszystkich opcji):
  - -public, -protected, -package, -private - poziom widoczności klas i metod które mają być wyświetlone
  - -v -verbose - wyświetl wszystkie informacje
  - -sysinfo - pokazuje informacje systemowe, np.: ścieżkę, datę zmiany, rozmiar

- **jdeps <options> <classes...>**- przedstawia zależności pomiędzy klasami np.: **jdeps -v -cp out pl.sda.calculator.CalculatorCli.class**
  Przykładowe opcje (jdeps -help - opis wszystkich opcji):
  - -s -summary - krótkie podsumowanie
  - -v -verbose - rozbudowany diagram zależności
  - -R –recursive - rekursywnie wyświetl wszystkie zależności

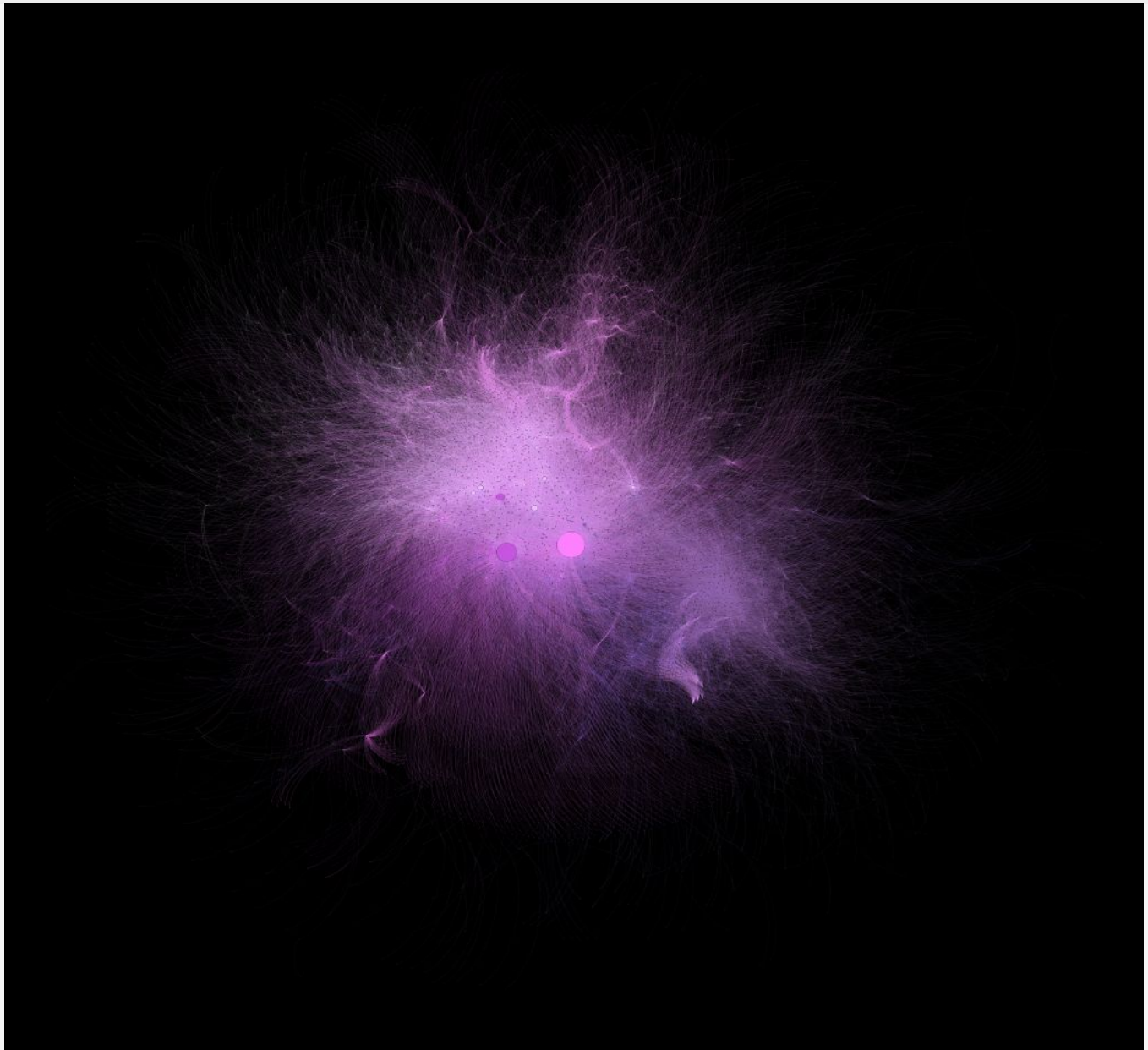
Zadania

#calculator

1. Tworzymy prosty program - kalkulator **(bez użycia IntelliJ IDEA!)**
   a. Utwórz nowy katalog **calculator** - to będzie nasz katalog bazowy
   b. Stwórz plik ***Calculator.java*** w katalogu bazowym, utwórz metody które będą: dodawać i mnożyć - jako argumenty metody mają przyjąć dwie zmienne typu **int**.
   c. Stwórz plik ***CalculatorCli.java*** w katalogu bazowym, w metodzie *main()* stwórz instancję klasy ***Calculator*** i wywołaj kilka operacji wypisując do konsoli ich wyniki
   d. Skompiluj obie klasy (polecenie **javac** - patrz poprzedni slajd), a następnie uruchom klasę ***CalculatorCli***
   e. * Użyj polecenia **jdeps** do sprawdzenia zależności dla ***CalculatorCli*** (co się stanie jak użyjemy flagi -R ?)
   f. * Spróbuj zdekompilować bytecode klasy ***Calculator*** poleceniem **javap**. Sprawdź datę ostatniej modyfikacji i rozmiar pliku.
2. Robimy porządki w kodzie:
   a. Z katalogu bazowego usuń pliki **.class*
   b. Przenieś pliki java do pakietów oraz dostosuj kod w obu klasach do zmian:
      pl.sda.calculator.*CalculatorCli*
      pl.sda.calculator.core.*Calculator*
   c. W katalogu bazowym dodaj katalog **out** (tu będą trafiać pliki **.class)* oraz katalog **src** (tu przenieś wszystkie pliki źródłowe)
   d. Spróbuj teraz skompilować (do katalogu **out**) obie klasy i uruchomić ***CalculatorCli***
   e. Wracamy do **Intellij IDEA** :) - kod piszemy w IntelliJ, kompilacja i uruchomienie nadal z wiersza poleceń
   f. Spróbuj otworzyć projekt z katalogu bazowego, dodaj metodę do odejmowania w ***Calculator*** i użyj jej w ***CalculatorCli***
   g. Skompiluj i uruchom program
   h. * Spróbuj zdekompilować bytecode klasy ***Calculator*** poleceniem **javap**.

* Dla chętnych.

Java Universe - jdeps + Gephi



Co się dzieje w czasie wykonywania programu Javy?

Wirtualna maszyna - **JVM** - składa się z trzech głównych podsystemów:

- **Classloader**
- **Runtime Data Area**
- **Execution Engine**

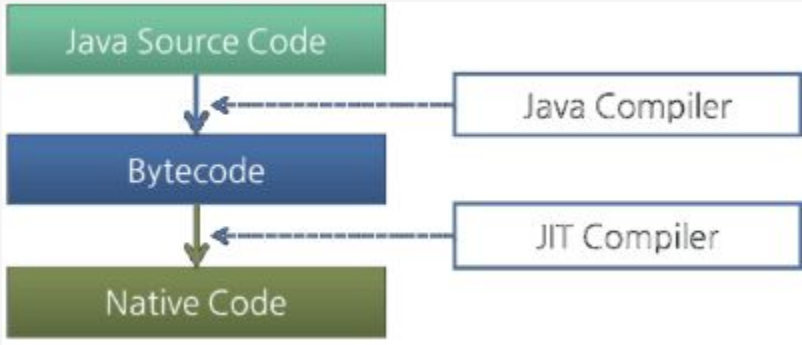
Wykonanie programu Javy krok po kroku:

1. Uruchomienie maszyny wirtualnej
2. **Classloader** wczytuje, sprawdza i inicjalizuje klasę
3. Dane potrzebne do działania klasy (statyczne i dynamiczne) są umieszczane w pamięci komputera przydzielonej wirtualnej maszynie zwanej **Runtime Data Area**
4. Uruchamiany jest **Execution Engine**, które wykorzystuje dane z **RDA**, interpretuje/kompiluje bytecode tworząc kod natywny (maszynowy)
5. Wykonanie kodu

ExecutionEngine

Kod klasy (bytecode), który został wczytany do **Runtime Data Area** jest wykonywany przy pomocy **Execution Engine**. Execution Engine czyta bytecode i wykonuje instrukcje krok po kroku. Zanim instrukcja zostanie wykonana musi być zmieniona w kod maszynowy zrozumiały dla procesora. Dzieje się to na dwa sposoby:

- **Interpreter** - czyta, interpretuje i wykonuje bytecode kawałek po kawałku. Jest to dosyć wolny proces.
- **JIT (Just-In-Time) compiler** - kompiluje bytecode do kodu maszynowego, który jest zapisywany w cache. Kompilacja może zająć trochę czasu, ale jest wykonywana tylko raz, a potem kod maszynowy może być wielokrotnie użyty.



źródło: https://www.cubrid.org/blog/understanding-jvm-internals/

Kompilacja za pomocą kompilatora **JIT** trwa dłużej niż praca interpretera. Dlatego jeśli kod ma być wykonany tylko raz lepiej go wykonać za pomocą interpretera niż kompilatora. **JVM** w trybie ciągłym analizuje wykonywany kod i deleguje wykonanie kodu do kompilatora **JIT** kiedy uzna że jest on wykonywany wystarczająco często. Taki kod nazywamy **HotSpot** – stąd nazwa **Hotspot VM.**

Classloader

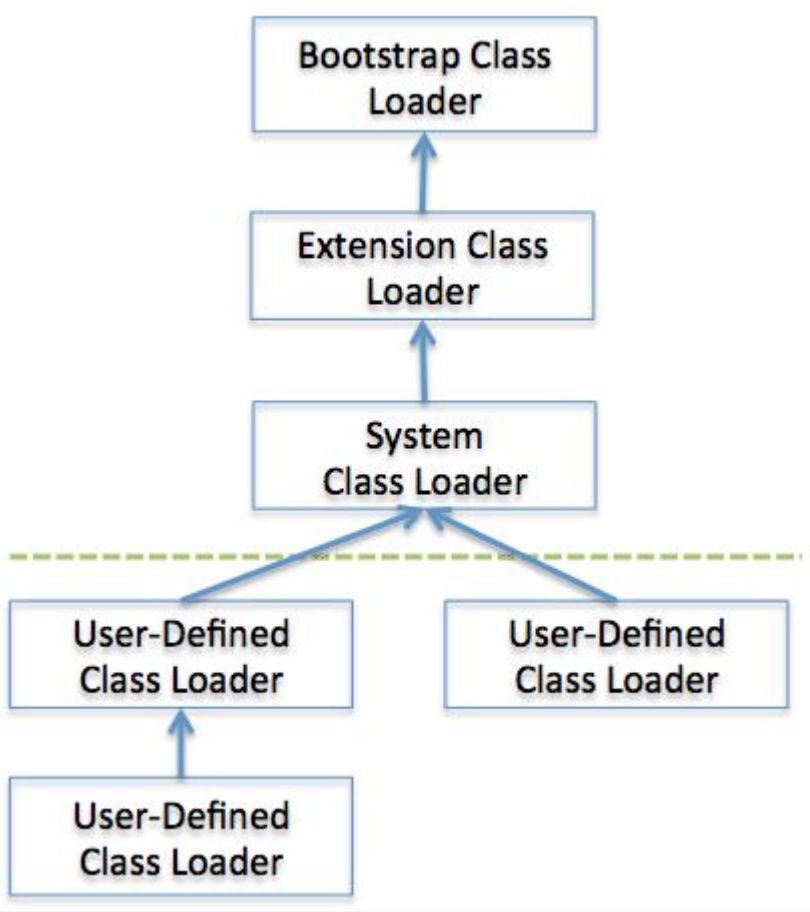
Klasy Javy ładowane są do pamięci poprzez mechanizm zwany **Classloaderem**, w sposób dynamiczny (czyli przy pierwszym użyciu klasy w programie, w czasie wykonania, a nie kompilacji).

Właściwości **classloadera**:

- struktura hierarchiczna – patrz diagram obok
- delegacja – najpierw sprawdź rodzica, potem u siebie
- ograniczona widoczność – classloader niżej w hierarchii może sprawdzić czy klasa jest dostępna u rodzica, ale rodzic nie może sprawdzić dziecka

Pracę **classloader'a** można podzielić na trzy części:

- **loading** - klasa jest wyszukiwana i ładowana do pamięci JVM – Method Area (nie tylko z plików!)
- **linking** - sprawdzana jest poprawność klasy pod kątem specyfikacji **JVM** i **JLS**, pola statyczne są alokowane w pamięci i inicjalizowane domyślnymi wartościami
- **initialization** - zmienne statyczne otrzymują zapisane w kodzie wartości, wykonywane zostają bloki statyczne



źródło: https://www.cubrid.org/blog/understanding-jvm-internals/

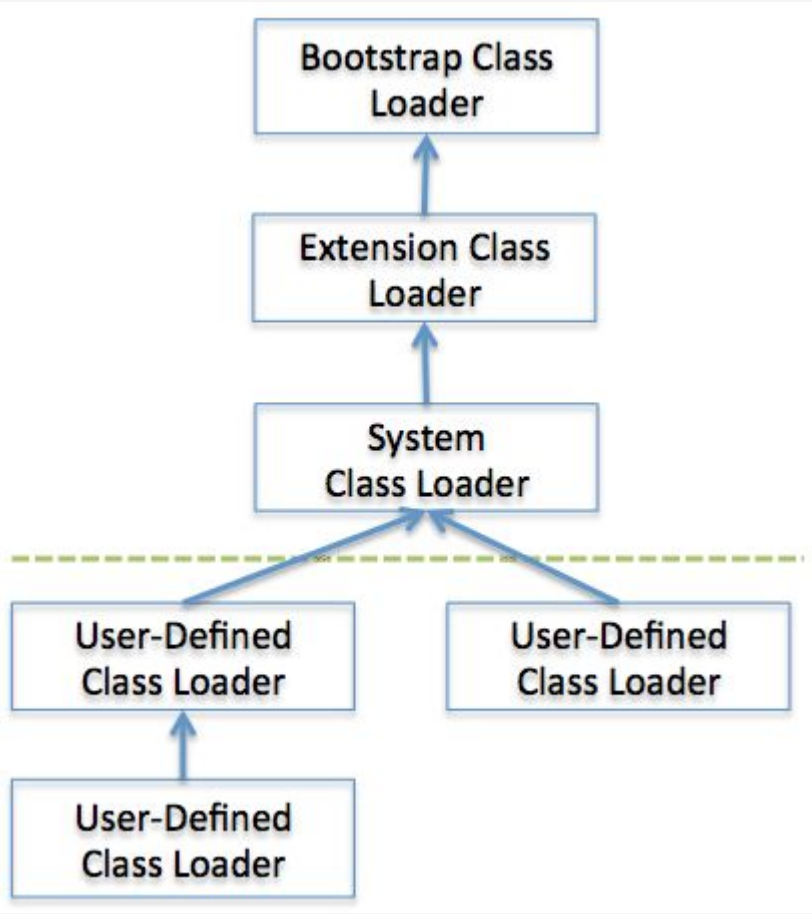
Classloader

Trzy podstawowe class loadery to:

- **Boostrap classloader -** występuje w każdej implementacji JVM. Opdowiedzialna za wczytywania klas z Java Api (z katalogu: *%JAVA_HOME%/**jre/lib**).* Jest zaimplementowana w języku natywnym (C, C++)
- **Extension classloader** – wczytuje klasy z katalogu z rozszerzeniami (*%JAVA_HOME%/**jre/lib/ext**)*
- **System/Application classloader** – odpowiedzialny za wczytywanie klas z **classpath** aplikacji. Możemy go konfigurować przez zmienną systemową **CLASSPATH**, która wskazuje gdzie szukać plików ***.class**.

Dodatkowo możemy dodać dowolną ilość własnych **classloader'ów**. **Classloader** to klasa napisana w Javie, każdy może napisać swoją i dodać do **JVM** (**User-Defined Class Loader**).

Tak się dzieje np. w serwerach webowych typu **Tomcat**. Tam każda webaplikacja ma własny **classloader** + jest jeden wspólny na cały serwer Tomcat.



źródło: https://www.cubrid.org/blog/understanding-jvm-internals/

Zadania - przygotowanie

#calculator

Do następnych zadań potrzebujemy kilku dodatkowych plików.

Należy je pobrać z repozytorium prowadzącego za pomocą komendy **git clone** lub przy pomocy **IntelliJ IDEA**

Link do repozytorium:
https://github.com/jaroslaw-skarzynski/calculator_resources

Instrukcja pobierania (klonowania) projektu ze zdalnego repozytorium przy pomocy IntelliJ IDEA:
https://goo.gl/kYzbEQ

Zadania

#calculator

3. Dodajemy zależności zewnętrzne
   a. Wyczyść katalog **out** i utwórz w katalogu bazowym katalog **lib**, następnie dodaj do niego plik **math.jar**
   b. W pliku znajduje się klasa **pl.sda.math.calculator**.***MathCalculator*** z metodami:
      public double *cos*(double a)
      public double *sin*(double a)
   c. Użyj nowego kalkulatora w klasie ***Calculator***, delegując do niej dwie podane wyżej metody
   d. Dodaj do ***CalculatorCli*** użycie nowych metod: *cos()* i *sin(),* następnie uruchom ***CalculatorCli***
   e. **Uwaga!** Podczas kompilacji i uruchamiania programu należy dodać do classpath ścieżkę do pliku jar: **lib\math.jar**

4. Dodajemy zasoby nie będące kodem Javy:
   a. Wyczyść katalog **out** i utwórz w katalogu bazowym katalog **resources**
   b. Dodaj tam plik **logo.txt** z logiem powitalnym programu, wpisz swoje imię/nicka w pliku.
   c. W klasie ***CalculatorCli*** wywołaj metodę **pl.sda.math.calculator.Utils**.*printLogo()* na początku działania programu
   d. Uruchom ***CalculatorCli***
   e. **Uwaga!** Podczas uruchamiania programu należy dodać do classpath ścieżkę do katalogu: **resources**

5. *Hackujemy nieznany kod
   a. Wyczyść katalog out i pobierz plik **secret.jar**
   b. Używając narzędzi poznanych do tej pory spróbuj odkryć jaka klasa i jakie metody znajdują się w pliku **secret_calculator.jar**
   c. Użyj metod z **secret_calculator.jar** w swoim kodzie
   d. Dodaj do ***CalculatorCli*** użycie nowych metod
   e. Uruchom ***CalculatorCli***

* Dla chętnych.

Java archive

JAR (ang. Java archive) – archiwum w formacie **ZIP** używane do strukturalizacji i kompresji plików klas języka Java oraz powiązanych z nimi metadanych. Archiwum **JAR** składa się z pliku manifestu umieszczonego w ścieżce **META-INF/MANIFEST.MF**, który informuje o sposobie użycia i przeznaczeniu archiwum. Archiwum **JAR**, o ile posiada wyszczególnioną klasę główną, może stanowić osobną aplikację. Archiwa **JAR** mogą być podpisywane cyfrowo.

źródło: https://pl.wikipedia.org

Podstawowym narzędziem do obsługi archiwów **JAR** jest program **JAR** dołączony do pakietu **JDK**.
Przykłady użycia:

- **jar cvfe calculator.jar pl.sda.calculator.CalculatorCli -C dest/ . -** stwórz plik jar z ustawioną klasą główną CalculatorCli i z zawartością pobraną z katalogu desc. Plik manifest.mf dodany ustawiony automatycznie.
- **jar cvfm calculator.jar manifest.txt -C dest/ . -** stwórz plik jar z zawartością pobraną z katalogu desc, dodaj do niego plik manifest.mf
- **java –jar calculator.jar -** uruchom aplikację zapisaną w calculator.jar

Opis parametrów:

- **c** - stwórz nowe archiwum **v** - wyświetl informacje w trakcie tworzenia archiwum
- **f** - ustaw nazwę archiwum **e** - ustaw klasę główną
- **m** - dołącz informacje manifestu z podanego pliku

Jar - problemy

Note: The Class-Path header points to classes or JAR files on the local network, not JAR files within the JAR file or classes accessible over Internet protocols. To load classes in JAR files within a JAR file into the class path, you must write custom code to load those classes. For example, if MyJar.jar contains another JAR file called MyUtils.jar, you cannot use the Class-Path header in MyJar.jar's manifest to load classes in MyUtils.jar into the class path.

źródło: https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html

Problem zgłoszony w 2002 roku!:
https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4648386

Dostępne rozwiązania:

- One-JAR: http://one-jar.sourceforge.net/
- Uber-JAR
- Maven
- IntelliJ jar artifact

Dokumentacja kodu

Javadoc – narzędzie automatycznie generujące dokumentację na podstawie zamieszczonych w kodzie źródłowym znaczników w komentarzach. Właściwie nawet bez komentarzy może stanowić użyteczną dokumentację (o ile nazwy pól i metod mają zrozumiałem nazwy). Javadoc został stworzony specjalnie na potrzeby języka Java.
Przykłady użycia:

- **javadoc -d -author -version docs src\pl\sda\calculator\core\Calculator.java**

```
/**
 * Prosty kalkulator do obsługi dodawania i mnożenia.
 *
 * @author  Jarosław Skarżyński
 * @version 1.0
 */
public class Calculator {
    /**
     * Metoda zwraca sumę podanych argumentów.
     * Jeżeli suma przekroczy <tt>Integer.MAX_VALUE</tt> wyjątek zostanie wyrzucony.
     *
     * @since 1.0
     * @return suma podanych argumentów
     */
    public int add(int a, int b) {
```

https://binfalse.de/2015/10/05/javadoc-cheats-sheet/

IntelliJ umożliwia podglądnięcie dokumentacji w kodzie:
*{w klasie}(**Ctrl + Q** lub **Menu → View → Quick Documentation**)*

Zadania

#calculator

6. Budujemy własnego jara
   a. Skopiuj do katalogu bazowego plik **manifest.txt**
   b. Umieść wszystkie pliki potrzebne do uruchomienia programu w katalogu **dest**
   c. Za pomocą polecenia **jar** zbuduj paczkę **calculator.jar**
   d. Uruchom aplikację **calculator.jar** z wiersza poleceń
   e. Przenieś plik **calculator.jar** do innego katalogu, spróbuj uruchomić...
7. Budowanie archiwum jar za pomocą **IntelliJ**
   a. Wyczyść katalog **out** i usuń plik **calculator.jar**
   b. Wykonaj instrukcje ze strony: https://blog.jetbrains.com/idea/2010/08/quickly-create-jar-artifact/
   c. Przenieś plik **calculator.jar** do innego katalogu, spróbuj uruchomić
   d. * Prześlij plik **calculator.jar** do innej osoby z grupy i poproś o uruchomienie
8. Dodajemy dokumentację kodu.
   a. Dodaj komentarze Javadoc od klasy ***Calculator***
   b. Wykorzystaj tagi: **@author**, **@version** + krótki opis na poziomie klasy
   c. Dodaj do metody, która mnoży, kod który sprawdzi czy parametry metody są liczbami dodatnimi, w przypadku nie spełnienia warunku wyrzuć odpowiedni wyjątek – opisz to w dokumentacji
   d. * Wykorzystaj tagi: **@see** i **@link** do wskazania skąd pochodzą implementacje metod: cos, sin, max, min
   e. * Oznacz jedną z metod tagiem **@deprecated** – zobacz jak wywołanie takiej metody wygląda w IntelliJ
   f. * Możesz dodać co uważasz za przydatne, skorzystaj ze strony: https://binfalse.de/2015/10/05/javadoc-cheats-sheet/
   g. Wykorzystując narzędzie javadoc wygeneruj dokumentację klasy ***Calculator***

* Dla chętnych.

JVM - specyfikacja i rozwój

- **JVM** - wirtualny „komputer” opisany przez specyfikację (publicznie dostępną), która pozwala różnym producentom oprogramowania na tworzenie własnych maszyn wirtualnych pracujących pod kontrolą różnych środowisk i urządzeń.

- **Java** - język programowania, który kompiluje się do kodu bajtowego wykonywanego w ramach **JVM**. Specyfikacje **JVM** i **Java** są publicznie dostępne pod adresem:
  https://docs.oracle.com/javase/specs/

- **Java Community Process (JCP)** - to proces, który formalizuje i standaryzuje technologie platformy Java. Członkami JCP są przedstawiciele firm: Oracle, Intel Corp., IBM, Hewlett Packard Enterprise, Adobe Systems Inc i wiele innych. Osoby prywatne również mogą zostać członkiem JCP – wystarczy wypełnić formularz zgłoszeniowy.
  https://jcp.org

- **Java Specification Requests (JSRs)** – to opis propozycji które mają ulepszyć platformę Java. Tworzony jest przez członków JCP. Propozycja przechodzi przez kilka faz, gdzie jest sprawdzana i implementowana. Jeżeli zostanie zaakceptowana przez JCP kończy jako działający kod w ramach platformy Java.

JVM - implementacja firmy Oracle

HotSpot - maszyna wirtualna Javy, dostarczana przez firmę Oracle Corporation razem z pakietem Java Runtime Environment. Wcześniej rozwijana przez firmę Sun Microsystems. Napisana jest w języku C++ (większość kodu). Kod źródłowy projektu to ok. 250 tys. linii kodu.

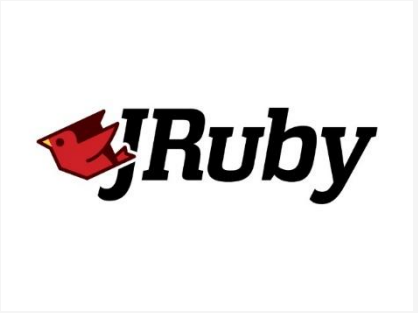
Na program składają się m.in.:

- ładowarka klas (**classloader**)
- interpreter kodu bajtowego (ang. **bytecode** interpreter)
- **JIT** (Just In Time) kompilator, produkujący kod natywny
- kilka mechanizmów odśmiecania pamięci (garbage collector)
- zestaw bibliotek wspierających wykonanie - Java API – udostępniających usługi takiej jak: obsługę plików czy GUI (Swing, AWT)

Skoro znana jest specyfikacja kodu bajtowego nic nie stoi na przeszkodzie, aby napisać własną maszynę wirtualną (prawnie zabroniona jest zmiana specyfikacji – jest ona chroniona patentem)

Przykłady JVM: **IBM J9**/**Eclipse OpenJ9**, **Jamiga**, **Dalvik** /**Android Runtime (ART)**

JVM - nie tylko Java



JDK vs JRE vs JVM

W zależności od potrzeb i liczby dostępnych narzędzi, wyróżniane są dwie główne dystrybucje **JVM** (**JVM** nie występuje w formie samodzielnej a jedynie jako część większej całości - JRE albo JDK):

- **Java Runtime Environment (JRE)** – zawiera wyłącznie narzędzia niezbędne do uruchomienia aplikacji, tzw. środowisko uruchomieniowe. Jeżeli chcemy tylko uruchamiać programy napisane w Javie to **JRE** nam wystarczy.

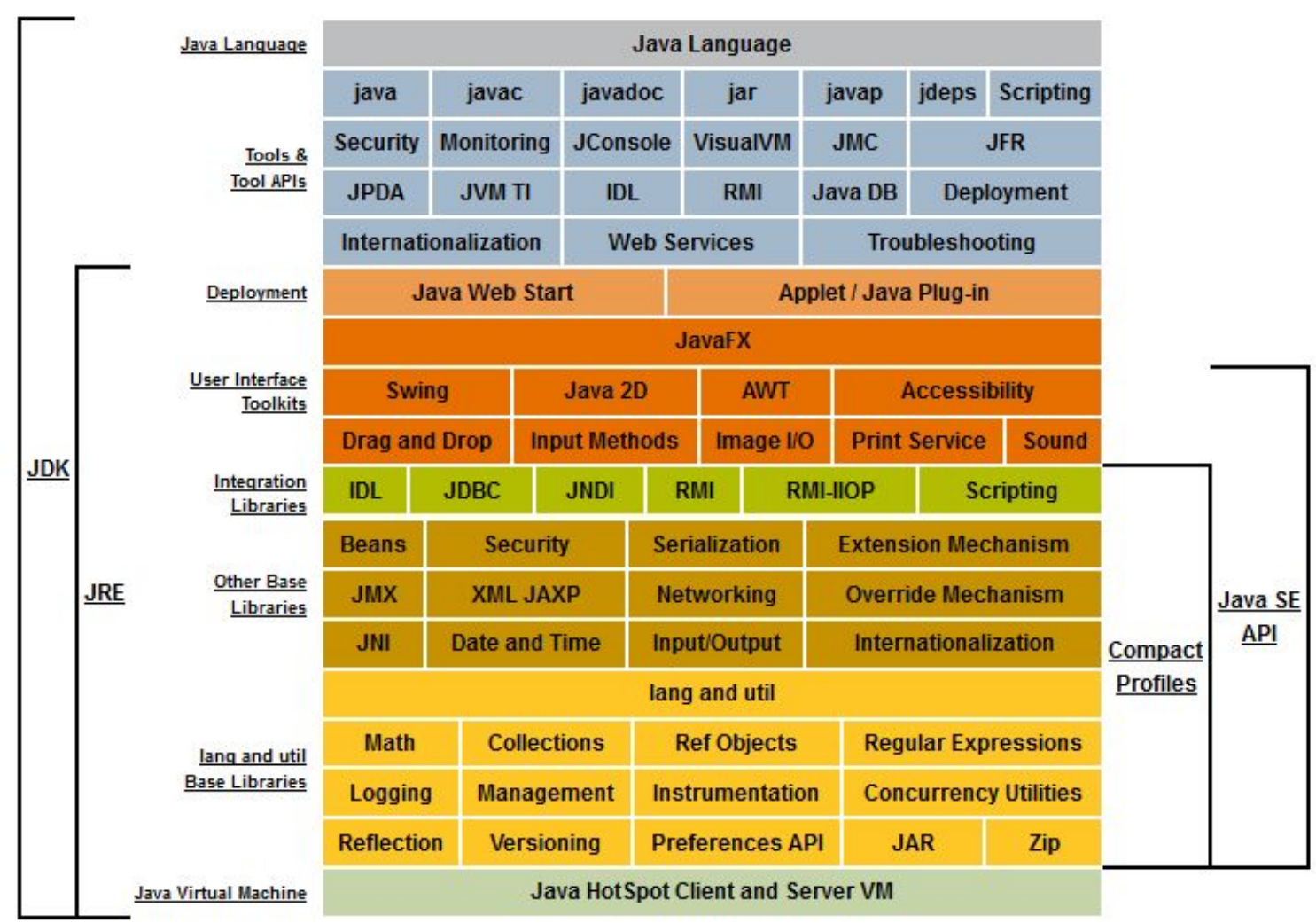
- **Java Development Kit (JDK)** – zawiera również narzędzia dla programistów pozwalające na tworzenie aplikacji na platformę **JVM**. Jeżeli chcemy implementować lub chociażby kompilować programy napisane w Javie to potrzebujemy **JDK**.

Najpopularniejsze wersje JDK to:

- **OpenJDK** – oficjalna implementacja wzorcowa (reference implementation) Java SE od wersji 7 o kodzie całkowicie otwartym. Licencja oparta na GPL v2. Rozwijany przez społeczność Java (JCP) w tym firmę Oracle.

- **OracleJDK** – w dużej mierze oparta na OpenJDK, ale zawiera też kod komercyjny (Java Flight Recorder, Java Plugin, Java WebStart), trochę więcej klas i jest bardziej stabilny. Brak wyraźnych różnic funkcjonalnych czy też wydajnościowych między obiema wersjami.

JDK vs JRE vs JVM



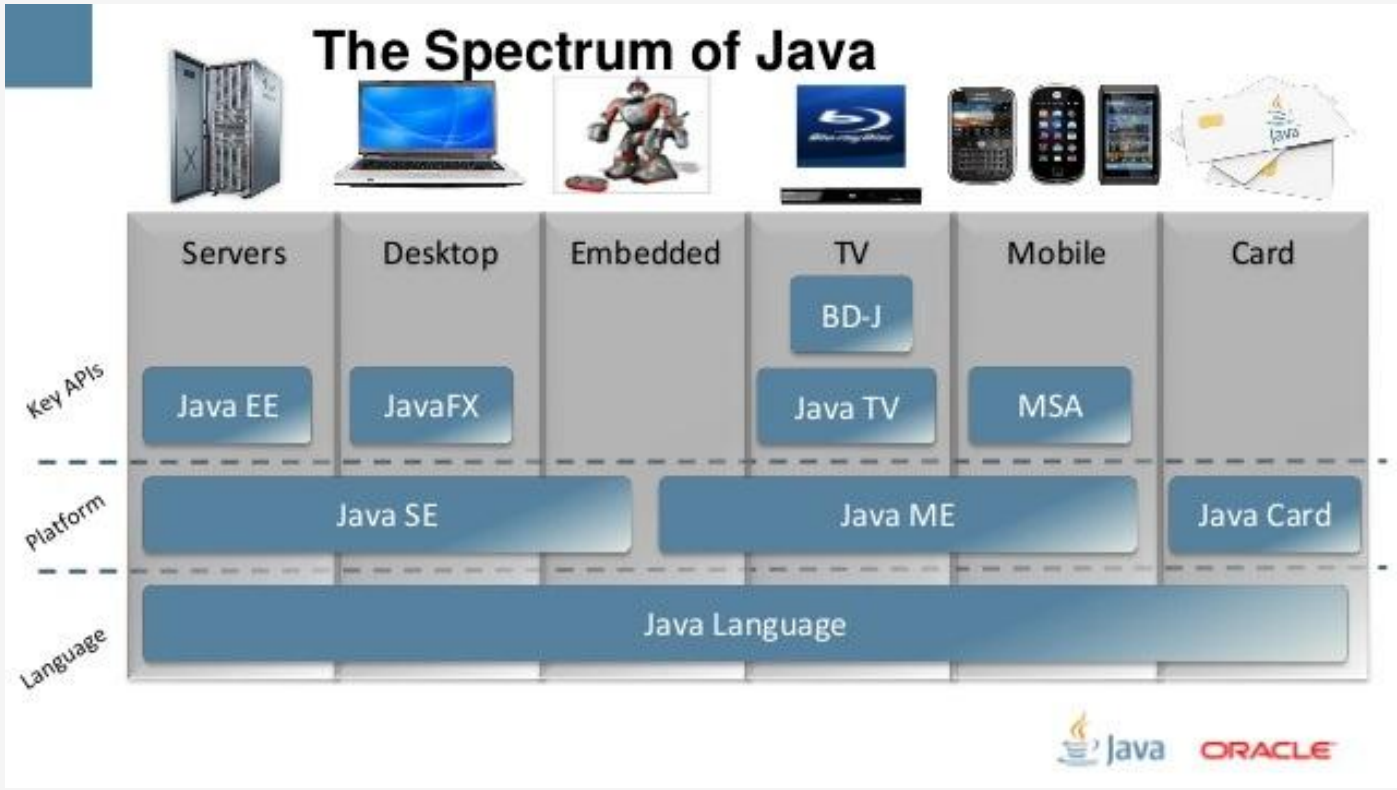
źródło: https://docs.oracle.com/javase/8/docs/index.html

Java to nie tylko język programowania, to platforma!

- **Java Platform, Standard Edition (Java SE)** – podstawowa wersja języka Java służąca głównie do tworzenia aplikacji dekstopowych. Aktualna wersja: **Java SE 11**.

- **Java Platform, Enterprise Edition (Java EE)** - zbudowana na bazie Java SE. Przeznaczona do rozwijania i uruchamiania dużych, wielowarstwowych, skalowalnych aplikacji typu enteprise. Aktualna wersja: **Java EE 8**.

- **Java Platform, Micro Edition (Java ME)** – podzbiór Java SE. Posiada własną **VM** przeznaczoną na małe przenośne urządzenia (małe zużycie pamięci).
  Aktualna wersja: **Java ME 8**.



Konfiguracja aplikacji - parametry linii komend

parametrem przekazywane do metody main() podanej klasu. Trafią do tablicy: String[] **args**

```
java   pl.sda.stat.MainArgs   Jarek   101
```

main() to publiczna, statyczna metoda od której zaczyna się wykonanie programu

parametrem metody main() jest tablica stringów, w której przekazane są parametry wywołania programu

zmienna **args** przechowuje referencje do tablicy z stringami, możemy się nią posługiwać jak zwykłą tablicą

```
public static void main(String[] args) {
    if (args.length == 0) {
        return;
    }

    String name = args[0];
    String number = "?";
    if (args.length > 1) {
        number = args[1];
    }
    …
```

Film z instrukcją: https://goo.gl/4djZM7

Konfiguracja aplikacji - parametry systemowe

parametrem systemowe przekazywane do wywoływanego programu

```
java  -Dapp.name=Jarek  -Dapp.number=101  pl.sda.stat.MainArgs
```

```
String name = System.getProperty("app.name");
String number = System.getProperty("app.number");
```

w dowolnym miejscu naszego programu możemy pobrać parametry systemowe

```
String version= System.getProperty("java.version");
String userName = System.getProperty("user.name");
String encoding = System.getProperty("file.encoding");
```

Java dostarcza również zestaw predefiniowanych zmiennych systemowych ustawianych globalnie przez JVM dla wszystkich programów

Lista wszystkich zmiennych systemowych: pl.sda.pl.sda.jvm.demo.SystemPropertiesSample

Konfiguracja aplikacji - plik *.properties i JMX

Konfiguracja przez zewnętrzny plik z parametrami:

1. Tworzymy plik tekstowy z parametrami, np.: application.properties
2. Umieszczamy w nim parametry w postaci {klucz}={wartość}, np.: **calculator.value.to=20**
3. Z poziomu kodu tworzymy obiekt klasy **java.util**.***Properties*** i wypełniamy go danymi za pomocą metody *load()*, np.: **properties**.*load*("/application.properties");
4. Za pomocą obiektu **properties** możemy pobrać parametry podając klucz, np.: **properties**.*getProperty*("calculator.value.to")

JMX - czyli **Java Management Extensions** – technologia Javy, która zawiera narzędzia potrzebne do zarządzania oraz monitorowania aplikacji. Zasoby reprezentowane są przez obiekty nazywane MBean'ami (Managed Bean). Dzięki **JMX** możliwe jest między innymi:

- "podglądanie" działającej aplikacji, m.in. można sprawdzić ile CPU i pamięci jest zużywane, z jakimi parametrami została uruchomiona aplikacja itp.
- zmiana konfiguracji w „locie" - dynamiczna zmiana parametrów w obiektach zarejestrowanych jako MBean - zobaczymy to w części praktycznej
- działanie programów-profilerów takich jak: **JConsole**, **VisualVM**, **Java Mission Control**

Zadania

#calculator

9. Tworzymy testera i ćwiczymy konfigurację aplikacji:
   a. Utwórz klasę ***CalculatorTester***, która będzie przyjmowała na wejściu 3 liczby (jako parametry wywołania programu podane z linii komend):

      - rodzaj operacji - 1 - dodawanie, 2 - mnożenie, 3 - odejmowanie
      - zakres od-do argumentów, które mają być przekazane do obliczeń, należy wywołać operację wskazaną przez pierwszy argument dla wszystkich liczb z podanego zakresu - wynik ma być wyświetlony na konsoli.

      np. wywołanie **java pl.sda.calculator.*CalculatorTester*** 2 10 20 – oznacza, że tester ma pomnożyć wszystkie liczby z przedziału [10, 20]: 10 * 11 * 12 * … * 20 a wynik wypisać na konsoli

   b. Uruchom i przetestuj swój program - w **IntelliJ**
   c. * Uruchom i przetestuj swój program - z wiersza poleceń (komendy: **javac** + **java**)
   d. * Zmień przekazanie parametrów wejściowych - tak żeby program skorzystał z parametrów systemowych (przekazana przez -D{nazwa_parametru}={wartość_parametru} w czasie wywołania programu
   e. * Jeżeli użytkownik nie poda parametrów wejściowych, mają być użyte domyślne, pobrane z pliku **application.properties** (należy do tego użyć klasy ***java.util.Properties***).

* Dla chętnych.

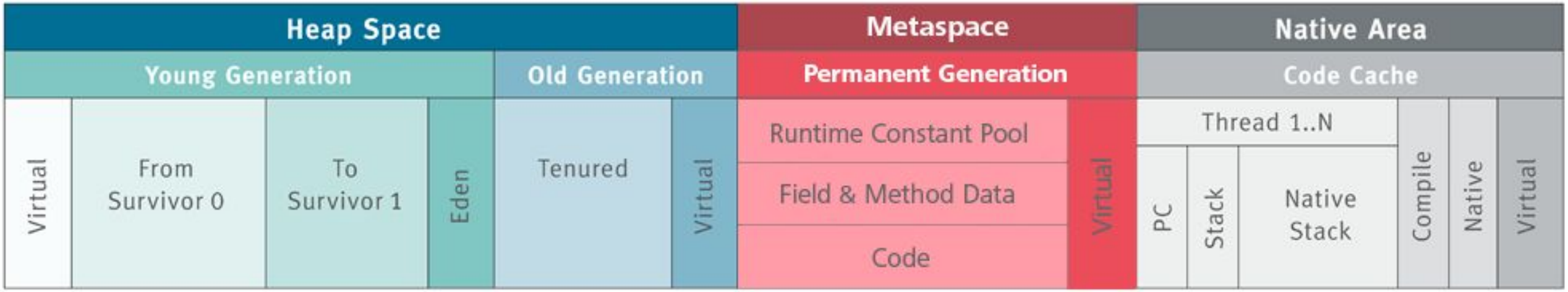
Zadania

#calculator demo

10. Demonstracja działania **JMX**:
    a. Pobierz projekt **CalculatorDemo** z repozytorium prowadzącego: https://github.com/jaroslaw-skarzynski/calculator.git
    b. Uruchom program **VisualVM** - znajduje się on w katalogu **%JAVA_HOME%/bin**, np.: **C:\Program Files\Java\jdk1.8.0_144\bin\jvisualvm.exe** + zainstaluj plugin do **MBeans**
    c. Uruchom ***CalculatorTester*** z projektu **CalculatorDemo** z parametrami programu: 1000 1 15 oraz z parametrem systemowym: **-Dcom.sun.management.jmxremote**
    d. Przejdź do programu **VisualVM** i znajdź uruchomiony przed chwilą program - otwórz go
    e. Przejdź do zakładki: **MBeans** i spróbuj zmienić parametry w klasie **pl.sda.calculator.properties**
    f. * Wykonaj kroki b-d za pomocą programów: **JConsole i Java Mission Control**
11. Demonstracja działania programu **VisualVM** - analiza zużycia **CPU**
    a. Uruchom program **VisualVM** - znajduje się on w katalogu **%JAVA_HOME%/bin**, np.: **C:\Program Files\Java\jdk1.8.0_144\bin\jvisualvm.exe**
    b. Uruchom klasę ***HighCpuSample*** z parametrem systemowym: **-Dcom.sun.management.jmxremote**
    c. Rozpocznij analizę programu:
        i. które wątki zużywają najwięcej czasu procesora?
        ii. które metody działają najdłużej?
        iii. co blokuje poszczególne wątki?

* Dla chętnych.

Runtime Data Area

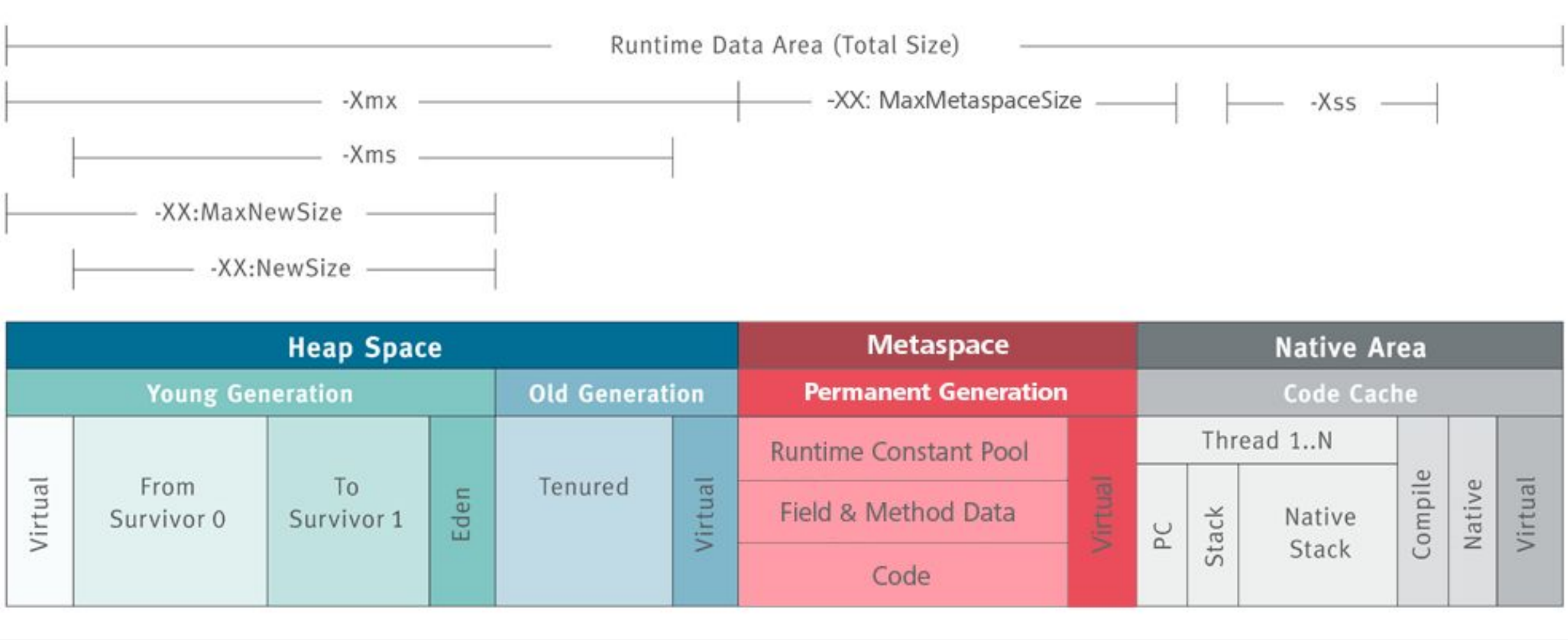


źródło: https://docs.deistercloud.com

Runtime Data Area - to obszar pamięci RAM przydzielony wirtualnej maszynie przez system operacyjny na którym **VM** działa. Obszar ten można podzielić na kilka części:

- pamięć współdzielona przez wszystkie wątki:
  - **Heap Space** (sterta) - tutaj przechowywane są wszystkie obiekty używane w programie.
  - **Metaspace** - przechowuje metadane opisujące pola i metody (nazwa, typ itp.), statyczne pola i bytecode
- pamięć przydzielona do pojedynczego (każdego) wątku:
  - PC register - przechowuje wskaźnik do instrukcji która jest aktualnie wykonywana w wątku
  - **JVM** stack (stos) - kolejka LIFO przechowująca tzw. ramki (frame) dla każdej metody wykonywanej w wątku
  - Native stack - stack wykorzystywany do wywołania kodu C/C++ poprzez **JNI** (Java Native Interface)

Runtime Data Area

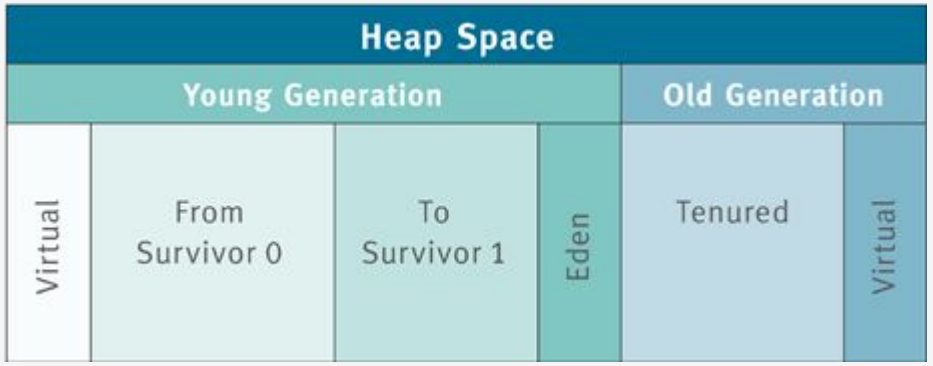


HeapSpace

HeapSpace to obszar pamięci (RAM), w którym przechowywane są obiekty.
Parametry do zarządzania wielkością pamięci:

-Xms125M, **-Xms**1G - startowa
-Xmx1024M, **-Xmx**2G - maksymalna

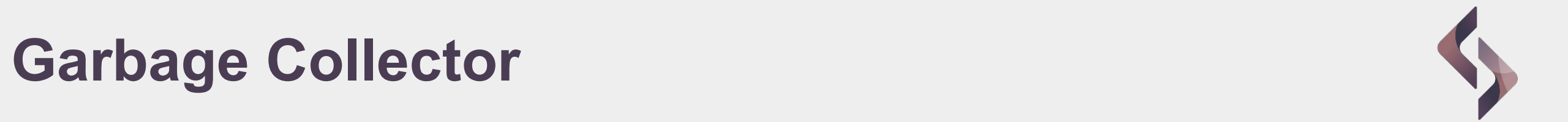
java -Xms512m –Xmx2g –jar calculator.jar



Dzieli się na kilka obszarów:

- **Young Generation** tutaj są przechowywane wszystkie nowo-stworzone obiekty, dzielimy go na kolejne obszary:
  - **Eden –** tutaj w pierwszej kolejności są umieszczane nowe obiekty
  - **Surivor 0 i 1 –** gdy skończy się miejsce w obszarze Eden, wszystkie obiekty które są jeszcze używane są przenoszone do jednego z obszarów Survivor (to tak zwany **Minor GC**). Aktualnie wypełniony obszar Survivor również jest sprawdzany i obiekty używane są przenoszone do tego samego obszaru **Surivor** co obiekty z **Edenu**. W ten sposób w danym czasie jeden z obszarów **Surivor** jest zawsze pusty.
- **Old Generation –** tutaj trafiają obiekty, które przetrwały wiele cykli „przenosin” pomiędzy obszarami **Eden**, **Survivor 1 i 2**

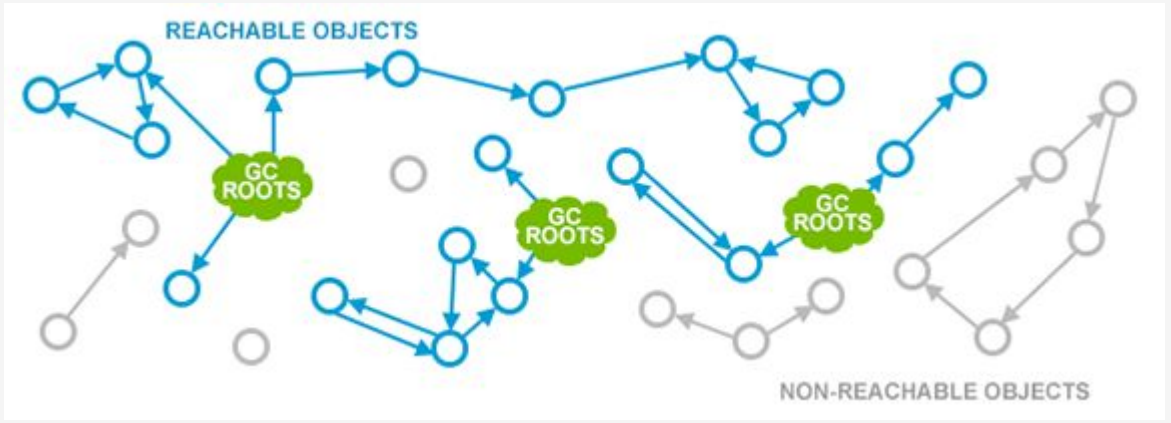
Garbage Collector



Garbage Collector – mechanizm oczyszczania pamięci JVM z nieużywanych danych. Działanie GC opiera się na hipotezie zwanej: **Generational Hypothesis** (stąd podział Heap Space)**,** która składa się z dwóch punktów:

- większość obiektów szybko przestaje być używana przez program
- te które przetrwają zwykle żyją bardzo długo

Stąd wysnuto wniosek że łatwiej zająć się obiektami które przeżywają (jest ich mniej) a resztę pamięci oczyścić. Aby to zrobić należy oznaczyć (**mark**) obiekty **„żywe”** i usunąć całą resztę (**sweep**). Stąd nazwa podstawowego algorytmy GC to: **Mark and Sweep.** Operacje te wiążą się z zatrzymaniem działania aplikacji (**stop-the-world**) w czasie działania GC.

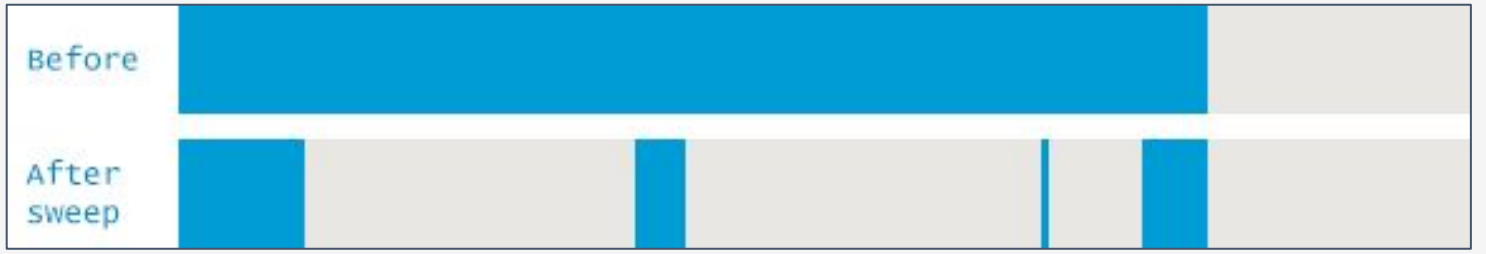


Obiekty **„żywe”** to takie które są połączone (bezpośrednio lub przez inne obiekty) z tzw. **GC roots**. Do **GC roots** zaliczamy:

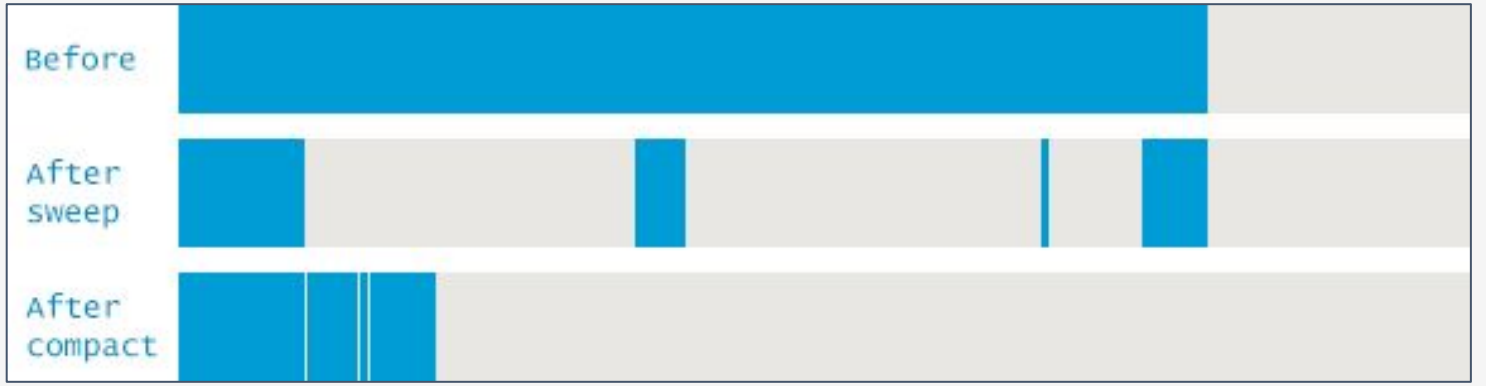
- zmienne lokalne
- aktywne wątki (zmienne typu thread local)
- pola statyczne
- referencje JNI

Garbage Collector – fazy usuwania

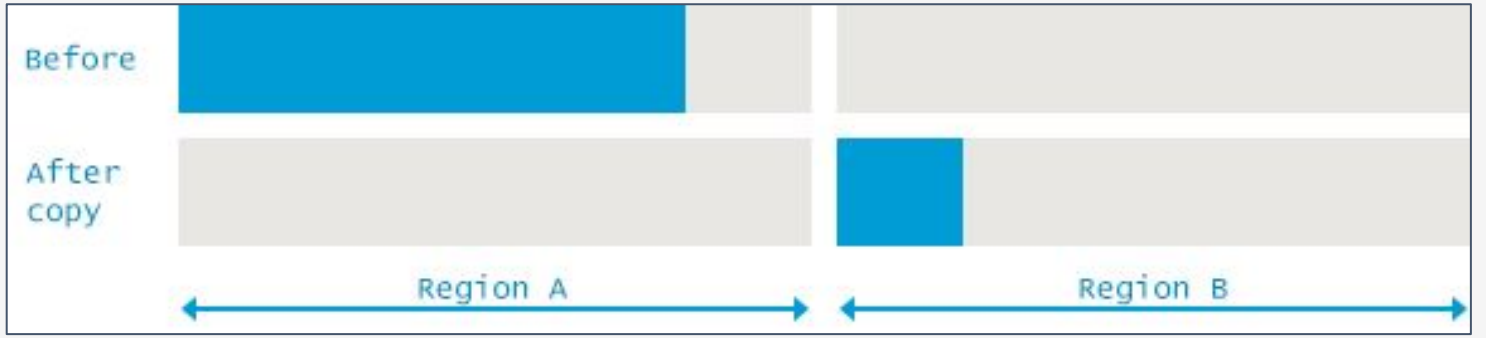
- **sweep (mark-and-sweep)** – po prostu „zapominamy” o „martwych” obiektach, dużo wolnych, ale małych obszarów



- **compact (mark-sweep-compact)** – po fazie **sweep** dodatkowo przesuwamy dane na początek obszaru (kompaktowanie), ale mamy dłuższą pauzę GC



- **copy (mark-and-copy) –** przenosimy wszystkie obiekty który przetrwały do nowego regiony, a stary uznajemy za pusty, tutaj potrzeba dodatkowej pamięci (dodatkowy region)



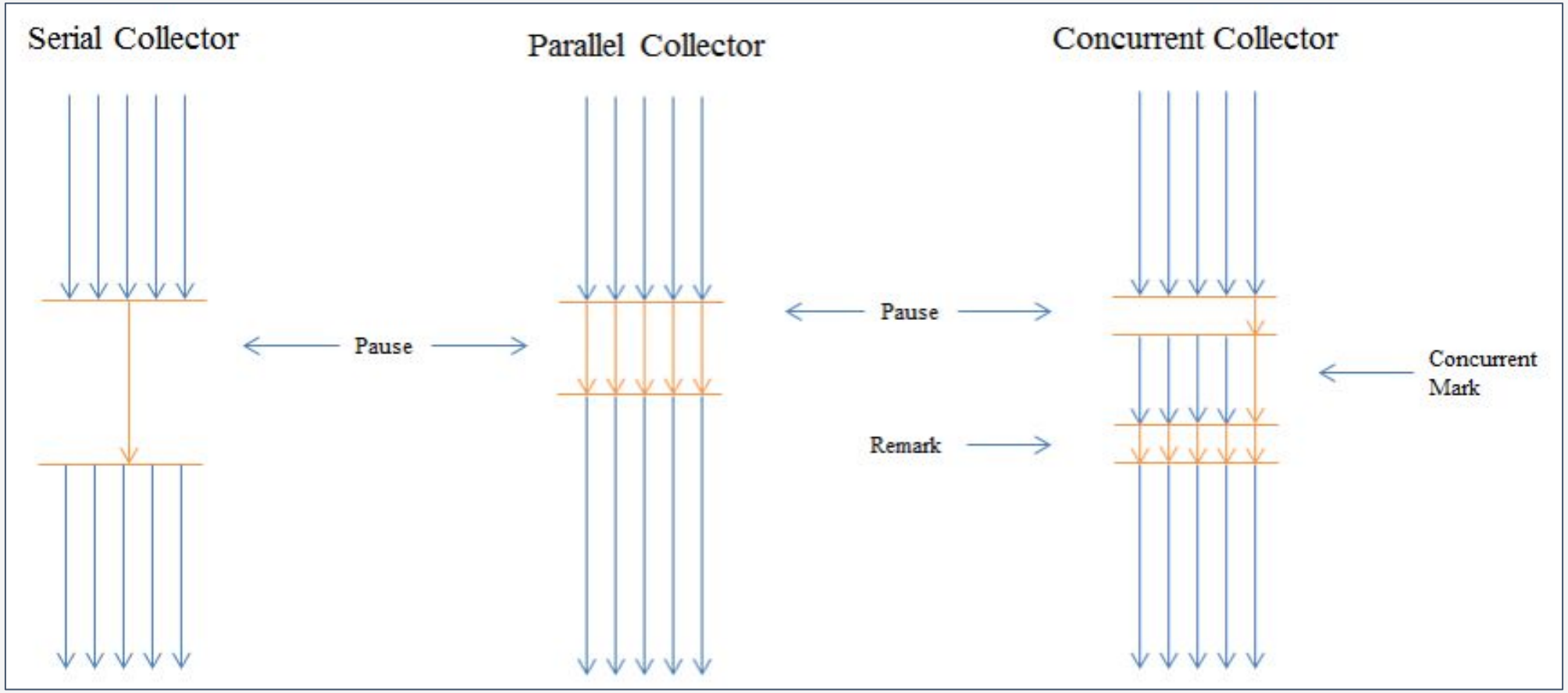
Garbage Collector- rodzaje

- **Serial GC(-XX:+UseSerialGC)** – używa algorytmu **mark-copy** dla Young Generation i **mark-sweep-compact** dla Old Generation. Działa w jednym wątku. Oba kolektory wywołują pauzę **stop-the-world**.

- **Parallel GC(-XX:+UseParallelOldGC –** używa algorytmu **mark-copy** dla Young Generation i **mark-sweep-compact** dla Old Generation. Działa na wielu wątkach. Oba kolektory wywołują pauzę **stop-the-world**.

- **Concurrent Mark and Sweep(-XX:+UseConcMarkSweepGC) –** używa algorytmu **mark-copy** dla Young Generation (na wielu wątkach, wywołuje pauzę **stop-the-world**) i **mark-sweep** dla Old Generation („prawie” współbieżnie z działaniem aplikacji, zaprojektowany by unikać długich pauz).

Garbage Collector- rodzaje



Garbage Collector- G1

G1 – Garbage First (przełącznik -XX:+UseG1GC) :

- następca algorytmu **CMS**, wprowadzony w **Java 7**, domyślny w **Java 9**
- stworzony po to by pauzy **stop-the-world** były bardziej przewidywalne i konfigurowalne (np.: żeby pauzy nie były dłuższe niż 5s w każdej minucie działania).
- **Heap Space** jest tu dzielona na małe regiony, w których są przechowywane obiekty. Dzięki temu **GC** nie musi przetwarzać całej pamięci Heap Space, może się skupić na mniejszym obszarze.
- **G1** zbiera informacje jak dużo „żywych” obiektów jest w każdym regionie. Region który zawiera najwięcej „śmieci” jest oczyszczany w pierwszej kolejności, stąd nazwa Garbage First.

Zadania

#calculator demo

12. Demonstracja działania programu **VisualVM** - analiza zużycia pamięci
    a. Uruchom program **VisualVM** - znajduje się on w katalogu **%JAVA_HOME%/bin**, np.: **C:\Program Files\Java\jdk1.8.0_144\bin\jvisualvm.exe**
    b. Uruchom klasę ***MemoryLeakSample*** z parametrem systemowym: **-Dcom.sun.management.jmxremote** oraz ustaw maksymalny rozmiar pamięci (**HeapSize**) na 100 MB
    c. Rozpocznij analizę programu:
        i. jak wygląda zużycie pamięci ?
        ii. czy GC czyści pamięć odpowiednio ?
        iii. czy są jakieś wycieki ?
        iv. zrób zrzut pamięci (HeapDump) i sprawdź która klasa zabiera najwięcej pamięci?
    d. Zmień maksymalny rozmiar pamięci (HeapSize) na 256 MB. Czy program dłużej działa ?
    e. Spróbuj znaleźć wyciek pamięci i naprawić go

Pytania?

Dziękuję ;)