Dobrá praxe - z *Josh Bloch: Effective Java*

Table of Contents

Dobré praktiky vytváření objektů

- Zvažte *statickou tovární metodu* místo konstruktorů.
- Uvažujte o "budovateli" (*builder*) namísto konstruktoru s mnoha parametry.
- Vynuťte *singleton* pomocí soukromých konstruktorů (nebo `enum`).
- Zabraňte přímé instanciaci pomocí soukromého konstruktoru.
- Upřednostněte vložení závislostí (*dependency injection*) před pevným propojením.
- Vyhněte se zbytečným objektům.
- Eliminujte zastaralé odkazy na objekty.
- Vyhněte se finalizátorům (metodám `finalize()`).

Tovární metoda namísto konstruktoru

- *Consider static factory method instead of constructors*

- Josh Bloch: "Java efektivně, Tip 1: Používejme statickou tovární ("výrobní") metodu namísto přímého volání konstruktoru (evt. s parametry)"
- Takových metod může být *více a mohou se jmenovat různě* (výstižněji) — viz `Person createMale()`
- *Nemusí vrátit* objekt za všech okolností, při všech možných voláních s různými parametry — někdy mohou na rozdíl od konstruktoru vrátit `null`
- Metoda může vrátit již *existující instanci* — to je klíčová výhoda, např. u singletonu nebo u skladiště (poolu) objektů — skladiště objektů šetří čas a výkon, singleton je bezpečný

Tovární metoda: variabilita vracených typů

- Metoda nemusí vrátit objekt přesně stejného typu, jako je deklarován, může vrátit i objekt podtřídy, potomka - např. metoda `static Person createEmployee(⋯)` vrátí objekt třídy `Employee`, jež je podtřídou `Person`
- Dokonce se skutečný vrácený typ může lišit dle předaných parametrů: `static Person createEmployee(boolean manager)` může dle vrátit `Manager` nebo `Employee`
- Tovární metoda někdy může vracet objekt třídy neznámé v době překladu, tzn. objekt pomocí reflexe dynamicky zavede — tato technika umožňuje modularitu a doplňování kódu a funkcionality dokonce za běhu

Jména továrních metod

- U statických továrních metod se držte běžného očekávaného pojmenování
- `newPerson`, `createPerson`, `Person.create`,
- pro přístup k poolu hotových objektů nebo singletonu např. `Person.getInstance()`
- logicky u *get* nikdo intuitivně nečeká, že se určitě bude tvořit nový objekt.

Namísto složitého konstruktoru použijme vzor "Výrobce"

- *Consider a builder instead of constructor with many parameters*
- Někdy máme složitější objekty s mnoha atributy
- Konstrukce pak obnáší volat konstruktor s mnoha parametry
- Příklad `new Person("Jan Novák", true, 22000)`, kde `true` značí, že jde o muže, a `22000` je plat.

Pojmenované parametry?

- V Javě jsou parametry poziční a nepojmenované (na rozdíl od novějšího Kotlinu např.), je pak problém vidět, co kde nastavujeme
- zejména když jsou parametry stejného typu — `new Line(0.0, 0.0, 1.0, 2.0)` znamená přesně co?

- Pro tyto případy se namísto složitého konstruktoru hodí tzv. builder ("budovatel").
- V moderních prostředích (IDEA, ale i VS Code s pluginy) vidíme, jaký parametr je na které pozici v závorce.

Příklad "budovatel"

```
public class PersonBuilder {
    private String name;
    //... male, salary
    public void setName(String name) {...}
    public void setGender(boolean male) {...}
    public void setSalary(double salary) {...}
    public Person getPerson() {
        // it can perform necessary checks
        // and refuse to create unless fulfilled
        return new Person(name, male, salary);
    }
}
```

Použití "budovatele"

```
PersonBuilder builder = new PersonBuilder();
builder.setName("Jan Novák");
builder.setGender(true);
builder.setSalary(22000.0);
// person can immediately be used
Person person = builder.getPerson();
```

Protipříklad - nepoužití "budovatele"

Nemáme budovatele, máme jen cílovou třídu objektů.

```
public class Person {
    private String name;
    //... male, salary
    public void setName(String name) {
        // set name here
    }
    public void setGender(boolean male) {...}
    public void setSalary(double salary) {...}
    public Person() {}
}
```

Bez použití "budovatele"

Je to špatně, objekt může bez kontroly zůstat neúplný.

```
Person person = new Person();
// now the person is NOT complete
// it is dangerous to use, cannot be used
person.setName("Jan Novák");
person.setGender(true);
person.setSalary(22000.0);
// now it is OK
```

Privátní konstruktory

- Ve výše uvedených tipech s tovární metodou i výrobcem jsme zamlčeli podstatnou věc:
- uvedené mechanismy vytvoření objektu se daly obejít jeho přímou konstrukcí `new Person(⋯)`.
- Aby toto nebylo možné, můžeme všechny konstruktory označit jako `private`
- Pak zvenčí nelze instanci vytvořit pomocí `new Person(⋯)`, ale musí např. být statická tovární metoda `Person.newInstance()` nebo budovatel `Builder builder = Person.builder()`.

Privátní nebo chráněné konstruktory?

- Rozdíl mezi `private` a `protected`
- Kdybychom konstruktory označili jako `private`, ale některé přesto ponechali `protected`, dovolíme tím *dědění* třídy, např. z `Person` můžeme podědit do `Employee`.
- Kdyby úplně všechny byly `private`, máme smůlu a z `Person` nikdy nic nepodědíme.
- Je to proto, že každá podtřída musí mít konstruktor, který jako první příkaz volá konstruktor předka — a ten kdyby byl soukromý, nebylo by možné.

Singletony

- Jsou klasickým návrhovým vzorem popsaným jinde.
- V souvislosti s konstrukcí platí, že objekt singletonu se konstruuje buďto předem nebo až je potřeba
- Nesmí být zkonstruovatelný jinak, např. přímo přes `new` (konstruktorem)
- *Vynucení singletonu pomocí privátních konstruktorů (nebo enum)* nebo *Prevent instantiation by private constructor* — když jsou konstruktory privátní, zvenčí je nelze volat a klient musí využít singleton, třeba `MyClass.getInstance()` nebo tovární metodu.

Příklad singletonu

```
public class DataManager {
    private static DataManager manager;
    private DataManager() {
        //... initialize data manager
    }
    public static DataManager getInstance() {
        if(manager == null) manager = new DataManager();
        return manager;
    }
}
```

Vyhněte se zbytečným objektům

- Špatně je toto, s každým průchodem je alokován nový řetězec.
- Nepomohlo by ani `s += String.valueOf(i) + " ";`
- Řešitelné pomocí `StringBuilder`, tam realokace nebude.

```
String s = "";
for(int i = 0; i < 100; i++) {
    // each time a new string is created
    // ... and the old forgotten
    s = s + String.valueOf(i) + " ";
}
```

Odstranění zastaralých odkazů na objekty

statické objekty

se dealokují až zcela na konci běhu programu — patří totiž celé třídě a ta se na rozdíl od objektu "nezapomíná"

vyrovnávací paměti

tím, že objekty se v cache pamatují a je to očekávané chování, musíme rozmyslet, kdy už je potřebovat nebudeme a odkazy smazat, nastavit `null`

datové struktury

například odkazy z pole jsou pořád "živé", dokud je "živé" celé pole

zabalení/vybalení z obálky

typicky čísla `new Integer(42)` vytvoří nový objekt

Vyhněte se finalizátorům a čističům

- Finalizér, tzn. metodu `finalize()`, mají všechny objekty,
- může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup při zániku objektu - 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 obsazenou již nepoužívaným objektem.
- 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.