Vzory řešící počet instancí

- Společné vlastnosti
- Knihovní třída (Library class, Utility)
- Jediináček (Singleton)
- Výčtový typ (Enumeration)
- Originál (Original)
- Fond (Pool)
- Muší váha (Flyweight)

Společné vlastnosti vzorů řešících problematiku počtu instancí

020

Motivace

- **Zabezpečit dodržení zadaných pravidel pro vznik a případný zánik instancí**
- **Aby mohla třída za něco ručit, musí se o vše postarat sama a nenechat si do toho mluvit**
- **Třída poskytuje globální přístupové body k vytvořeným instancím – tovární metody**
  - Rozhodnou zda vytvořit instanci či vrátit některou starší
  - Vyberou třídu, jejíž instance se vytvoří
  - Mohou před vlastním vytvořením instance provést nějaké pomocné akce

Soukromý konstruktor

- **Potřebují mít „v rukou“ vytváření instancí, takže nesmí nikomu umožnit, aby je vytvářel po svém**
- **Konstruktor vytvoří při každém zavolání novou instancí => je třeba jej znepřístupnit**
- **Definují konstruktor jako soukromý, a zveřejní místo něj tovární metodu**
  - Nikdo nemá šanci vytvořit instanci bez vědomí mateřské třídy
  - Třída může rozhodovat, zda se vytvoří nová instance, nebo zda se použije existující
  - Existence konstruktoru zamezí překladači vytvořit implicitní => je třeba jej vytvořit, i když žádný nechceme

Problémy se serializovatelností

- Načtení objektu z proudu obchází konstruktor
- Je třeba zabezpečit, aby i při načtení z proudu udržela třída kontrolu nad tvorbou svých instancí
- Třída musí rozhodnout, zda místo načteného objektu vrátí některý z existujících, anebo opravdu vytvoří nový
- Řešení závisí na realizaci
  - Knihovny proudů
  - Procesu serializace

Knihovní třída (Library class, Utility) 021

- *Knihovní třída* slouží jako obálka pro soubor statických metod. Protože k tomu nepotřebuje vytvářet instance, je vhodné jejich vytváření znemožnit.

Charakteristika

- **Slouží jako kontejner na metody, které nepotřebují svoji instanci**
  - Matematické funkce
  - Pomocné funkce pro rodinu tříd
- **Všechny metody jsou statické**
- **Nepotřebují instanci => neměly by ji ani umožnit vytvořit**
  - Definují soukromý konstruktor, aby místo něj nemohl překladač definovat vlastní verzi

Jedináček (Singleton) 022

- ***Jedináček*** **specifikuje, jak vytvořit třídu,**
  která bude mít nejvýše jednu instanci.
  Tato instance přitom nemusí být vlastní instancí dané třídy.

Charakteristika

- **Zabezpečuje vytvoření nejvýše jedné instance**
  - Tovární metoda vrací pokaždé odkaz na tutéž instanci jedináčka
- **Většinou se vytváří instance vždy (tj. právě jedna), ale je-li její vytvoření drahé, lez použít odloženou inicializaci (je ale náročnější)**
- **Tovární metoda může vracet i instance potomků**
  - Je-li to účelné, může třída rozhodnout, zda nechá vytvořit instanci svoji či některého ze svých potomků
  - Tovární metoda pak bude vracet odkaz na tu instanci, která byla při inicializaci vytvořena
- **Příklady: Schránka (clipboard), Plátno**

Realizace

- Odkaz na jedináčka je uložen ve statickém atributu
- Není-li požadována odložená inicializace,
  lze atribut inicializovat již v deklaraci
  (nejlepší řešení)
- Je třeba zabezpečit, aby všechny objekty,
  které tato inicializace používá, byly již validní
- Odložená inicializace přináší problémy v programech,
  které používají vlákna či podprocesy
- Je třeba ošetřit neklonovatelnost
  a případnou serializaci

Ukázka

```
public class Jedináček
{
  private static final Jedináček
                    JEDINÁČEK = new Jedináček();


  public static Jedináček getInstance() {
    return JEDINÁČEK;
  }
}
```

Výhody oproti statickému atributu

- **Jedináček: jediná instance třída je získávána voláním veřejné tovární metody**
- **Atribut: Odkaz na jedinou instanci je uložen ve veřejném statickém atributu**
- **Výhody atributu**
  - Trochu méně psaní
  - Většinou nepatrně větší rychlost
- **Výhody jedináčka**
  - Snazší dynamická kontrola nad poskytováním přístupu
  - Otevřenější vůči případným budoucím úpravám (např. náhrada jedináčka fondem)

Jednovláknová odložená inicializace

```
public static Jedináček getInstance()
{
    if (JEDINÁČEK == null) {
        JEDINÁČEK = new Jedináček();
    }
    return JEDINÁČEK;
}
```

Problémy při více vláknech

První vlákno	Druhé vlákno

`Jedináček.getInstance()`
`{`
`  if( JEDINÁČEK == null )`
`  {`	

	`Jedináček.getInstance()`
	`{`
	`  if( JEDINÁČEK == null )`
	`  {`
	`  JEDINÁČEK = new Jedináček();`
	`  }`
	`  return JEDINÁČEK;`
	`}`

`  JEDINÁČEK = new Jedináček();`
`  }`
`  return JEDINÁČEK;`
`}`	

Možné řešení

```
private static volatile Jedináček JEDINÁČEK = null;

public static Jedináček getJedináček()
{
    if (JEDINÁČEK == null ) {
        synchronized( Jedináček.class ) {
            if (JEDINÁČEK == null ) {
                JEDINÁČEK = new Jedináček();
            }
        }
    }
    return JEDINÁČEK;
}
```

Výčtový typ (Enumeration) 023

- *Výčtové typy* slouží k definici skupin předem známých hodnot a k umožnění následné typové kontroly.
  Vedle klasických hodnotových výčtových typů umožňuje Java definovat i *Funkční výčtové typy*, u nichž se jednotlivé hodnoty liší reakcemi na zasílané zprávy.

Charakteristika

- **Předem pevně daný počet instancí s předem pevně danými hodnotami**
- **Ekvivalent enum v C++, ale s OO nadstavbou a důslednou typovou kontrolou**
- **Výhody**
  - Důsledná typová kontrola
  - V kontejnerech instancí lze využít známý maximální počet
- **Problémy**
  - Je třeba ošetřit serializovatelnost a klonovatelnost
  - Standardní knihovna Javy ji řeší

Implementace

- **Sada statických atributů odkazujících na instance své třídy**
- **Atributy jsou inicializovány v deklaraci**
- **Jazyk často nabízí prostředky na efektivní použití v přepínačích a cyklech s parametrem**
  - Umí vrátit vektor seřazených hodnot
  - Umí vrátit instanci se zadaným pořadím
  - Umí vrátit pořadí zadané instance ve výčtu

Syntaxe výčtových typů

- V definicích výčtových typů je klíčové slovo `class` nahrazeno slovem `enum`
- Za hlavičkou třídy následuje seznam hodnot daného typu
- Za čistým výčtem (jenom výčet bez metod) nemusí následovat středník; jakmile však výčet není osamocen, středník je nutný (raději psát vždy)
- Výčtový typ překladač přeloží stejně jako kdyby byl definován jako potomek třídy `java.lang.Enum`

```
public enum Období
{
    JARO, LÉTO, PODZIM, ZIMA;
}
```

„Chytřejší“ výčtové typy

- Často je výhodné, aby hodnoty výčtového typu vykazovaly jistou „dodatečnou inteligenci“
- Výčtové typy nemusí být pouze čistým výčtem hodnot, jsou to standardní typy
  - Mohou mít vlastní atributy a vlastnosti
  - Mohou mít vlastní metody
  - Mohou využívat konstruktoru s parametry
- Příklad: `Směr8`

Funkční výčtový typ

- **Jednotlivé hodnoty výčtového typu mohou být instancemi jeho podtříd**
- **Každá hodnota pak bude na danou zprávu (volání metody) reagovat jinak**
- **Oblíbené použití:**
  - Každá instance představuje operaci, takže se v závislosti na výběru instance provede příslušná operace
- **Příklady**
  - Výčtový typ operací (např. PLUS, MINUS, KRAT, DELENO)
  - Výčtový typ zdrojů (např. KLÁVESNICE, SOUBOR, SÍŤ)
  - Třída `Člověk` v doprovodných programech
  - Diamanty (obrázek)

Příklad funkčního výčtového typu

- Metody instancí musí překrývat metodu předka, protože jinak by nebyly zvenku vidět
- Metody mohou definovat i „funkční vlastnosti“ instancí – např. jak se daná instance-tvar nakreslí – viz Diamanty

```
public enum Operace {
    PLUS {
        public int proveď( int a, int b) {
            return a + b;
        }
    },
    MINUS {
        public int proveď( int a, int b) {
            return a - b;
        }
    };

    public abstract proveď( int a, int b );
}
//...
Operace op = PLUS;
int výsledek = op.proveď( x, y );
//...
```

Vrstvený výčtový typ

- Někdy potřebujeme, aby sada dostupných instancí závisela na jistých okolnostech
  - Příklad: `Směr360` – `Směr8` – `Směr4`
- Standardní výčtový typ v Javě nepodporuje dědičnost (je třeba se vrátit ke „klasice“)
  - Nelze vytvořit explicitního potomka třídy `Enum`
  - Není možné vytvořit potomka výčtového typu
- Potomek musí být speciálním případem předka
- Pozor na problémy s typy návratových hodnot

Originál (Original) 024

- Návrhový vzor *Originál* použijeme v situaci, kdy budeme dopředu vědět, že se v aplikaci bude používat pouze malý počet různých instancí, avšak tyto instance budou požadovány na řadě míst kódu.

Charakteristika

- **Hodnotový neměnný typ, který zabezpečuje, že od každé hodnoty bude existovat jen jedna instance**
- **Počet hodnot není předem omezen, vychází však často ze společné výchozí sady**
- **Nevýhody**
  - Při žádosti o instanci je třeba zjistit, zda již neexistuje
- **Výhody**
  - Zbytečně se nám nekumulují duplicitní instance
  - Není třeba používat `equals`
- **Příklad: `Barva` v knihovně Tvary**

Implementace

- Tovární metody specifikují parametry potřebné k určení požadované hodnoty
- Alternativně je možno přiřadit instanci název a instance pak vybírat z mapy jako u výčtových typů
- Pro zrychlení vyhledávání se přehled o existujících instancích se udržuje ve formě mapy `<Hodnota,Instance>` příp. `<Název,Instance>`
- Pokud instance málo vytváříme, ale často používáme – šetříme tak čas správce paměti
- Příklad: `rup.česky.tvary.Barva`

Fond (Pool) 025

- *Fond* využijeme ve chvíli, kdy potřebujeme z nějakého důvodu omezit počet vytvořených instancí a místo vytváření instancí nových dáme přednost „reinkarnaci“ (znovuoživení) instancí dříve použitých a v danou chvíli již nepoužívaných.

Charakteristika

- Používá se v situacích, kdy potřebujeme omezit počet existujících instancí nějaké třídy
- Příklad
  - Připojení k databázi
  - Objekty náročné na různé zdroje
- Vždy, když objekt potřebujeme, požádáme o něj fond, a po použití objekt zase do fondu vrátíme
- Umí-li si objekt sám zjistit, že jeho práce skončila, můžeme explicitní vracení objektu fondu vypustit

Možné přístupy

- **Pevný počet instancí s frontou požadavků**
  - Je předem dán maximální počet instancí
  - Objeví-li se víc požadavků, než je povolený počet instancí, nepokryté požadavky se řadí do fronty, kde čekají, až se některá z instancí uvolní
  - Typické použití: databázová připojení
- **Dynamické zvyšování počtu instancí**
  - Počet instancí se dynamicky zvyšuje podle potřeby
  - Když přijde požadavek a všechny instance jsou „rozebrané“, vytvoří se nová
  - Typické použití: instance, jejichž vytvoření je „drahé“ a vyplatí se je proto nerušit, ale místo toho znovu použít
  - Příklad: vlákna, animované objekty

Implementace – řešené problémy

Vyhledávání volných instancí

- Je-li instancí ve fondu málo (typicky do 5),
  lze vyhledat prázdnou instancí projitím pole instancí
- Je-li jich víc, dvě možnosti:
  - Ukládat přidělené do mapy,
    kde ji bude možno po uvolnění rychle vyhledat a přeřadit mezi volné
  - Ukládat instance do zřetězeného seznamu
    (přidělená se přesune z počátku na konec, uvolněná se přesune naopak na počátek)

Přidělování a vracení jednotlivých instancí

- Jak zabezpečit, aby objekt, který instanci vrátil,
  už nemohl danou instanci používat
  - Např. když si instanci půjčenou z fondu předává několik metod
    a jedna metoda instanci vrátí dřív, než ji jiná přestane používat
- Jak zjistit, kdo instanci zapomněl vrátit

Přidělování a vracení instancí

- **Použití vrácené či cizí instance**
  - Instance dostane po přidělení jednoznačné ID,
    které bude parametrem všech jejích metod a jehož prostřednictvím
    se bude oprávněný „vlastník“ instance vždy identifikovat
  - Po vrácení instance bude ID odebráno
    a nový vlastník dostane přidělen nové ID
- **Zapomenuté uvolnění**
  - Při přidělení instance se může vytvořit a zapamatovat výjimka,
    která jednoznačně identifikuje místo přidělení i žadatele –
    tu si bude instance pamatovat a na požádání uložené informace prozradí
- **Bezpečné řešení zdržuje,
  takže je v časově kritických případech lze modifikovat tak,
  že se bude kompletně spouštět pouze v režimu ladění**

Muší váha (Flyweight) 026

- Vzor označovaný jako *Muší váha* používáme ve chvíli, kdy řešení problému vyžaduje vytvoření značného množství objektů. Ukazuje, jak je možné toto množství snížit tím, že místo skupiny objektů s podobnými charakteristikami použijeme jeden sdílený objekt tak, že z něj vyjmeme část jeho stavu, která odlišuje jednotlivé zastupované objekty, a volané metody se tyto informace v případě potřeby dozvědí z vnějšího zdroje prostřednictvím svých parametrů.

Charakteristika

- **Řeší situace, které při standardním přístupu vyžadují vytvoření příliš velkého množství objektů**
- **Vzor řeší problém tak, že jeden objekt slouží jako zástupce několika „virtuálních“ objektů**
- **Příklady**
  - V textovém editoru není představován každý znak v dokumentu samostatným objektem, ale všechny stejné znaky zastupuje představitel daného znaku
  - V situacích, kde vystupuje malá množina různých objektů v řadě různých navzájem si velmi podobných mutacích (viz hra `rup.česky.vzory._13_muší_váha.diamanty.Diamanty`)

Implementace

- **Stav objektu je rozdělen do dvou částí**
  - Vnitřní stav, který nese klíčové informace o objektu a který je společný všem „virtuálním“ objektům zastupovaným daným objektem
  - Vnější stav, v němž se jednotlivé zastupované objekty liší, a který je proto předáván metodám v parametrech
- **Příklady**
  - Textový editor:
    - Vnitřním stavem objektu je zastupovaný znak a konkrétní formát,
    - Vnějším stavem je pozice znaku v dokumentu
  - Diamanty (pgm)
    - Vnitřním stavem je vzhled obrazce,
    - Vnějším stavem je jeho umístění na hrací desce