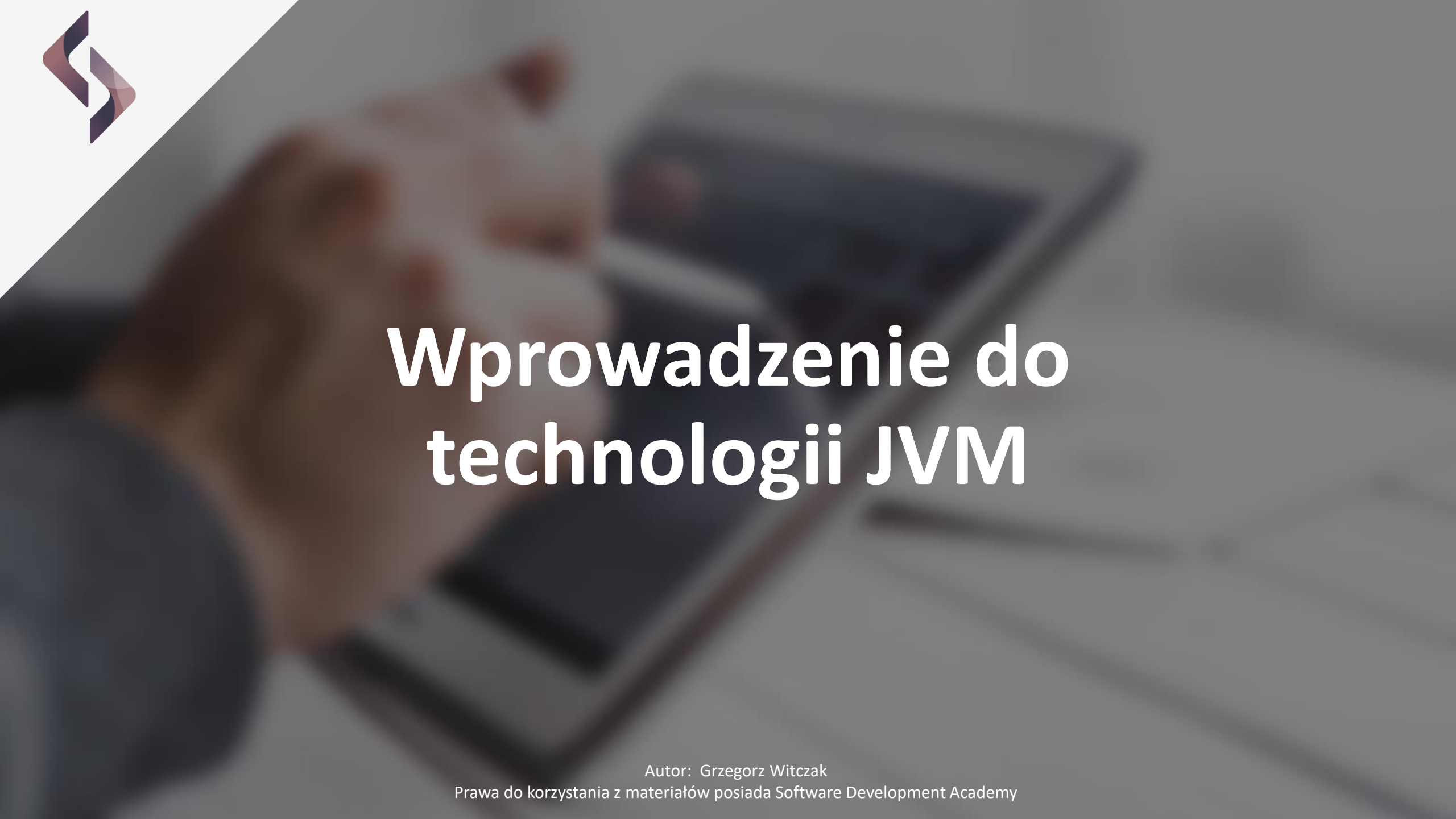
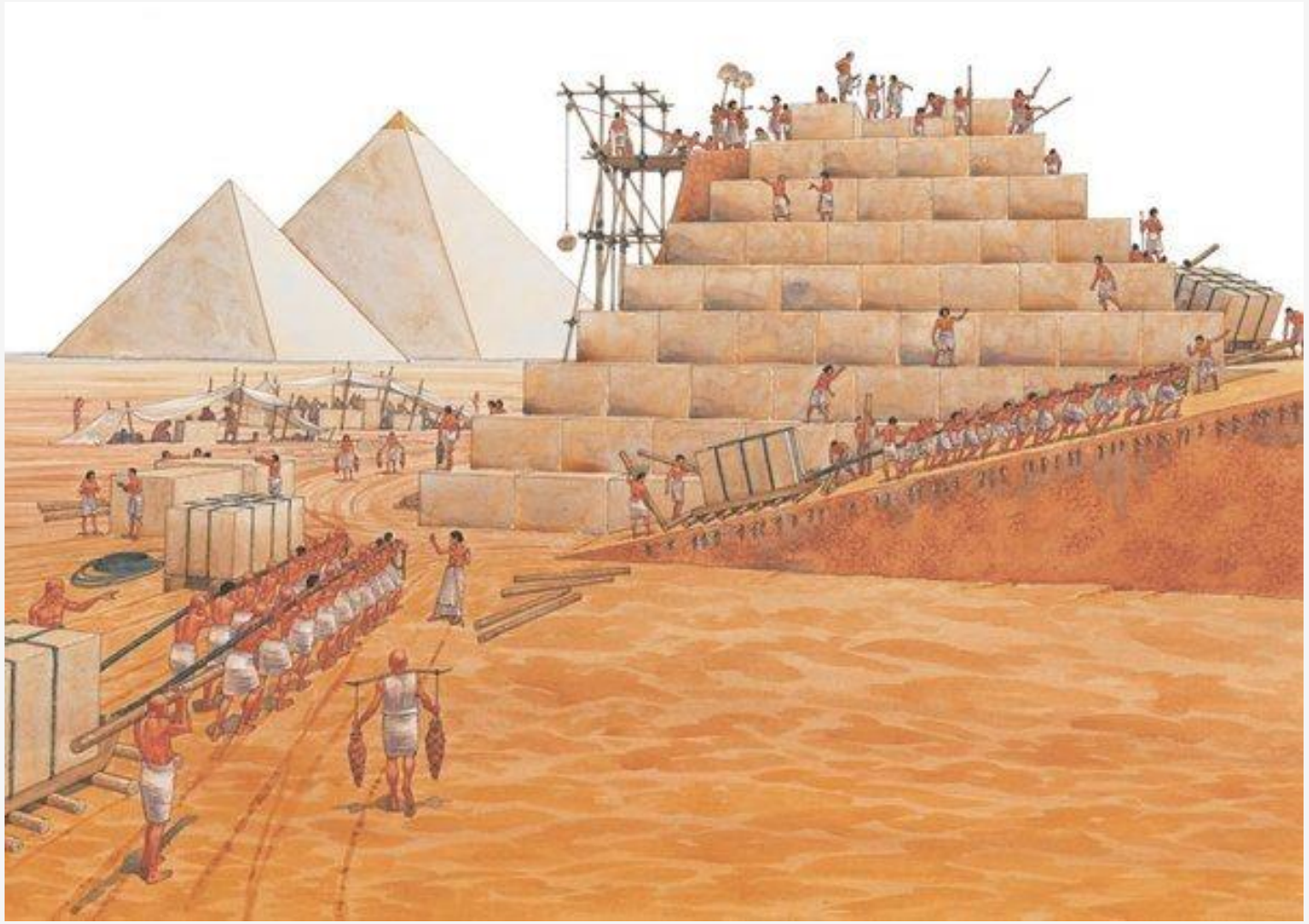


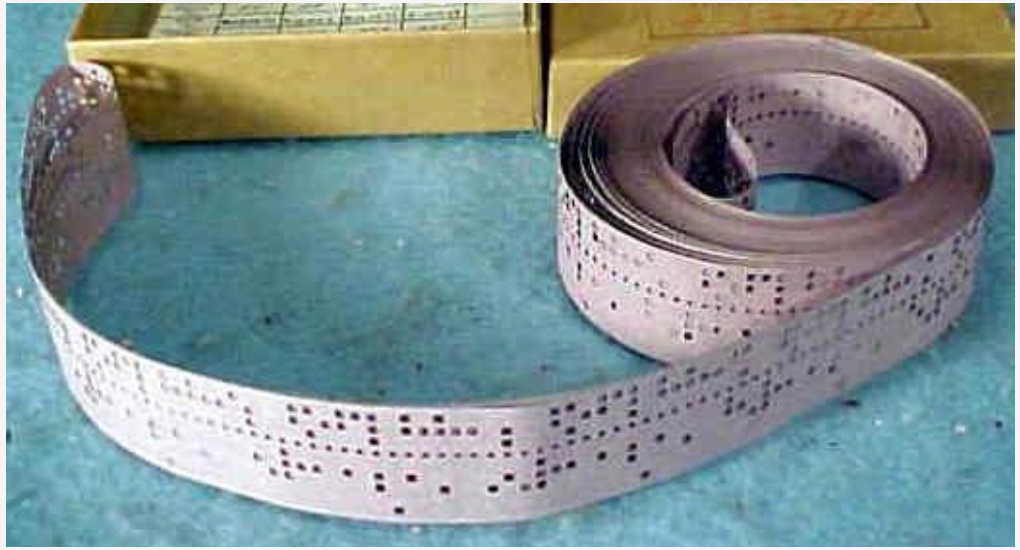
Wprowadzenie do technologii JVM

Autor: Grzegorz Witczak



Jak to widzi procesor? (1)
Język (bardzo) niskiego poziomu

- **Język maszynowy** (ang. *machine language*) – **zestaw rozkazów procesora**, które musi on wykonać, żeby powstał efekt, który nazywamy "działaniem programu"
- Praktycznie nieczytelny dla człowieka



Address	Machine Language			
0000 0000	0000	0000	0000	0000
0000 0001	0000	0000	0000	0010
0000 0010	0000	0000	0000	0011
0000 0011	0001	1101	0000	0001
0000 0100	0001	1110	0000	0010
0000 0101	0101	1111	1101	1110
0000 0110	0010	1111	0000	0000
0000 0111	1111	0000	0000	0000

Jak to widzi procesor? (2)
Język niskiego poziomu

- Język programowania niskiego poziomu (ang. *low-level programming language*) – język programowania umożliwiający **zapis rozkazów maszynowych** za pomocą oznaczeń symbolicznych
- **Asemblery** – rodzina języków niskiego poziomu, których jedno polecenie odpowiada jednemu rozkazowi procesora
- Kod nieprzenośny, zależy od architektury procesora
- Asemblacja – tłumaczenie kodu asemblerowego na kod maszynowy

```
TOTAL  .BLOCK 1
ABC    .WORD  2
XYZ    .WORD  3
       LOAD   REGD, ABC
       LOAD   REGE, XYZ
       ADD    REGF, REGD, REGE
       STORE  REGF, TOTAL
       HALT
```

Jak to widzi procesor? (3)
Języki wysokiego poziomu

- Język programowania wysokiego poziomu (ang. *high-level programming language*) – język programowania, który maksymalnie ułatwia zrozumienie kodu programu dla człowieka
- Skupienie na **abstrakcji**, czytelności kodu, a nie na sprzęcie i wykonaniu na nim kodu
- Języki takie jak **C**, Pascal, BASIC
- Kod przenośny, potrzebujemy jedynie kompilatora na daną platformę
- **Kompilacja** – tłumaczenie kodu źródłowego na kod maszynowy

```
byte main() {
  byte result = 0;
  short int ABC = 2;
  short int XYZ = 3;
  result = ABC + XYZ;
  return result;
}
```

Jak to widzi procesor? (4)
Języki (bardzo) wysokiego poziomu

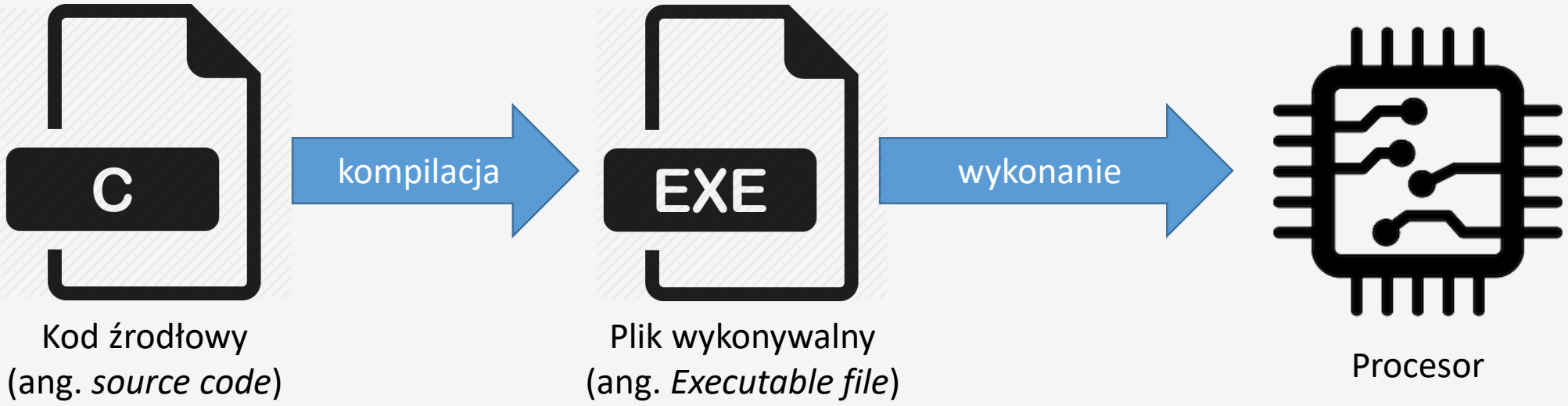
- Języki programowania wysokiego poziomu dalej **ewoluują**, dając programistom coraz większą wygodę.
  - Z jednej strony – **strukturalnie**, zwiększając abstrakcję (języki obiektowe, programowanie funkcyjne)
  - Z drugiej strony – **technologicznie** (maszyny wirtualne, kod pośredni, GC, JIT)
- Języki takie jak **Java, C# (.NET), Python, C++**

```
@NotNull
@Override
public List<FoldingBuilder> allForLanguage(@NotNull Language l) {
  FoldingBuilder result = forLanguage(l);
  if (result == null) return Collections.emptyList();
  return result instanceof CompositeFoldingBuilder ? ((CompositeFoldingBuilder)result).g
                                                   : Collections.singletonList(result);
}

@NotNull
public static FoldingDescriptor[] buildFoldingDescriptors(@Nullable FoldingBuilder build
  if (!DumbService.isDumbAware(builder) && DumbService.getInstance(root.getProject()).is
    return FoldingDescriptor.EMPTY;
  }

  if (builder instanceof FoldingBuilderEx) {
    return ((FoldingBuilderEx)builder).buildFoldRegions(root, document, quick);
  }
  final ASTNode astNode = root.getNode();
  if (astNode == null || builder == null) {
    return FoldingDescriptor.EMPTY;
  }
```

Kompilacja i uruchomienie (Język C)

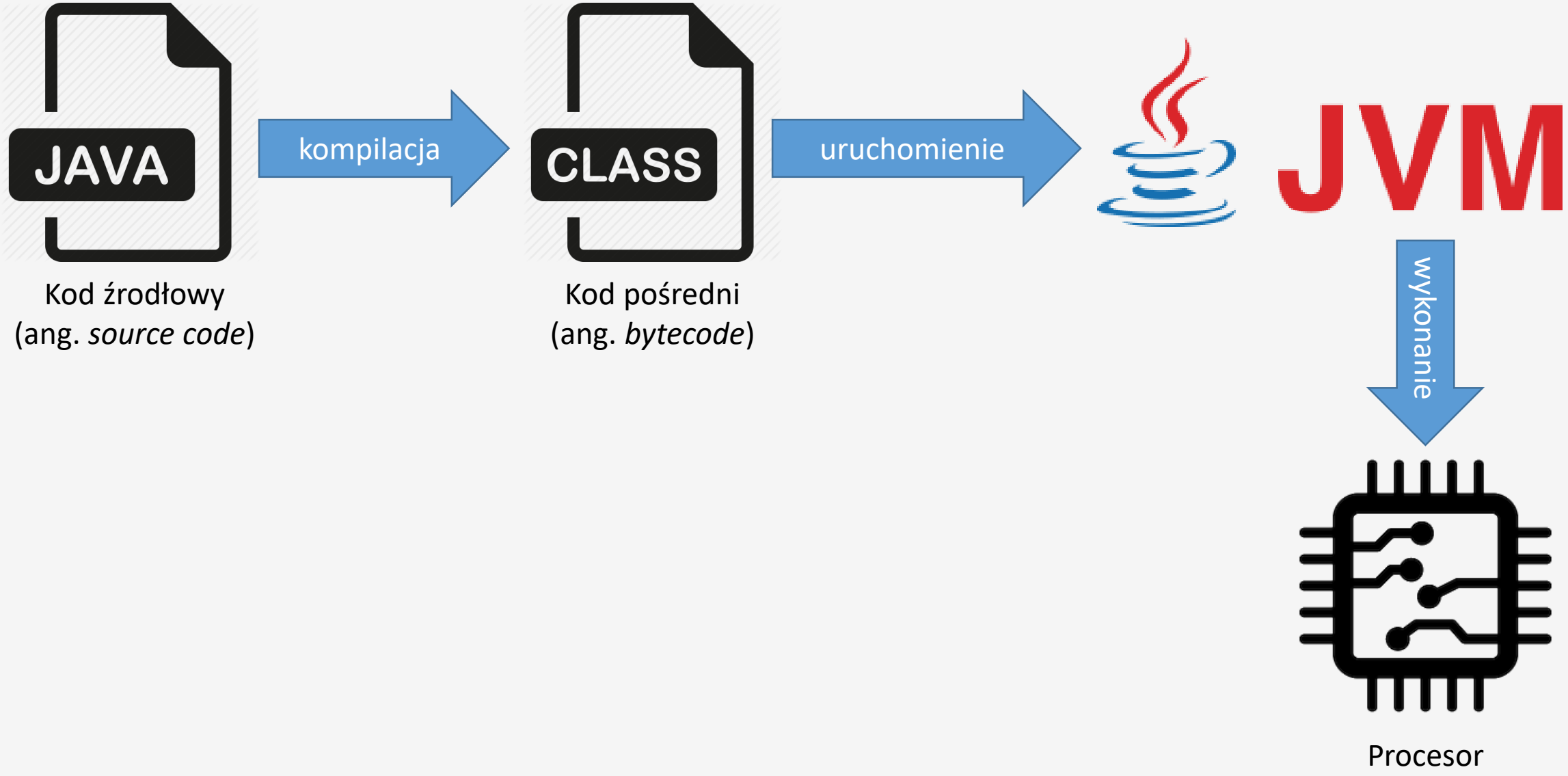


Kod źródłowy
(ang. *source code*)

Plik wykonywalny
(ang. *Executable file*)

Procesor

Kompilacja i uruchomienie (Java)

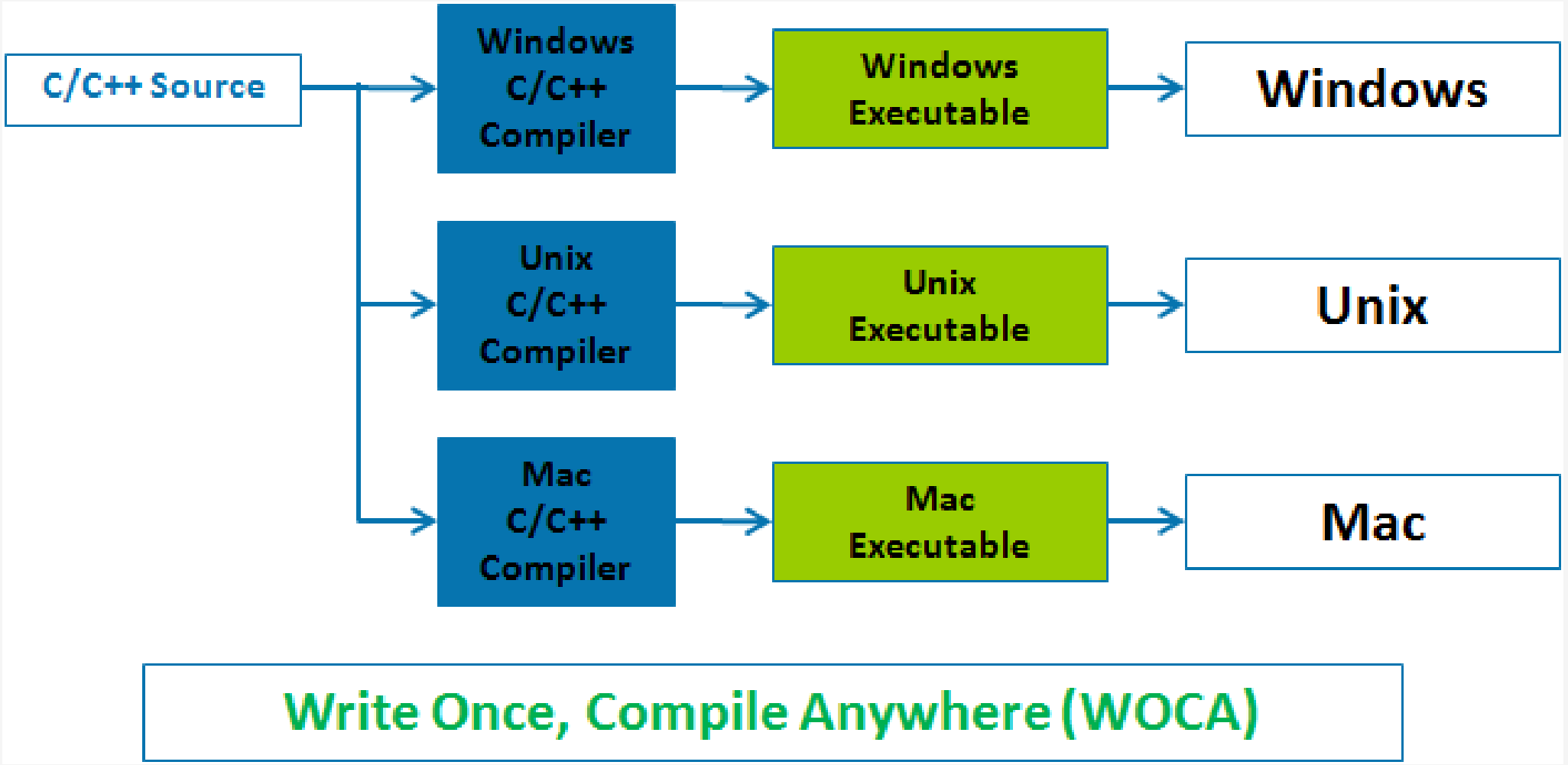


JVM

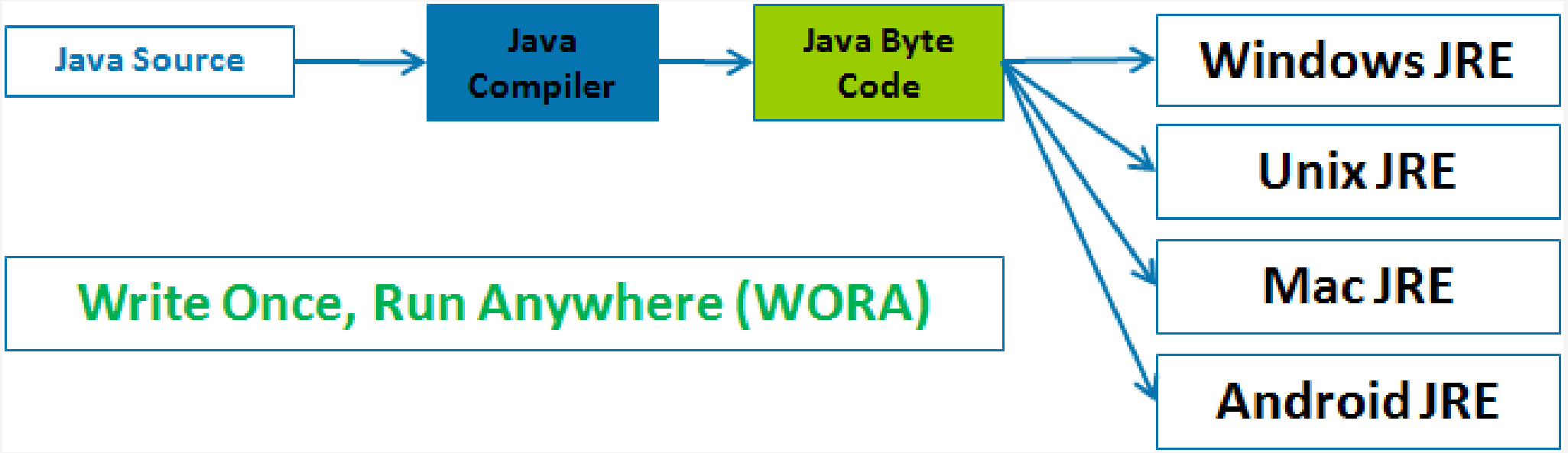
- **JVM** – Java Virtual Machine to napisany w C++ (w większości) **program**, który ma za zadanie wykonywać kod bajtowy Javy (przygotowany przez kompilator Javy)
- **Kod pośredni (bajtowy)** (pliki **`.class`**) jest interpretowany i tłumaczony na konkretne rozkazy procesora
- W wyniku tego procesu (kod nie jest natywnie uruchamiany na procesorze tylko przez „pośrednika”) zyskujemy **niezależność od architektury**, ale też wolniejsze działanie.

```
 0 aload_0
 1 new #3 <acceptanceTests/treeset_personOK/Main$A>
 4 dup
 5 new #8 <java/lang/Object>
 8 dup
 9 invokespecial #10 <java/lang/Object.<init>>
12 new #12 <java/lang/Integer>
15 dup
16 iconst_2
17 invokespecial #14 <java/lang/Integer.<init>>
20 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
23 new #12 <java/lang/Integer>
26 dup
27 iconst_1
28 invokespecial #14 <java/lang/Integer.<init>>
31 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
34 getstatic #20 <java/lang/System.out>
37 new #3 <acceptanceTests/treeset_personOK/Main$A>
40 dup
41 new #8 <java/lang/Object>
44 dup
45 invokespecial #10 <java/lang/Object.<init>>
48 new #12 <java/lang/Integer>
51 dup
52 iconst_2
53 invokespecial #14 <java/lang/Integer.<init>>
56 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
59 invokevirtual #26 <java/io/PrintStream.println>
62 return
```

WORA


C/C++ Source
Windows C/C++ Compiler
Unix C/C++ Compiler
Mac C/C++ Compiler
Windows Executable
Unix Executable
Mac Executable
Windows
Unix
Mac
Write Once, Compile Anywhere (WOCA)

WORA


Java Source
Java Compiler
Java Byte Code
Windows JRE
Unix JRE
Mac JRE
Android JRE
Write Once, Run Anywhere (WORA)

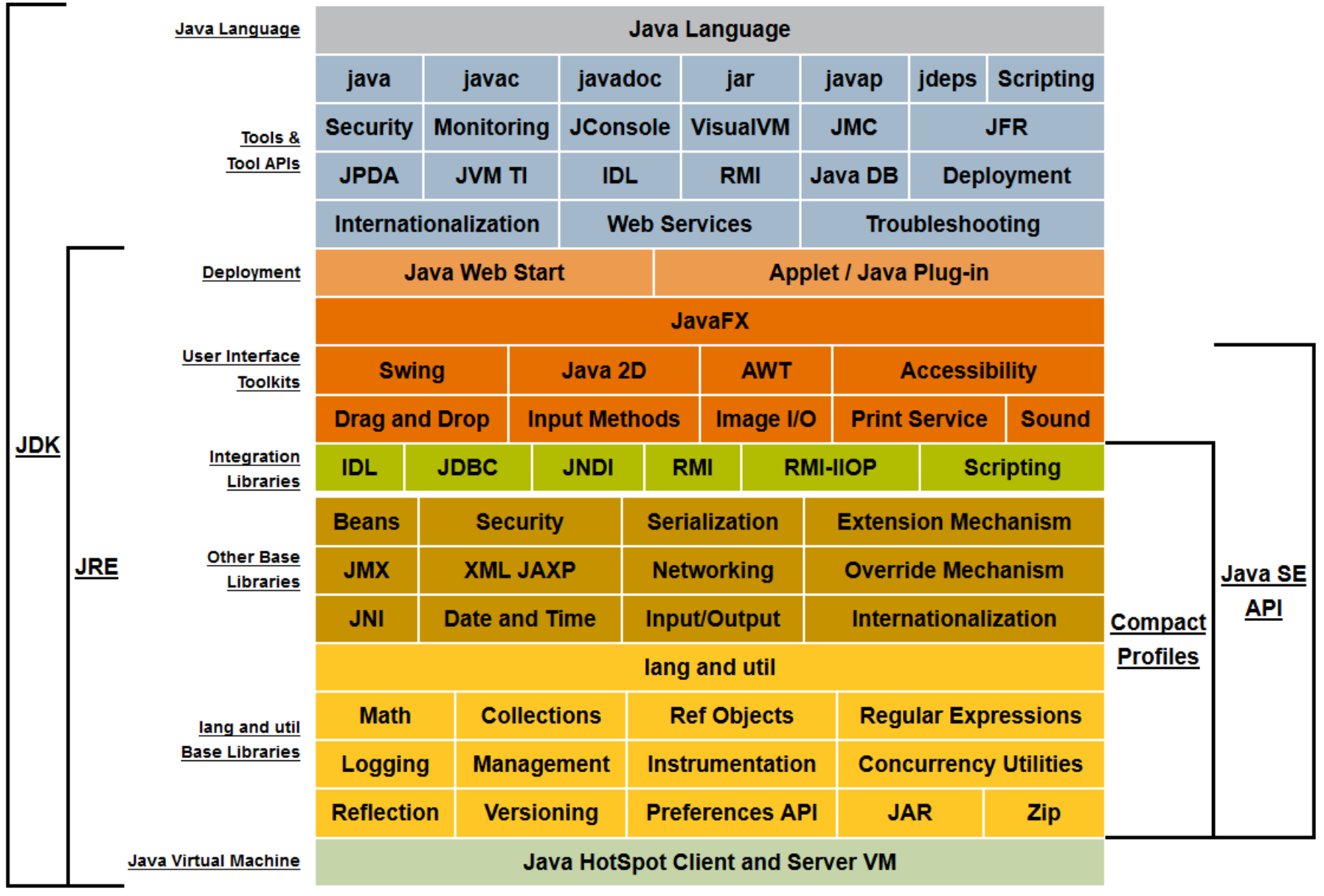
JRE vs JDK

- **JRE** to akronim dla **Java Runtime Environment**, tj. Środowisko Uruchomieniowe Javy. JRE składa się z JVM (Java Virtual Machine - Wirtualnej Maszyny Javy) oraz zestawu klas i narzędzi niezbędnych dla uruchamiania programów napisanych w Javie.
- **JDK** to akronim dla Java Development Kit, tj. Pakiet Programisty Javy. JDK zawiera JRE oraz narzędzia niezbędne do pisania i kompilacji aplikacji napisanych w języku Java.
- Podsumowując - jeśli chcemy implementować, lub chociażby kompilować, programy napisane w Javie - to musimy zainstalować JDK. Jeśli nie jesteśmy programistami i chcemy jedynie uruchamiać skompilowane już aplikacje to wystarczy nam JRE. JVM nie występuje w formie samodzielnej, a jedynie jako część większej całości - JRE albo JDK.

JRE vs JDK

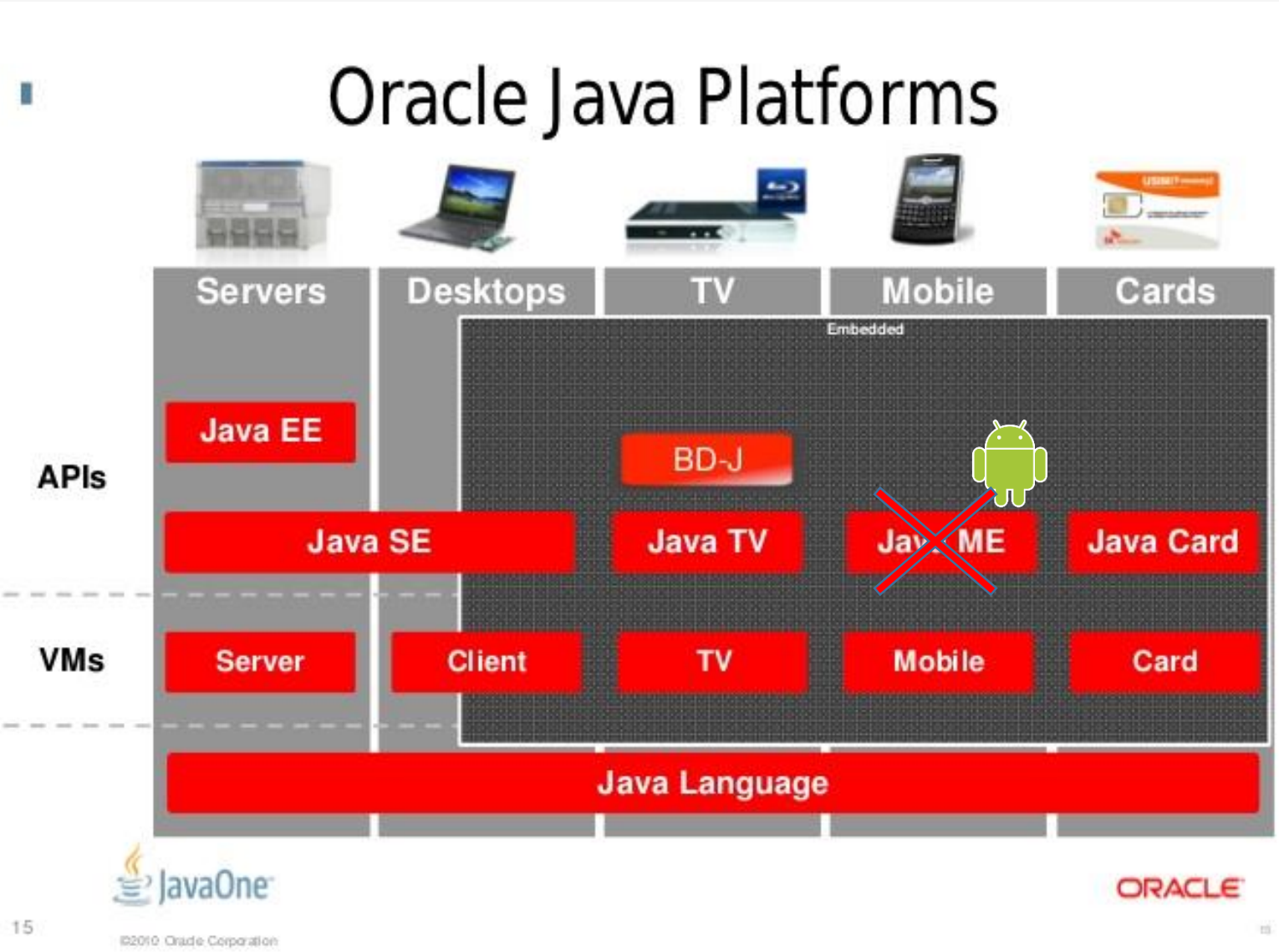

JDK (Java Development Kit)
javac
JRE (Java Runtime Environment)
java
JVM
JIT (Just In Time compiler)

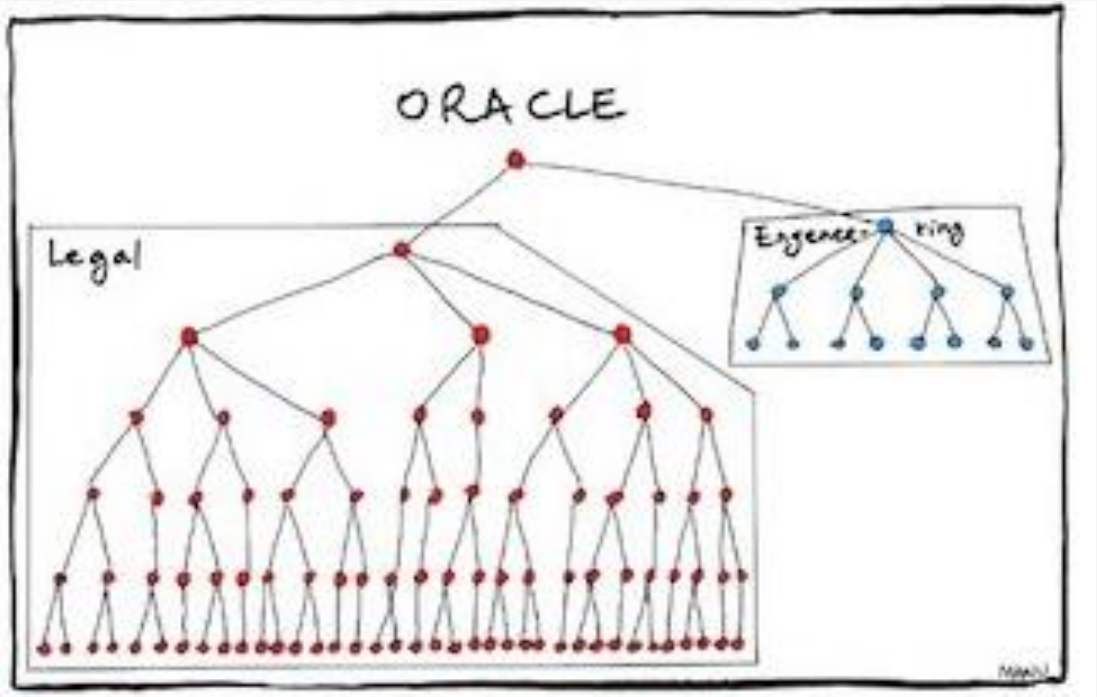
Z czego składa się Java?



Z czego składa się platforma Java?

- Java Platform, Standard Edition (**JavaSE**) – podstawowa wersja języka, pierwotnie z myślą o desktopach
- Java Platform, Enterprise Edition (**JavaEE**) – zbudowana na bazie JavaSE, do rozwijania aplikacji serwerowych. Uruchamiana na serwerze aplikacji (np. Glassfish)
- Java Platform, Micro Edition (**JavaME**) – Java na telefony komórkowe, obecnie de facto martwa. Java na smartfonach to przede wszystkim Android




ORACLE
Legal
Engineering

Z czego składa się platforma Java (biblioteki)?

- **Biblioteka** – zbiór klas, które są gotowe do wykorzystania w naszej aplikacji. Biblioteki znacząco ułatwiają nam pisanie programów, ponieważ zawierają gotowe rozwiązania na najczęstsze problemy.
- Biblioteki są dystrybuowane w postaci plików `.jar` (Java Archive)
- Najpopularniejsze biblioteki:
  - **Apache Commons** – zbiór bardzo popularnych bibliotek do różnych zastosowań, m.in.:
    - **Apache Commons IO** – operacje wejścia/wyjścia, na plikach
    - **Apache Commons Lang** – operacje na Stringach, liczbach
    - **Apache Commons Math** - operacje matematyczne
  - **Guava** – duża biblioteka od Google, rozszerza funkcjonalność kolekcji
  - **Gson** – biblioteka Google do zamiany JSONa na obiekty Javowe i vice versa

Z czego składa się platforma Java (frameworki)?

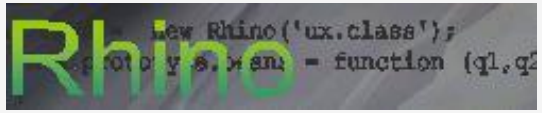
- **Framework** – szkielet aplikacji (lub jej części), który dostarcza dodatkowe funkcje, ułatwia pracę, wymusza porządek itd.
- Najpopularniejsze frameworki:
  - **Spring** (alternatywa dla EJB, do tworzenia aplikacji webowych, serwisów www itp.)
  - **JPA** – Java Persistence API (do obsługi baz danych, mapowanie obiektowo relacyjne. Najpopularniejsza implementacja: **Hibernate**)
  - **JSF** - JavaServer Faces (do tworzenia frontendu w Javie i HTMLu. Najpopularniejsza imlementacja: **PrimeFaces**)
  - **GWT** – Google Web Toolkit (do tworzenia frontendu w Javie i XMLu)

Z czego składa się ekosystem Javy?

Inne języki na JVM

- Skoro wystarczy skompilować kod do kodu bajtowego, zrozumiałego przez JVM - nic nie stoi na przeszkodzie, aby stworzyć jakiś inny język, który będzie mógł zostać skompilowany do kodu bajtowego i wykonać go na maszynie wirtualnej
- Przykłady języków:

- Można również napisać taki kompilator dla istniejących już języków, np.:
  - Jython (Python)
  - Jruby (Ruby)
  - Rhino (Javascript)



Inne wersje JVM

- JVM to **specyfikacja** – zbiór wymagań i regulacji które musi spełniać program, by być maszyną wirtualną Javy i uruchamiać programy Javy.
- Specyfikacja jest chroniona patentem i piecze nad nią sprawuje Oracle
- Dostępna pod adresem: https://docs.oracle.com/javase/specs/jvms/se10/html/
- Referencyjna (wzorcowa) implementacja specyfikacji JVM to **HotSpot (OpenJDK)**
- Oracle HotSpot JVM to OpenJDK HotSpot JVM z komercyjnymi dodatkami
- Są również inne JVM:
  - IcedTea (wolne oprogramowanie)
  - Eclipse OpenJ9 (dawniej IBM J9)
  - Dalvik (dla Androida, optymalizacja zajmowanego miejsca)
  - Jamiga (na komputery Amiga)


JAVA is to JAVASCRIPT
as HAM is to HAMSTER
ILLUSTRATED BY SEGUE TECHNOLOGIES

Ćwiczenie: Jak skompilować i uruchomić program?

javac i java

Ćwiczenie: Jak podejrzeć kod pośredni?

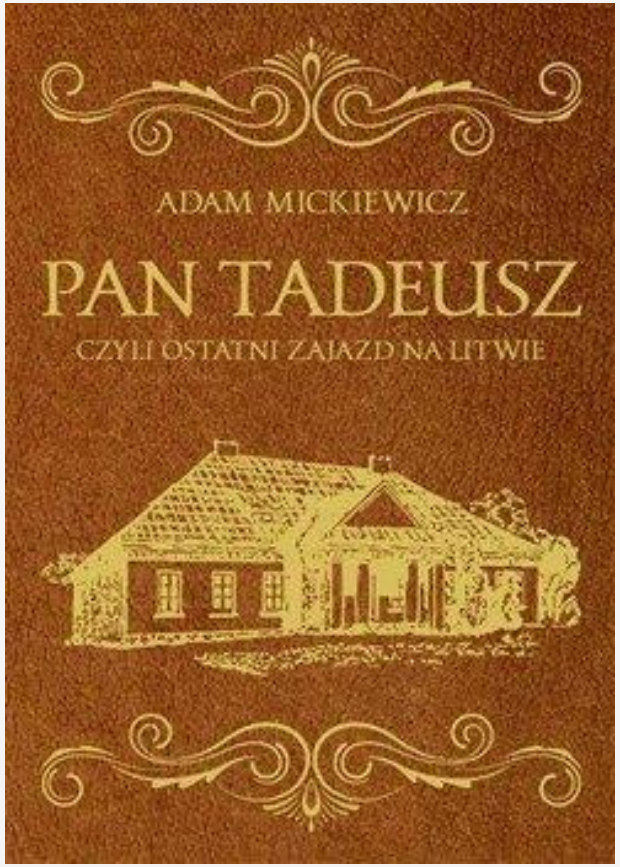
javap

Inne narzędzia konsolowe w JDK

- **`javaws`** – Java Web Start, pozwala łatwo pobrać i uruchomić aplikację Javy z Internetu - Java Network Launching Protocol (JNLP)
- **`javadoc`** – generowanie z komentarzy JavaDoc w kodzie dokumentacji w HTMLu
- `javaw` – to samo co java, tyle że “cicho”, bez okienka konsoli
- `jdb` – narzędzie do debugowania w okienku konsoli
- `javah` – generowanie nagłówków języka C do integracji kodu natywnego z Javą
- `jdeps` – analizowanie zależności pomiędzy klasami
- `jhat` - Java Heap Analysis Tool – analizowanie zrzutu sterty (heap dump)
- i wiele, wiele innych…

Ćwiczenie: Pan Tadeusz

- Zadanie: Napisz program, który wyświetli **pięć słów występujących najczęściej w Panu Tadeuszu**.
- Pana Tadeusza w formie tekstowej należy pobrać z https://wolnelektury.pl/media/book/txt/pan-tadeusz.txt
- Wskazówki:
  - Musimy wczytać plik, policzyć słowa (ile razy każde z nich występuje) i wypisać pięć najpopularniejszych (najczęściej występujących)
  - Co oddziela od siebie słowa w tekście?
  - Jakiej struktury danych należy użyć do przechowywania słów i liczby ich wystąpień?
- Zadania dodatkowe:
  - Zadbaj o ignorowanie interpunkcji, wielkich/małych liter
  - Ile jest słów, które występują tylko raz?
  - Ile rymów w książce się powtarza (występują więcej niż raz)?

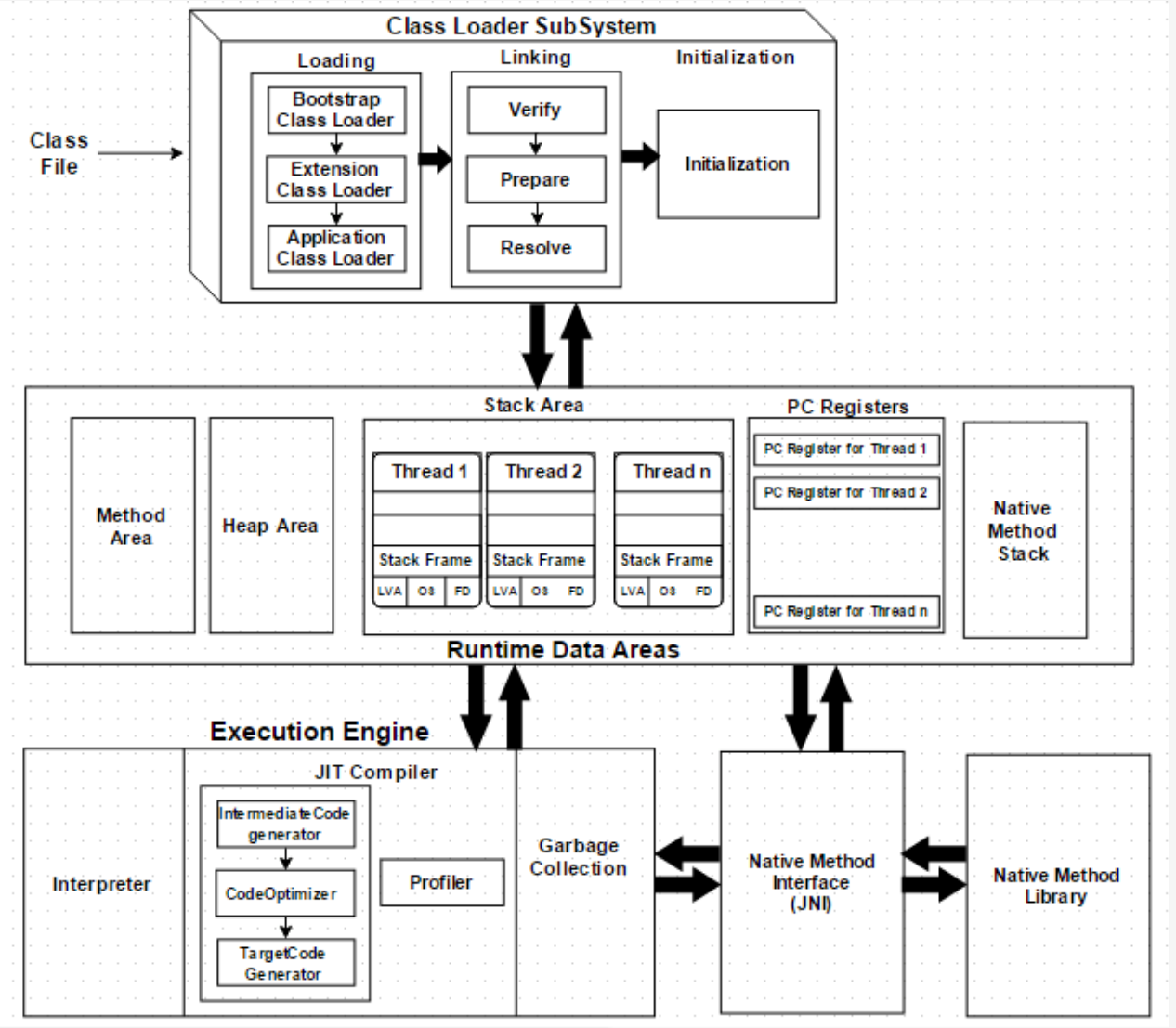


Z czego składa się JVM (HotSpot)?

- **Class loader** (ładowarka klas)
- Bytecode interpreter (interpreter kodu pośredniego)
- **JIT compiler** (Just In Time, kompilator kodu pośredniego do natywnego)
- **Garbage Collector** (odśmiecacz, do czyszczenia nieużywanej pamięci)
- Bytecode verifier (weryfikator kodu pośredniego)
- JNI (Java Native Interface) i Native Method Libraries (natywne, w C/C++/ASM)
- Java API (zestaw bibliotek, napisany w Javie)

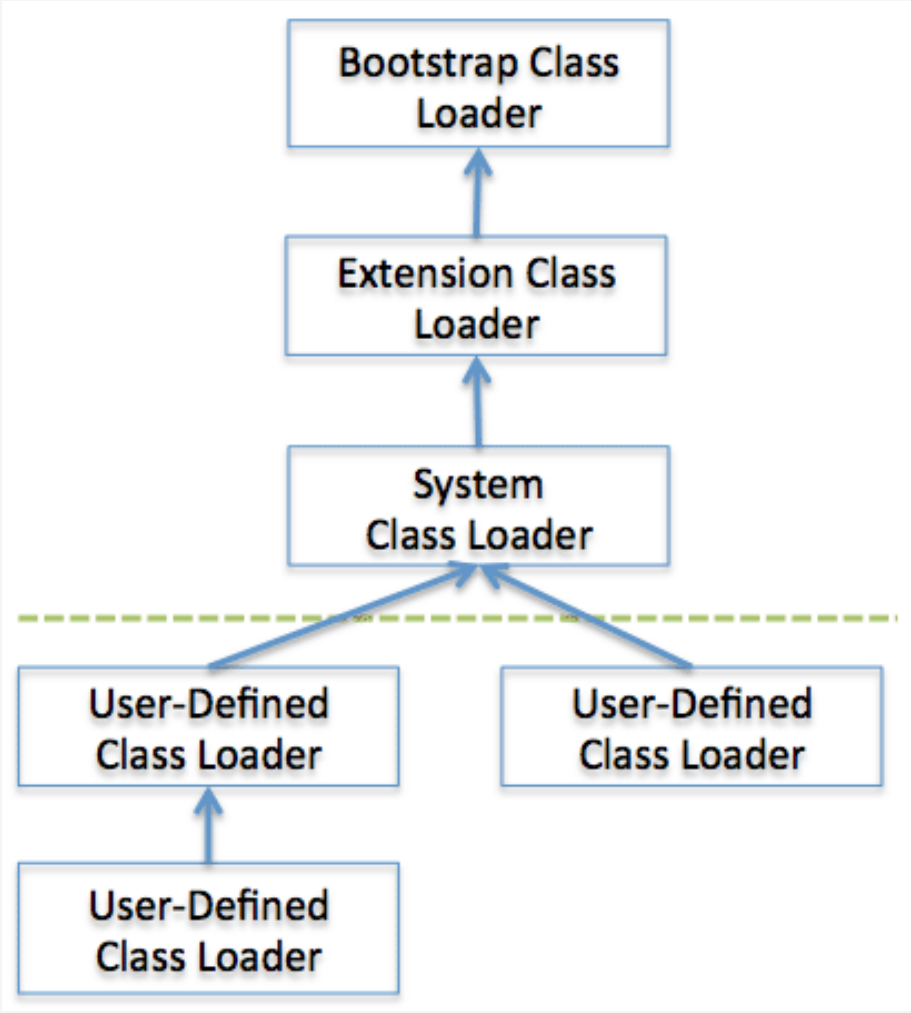
Jak wykonywany jest program Javowy?

- JVM składa się z trzech głównych podsystemów:
  - Classloader
  - Runtime Data Area
  - ExecutionEngine
- 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 ExecutionEngine, które wykorzystuje dane z RDA, interpretuje/kompiluje bytecode tworząc kod maszynowy
  5. Wykonanie kodu



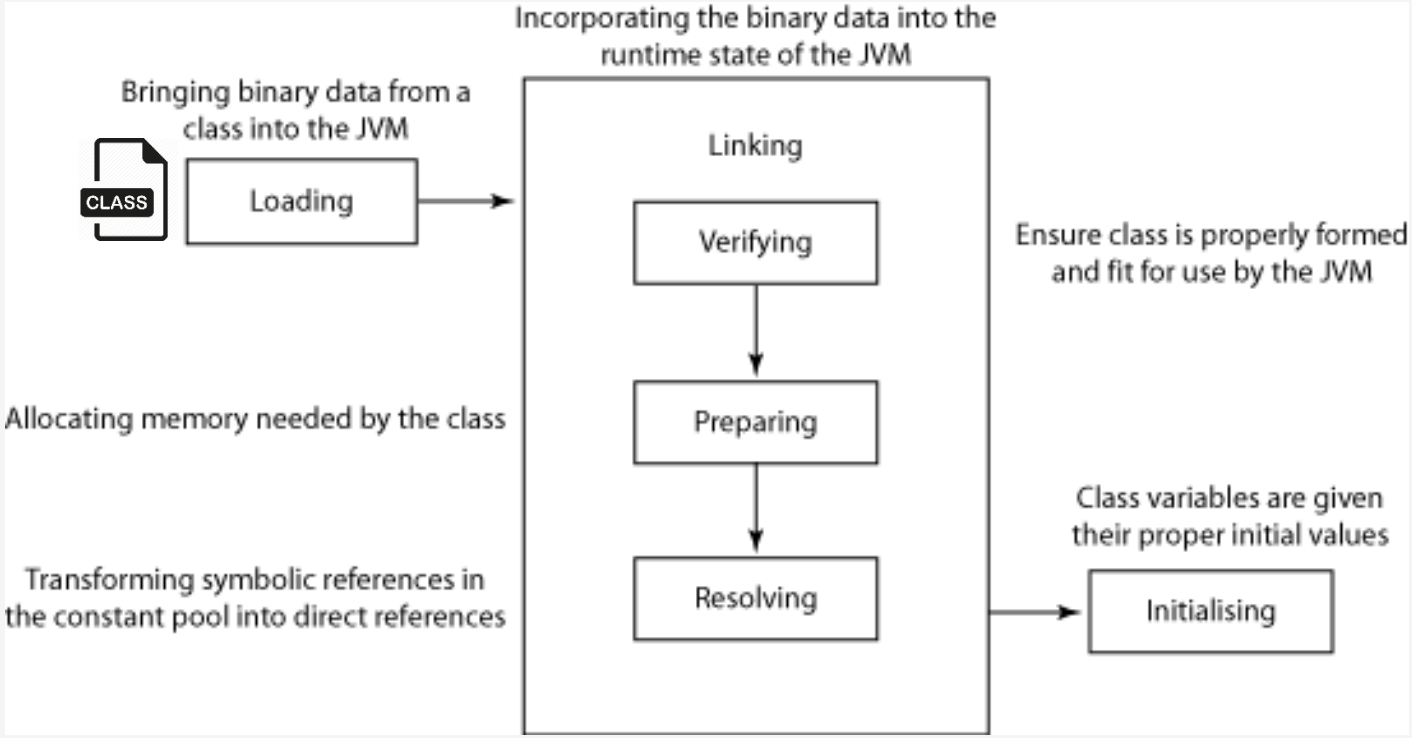
Class Loader

- **Class loader** – ładuje do pamięci klasy dynamicznie (leniwie) – dopiero wtedy, kiedy są potrzebne.
- Kiedy potrzebujemy wczytać klasę?
  - Kiedy chcemy stworzyć nowy obiekt tej klasy: `Dog reksio = new Dog();`
  - Kiedy korzystamy z pól/metod statycznych klasy: `System.out.println(…)`
- Class loader jest hierarchiczny, zawsze sprawdza czy klasa została już załadowana przez jego rodzica.



Ładowanie klas za pomocą Class Loadera

- **Ładowanie** (wczytanie pliku `.class`)
- **Łączenie** (linkowanie)
  - **Weryfikacja** – sprawdzenie, czy wczytany plik jest poprawny składniowo
  - **Przygotowanie** – przygotowanie pól statycznych klasy, nadanie im domyślnych wartości i inne
  - **Rozwiązywanie** – wczytanie wszystkich innych potrzebnych klas
- **Inicjalizacja** – nadanie wartości polom statycznym klasy, wykonanie bloku `static { }`



*UWAGA! To **ładowanie klasy**, a nie tworzenie obiektu – więc nie ma wywołania konstruktora!*

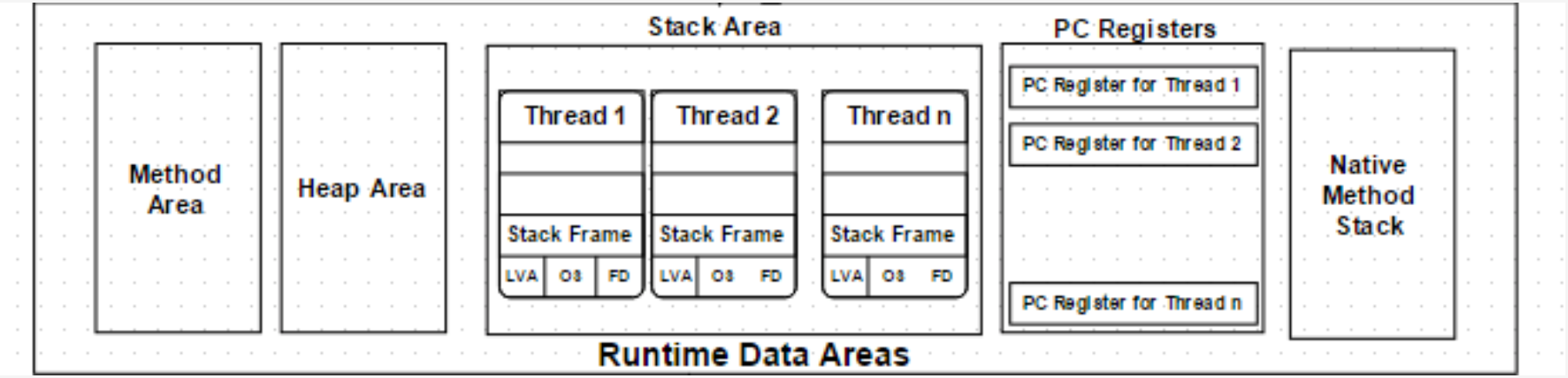
Execution Engine - JIT Compiler

- Kod zawarty w plikach `.class` jest interpretowany przez JVM, a dokładniej **ExecutionEngine**
  - *Czytam linijka po linijce tekst metody i każdą linijkę „sprawdzam w słowniku” na jaki ciąg zer i jedynek się tłumaczy, a następnie go wykonuje*
- **Kompilator JIT** (*Just In Time compiler*) pozwala na kompilację kodu „w locie” w trakcie działania programu
  - Standardowy kompilator np. języka C kompiluje cały kod programu do pliku .exe (gdzie zapisany jest tylko kod maszynowy dla danej platformy)
  - JIT decyduje się na kompilację danego kawałka kodu dopiero jak wykryje, że jest on często używany (np. często wywoływana metoda)
  - W trakcie kompilacji JIT dokonuje szeregu optymalizacji, dzięki czemu powstały kod maszynowy jest lepszy od kodu wykonywanego przy intepretowaniu. Kod maszynowy trafia do cache’a.
  - *Ta metoda jest często wykonywana, więc przeczytam ją w całości i przetłumaczę najlepiej jak umiem na zera i jedynki, a potem zapisze sobie na półeczkę i następnym razem jak będę chciał ją wykonać, to użyje przygotowanej wersji*
- Jeżeli dany kod wykonuje się co najwyżej kilka razy – nie opłaca się kompilować za pomocą JITa – „szkoda czasu”. Jeżeli częściej – inwestycja w kompilację nam się opłaci.

Runtime Data Area

Runtime Data Area - obszar pamięci przydzielony javie przez system operacyjny na którym JVM działa. Obszar ten można podzielić na kilka części:

- pamięć współdzielona przez wszystkie wątki:
  - **Heap Space** (sterta) - tutaj przechowywane są wszystkie obiekty używane w programie.
  - **Meta Space** - 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** – stos wykorzystywany do wywołania kodu C/C++ poprzez JNI (Java Native Interface)



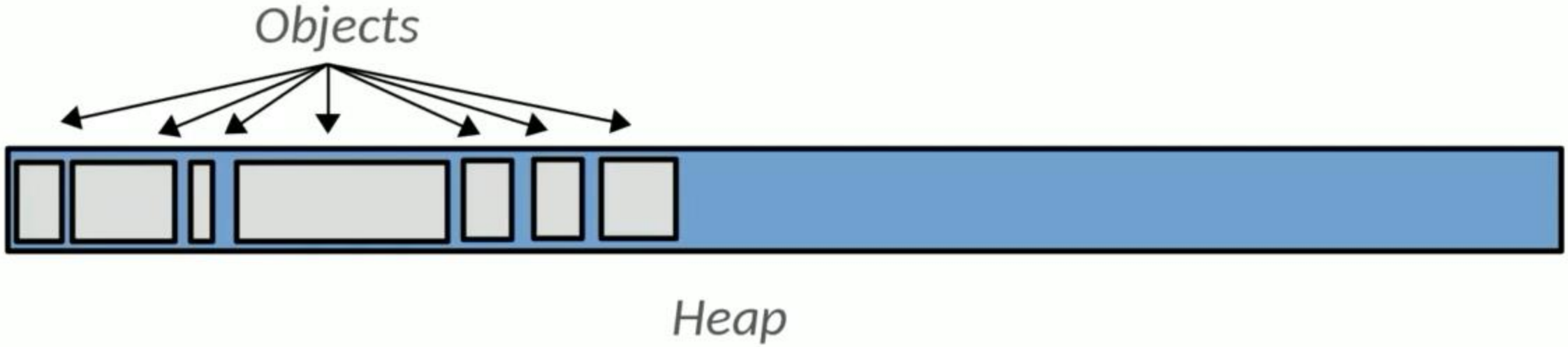
Garbage Collector

- **Garbage Collector** (odśmiecacz) – mechanizm oczyszczana pamięci JVM z nieużywanych danych.
- Obiekty w Javie mają prosty cykl życia – tworzymy je, używamy, a następnie **porzucamy** (nie niszczymy!).
- Niszczeniem zajmuje się GC, kiedy odkryje, że obiekty są porzucone i nikt nigdy ich nie będzie mógł użyć.
- Dzięki obecności GC programista nie musi myśleć o usuwaniu niepotrzebnych obiektów (jak w C/C++) – wystarczy, że je porzuci.

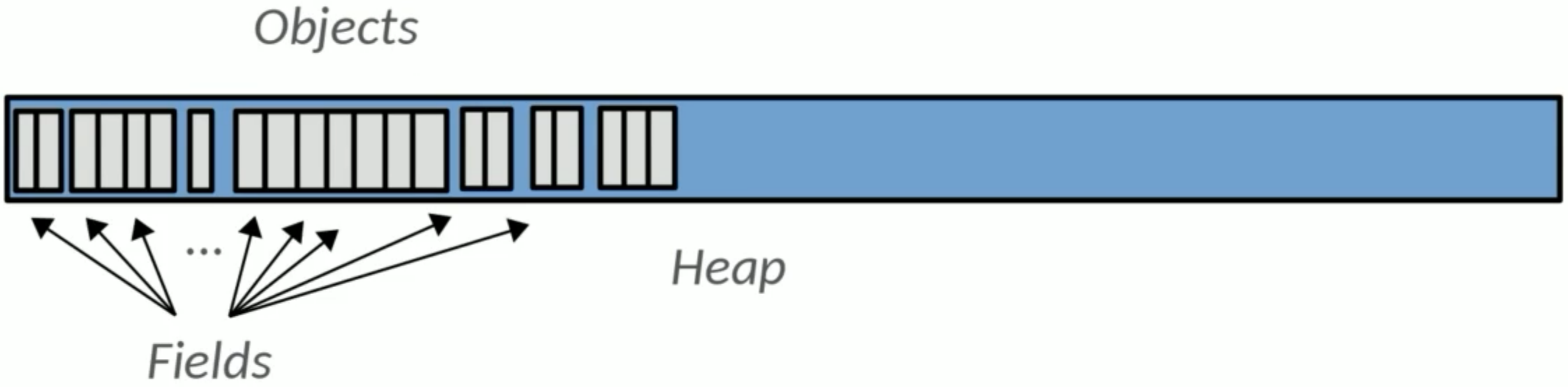
Idea Garbage Collectora

Heap

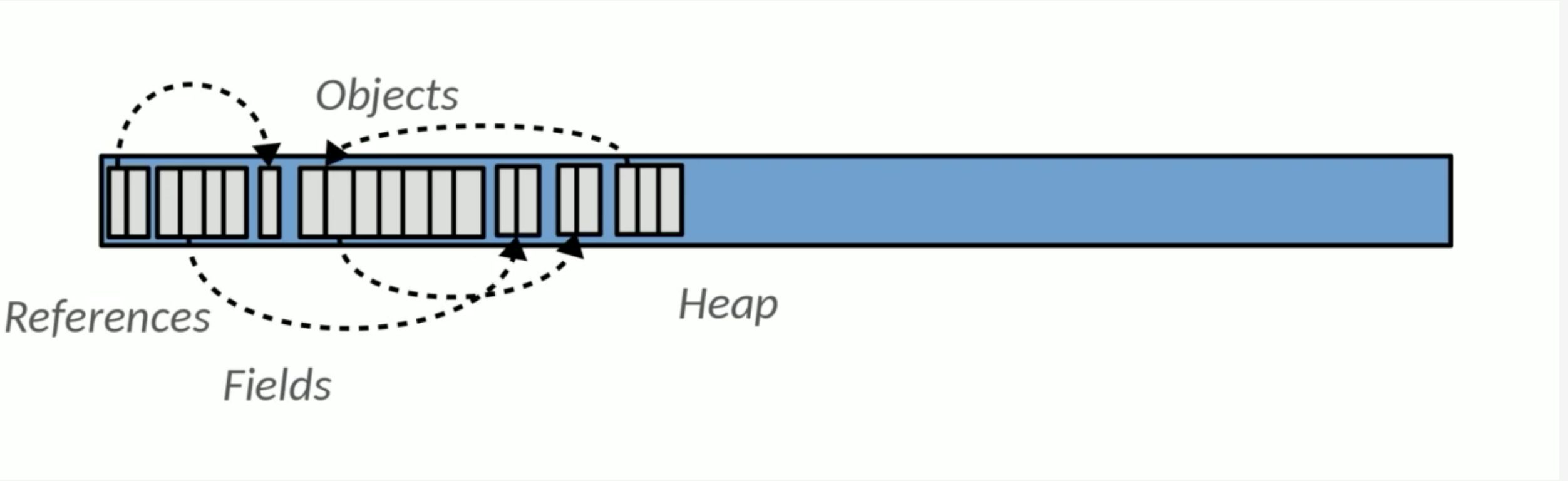
Idea Garbage Collectora



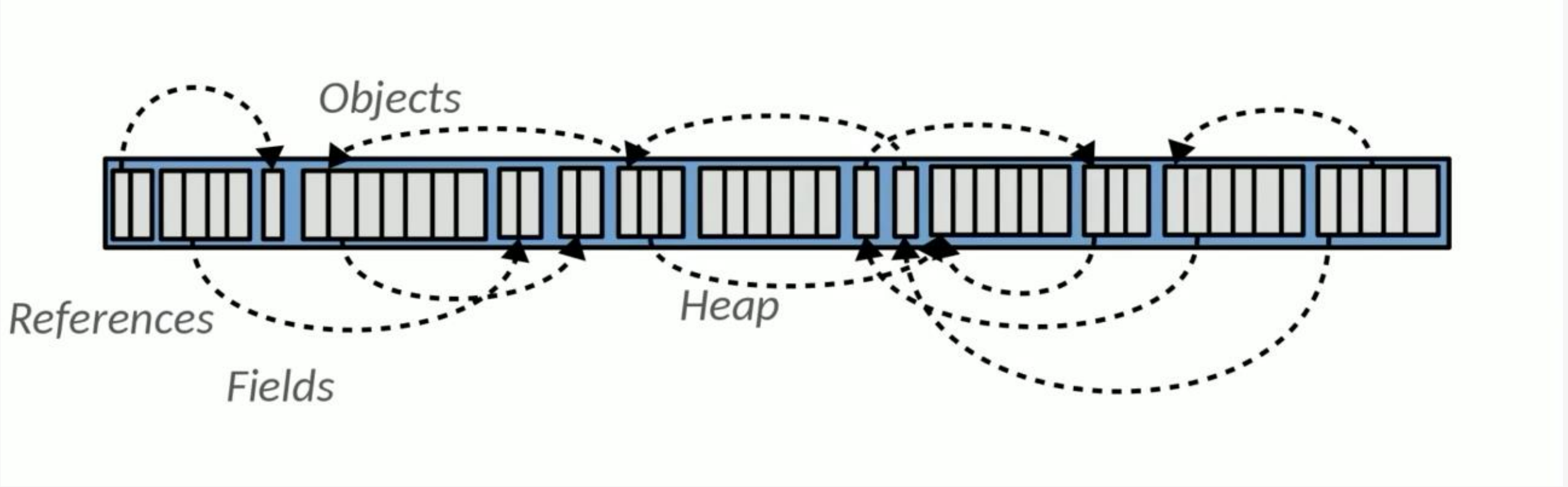
Idea Garbage Collectora



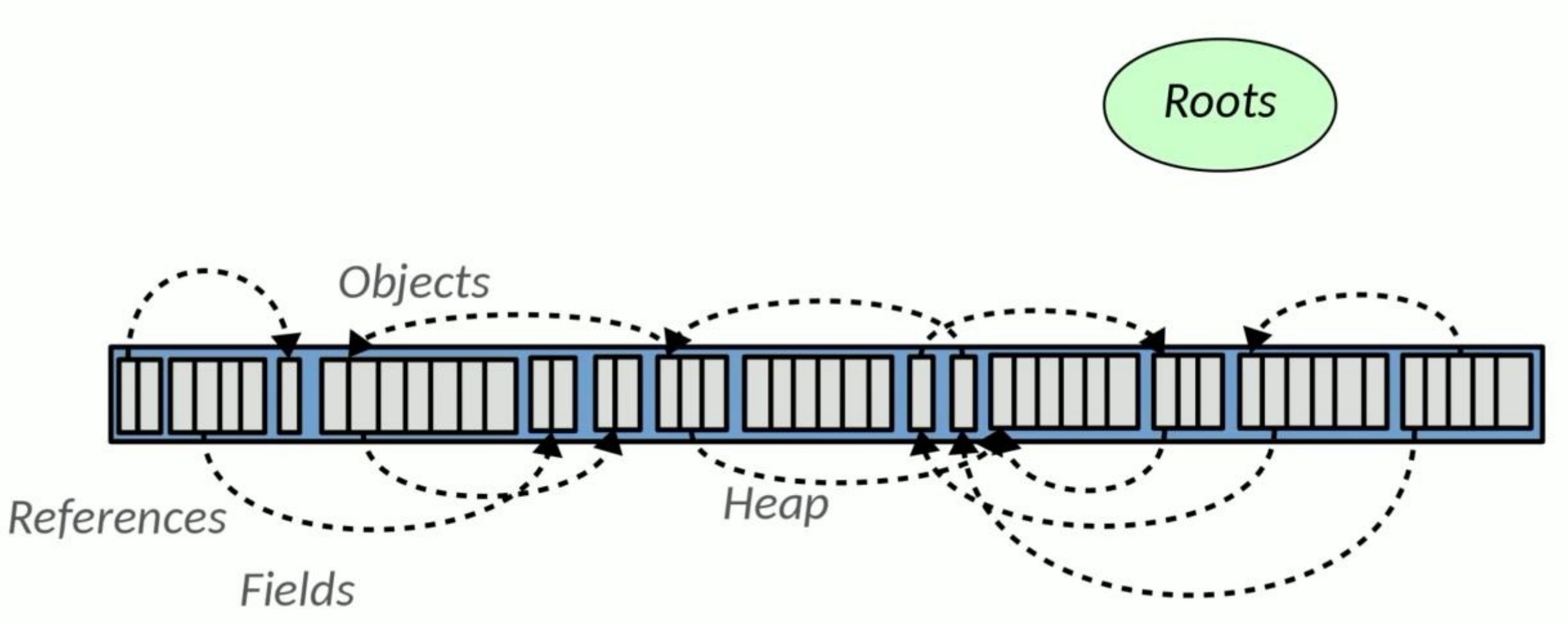
Idea Garbage Collectora



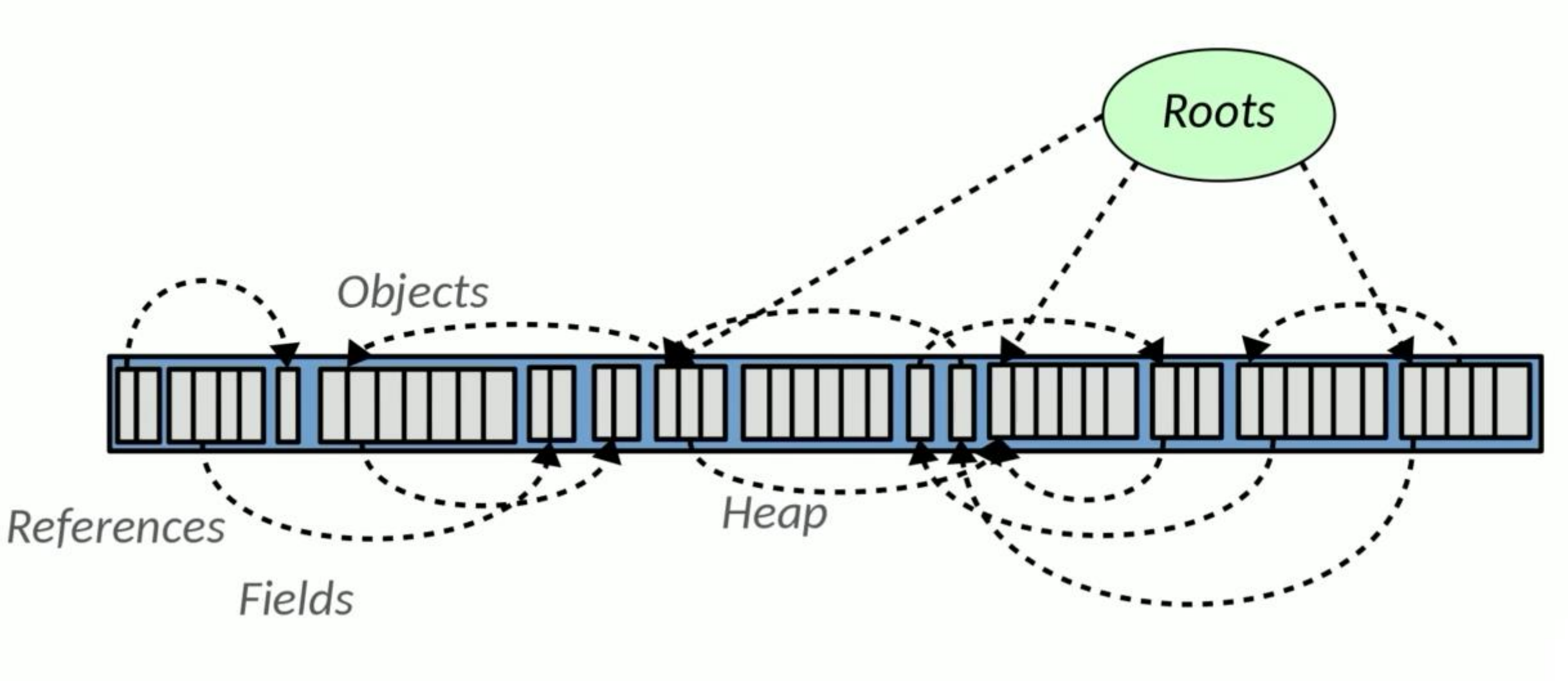
Idea Garbage Collectora



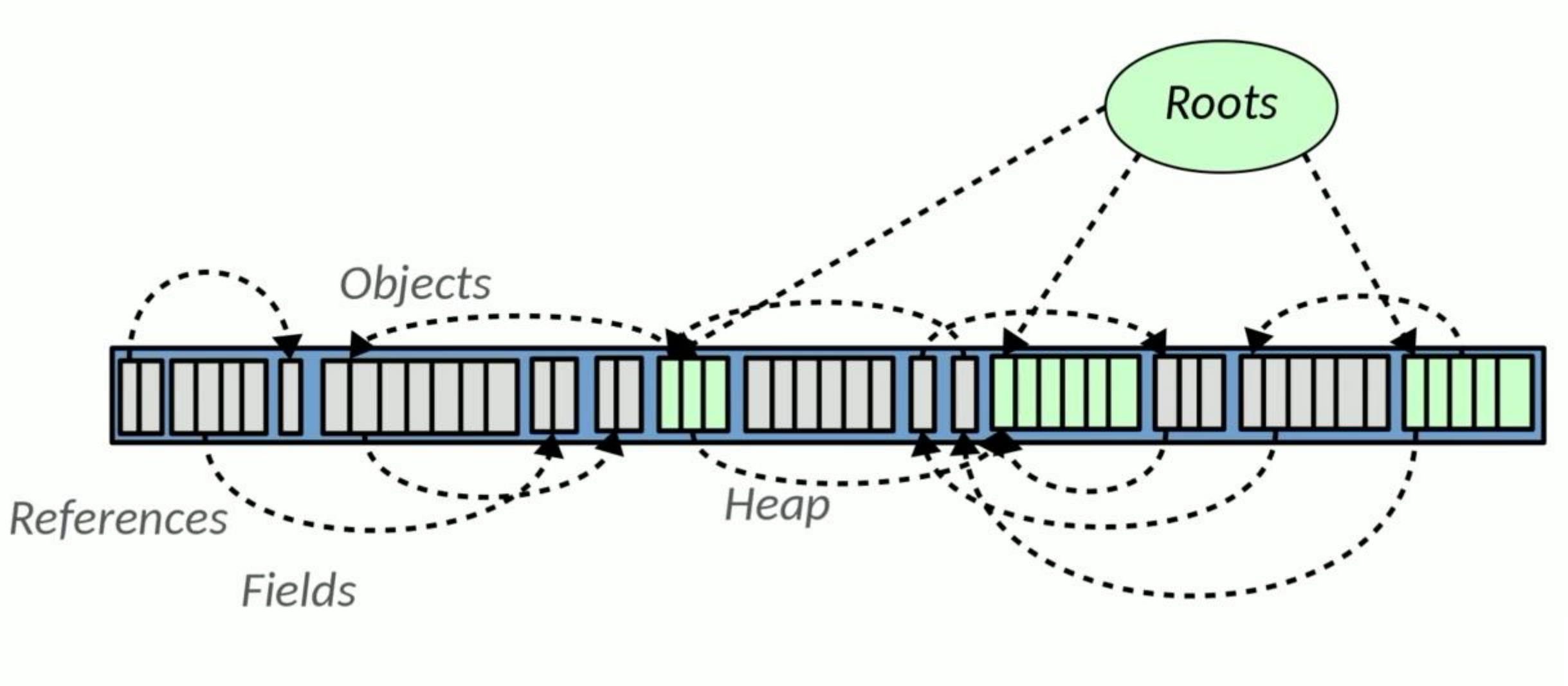
Idea Garbage Collectora



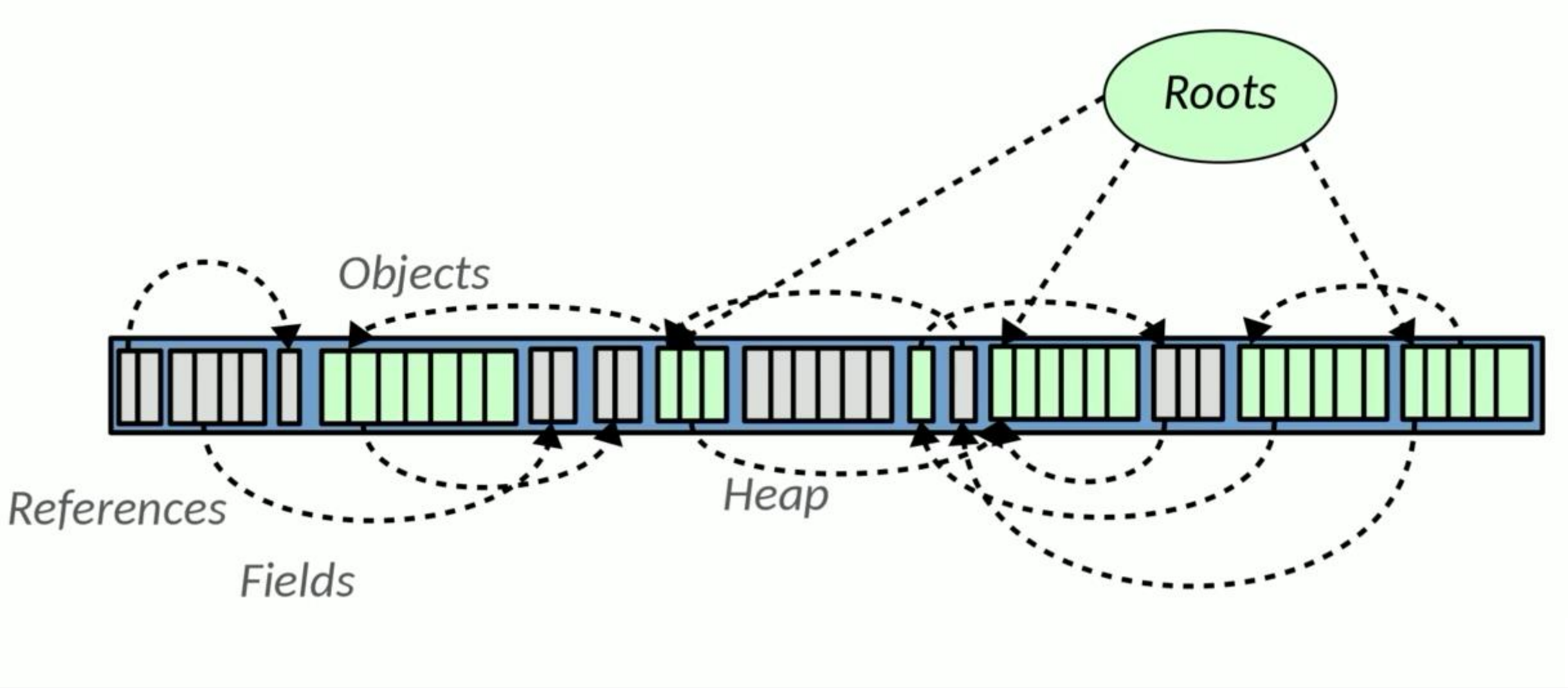
Idea Garbage Collectora



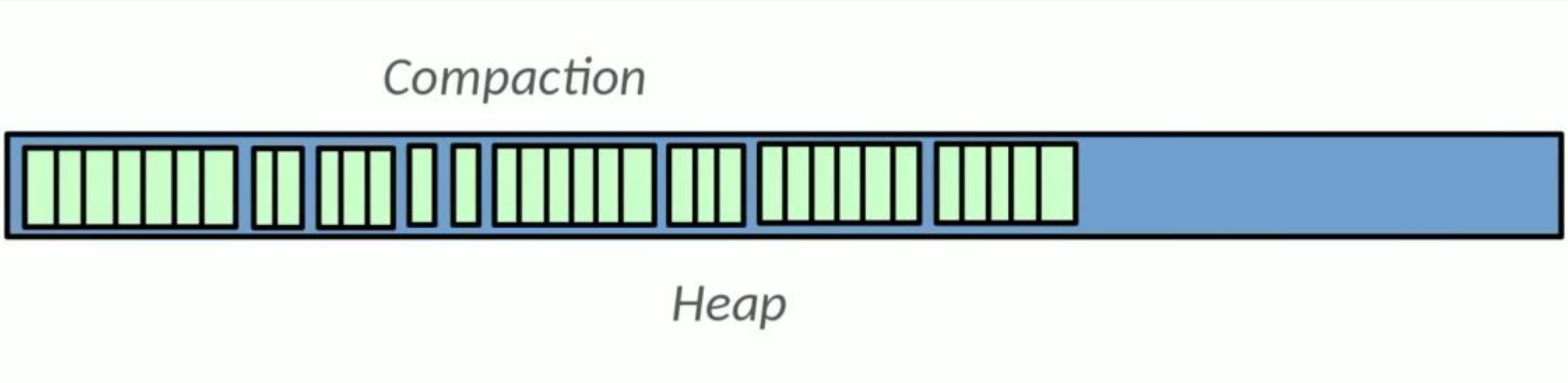
Idea Garbage Collectora



Idea Garbage Collectora



Idea Garbage Collectora



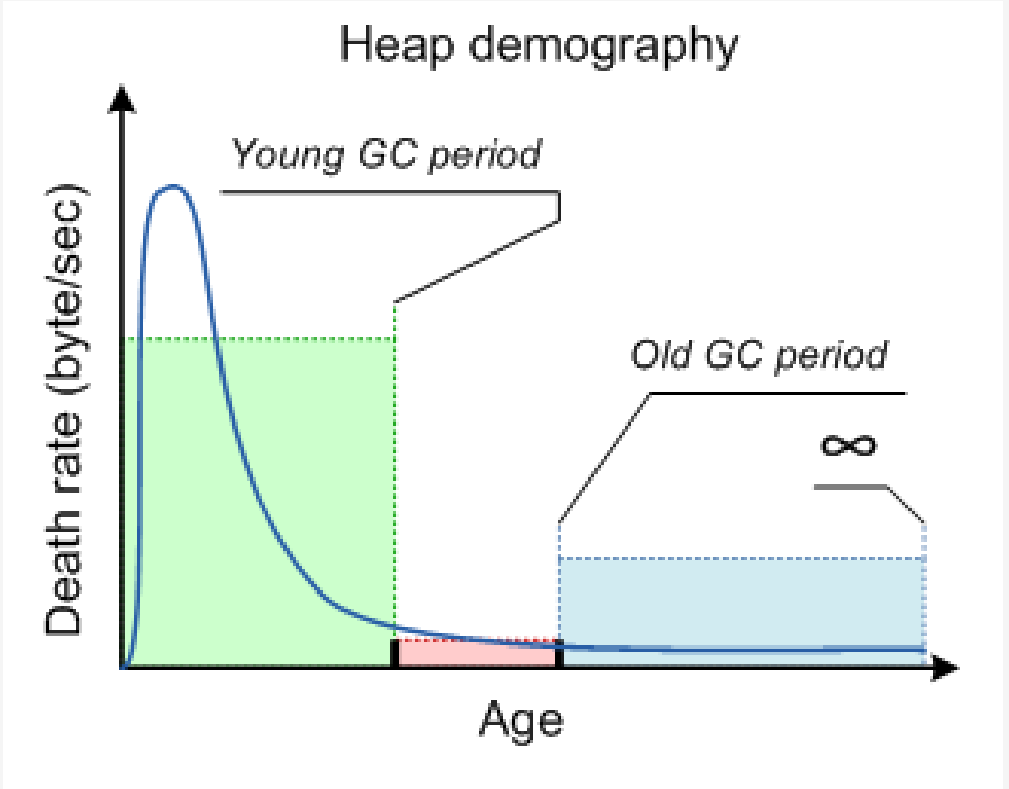
Ćwiczenie: Oglądamy działający GC

jvisualvm i jmc

Śmiertelność niemowląt

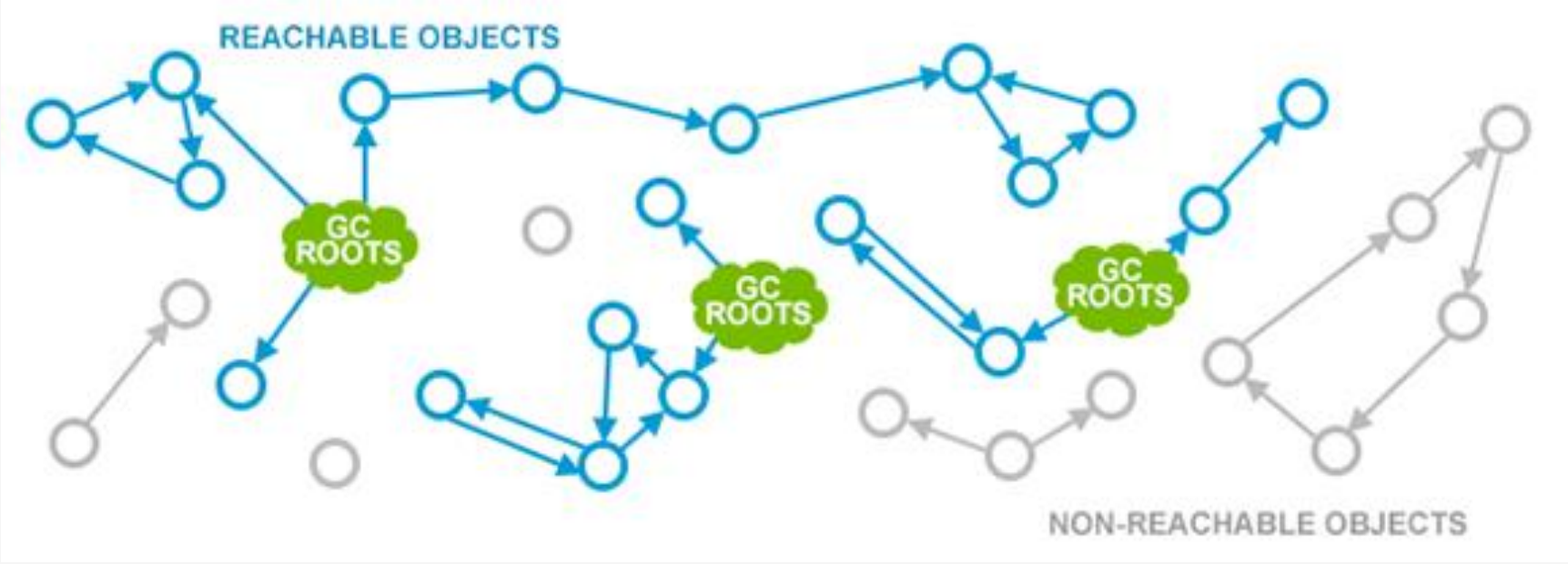
Hipoteza generacyjna (śmiertelność niemowląt)

- Działanie GC opiera się na tzw. hipotezie generacyjnej (Generational Hypothesis), która zakłada, że:
  - Większość obiektów szybko przestaje być używana przez program (zostają „porzucone”)
  - Te, które przetrwają pewien czas, zwykle żyją bardzo długo
- Mając taką wiedzę, jako JVM chcemy „śledzić losy” młodych obiektów, bo jest duża szansa, że będziemy mogli się ich pozbyć

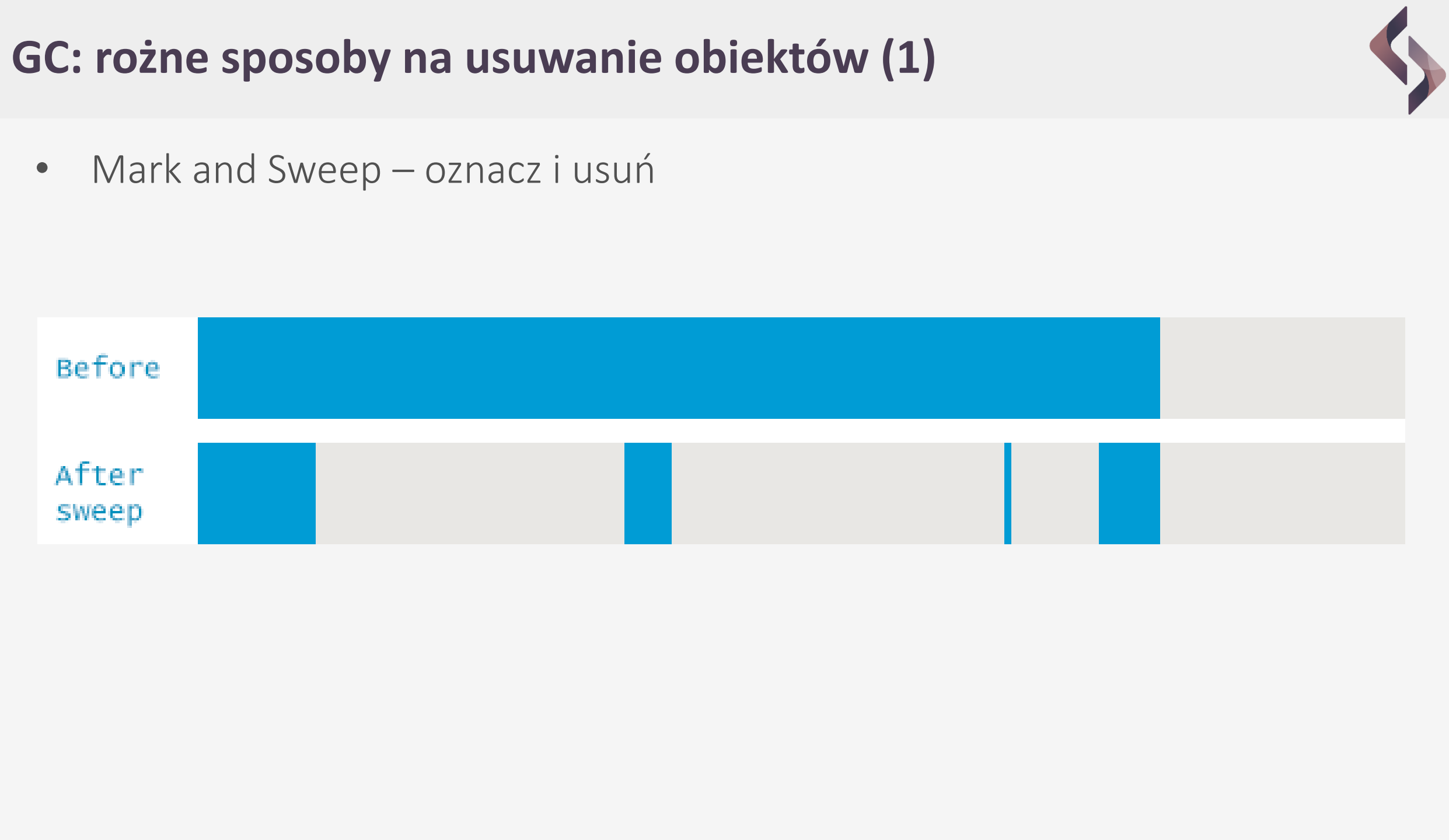


Korzenie GC (GC roots)

- Oznaczanie obiektów, które są żywe rozpoczynamy od tzw. **korzeni GC**
- Przykłady korzeni:
  - Aktywne wątki (w tym wątek „main”)
  - Zmienne lokalne
  - Pola statyczne
  - Referencje JNI



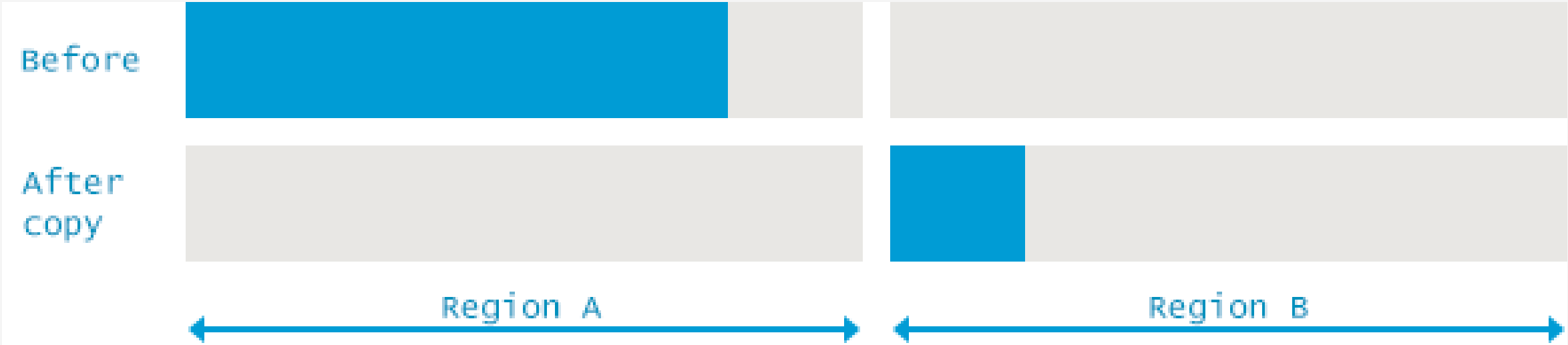
GC: rożne sposoby na usuwanie obiektów (1)



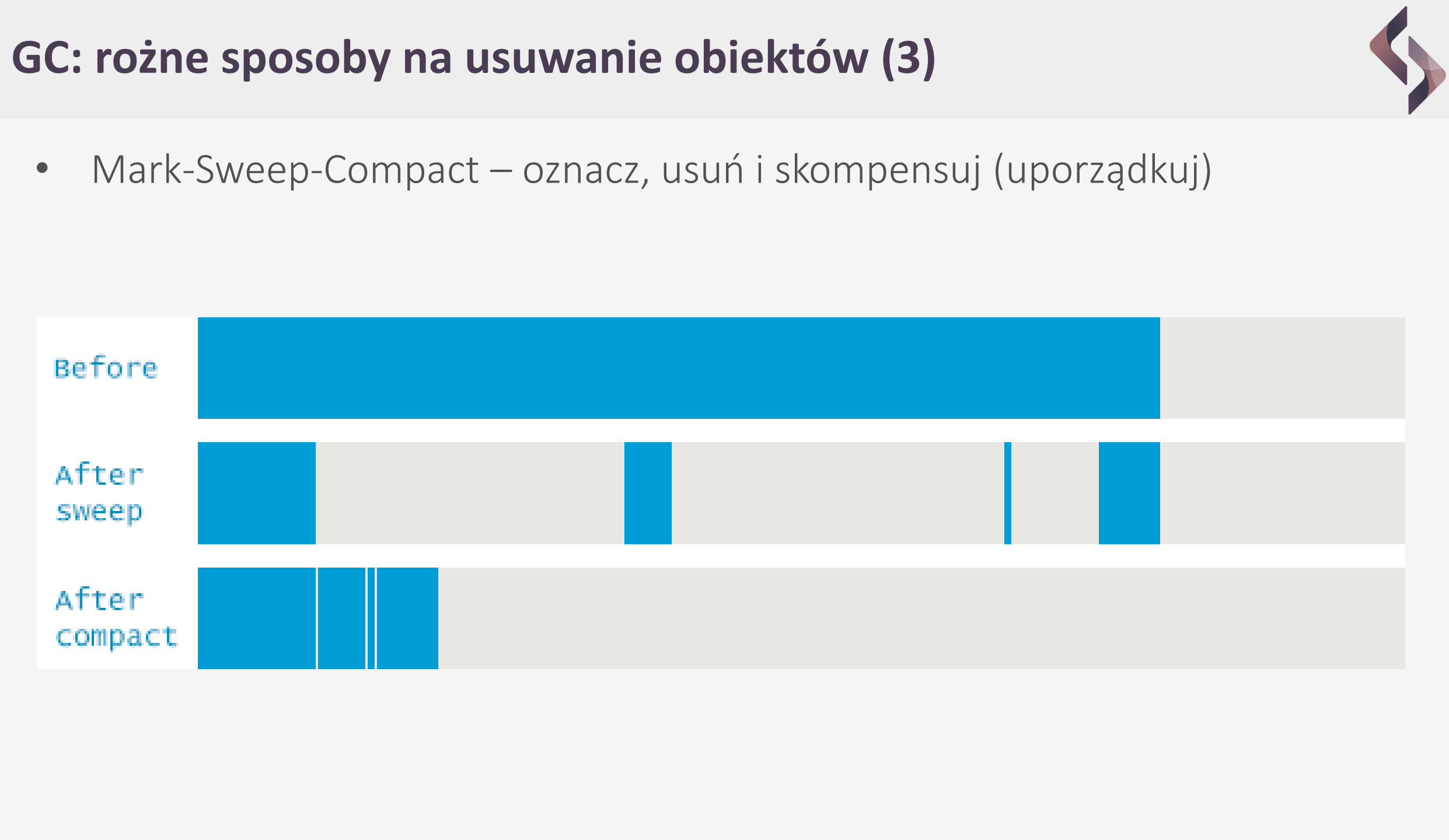
- Mark and Sweep – oznacz i usuń

GC: rożne sposoby na usuwanie obiektów (2)

- Mark and Copy – oznacz i skopiuj (przenieś)



GC: rożne sposoby na usuwanie obiektów (3)

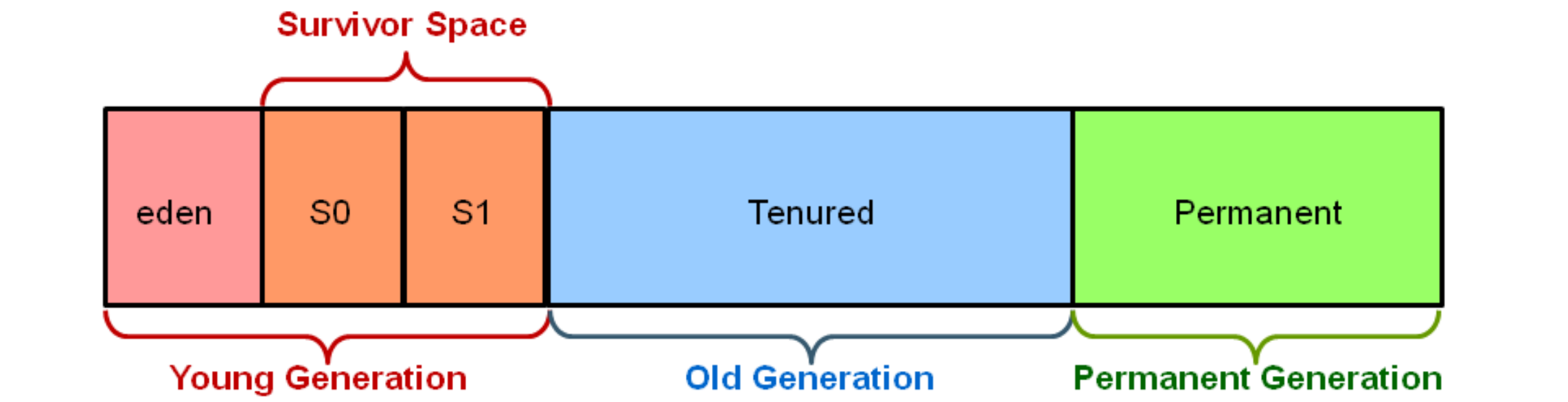


- Mark-Sweep-Compact – oznacz, usuń i skompensuj (uporządkuj)

Struktura sterty JVM

Całą pamięć przeznaczoną na stertę (heap) dzielimy na kilka obszarów:

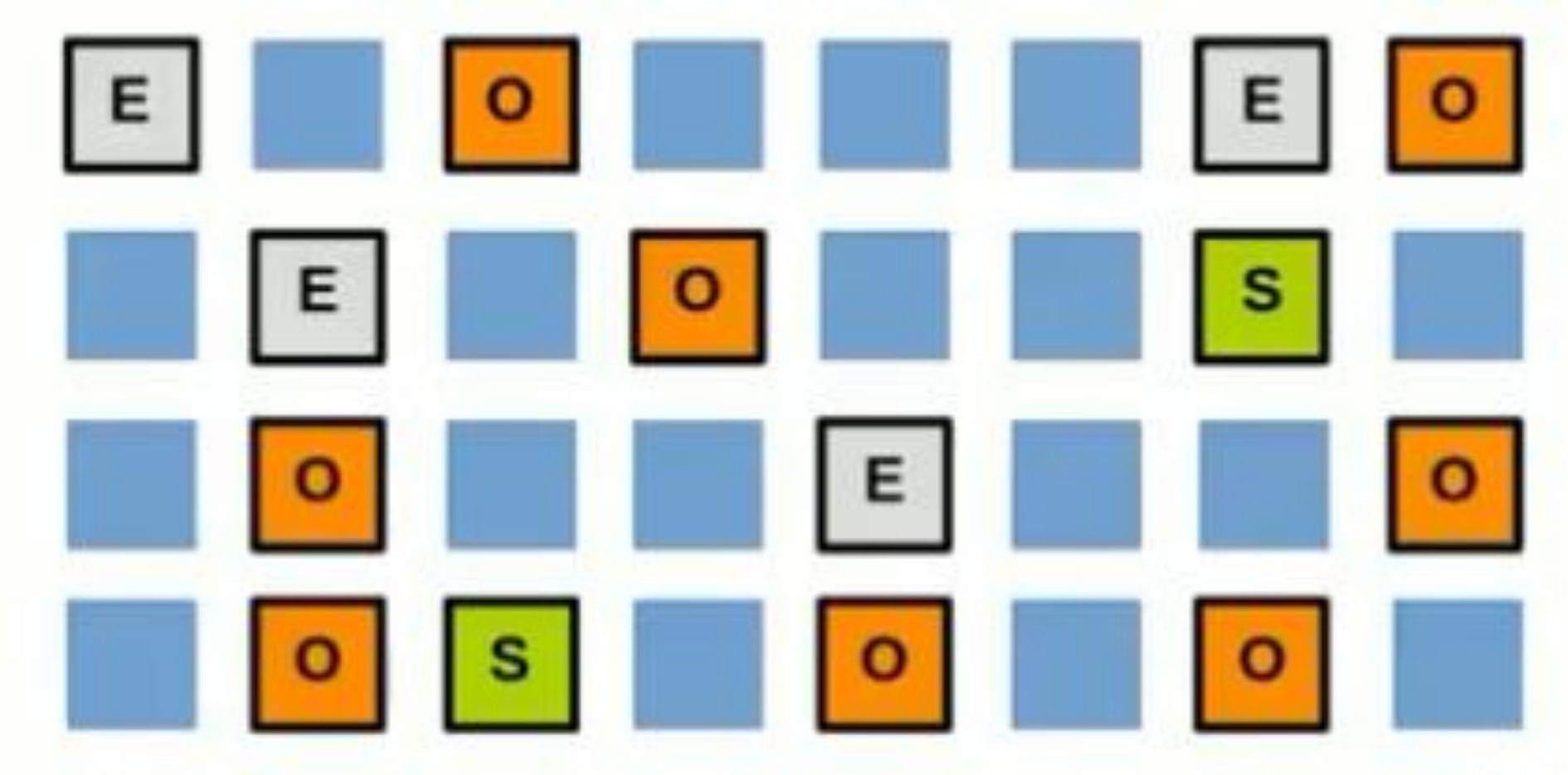
- Młoda generacja (young generation) – krótko żyjące obiekty
  - Eden (miejsce na nowe obiekty)
  - Przetrwalniki S0 i S1 (survivor space S0/S1)
- Stara generacja (old generation, tenured) – długo żyjące obiekty
- Stała generacja (permanent generation) – miejsce na załadowane klasy, kod itp.



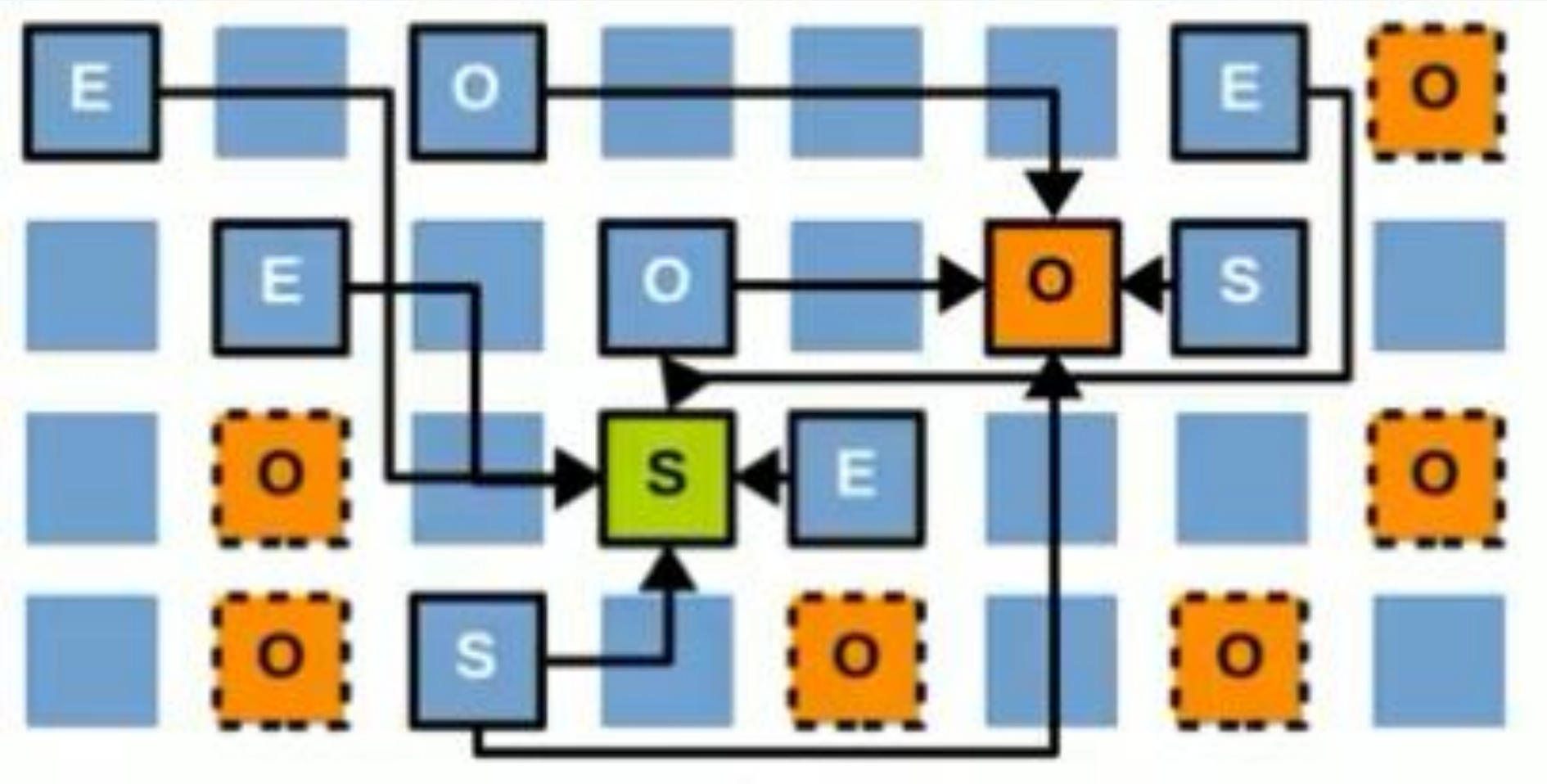
Rodzaje GC

- **Serial GC** – 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** – 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** – 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).
- **G1 GC** – Garbage First – następca algorytmu CMS. Stworzony po to by pauzy stop-the-world były bardziej przewidywalne i konfigurowalne. 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.

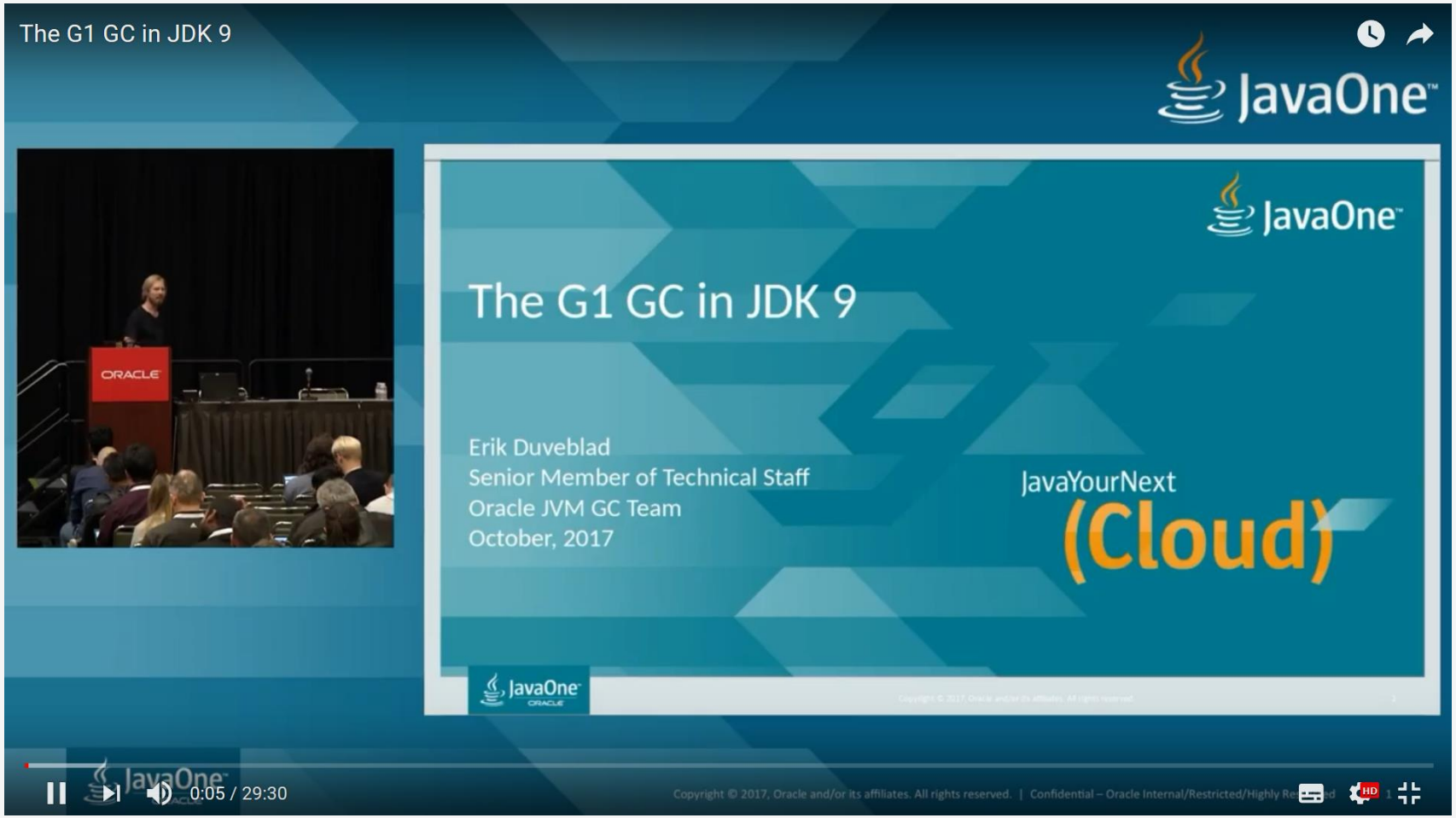
G1 GC



G1 GC



G1 GC



https://www.youtube.com/watch?v=OhPGN2Av44E

Kiedy mój obiekt umrze?

- **Nie wiadomo kiedy zadziała GC** – zależy to od konkretnej implementacji JVM, wybranego algorytmu GC, jego konfiguracji, tempa alokacji nowych obiektów i innych czynników…
- Mamy do dyspozycji ręczne wywołanie odśmiecania - metodę `System.gc()`, natomiast jest to jedynie sugestia dla JVM, że programista chciałby „mieć sprzątanie” w tym momencie. Wywołanie tej metody wcale nie musi zakończyć się podjęciem jakiejkolwiek akcji (i często tak właśnie jest).