



Wprowadzenie do technologii JVM

Łukasz Bednarski



Jak to się dzieje że napisany ciąg znaków w IntelliJ tworzy program i wykonuje zdefiniowane operacje?



Charakterystyka procesora

1. Układ scalony składający się z tranzystorów
2. Potrafi pobierać rozkazy z pamięci operacyjnej interpretować je oraz wykonywać.
3. Zna kilkadziesiąt różnych instrukcji - zwykle bardzo prostych (na przykład typu: „skopuj bajt z jednego adresu w pamięci pod inny”, „dodaj do siebie dwie liczby” lub „wyślij pewną wartość na odpowiedni port”).
4. Procesor, jak większość urządzeń cyfrowych, posługuje się tylko dwoma wartościami: 0 i 1.

Programowanie to pisanie ciągów instrukcji (programu), który ma wykonać procesor. Instrukcje te powinny być zapisane w postaci ciągów zer i jedynek, aby procesor był w stanie je poprawnie zinterpretować (zrozumiały dla procesora język zer i jedynek nazywamy **językiem maszynowym**).



Poziomy języków

Język niskiego poziomu - typ języka programowania, który w małym stopniu abstrahuje od konstrukcji jednostki centralnej komputera. Innymi słowy, język ten wykazuje duże podobieństwo do kodu maszynowego, zaś kompilacja jest w miarę nieskomplikowana.

Najbardziej typowym przykładem języka niskiego poziomu są asemblerы.

W asemblerach zasadniczo jedno polecenie odpowiada jednemu rozkazowi procesora. Są to języki powstałe na bazie języka maszynowego poprzez zastąpienie liczb odpowiadających fragmentom rozkazów kodu maszynowego ich symbolicznymi odpowiednikami. Dzięki zamianie liczb na tzw. mnemoniki można pisać programy w miarę zrozumiałe dla człowieka, a jednocześnie bezpośrednio tłumaczone na kod maszynowy procesora, co pozwala zapewnić duży stopień kontroli programisty nad zachowaniem procesora.

Współcześnie praktycznie nie używa się asemblera do pisania całych programów dla komputerów osobistych. Jest on za to wciąż używany do pisania fragmentów wymagających bardzo wysokiej wydajności lub mających inne specjalne wymagania, np. dla oprogramowania mikrokontrolerów o niewielkich rozmiarach pamięci programu.



Języki wysokiego poziomu

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.

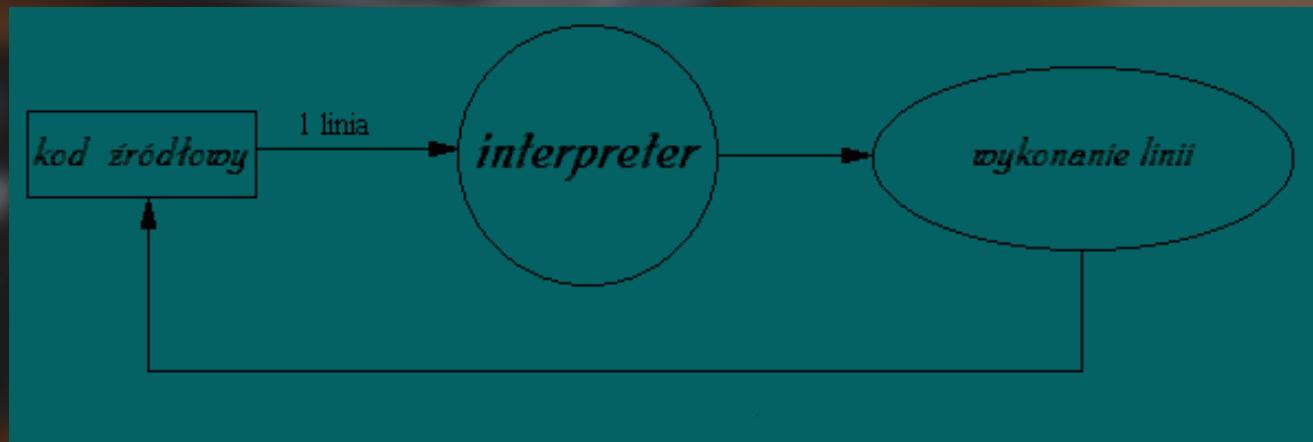
Kod napisany w języku wysokiego poziomu nie jest bezpośrednio „zrozumiały” dla komputera - większość kodu stanowią tak naprawdę normalne słowa, np. w języku angielskim. Aby umożliwić wykonanie programu napisanego w tym języku należy dokonać kompilacji lub interpretacji.



Interpreter

Programy napisane w językach interpretowanych, są wykonywane przez interpreter linijka po linijce i nie muszą być wcześniej skompilowane. Programista nie musi podczas każdej edycji kodu źródłowego kompilować programu, co w środowisku w którym modyfikacje kodu są codziennością jestową zaletą. Sposób działania

1. Pobierz jedną instrukcję
2. Przetłumacz na język maszynowy
3. Przekaż procesorowi do wykonania
4. Wróć do pkt 1.

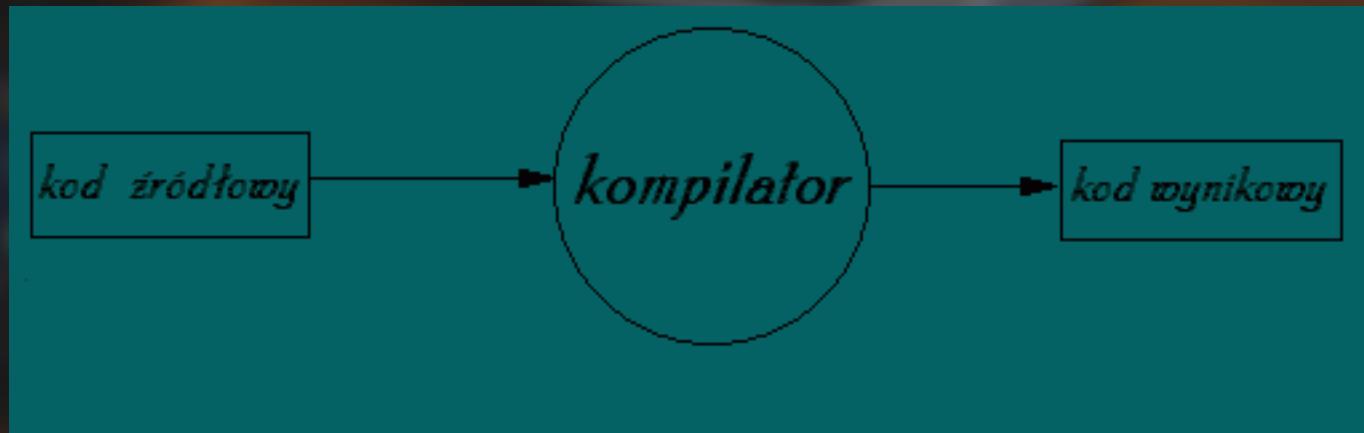




Kompilator

Program napisany w języku komplelowanym, zanim zostanie uruchomiony musi zostać skompilowany do kodu maszynowego. Co więcej, kompilator zazwyczaj tworzy kod maszynowy, który działa jedynie na określonej architekturze np. tylko na procesorach x86. Niektóre języki, jak na przykład Java komplują kod źródłowy do kodu pośredniego zwanego bajtkodem. Taki kod pośredni jest następnie wykonywany przez maszynę wirtualną. Sposób działania:

1. Przetłumacz cały program
2. Zapisz przetłumaczony program w pamięci komputera lub na dysku





Podsumowanie

Rodzaj podziału	Opis	Przykład
poziom wykonywania programu	wysokiego poziomu	Pascal, C i inne wymienione w tej tabeli poza Assemblerami oraz Cg
	niskiego poziomu (poziom maszynowy)	Assembler, Cg (programowanie kart graficznych)
sposób wykonania	interpretowane	Basic, JavaScript, PHP, LOGO
	kompilowane	Pascal, C, C++, Java
zastosowanie	tworzenie aplikacji internetowych	Java, JavaScript, PHP
	dostęp do baz danych	SQL
	obliczenia matematyczne	Fortran
	dydaktyczne	LOGO
	inne (uniwersalne)	Pascal, C, C++
	programowanie wizualne	Visual C, Visual Basic, Delphi, Kylix
	opis danych	PostScript, HTML, XML
	tworzenie aplikacji współbieżnych	Ada, Occam
	przetwarzanie tekstu	PERL, REXX, Python
	programowanie sztucznej inteligencji	LISP, Prolog
	programowanie grafiki	OpenGL
	liniowe	BASIC, Fortran
model programowania	strukturalne	Pascal, C
	zdarzeniowe	Visual Basic
	obiektowe	C++, Object Pascal, Java



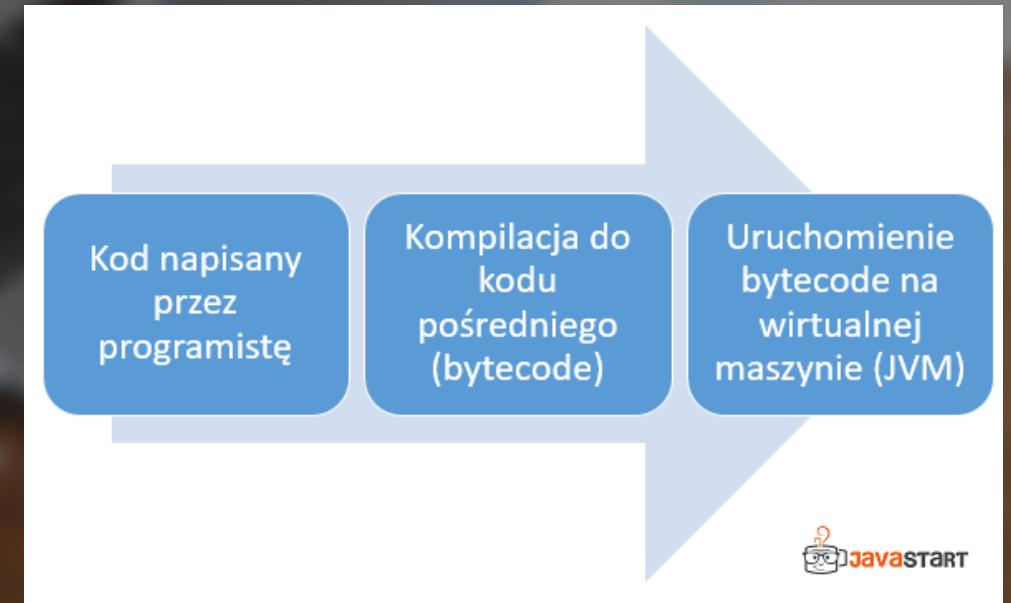
Podstawy JVM



Podstawy JVM

Virtualna maszyna Javy (ang. Java Virtual Machine, w skrócie JVM) - maszyna wirtualna oraz środowisko zdolne do wykonywania kodu bajtowego Javy.

Jest to ciąg instrukcji rozumianych bezpośrednio przez procesor komputera. Efekt kompilacji zapisywany jest w pliku. W momencie uruchomienia danego programu system operacyjny wysyła zawartość takiego pliku bezpośrednio do procesora. W ten sposób kod napisany przez programistę, oraz przetworzony przez kompilator jest uruchamiany i można zobaczyć jego efekty.



Java Runtime Environment



JRE, czyli Java Runtime Environment (środowisko uruchomieniowe Java). Programy napisane w Java, nie będą działały bezpośrednio na komputerze. Do ich uruchomienia potrzebna jest maszyna wirtualna. Żeby móc uruchomić aplikację w Java trzeba posiadać maszynę wirtualną, czyli zainstalowane JRE na swoim komputerze.

W skład JRE wchodzą wszystkie komponenty potrzebne do uruchomienia aplikacji Java.

JRE instalowane jest na komputerach użytkowników końcowych aplikacji oraz na serwerach udostępniających aplikację w formie strony WWW w internecie.

Java Development Kit



JDK, czyli Java Development Kit - to takie “JRE” na sterydach, gdyż oprócz wszystkiego tego, co znajduje się w JRE, dostarcza również narzędzia potrzebne do stworzenia oprogramowania takie, chociażby jak kompilator oraz inne umożliwiające min. analizę działania aplikacji.

Bez JDK nie ma możliwości tworzenia aplikacji Java.

JDK instalowane jest na komputerach programistów piszących oprogramowanie w Java.



Zarządzanie pamięcią

Zadaniem JVM jest między innymi zarządzanie pamięcią.

Oznacza to, że miejsce na obiekty, które są tworzone jest rezerwowane (i zwalniane jeśli obiekt nie jest już wykorzystywany) w tej pamięci.

Pamięć ta nazywana jest stertą (ang. *heap space*).

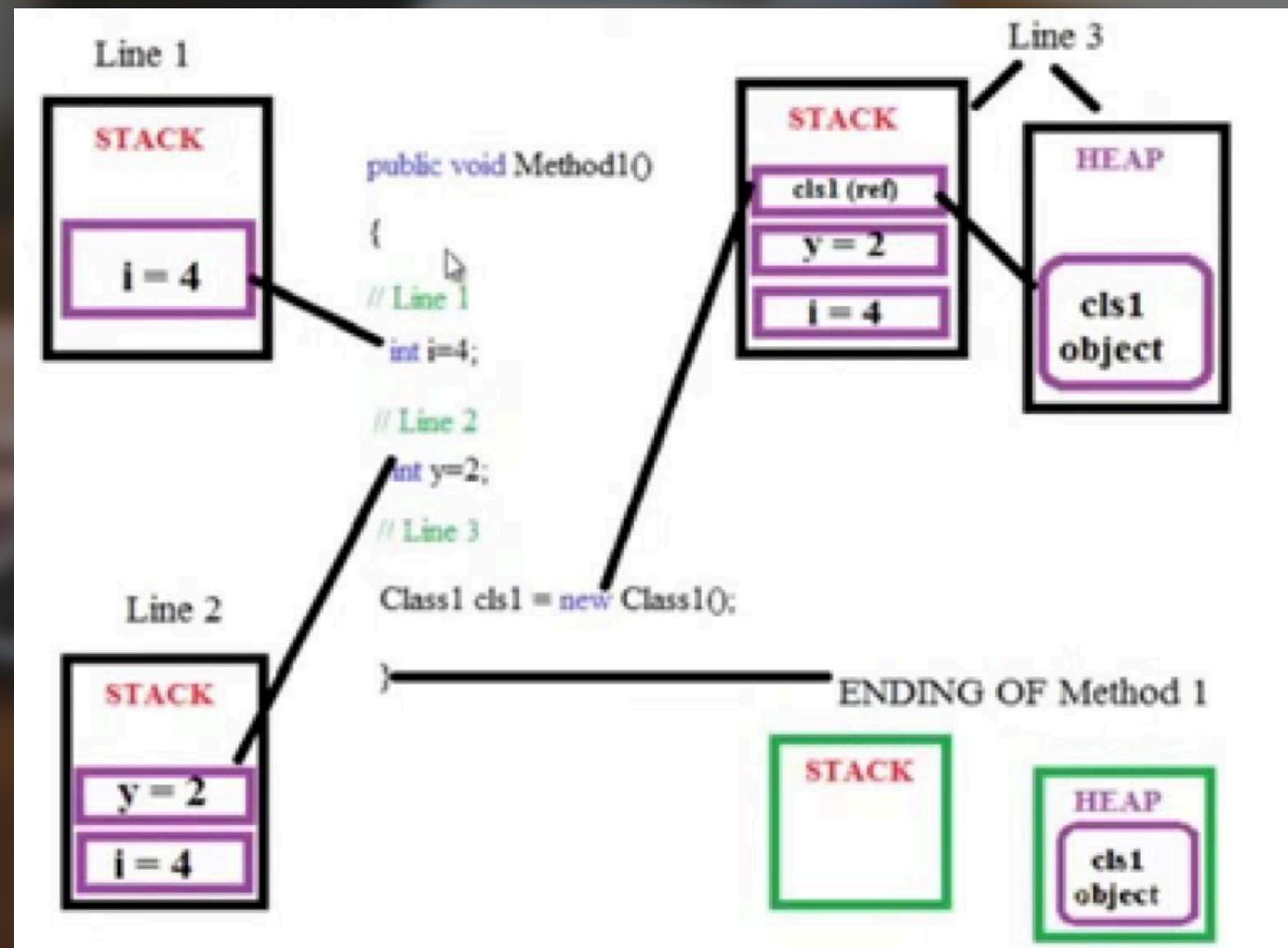
Maszyna wirtualna Javy startuje zapewniając sobie miejsce na stertę w pamięci RAM (alokuje tę pamięć).

Na przykład jeśli Twój komputer ma 4GB pamięci RAM maszyna wirtualna podczas startu może przydzielić sobie 512MB. Obszar ten potencjalnie może się powiększyć w zależności od ustawień maszyny wirtualnej i dostępnej pamięci.



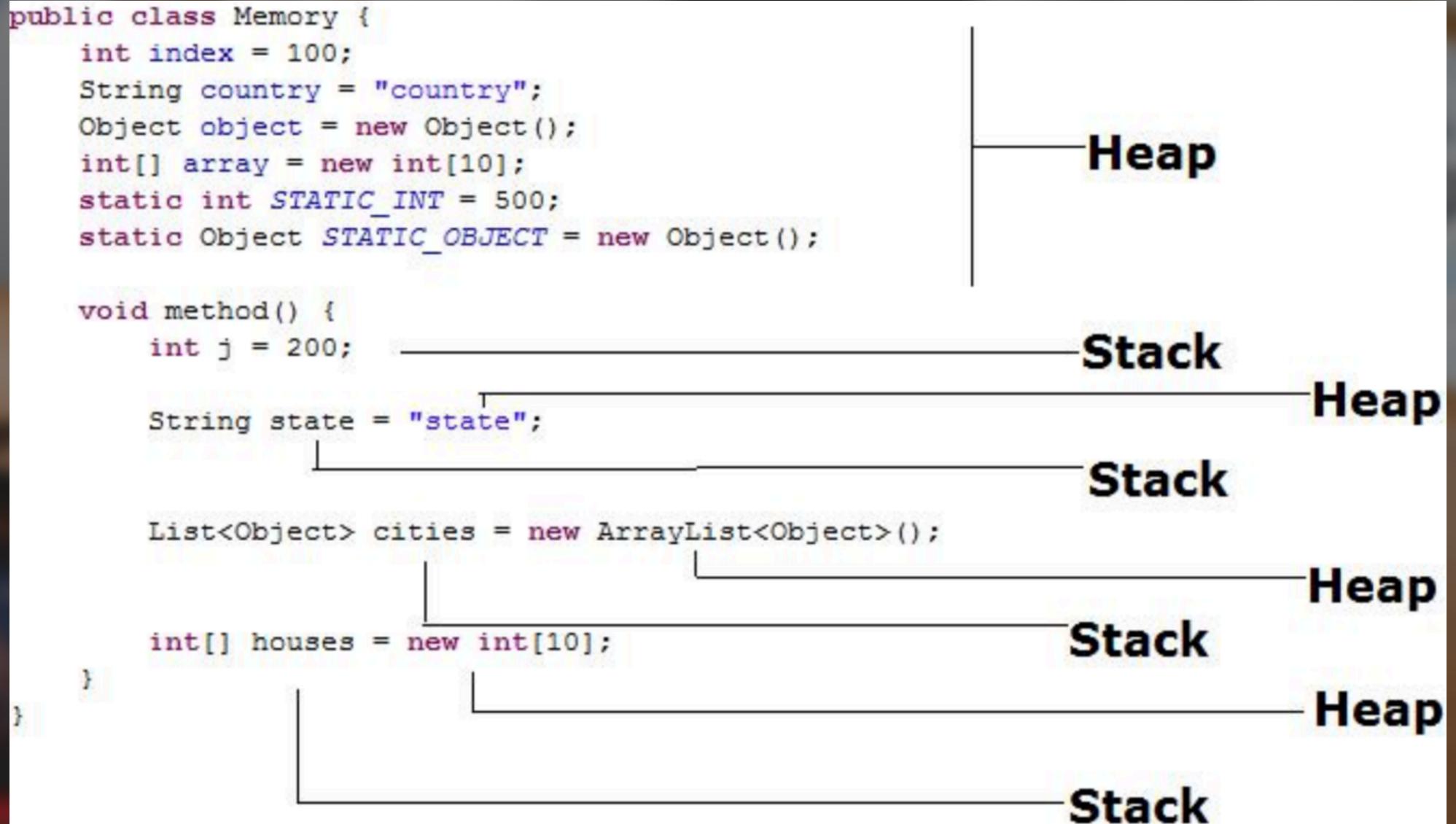
Stos i sterta

1. Na stercie alokowane są wszystkie obiekty jakie tworzymy oraz zmienne klasy (w tym typy proste).
2. dostęp do sterty odbywa się poprzez wskazanie komórki pamięci (przez referencje).
3. Stertę współdzielą ze sobą wszystkie wątki programu.
4. Na stosie odkładane są zmienne lokalne (typy proste jak i referencyjne)
5. Każdy z wątków ma swój własny oddzielny stos.
6. Stos to kolejka LIFO.





Stos i sterta



Zwiększenie pamięci



Help PRO Czw. 20:36 Łukasz Bednarski

Search |

Find Action... ⌘A

Android Studio Help
IntelliJ IDEA Help
Getting Started
Licenses
Keymap Reference
Demos and Screencasts
Tip of the Day
What's New in IntelliJ IDEA

Productivity Guide

Support Center
Submit Feedback
Show Log in Finder
Settings Summary
Compress Logs and Show in Finder
Edit Custom Properties...
Edit Custom VM Options...
Debug Log Settings...
Data Sharing Options...

idea.vmoptions

```
1 # custom IntelliJ IDEA VM options
2
3 -Xms128m
4 -Xmx2048m
5 -XX:ReservedCodeCacheSize=240m
6 -XX:+UseCompressedOops
7 -Dfile.encoding=UTF-8
8 -XX:+UseConcMarkSweepGC
9 -XX:SoftRefLRUPolicyMSPerMB=50
10 -ea
11 -Dsun.io.useCanonCaches=false
12 -Djava.net.preferIPv4Stack=true
13 -XX:+HeapDumpOnOutOfMemoryError
14 -XX:-OmitStackTraceInFastThrow
15 -XX:MaxJavaStackTraceDepth=-1
16 -Xverify:none
17
18 -XX:ErrorFile=$USER_HOME/java_error_in_idea_%p.log
19 -XX:HeapDumpPath=$USER_HOME/java_error_in_idea.hprof
20 -agentlib:yjpagent=probe_disable=*,disablealloc,disabletracing,onlylocal
21 -Didea.no.platform.update=true
22
```



Obiekt a referencja zmiennej

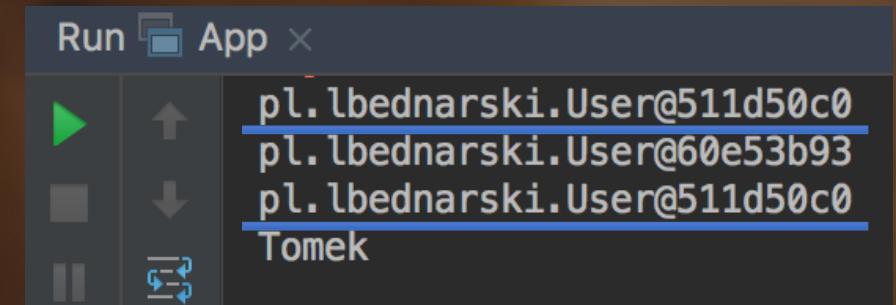
Każdy obiekt w Javie zajmuje jakiś obszar na stercie.

Zmienne, które wskazują na obiekty na stercie zawierają referencje.

Referencje możemy przedstawić jako „sznurki” łączące zmienną z właściwym obiektem na stercie (obszarem pamięci na stercie gdzie znajduje się obiekt).

Referencji (sznurków) do obiektu może być wiele.

```
public static void main( String[] args ) throws IOException {  
    User user = new User("Jan", "Nowak"); // Nowa zmienna  
    User user2 = new User("Jan", "Nowak"); // Nowa zmienna  
    User user3 = user; // user3 zawiera referencję do user  
  
    System.out.println(user);  
    System.out.println(user2);  
    System.out.println(user3);  
  
    user.setFirstName("Tomek");  
    System.out.println(user3.getFirstName());  
}
```





Porównywanie obiektów String w praktyce

Co będzie efektem działania kodu:

```
public static void main(String[] args) {  
    String jan1 = new String(original: "Jan");  
    String jan2 = new String(original: "Jan");  
    String jan3 = "Jan";  
    String jan4 = "Jan";  
  
    System.out.println(jan1 == jan2);  
    System.out.println(jan2 == jan3);  
    System.out.println(jan3 == jan4);  
  
    System.out.println();  
  
    System.out.println(jan1.equals(jan2));  
    System.out.println(jan2.equals(jan3));  
    System.out.println(jan3.equals(jan4));  
}
```

▶	⬇	false
■	⬇	false
	⬇	true
↔	⬇	
⬇	⬇	true
.....	⬇	true
⬇	⬇	true



"string" a new String("string")

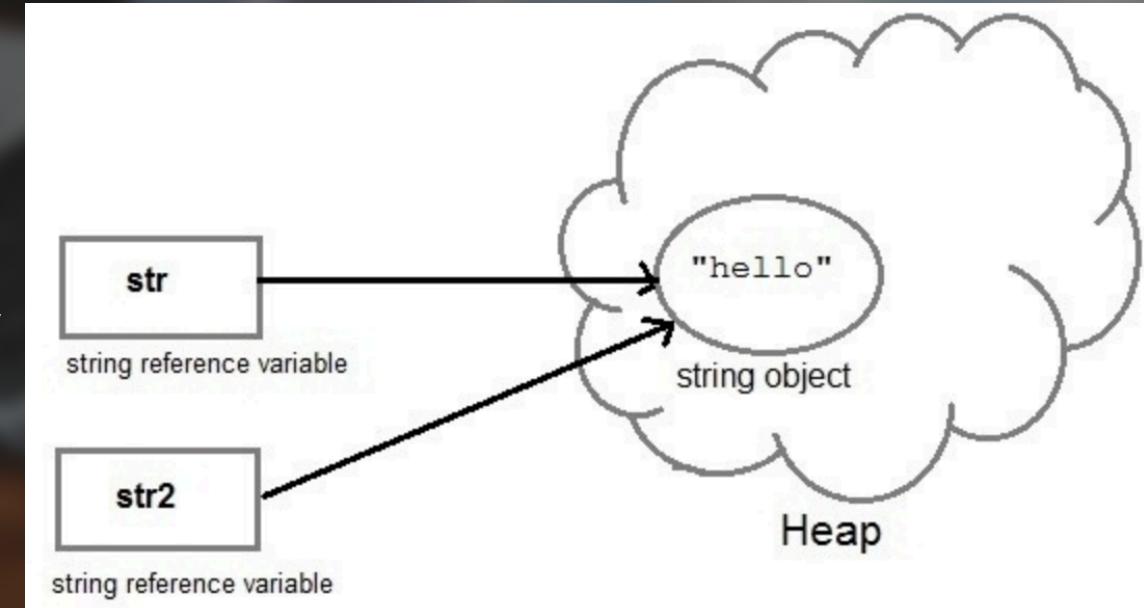
Czym różni się poniższa implementacja tworzenia obiektu typu string:

```
public static void main( String[] args ) {  
    String x1 = new String("x");  
    String x2 = new String("x");  
    String x3 = "x";  
    String x4 = "x";  
}
```

Dwie pierwsze linijki to dwie zmienne, dwie referencje i dwa różne obiekty (które mają w taką samą zawartość).

Dwie ostatnie linijki to dwie zmienne, dwie referencje i jeden obiekt. Obie referencje pokazują na obiekt utworzony w 3 linijce.

Innymi słowy konstruktor tworzy kopię przekazanego literału. Literał może czasami ograniczyć ilość użytego miejsca na stercie.





Podstawy Garbage Collection



Garbage Collector

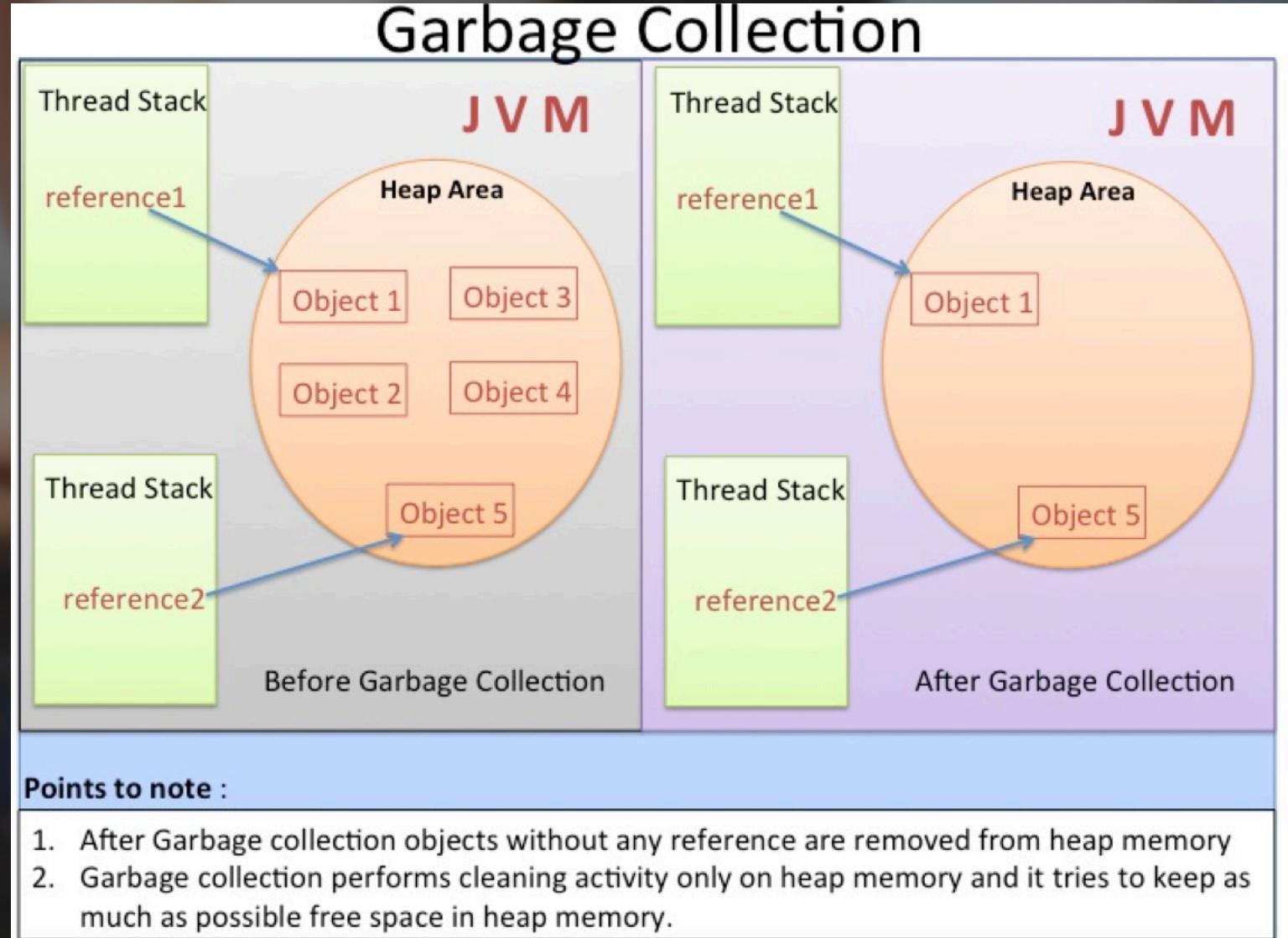
Garbage collector, jest komponentem wirtualnej maszyny Javy odpowiedzialnym za czyszczenie sterty. Jeśli na stercie znajdą się obiekty, które już nie są używane zostają one usunięte aby zwolnić miejsce dla nowych obiektów.

Gdyby nie było tego mechanizmu pamięć nie byłaby zwalniana i po pewnym czasie program nie mógłby działać, nie miałby miejsca na alokację pamięci dla nowo tworzonych obiektów. GC używa różnych zaawansowanych algorytmów, które pozwalają zdecydować czy dany obiekt jest aktualnie wykorzystywany przez działający program.

GC działa w tle, w większości przypadków nawet nie będziesz wiedział kiedy. Wirtualna maszyna sama decyduje kiedy powinna go uruchomić. Istnieją sposoby aby zasugerować uruchomienie GC ale w rzeczywistości nie masz na to wpływu.



Garbage Collector





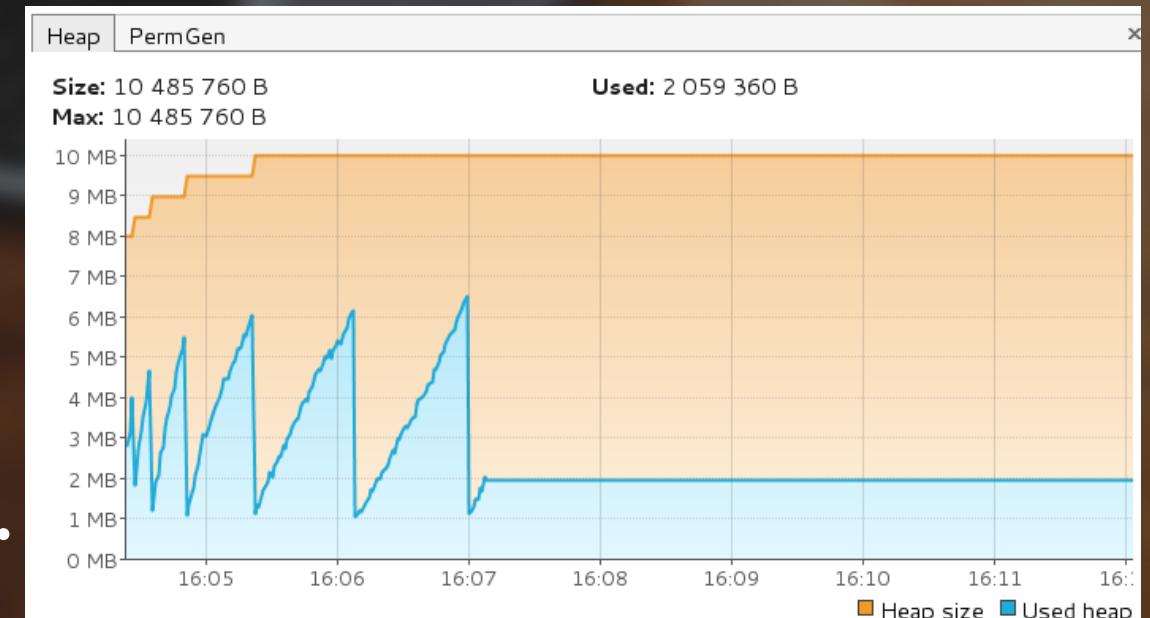
Garbage Collector

Co się wydarzy po uruchomieniu poniższego kodu jeśli JVM została uruchomiona z parametrem `-Xmx10M`.

```
public static void main(String[] args) {  
    System.out.println("HELLO MEMORY LEAK");  
  
    while (true){  
        for (int i = 0; i < 100000; i++) {  
            User user = new User("Jan", "Nowak");  
        }  
    }  
}
```

Nie wydarzy się nic :)

GC usunie nieużywane obiekty
kiedy pamięć będzie się kończyć.



Garbage Collector

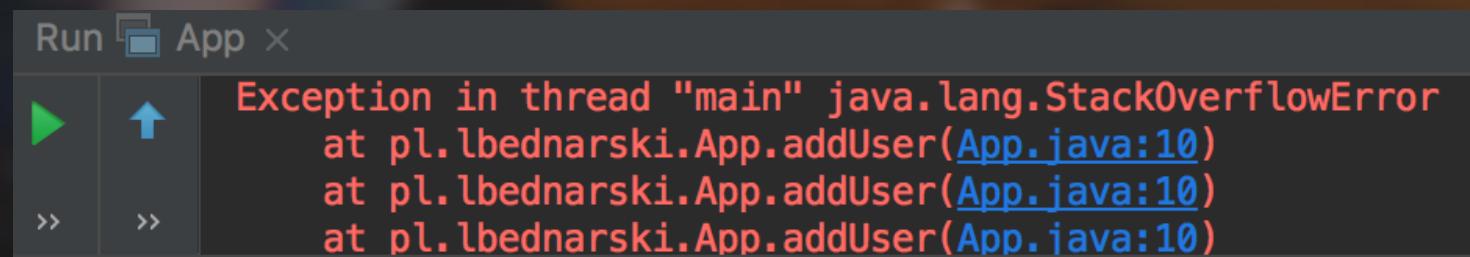


Jak w takim razie przepełnić pamięć tak aby rzucony został wyjątek StackOverflowError?

Wystarczy wywołać metodę rekurencyjnie i nigdy jej nie przerywać.

```
public static void main( String[] args ) {  
    addUser(new User("Jan", "Nowak"));  
}
```

```
static private void addUser(User user){  
    addUser(user);  
}
```





Algorytmy Garbage Collector

1. Reference Counting

Najprostsza metoda odśmiecania. Alokowany obiekty posiada dodatkowe pole, które przechowuje liczbę odwołań. Na starcie pole to ustawiane jest na 1. Podczas tworzenia obiektów lub ich usuwania, licznik jest zwiększany, lub zmniejszany. Wyzerowanie licznika oznacza, iż przydzielona pamięć może zostać zwolniona.

2. Concurrent GC

Inaczej mark-and-sweep, podstawowy algorytm GC. Każdy obiekt otrzymuje tzw. markbit mówiący czy obiekt jest potrzebny czy nie(0, 1). Pamięć nie jest odzyskiwana gdy stwierdzi się, że obiekt jest już niepotrzebny, lecz gdy zostanie przekroczena pamięć pewnego progu. **Program jest wtedy zatrzymywany i uruchamiana jest faza odśmiecania.** Wyróżniamy dwie fazy:

1. Mark - GC oznacza używane obiekty(markbit = 1)
2. Sweep - GC usuwa obiekty, których markbit wynosi 0 oraz rejestruje nowe, czyli te które nie są już potrzebne.



Narzędzia CLI



Aplikacje JVM

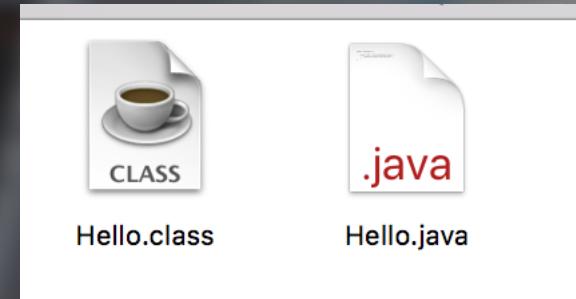
- 1.JDK dostarcza szereg aplikacji do budowy aplikacji z wykorzystaniem języka JAVA.
- 2.Dodatkowo też udostępnia aplikacje do analizy działania wirtualnej maszyny Javy.
- 3.Narzędzia te pozwalają analizować pracę aplikacji online jak i analizować zapisane wcześniej dane.
- 4.Analizować można informację na temat pamięci, wątków tworzonych obiektów itp.

Javac i Java



1. Aby uruchomić aplikację napisaną w Javie, trzeba ją najpierw skompilować.
2. Służy do tego aplikacja javac.
3. Aby uruchomić aplikację należy wykorzystać aplikację java.

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```



```
Last login: Sat Feb 24 18:10:55 on ttys001  
[MacBook-Pro-ukasz:~ lukasz$ cd Desktop/app/  
[MacBook-Pro-ukasz:app lukasz$ javac Hello.java  
[MacBook-Pro-ukasz:app lukasz$ java Hello  
Hello World  
MacBook-Pro-ukasz:app lukasz$
```

Zmiana katalogu
Skompilowanie klasy
Uruchomienie aplikacji
Efekt działania aplikacji

```
[MacBook-Pro-ukasz:app lukasz$ javac *.java  
Message.java:3: error: <identifier> expected  
    private String;  
          ^  
1 error  
MacBook-Pro-ukasz:app lukasz$
```



Gdzie mieszkają problemy ?

Rzeczywista ilość zasobów

Wydajność dysków

Wydajność sieci

Niewydajny model danych

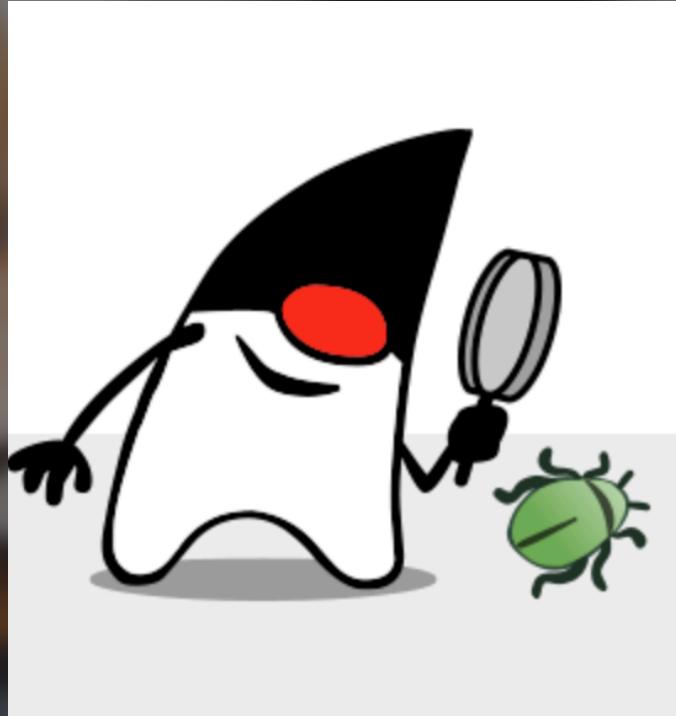
Niewydajne algorytmy

Wycieki pamięci

Cache

Wydajność procesora i pamięci

Pule wątków



Pule połączeń

Zarządzanie pamięcią

I/O

Komunikacja z systemami zewnętrznymi

Inne aplikacje

Konfiguracja pamięci

Niewłaściwe kolekcje



Wnioski

1. Szukanie problemów w kodzie jest jak szukanie igły w stogu siana.
2. Warto sięgnąć po narzędzia które poprowadzą nas dokładnie do źródła problemu.
3. Czasami problemy nie wynikają z kodu a z konfiguracji całego otoczenia aplikacji.
4. W katalogu %JAVA_HOME%/bin możemy znaleźć listę przydatnych aplikacji (JAVA_HOME to katalog gdzie zainstalowano java).



1. Narzędzie pozwalające wypisować uruchomione (domyślnie) na lokalnym komputerze maszyny wirtualne.
2. W charakterze parametru można podać dowolny adres komputera (zdalnego), oczywiście pod warunkiem, że mamy do niego dostęp.
3. W dokumentacji aplikacja oznaczona jako niewspierana a więc w nowych wydaniach Javy może się już nie pojawić.

```
app — -bash — 80x21
~/Desktop/app — -bash ... ~/Desktop/app — -bash
MacBook-Pro-ukasz:app lukasz$ jps
1968 RemoteMavenServer
1440
2017 Launcher
1972 Launcher
2276 Jps
MacBook-Pro-ukasz:app lukasz$ jps
1968 RemoteMavenServer
1440
2017 Launcher
2277 Launcher
2278 App
2279 Jps
MacBook-Pro-ukasz:app lukasz$
```

Wywołanie programu JPS

Ponowne wywołanie programu JPS ale w między
czasie uruchomiona została aplikacja w IntelliJ

Aplikacja uruchomiona w IntelliJ



1. Szczegółowe informacje na temat konkretnej wirtualnej maszyny Javy.
2. Udostępnia informację na temat parametrów, flag czy konfiguracji JVM.

```
MacBook-Pro-ukasz:app lukasz$ jps -l
1968 org.jetbrains.idea.maven.server.RemoteMavenServer
1440
2481 sun.tools.jps.Jps
2423 org.jetbrains.jps.cmdline.Launcher
2424 pl.lbednarski.App
MacBook-Pro-ukasz:app lukasz$ jinfo 2424
Attaching to process ID 2424, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.60-b23
Java System Properties:
```

Wywołanie JPS aby pobrać informację o uruchomionych JVM

Uruchomienie analizy podając PID procesu

Wyświetlenie listy parametrów JVM

```
java.runtime.name = Java(TM) SE Runtime Environment
java.vm.version = 25.60-b23
sun.boot.library.path = /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib
gopherProxySet = false
java.vendor.url = http://java.oracle.com/
java.vm.vendor = Oracle Corporation
path.separator = :
file.encoding.pkg = sun.io
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
sun.os.patch.level = unknown
sun.java.launcher = SUN_STANDARD
user.country = PL
```



1. Aplikacja służąca do tworzenia oraz analizy zrzutów pamięci

2. Dzięki niej możemy zobaczyć ile aktualnie pamięci zajmuje nasza instancja JVM, ile miejsca maksymalnie zostało jej przydzielone itp.

```
MacBook-Pro-ukasz:app lukasz$ jmap -heap 2424
Attaching to process ID 2424, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.60-b23

using thread-local object allocation.
Parallel GC with 4 thread(s)
```

```
Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize           = 10485760 (10.0MB)
  NewSize               = 3145728 (3.0MB)
  MaxNewSize             = 3145728 (3.0MB)
  OldSize               = 7340032 (7.0MB)
  NewRatio              = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize       = 17592186044415 MB
  G1HeapRegionSize        = 0 (0.0MB)
```

Uruchomienie jmap wraz z PID

Ustawienia JVM pod kątem pamięci



1. Zwraca информацию na temat o pakiecie oraz publicznych i chronionych polach/metodach.
2. Narzędzie o raczej niedużej użyteczności
3. Najczęściej wykorzystywany do przejrzenia wygenerowanego kodu binarnego (parametr -c)

```
1  public class Hello {  
2  
3      public static void main(String[] args) {  
4          int a = 5;  
5          int b = 6;  
6          int c = add(a,b);  
7  
8          System.out.println(c);  
9      }  
10  
11     private static int add(int a, int b) {  
12         return a+b;  
13     }  
14 }
```

```
[MacBook-Pro-ukasz:app lukasz$ javac Hello.java  
[MacBook-Pro-ukasz:app lukasz$ javap Hello  
Compiled from "Hello.java"  
public class Hello {  
    public Hello();  
    public static void main(java.lang.String[]);  
}
```

Wywołanie aplikacji javap

Informacje na temat tego skąd pochodzi dana klasa i o jej publicznych metodach.



1. Mechanizm debuggowania dla Javy. Pozwala uruchomić aplikację i zatrzymać w dowolnym momencie.
2. Narzędzie trudne do wykorzystania z poziomu konsoli (polecane raczej dla geeków ;)

```
1 public class Hello {  
2  
3     public static void main(String[] args) {  
4         int a = 5;  
5         int b = 6;  
6         int c = add(a,b);  
7  
8         System.out.println(c);  
9     }  
10  
11    private static int add(int a, int b) {  
12        return a+b;  
13    }  
14 }
```

The terminal window shows the following sequence of commands and responses:

- javac Hello.java
- jdb Hello
- Initializing jdb ...
- > stop in Hello.add
- Deferring breakpoint Hello.add.
- It will be set after the class is loaded.
- > run Hello
- run Hello
- Set uncaught java.lang.Throwable
- Set deferred uncaught java.lang.Throwable
- >
- VM Started: Set deferred breakpoint Hello.add
- Breakpoint hit: "thread=main", Hello.add(), line=12 bci=0
- 12 return a+b;
- main[1] cont
- > 11
- The application exited

Uruchomienie trybu debuggowania
Dodanie breakpointa w metodzie add klasy Hello
Uruchomienie aplikacji

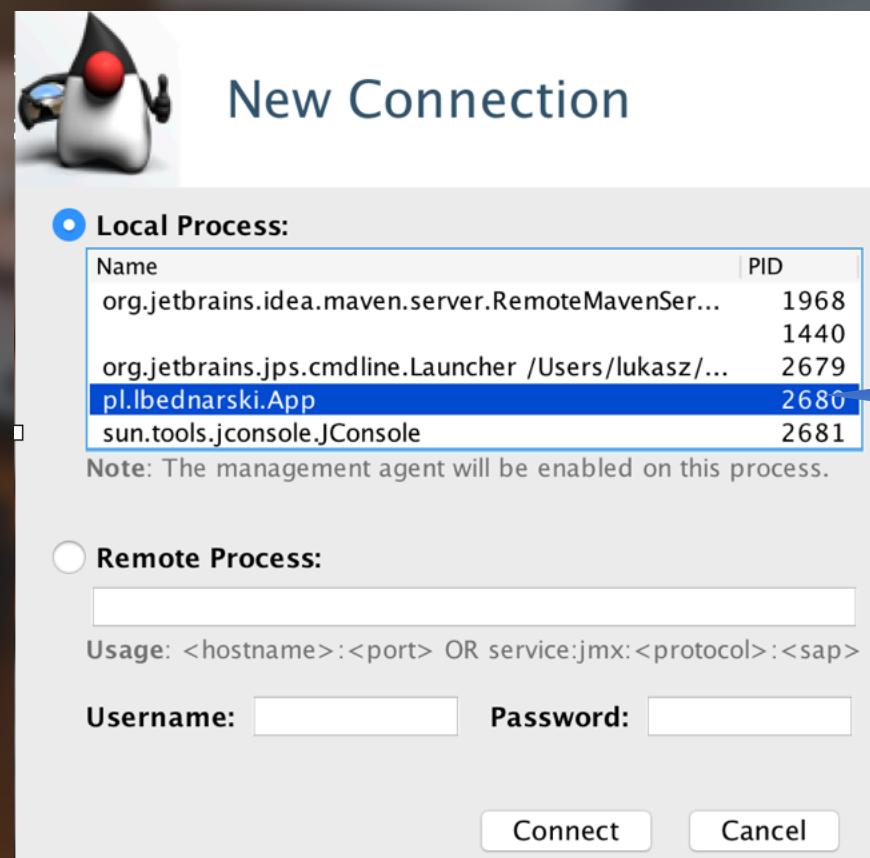
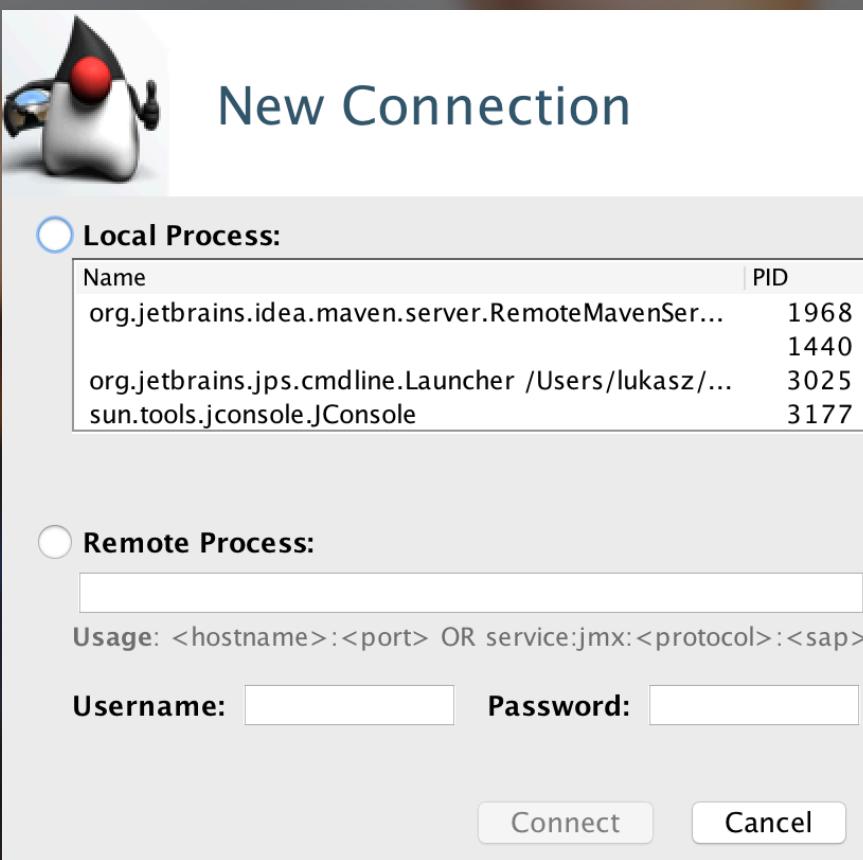
Zatrzymanie aplikacji w 12 linii aplikacji

Kontynuacja pracy aplikacji

jConsole



1. Pozwala w graficzny sposób analizować aplikację uruchomioną na konkretnej instancji JVM.
2. Zdecydowanie bardziej czytelne podejście niż konsolowe narzędzia.
3. Możliwe jest także podłączenie się do zdalnej instancji JVM na serwerze



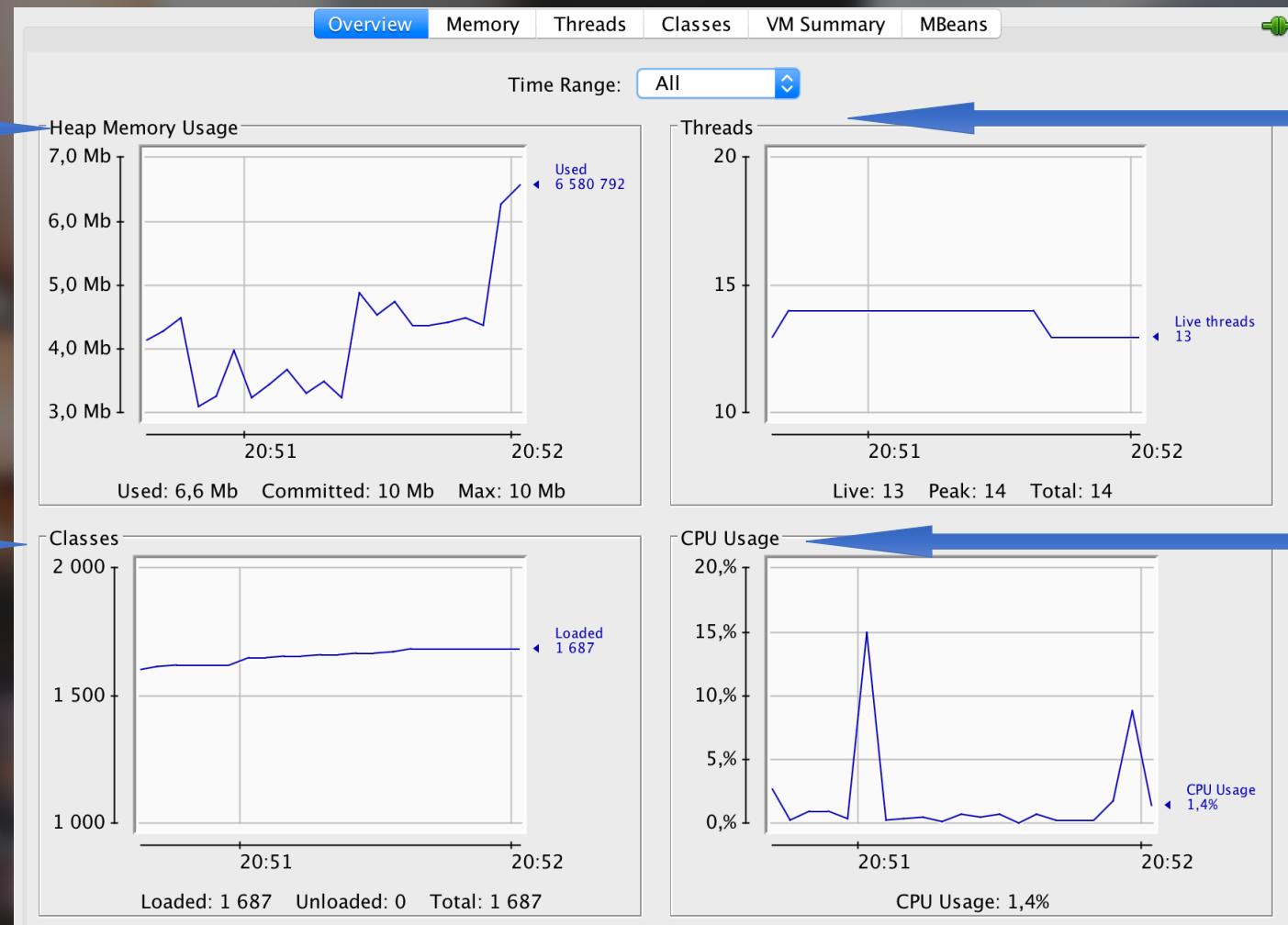
Wybieramy aplikację którą chcemy analizować.



jConsole

Informacje o wykorzystaniu dostępnej pamięci przydzielonej JVM

Ilość klas wczytanych przez mechanizm ClassLoader



Informacje o aktywnych wątkach w aplikacji

Informacje o zużyciu zasobów procesora

jConsole



1. Zawiera informację na temat pamięci przydzielonej wirtualnej maszynie.
2. Na screenie bardzo dobrze widać że kiedy pamięć zaczyna się kończyć (10Mb) to GC wykonuje swoją pracę





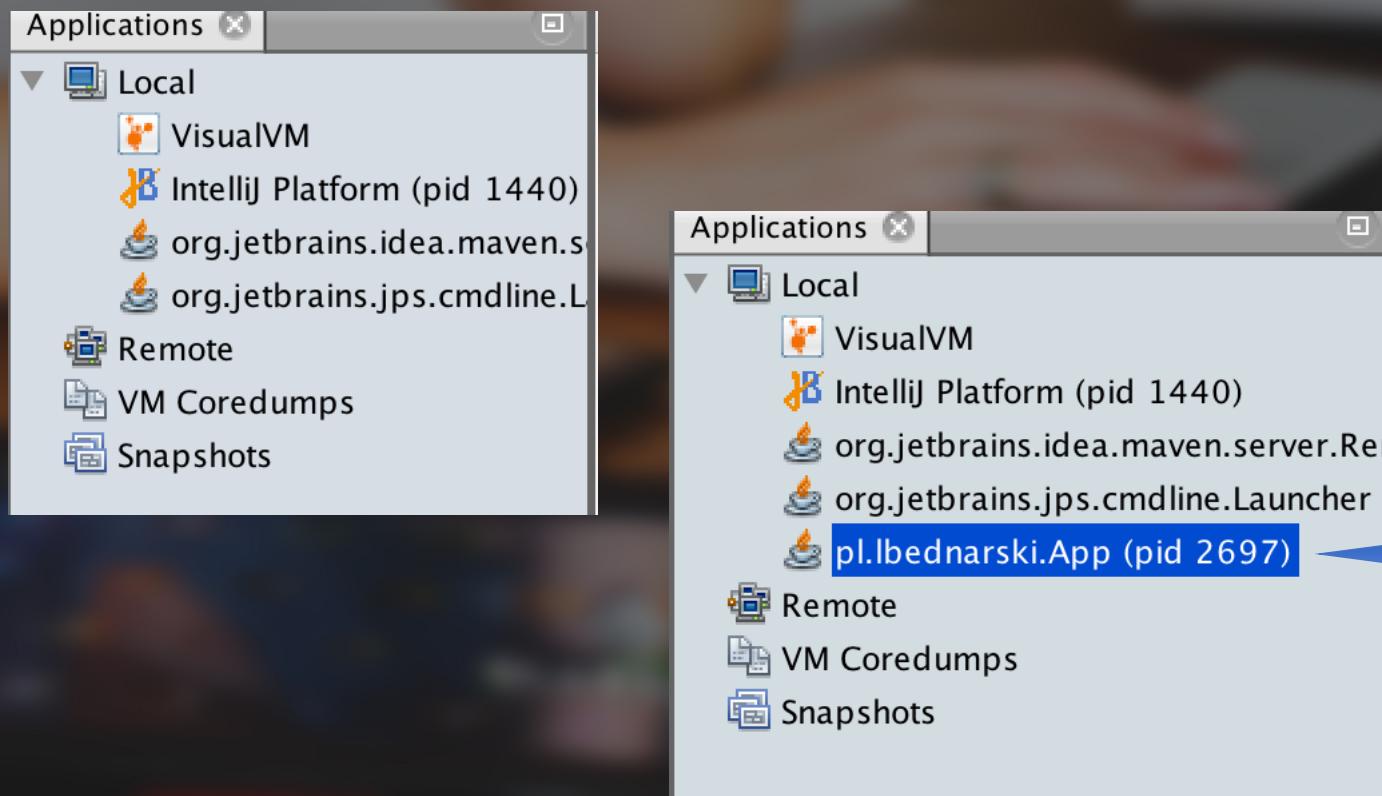
VM Summary zawiera informację i konfigurację JVM.

Overview	Memory	Threads	Classes	VM Summary	MBeans
VM Summary sobota, 24 lutego 2018 21:00:28 CET					
Connection name: pid: 2680 pl.lbednarski.App Virtual Machine: Java HotSpot(TM) 64-Bit Server VM version 25.60-b23 Vendor: Oracle Corporation Name: 2680@MacBook-Pro-ukasz.local					
				Uptime: 10 minutes Process CPU time: 1 minute JIT compiler: HotSpot 64-Bit Tiered Comp Total compile time: 7,336 seconds	
Live threads: 11 Peak: 14 Daemon threads: 10 Total threads started: 19				Current classes loaded: 1 660 Total classes loaded: 1 998 Total classes unloaded: 338	
Current heap size: 8 834 kbytes Maximum heap size: 9 728 kbytes				Committed memory: 9 728 kbytes Pending finalization: 0 objects	
Garbage collector: Name = 'PS MarkSweep', Collections = 123, Total time spent = 5,037 seconds Garbage collector: Name = 'PS Scavenge', Collections = 1 663, Total time spent = 3,145 seconds					
Operating System: Mac OS X 10.13.3 Architecture: x86_64 Number of processors: 4 Committed virtual memory: 5 761 960 kbytes				Total physical memory: 8 388 608 kbytes Free physical memory: 83 916 kbytes Total swap space: 2 097 152 kbytes Free swap space: 729 344 kbytes	

jVisualVM



1. Pozwala w graficzny sposób analizować aplikację uruchomioną na konkretnej instancji JVM.
2. Możliwe jest także podłączenie się do zdalnej instancji JVM na serwerze.
3. Nowsze i bardziej rozbudowane narzędzie niż jConsole.



Wybieramy aplikację którą chcemy analizować.



Pierwsza zakładka to podsumowanie konfiguracji JVM

pl.lbednarski.App (pid 2697)

Overview Monitor Threads Sampler Profiler

pl.lbednarski.App (pid 2697)

Overview Saved data Details

PID: 2697
Host: localhost
Main class: pl.lbednarski.App
Arguments: <none>

JVM: Java HotSpot(TM) 64-Bit Server VM (25.60-b23, mixed mode)
Java: version 1.8.0_60, vendor Oracle Corporation
Java Home: /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre
JVM Flags: <none>

Heap dump on OOME: disabled

Saved data **JVM arguments** System properties

Thread Dumps: 0 **Heap Dumps:** 0 **Profiler Snapshots:** 0

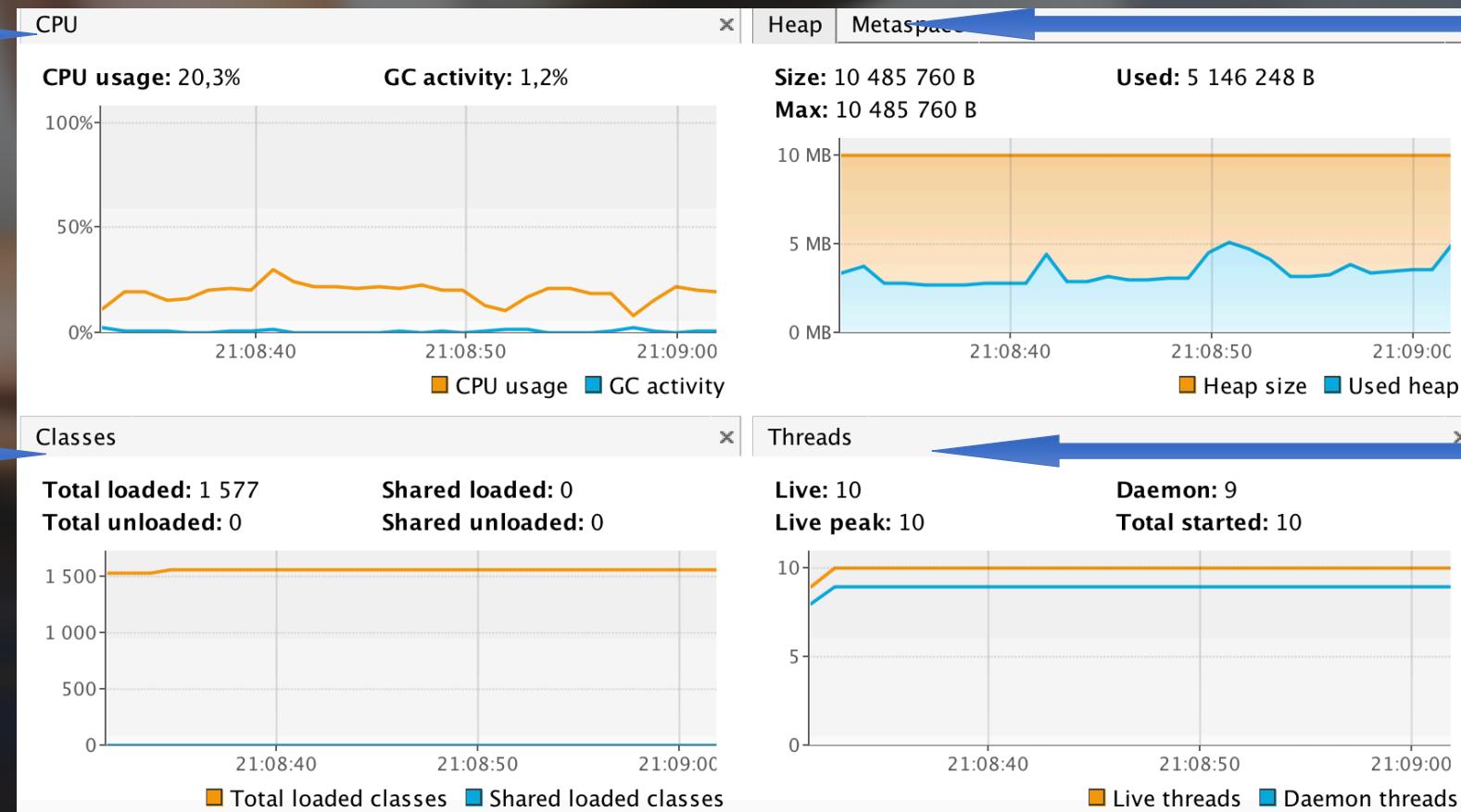
```
-agentlib:jdwp=transport=dt_socket,address=127.0.0.1:55042,suspend=y,server=n
-Xmx10M
-XX:MaxPermSize=1024M
-javaagent:/Users/lukasz/Library/Caches/IntelliJIdea2018.1/captureAgent/debugger-agent
-Dfile.encoding=UTF-8
```

Informacje na temat środowiska, wersji Java, konfiguracji itd.



Statystyki na temat zużycia zasobów przez JVM

Informacje o
zużyciu zasobów
procesora



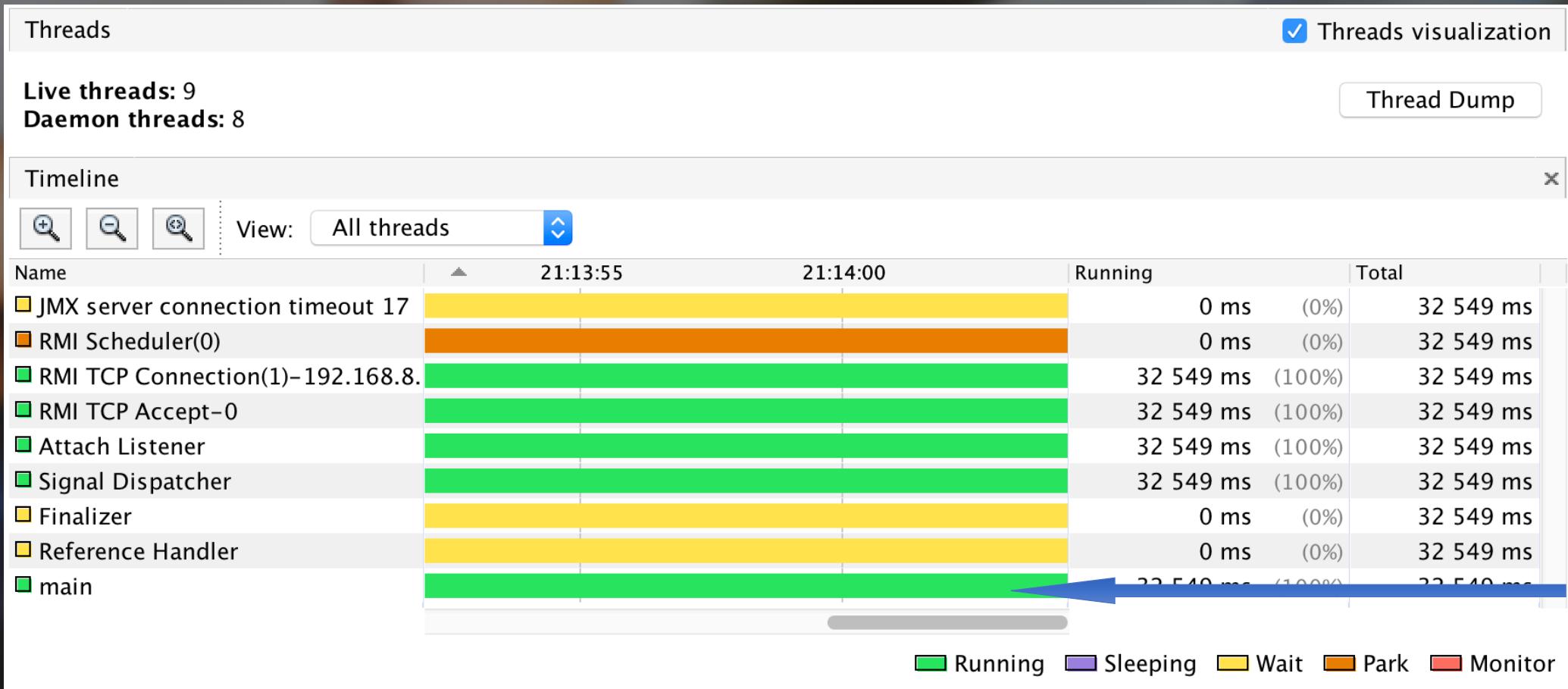
Informacje o
wykorzystaniu
dostępnej pamięci
przydzielonej JVM

Ilość klas
wczytanych przez
mechanizm
ClassLoader

Informacje o
aktywnych
wątkach w
aplikacji



1. Informacje o aktywnych wątkach uruchomionej aplikacji.
2. Pozwala dość szybko znaleźć zakleszczające się wątki.



Główny wątek uruchomionej aplikacji



Profiler i Sampler pozwalają analizować ile czasu procesora zajmuje wykonanie danej metody.

Sampler

Sample: CPU Memory Stop

Status: CPU sampling in progress

CPU samples Thread CPU Time

Snapshot Thread Dump

Hot Spots - Method	Self Time ...	Self Time	Self Ti...	Total Time	Total Time ...	Samples
java.lang.Object.wait[n]	832 250 ... (30,7%)	0,000 ms	832 250 ms	0,000 ms	2 680	▲
sun.misc.Unsafe.park[n]	445 088 ... (16,4%)	473 ms	445 088 ms	473 ms	1 685	☰
java.net.SocketInputStream	441 255 ... (16,3%)	441 25...	441 255 ms	441 255 ms	226	
sun.management.ThreadIn	421 077 ... (15,5%)	421 07...	421 077 ms	421 077 ms	226	
java.lang.Thread.sleep[408 372 ... (15,1%)	0,000 ms	408 372 ms	0,000 ms	82	
java.net.DualStackPlainSo	124 769 ... (4,6%)	0,000 ms	124 769 ms	0,000 ms	3	
sun.java2d.d3d.D3DRendi	30 535 ms (1,1%)	30 535 ...	30 535 ms	30 535 ms	1 278	
java.lang.Object.notifyA	1 326 ms (0%)	1 326 ms	1 326 ms	1 326 ms	67	
java.security.AccessContr	1 236 ms (0%)	1 236 ms	1 236 ms	1 236 ms	64	
java.lang.Thread.yield[r	831 ms (0%)	831 ms	831 ms	831 ms	42	
sun.awt.windows.WGlobal	755 ms (0%)	755 ms	755 ms	755 ms	39	▼

Method Name Filter (Contains)



Profiler i Sampler pozwalają także analizować liczbę obiektów i ilość zajmowanego przez nie miejsca.

pl.lbednarski.App (pid 2986)

Sampler

Sample: CPU Memory Stop

Status: memory sampling in progress

Heap histogram Per thread allocations

Classes: 739 Instances: 75 665 Bytes: 6 936 640

Class Name	Bytes [%]	Bytes	Instances
int[]		3 429 064 (49.4%)	1 558 (2.0%)
char[]		1 189 912 (17.1%)	18 860 (24.9%)
java.lang.String		316 704 (4.5%)	13 196 (17.4%)
java.nio.HeapCharBuffer		268 656 (3.8%)	5 597 (7.3%)
java.lang.Class		198 600 (2.8%)	1 756 (2.3%)
byte[]		189 968 (2.7%)	525 (0.6%)
java.lang.Object[]		162 600 (2.3%)	3 207 (4.2%)
java.lang.reflect.Method		120 472 (1.7%)	1 369 (1.8%)
java.util.HashMap		74 928 (1.0%)	1 561 (2.0%)
pl.lbednarski.User		67 200 (0.9%)	2 800 (3.7%)
java.lang.StringBuilder		67 200 (0.9%)	2 800 (3.7%)
java.lang.reflect.Field		47 448 (0.6%)	659 (0.8%)
java.util.Hashtable\$Entry[]		45 784 (0.6%)	690 (0.9%)
java.util.Vector		43 168 (0.6%)	1 349 (1.7%)
java.util.List		27 056 (0.4%)	1 600 (2.1%)

Class Name Filter (Contains)

Ilość zajmowanego miejsca przez obiekty poszczególnych klas oraz liczba instancji tych klas

jVisualVM



pl.lbednarski.App (pid 2986)

Sampler Settings

Sample: CPU Memory Stop

Status: memory sampling in progress

Heap histogram Per thread allocations

Deltas Snapshot Perform GC Heap Dump

Classes: 699 Instances: 87 553 Bytes: 6 220 056

Class Name	Bytes [%]	Bytes	Instances
pl.lbednarski.User	1	101 976 (1.6%)	4 249 (4.8%)

 user

Możemy także
znać obiekty
konkretniej klasy



Debugger



Podstawy

1. Debugger pozwala zatrzymać działanie aplikacji w dowolnym momencie.
2. Możemy dzięki temu przeanalizować wartości zmiennych itp

LPM - Dodanie brekpointa czyli punktu w którym aplikacja ma się zatrzymać zanim linijka kodu się wykona



```
5 ► ⌂ public static void main(String[] args) {  
6   while (true){  
7     for (int i = 0; i < 10000000; i++) {  
8       User user = new User("Jan", "Nowak");  
9       System.out.println(user);  
10    }  
11  }  
12 }  
13 }  
14 }
```

```
▶ ⌂ public static void main(String[] args) { args: {}  
  while (true){  
    for (int i = 0; i < 10000000; i++) { i: 192414  
      User user = new User("Jan", "Nowak"); user: User@466  
      System.out.println(user); user: User@466  
    }  
  }  
}
```

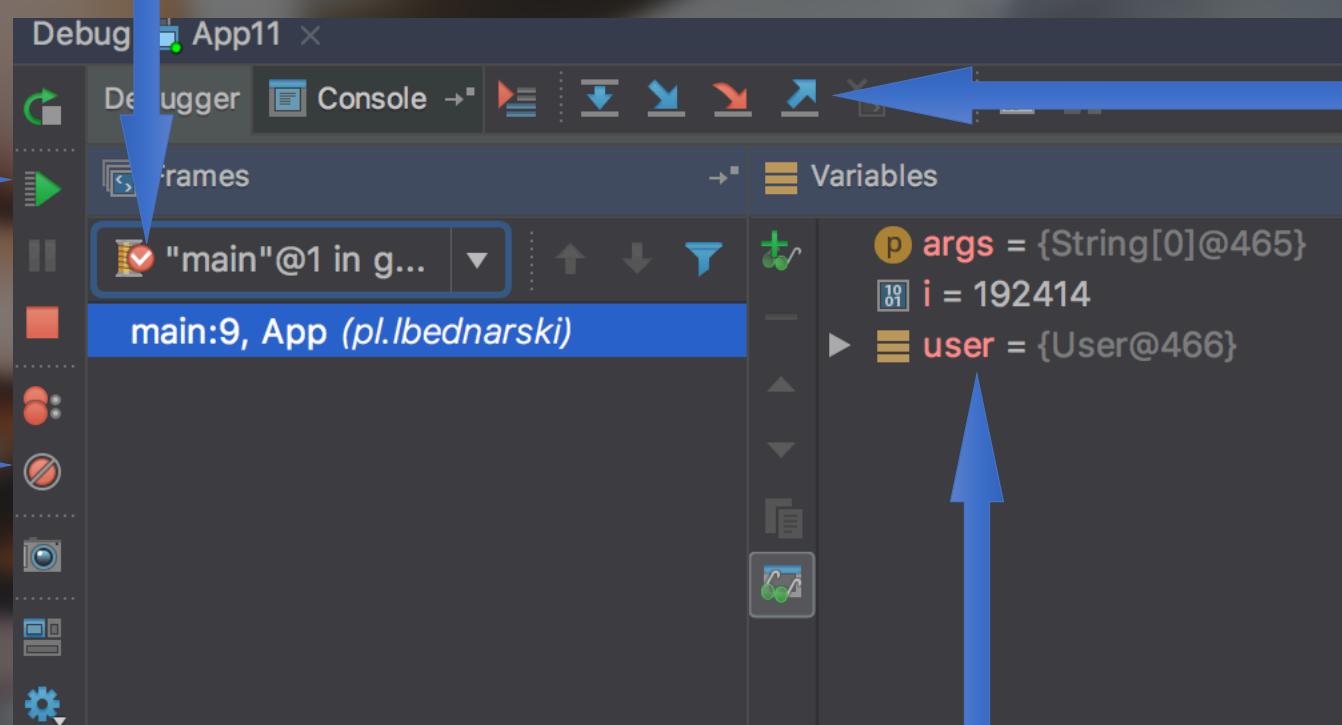


Podstawy

Aktualny wątek.

Kontynuacja działania aplikacji. Debugger zatrzyma aplikację po ponownym trafieniu na breakpointa.

„Wyciszenie” breakpointów. Kliknięcie spowoduje że aplikacja nie będzie zatrzymywała się podczas pracy.



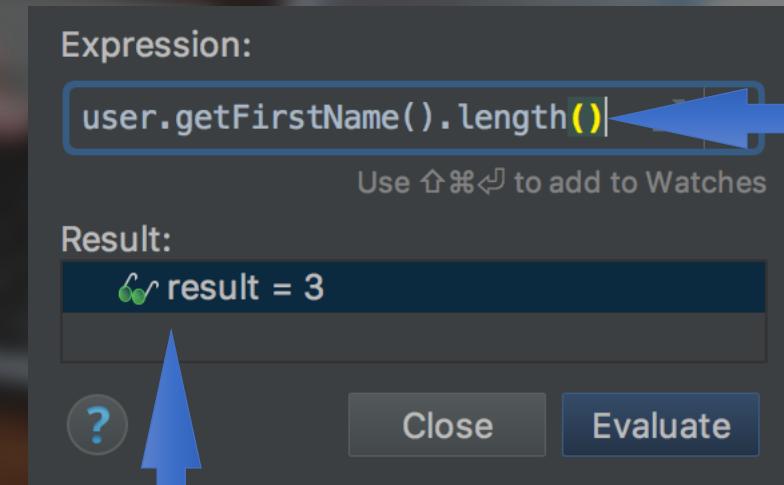
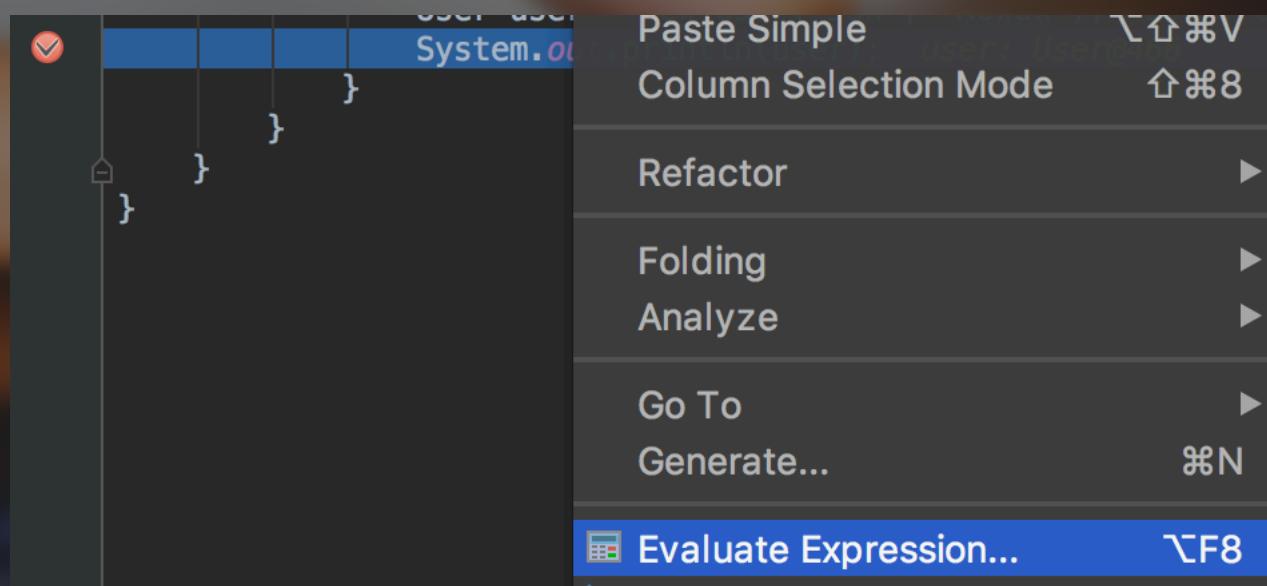
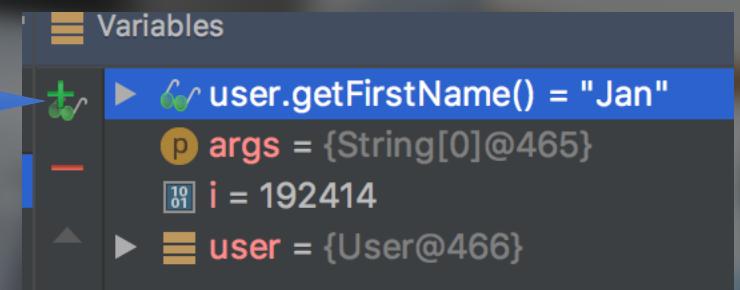
Wartości zmiennych

Wykonywane kroki.
Można wejść do implementacji danej metody, przejść do następnej linijki itd.



Podstawy

Dodanie nowej zmiennej



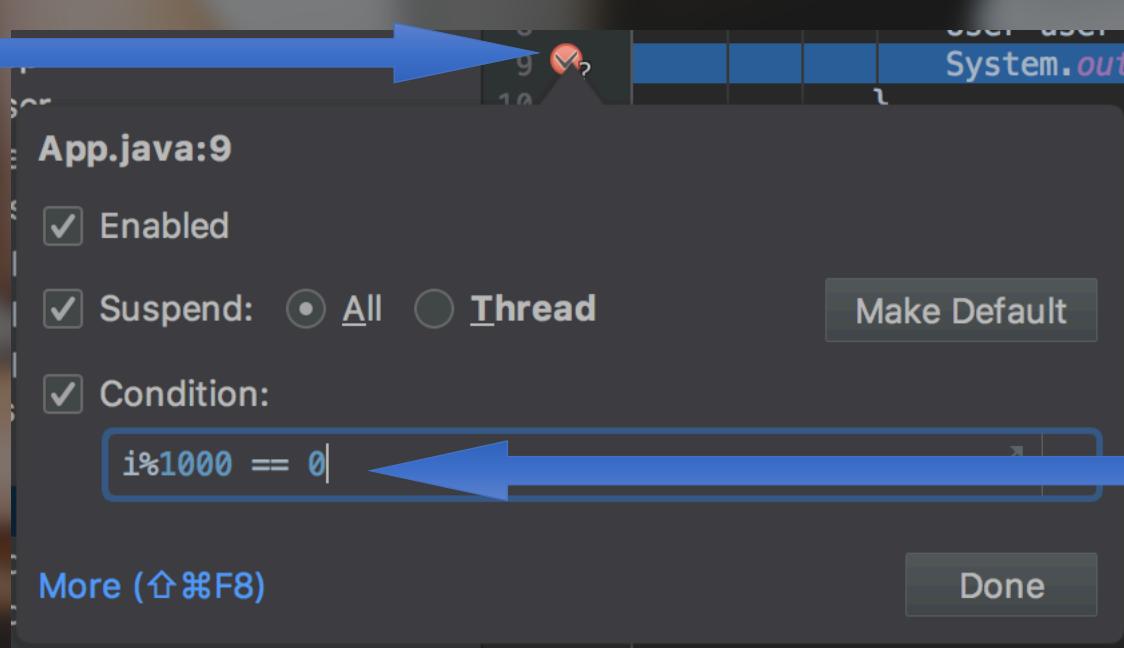
Dowolny kod javowy który ma zostać uruchomiony

Rezultat wywołania polecenia.

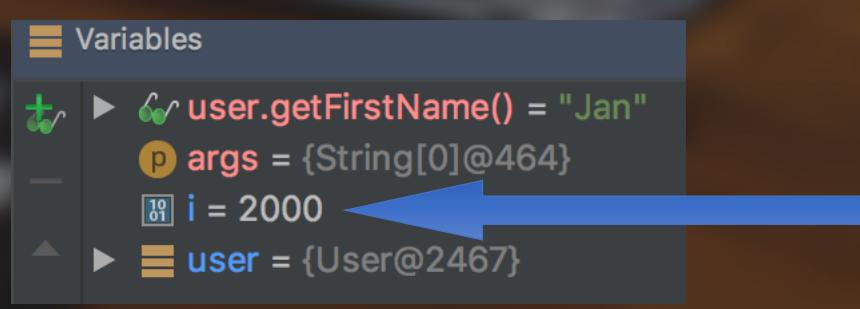
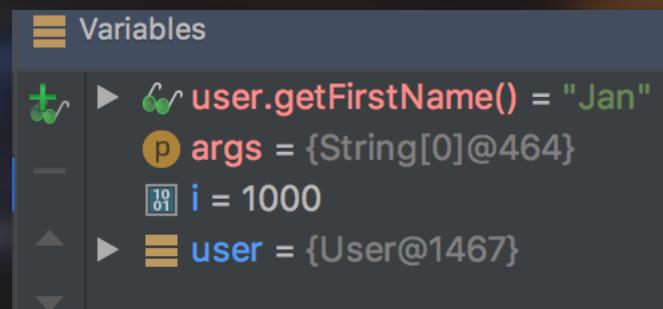


Podstawy

Kliknięcie PPM otwiera okno do zdefiniowania warunku po którego spełnieniu debugger zatrzyma działanie aplikacji



Zdefiniowanie warunku.
Warunek może być dowolnym kodem Java który musi zwracać true/false



Aplikacja zatrzymuje się tylko po spełnieniu warunku.