Uklízení odpadků a analýza úniku

Jiří Vokřínek

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 11

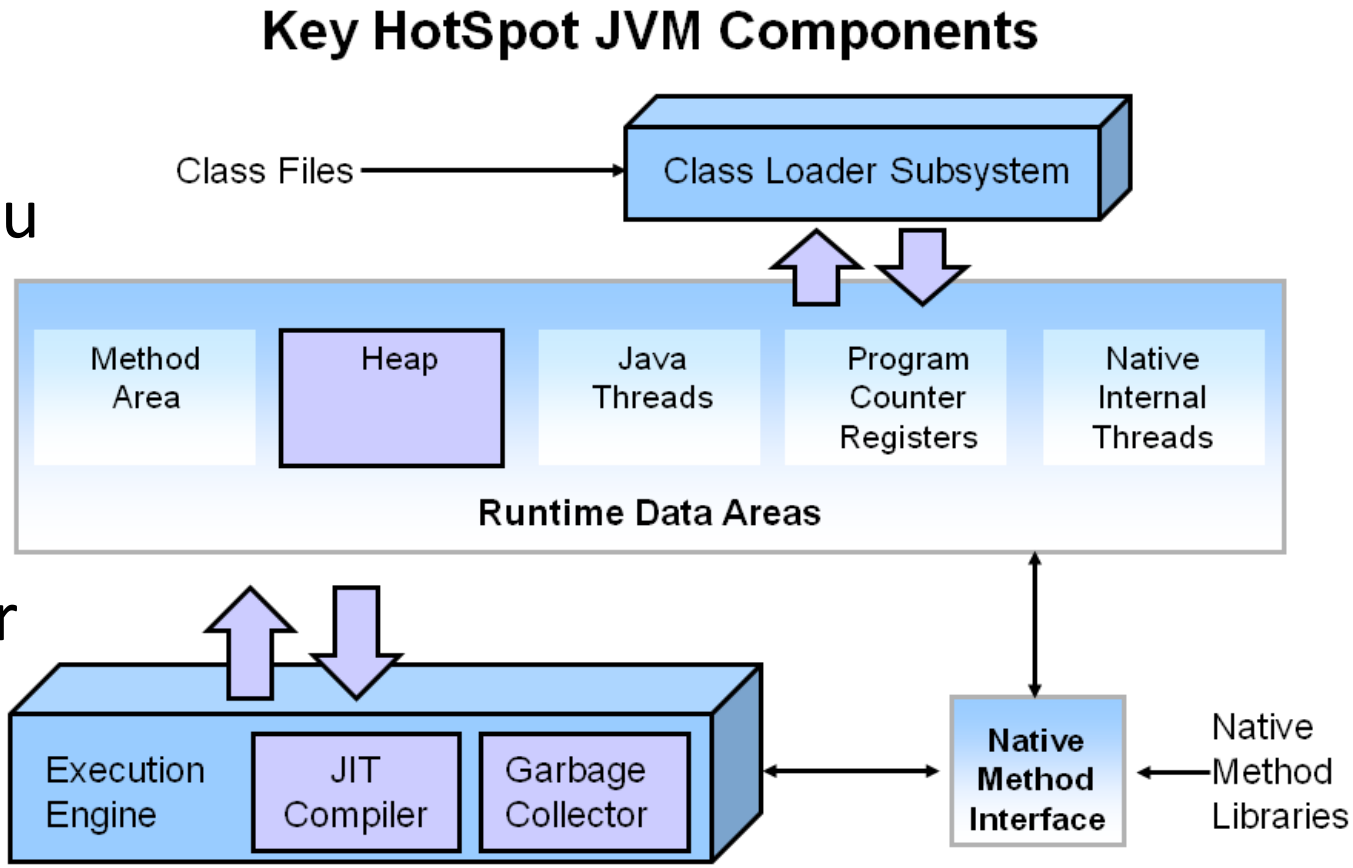
B0B36PJV – Programování v JAVA

Uklízení odpadků a analýza úniku

- Správa paměti
- Garbage Collection
- Escape Analysis
- Příklady
  - GC
  - EA
  - „memory leak“

JVM a optimalizace vývkonu

- Heap – ukládání objektů
- Garbage Collector – úklid heapu
- JIT – kompilace za běhu do nativního kódu



Správné nastavení heapu a výběr správného GC má vliv na výkon

Většinou ale stačí default nastavení

Správa paměti

Heap

- Společný paměťový prostor
- Ukládání objektů a dat
- Spravován GC

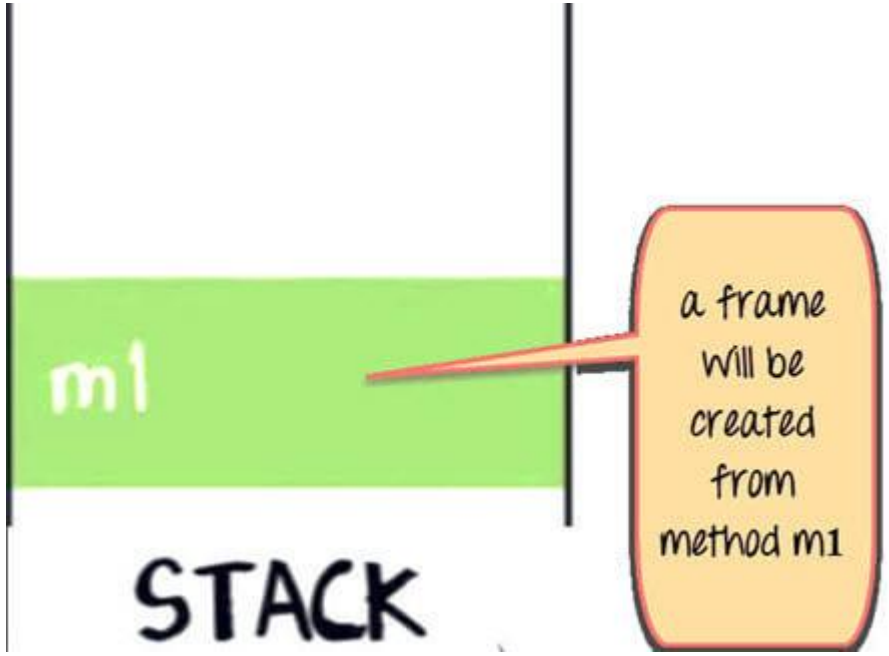
Stack

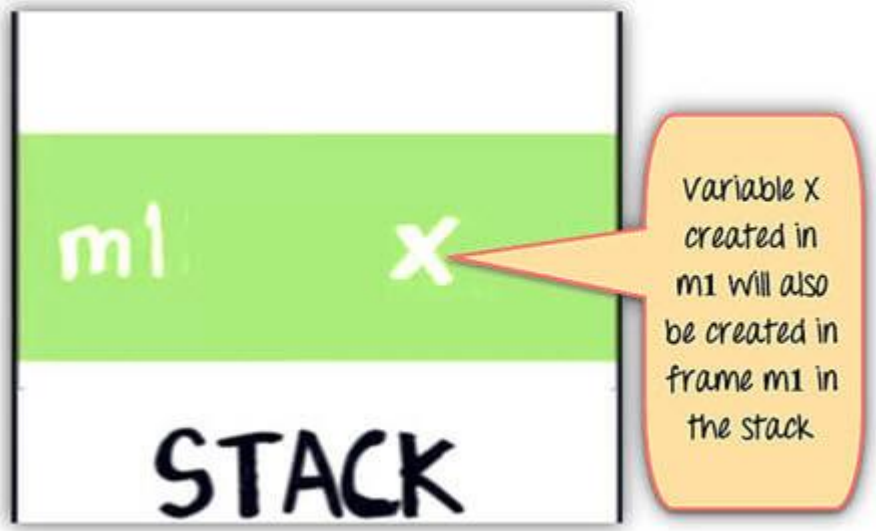
- Dočasný paměťový prostor
- Lokální proměnné a reference
- Pro každé volání metody vlastní stack, který je po návratu uvolněn jako celek

Příklad z https://www.guru99.com/java-stack-heap.html

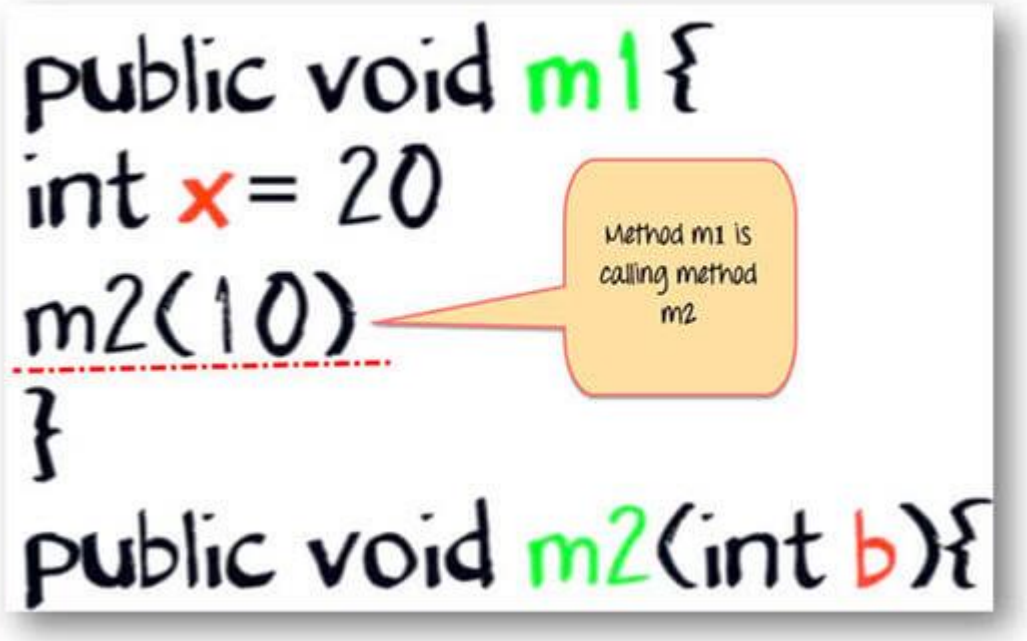
Správa paměti

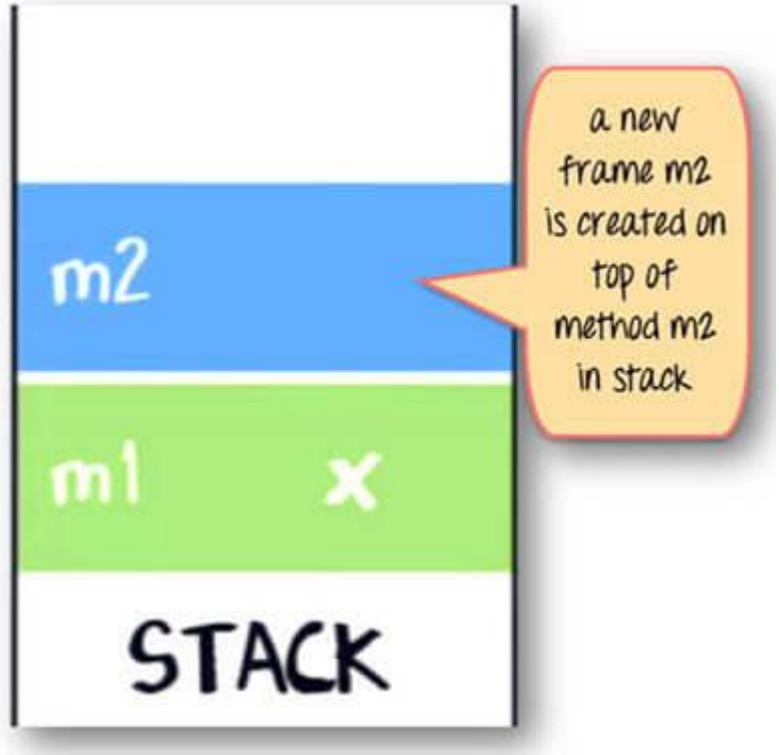
```
public void m1{
int x=20
}
```





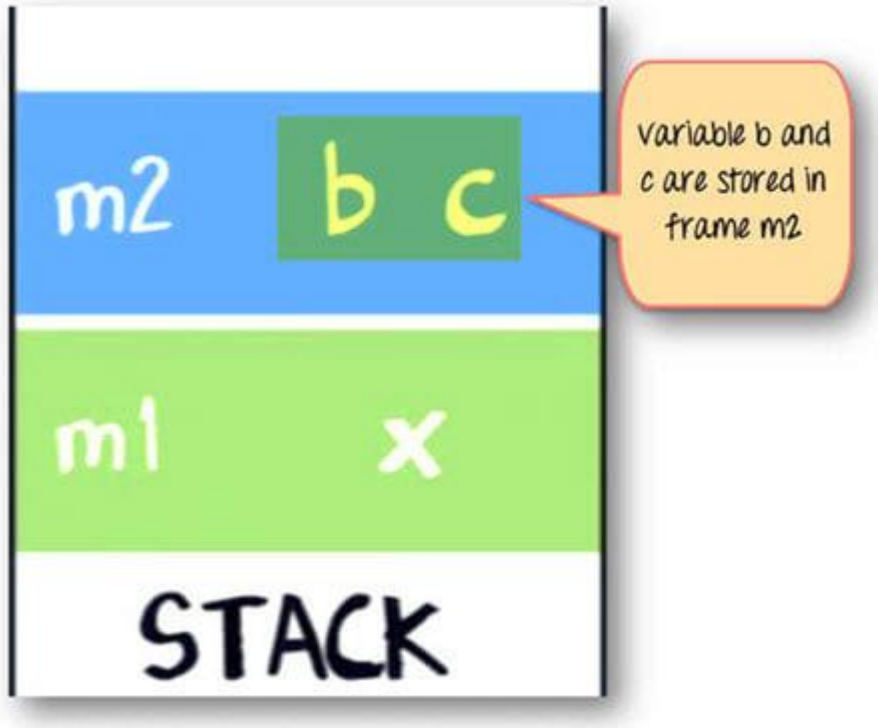
Správa paměti





Správa paměti

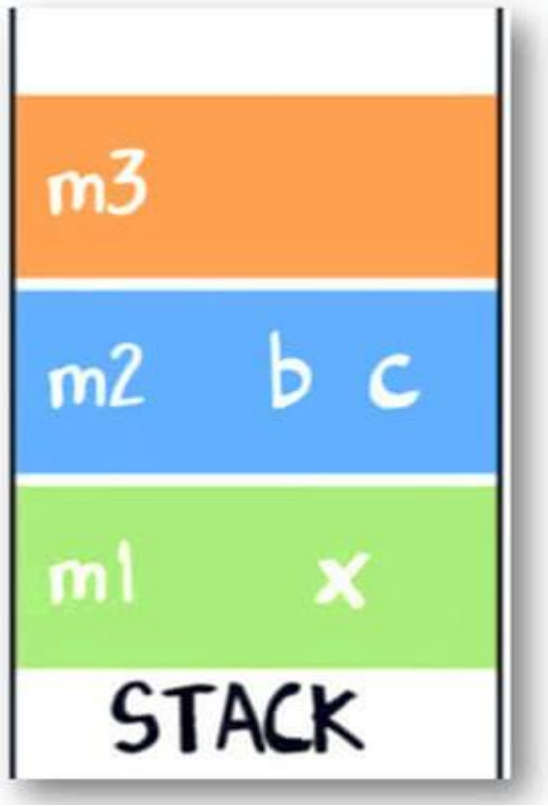
```
public void m2(int b){
boolean c;
}
```



Správa paměti

```
public void m2(int b){
    boolean c;
    //more code
    m3();
}
public void m3()
```

another method m3 is called by method m2

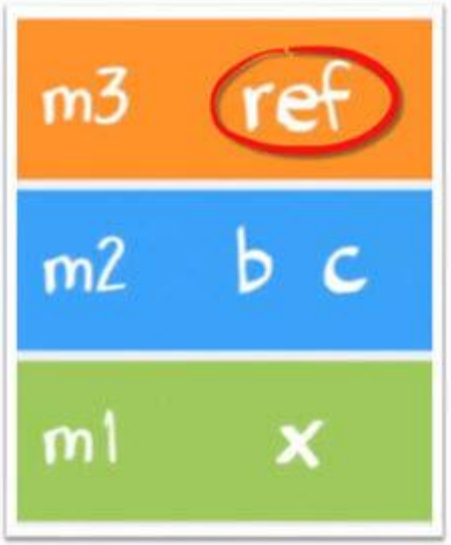


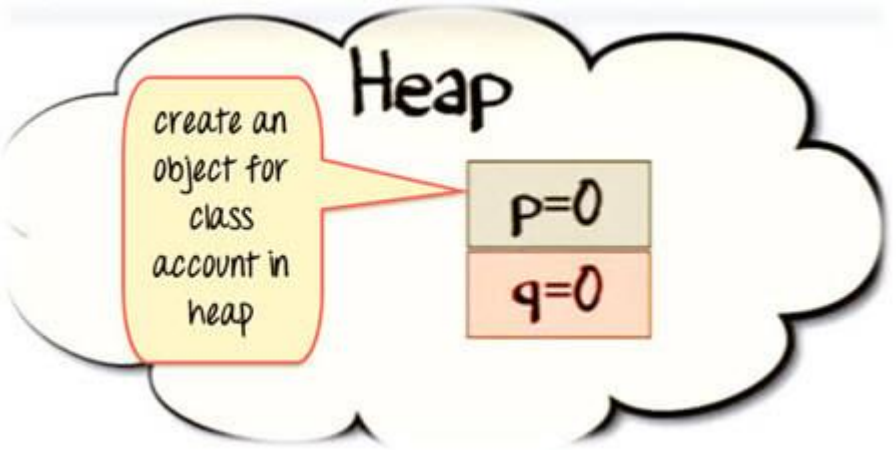
Správa paměti

```
Account {
            Int p;
            Int q;
        }
```

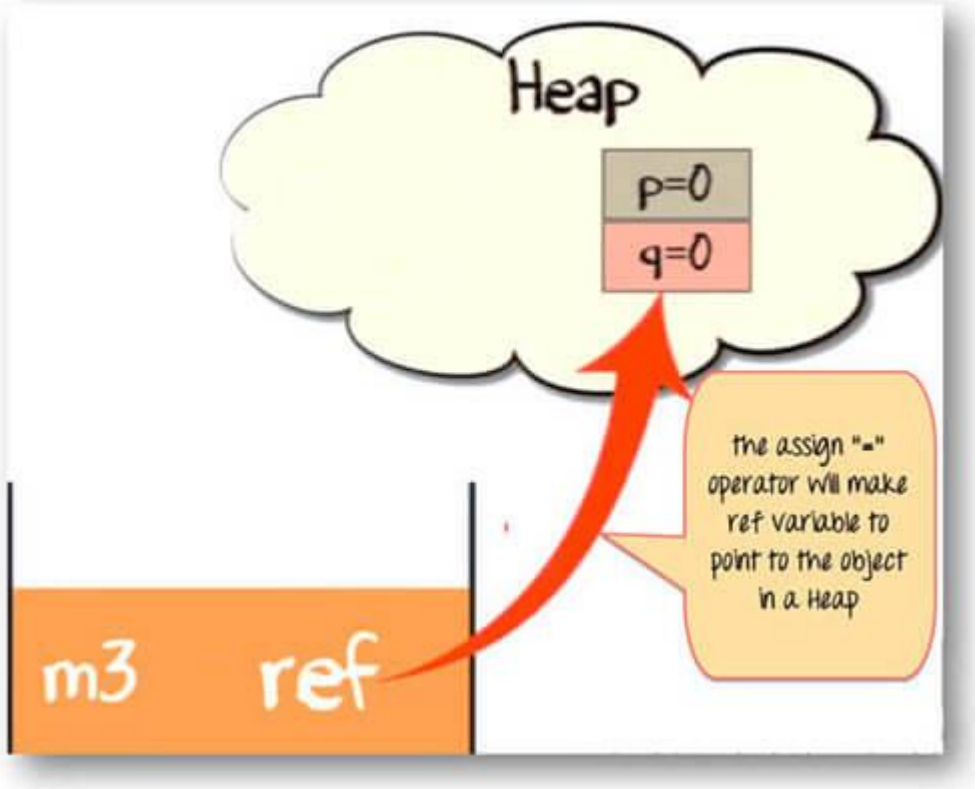
```
public void m3(){
        Account ref = new Account();
        // more code
}
```

Správa paměti

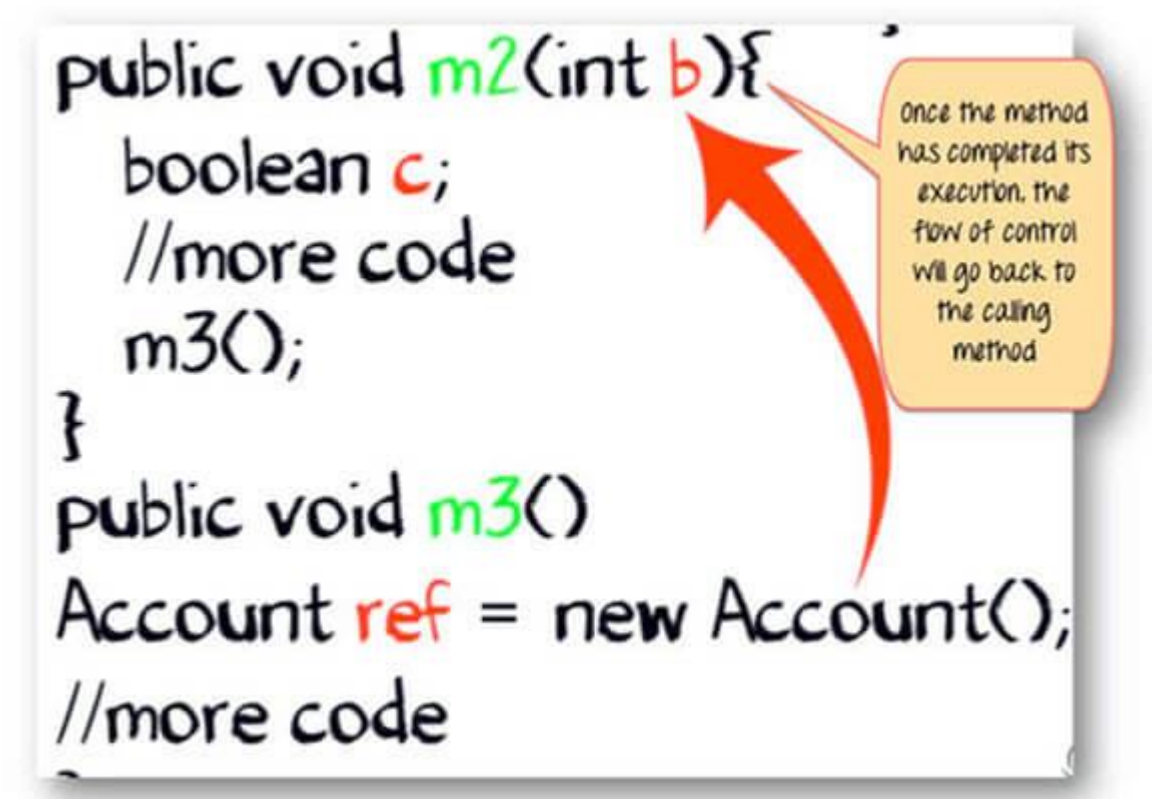


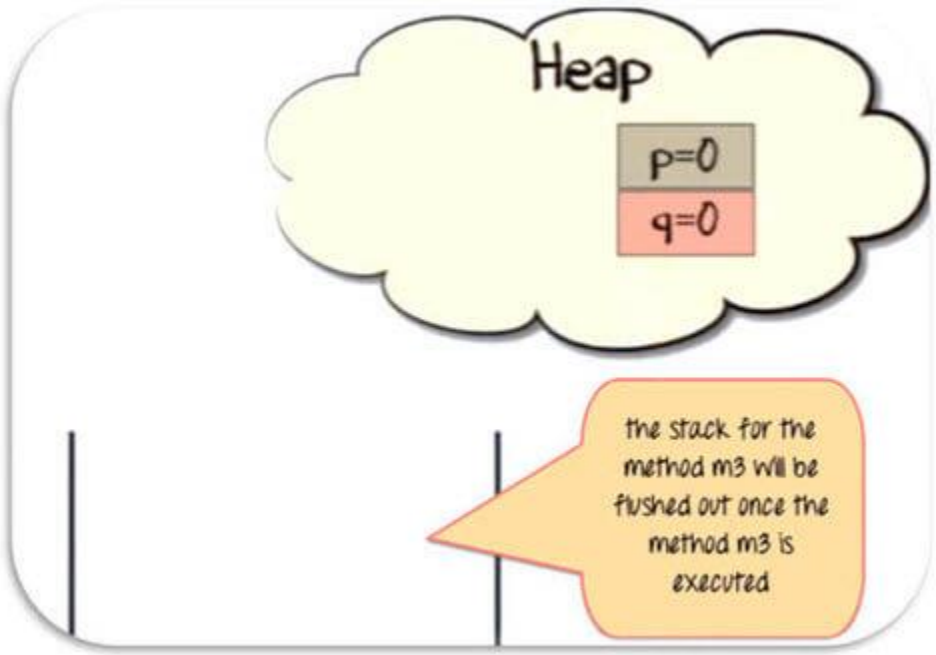


Správa paměti



Správa paměti





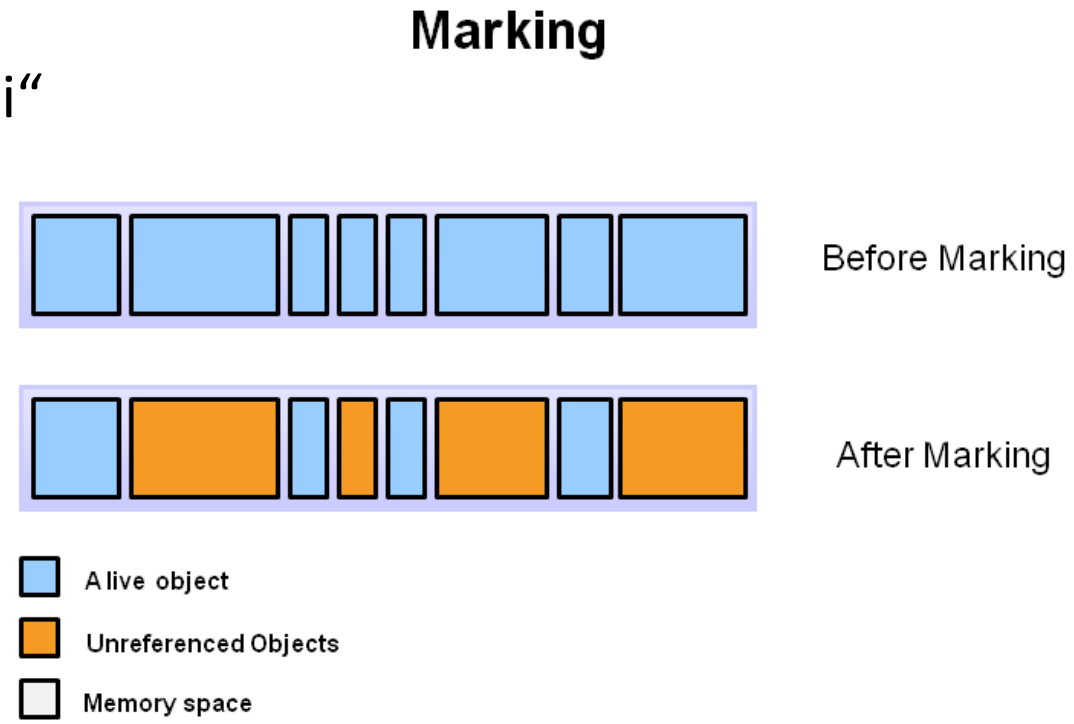
Správa paměti

Správa paměti

- Pokaždé, když je volána metoda, rezervuje se příslušná část stacku.
- Jakmile je metoda dokončena, běh programu se vrací na místo volání a příslušný stack je uvolněn.
- Proměnné v argumentech metody jsou vytvořeny na stacku.
- Lokální proměnné jsou vytvářeny na stacku.
- Referenční proměnné jsou vytvářeny na stacku.
- Instanční proměnné jsou vytvářeny na heapu.

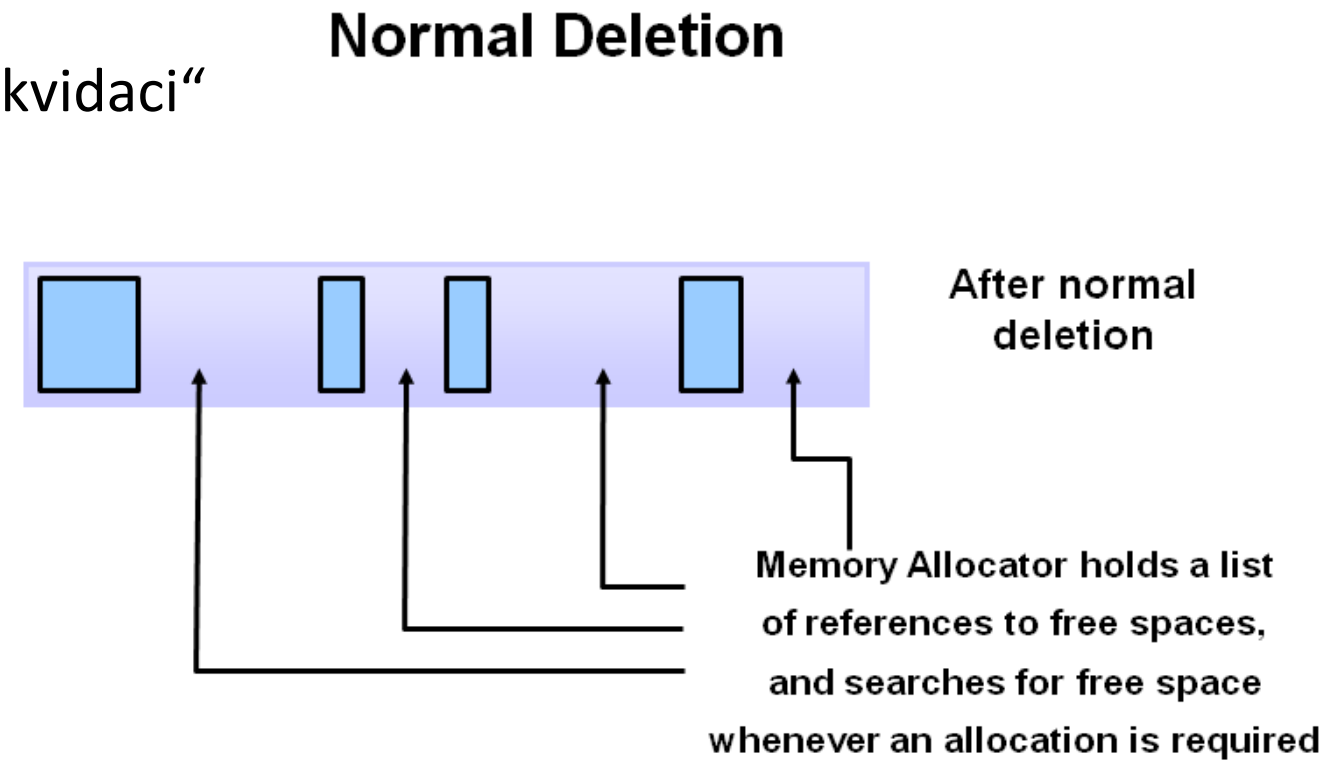
Garbage collection

- Běží automaticky na pozadí
- Muže být různý v závislosti na implementaci JVM a nastavení
- Obecně probíhají fáze:
  - Označení objektů určených k „likvidaci“



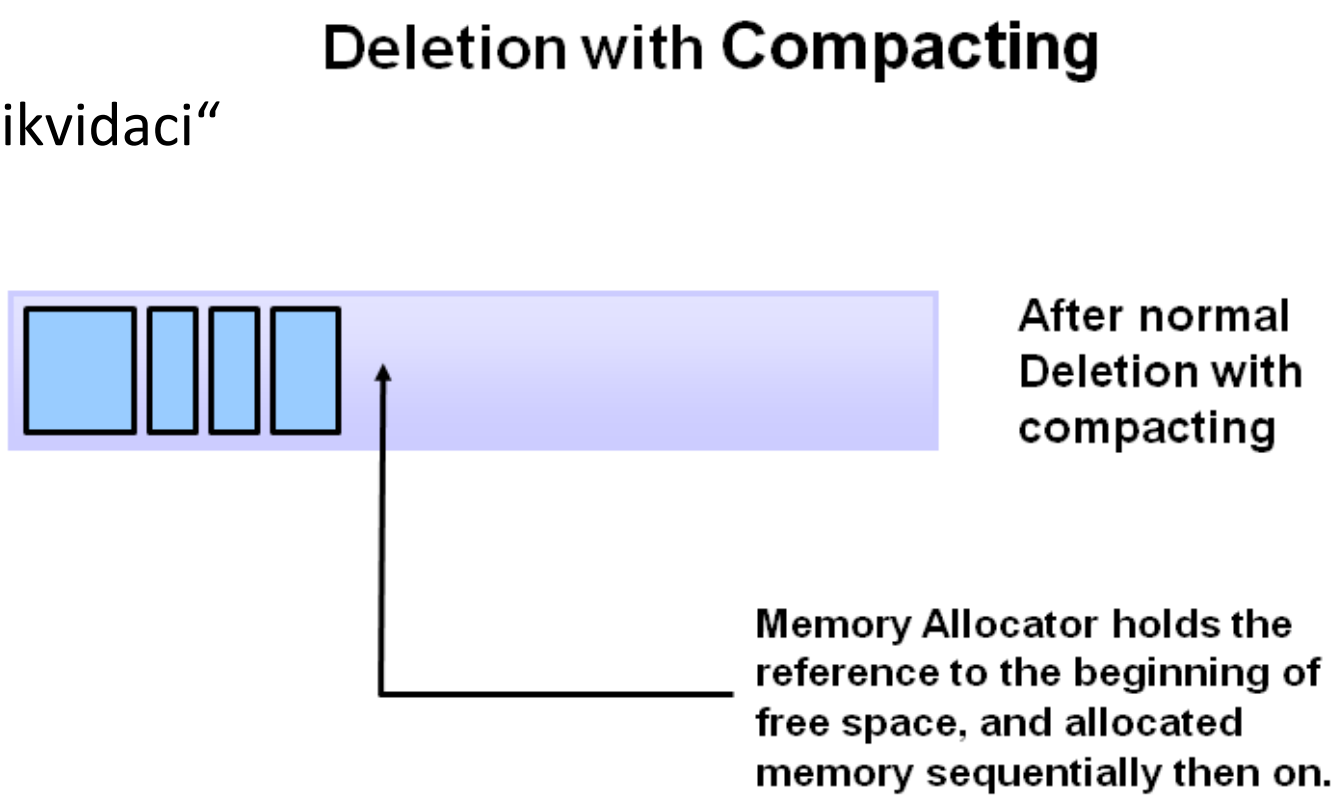
Garbage collection

- Běží automaticky na pozadí
- Muže být různý v závislosti na implementaci JVM a nastavení
- Obecně probíhají fáze:
  - Označení objektů určených k „likvidaci“
  - Uvolnění paměti



Garbage collection

- Běží automaticky na pozadí
- Muže být různý v závislosti na implementaci JVM a nastavení
- Obecně probíhají fáze:
  - Označení objektů určených k „likvidaci“
  - Uvolnění paměti
  - Zkompaktnění heapu



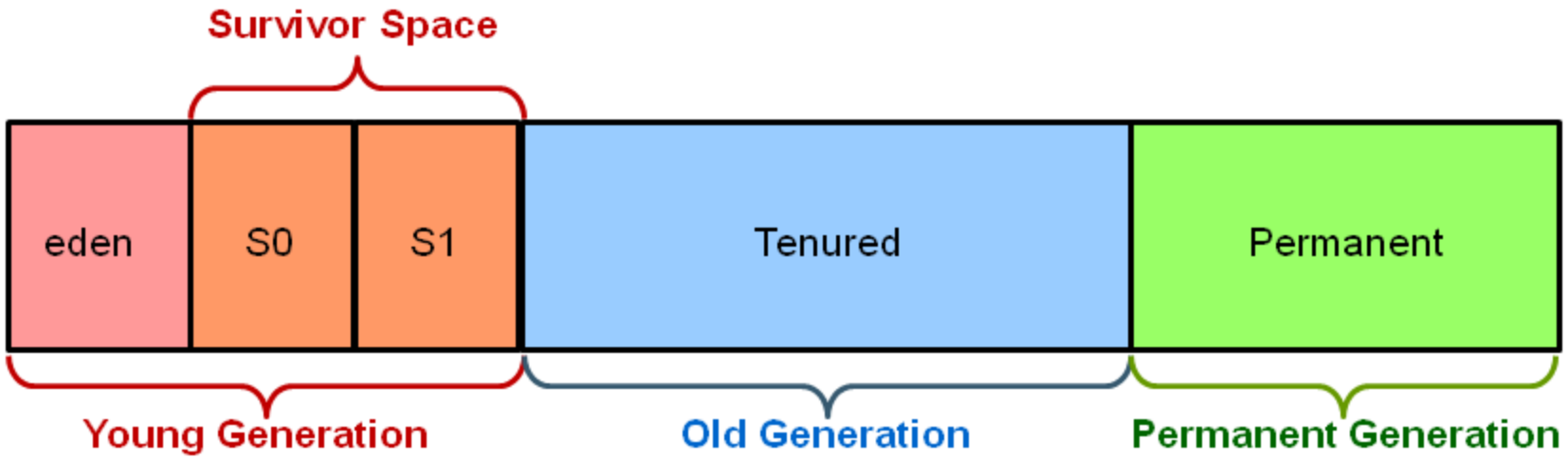
Garbage collection

- Běží automaticky na pozadí
- Muže být různý v závislosti na implementaci JVM a nastavení
- Obecně probíhají fáze:
  - Označení objektů určených k „likvidaci“
  - Uvolnění paměti
  - Zkompaktnění heapu

- Většina objektů zaniká brzy po vzniku
  - „Generační“ GC

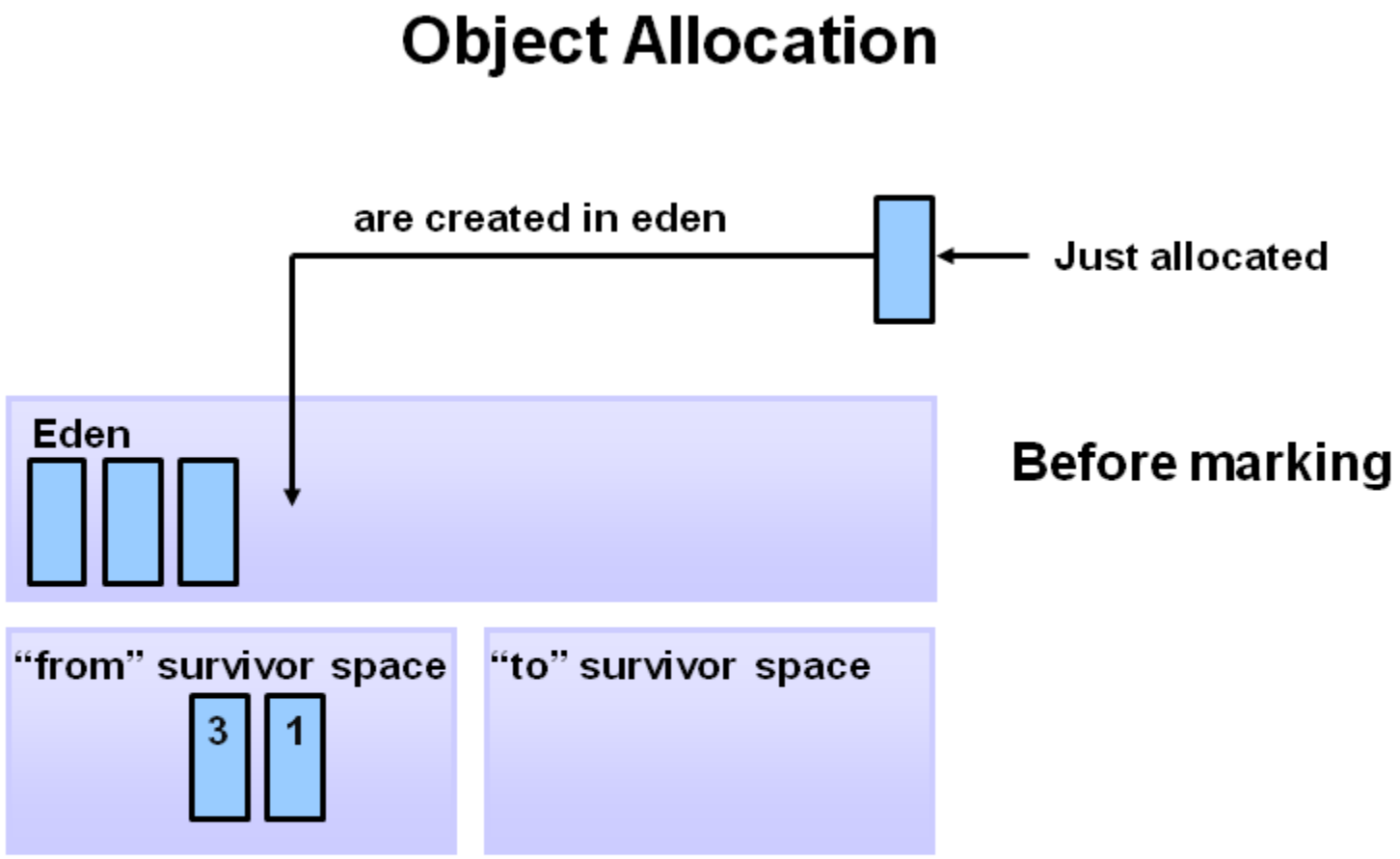
Generační GC

- Struktura HotSpot heapu

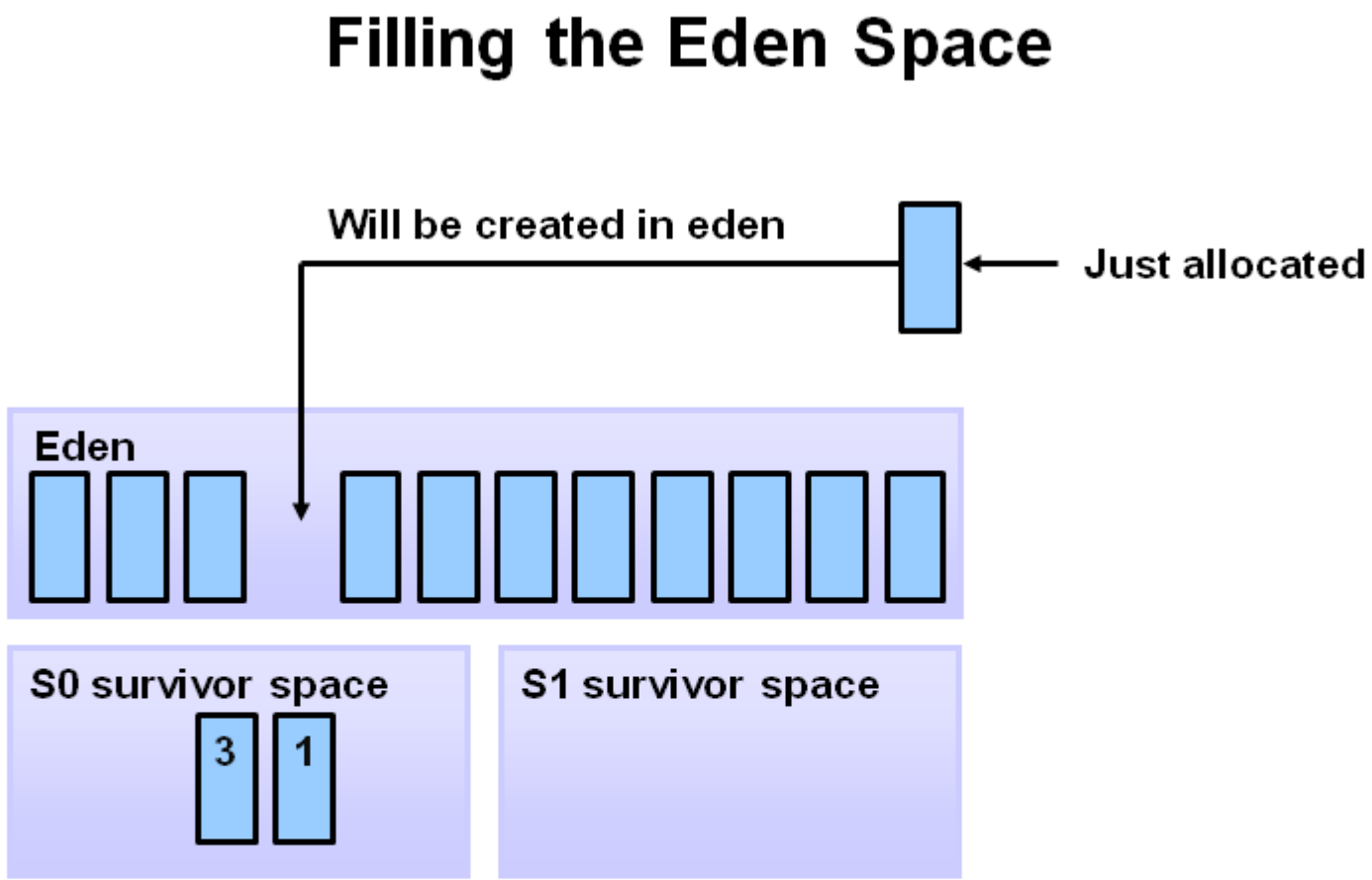


- **Minor garbage collection** při zaplnění *Young Generation*
- „Přeživší“ jsou přesouvány do *Old Generation*
- **Major garbage collection** při zaplnění *Old Generation*
- **Full garbage collection** kontroluje vše

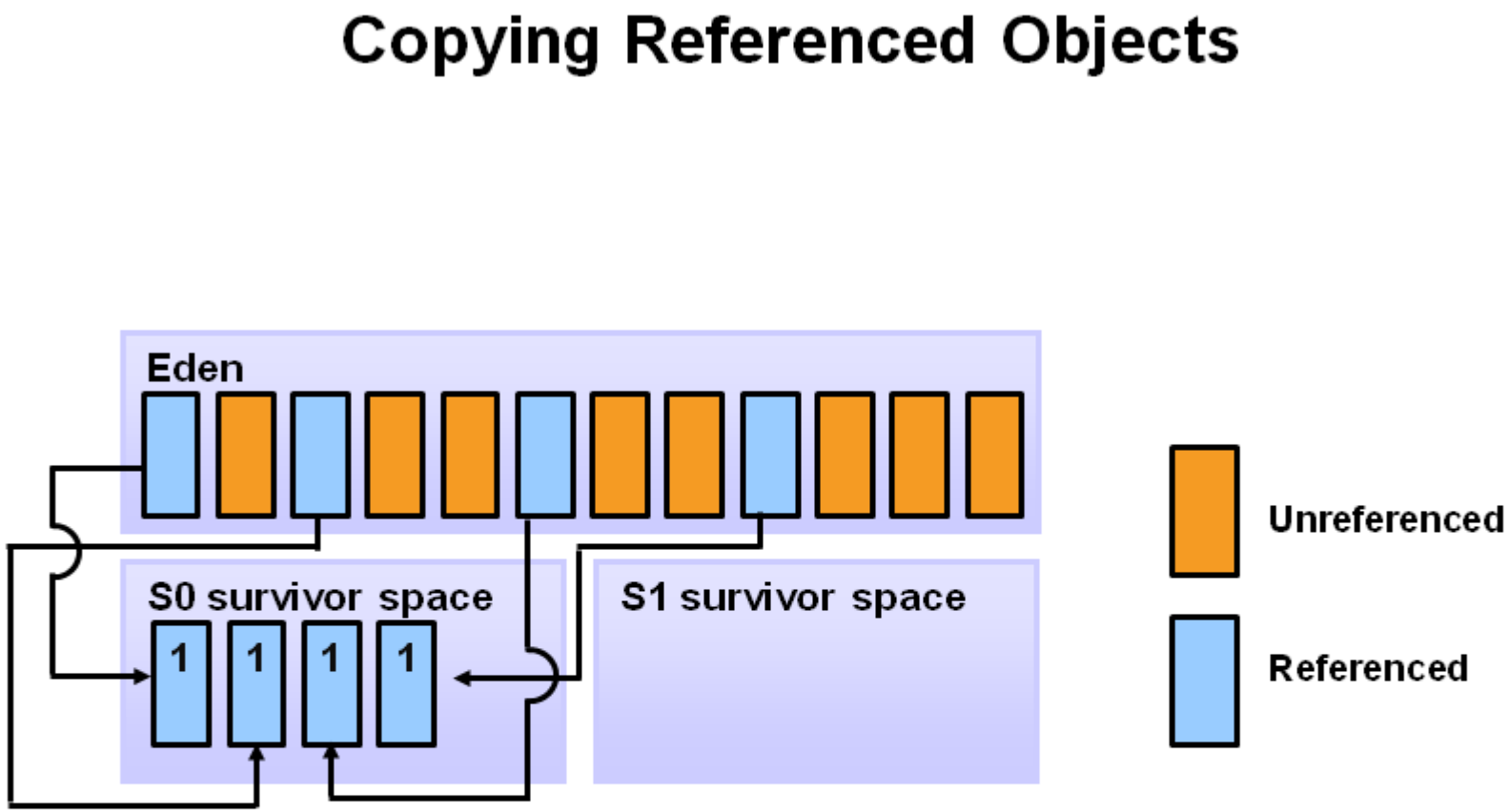
Generační GC



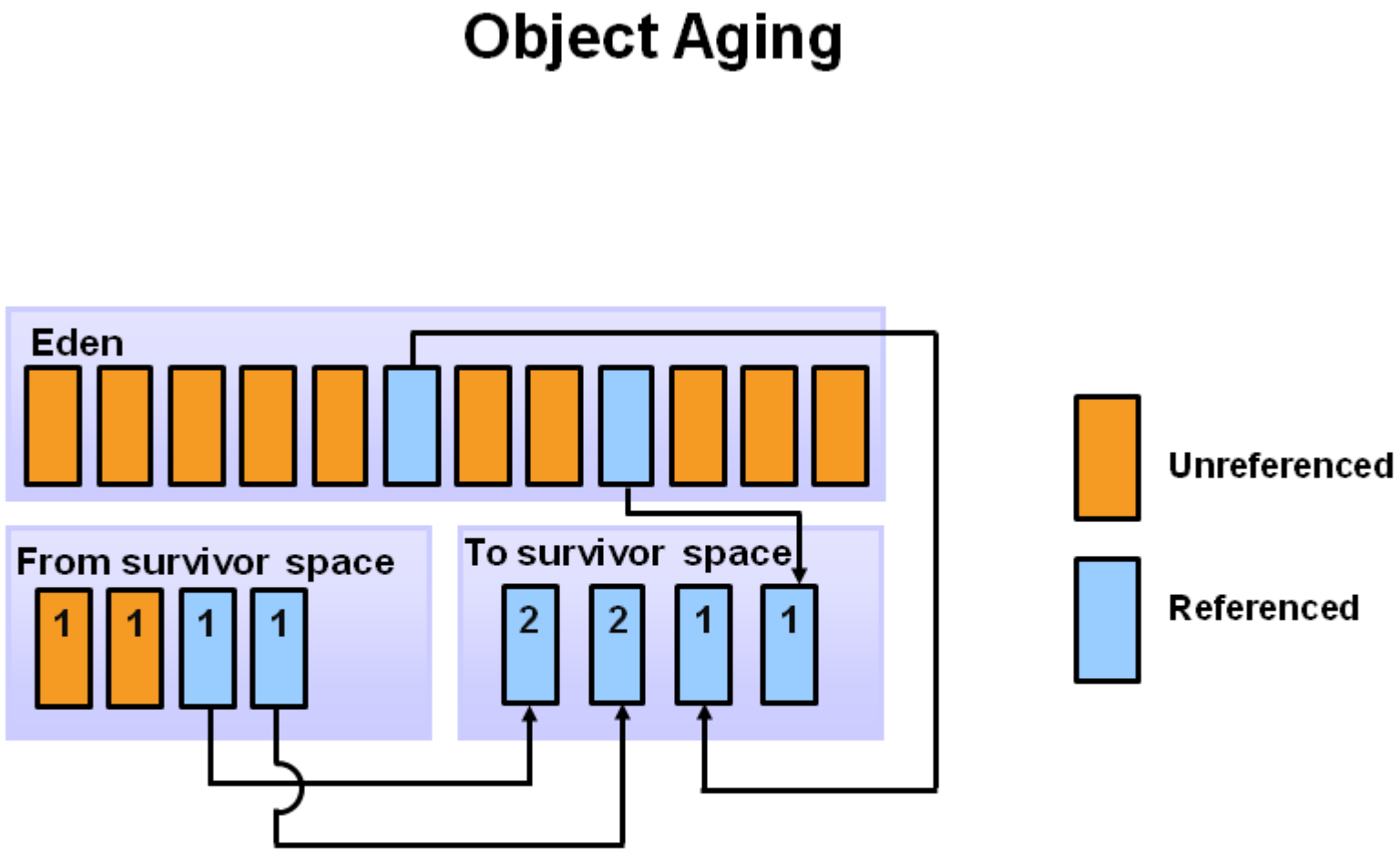
Generační GC



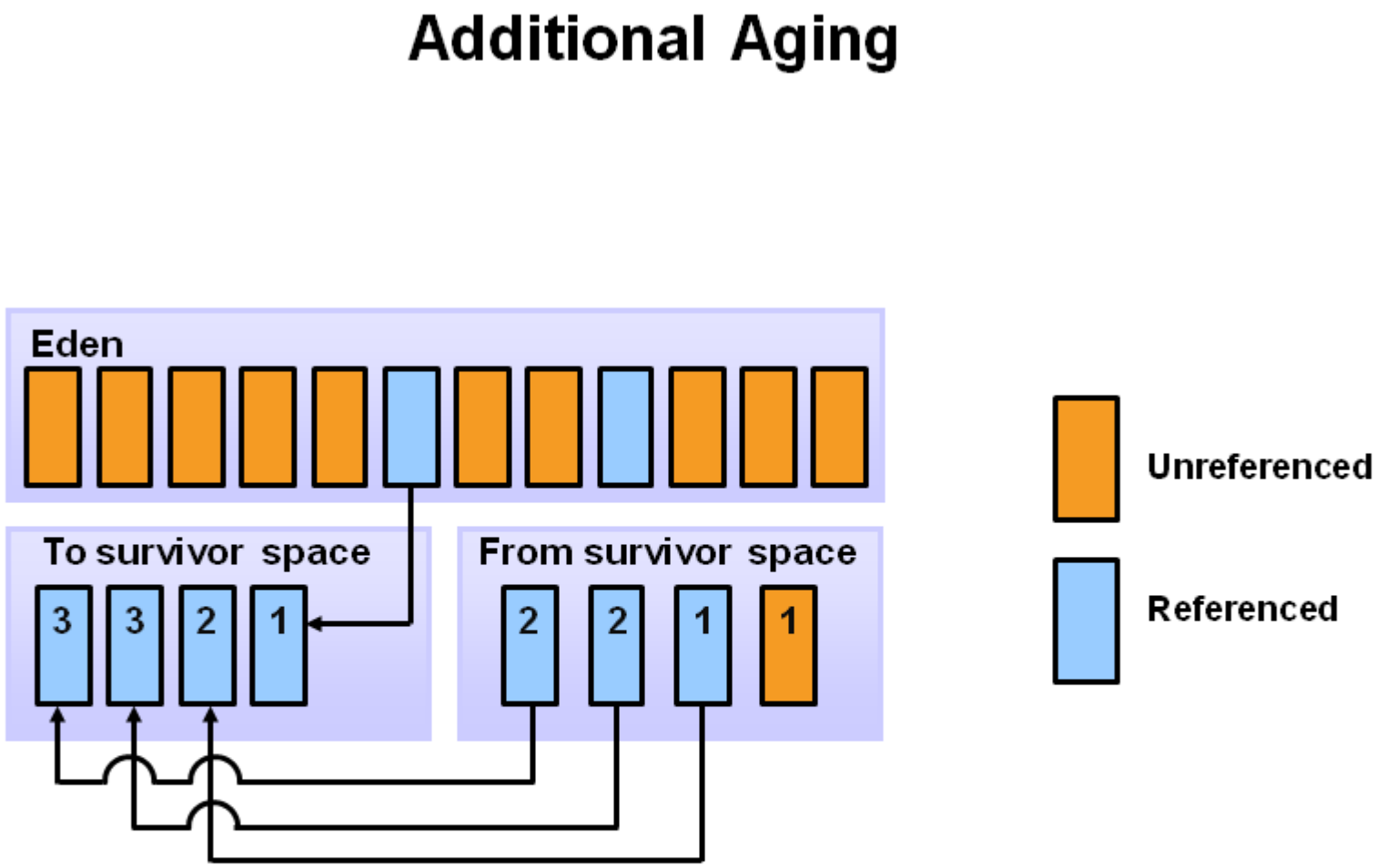
Generační GC



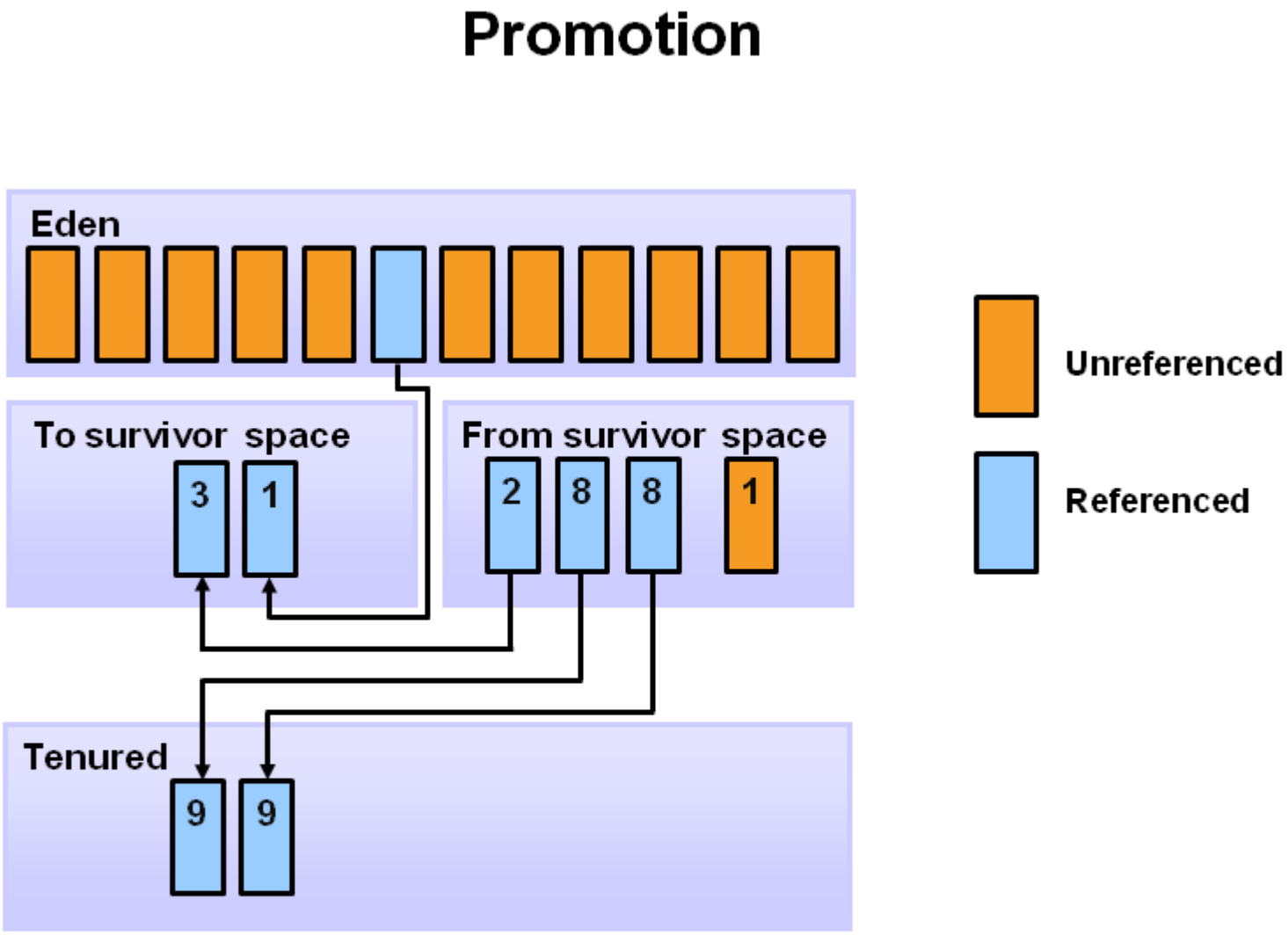
Generační GC



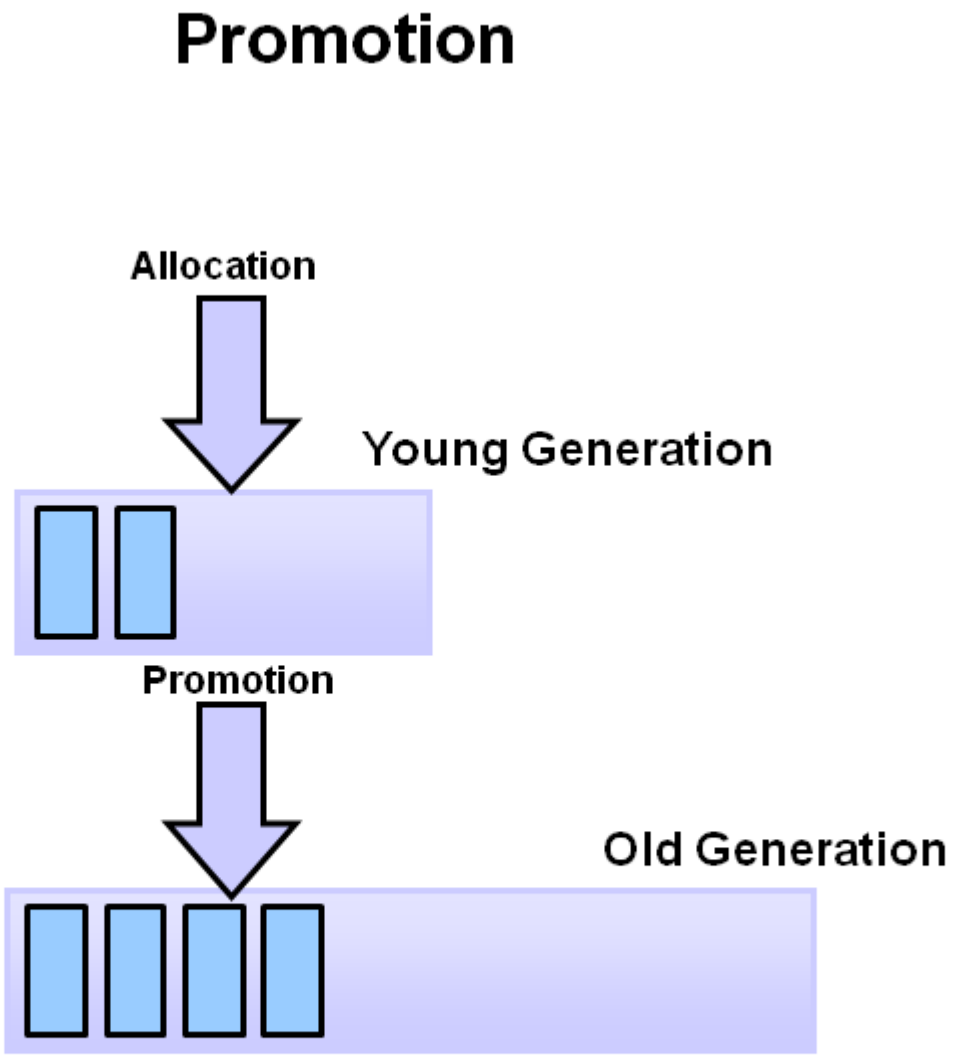
Generační GC



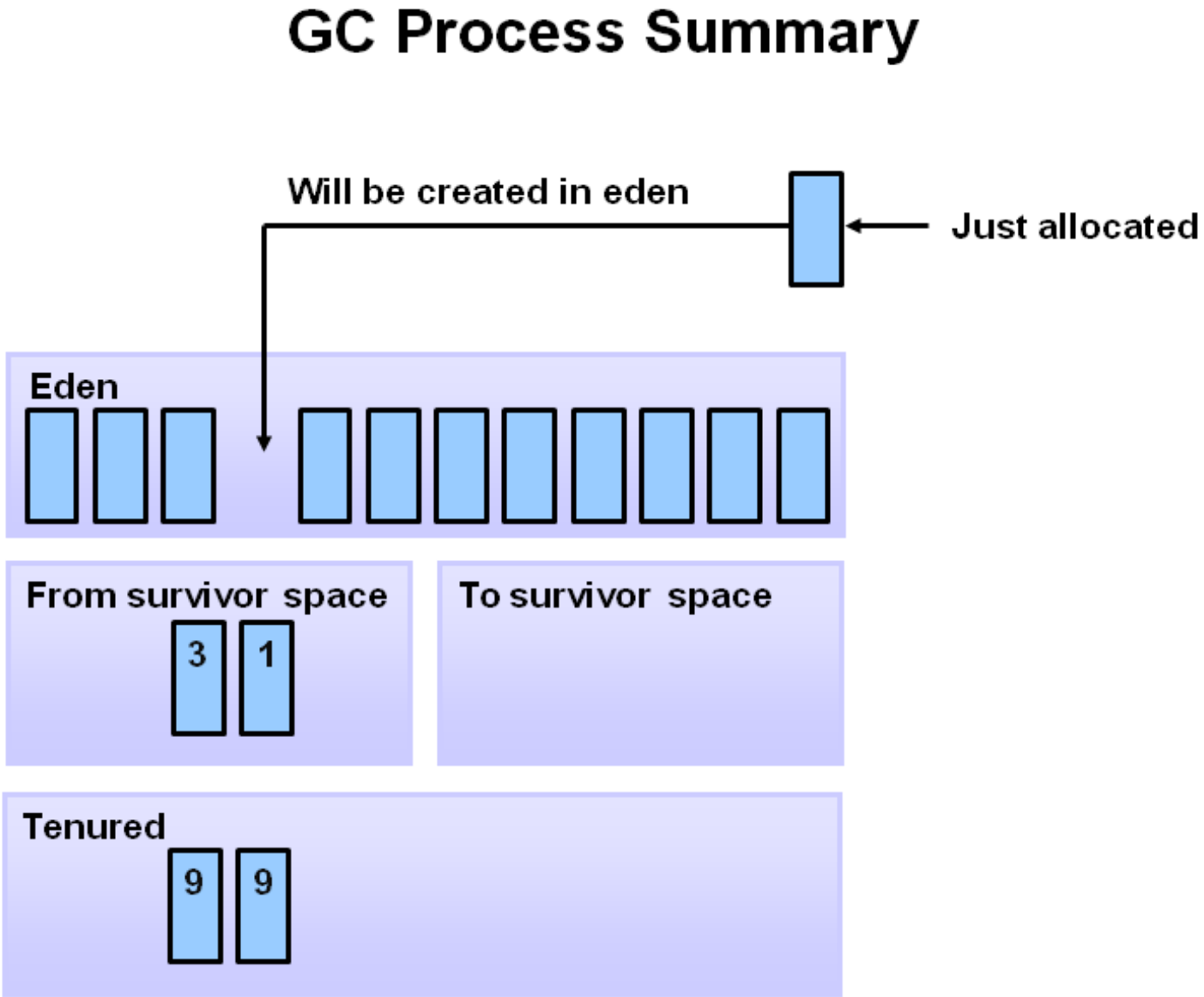
Generační GC



Generační GC



Generační GC



JVM Garbage Collectors

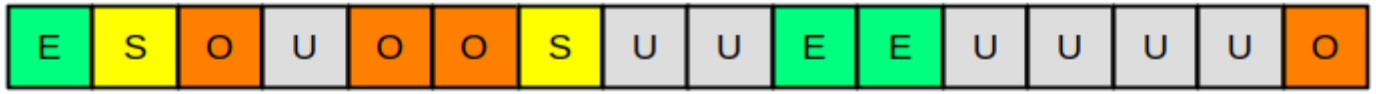
- Paralelní vs. Sériový běh GC
- Concurrent vs. Stop-the-world

Concurrent Mark Sweep (CMS)

- nízká latence, fragmentace

Garbage First (G1)

- Od Java 7 (od Java 9 default, od java 10 paralelní Full GC)
- Paralelní, konkurentní, nízká latence
- Nespojitý prostor



JVM Garbage Collectors

Aktivita GC

- Monitor nebo profiler
- Výpis na příkazovou řádku

`-verbose:gc`

```
[GC (Allocation Failure)  65536K->440K(251392K), 0.0007151 secs]
[GC (Allocation Failure)  65976K->344K(251392K), 0.0012630 secs]
[GC (Allocation Failure)  65880K->360K(251392K), 0.0007376 secs]
[GC (Allocation Failure)  65896K->344K(316928K), 0.0007838 secs]
[GC (Allocation Failure)  131416K->344K(316928K), 0.0011010 secs]
[GC (Allocation Failure)  131416K->376K(437760K), 0.0005855 secs]
[GC (Allocation Failure)  262520K->320K(437760K), 0.0017900 secs]
[GC (Allocation Failure)  262464K->320K(699392K), 0.0004259 secs]
```

Escape Analysis

- Důležitá technika pro JIT
- Ovlivňuje (ne)alokaci objektů na heapu
- Odstraňuje zbytečné zámky a synchronizace paměti
- Stav objektů může být
  - `NoEscape` – objekt není viditelný mimo aktuální metodu a vlákno
  - `ArgEscape` – objekt je viditelný pouze jako argument metody
  - `GlobalEscape` – objekt uniká z metody (návratový argument, statická proměnná, apod.)
- Cílem je udržet objekty nejlépe jako `NoEscape` aby mohli být optimalizovány

Escape Analysis

Zapnutí EA

```
-XX:+DoEscapeAnalysis
```

Vypnutí EA

```
-XX:-DoEscapeAnalysis
```

Defaultně aktivováno pro server HotSpot JVM, zřejmě i 64bit Java 8 a dále

EscapeAnalysis.java

- Bez analýzy – velká aktivita GC, runtime 13s, ~135MB (~176 in Java 11)
- S analýzou – žádná aktivita GC, runtime 11s, ~12MB (~25 in Java 11)
- Lokální proměnné jsou vytvářeny na „stacku“

EscapeAnalysis1.java

- Bez analýzy – velká aktivita GC, runtime 14s, ~830MB (~380 in Java 11)
- S analýzou – žádná aktivita GC, runtime 4s, ~16MB (~30 in Java 11)
- Výpočetní kód metody je v cyklu „inlineován“
- EA identifikuje vytváření instance jako zbytečnou
- „Vylepšení“ uložením objektů – EA nezabere, stvořen „memory leak“
- Co kdybychom neukládali výsledek do proměnné `computation`?

Reference

- http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
- https://www.dynatrace.com/news/blog/understanding-g1-garbage-collector-java-9/
- https://www.guru99.com/java-stack-heap.html

Příště

úterý a pak písemná **zkouška**