7V2

## **Object pool**

Object pool je návrhový vzor, který používáme v situacích, kdy je vytvoření nové instance velmi náročné. Dovoluje vytvořit maximálně námi nastavené množství objektů, které dále využíváme. Když nějaký prvek z poolu vezmeme, není dostupný dokud ho znovu nevrátíme.

V podstatě taková banka na objekty.

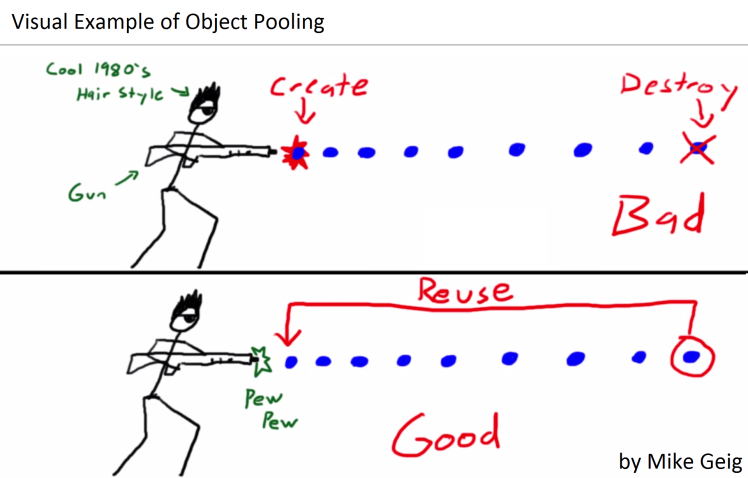
Pracujeme-li na mnoha místech se třídou, která je složitá na vytváření, její opakovaná konstrukce nám zpomaluje celý program. Návrhový vzor *object fond* se sám stará o vytváření a předávání objektů zbytku programu s tím, že již vytvořené objekty nemaže, ale uchovává na další použití.

*Fond* použijeme v programech, kde chceme zamezit opětovnému vytváření instance, nebo chceme mít kontrolu nad tím, kolik objektů existuje. Můžeme počet existujících objektů kontrolovat, a to i když při psaní kódu nevíme, kolik objektů bude aplikace reálně potřebovat.

Pooly samotné jsou navržené podle vzoru Singleton. To znamená že existuje pouze jedna instance poolu se kterou pracujeme. V ní je také uloženo těch několik málo objektů, ke kterým přistupujeme. Všechny vypůjčené objekty musí být pak vráceny.

Object pool tak obsahuje metody na

* Získání instance
* Získání objektu z poolu
* Vrácení objektu do poolu



## **Singleton**

Někdy v programu potřebujeme sdílet jednu instanci mezi několika bloky, objekty atd., aniž bychom ji museli stále předávat v konstruktoru. Ukázkový příklad je databázové připojení, celý program pracuje s jedním připojením a bylo by nepraktické ho stále předávat. Nabízí se řešení, udělat třídu poskytující databázové API jako statickou. Může však nastat případ, kdy se nám hodí ji mít instanciovatelnou (např. někdy pracujeme s více připojeními) nebo používáme hotovou třídu, která statická není. Vložíme ji tedy do Singletonu.

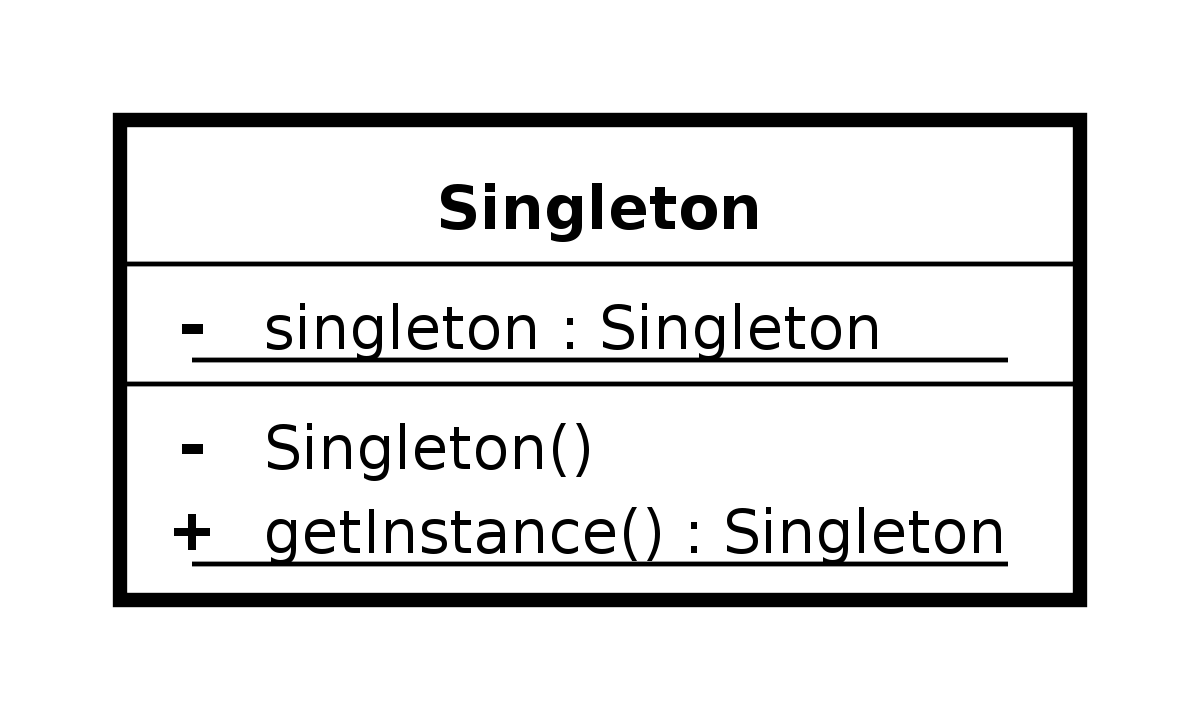
(Kdybychom měli ds API jako statickou, nemohli bychom vytvořit instance a tudíž by nemohlo existovat více připojení)

Vzor je tvořen třídou, která se stará o to, aby její instance existovala jen jednou.

Singleton znamená jedináček, tedy instance, která nemá žádné sourozence.

Někdy s jeho pomocí můžeme optimalizovat výkon programu. Není thread safe a v případě, že bychom pracovali s více vlákny, je třeba dát inicializaci instance do locku.

Slovo *globální* může být poněkud kontroverzní a Singleton je kvůli tomu někdy označován jako anti-pattern, tedy špatný vzor. Ačkoli Singleton by měl každý programátor znát, určitě to **není ideální vzor pro předávání závislostí v aplikaci**.



## **Factory**

<https://dev.to/gary_woodfine/how-to-use-factory-method-design-pattern-in-c-3ia3>

Factory používáme, když máme několik tříd, implementujících stejný interface, nebo ze kterých můžeme udělat instance a chceme nechat rozhodnout program, která z těchto tříd bude nejlepší. Factory metodu také používáme když chceme vytvářet instance z těchto tříd na základě určité business logiky nebo jiných podmínek.

Návratový typ nemusí být u factory specifikován přesně na typ objektu, který vytváříme. Můžeme vracet některou z rodičovských tříd nebo i rozhraní. Na každý z těchto případů se podíváme blíže.

Návrhový vzor **Factory method** využívá metody volající konstruktor. Má poměrně mnoho různých podob, někdy může být použito dědění a většinou se píše proti rozhraní. My si zde ukážeme úplně nejjednodušší implementaci. Klíčové je oddělení konstrukce. V aplikacích se nám občas stává, že potřebujeme **vytvořit instanci nějaké třídy a tu dodatečně inicializovat**. Dobrým praktickým příkladem jsou formulářové komponenty, u kterých nestačí pouze instanci vytvořit, ale musíme ji také nastavit spoustu dalších prvků.

Dále vás napadne oddělit tento kód do metody. Gratuluji, právě jste vynalezli factory. (samozřejmě má vzor nějaké další konvence)

Faktorka může také uchovávat proměnné, které potřebujeme k vytváření instancí. Tyto proměnné potom nemusí prostupovat celou aplikací. Další výhodoní instance do jiné třídy, čímž se **původní třída neznečistí konstrukčním kódem**.

Návratová hodnota **nemusí** být u *Faktory* stejná, jako je typ vytvářené instance. Klidně můžeme vytvářet při každém volání instanci jiné typ třídy a vracet pouze rozhraní, které všechny třídy implementují. Jako příklad uvedu grafický program, který vykresluje různé obrazce na obrazovku. Řekněme tedy, že máme čtverec, kruh a trojúhelník, které všechny implementují rozhraní IVykreslitelny. Data získáváme z řetězce (např. parsujeme nějaký textový soubor). Podle dodaných dat se rozhodneme, který typ tvaru vytvořit a vrátíme jej pouze jako rozhraní. Samotný program neví, co je to za obrazec. Pouze ví, že jej lze vykreslit.ších vlastností (rozměry, titulek, pozici, barvu...). Pokud někde v aplikaci vytváříte 20 podobných tlačítek a vytvoření takového tlačítka zabírá 10 řádků, nutně