Pokročilé programování na platformě Java:

# Úvod do předmětu
# Java Virtual Machine (JVM)
# (novinky v Javě 7 a 8)

# Cíle předmětu - přehled

- Ekosystém platformy Java
  - IDE
  - JVM
  - Verzování
  - Správa projektu

- Problematika vývoje na platformě Java
- Životní cyklus projektu
- Tvorba větších aplikací v rámci Spring
- Aktuální trendy vývoje aplikací

# Cíle předmětu - vývoj

- Využití řady technologií tvořící základ větších projektů na platformě Java
- Správné návyky při programování
- Získané zkušenosti je možné přenést do libovolného jiného jazyka/platformy
- Vlastní mini-projekt postupně využívající nově naučené technologie
- Důraz bude kladen na pochopení způsobu řešení, ne znát z hlavy konkrétní implementaci

- **Být lepší programátor i SW architekt**

# Vstupní požadavky

- Zkušenosti s programováním v Javě
- Znalost principů webových technologií
- Základní znalost databázových systémů

- Chuť se učit nové věci
- Aktivní samostudium

# Vstupní testík - Java

- Kolekce Java
  - HashMap<Int,String>
  - ArrayList<String>
  - Set<String>
- Rozdíl mezi abstraktní třídou a rozhraním
- Modifikátory přístupu
- Konstruktor
- Vnitřní třídy, final, override
- Java Concurrency – Thread, ThreadPool, Lock

# Vstupní testík - Web

- HTTP – (Hlavičky, stavové kódy)

- Statické X Dynamické web stránky
- Blokující, neblokující, asynchronní komunikace
- Webový x aplikační server, proxy
- HTML, CSS
- JSON, XML

# Vstupní testík - DB

- Jak probíhá připojení k databázi
- Index, klíče, datové typy
- Kadinalita, parcialita
- Transakce

# Požadavky - zápočet

- Aktivní účast (max 2 absence)
- Odevzdání úloh (max 3 týdny po zadání)

# Požadavky - zkouška

- Absolvování testu (60%)  -  nejlépe za 3
- Ústní zkouška

# Proč JVM?

- JVM není jen Java (Scala, Groovy, Clojure, JRuby, Jython)
- Podpora pro řadu platforem (přenositelnost)
- Výkon, stabilita, popularita
- I velké projekty lze řešit na čistě open source řešeních
- Rozšířená a oblíbená v průmyslu

# Java není univerzální řešení

- Hledáme "Best tool for the job"
- "Law of the instrument"

  **If all you have is a hammer, everything looks like a nail**
- Cílem předmětu je ukázat cestu a princip jednoho způsobu řešení
- Stejné řešení lze realizovat v Python, Ruby, PHP, C, C# - cílem předmětu není porovnávat

# Jazyk Java

# Java

- Silně typový jazyk =typy všech proměnných musí být deklarovány před jejich použitím
  - Kontrola délky polí, nelze pracovat přímo s pointery
- Staticky typovaný jazyk
  - Typy známy při překladu
    Kontrolovány
- Interpretovaný/Kompilovaný - Java kóduje do bytecode a poté jej interpretuje a/nebo JIT kompiluje během běhu.
- Objektově orientovaný, funkcionální rysy (Java 8)
  - lambda výrazy a funkční rozhraní, které umožňují psát kód v funkcionálním stylu.
- Multiplatformní
  -program napsaný v Javě může běžet na jakémkoli operačním systému,
- Automatická správa paměti (GC)
- Aktuální verze je Java ~~8~~ 21

# JDK + JRE

- JRE
  - Běhové prostředí, pro spuštění zkompilovaného programu, např. jar
  - Obsahuje JVM
- JDK
  - JRE + nástroje pro vývoj
  - javac (kompiler), javap (dissasembler), jdb (debugger)

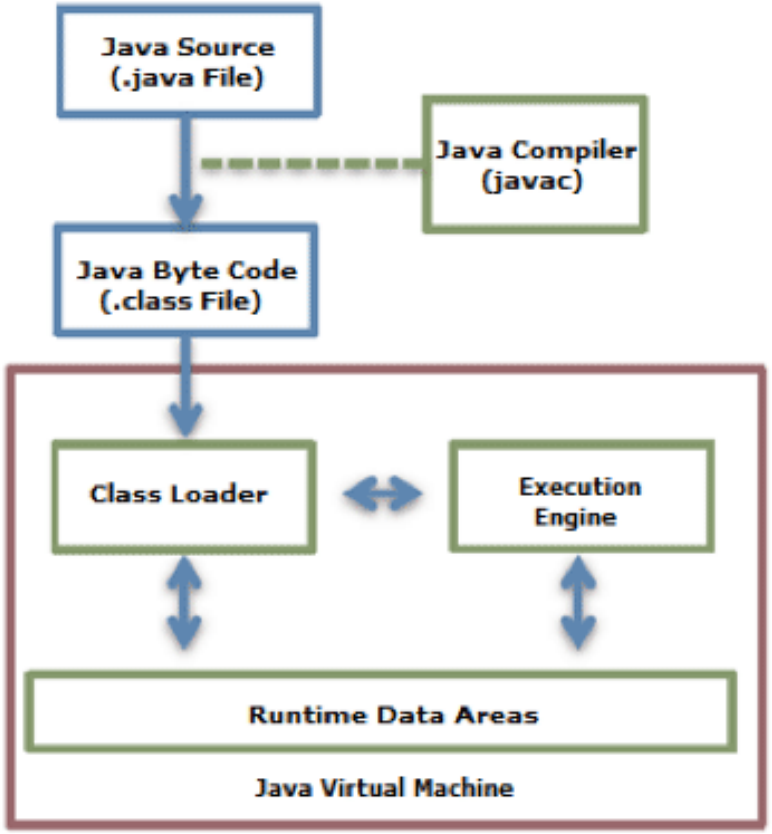
# JVM

- Softwarový stroj vykonávající instrukce (bytekód)
- V základu instrukce interpretuje, při častém volání je kód optimalizován a zkompilován JITem (Just in time kompilace) do strojového kódu daného systému
- Zásobníkový stroj
- Garbage Collector
- Pevně definované typy (Big Endian, nezávislost)

# Proč se zabývat architekturou JVM

- "Dobrý řemeslník zná své nástroje"
- Jazyk bez běhového prostředí je jako auto bez kol
- Schopnost řešit komplexní chyby (ty co těžko vygooglíte)
- Co vlastně to IDE dělá pod pokličkou

# Struktura JVM

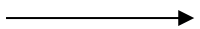



- Bytekód
- Classpath
- Classloader
- Paměťový model
- Běhový model

# Bytekód

- Strojový kód pro JVM
- Instrukce 1 byte + parametry
- Některé instrukce mají v názvu i typ
  - istore, astore, lstore, fstore

```
public class Example {
   public int plus(int a){
     int b = 1;
     return a + b;
   }
 }
```



```
Stack=2, Locals=3, Args_size=2
0:   iconst_1
1:   istore_2
2:   iload_1
3:   iload_2
4:   iadd
5:   ireturn
```

# Vytvoření a zobrazení bytekódu

- Program (Program.java)
- Kompilace (javac)
- javac Program.java -> Program.class
- Bytekód (Program.class)
- javap -v Main.class -> zobrazení bytecode

# Z třídy k archivu

- Pokud máme více tříd - Balík tříd (Java ARchive - JAR)
- jar cf my.jar  soubor1.class soubor2.class -> vytvoření archivu

- Archív je základní jednotka pro sdílení kódu
  - Knihovna  = 1 JAR

# Classpath

- Místo pro vyhledání tříd **classloaderem**
- Musí být schopen vyhledat všechny třídy i třídy knihoven – jinak ClassNotFoundException
- Specifikována pomocí systémové proměnné nebo parametru -classpath/-cp
- Bežně není třeba nastavovat, postará se o to IDE a nástroje pro sestavení (Maven)

# Classloader

- Součást runtime prostředí JVM
- Pracuje s proudem bytekódu
- Převádí ho na třídy (instance třídy Class)
- Načtení třídy při prvním použití
  - Třídy se nenačítají ihned při startu
- Hierarchie classloaderů – delegace
  - Bootstrap classloader – načtení core Java tříd (rt.jar)
  - Extension classloader – rozšíření - lib in JAVA_HOME
  - System-Classpath – načtení z classpath

# Spuštění archivu 1

- Ruční přidání archivu na classpath a uvedení vstupního bodu
  - java -cp my.jar cz.tul.Main
- Spuštění archivu pomocí manifestu (automatické přidání do classpath)
  - Musí obsahovat META-INF/MANIFEST.MF (vstupní bod)

```
Manifest-Version: 1.0
Created-By: 1.8.0_25 (Oracle Corporation)
Main-Class: cz.tul.Main
```

  - java -jar my.jar = spuštění

# Archiv jako knihovna

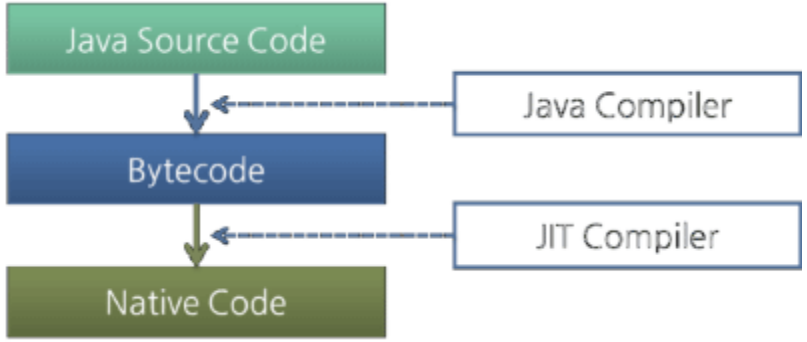
- Stačí přidat do classpath
- Pomocné informace v manifestu
  - Export-Package
  - Import-Package

# Spuštění archivu (souhrn)

- java -cp my.jar cz.tul.Main
- Načtení třídy cz.tul.Main pomocí classloaderu, který třídu nalezne na classpath (uvnitř my.jar)
- Zavolání vstupního bodu (metoda main)
- A co pak?
- **Běhový model JVM**
- **Paměťový model JVM**

# Běhový model JVM

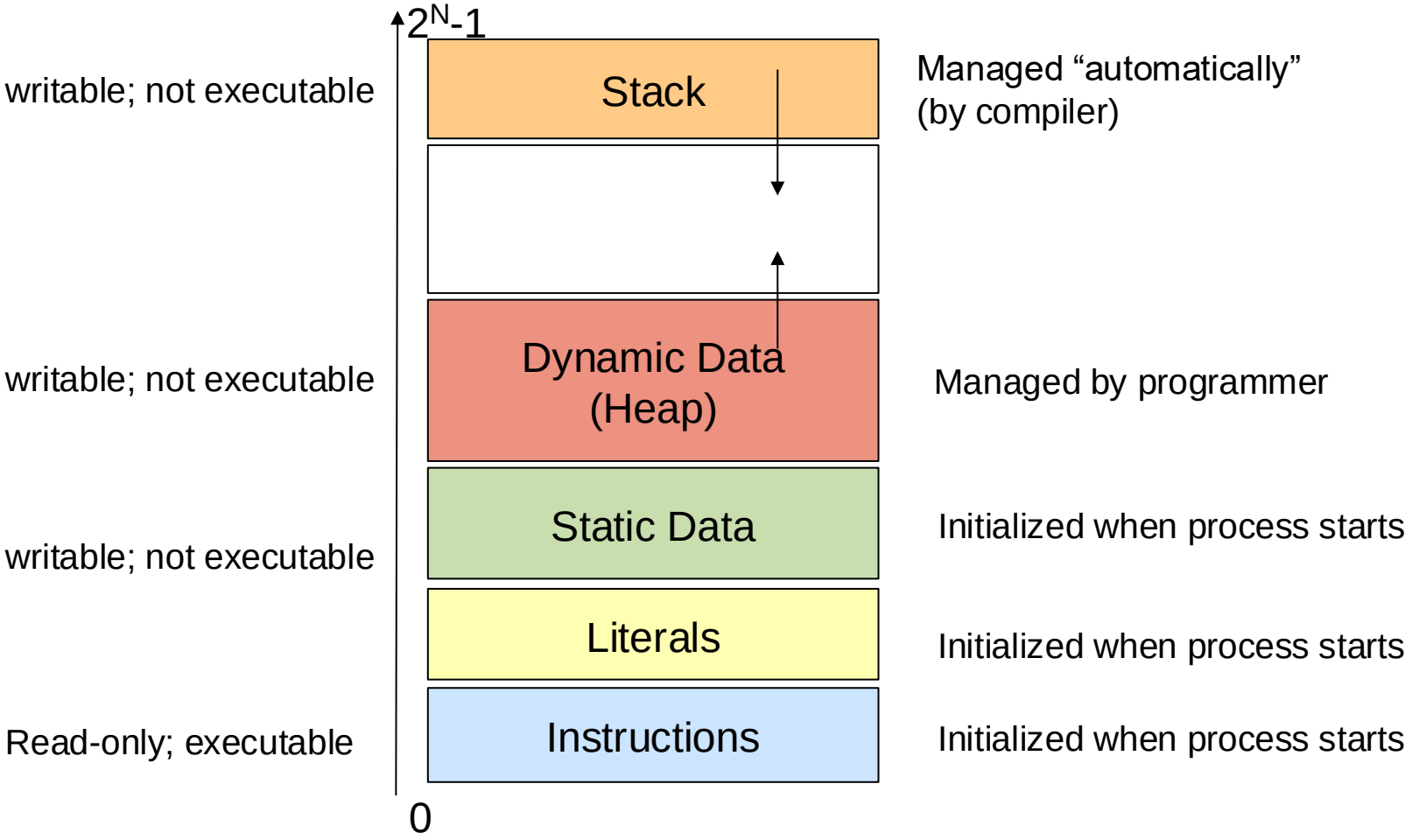
- Interpreter
  - Použit jako první
  - Pro málo volaný kód
- JIT Compiler
  - Pro „horký“ kód (Hotspot)
  - Přeloženo do nativních instrukcí
  - Lze nastavit úroveň optimalizací
    - Client
    - Server



# Paměťový model JVM

# Opakování – paměťový model - IA32 (i386)

- IA32 = 32-bit verze instrukčního setu x86 implementovaná poprvé na mikroprocesoru 80386 v roce 1985



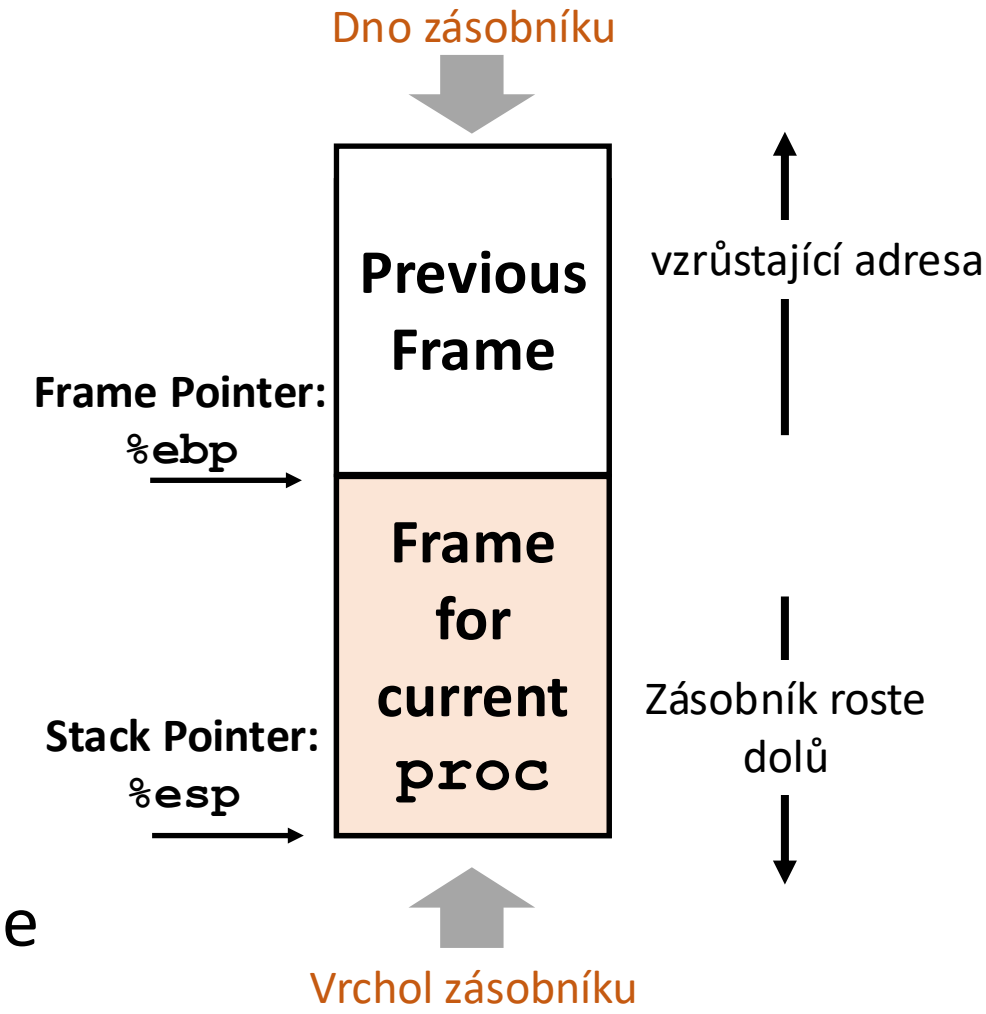
# Opakování – „zásobníkové jazyky“

- jazyky podporující rekurzy
  - např. C, Pascal, Java
  - vykonávaný kód je „vícenásobně přístupný“
    - jeho vykonávání může být přerušeno a po té bezpečně znovu spuštěno, aniž by předchozí přerušené vykonávání bylo ukončeno
    - jednotlivé procedury resp. funkce mohou běžet ve více instancích

  $\Rightarrow$ stav vykonávání každé procedury je třeba ukládat (na zásobník)
    - argumenty procedury
    - lokální proměnné
    - návratový ukazatel

# Zásobník – struktura a konvence

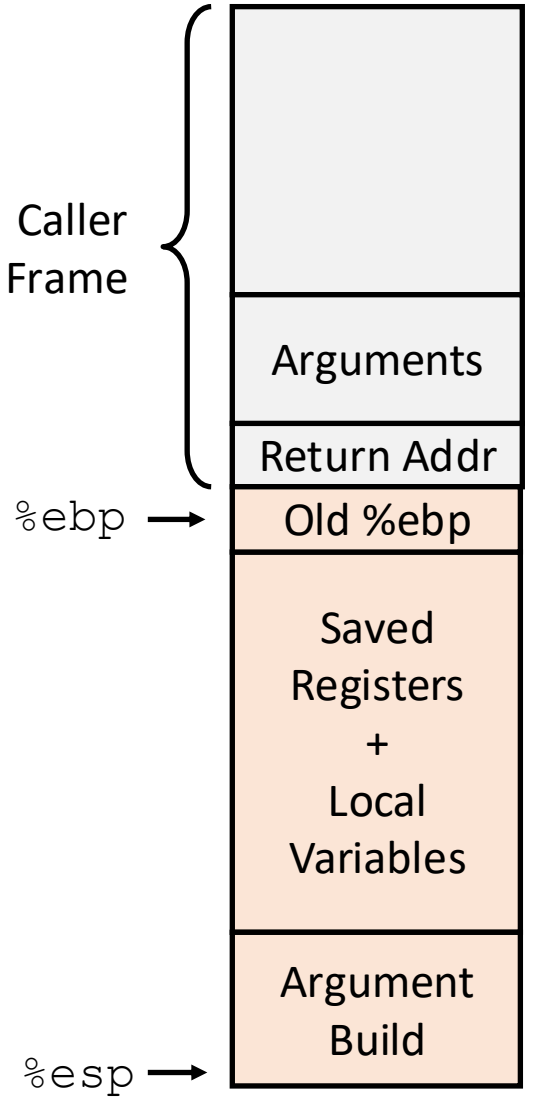
- stav procedury je ukládán po omezenou dobu
  - od času spuštění do času ukončení
  - zavolaná procedura musí vždy skončit dříve než ta, která volání vyvolala
- zásobník je rozdělen na rámce (framy)
  - rámec uchovává stav jedné instance procedury
- registr `%esp` obsahuje nejnižší adresu zásobníku respektive adrese prvku na vrcholu zásobníku
- registr `%ebp` obsahuje adresu začátku rámce



# Práce se zásobníkem při volání nové procedury

- rámec volající procedury obsahuje na svém konci
  - argumenty, s kterými byla procedura zavolána
  - návratovou adresu pro PC (program counter)
    - adrese instrukce ve zkompilovaném programu, kterou se má pokračovat po skončení procedury
- po zavolání nové procedury se vytvoří další rámec obsahující
  - minulou hodnotu registru `%ebp`
  - lokální proměnné a hodnoty dalších registrů
  - ….

⇒ hodnota registru `%esp` se při volání nové procedury snižuje a při skončení procedury naopak zvyšuje



# Další architektury

- x86-64
  - 64-bitové rozšíření architektury x86
  - zásobník je využíván jen minimálně
  - preferovány jsou registry
    - jsou rychlejší
    - je jich k dispozici více než v IA32
    - neexistuje frame pointer
      - adresy jsou počítány relativně ke stack pointeru
- ostatní architektury
  - společná snaha co nejvíce využívat registry

# Souvislost s JVM a Javou

- Java je zásobníkový jazyk
- Java je interpretovaný jazyk = je přeložen do bytekódu
  - program ke přeložen do bytekódu a implementován
  - část navíc kvůli rychlosti přeložena do nativního kódu dané platformy JIT kompilerem
- Součástí JVM je Garbage Collector
  - automatická dealokace nepotřebných dat
- Java podporuje od počátku multi-threading
- JVM napsaný v C++
  - odlišná implementace pro každou platformu

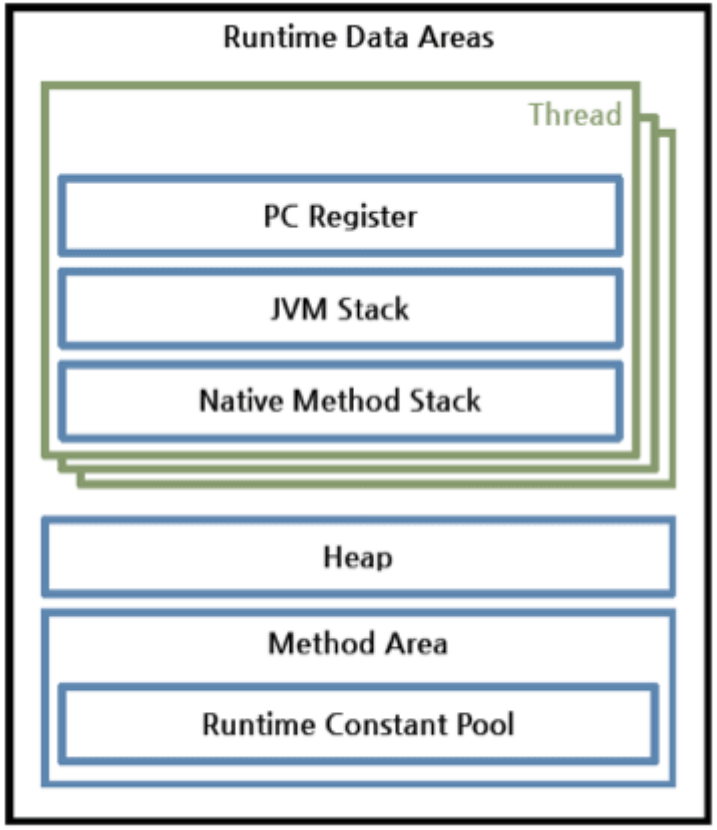
$\Rightarrow$ paměťový model JVM je složitý

$\Rightarrow$ pracuje se dvěma typy zásobníků (JVM pro interpretované metody a nativním)

# Paměťový model JVM - součásti

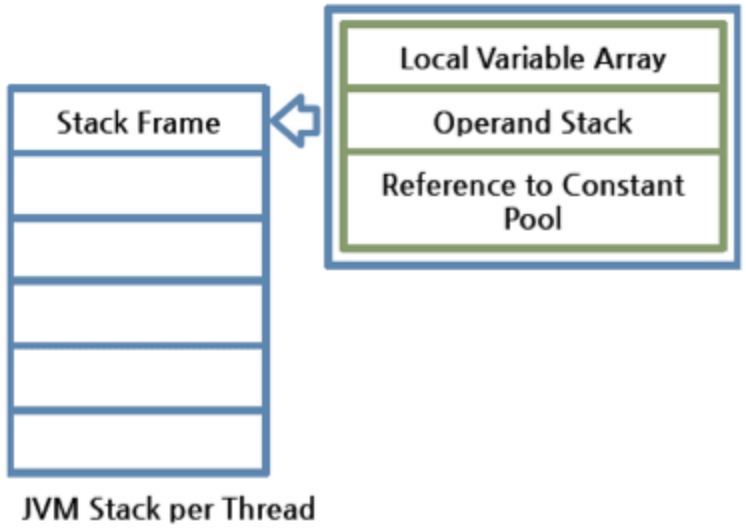
- **Halda**
  - Instance (objekty, pole, definice tříd)
- **Zásobník**
  - Lokální proměnné (argumenty, lok. prom.)
- Programové čítače pro vlákna JVM
- Metaspace (součást nativní paměti)
  - Místo uložení definic metod tříd (Method Area)
  - Konstanty + metada JVM, až v Java 7, předtím odlišný způsob – PermGen (vedle haldy)
- Code cache – pro JIT

# Paměťový model a vlákna




- Každé vlákno má vlastní
  - Program counter
  - Zásobník pro interpretované metody
  - Zásobník pro nativní metody
- Společné pro instanci JVM
  - Halda
  - Method area
    - Definice tříd, metod, atributů

# Paměťový model - rámec



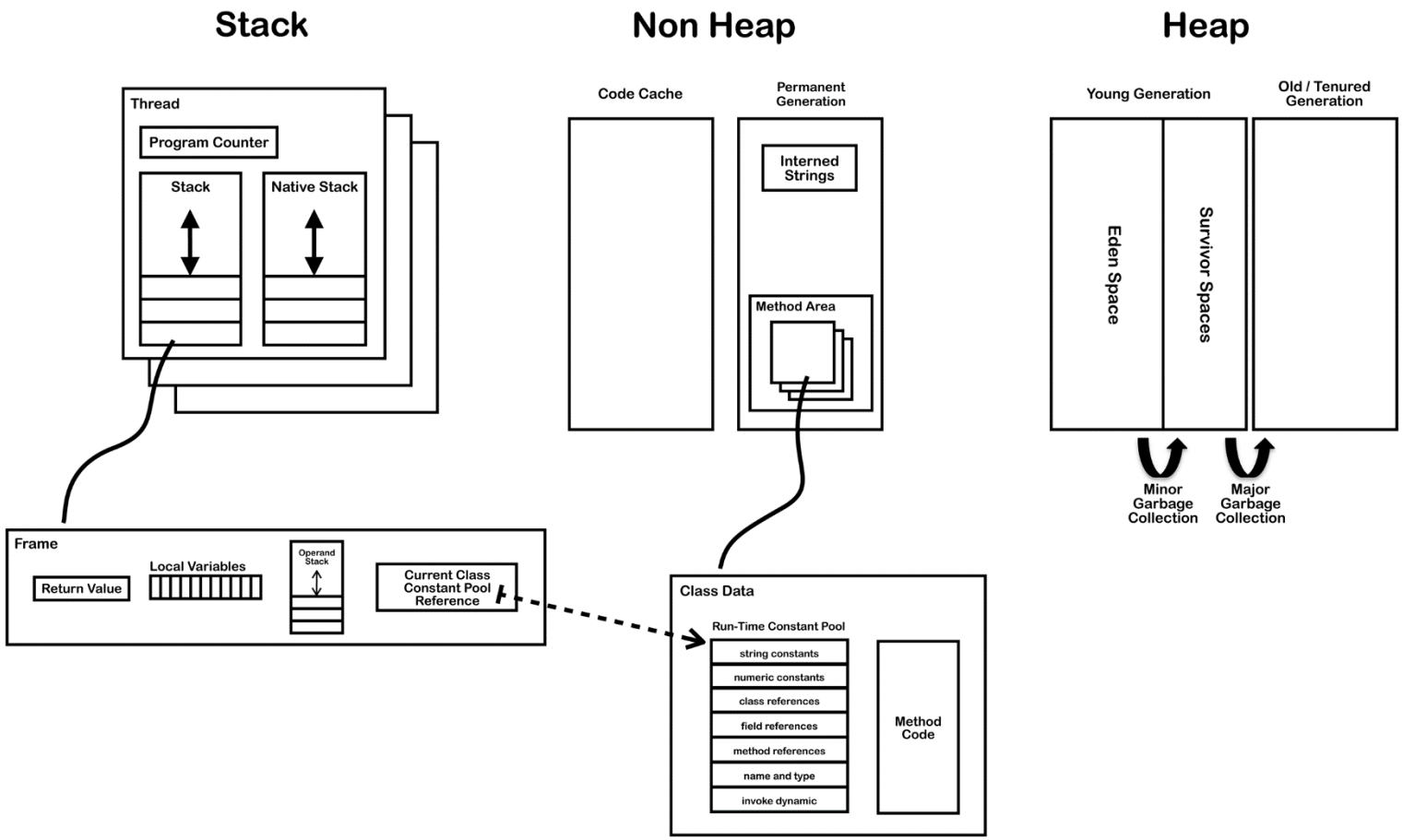

- Pole lokálních proměnných
  - Velikost známá předem
- Zásobník operandů
  - Reference na this
- Pole konstant
  - Viz. Metaspace
  - Při kompilaci relativní odkazy
  - Za běhu převedeny na absolutní

# JVM paměťový model – Twitter ed.

- Součásti
  - Zásobník + PC (per vlákno),
  - Halda (společná)
  - Metaspace (definice tříd, mimo paměť přidělenou pro JVM)
  - Code cache (pro JIT)

- Parametry, lok. proměnné -> zásobník
- Instance objektů, pole -> halda
- Definice tříd -> metaspace

# Paměťový model - kompletní schéma



# Garbage Collector

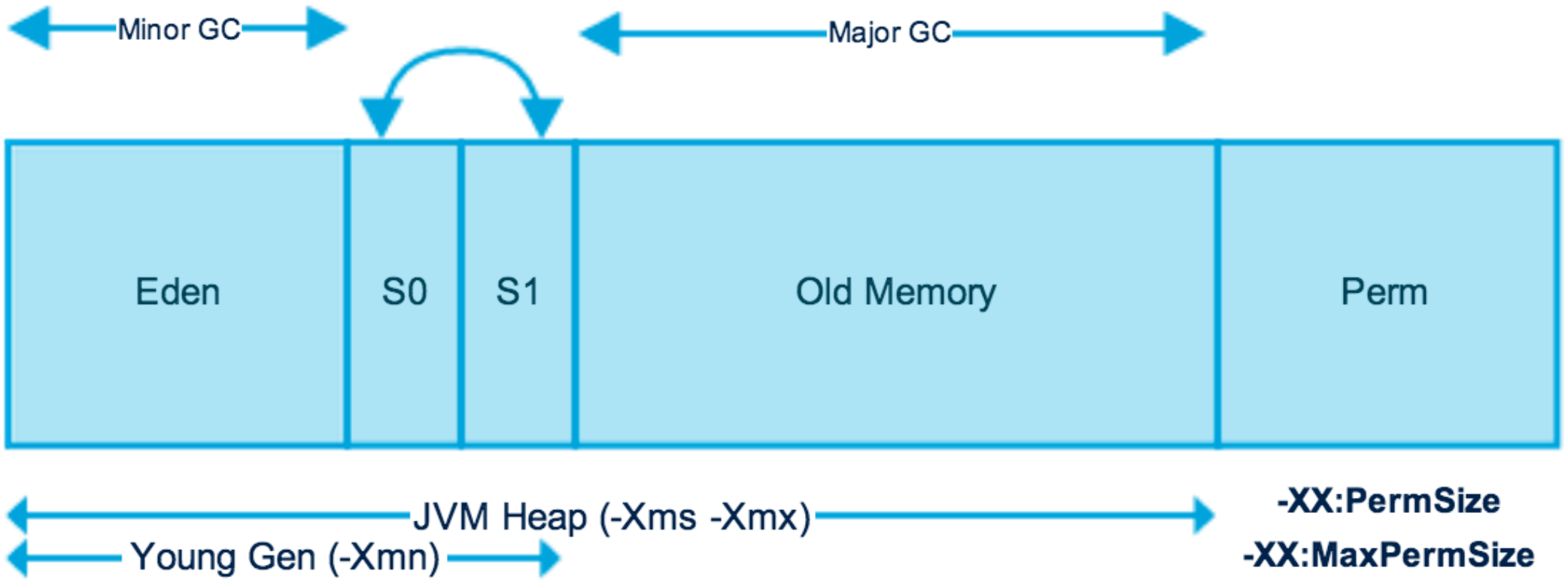
# Garbage collector (GC)

- Automatické navrácení paměti z nedostupných instancí objektů
  - Pracuje nad haldou popřípadě nad oblastí Perm
- Jak poznáme, že je instance nedostupná?
  - Neexistuje (bezpečný) způsob, jak se z libovolného vlákna dostat k jeho referenci
- Začneme z známých míst (z kořenů)
  - Např. lokální proměnné aktivních vláken na zásobníku
- Pracuje v cyklech
- Volán v případě nedostatku volné paměti

# Generační GC

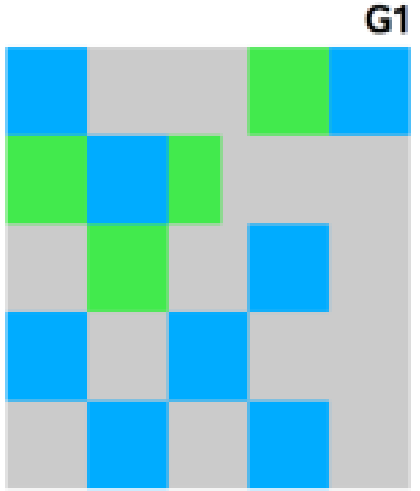
- Proč generační?
  - většina objektů je aktivní pouze krátkou dobu
  - zbytek je většinou aktivní mnohem déle
- Generace
  - Young – nově vytvořené objekty
  - Old – ty co přežily dostatečně dlouho v young
  - Perm – definice tříd + metada

# Generace haldy



# Další typy GC

- Serial GC - mark-sweep-compact
- Parallel GC - throughput GC
- CMS - velmi krátké **stop the word** události
- G1 (garbage first)
  - není generační, dělí haldu na bloky
  - nejprve řeší bloky s největším nepořádkem




http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

# Stop the world

- Abychom mohli pokračovat v cyklu kolektoru je nutné pozastavit (zaparkovat) všechny vlákna
- Aplikace prakticky zastaví a nevykonává užitečnou práci – problém s timeout
- Aby mohly vlákna zaparkovat, musí dorazit do místa, odkud je to bezpečné (návrat z metody, konec cyklu) – **safepoint**
- Problém pro realtime aplikace
- Pro velkou haldu (100GB+) může trvat i desítky sekund - problém pro uživatele – timeout požadavků

# Výhody GC

- Nemusíme se zabývat alokací a uvolňováním paměti
- Můžeme zvolit efektivní GC algoritmus dle chování aplikace
- Díky automatické správě odstraníme riziko vzniku tzv. memory leaků – pomalým dlouhodobým neuvolňováním paměti dojde aplikaci volná paměť a spadne (např. po  týdnu běhu) – extrémně těžké detekovat/ladit

# GC nevýhody

- Bez kontroly nad alokací a uvolňováním paměti – problém pro performance critical aplikace a low level aplikace (ovladače, OS)
- Stop the world události
- Zabírá část výpočetního výkonu (obvykle velmi malou) – stejně problém pro mobilní klienty

# GC - TL;DR ☺

- Automatická kolekce nedostupných instancí
- Průchod grafem od kořenů
- Stop the world událost – nutné zastavit vykonávání programu kvůli činnosti GC
- CMS, Serial/Parallel GC – generační
- G1 – blokové
- Vždy je výhodné využít platformy s GC pakliže pro to nemáme dobrý důvod (real-time aplikace, ovladače, velmi malé množství paměti).

# Otázky na běhové prostředí

- Bytekód
- Interpretace vs JIT kompilace
- Kdy se provádí JIT kompilace
- Jak funguje volaní funkce v C
- Jak funguje volání metody v JVM
- Rozdíl v schématu rámce C vs JVM

# Otázky - paměť JVM

- Popište hlavní komponenty paměť. modelu JVM + jejich funkci
- Pro vypsané entity napište jejich umístění v paměti
  - name
  - age
  - metoda greet
  - count
  - myString
  - m
  - result
  - řetězec “Ja”

```
public class Memory {

    private static final String myString = "Hello";
    private int count = 5;

    public String test(String name, int age) {
        String myName = "PPJ";
        return name + " " + age;
    }

    public static void main(String[] args) {

        Memory m = new Memory();
        String result = m.test("Ja",52);
    }
}
```

# Otázky - GC

- Příklad aplikací, pro které je nevhodné použít GC a pro které
- Uveďte příklad aplikace, která je vhodná pro generační kolektor a příklad nevhodné aplikace
- GC započne po ihned vytvoření instance, je prom. *greetingService* vhodná pro garbage kolekci, může být kořenem GC?

```
public class SimpleComponentWithAutowiredConstructor {

    private final GreetingService greetingService;

    @Autowired
    public SimpleComponentWithAutowiredConstructor(GreetingService greetingService) {
        this.greetingService = greetingService;
    }
```

# Reference

- Oficiální (JVM specifikace)
  - https://docs.oracle.com/javase/specs/jvms/se8/html
- Blogy
  - http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/
  - http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html
  - http://blog.jamesdbloom.com/JVMInternals.htm

# Novinky Java 7 a 8