Životní cyklus a likvidace objektů

Table of Contents

Co je životní cyklus? . . . 1
Závislosti mezi objekty . . . 1
Vyhnout se zbytečným objektům . . . 1
Typická neefektivita . . . 1
Obecně . . . 2
Metody místo `new` . . . 2
Internalizace . . . 2
Předkompilace u regex . . . 2
Nepoužívejme `finalize()` . . . 3

Co je životní cyklus?

- Doba mezi vytvořením (new), používáním, nepoužíváním a zánikem objektu (instance).
- Začíná vytvořením pomocí `new`.
- Faktické používání objektu končí ve chvíli, kdy na něj ztratíme odkaz — např. když odkaz je lokální proměnná metody, kterou opustíme.
- Fyzická existenci končí buďto koncem programu nebo zrušením objektu pomocí garbage collectoru, "uklízeče smetí".
- Chování je přibližně stejné jako v Pythonu, ale zcela jiné než v C++ či Rust.

Závislosti mezi objekty

- správě závislostí mezi objekty, tzn. jeden objekt se své činnosti potřebuje další objekt nebo objekty, se věnuje speciální sekce Dependency Injection.

Vyhnout se zbytečným objektům

- *Avoid unnecessary objects*
- Vytváření objektů je "drahá" operace - stojí výpočetní čas i paměť.
- Objekt se musí v řadě případů též dealokovat, zlikvidovat, což stojí další čas.

Typická neefektivita

V řadě případů není třeba objekt vytvářet, ukážeme si typické situace:

- řetězce: každé zřetězení `s = s + t` znamená vytvoření nového objektu!

- úplně stejně `s += t` vede k vytvoření nového objektu
- u řetězců řešitelné modifikovatelnými řetězci (`StringBuilder`)

Obecně

- `new` namísto zavolání (statické) tovární metody rovněž vždy vytvoří objekt
- řadu složitějších objektů lze vytvořit jen jednou a znovupoužívat
- pooling (skladiště) objektů

Metody místo `new`

- Raději `Boolean.valueOf("true")` namísto `new Boolean("true")`, neboť to vytvoří pokaždé (obsahově stejný) objekt `Boolean`
- Obdobně pro další typy - navíc statické metody mohou obecně vrátit i `null`, když se nezdaří.
- U řetezců se mohutně využívá *internalizace* (neplést s internacionalizací), tzn. uložení do poolu v rámci běžící JVM.
- Každý řetězec zadaný jako literál (do uvozovek, třeba `"abcde"`) je internalizován a opakované použití odkazuje na tutéž instanci a nezabírá další paměť.

Internalizace

- Internalizaci lze vyvolat i pomocí metody `s.intern()` - u dlouhých řetězců by sdílení mohlo mít smysl!
- Je to proto, že řetězcové literály - nebo obecněji řetězce, které jsou hodnotami konstantních výrazů (§15.28) - jsou "internalizovány" tak, aby sdílely jedinečné instance pomocí metody String.intern (§12.5).
- blíže viz *Java Language Specification*, 3.10.5. String literals

Předkompilace u regex

- Regulární výrazy jsou silnou a častou používanou technikou, jak nalézat nebo ověřovat vzory v řetězcích.
- Jsou použitelné ve všech běžných jazycích vč. Javy.
- Lze je buďto:
  - rychle napsat, ale pomalu používat: `"řetězec".matches("regex")`
  - zdlouhavěji zapsat a rychle - i opakovaně - vykonávat

```
Pattern pattern = Pattern.compile("regex");
// i opakovaně, bleskově provedeno (násobně rychleji):
String s = ...
```

```
if(pattern.matcher(s).matches()) ...
```

Nepoužívejme `finalize()`

- Finalizér, tj. metoda `finalize()` může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup — sestávající obvykle z uvolnění systémových zdrojů - síťové sokety, spojení na databázi atd.
- V Javě nicméně *není zaručeno*, že se finalizér skutečně zavolá — JVM jej nemusí volat, pokud nepotřebuje fyzicky uvolnit paměť.
- I kdyby finalizér zavolán byl, zůstává problém s určením okamžiku, *kdy* je volán a v jakém pořadí jsou finalizéry na mrtvých objektech volány.
- Celkově tedy na `finalize()` nespoléhat a nepoužívat je - zdroje uvolňovat explicitním zavoláním vhodné metody.

Tento tip je javově-specifický, jde o to, jak a kdy pracuje garbage collector při likvidaci nepřístupných ("mrtvých") objektů.