7 Koncept jazyka na bázi virtuálniho stroje, JVM memory management, datové struktury, výjimky, objektové programování, vlákna a synchronizace. (A4B77ASS)

7.1 Typy programovacích jazyků

Z hlediska vykonávání zdrojového kódu lze programovací jazyky dělit do dvou kategorií.

- **Interpretované** - přímo zdrojový kód (skript či překompilovaný byte-code) je interpretován virtuálním strojem, který běží na cílovém zařízení
- **Kompilované** - zdrojový kód je nutné zkompilovat do strojového kódu cílového zařízení a poté lze program spustit přímo

Vlastnosti interpretovaných jazyků:

- $\oplus$ nezávislost na platformě - architektura (RISC/CISC), operační systém
- $\oplus$ reflexe - sledování a modifikace kódu za běhu
- $\oplus$ dynamické typování
- $\oplus$ malá velikost zdrojových souborů
- $\ominus$ pomalejší vykonávání kódu v interpretovaném módu

7.2 Java Virtual Machine (JVM)

Je zásobníkově orientovaný virtuální stroj Javy, který interpretuje Java byte-code. Zdrojové kódy (`.java`) je proto nutné zkompilovat do byte-code (`.class`), při kompilaci nedochází k žádným optimalizacím kódu. Před spuštěním je byte-code verifikován (skoky jsou pouze na validní umístění, správná inicializace dat, type-safe reference, kontrola private a protected přístupů). Při běhu se používá JIT.

Zásobníkový způsob předávání parametrů: $(2 + 3) \times 11 + 1$

Input	2	3	add	11	mul	1	add
Stack	2	3 2	5	11 5	55	1 55	56

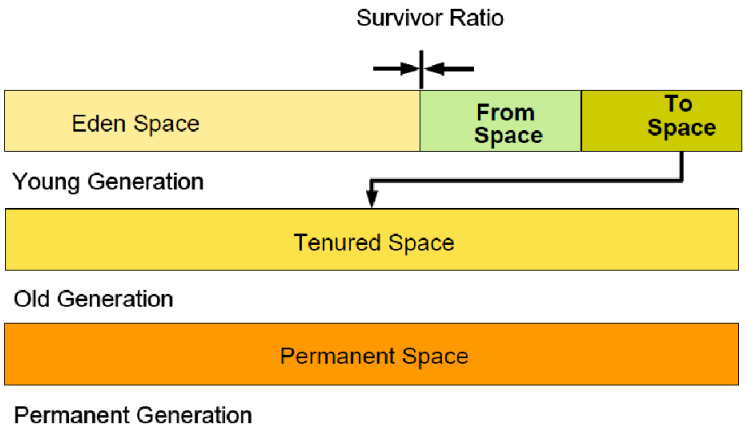
7.2.1 Just-in-time compiler (JIT)

Za běhu konvertuje byte-code na nativní strojový kód. Existují různé verze pro serverové a klientské aplikace (různý stupeň optimalizace). Některé prováděné optimalizace: inlining of functions, loop unrolling, dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, optimize branches.

7.3 JVM memory management

V JVM je použita automatická správa paměti pomocí **Garbage Collectoru (GC)** - živé (dosažitelné) objekty jsou ponechány v paměti a mrtvé (nedosažitelné) jsou smazány. Halda (**Heap**) je oblast pro dynamickou alokaci paměti pro všechny objekty a je rozdělena do secí, tzv. generací - Young a Old.

Generační koncept Objekty jsou v haldě rozděleny do generací podle svého stáří, zpočátku jsou alokovány v Young generaci, pokud přežijí několik cyklů GC jsou povýšeny (Tenuring) do Old generace. Koncept je postaven na hypotéze, že většina objektů je krátko-žijících, tj. nedostanou se do Old generace.



Young malá velikost, časté a rychlé cykly GC

Old velká velikost, málo časté a pomalé cykly GC

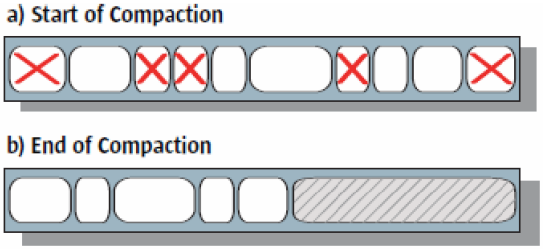
Eden místo pro alokaci nových objektů

From/To místo kam jsou zkopírovány přeživší objekty po běhu GC

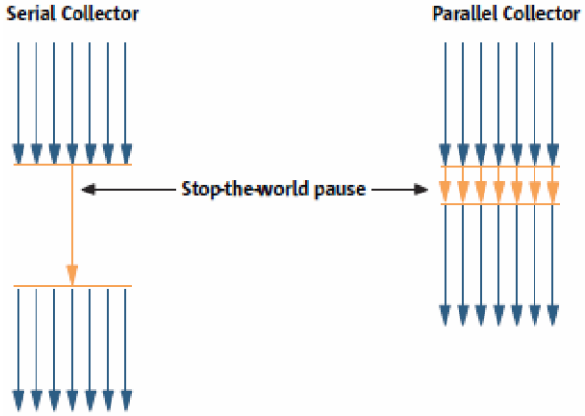
Permanent je mimo haldu, obsahuje data pro JVM jako definice tříd, metod a další

7.3.1 Druhy Garbage Collectorů

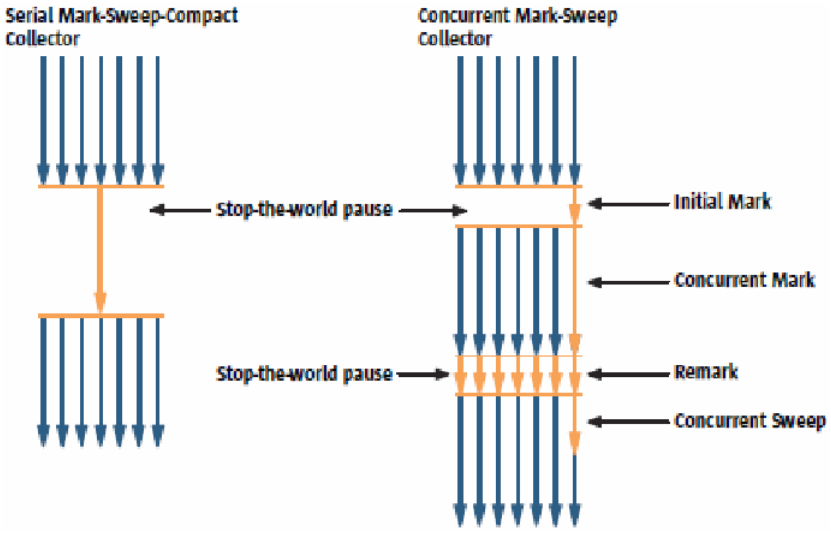
- **Sériový collector** - používá algoritmus *mark-sweep-compact* způsobem *stop-the-world*, výchozí pro klientské aplikace, efektivní na 64MB haldě



- **Paralelní collector** - pro Old generaci obyčejný sériový collector, pro Young generaci navíc využívá více vláken/CPU, výchozí pro serverové aplikace a na vícejádrových systémech



- **Souběžný (Concurrent) collector** - pro Young generaci jako paralelní collector, pro Old generaci běží souběžně s aplikací, má nízkou latenci a snižuje počet zastavení aplikace, vyžaduje ale větší haldu



7.4 Java datové struktury

- **primitiva** - bez implicitní alokace, uložené na zásobníku, typy: boolean, byte, char, int, long, float, double
- **objekty** - každý je potomek třídy Object, uloženy na haldě, existují objekty pro primitiva: Boolean, Byte, Char, Integer, Long, Float, Double
- **pole (arrays)** - speciální datová struktura pro uchovávání více primitiv/objektů stejného typu v lineárním pořadí, mají definovaný limit, který je automaticky za běhu kontrolován, jsou uložené na haldě, více-dimenzionální pole = pole polí

7.4.1 Autoboxing, Unboxing, Widening

Autoboxing automatická konverze primitivních datových typů na jejich objektové reprezentace

Unboxing opačný postup, objekty na primitiva

Widening automatická konverze menších primitiv na větší:
byte $\prec$ char $\prec$ int $\prec$ long a float $\prec$ double

Používání boxingu a unboxingu přináší neefektivitu. V případě více možností má widening přednost před autoboxingem. Při volání přetížených metod Java vybere tu s nejvíce specifickými typy parametrů.

7.4.2 Výjimky

Všechny extendují třídu Throwable (Error, Exception) a slouží k reprezentaci různých chybových stavů aplikace. Výjimky lze dělit do dvou kategorií - kontrolované (Exception) a

nekontrolované (Error a RuntimeException). Kontrolované výjimky je nutné odchytávat pomocí **try-catch** bloku. Pro vyhození vlastní výjimky v programu souží příkaz **throw**. Výjimky také obsahují záznam obsahu zásobníku ve chvíli kdy byla tato výjimka vyhozena a případně další informace o vzniklé chybě.

7.5 Vlákna a synchronizace - Java

7.5.1 Vlákna (Threads)

Vlákna umožňují souběžné vykonávání více úkolů najednou, vytváří se buď extendováním třídy Thread nebo implementací rozhraní Runnable. Hlavní je metoda **run()** obsahující kód, který vlákno vykonává. Jeden proces může mít více vláken, která sdílí společný adresní prostor, ale všechny mají vlastní zásobník a lokální proměnné. Každé vlákno má: ID, jméno, prioritu, thread group, uncaught exception handler, deamon flag, class loader, interrupted flag, status.

7.5.2 Synchronizace

Při práci s více vlákny je často nutná jejich synchronizace a umožnění bezpečného přístupu ke sdíleným prostředkům. K tomu existuje několik technik.

- **Synchronized** - S každým objektem je asociován tzv. *monitor*, který umožňuje synchronizaci vláken nad tímto objektem použitím bloku **synchronized**. Pro vstup do tohoto bloku je nutné aby příslušné vlákno vlastnilo tento monitor, v jednu chvíli ale může monitor vlastnit pouze jedno vlákno. Pokud není monitor pro vlákno dostupný, jeho běh je zastaven dokud se monitor neuvolní. Pro synchronizaci celé metody je možné použít klíčové slovo **synchronized**.
- **Reentrant Locks** - Obdoba předchozí synchronizační techniky s manuálním získáváním a uvolňováním zámků - metody: **lock()**, **unlock()**, **tryLock()**. Zámky je nutné manuálně vytvářet jako objekty ReentrantLock, zámky je možné vytvořit *fair*, tj. první čekající vlákno bude první kdo získá zámek. Používání Reentrant zámků je v praxi efektivnější než použití synchronized, ale zámky jsou běžné objekty na haldě.

```
lock.lock();
try {
    // do some staff ...
} finally {
    lock.unlock()
}
```

- **Volatile proměnné** - Jsou obyčejné proměnné, které ale mají zaručen atomický read/write přístup. Tyto proměnné nejsou nikdy uloženy lokálně pro jednotlivá vlákna, ale je vždy přistupováno přímo do hlavní paměti. Nevhodné pro *read-update-write* operace. Použití pomocí klíčového slova **volatile**.

- **Atomické proměnné** - Umožňují atomický read/write přístup i *read-update-write* operace. Používají se speciální atomické instrukce procesoru - **CMPXCHG** (compare and exchange) nebo **CAS** (compare and swap). Pro základní datové typy existují AtomicBoolean, AtomicInteger, AtomicLong a AtomicReference s operacemi **get()**, **set()**, **compareAndSet()**, **addAndGet()**, **incrementAndGet()**, **decrementAndGet()**.
- **Neblokující algoritmy** - Synchronizační algoritmy, které nepoužívají zámky ani čekání vláken, ale jsou postavené na atomických CAS operacích. Princip je, že se v nekonečné smyčce kontroluje stav atomické proměnné, dokud není možné bezpečně pokračovat dále. Tento princip výkonnostně převyšuje blokující algoritmy protože většina CAS operací uspěje na první pokus. Další výhodou je, že odstraňuje nadbytečné uspávání vláken a přepínání kontextu.

7.6 Návrhové vzory pro objektové programování

- **Immutable object** - objekt, který po dobu svého života nemění žádné své vlastnosti, all fields are final, no setters
- **Factory method, Abstract factory** - k vytváření objektů se používají speciální metody či tovární třídy
- **Lazy initialization** - k výpočtům, inicializacím objektů a dalších dat dochází až v případě prvního použití
- **Singleton** - pro danou třídu existuje pouze jedna instance v rámci celé aplikace
- **Multiton** - pro danou třídu existuje několik instancí s různými parametry v rámci celé aplikace
- **Strategy** - definice společného rozhraní pro skupinu podobných algoritmů, umožňuje zaměňovat různé implementace
- **Composite** - skládání objektů do stromových struktur, ke skupině objektů je přistupováno jako k jediné instanci (Nodes and Leafs)
- **Iterator** - umožňuje sekvenční procházení kolekce po jednotlivých prvcích bez odhalení vnitřní struktury této kolekce
- **Command** - zastřešuje veškeré informace potřebné pro volání metod později - client, invoker, receiver