

# Přednášky z OOP

# Obsah

1. Úvodní informace o OOP .....	1
1.1. Základní informace .....	1
1.1.1. Výhody Javy .....	1
1.1.2. Vývojová prostředí obecně .....	3
1.1.3. Programy dříve a nyní .....	4
1.1.4. UML .....	7
1.1.5. Testování .....	8
1.2. Obecné pojmy z objektově orientovaného přístupu .....	8
1.2.1. Základní pilíře OOP .....	9
1.2.2. Třídy a objekty v interaktivním režimu BlueJ .....	11
1.2.3. Zasílání zpráv = volání metod .....	18
1.2.4. Zautomatizování interaktivního režimu BlueJ .....	21
2. Třída a její části .....	26
2.1. Třída .....	26
2.2. Konstruktor .....	26
2.2.1. Přetížené konstruktory .....	27
2.3. Atributy .....	29
2.3.1. Atributy versus vlastnosti .....	31
2.3.2. Atributy v BlueJ .....	32
2.3.3. Atributy, lokální proměnné, parametry – komplexní pohled .....	32
2.3.4. Pokračování příkladu s třídou Strom .....	35
2.4. Metody .....	35
2.4.1. Porovnávání objektů – úvod .....	37
2.4.2. Využití <code>this</code> u metod .....	37
2.4.3. Atributy a metody třídy .....	38
2.4.4. Odložená inicializace ( <i>lazy initialization</i> ) .....	40
2.4.5. Využití testů v BlueJ .....	40
2.5. Rozhraní versus implementace .....	45
2.5.1. Signatura versus kontrakt .....	46
2.6. Komentáře .....	48
2.6.1. Dokumentační komentáře .....	48
2.7. Kvalita kódu .....	56
2.8. Společný předek <code>Object</code> .....	59
2.8.1. Řetězce a metoda <code>toString()</code> .....	60
2.8.2. Metoda <code>getClass()</code> .....	62
3. Návrhové vzory a rozhraní .....	64
3.1. Jednoduché návrhové vzory .....	64
3.1.1. Knihovní třída ( <i>Utility class</i> ) .....	64
3.1.2. Statická tovární metoda ( <i>Static factory method</i> ) .....	66
3.1.3. Jedináček ( <i>Singleton</i> ) .....	67
3.1.4. Přepravka ( <i>Messenger</i> ) .....	68
3.1.5. Výčtový typ ( <i>Enum</i> ) .....	71
3.2. Rozhraní .....	73
3.2.1. Terminologie .....	74
3.2.2. Konstrukce <code>interface</code> .....	75
3.2.3. Implementace <code>interface</code> .....	76
3.2.4. Výhoda použití rozhraní .....	78
3.2.5. Značkovací rozhraní ( <i>marker interface, tagging interface</i> ) .....	79
3.2.6. Funkční rozhraní ( <i>functional interface</i> ) .....	79
3.2.7. Defaultní metody v rozhraní .....	79
3.2.8. Statické metody v rozhraní .....	82

3.3. Návrhové vzory – pokračování .....	84
3.3.1. Motivace .....	84
3.3.2. Posluchač ( <i>Listener</i> ) .....	85
3.3.3. Prostředník ( <i>Mediator</i> ) .....	87
3.3.4. Služebník ( <i>Servant</i> ) .....	88
3.4. Třída implementuje více rozhraní .....	93
3.4.1. Přetypování .....	95
4. Datové typy, balíky, JAR .....	97
4.1. Metody třídy Object – pokračování .....	97
4.1.1. Metoda <code>equals()</code> .....	97
4.1.2. Metoda <code>hashCode()</code> .....	98
4.2. Datové typy Javy .....	99
4.2.1. Primitivní datové typy .....	99
4.2.2. Objektové datové typy .....	100
4.2.3. Práce s řetězci .....	103
4.3. Praktické náležitosti Java programů .....	104
4.3.1. Hlavní třída .....	104
4.3.2. JAR soubory .....	108
4.3.3. Balíky .....	111
5. Dědičnost .....	116
5.1. Typy dědění .....	116
5.2. Dědičnost rozhraní .....	117
5.3. Skládání .....	119
5.4. Dědění tříd .....	123
5.4.1. Principy dědičnosti implementace .....	124
5.4.2. Realizace dědičnosti .....	126
5.4.3. Konstrukce třídy a spolupráce s nadřídu .....	132
5.4.4. Dědičnost a metody .....	138
5.4.5. Výhoda dědění .....	142
5.4.6. Konečné třídy .....	144
5.4.7. Závěrečné poznámky o nevhodnosti dědičnosti implementace .....	145
5.5. Abstraktní třída .....	145
5.5.1. Speciální případ abstraktní třídy .....	148
5.5.2. Konstrukce abstraktní třídy z potomků .....	148
6. Arrays, řazení, kolekce a genericita .....	150
6.1. Podpora práce s poli – třída <code>Arrays</code> .....	150
6.1.1. Možnosti třídy <code>Arrays</code> .....	150
6.2. Řazení objektů .....	151
6.2.1. Přirozené řazení ( <i>natural ordering</i> ) .....	152
6.2.2. Absolutní řazení ( <i>total ordering</i> ) .....	154
6.3. Kolekce a genericita – úvodní informace .....	156
6.4. Typové parametry a parametrizované typy .....	160
6.4.1. Použití žolíků – <i>unbounded wildcard</i> .....	162
6.4.2. Omezené využití žolíků – <i>bounded wildcard</i> .....	163
6.5. Rozhraní <code>Collection</code> .....	164
6.6. Rozhraní <code>List</code> .....	165
6.6.1. Implementace pomocí <code>ArrayList</code> .....	166
6.7. Zajištění algoritmů – třída <code>Collections</code> .....	170
6.8. Postupný průchod kolekcí .....	173
6.8.1. For-Each .....	173
6.8.2. Iterátory .....	174
6.9. Ochrana proti nekonzistenci dat .....	177
6.10. Výhodnost jednotlivých seznamů .....	177

7. Kolekce – množiny a mapy .....	179
7.1. Množiny – rozhraní Set .....	179
7.1.1. Práce s vlastní třídou v množině .....	181
7.1.2. Problémy objektů v hešovacích třídách .....	188
7.1.3. Použití Collections .....	194
7.1.4. Rozhraní SortedSet .....	194
7.1.5. Množinové operace a triky .....	195
7.2. Mapy – rozhraní Map .....	196
7.2.1. Třída TreeMap .....	199
7.3. Složitější datové struktury .....	201
8. UML .....	204
8.1. Základní informace .....	204
8.1.1. Různé diagramy pro různé fáze vývoje projektu .....	206
8.2. Společné části diagramů .....	207
8.2.1. Poznámka ( <i>note</i> ) .....	207
8.2.2. Stereotyp ( <i>stereotype</i> ) .....	207
8.2.3. Omezení ( <i>constraint</i> ) .....	208
8.3. Diagram případů užití .....	208
8.4. Diagram tříd .....	210
8.4.1. Způsoby zápisu třídy .....	211
8.4.2. Vazby tříd .....	213
8.4.3. Speciální případy vazeb .....	220
8.5. Příklad na různé vztahy mezi třídami .....	222
8.6. Jak připravit diagram tříd snadno a rychle .....	226
8.6.1. Zdrojové kódy .....	226
8.6.2. Nakreslení tříd, rozhraní a výčtových typů .....	228
8.6.3. Nakreslení nejsilnějších vazeb .....	229
8.6.4. Nakreslení vazeb agregací .....	230
8.6.5. Nakreslení vazeb závislostí .....	230
8.6.6. Přeskupení objektů .....	231
8.6.7. Doplnění multiplicit .....	232
9. Polymorfismus, interní datové typy .....	234
9.1. Polymorfismus .....	234
9.1.1. Využití polymorfismu pomocí abstraktní třídy .....	234
9.1.2. Využití polymorfismu pomocí rozhraní .....	239
9.1.3. Využití rozhraní různými datovými typy .....	240
9.2. Interní datové typy .....	243
9.2.1. Vnořené typy .....	245
9.2.2. Vnitřní třídy .....	245
9.2.3. Anonymní lokální .....	250
9.2.4. Pojmenované lokální .....	253
9.3. Metody s proměnným počtem parametrů .....	253
10. Lambda výrazy a spol. .....	255
10.1. Funkční rozhraní .....	255
10.1.1. Funkční rozhraní z Java Core API .....	256
10.2. Lambda výrazy .....	257
10.2.1. Přechod z anonymních vnitřních tříd na lambda výrazy .....	257
10.2.2. Syntaxe lambda výrazů .....	264
10.2.3. Lambda výrazy jako hodnoty funkčního rozhraní .....	265
10.2.4. Lambda výrazy v GUI .....	266
10.3. Reference na metody .....	267
10.3.1. Princip reference .....	267
10.3.2. Syntaxe zápisu referencí .....	269

10.4. Agregované operace ( <i>Aggregate Operations</i> ) .....	270
10.4.1. Vytvoření <i>streamu</i> .....	272
10.4.2. Průběžné operace ( <i>intermediate</i> ) .....	273
10.4.3. Koncové redukční agregované operace .....	274
10.4.4. Koncové agregované operace vracející kolekce .....	275
10.5. Třída <code>Optional</code> .....	278
10.5.1. Ukázka možného použití <code>Optional</code> .....	279
10.5.2. Ukázka „skrytí“ <code>Optional</code> ve třídě <code>Predmet</code> .....	281
11. Reflexe, invokace a anotace v Javě .....	285
11.1. Reflexe .....	285
11.2. Invokace .....	288
11.3. Anotace .....	289
11.3.1. Úvodní informace .....	289
11.3.2. Anotace z <code>java.lang</code> .....	290
11.3.3. Metaanotace .....	291
11.3.4. Využití anotací v programu pomocí reflexe .....	293
11.3.5. Praktický příklad využití anotací – JUnit testy .....	294
11.3.6. Vytváření vlastních anotací .....	296
11.3.7. Použití parametrů anotací .....	303
12. Návrhové vzory .....	309
12.1. Základní informace .....	309
12.2. Konstrukční vzory .....	311
12.2.1. Stavitel ( <i>Builder</i> ) .....	311
12.2.2. Abstraktní továrna ( <i>Abstract factory</i> ) .....	314
12.3. Strukturální vzory .....	318
12.3.1. Kompozit ( <i>Composite</i> ) .....	318
12.3.2. Dekorátor ( <i>Decorator</i> ) .....	320
12.3.3. Adaptér ( <i>Adapter</i> ) .....	324
12.3.4. Most ( <i>Bridge</i> ) .....	326
12.4. Vzory chování .....	329
12.4.1. Příkaz ( <i>Command</i> ) .....	329
12.4.2. Pozorovatel ( <i>Observer</i> ) .....	331
12.4.3. Prostředník ( <i>Mediator</i> ) .....	333
12.4.4. Stav ( <i>State</i> ) .....	335

# Kapitola 1. Úvodní informace o OOP

## 1.1. Základní informace

- první přednášky podle knih od Rudolfa Pecinovského: **Myslíme objektově v jazyku Java 5.0 (2. vydání)** a **OOP – Naučte se myslet a programovat objektově**
- metodika *design patterns first*
  - návrhové vzory – soubor doporučení, jak programovat efektivně a OOP čistě
  - na začátku se neprogramuje, ale vysvětlují se OOP principy, pojmy a zákonitosti
    - ◆ to je možné díky existenci moderních výukových pomůcek – BlueJ
  - pak architektura programu a nakonec programování
- pragmatický přístup – nemusíte znát hned všechno najednou
  - začíná s výukou klíčových dovedností; další dovednosti zařazuje až tehdy, když jsou pro řešení úloh potřebné
- některá témata budou později vysvětlena do detailu (kolekce, JUnit)
- přednášky předpokládají znalosti probírané v KIV/PPA1 – tyto znalosti se zde maximálně stručně shrnou
  - odkaz [PPA1-9] znamená učební text Herout, P.: **Počítače a programování 1**, str. 9
    - ◆ učební text je dostupný na Courseware OOP ve formě PDF

### 1.1.1. Výhody Javy

- výhod je mnoho, z našeho pohledu stačí:
  - přenositelná (platformně nezávislá, tj. Windows, Linux, MacOs – všude stejné)
  - objektově orientovaná
  - robustní (bezpečná) – brání se chybám programátora, resp. málo příležitostí k dělání chyb
  - univerzální – vhodná pro všechny typy aplikací
  - nejpoužívanější – např. *TIOBE Programming Community Index* – [www.tiobe.com](http://www.tiobe.com)
- viz též [PPA1-9]
- pojem platforma – spojení hardwaru (HW) a operačního systému (OS)
  - každá rodina procesorů (HW) má vlastní instrukční soubor
  - OS zakrývá z pohledu aplikace drobné odchylky HW – stejné platformy
  - na stejném HW mohou být různé OS (Windows, Linux) – různé platformy

- překládaný program musí být přeložen a odladěn pro danou platformu
- pojednat se o virtuální stroj – nadstavba nad platformou – vytváří ji „výrobce“ Javy
- jedná se o interpret mezi jazykem – .class u Java platformy
  - pro každou existující platformu (HW + OS) stačí připravit příslušný virtuální stroj
  - Java přeložený program (.class) zůstává stále stejný
  - inovace HW a/nebo SW doplněné inovací virtuálního stroje neovlivní chod původních Java programů
- Java platforma
- spojení virtuálního stroje + knihovny + verze Java Jazyka

### 1.1.1.1. Dílčí platformy Javy

- Java SE – *Standard Edition*
- desktopové aplikace
  - budeme používat
- Java EE – *Enterprise Edition*
- rozšířená verze SE
    - ♦ jazyk stejný jako u SE
  - velké distribuované aplikace
  - podpora vícevrstvé architektury
- Java ME – *Micro Edition*
- mobilní telefony a podobná zařízení limitovaná zejména pamětí
  - ořezaná verze SE
- Java FX – *effects*
- od JDK 1.8 standardní součást Java Core API
  - moderní způsob vytváření (nejen) GUI
    - ♦ vývoj tzv. RIA aplikací – *Rich Internet Applications* – multimedální uživatelské rozhraní
    - ♦ viz KIV/UUR

### 1.1.1.2. Vývojová a běhová prostředí Javy

- JRE – *Java Runtime Environment*
- virtuální stroj + knihovny
  - pouze spuštění hotových programů

- velikost do 100 MB
- JDK – *Java Development Kit* – nutné pro vývoj programů
  - JRE + překladač + další podpůrné prostředky
  - samostatně dokumentace
  - instalace s dokumentací asi 500 MB
  - popis instalace viz PPA1

## 1.1.2. Vývojová prostředí obecně

- sada nástrojů, které mají (co nejvíce) usnadnit vývoj aplikací
- další pojmy (synonymní)
  - IDE – *Integrated Development Environment*
  - RAD – *Rapid Application Development*

### 1.1.2.1. JDK – viz dříve

- nelze mluvit o IDE – řádkový překlad a spuštění

### 1.1.2.2. BlueJ

- navrženo pro **výuku začátků** OO programování
- [www.bluej.org](http://www.bluej.org)
- popis instalace verze 4.1.2 je na CourseWare
  - od března 2017 verze 4.0 (stabilní od verze 4.1.2, z října 2017)
    - ◆ výrazné změny oproti verzi 3.x
      - GUI přepsáno do Java FX
      - mnohem lepší JavaDoc
      - vylepšené editory apod.
- jednoduché
- speciální výukové funkce
  - koncepční pohled na projekt jako na UML diagram tříd – struktura tříd a vztahy mezi nimi
  - možnost interaktivní komunikace s vytvořenými objekty
  - dvojí pohled na třídu – implementace (zdrojový kód) a rozhraní (dokumentace)
- speciální praktické funkce

- testy integrované do samé podstaty BlueJ
  - integrované PMD pro kontrolu kvality kódu
- animace návodů: <http://vyuka.pecinovsky.cz/animace>

### 1.1.2.3. Eclipse

- profesionální IDE – vývoj pro mnoho jazyků (Java, C++, PHP, ...)
- vývoj dříve IBM, nyní [www.eclipse.org](http://www.eclipse.org)
- široce používané, značné množství pluginů
- popis instalace viz PPA1

### 1.1.2.4. NetBeans

- profesionální IDE – vývoj zejména Java programů, ale i dalších
- vývoj Sun / Oracle
- přímá alternativa k Eclipse

### 1.1.2.5. IntelliJ IDEA

- vysoce profesionální IDE – „*The Most Intelligent Java IDE*“
  - „Důraz na i ty nejmenší detaily je to, co dělá IntelliJ IDEA mimořádnou.“
- dříve pouze placené, nyní i free verze
- často nástupnické IDE po Eclipse / NetBeans

### 1.1.2.6. PMD

- nejedná se o vývojové prostředí, ale o nástroj pro zvyšování kvality zdrojového kódu (statický analyzátor)
- [pmd.sourceforge.net](http://pmd.sourceforge.net)
- lze použít samostatně, ale i zaintegrovat do BlueJ, Eclipse, ...
- konfigurovatelný nástroj, který z počátku obtěžuje, ale ve svém důsledku výrazně pomáhá

## 1.1.3. Programy dříve a nyní

tato část je převzata od R. Pecinovského

- existující řešení – 1
  - dříve: Řada běžných, často se vyskytujících úloh stále čekala na vyřešení
  - nyní: Většina běžných úloh je vyřešena a řešení jsou dostupná v komponentách či knihovnách
- existující řešení – 2

- dříve: Prvotní úlohou programátora bylo vymyslet, jak úkol vyřešit
- nyní: Prvotní úlohou programátora je zjistit, jestli už někde není problém vyřešen

## ■ spolupráce programů

- dříve: Programy pracovaly samostatně, navzájem příliš nespolupracovaly
- nyní: Nové programy jsou téměř vždy součástí rozsáhlých aplikací a rámců

## ■ algoritmizace

- dříve: Klíčovou úlohou programátora byl návrh algoritmů a základních datových struktur
- nyní: Důležitější než znalost algoritmů je znalost knihoven a aplikačních rámců, v nichž jsou potřebné algoritmy a datové struktury připraveny

Klíčovou úlohou je návrh architektury systému !

## ■ zadání problému

- dříve: Metodika vývoje programů počítala s pevným zadáním
- nyní: Zadání většiny vyvíjených projektů se v průběhu vývoje neustále mění

## ■ poptávka versus nabídka

- dříve: Zákazníci hledali firmu, která jejich projekt naprogramuje
- nyní: Programátorské firmy hledají zákazníky, kteří si u nich objednají tvorbu projektu

## ■ zákaznický přístup

- dříve: O výsledné podobě projektu rozhodovali analytici a programátoři
- nyní: O výsledné podobě projektu rozhoduje zákazník

## ■ požadavky na řešení

- dříve: Při vývoji programů se kladla váha především na jejich efektivitu (paměťová / časová)
- nyní: Při vývoji programů se klade váha především na jejich spravovatelnost a modifikovatelnost

## ■ požadavky na kvalifikaci programátora

- dříve: U programátorů byla oceňována jejich schopnost vyvíjet programy s malými HW požadavky
- nyní: U programátorů je oceňována jejich schopnost vyvíjet programy, které je možno rychle a levně přizpůsobovat neustále se měnícím požadavkům zákazníka

## ■ přístup k testování

- dříve: Testy se většinou navrhovaly po dokončení projektu či jeho části a spouštěly se na závěr před odevzdáním projektu (byl-li čas)
- nyní: Testy se většinou navrhují před začátkem vývoje každé části a spouští se v průběhu celého vývoje po každé drobné změně

## ■ způsob testování

- dříve: Testy navrhovali programátoři a ověřovali v nich, že program dělá to, co chtěl programátor na-programovat
- nyní: Testy se navrhují ve spolupráci se zákazníkem a ověřuje se v nich, že program dělá to, co po něm zákazník požadoval

## ■ nutnost testování

- dříve: Návrh testů byl interní záležitostí vývojového týmu
- nyní: Návrh testů se často stává součástí smlouvy o vývoji programu

## ■ Závěry ze změn:

- doba programování jako **umění** skončila, nastupuje programování jako **technologie**
- priority současného programování, které jsou podporované návrhovými vzory:
  - ◆ funkčnost
  - ◆ robustnost (testy)
  - ◆ znovupoužitelnost (rozhraní + srozumitelnost)
  - ◆ modifikovatelnost
    - srozumitelnost = dokumentace včetně UML, kvalitní zdrojový kód (PMD a spol.)
    - vstřícnost ke změnám
- přímo viditelné změny ve zdrojovém kódu OOP
  - ◆ velké množství velmi krátkých metod (i jednořádkových)
  - ◆ v mnoha třídách též úplná absence „algoritmů“ a s nimi příkazů cyklů, podmínek apod. – stačí víceméně pouze přiřazovací příkazy
    - většina metod z Java Core API má jeden až pět příkazů
    - metoda, která má více než 10 řádek pravděpodobně dělá více věcí najednou

### Varování

Struktura typů tříd se podle účelu tříd výrazně odlišuje – viz postupně dále.

## 1.1.3.1. Strukturovaný versus OO program

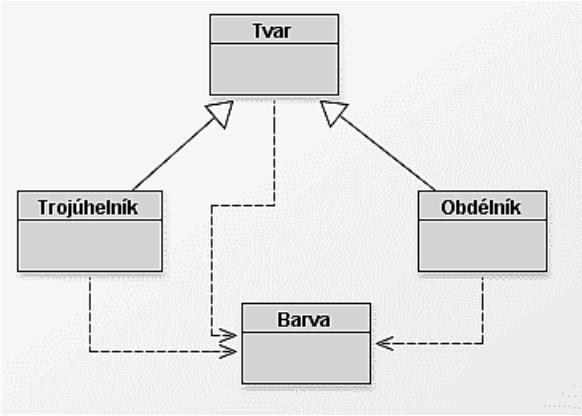
### ■ strukturovaný program = posloupnost příkazů

- analýza: vymýšlejí se postupy
- stavební kameny: procedury/funkce; (proměnné)
- výsledek: většinou samostatný program

- objektově orientovaný program = množina tříd a jejich instancí, které si posílají zprávy
  - analýza: definují se účastníci a jejich spolupráce
  - stavební kameny: třídy a objekty (o stupeň vyšší úroveň než u strukturovaného programování)
  - výsledek: velmi často komponenta, služba či jiná část celku
- OOP vyžaduje výrazně jiný způsob uvažování než klasické strukturované programování, za to ale poskytuje výhody:
  - zmenšuje sémantickou mezeru
    - ◆ přibližuje vyjadřování programátora vyjadřování zákazníka
  - dovoluje lépe zvládat velké projekty
    - ◆ snáze rozkládá celý problém na menší, mentálně uchopitelné části
    - ◆ umožňuje snáze spravovat vzájemnou spolupráci těchto částí
  - vyšší stabilita kódu
    - ◆ umožňuje zapouzdření kódu spolu s daty, nad nimiž kód pracuje
    - ◆ dokáže program zabezpečit proti řadě dříve běžných chyb
    - ◆ dovoluje snazší testování již během vývoje
  - výrazně levnější a snazší údržba kódu
    - ◆ umožňuje program předem připravit na změny zadání
    - ◆ kód je přehlednější
  - snazší příprava znovupoužitelného kódu

## 1.1.4. UML

- *Unified Modeling Language* – jednotný jazyk pro modelování
- univerzální jazyk pro grafické modelování software – pro návrh architektury systému
- obsahuje 14 diagramů (verze 2.4.1)
- využijeme pouze jeden z nich *Class diagram* (diagram tříd)



## 1.1.5. Testování

- existuje pojem **vývoj řízený testy** (TDD – *Test Driven Development*)

- nejprve vytvořit testy a pak teprve testovaný program
- kniha Beck, K.: Programování řízené testy
- extrémní případ využití testování – většinou se používají mírnější varianty

- jednotkové testy – JUnit

- předpřipravená knihovna pro snazší vytváření a vyhodnocování testů (autor Beck)
- standard, který je ve všech IDE
- testy programových jednotek – metod, tříd, ...
- většinou bývá stejná sada objektů testována z různých pohledů / funkčností; JUnit proto zavádí následující technologii (kterou přejímá BlueJ):
  - ◆ testovaná sada objektů označovaná jako **testovací přípravek** (*test fixture*) je připravena samostatně
  - ◆ testovací přípravek je nahrán před spuštěním každého testu – test pak pouze prověřuje **chování** objektů z přípravku

## 1.2. Obecné pojmy z objektově orientovaného přístupu

- každý program je model reálného či virtuálního světa, který sestává z objektů
- programovací jazyk musí podporovat konstrukce, které umí s objekty pracovat
- objekt je vše, co lze označit podstatným jménem, např.
  - „skutečné“ objekty (auto, pes, obdélník, ...)
  - vlastnosti (velikost, barva, směr, ...)
  - skupiny aktivit (výpočty [Math], testy, ...)

- např. obdélník má červenou barvu (dva objekty – obdélník, barva)

## ■ třídy a objekty

- objekty se stejnými vlastnostmi se sdružují do tříd
- objekty patřící do skupiny třídy jsou instance třídy
- termíny objekt a instance jsou synonyma
- třída je zvláštní druh objektu, který umí na požadání vytvořit svoji instanci
  - ◆ třída definuje možné vlastnosti a schopnosti svých instancí
  - ◆ strom je třída
  - ◆ zelený strom a žlutý strom jsou instance této třídy

## ■ v reálném světě spolu objekty spolupracují

- v OOP je spolupráce realizována zasíláním zpráv, kdy jeden objekt posílá zprávu druhému objektu
  - ◆ třída dostává zprávu, že má vytvořit nový objekt
  - ◆ objekt dostává zprávu, že má v rámci svých možností něco provést
- zpráva má vždy adresáta
- v aktivitě, kterou objekt vykonává jako reakci na zprávu, je skryt algoritmus její činnosti, např.:
  - ◆ vytvoř novou instanci obdélníka
  - ◆ posuň tuto instanci vpravo
  - ◆ změň barvu
- zprávy jsou metody (funkce, podprogramy)
  - ◆ „zasílání zprávy“ a „volání metod“ jsou synonyma
- každý objekt stejné třídy má k dispozici stejné metody
- zprávy mohou mít stejné názvy (přetížení)
  - ◆ pak se liší počtem parametrů

## 1.2.1. Základní pilíře OOP

### ■ zapouzdření

- data (vlastnosti) a kód (schopnosti) jsou pohromadě
- důsledek – skrývání implementace – nikomu není nic do toho, jak objekt realizuje svoji schopnost
  - ◆ tomu, kdo zasílá zprávy, stačí, aby věděl, že objekt má požadovanou schopnost
- výhody: robustnost, možné modifikace

## ■ jasně stanovená identita

- každý objekt někomu patří – tomu, kdo jej vytvořil
- každá zpráva musí mít svého adresáta, tj. objekt, kterému je zaslána
- důsledek – objekt po zaslání zprávy sám rozhodne, jak na ni bude reagovat – možný **polymorfismus**, kdy se až za běhu programu určí skutečná reakce
  - ◆ po zprávě `jez()` zareagují objekty, které se navenek chovají / tváří jako objekty třídy `Host`, trochu jinak – oba začnou jíst (tj. hlavní činnost shodná), ale objekt `evropan` použije příbor, objekt `japonec` hůlky

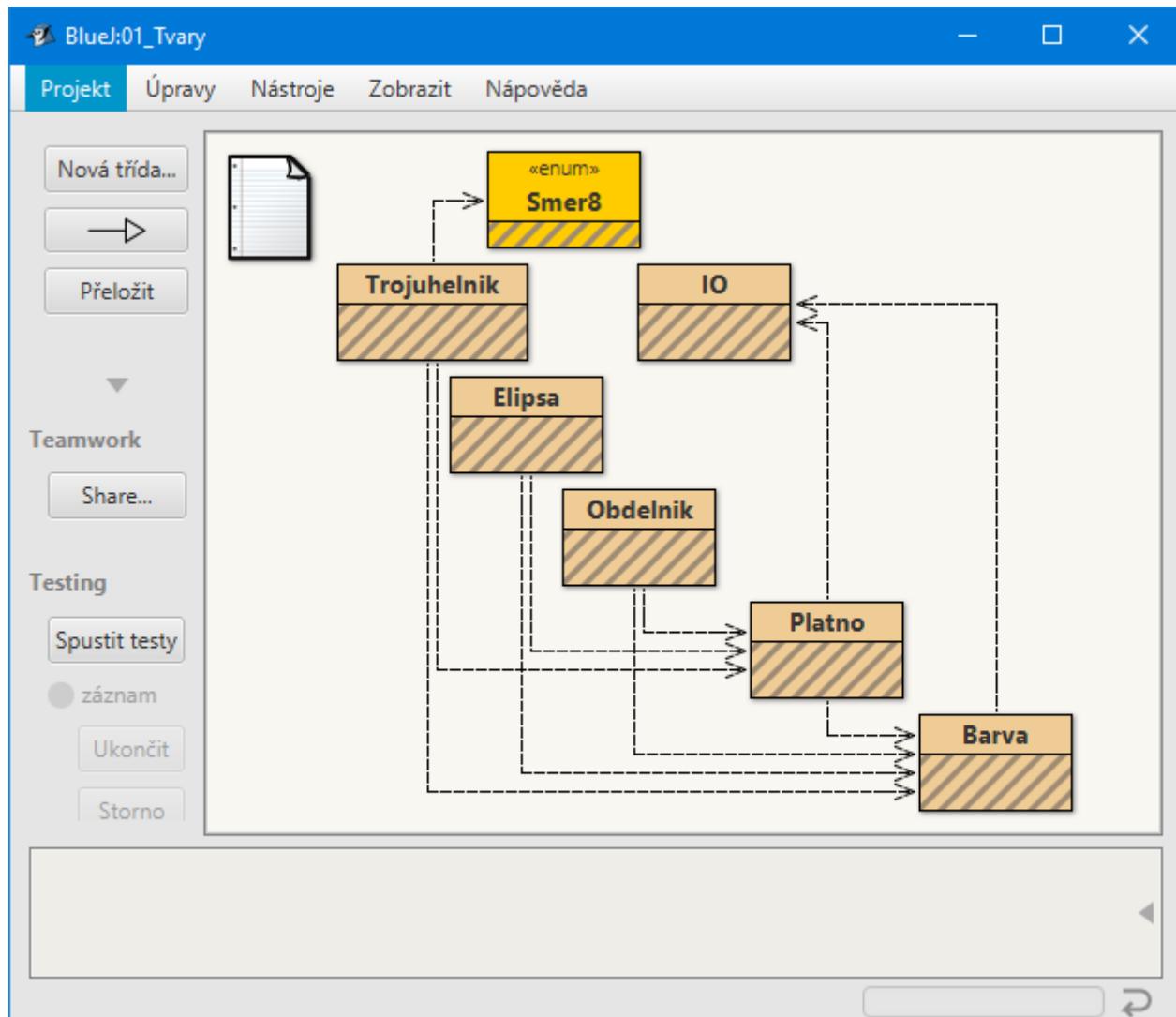
## ■ skládání

- objekt může obsahovat jiné objekty
- je více způsobů skládání, jedním z možných je dědění

## ■ používání návrhových vzorů (*design patterns*)

- obdoba matematických vzorečků
- je-li úkol zjistit plochu kruhu, pak jej lze vyřešit např. integrováním
- naprostá většina lidí ale použije vyzkoušený vzorec:  $s = \pi \times r^2$
- výhody:
  - ◆ rychlé řešení – nevymýšlí se, pouze se použije
  - ◆ vyzkoušené řešení – malá pravděpodobnost, že uděláme chybu
  - ◆ řešení každý rozumí – zjednodušená komunikace mezi členy týmu a dokumentace
- znalost návrhových vzorů patří k povinné výbavě současného objektově orientovaného programátora

## 1.2.2. Třídy a objekty v interaktivním režimu BlueJ

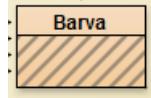
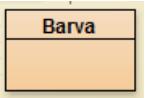


- primární pohled BlueJ je na strukturu tříd a vztahy mezi nimi – koncepční pohled
  - je to víceméně UML diagram tříd
  - plně podporuje myšlenku OOP „definují se účastníci a jejich spolupráce“
- každá třída je představena obdélníkem
  - název třídy je v horní části obdélníka
  - šrafování znamená, že třída nebyla dosud přeložena
- šipky závislostí, např. Elipsa závisí na třídě Barva
  - pokud nebude Barva existovat (nebo bude chybná), nepůjde přeložit ani Elipsa
- některé třídy jsou speciální – Smer8 – jsou odlišeny barvou a tzv. UML stereotypem <>enum<>
  - lze použít diakritiku, u názvů tříd obecně nedoporučuji (je podle nich pojmenován soubor – **problémy s přenositelností** – řeší se pomocí .jar)
- speciality BlueJ

- soubor README
- přímý přístup k jednotlivým třídám – zdrojový kód a dokumentace
- nepotřebujeme `main()` – zprávy budeme posílat sami
- na disku jsou:
  - ◆ `.java` – soubory jednotlivých tříd
  - ◆ `bluej.pkg` – konfigurace tohoto projektu (a `package.bluej` se stejným obsahem)
  - ◆ další soubory (`.class` a `.ctxt`) a adresář (`doc`) se vytvářejí při překladu či generování dokumentace a je možné je po uzavření projektu smazat

## Varování

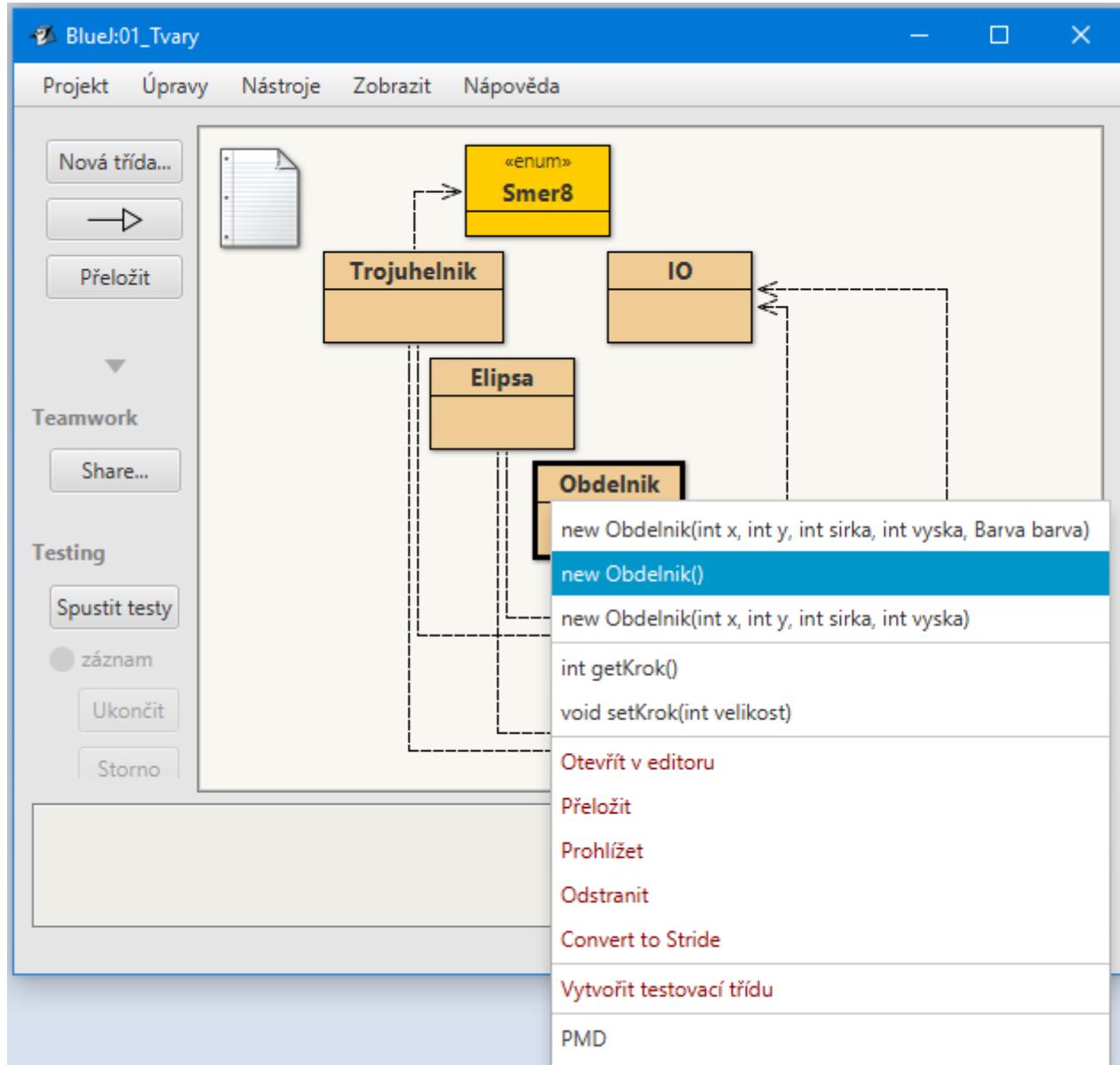
Pokud je projekt v BlueJ otevřen, neměnit z vnějšku jeho soubory!

- nepřeložená  a přeložená třída 

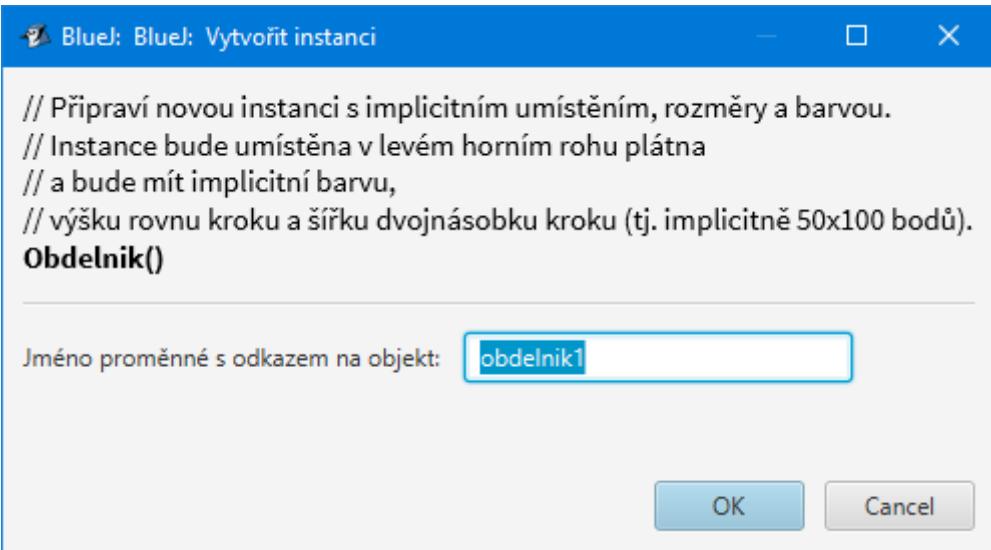
### 1.2.2.1. Zasílání zpráv

#### Poznámka

Podrobnosti o zprávách viz souhrnně dále.



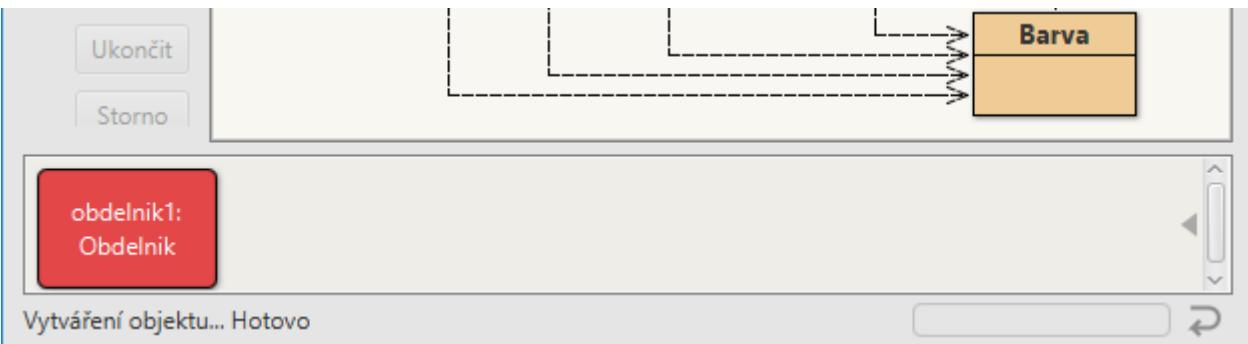
- v interaktivním režimu lze zaslat zprávu třídě zadáním příkazu z její místní nabídky (pravé tlačítko myši)
- typicky vytváříme novou instanci pomocí `new` (volání konstruktoru)
  - je vidět i možné přetížení – tři konstruktory vytvářející `Obdelnik`, kdy parametry upřesňují zasílanou zprávu
  - k instanci (objektu – synonyma) máme přístup vždy jen přes její referenci (odkaz), která je pojmenována pomocí identifikátoru



- pravidla pro identifikátory v Javě [PPA1-24]:

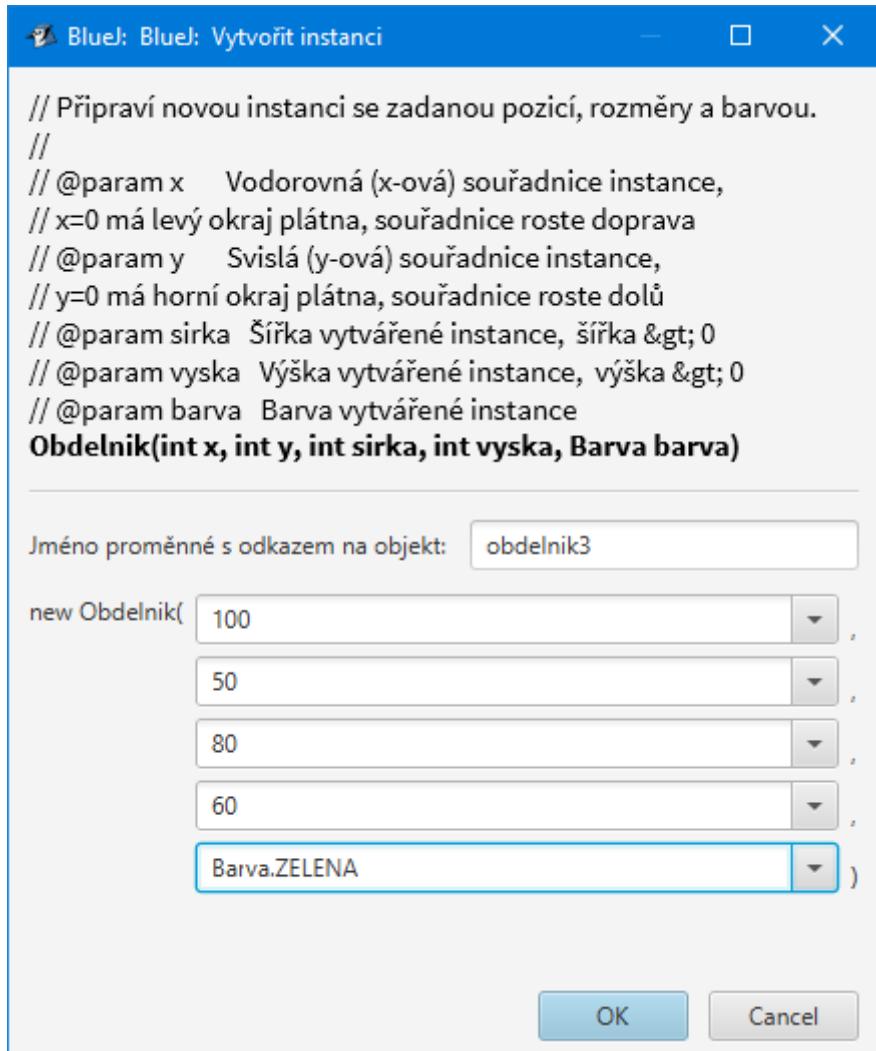
- ◆ identifikátor je libovolná posloupnost písmen, číslic a \_ nezačínající číslicí
- ◆ třída – první velké písmeno Obdelnik, ZavodniAuto
- ◆ proměnná – první malé písmeno obdelnik1, mojeCerveneZavodniAuto
- ◆ balík – všechna malá písmena auta
- ◆ symbolická konstanta – všechna velká písmena POCET\_AUT

- odkaz na vytvořenou instanci BlueJ umístí do zásobníku odkazů a označí jej zadaným názvem spolu s názvem její třídy – zde obdelnik1:Obdelnik na červeném podkladě

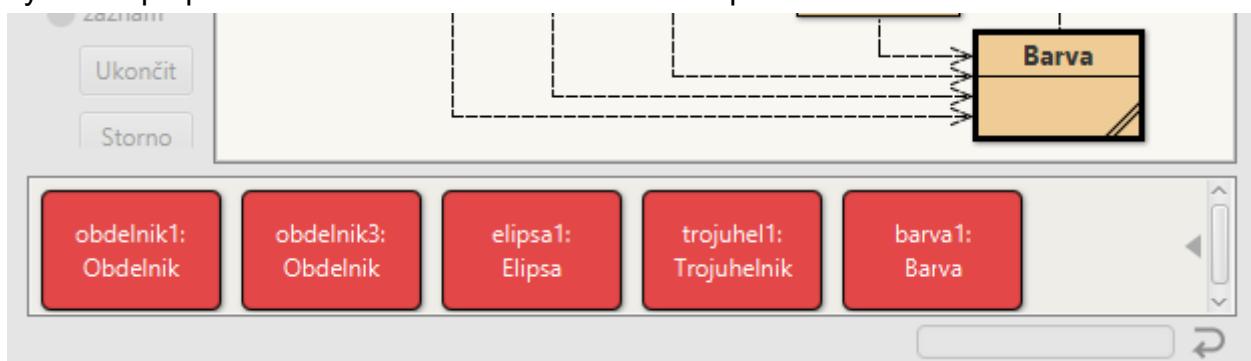


- odkazů lze vytvořit libovolné množství

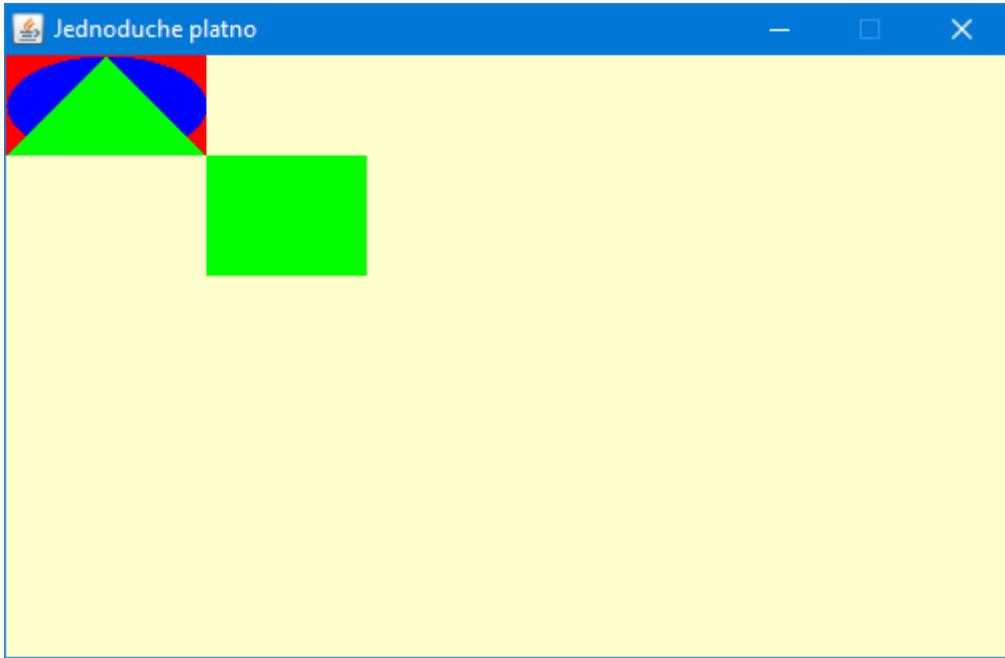
- ◆ pokud má konstruktor parametry, BlueJ je umožní zadat v dialogovém boxu



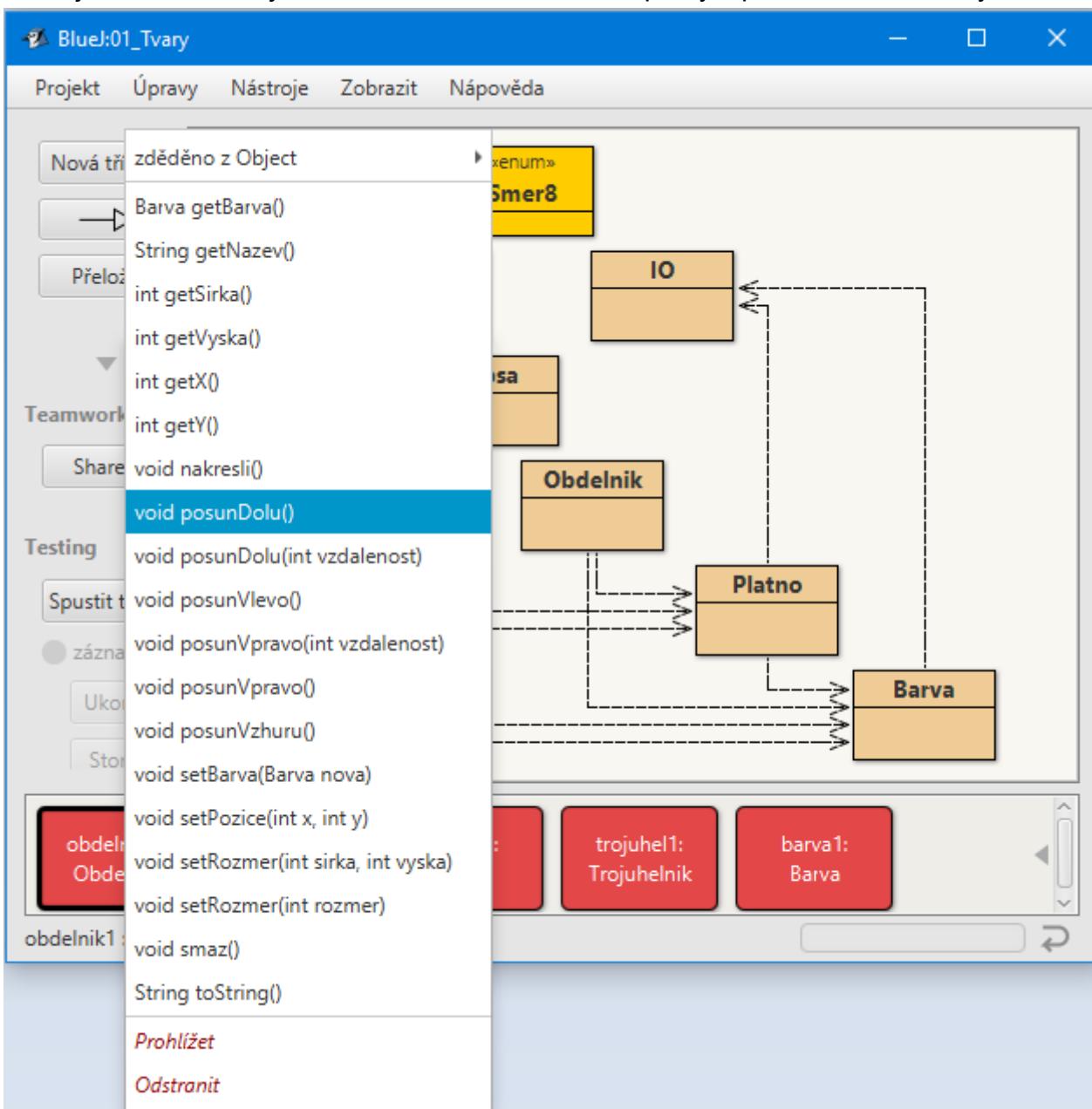
### ■ výsledek po použití několika konstruktorů – vidíme pět instancí



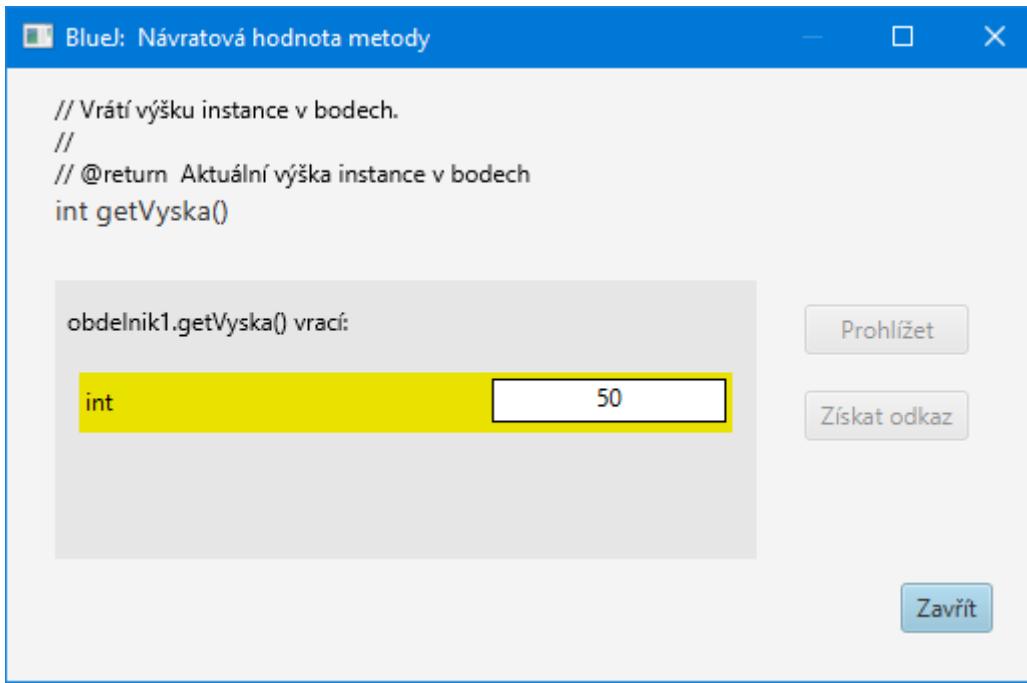
### ■ tato aplikace umí zobrazit některé vzniklé objekty na plátno



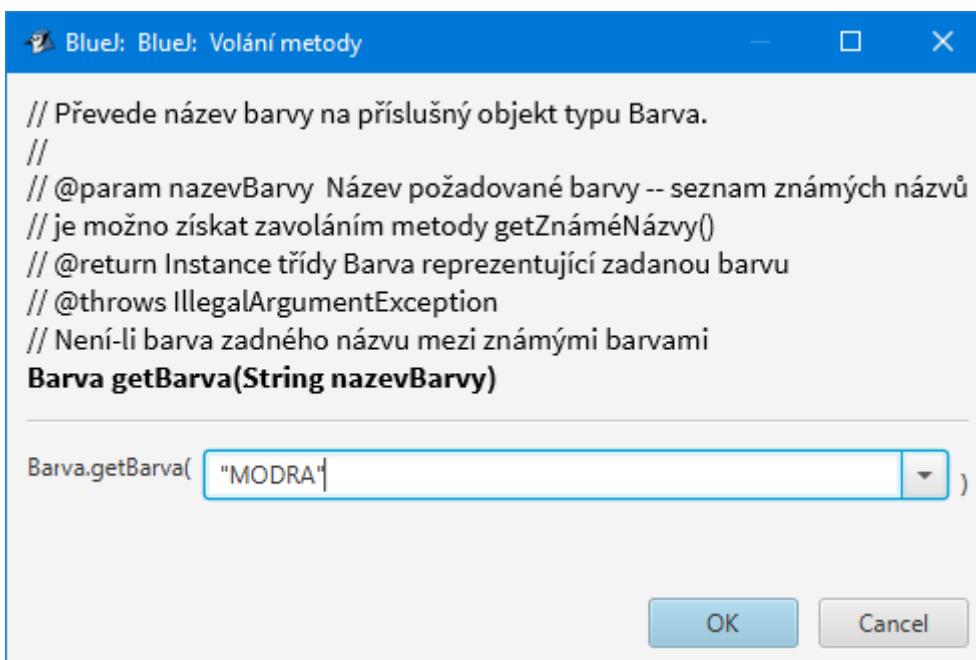
- existujícím instancím je možné interaktivně zasílat zprávy, opět z místní nabídky



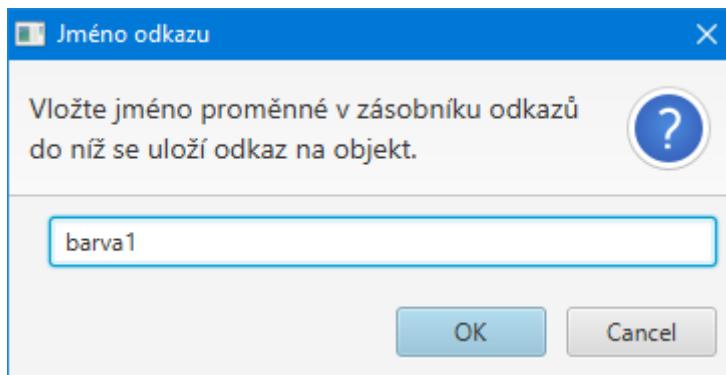
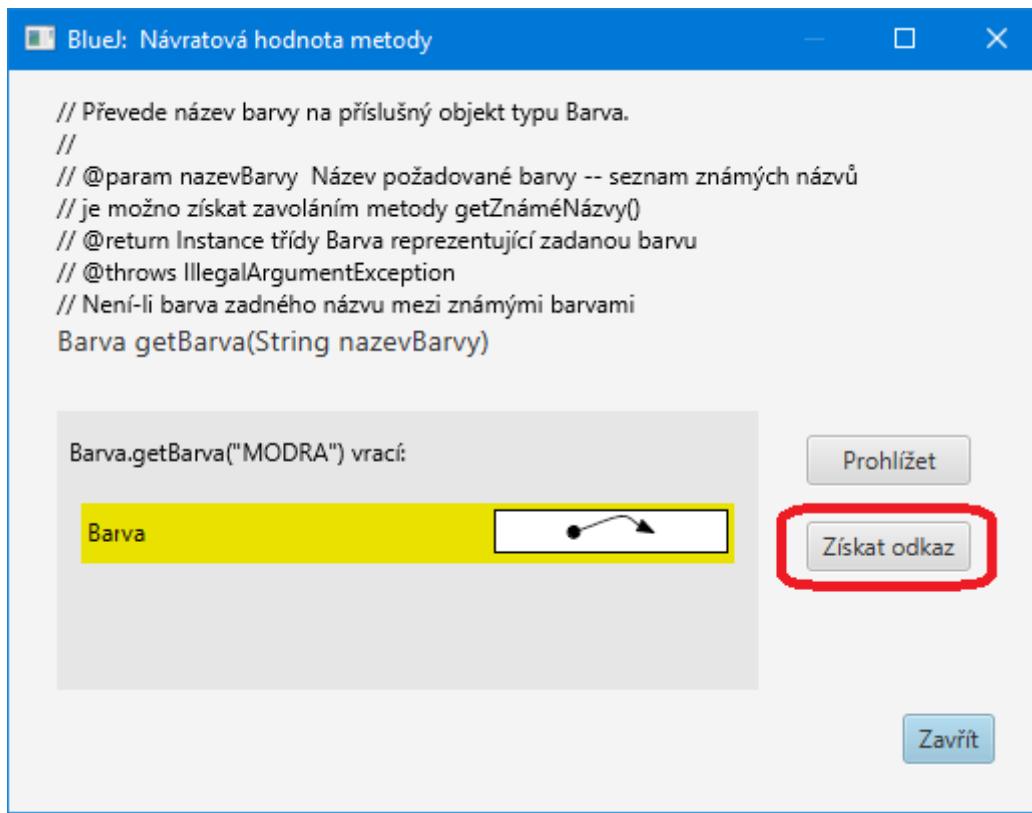
- některé metody vracejí hodnotu, kterou BlueJ zobrazí v dialogovém boxu



- třída Barva je speciální třída, jejíž instance se nezískávají pomocí konstruktoru, ale voláním jednoduché tovární metody (např. Barva getBarva(String názevBarvy)) – Pozor: uvozovky a velká písmena nutná "MODRA" !



- návratová hodnota je vrácena a je možné ji uložit jako odkaz (barva1), který lze dále používat



- animované ukázky: <http://vyuka.pecinovsky.cz/animace>

### 1.2.3. Zasílání zpráv = volání metod

- metody mají návratovou hodnotu – informaci o své činnosti, kterou předávají volajícímu
  - pokud nic předávat nechtějí, použijí typ `void` (prázdný)
- metody několika typů
  - udělají nějakou akci – posun objektu, vykreslení objektu
    - ◆ návratová hodnota je často `void`
      - není-li `void`, pak často indikace chyby (akce se podařila / nepodařila)
  - vrací informaci o svém vnitřním stavu – obdélník má červenou barvu
    - ◆ návratová hodnota je číslo nebo odkaz na jiný objekt
  - mění vnitřní stav – obdélník se posunul dolů (změnil svoji y-souřadnici)

- ◆ návratová hodnota je často `void`
- vytvářejí nové objekty – `new()` (volání konstruktoru) nebo `getBarva()` (jednoduchá tovární metoda)
  - ◆ návratová hodnota je vždy odkaz na nový objekt
- volání metod = zasílání zpráv, tzn. zpráva musí mít vždy adresáta, který se ve **zdrojovém kódu** píše jako první a odděluje se tečkou (v BlueJ je v interaktivním režimu adresát označen myší)
- metody mohou být volány
  - nad třídou – jméno třídy `Obdelnik.getKrok()`
  - nad objektem (instancí) – jméno proměnné, ve které je odkaz `obdelnik1.getBarva()`
- metody mohou mít parametry – viz později
  - pokud je nemají, musí být volány s prázdnými závorkami

### 1.2.3.1. Datové typy

- dvojice charakteristik specifikujících vlastnosti hodnot dat daného typu
  1. množina přípustných hodnot, např. čísla od 1 do 100 nebo barvy červená, žlutá a zelená
  2. operace, které lze s hodnotami daného typu provádět
    - tzn. co lze od dat očekávat a co s nimi lze provádět
      - ◆ zvyšuje se efektivita programu – optimalizované operace
      - ◆ zvyšuje se bezpečnost programu – nelze provádět cokoliv s čímkoliv
- datové typy jsou [PPA1-19] :
  - primitivní (jednoduché, základní, ...)
    - ◆ čísla, znaky, logické hodnoty
    - ◆ pracujeme s nimi přímo (ne přes odkaz pomocí zasílání zpráv)
    - ◆ z důvodů rychlosti zpracování se nejedná o objekty
      - `int` – celá čísla; rozsah  $\pm 2$  miliardy
      - `double` – reálná čísla; rozsah  $\pm 10^{\pm 300}$
      - `char` – znaky; rozsah – všechny známé znaky (existuje jich asi 100 tisíc)
      - `boolean` – logické hodnoty; rozsah – pouze dvě hodnoty `true` a `false`
    - ◆ kromě nich se občas ještě používají – `byte`, `short`, `long` a `float`
  - objektové – vše ostatní, které nejsou primitivní
    - ◆ typ definuje třída, jejíž je objekt instancí – typ `Obdelnik`

- ◆ pracujeme s nimi pomocí odkazů (referencí), přes které jim zasíláme zprávy
  - odkaz je jako dálkové ovládání televize
    - bez něj se nedá televize ovládat, i když existuje
    - pro ovládání jedné televize lze použít i několik ovladačů
- ◆ objekty vznikají v paměti na hromadě (*heap*)
- ◆ příkaz ke vzniku dáváme nejčastěji zasláním zprávy `new()`
- ◆ příkaz k zániku dává *garbage collector* (správce paměti, „sběrač nepotřebných objektů“)
  - výhoda – mnohem větší robustnost programu
  - odstraní objekt, na který není žádný odkaz, tj. nikdo mu už nemůže zaslat zprávu
- `String` (řetězec) je objektový typ, ale často používaný
  - ◆ má některé vlastnosti primitivních typů, např. nepotřebuje konstruktor `new()`
  - ◆ hodnoty instance `String` jsou v uvozovkách "MODRA"
    - bez uvozovek MODRA znamená jméno identifikátoru
- datové typy jsou vraceny metodami, ale používají se i pro deklaraci proměnných či vlastností objektů

### 1.2.3.2. Getry a setry

- speciální typ metod, které pracují pouze s atributy (stavem) objektu
  - mají název **přístupové metody**
- jmenují se stejně, jako příslušný atribut `getBarva()`, `setBarva()`
  - výjimkou jsou atributy typu boolean – viditelný
    - ◆ pak se místo `get` používá často `is` – `isViditelný()`
- `get`, `set`, `is` je konvence, se kterou počítá množství vývojových nástrojů – neměnit, nepočešťovat
- `get a is` nemají parametry
- `set` nevracejí hodnotu (jsou `void`)

### 1.2.3.3. Metody s parametry

- parametry posílaných zpráv dovolují přesněji určit, co chceme
  - zprávy bez parametrů (často u `new()` typicky u `get()`) se spoléhají na implicitní hodnoty nebo na svoji jednoznačnou činnost
- v deklaraci metody je v hlavičce metody vidět, jakých typů a významu jsou parametry
  - navíc často pomáhají dokumentační komentáře

- jednotlivé parametry jsou oddělené čárkami
- musí se dodržet počet i pořadí parametrů
- jsou-li parametry objektového typu, je nutné:
  - na něj nejdříve získat odkaz Barva barval = Barva.getBarva ("MODRA") ten následně předávat obdelnik.setBarva (barval)
  - nebo vytvořit v parametru nový objekt, např. obdelnik.setBarva (Barva.getBarva ("MODRA"))
  - nebo použít již hotový objekt obdelnik.setBarva (Barva.MODRA) – viz dále

### Poznámka

Třída Barva je netypická třída. Protože je hodně využívaná všemi ostatními třídami, má naprogramováno více možností (než je běžné), které se využijí pro snažší použití v různých případech.

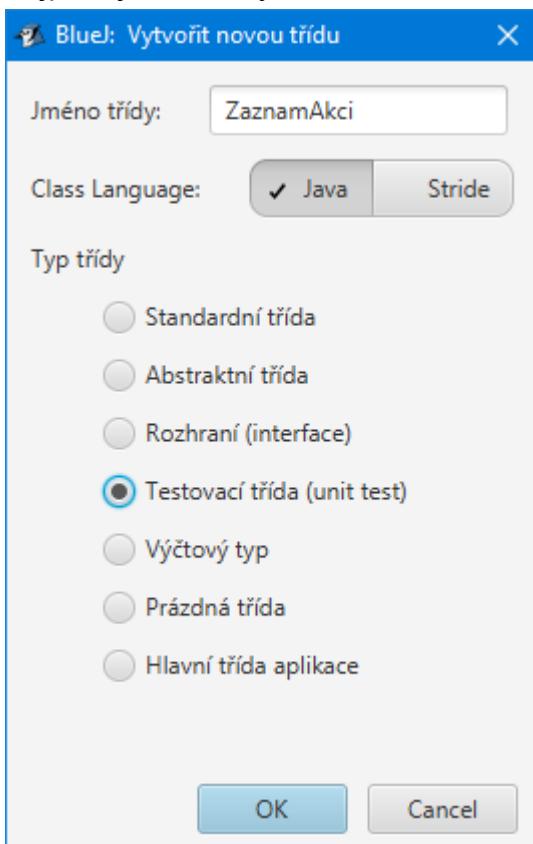
## 1.2.4. Zautomatizování interaktivního režimu BlueJ

- využívá se možností jednotkových testů (JUnit), i když se (z počátku) nic netestuje

### Poznámka

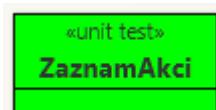
Od samého začátku práce je důležité vhodně pojmenovávat identifikátory. Pozdější oprava je možná, ale zdlouhavá.

- BlueJ od verze 3.0.5 využívá JUnit 4.x (s anotacemi)
- nejprve je nutno vytvořit testovací třídu: Úpravy/Nová třída



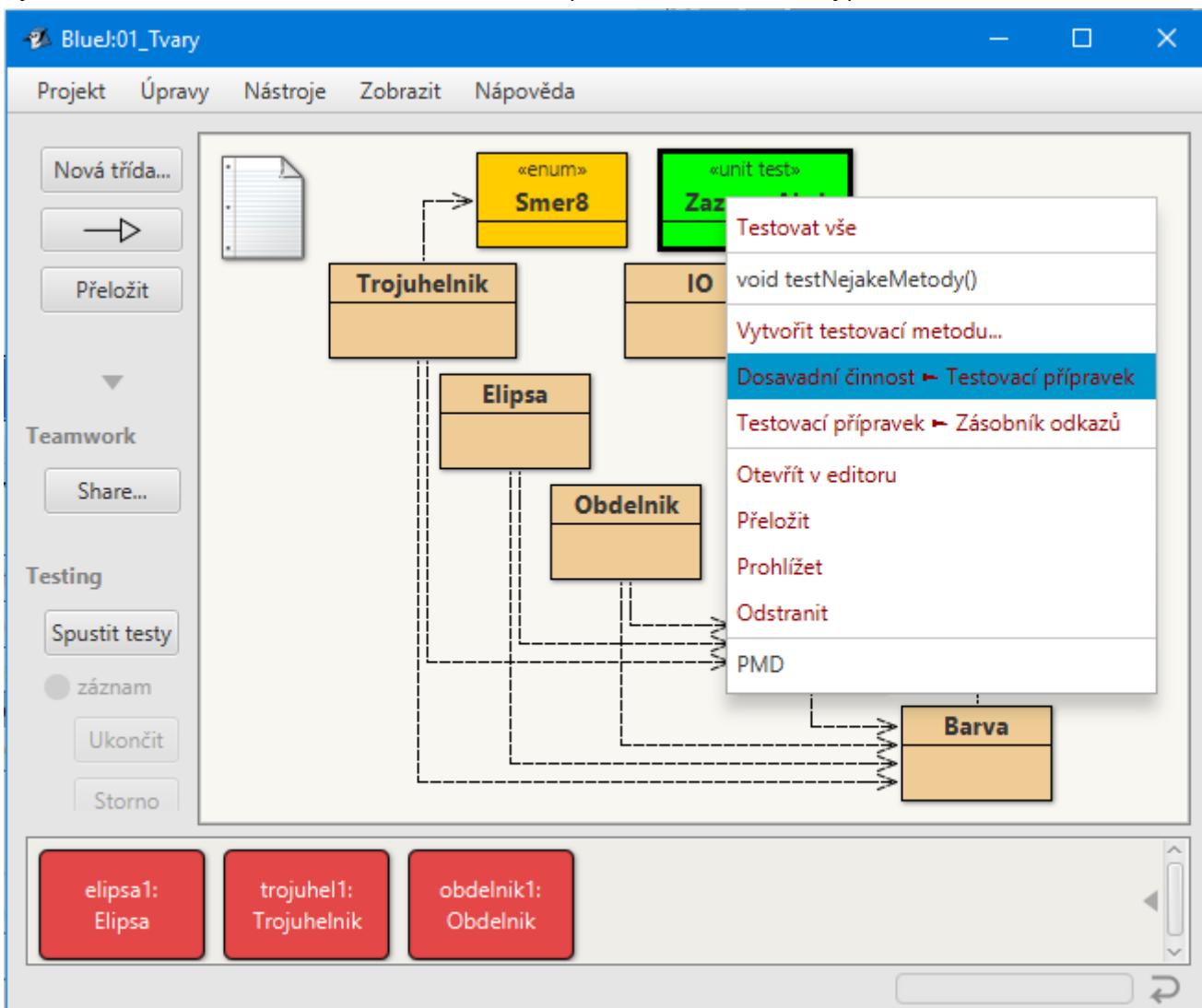
- na pojmenování nezáleží, typicky je v jejím jménu na konci slovo Test (např. ObdelnikTest), ale protože ji teď budeme používat jen pro záznam akcí, je jméno ZaznamAkci

♦

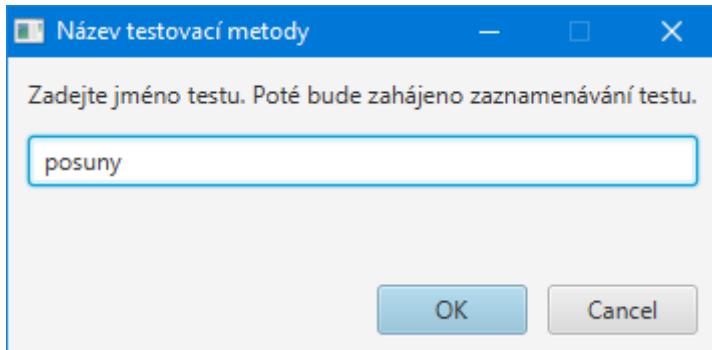


v diagramu tříd má stereotyp <<unit test>>

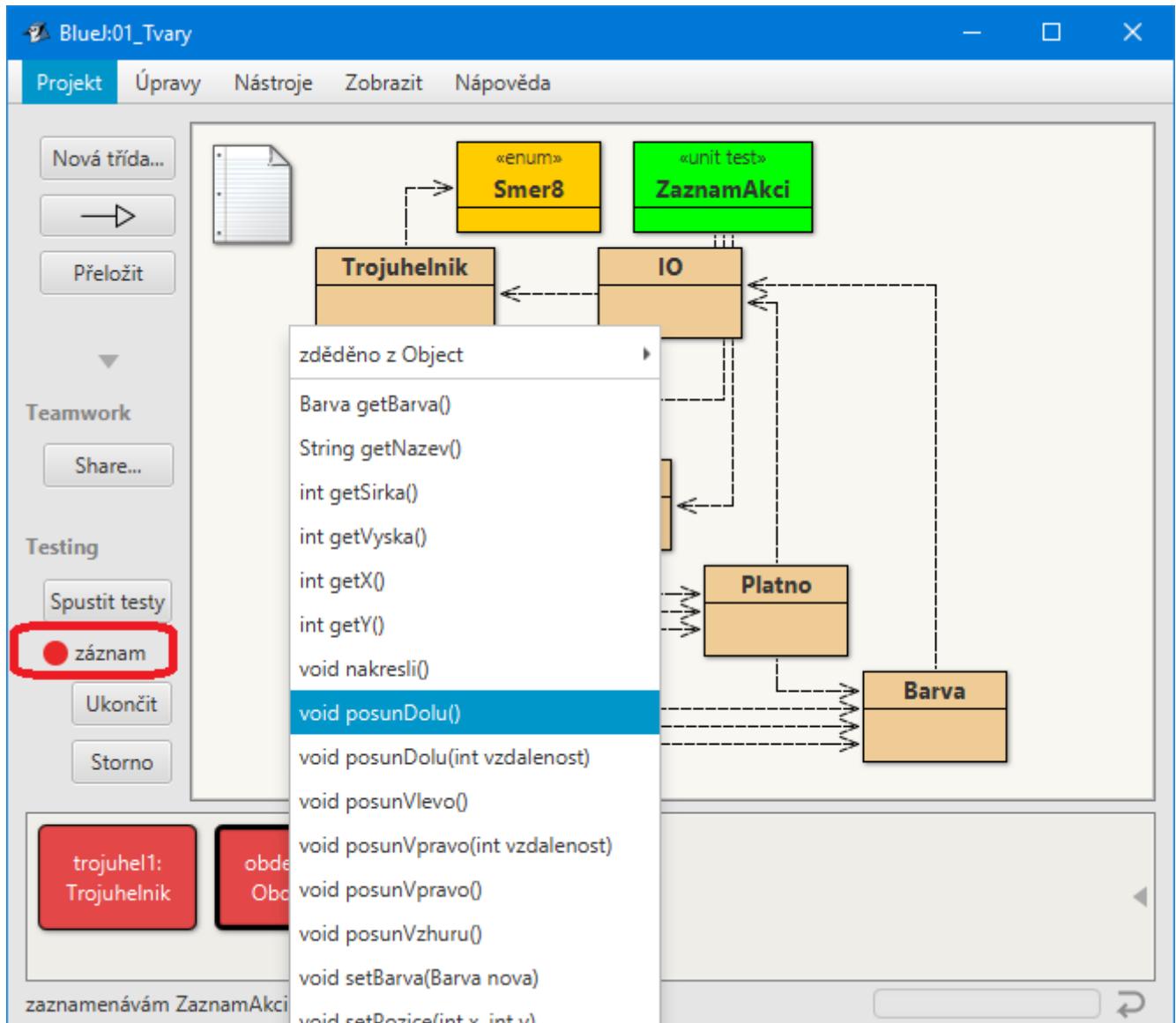
- pak je vhodné: Nástroje/Restartovat virtuální stroj
- využíváme dvou možností záznamu aktivit (obě z místní nabídky)



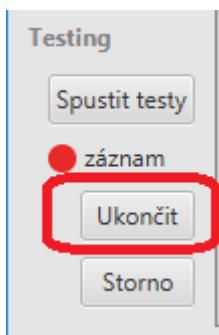
- **testovací přípravek** (Dosavadní činnost -> Testovací přípravek) – pro počáteční inicializaci objektů
  - ♦ může být pro jednu testovací třídu pouze jeden
- **testovací metodu** (Vytvořit testovací metodu) – pro libovolné další akce s již existujícími objekty nebo i s objekty novými



- ◆ pro jednu testovací třídu jich může být několik
- ◆ umožní „nahrávat“ všechny zprávy zaslané instancím

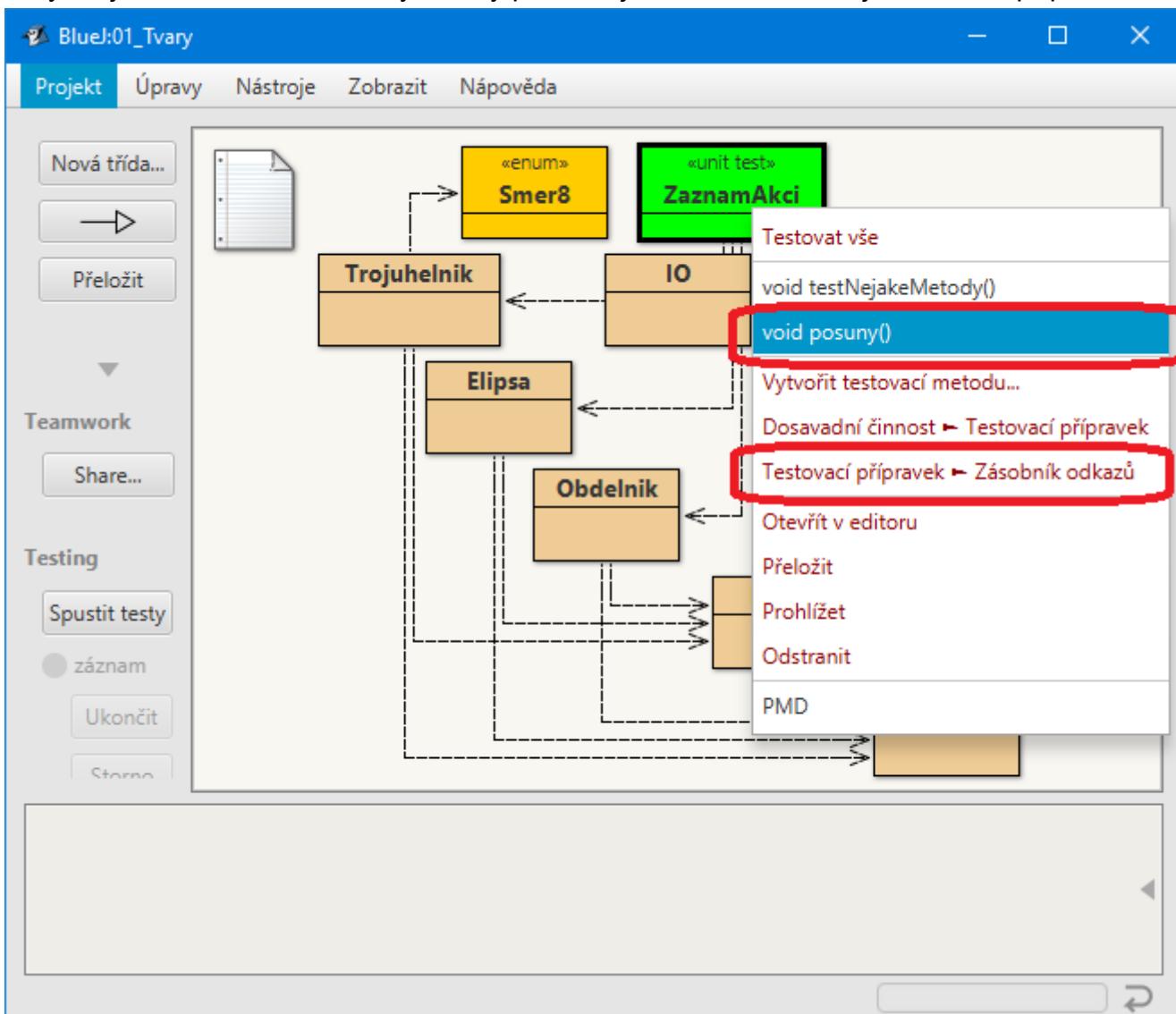


- ◆ po skončení zasílání zpráv je nutné ukončit vytváření testovací metody



■ spouštění zaznamenaných akcí – z místní nabídky třídy `ZaznamAkci`

- Testovací přípravek -> Zásobník odkazů
- výběr jména testovací metody – vždy před svojí činností aktualizuje testovací přípravek



■ oprava zaznamenaných akcí

- otevřít třídu `ZaznamAkci` v editoru – a tam editovat (čísla, jména proměnných, ...)
- ◆ testovací přípravek

```
public class ZaznamAkci
{
```

```
private Elipsa elipsa1;
private Trojuhelnik trojuheli1;
private Obdlnik obdelnik1;

@Before
public void setUp()
{
    elipsa1 = new Elipsa();
    trojuheli1 = new Trojuhelnik();
    obdelnik1 = new Obdlnik();
}
```

#### ◆ testovací metoda

```
@Test
public void posuny()
{
    elipsa1.posunDolu();
    elipsa1.posunDolu();
    trojuheli1.posunDolu();
    obdelnik1.posunVpravo();
}
```

- po opravě zdrojového souboru je nutný překlad



# Kapitola 2. Třída a její části

## 2.1. Třída

- třída je šablona pro vytváření instancí
- má dvě základní části:
  - atributy (data) – určují stav objektu
  - metody – určují schopnosti objektu
- dvě základní části se dělí do mnoha podskupin – podrobně viz dále
- třída většinou obsahuje obě základní části, ale není to nezbytné – existují speciální typy tříd
- skutečnost, že data a kód jsou pohromadě, se nazývá **zapouzdření** a je to jeden z pilířů OOP
- formální náležitosti zápisu třídy

```
public class Strom {  
}
```

- public – modifikátor přístupu – třída je veřejná a může ji používat kdokoliv (další možnost je private)
- class – jedná se o třídu (další speciální možnosti jsou enum a interface)
- Strom – název třídy, typicky podstatné jméno – musí vyhovovat pravidlům pro identifikátory
  - ◆ zdrojový kód musí ležet ve stejně pojmenovaném souboru s příponou .java (Strom.java)
- {} – blokové závorky vymezující tělo třídy – data i kód musí být uvnitř nich

## 2.2. Konstruktor

- speciální typ zprávy posílaný třídě
- má za úkol vytvořit novou instanci třídy
  - to nikdo jiný neumí – ostatní přístupy (např. Barva) vždy skrytě využívají konstruktor
- instance se vytváří dvoukrokově
  1. začne virtuální stroj, který zajistí přidělení potřebné paměti – to provede operátor new
  2. další případné doprovodné aktivity jsou zajištěny pomocí příkazů v těle konstruktoru, které mohou využít případných parametrů
- příkaz pro vytvoření nové instance

```
new NázevTřídy(seznam skutečných parametrů)
```

- seznam skutečných parametrů – specifikuje podmínky vzniku instance nebo nastavení atributů nebo je prázdný

- odkaz na novou instanci je vrácen jako hodnota operátoru new

- pokud s tímto odkazem nebudeme dále pracovat, není nutné jej přiřazovat do referenční proměnné

```
new Elipsa();
```

vytvoří objekt elipsy, který se sám (v této aplikaci) vykreslí na plátno

- zjednodušení – „konstruktor se jmenuje jako třída“

- ve skutečnosti má konstruktor skryté jméno <init>, které se objevuje např. při debugování, a jméno třídy je vlastně datový typ, který konstruktor vrací

```
public class Strom {  
    public Strom() {  
        new Elipsa(0, 0, 100, 100, Barva.ZELENA);  
        new Obdelnik(45, 100, 10, 50, Barva.HNEDA);  
    }  
}
```

- vytvoření nové instance

```
new Strom();
```

## Poznámka

Třída nemůže existovat bez konstruktoru. Nenapíše-li programátor žádný vlastní konstruktor, doplní překladač do .class **implicitní konstruktor**.

```
public Strom() {}
```

Jakmile ale uživatel jakýkoliv konstruktor vytvoří, překladač již nic nedoplňuje.

### 2.2.1. Přetížené konstruktory

- velmi často existuje více verzí konstruktoru jedné třídy – jsou přetížené

- musí se lišit alespoň jedním z:

- počtem formálních parametrů
  - typy formálních parametrů
  - pořadím formálních parametrů

```
public class Strom {  
    public Strom() {  
        new Elipsa(0, 0, 100, 100, Barva.ZELENA);  
        new Obdelnik(45, 100, 10, 50, Barva.HNEDA);  
    }  
  
    public Strom(int x, int y) { // x a y jsou formální parametry  
        new Elipsa(x, y, 100, 100, Barva.ZELENA);  
    }  
}
```

```

        new Obdelnik(x + 45, y + 100, 10, 50, Barva.HNEDA);
    }

    public Strom(int x, int y, Barva barvaKoruny) {
        new Elipsa(x, y, 100, 100, barvaKoruny);
        new Obdelnik(x + 45, y + 100, 10, 50, Barva.HNEDA);
    }
}

```

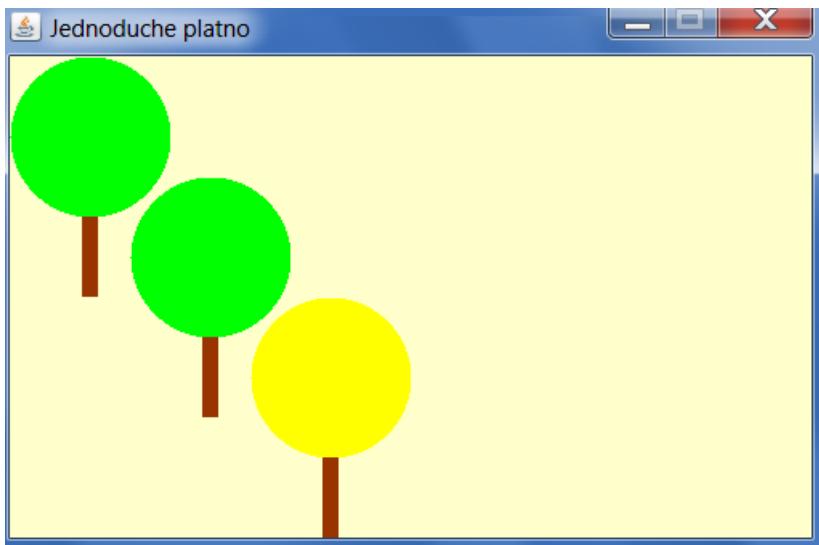
- jako skutečné parametry se zadávají hodnoty, na jejichž základě objekt modifikuje svoji reakci na zprávu

- typ předávané hodnoty musí přesně odpovídat deklarovanému typu daného formálního parametru
- jako hodnoty skutečných parametrů objektových typů se předávají odkazy na instance těchto typů (tříd)

```

new Strom();
new Strom(75, 75);      // 75 a 75 jsou skutečné parametry
new Strom(150, 150, Barva.ZLUTA);

```



## 2.2.1.1. Využití `this`

- každá instance má skrytý parametr `this`, který odkazuje na ni samu
  - pomocí konstrukce `this.jméno` lze přistupovat k libovolným instančním částem třídy (atributům či metodám)
  - u konstruktorů tak pomáhá vyvolat jiný přetížený konstruktor
    - ◆ protože konstruktor nemá jméno, používá se `this` bez následující tečky, tedy `this()`
  - využití jiného kódu (volání jiného konstruktoru) umožňuje neopakovat kód – zásada DRY (*Don't Repeat Yourself*)
    - ◆ jediná konstanta současného programování: „Zadání se brzy změní“
    - ◆ při opravách je třeba projít všechny výskyty daného kódu a všude je stejně opravit

- ◆ velké nebezpečí chyb ze špatné opravy či opomenutí výskytu
- ◆ opakovaný kód vytváří skryté vazby, které při opravách způsobují dominový efekt

- opakující se kód se umístí pouze na jedno místo a z ostatních míst se volá (pomocí `this()`)
  - volání pomocí `this()` musí být úplně prvním příkazem v těle konstruktoru
  - konstruktory mohou využívat `this()` i postupně (od jednodušších ke složitějším)

```
public class Strom {
    public Strom() {
        this(0, 0);
    }

    public Strom(int x, int y) {
        this(x, y, Barva.ZELENA);
    }

    public Strom(int x, int y, Barva barvaKoruny) {
        new Elipsa(x, y, 100, 100, barvaKoruny);
        new Obdélník(x + 45, y + 100, 10, 50, Barva.HNEDA);
    }
}
```

- předávání řízení lze hezky ukázat pomocí debuggeru BlueJ – Zobrazit / Zobrazit debugger

## 2.3. Atributy

- proměnné či konstanty popisující stav objektu (uchovávají hodnoty potřebné pro správnou funkci programu)
  - v JDK a v BlueJ jsou souhrnně označované jako *fields*
- každá třída definuje, jaké bude mít atributy a ty pak převezmou (budou je mít) všechny její instance
- atributy jsou dvojího typu
  - **třídní (statické atributy)** – existují pouze jednou a všechny instance je sdílejí – je před nimi klíčové slovo `static`
    - ◆ např. ve třídě `ČlenRodiny` je pouze jedno bankovní konto (tj. třídní atribut) společné pro všechny členy – pokud někdo z konta vybere nebo do něj uloží, poznají to všichni
  - **instanční** – každá instance má svoji kopii
    - ◆ instanční atribut je peněženka, kterou má každá instance třídy `ČlenRodiny` (otec, matka, dcera, syn) svoji a nikdo další k ní nemá přístup
- atributy (třídní i instanční) mohou být:
  - proměnné – lze je během života objektu / třídy libovolně nastavovat
  - konstanty (s modifikátorem `final`) – lze je nastavit pouze jednou – v deklaraci nebo v konstruktoru

- ♦ kdykoliv víme, že se hodnota atributu měnit nebude, dáme přednost konstantě, aby mohl překladač ohlídat, že ji ani omylem nezměníme

■ atributy mají přístupová práva (stanovená pomocí modifikátorů přístupu), nejčastěji:

- `public` (veřejný) – přístupný z vnějšku
  - ♦ používá se jen u některých konstant (`Barva.CERNA`)
  - ♦ pro použití `public` musí být skutečně vážný důvod
- `private` (soukromý) – přístupný jen pro vlastní třídu / instanci
  - ♦ je to typické nastavení – z vnějšku se čtou pomocí getrů a nastavují pomocí setrů = autorizovaný přístup, zapouzdření
  - ♦ každý smí vědět, co všechno objekt umí, ale nikomu není nic do toho, jak to vnitřně dělá
- v třídních attributech objektových typů mohou být již předpřipravené objekty (počítá se, že budou často využívány, proto se vytvoří hned po spuštění a všichni je pouze používají) – `Barva.MODRA`
  - `obdelnik.setBarva(Barva.MODRA)`
  - to je rozdíl, kdy dříve bylo nutné požádat o vrácení nové barvy podle jejího `String` jména `Barva.getBarva("MODRA")`, tento odkaz uložit do pomocné proměnné `modra` a tu pak použít při nastavování `obdelnik.setBarva(modra)`
- příklady třídních a instančních atributů

```
// Instanční proměnná
private int velikost;

// Inicializovaná instanční proměnná
private int pocet = 0;

// Instanční konstanta nastavená při deklaraci
private final int MAX = 10;

// Instanční konstanta nastavená později v konstruktoru
private final Elipsa koruna;

// Inicializovaná třídní proměnná
private static Barva implicitniBarva = Barva.CERNA;

// Inicializovaná třídní konstanta
private static final Barva IMPLICITNI_BARVA = Barva.CERNA;
```

■ příklady **inicializací v deklaraci** (platí pro třídní i instanční, pro proměnné i konstanty):

- přiřazením předem známé hodnoty

```
private int sirka = 100;
```

- přiřazením obsahu jiné (již definované a inicializované) konstanty či proměnné:

```
private Barva barvaKmene = Barva.HNEDA;
private int vyska = sirka;
```

- přiřazením odkazu získaného použitím jiného konstruktoru (málo časté – lépe dát do konstruktoru):

```
private Obdelnik kmen = new Obdelnik(0, 0, sirka, vyska, barvaKmene);
private Elipsa koruna = new Elipsa();
```

- přiřazením hodnoty vrácené nějakou metodou:

```
private Barva barvaKoruny = Barva.getBarva("ZELENA");
```

- přiřazením hodnoty výrazu:

```
private String jmenoBarvyKoruny = koruna.getBarva().getNazev();
private int vyska = sirka * 2;
```

### 2.3.1. Atributy versus vlastnosti

- běžně se říká, že atributy popisují stav či vlastnosti objektu a metody jeho schopnosti
- stejně běžně se uvažuje, že každý atribut (`private`) je přístupný přes svůj getr/setr
- není to úplně přesné, což si později uvědomíme při přípravě metod
- vlastnost (*property*)
  - můžeme ji u objektu zjistit (velikost stromu), případně i nastavit či změnit (barva koruny)
  - vlastnost je v rozhraní objektu (viz dále)
  - mnoho vlastností je realizováno pomocí atributů, ale ne všechny
  - některé vlastnosti se dají odvodit z jiných, např. vlastnost koncové x-souřadnice se dá odvodit z počáteční x-souřadnice a šířky objektu
- atribut
  - implementační záležitost, jak zajistit to, co si potřebujeme pamatovat
  - principiálně nikomu z vnějšku není nic do toho, co jsou atributy zač (typy, jména)
  - prakticky – většina atributů se kryje s vlastnostmi – stejná jména
  - neplatí ale bez zbytku pravidlo, že každý atribut má svůj getr a setr
  - existují skryté atributy, např. u Strom atributy koruna a kmen – nedovolujeme nikomu z vnějšku pracovat odděleně s částmi stromu (např. přesouvat jen kmen), vždy jen s celým stromem

## 2.3.2. Atributy v BlueJ

- instanční i třídní atributy lze prohlížet (Prohlížet – někdy je nutné rozšířit okénko)
  - u primitivních typů se zobrazí přímo hodnota
  - u objektových typů odkaz – po vybrání lze odkaz dále zkoumat (Prohlížet) nebo Získat odkaz a uložit do nově definované proměnné
- prohlížeče se dá využít i pro zobrazování průběžně měnících se hodnot – objektům se posílají příkazy a v prohlížeči se mění hodnoty atributů
- u prohlížení instancí lze přes tlačítko „Ukázat statické atributy (atributy třídy)“ zobrazit prohlížení třídních atributů

## 2.3.3. Atributy, lokální proměnné, parametry – komplexní pohled

- ve třídě se ve skutečnosti vyskytují tři typy proměnných – atributy, lokální proměnné a formální parametry

```
public class A {  
    int instančníAtribut;  
    static int třídníAtribut;  
  
    void metoda(int formálníParametr) {  
        int lokálníProměnná = formálníParametr;  
        this.instančníAtribut = lokálníProměnná;  
        A.třídníAtribut = lokálníProměnná;  
    }  
}
```

- lokální proměnné jsou proměnné definované kdekoliv v libovolné metodě, případně v jejím bloku {}
  - slouží k dočasnému uchování pomocné hodnoty
- rozdíly mezi těmito typy proměnných jsou z několika pohledů
  - atributy (instanční) a třídní atributy se z daných pohledů od sebe neliší a budou považovány za shodné

### 2.3.3.1. Rozsah platnosti

- neboli viditelnost, tj. odkud je proměnná přístupná
- atributy
  - nejjednodušší – jsou vidět kdekoliv v implementaci celé třídy
  - přístupová práva (private, public) omezují jejich viditelnost pouze zvnějšku třídy
- formální parametry

- jsou vidět pouze ve své metodě
  - přístupová práva (private, public) nemají smysl
- lokální proměnné
- jsou vidět pouze v bloku, ve kterém jsou deklarovány – nejčastěji je to celá metoda
    - ◆ a to od místa svojí deklarace dále (nikoliv dopředu)
  - přístupová práva (private, public) nemají smysl

### 2.3.3.2. Inicializace

- kde se jim nastavuje počáteční hodnota
- atributy
- nejčastěji v konstruktoru a to i nepřímo pomocí volání příslušného setru z konstruktoru
    - ◆ je to nutnost, pokud je hodnota atributu výsledkem složitějšího výpočtu
  - je-li to jednoduché a vždy stejné přiřazení, uvádí se často na místě definice atributu (`int atribut = 5;`)
  - zapomeneme-li je inicializovat, mají nulovou hodnotu – nikdy nedělat !

```
public class A {
    int instančníAtribut = 5;
    static int třídníAtribut = 8;

    public A(int param) {
        A.třídníAtribut = A.třídníAtribut + 1;
        setInstančníAtribut(param * 123);
    }

    void setInstančníAtribut(int par) {
        this.instančníAtribut = par;
    }
}
```

- formální parametry
- při volání metody se použijí skutečné parametry (`setAtribut(param * 123);`)
- lokální proměnné
- typicky při deklaraci, ale nejpozději do okamžiku prvního použití – hlídá překladač

```
void metoda(int formálníParametr) {
    int lokálníProměnná = formálníParametr / 123;
    this.instančníAtribut = lokálníProměnná;
    A.třídníAtribut = lokálníProměnná;
}
```

### 2.3.3.3. Doba života

- jak dlouho je proměnné přidělena paměť
- atributy se zde odlišují
  - instanční atributy – od zavolání konstruktoru po celou dobu existence objektu
  - třídní atributy – od začátku do konce programu
- formální parametry
  - po vstupu do metody až do skončení metody
- lokální proměnné
  - od okamžiku své deklarace až do skončení bloku

### 2.3.3.4. Měnitelnost

- kdy je možné nebo typické proměnnou měnit
- atributy
  - počáteční nastavení v konstruktoru či deklaraci
  - průběžná změna nejčastěji pomocí setru
    - ◆ méně často nějakou jinou specializovanou metodou
- formální parametry
  - je možné je změnit v metodě, ale téměř nikdy se to nedělá
  - formální parametr je považován za vstupní, tj. pouze ke čtení – PMD hlídá
  - chceme-li zajistit, aby byl formální parametr jen vstupní, použijeme `final`

```
void metoda(final int formálníParametr) {  
    int lokálníProměnná = formálníParametr;  
    // formálníParametr = 10;    // nelze  
}
```

- lokální proměnné

- běžně měníme pomocí přiřazovacích příkazů

```
void metoda(int formálníParametr) {  
    int lokálníProměnná = formálníParametr / 123;  
    this.instančníAtribut = lokálníProměnná;  
    lokálníProměnná = 58;  
    A.třídníAtribut = lokálníProměnná;  
}
```

## 2.3.4. Pokračování příkladu s třídou Strom

- vytvořený strom se uměl pouze zobrazit
- pokud budeme chtít v budoucnu např. změnit barvu koruny (na podzim zežloutne), musíme mít na ni uschovaný odkaz
  - odkaz na korunu bude typu `final`, protože elipsa se nebude měnit (nahrazovat se jinou elipsou), bude měnit jen svoji barvu
  - nemusíme si schovávat aktuální barvu koruny, protože ta je schovaná v objektu elipsy
  - `x` uchovává x-souřadnici stromu – v konstruktoru musíme použít `this`, aby se odlišil od formálního parametru

```
public class Strom {  
  
    private static final Barva BARVA_KMENE = Barva.HNEDA;  
    private static final Barva IMPL_BARVA_KORUNY = Barva.ZELENA;  
    private static final Barva BARVA PODZIMU = Barva.ZLUTA;  
  
    private final Elipsa koruna;  
    private final Obdelnik kmen;  
    private int x;  
  
    public Strom() {  
        this(0, 0);  
    }  
  
    public Strom(int x, int y) {  
        this(x, y, IMPL_BARVA_KORUNY);  
    }  
  
    public Strom(int x, int y, Barva barvaKoruny) {  
        koruna = new Elipsa(x, y, 100, 100, barvaKoruny);  
        kmen = new Obdelnik(x + 45, y + 100, 10, 50, Barva.HNEDA);  
        this.x = x;  
    }  
}
```

### Poznámka

Atribut `x` je zde ve skutečnosti zbytečný, protože jeho hodnota je totožná s hodnotou, kterou lze získat z objektu `koruna`. Je použit pro ukázku `this` – viz též dále.

## 2.4. Metody

- podrobnosti o konstrukcích metod [PPA1-70]
- je vhodné rozlišovat atributy objektu a vlastnosti objektu – viz dříve
- názvem metody by mělo být sloveso – je to příkaz nějaké akce

## ■ metody mají za úkol

- poskytnout informace o vlastnostech objektu – getry
  - ◆ vrací přímo hodnotu atributu – getX()
  - ◆ vrací právě vypočtenou (zjištěnou) hodnotu – isPodzim(), getVyska()
- změnit vlastnosti objektu – setry – zezloutni(), setBarvaKoruny()
- provádět další činnosti spojené s objektem, např. jeho vykreslení – nakresli(), smaz()

## ■ ke statickým atributům mohou přistupovat pouze statické metody

- instanční metody mohou přistupovat ke všem atributům

```
public int getX() {  
    return this.x;  
}  
  
public int getY() {  
    return koruna.getY();  
}  
  
public int getVyska() {  
    return koruna.getVyska() + kmen.getVyska();  
}  
  
public boolean isPodzim() {  
    return koruna.getBarva() == BARVA PODZIMU;  
}  
  
public void zezloutni() {  
    setBarvaKoruny(BARVA PODZIMU);  
}  
  
public void postupneZezloutni() {  
    setBarvaKoruny(Barva.KHAKI);  
    IO.cekej(500);  
    setBarvaKoruny(BARVA PODZIMU);  
}  
  
public void setBarvaKoruny(Barva barva) {  
    koruna.setBarva(barva);  
}  
  
public void nakresli() {  
    koruna.nakresli();  
    kmen.nakresli();  
}  
  
public void smaz() {  
    koruna.smaz();  
}
```

```
    kmen.smaž();  
}
```

## Poznámka

Metoda `isPodzim()` je příklad analyticky nevhodné metody. Bylo by krajně problematické usuzovat na existenci podzimu z hodnoty jedné z mnoha možných instancí stromů. Když bychom tuto metodu potřebovali použít, měla by se jmenovat `isPodzimníBarva()`, což by mnohem lépe vypovídalo o tom, že se jedná o vlastnost konkrétní instance.

Metoda `isPodzim()` bude dále použita v ukázce testů a pak již používána nebude.

## 2.4.1. Porovnávání objektů – úvod

- metoda `isPodzim()` je i chybně napsaná

```
public boolean isPodzim() {  
    return koruna.getBarva() == BARVA PODZIMU;  
}
```

- je funkční jen díky mimořádným nadstandardním vlastnostem třídy `Barva` (kde každá barva existuje jen v jedné instanci)
- pokud porovnáváme objekty, ve většině případů je nefunkční je porovnávat pomocí `==`
- je třeba použít metodu `equals()`
- `==` testuje vždy, zda jsou objekty totožné, zatímco metoda `equals()` může testovat, zda objekty mají stejné hodnoty atributů

```
public boolean isPodzim() {  
    Barva barva = koruna.getBarva();  
    boolean jePodzimni = barva.equals(BARVA PODZIMU);  
    return jePodzimni;  
  
    /* nebo stručněji  
     * return koruna.getBarva().equals(BARVA PODZIMU);  
     */  
}
```

- podrobně viz později

## 2.4.2. Využití `this` u metod

- `this.` před voláním metody říká, že zasíláme zprávu sami sobě (voláme metodu z této instance)
  - u metod nemůže (běžně – viz později vnořené třídy) dojít ke konfliktu jmen – na rozdíl od atributů, které se mohly jmenovat jako formální parametry nebo lokální proměnné
  - voláme-li metody vlastní třídy, nemusíme `this` psát

```
public void zežloutni() {  
    setBarvaKoruny(BARVA PODZIMU);  
}
```

je stejně jako:

```
public void zežloutni() {  
    this.setBarvaKoruny(BARVA PODZIMU);  
}
```

profesionálové používají spíše druhý způsob – na první pohled je vidět, kdo je adresátem zprávy

## 2.4.3. Atributy a metody třídy

- protože je před nimi klíčové slovo `static`, nazývají se též **statické**
- velmi typické použití je pro konstanty
- další typické použití je pro počítadlo, kolik objektů této třídy již vzniklo, což se často spojuje i s názvem třídy

```
/** Počet doposud vytvořených instancí. */  
private static int pocet = 0;  
  
/** Pořadí kolikátá byla daná instance vytvořena v rámci třídy. */  
private final int PORADI = ++pocet;  
  
/** Název instance sestávající z názvu třídy a pořadí instance */  
private final String NAZEV = "Strom_" + PORADI;  
  
public int getPoradi() {  
    return PORADI;  
}  
  
public String getNazev() {  
    return NAZEV;  
}
```

### 2.4.3.1. Inicializace atributů třídy

- pokud nelze atribut třídy inicializovat jednoduchým příkazem v jeho deklaraci, lze použít **statické inicializační bloky**
- začínají klíčovým slovem `static` a pak je kód v blokových závorkách `{ }`

```
public class StatickeInicializace {  
  
    public static final int STATICKA_KONSTANTA = 10;  
  
    private static int jednoducheCislo = STATICKA_KONSTANTA * 3;  
  
    private static int sloziteCislo;  
    private static int slozitejsiCislo;
```

```

// statický inicializační blok
static {
    sloziteCislo = (int) Math.log(Math.pow(jednoducheCislo, 2.5));
    slozitejsiCislo = sloziteCislo / 2;
}
}

```

- v součinnosti s konstruktory (vytváření instancí) je možné množství kombinací, které mohou způsobit komplikace

- proto je vhodné dodržovat několik jednoduchých doporučení (podrobně viz Pecinovský NMPO)
  - kód začíná deklaracemi statických konstant – STATICKA\_KONSTANTA
  - ◆ jsou inicializovány pouze přímým přiřazením nejlépe literálu (zde 10), případně jednoduchým výrazem

```
STATICKA_KONSTANTA = 10;
```

- po konstantách následují proměnné statické atributy – jednoducheCislo
  - ◆ inicializace v deklaraci může využívat již zavedených konstant, opět se omezujeme na přiřazení jednoduchého výrazu
  - ◆ ve výrazech používáme pouze ty statické atributy, které byly před tím jednoznačně inicializovány
- všechny výpočty složitějších inicializačních výrazů dáváme do statického inicializačního bloku
  - ◆ ten je uveden až za všemi statickými atributy (na pořadí **za nimi** záleží !)
  - ◆ používá se zásadně pouze jeden statický inicializační blok
  - ◆ pokud jsou v něm volány statické metody, je třeba ověřit, že používají pouze již inicializované statické atributy

- pro instanční atributy platí v podstatě stejná pravidla

- pouze složitější konstantní inicializační výrazy se uvádějí v konstruktoru
- velmi časté je, že atribut je inicializován stejně pojmenovaným formálním parametrem – viz z dřívějška:

```

public class Strom {
    private final Elipsa koruna;
    private final Obdelnik kmen;
    private int x;

    public Strom(int x, int y, Barva barvaKoruny) {
        koruna = new Elipsa(x, y, 100, 100, barvaKoruny);
        kmen = new Obdelnik(x + 45, y + 100, 10, 50, Barva.HNEDA);
        this.x = x;
    }
}

```

```
    }  
}
```

## 2.4.4. Odložená inicializace (*lazy initialization*)

- občas má třída instanční či statický atribut s hodnotou, jejíž zjišťování je náročné a přitom daný atribut nikdo ani nemusí v konkrétní aplikaci využít
- tento atribut nenastavujeme v konstruktoru, ale v metodě `get...()`
  - ta „komplikovanou hodnotu“ jednou vypočte a vrátí a současně ji uloží do atributu
- při každém dalším volání už vrací hodnotu atributu
- doporučuje se, aby atribut byl objekt (v případě primitivních datových typů pak obalová třída – zde `Integer`), aby se ještě nenastavený atribut snadno poznal podle hodnoty `null`
  - nebo je lépe použít třídu `Optimal` – viz později

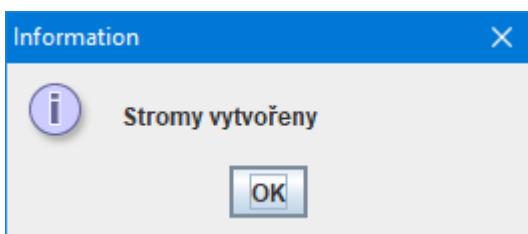
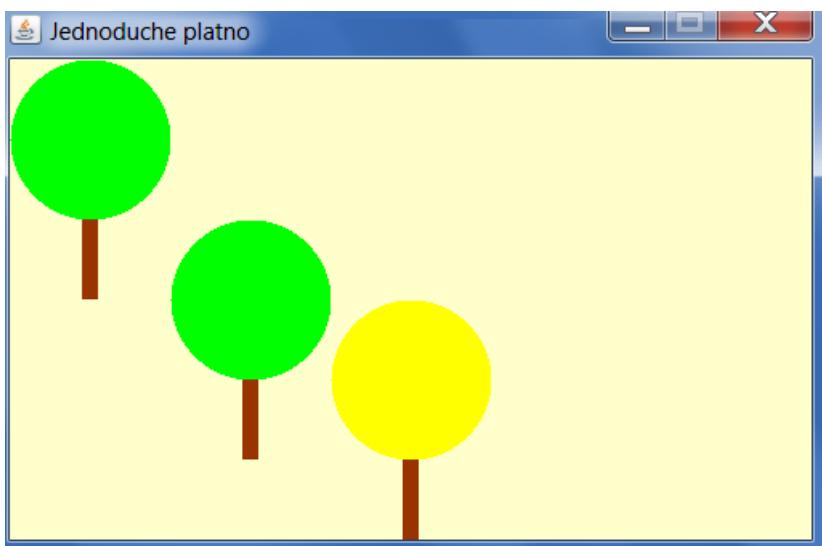
```
class Osoba {  
    // atributy  
    private final String jmeno;  
    private final double vaha;  
    private final int vyska;  
  
    private Integer BMI = null;  
  
    // konstruktor  
    public Osoba(String jmeno, double vaha, int vyska) {  
        this.jmeno = jmeno;  
        this.vaha = vaha;  
        this.vyska = vyska;  
    }  
  
    public Integer getBMI() {  
        if (BMI == null) {  
            double vyskaMetry = vyska / 100.0;  
            double bmi = vaha / (vyskaMetry * vyskaMetry);  
            int bmiCele = (int) Math.round(bmi); // zaokrouhleni  
            BMI = new Integer(bmiCele);  
        }  
        return BMI;  
    }  
}
```

## 2.4.5. Využití testů v BlueJ

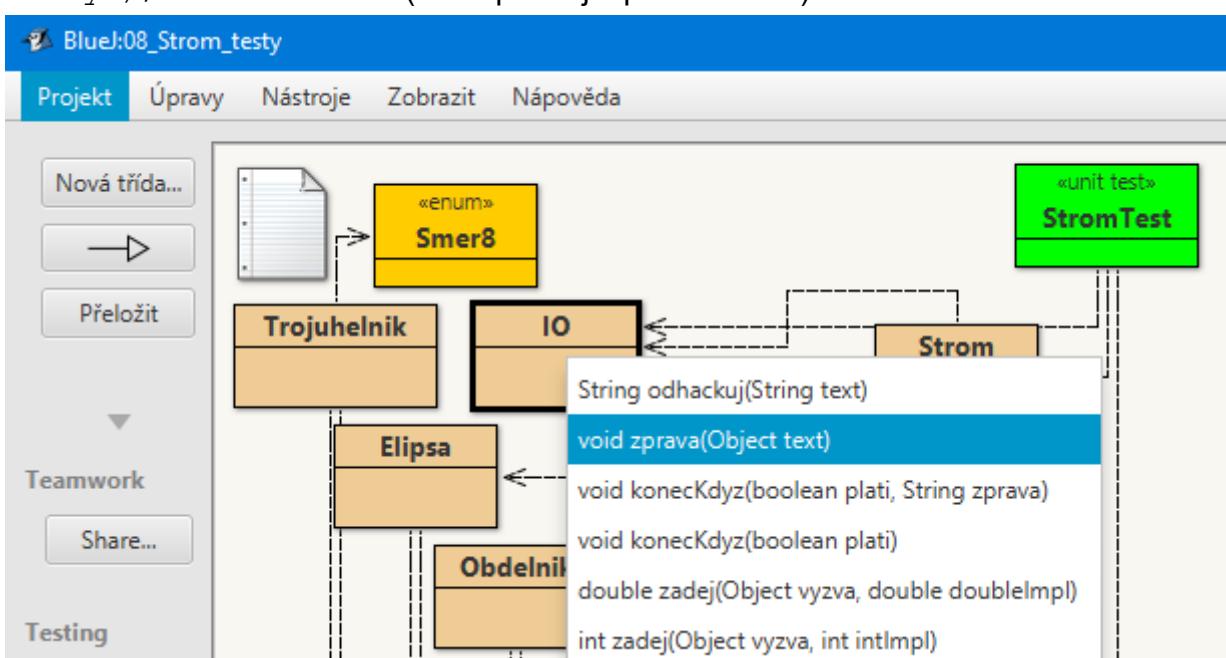
- při přidávání (tj. programování) metod je vhodné je ihned testovat
- do testů můžeme vložit čekání na událost od myši – potvrzení dosavadního průběhu

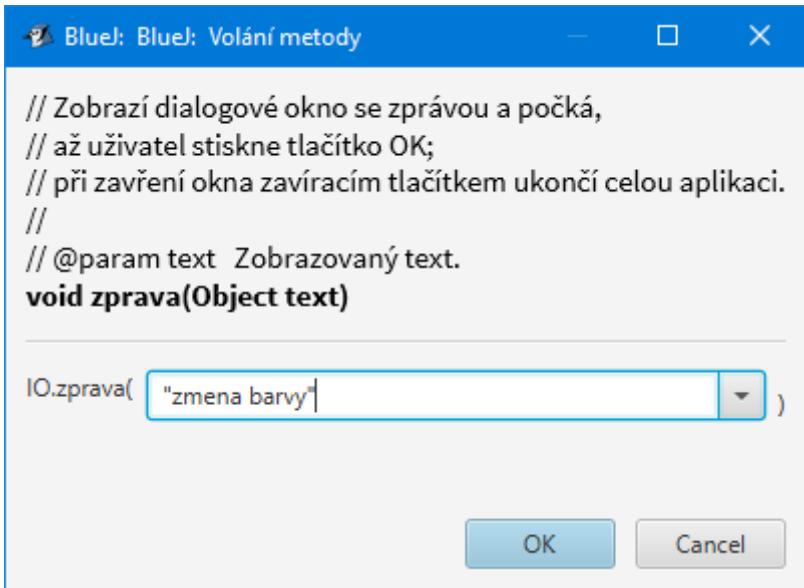
- např. je vhodné vložit do kódu testovacího přípravku zprávu, že jsou instance připraveny pomocí `IO.zprava()`

```
@Before
public void setUp() {
    strom1 = new Strom();
    strom2 = new Strom(100, 100);
    strom3 = new Strom(200, 150, Barva.ZLUTA);
    IO.zprava("Stromy vytvořeny");
}
```

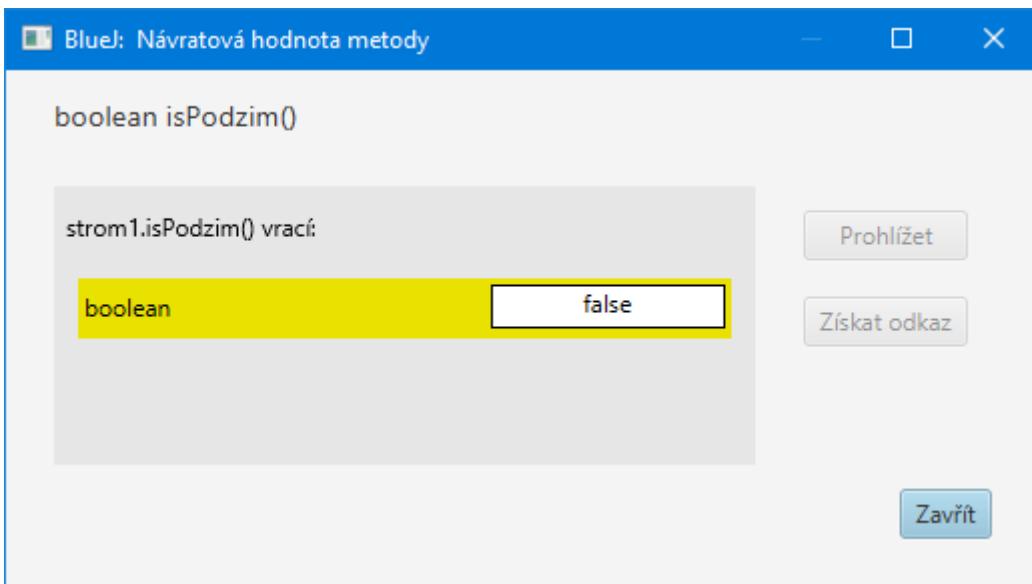


- podobnou zprávu je vhodné vložit i mezi jednotlivé příkazy testovací metody – `IO.zprava ("změna barvy")`; buď interaktivně (nebo později úpravou kódu)

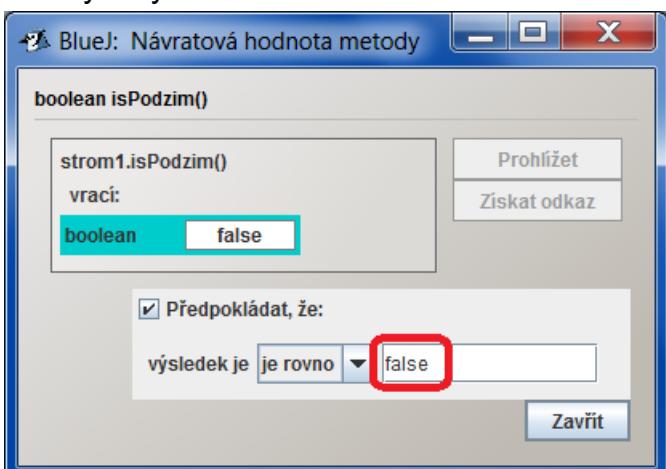




- voláme-li při interaktivních pokusech metodu, která vrací hodnotu, např. `isPodzim()`, zobrazí BlueJ okénko s aktuální návratovou hodnotou



- voláme-li při též metodu při záznamu testů, zobrazí BlueJ okénko s aktuální návratovou hodnotou a možným vyhodnocením testu



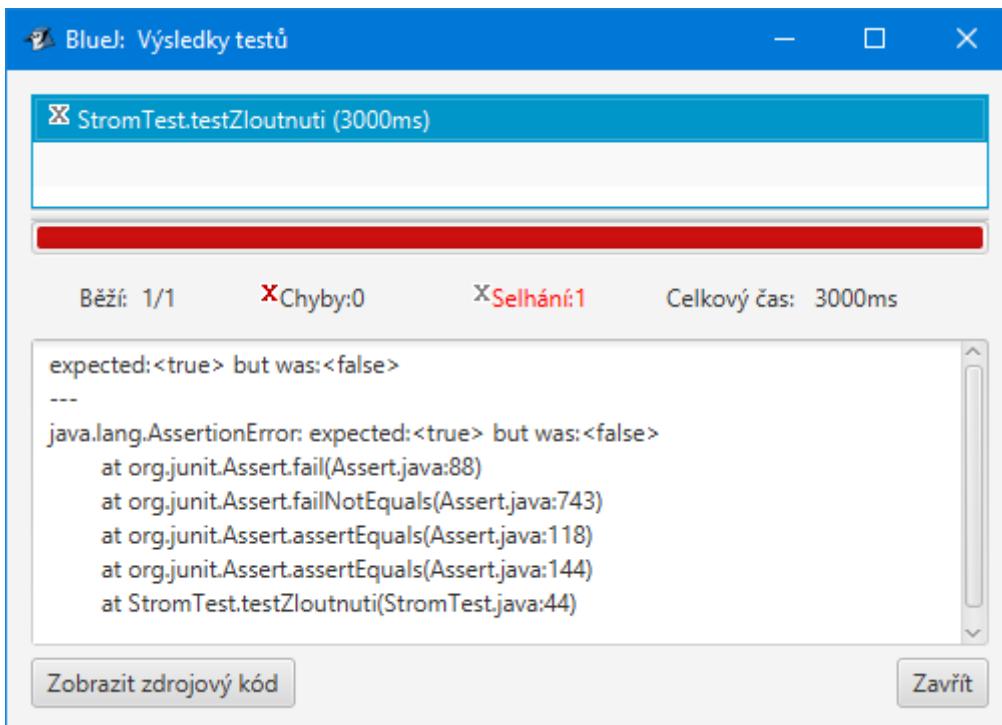
- očekáváme-li, že hodnota bude `false` (strom je zelený), doplníme tuto hodnotu a v testu bude následně ověřena – generovaný kód je `assertEquals(false, strom1.isPodzim());`

```

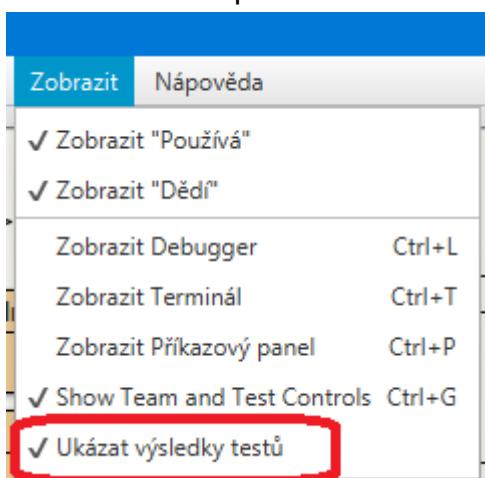
@Test
public void testZloutnuti() {
    assertEquals(false, strom1.isPodzim());
    strom1.zezloutni();
    assertEquals(true, strom1.isPodzim());
    IO.zprava("změna barvy");
    strom2.postupneZezloutni();
    IO.zprava("změna barvy");
    assertEquals(true, strom3.isPodzim());
}

```

- zadáme-li špatnou hodnotu (zde true), průběh testu selže a vypíše se:



- musíme mít ale před tím v hlavním okně BlueJ zapnuto zobrazování výsledků testů



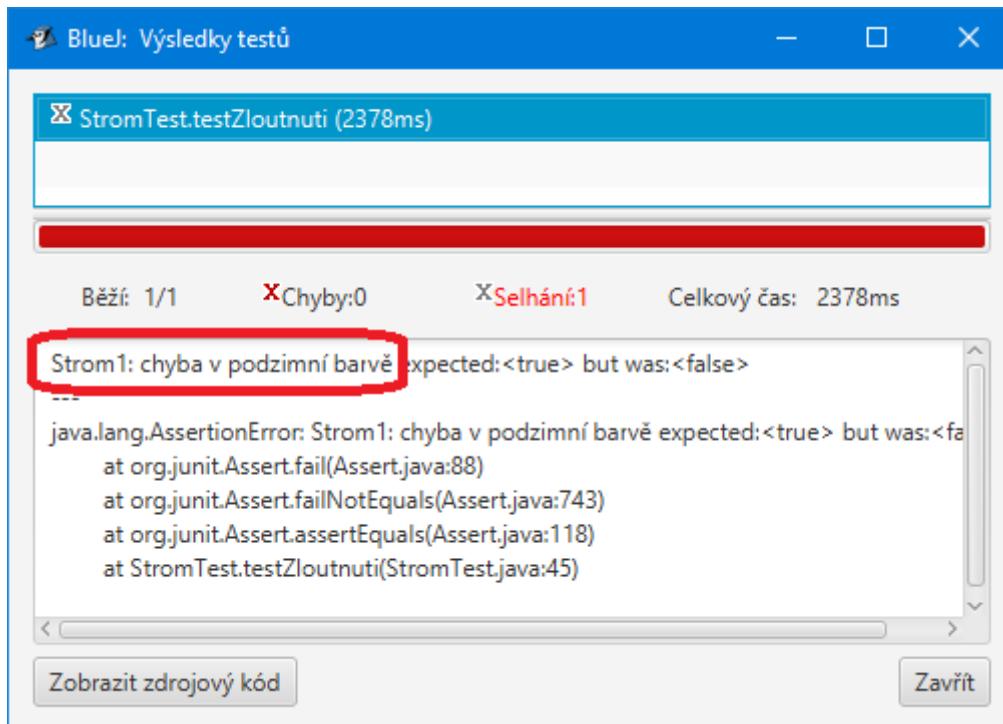
- metody `assertEquals()` mají přetíženou verzi, kdy prvním parametrem je řetězec s chybovou zprávou
  - je velmi vhodné tuto možnost využívat
  - ukázka předchozího testu

```

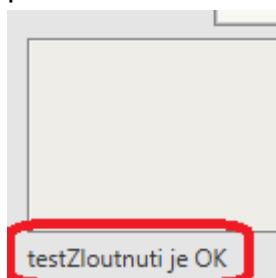
@Test
public void testZloutnuti() {
    // assertEquals(true, strom1.isPodzim());
    assertEquals("Strom1: chyba v podzimní barvě",
                true, strom1.isPodzim());
}

```

- vrátí-li `strom1.isPodzim()` špatnou hodnotu (zde `false`), průběh testu selže a vypíše se:



- proběhnou-li všechny testy v pořádku, na konci se v hlavním okně BlueJ vypíše



- toto lze zevšeobecnit na testy všech metod vracejících libovolnou hodnotu

- v `assertEquals()` je první hodnota očekávaná a druhá skutečná

```

@Test
public void testSouradniceX() {
    assertEquals(75, strom2.getX());
}

```

- opět je vhodné doplnit chybovým výpisem

```

@Test
public void testSouradniceX() {
    assertEquals("Chybná x souřadnice: ", 75, strom2.getX());
}

```

- tímto způsobem je vhodné připravit testovací metody pro všechny metody, které třída poskytuje

- v testech lze využívat naše vlastní metody – výhodné při opakovaných činnostech
  - metoda `zloutnutiJednohoStromu()` je `private`, protože je pomocná a není před ní anotace `@Test`

```
private void zloutnutiJednohoStromu(Strom strom) {  
    strom.zezloutni();  
    assertEquals("Strom: chyba v podzimní barvě", true, strom.isPodzim());  
    IO.zprava("změna barvy");  
}  
  
@Test  
public void testZloutnutiVsech() {  
    zloutnutiJednohoStromu(strom1);  
    zloutnutiJednohoStromu(strom2);  
    zloutnutiJednohoStromu(strom3);  
}
```

- v BlueJ lze samostatně testovat každou třídu – místní nabídka "Vytvořit testovací třídu"

- je sdružena s testovanou třídou

## 2.5. Rozhraní versus implementace

- současné programy (třídy) mají dvě podoby
  - vnější – reprezentovanou rozhraním
    - ◆ říká, co entita (program, třída, metoda, ...) umí a jak se s ní komunikuje
    - ◆ rozhraní pouze popisuje (slibuje), jak to bude fungovat
    - ◆ rozhraní je souhrnná informace, kterou potřebuje uživatel pro využívání entity
      - vnitřek entity ho nezajímá – od entity žádá provedení služby
    - ◆ tato informace má být uvedena v dokumentaci
  - vnitřní – reprezentovanou implementací
    - ◆ jak je zajištěno, aby entita splňovala sliby rozhraní
    - ◆ snahou je maximálně skrýt implementaci (`private` atributy apod.)
    - ◆ čím méně se o implementaci ví, tím jsou snažší její pozdější změny, ALE **rozhraní musí po svém zveřejnění zůstávat stejné**
      - pokud jej navrhнемe chybně a zveřejníme jej, je náprava velmi zdlouhavá a obtížná
      - ◆ jsou-li navenek známé detailly implementace (`public` atributy), může je někdo použít a při změně implementace je třeba ošetřit, že to neovlivnilo všechna předchozí použití – obtížné, někdy nemožné
- toto dělení na dva pohledy důsledně uplatňujeme i ve svých programech, kde máme vše pod kontrolou a vše využíváme jen my sami

- všechna porušení (zdůvodňovaná zjednodušeními) se dříve či později vymstí

## Poznámka

BlueJ ideálně využívá možností zobrazit u třídy pohled na implementaci (zdrojový kód)

```

1 public class Strom {
2
3     /** Počet doposud vytvořených instancí. */
4     private static int pocet = 0;

```

a na rozhraní (dokumentaci)

**Class Strom**

```

java.lang.Object
└ Strom

public class Strom
extends java.lang.Object

```

**Constructor Summary**

## Poznámka

Pro možnost fyzického oddělení rozhraní a implementace ve zdrojovém kódu existuje v Javě konstrukce interface (rozhraní) – viz později

### 2.5.1. Signatura versus kontrakt

- rozhraní můžeme rozdělit na dvě složky:
  - signatura – charakteristiky, které může při použití zkontolovat překladač

- ◆ dostupnost dané entity (public, private, ...) a její název
- ◆ u metod navíc seznam typů formálních parametrů a typ návratové hodnoty
- ◆ u atributů jejich typ, ...
- kontrakt – specifikuje informace, které překladač zkontoval nedokáže
  - ◆ podmínky, které je třeba dodržet, např. že souřadnice obrazce jsou souřadnicemi levého horního rohu
  - ◆ možné vedlejších efekty funkcí, např. že konstruktory grafických tvarů zabezpečí případné vytvoření plátna
  - ◆ implementační detaily, např. informaci o extrémním množství potřebné paměti
  - ◆ další důležitá sdělení, např. verze JDK
- kontrakt je dohoda mezi tvůrcem třídy či metody a jejím uživatelem
- popis kontractu by měl být uveden v dokumentačních komentářích
- příklad pro metodu

```
*****
 * Změní barvu koruny na KHAKI, čeká půl vteřiny
 * a pak ji změní na podzimní barvu.
 */
public void postupneZezloutni() {
    setBarvaKoruny(Barva.KHAKI);
    IO.cekej(500);
    setBarvaKoruny(BARVA PODZIMU);
}
```

- co říká rozhraní-signatura:

- ◆ je přístupná z vnějšku (public)
- ◆ nic nevrací (void)
- ◆ jmenuje se postupneZezloutni
- ◆ nemá žádné parametry

- co říká rozhraní-kontrakt:

- ◆ korunu přebarví na barvu khaki
- ◆ nechá ji zobrazenou půl vteřiny
- ◆ korunu přebarví na podzimní barvu

- co říká implementace:

- ◆ k přebarvení používá svoji metodu setBarvaKoruny()

- ◆ půl vteřinové čekání zabezpečí pozastavením programu pomocí metody `cekej()` třídy `IO`, které předá počet 500 milisekund čekání
  - ◆ k přebarvení používá svoji metodu `setBarvaKoruny()`
- podobný příklad lze udělat pro celou třídu `Strom` – nejlepší informace z generované dokumentace, ve které je uvedeno to, co je v rozhraní–kontrakt
- to, co představuje rozhraní–signaturu, se do dokumentace dodá automaticky

## 2.6. Komentáře

- pro komentáře platí obecné pravidlo:

„Komentář v kódu nemá popisovat co příkaz dělá (to čtenář vidí), ale vysvětlovat jakou službu a proč příkaz či metoda či atribut poskytuje a proč v programu je.“

- jsou trojí

- řádkové `//` do konce řádky
  - ◆ vývojová prostředí dávají možnost zakomentovat blok
  - ◆ BlueJ – Upravit / Zakomentovat F8 a Upravit / Odkomentovat F7

```
// public void vratBarva() {
//     setBarva(barva);
// }
```

- blokové – cokoliv (i `//`) mezi `/*` a `*/` i na více řádek

```
/*
 public void vratBarva() {
     setBarva(barva);           // nastavení původní barvy
 }
*/
```

- dokumentační – `/**` a `*/` s použitím klíčových slov – viz dále

### 2.6.1. Dokumentační komentáře

- též „javadoc komentáře“
- nezbytná součást zdrojového kódu

#### Poznámka

Nenapíšeme-li žádný dokumentační komentář, `javadoc` vygeneruje do dokumentace informace z rozhraní–signatura.

- dokumentace je z nich automaticky generovaná (do HTML) pomocí nástroje `javadoc.exe`

- v BlueJ je `javadoc.exe` integrovaný, takže pohled na dokumentaci je přímo přístupný z editoru (druhý pohled je implementace)
  - ◆ formát dokumentace se dá měnit (soubor doclet), takže může vizuálně vypadat i jinak, než dokumentace Java Core API
- dokumentace je tak samozřejmá, že je využívána pro další doplňkové akce – viz BlueJ interaktivní zasílání zpráv
- u jednoduchých příkladů se zdá, že dokumentace zdržuje a znepřehledňuje kód – snaha ji nepsat
  - zásadní chyba – kód je nutno zdokumentovat v okamžiku, kdy jej vytváříme – víme o něm nejvíce
- využívají se dokumentační komentáře (viz též [PPA1-32] a [PPA1-88])

```
/** toto je dokumentační komentář */
```

- je-li komentář na více řádcích, uvozují se tyto řádky (nepovinně) znakem \*

```
/** toto je dokumentační komentář
 * na více řádcích
 */
```

- musí být zdokumentováno vše, co má atribut `public` – třída, atributy, konstruktory, metody
  - je velmi vhodné zdokumentovat i vše ostatní, byť to standardně není do dokumentace generováno (rozšíření generace lze zajistit volbou přepínačů `javadoc.exe`)
  - dokumentace u `public` entit slouží pro popis kontraktu, dokumentace u `private` slouží pro autora kódu nebo pro příštího programátora, který bude kód upravovat

„Dobrý programátor věří, že jeho programy jsou natolik dobré, že si zaslouží budoucí vývoj.“

## Varování

Dokumentační komentář musí být v kódu uveden bezprostředně před dokumentovanou entitou – problém zejména při dokumentování tříd.

- pro zvýšení přehlednosti lze v textech komentářů používat běžné HTML značky, zejména `<p>`, `<br>`, `<ol>`, `<ul>`, `<li>`, `<i>`, `<b>`
  - nepoužíváme znaky `>` a `<` např. `x >= 0` ale jejich nahradily `&gt;`; `a &lt;`; např. `x &gt;= 0`
- dále se používají speciální značky pro `javadoc` komentáře, které začínají znakem @
  - `@author` – autor kódu
  - `@version` – verze kódu
  - `@param` – popis jednoho parametru metody
  - `@return` – popis návratové hodnoty metody
  - `@throws` – výjimka vyhazovaná metodou

- {@code název} – název entity, který bude formátován neproporcionálním písmem (jako <code>název</code>)
- a další – viz v dokumentaci k javadoc.exe

## ■ komentování třídy

```
/*
 * Třída {@code Barva} definuje skupinu základních barev
 * pro použití při kreslení tvarů.
 * Není definována jako výčtový typ,
 * aby uživatel mohl libovolně přidávat vlastní barvy.
 *
 * @author Rudolf PECINOVSKÝ
 * @version 1.09.2629 – 2011-01-03
 */
public class Barva {
```

## ■ komentování atributu

```
/** Černá = RGBA( 0, 0, 0, 255); */
public static final Barva CERNA = new Barva(Color.BLACK, "cerna");
```

## ■ komentování metody

```
/*
 * Vrátí instanci neprůhledné barvy se zadanými barevnými složkami
 * a zadným názvem. Pokud takováto barva neexistuje, vytvoří ji.
 * Existuje-li barva se zadaným názvem ale jinými složkami, anebo
 * existuje-li barva se zadanými složkami, ale jiným názvem,
 * vyhodí výjimku {@link IllegalArgumentException}.
 *
 * @param red      Velikost červené složky
 * @param green    Velikost zelené složky
 * @param blue     Velikost modré složky
 * @param nazev    Název vytvořené barvy
 *
 * @return Barva se zadaným názvem a velikostmi jednotlivých složek
 * @throws IllegalArgumentException má-li některé ze známých barev některý
 *                   ze zadaných názvů a přitom má jiné nastavení barevných složek
 *                   nebo má jiný druhý název.
 */
public static Barva getBarva( int red, int green, int blue, String nazev )
{
    return getBarva( red, green, blue, 0xFF, nazev );
}
```

## Poznámka

Dodatečně komentovat stovky řádek již hotového nekommentovaného kódu je ubíjející. Mnohem lepší je komentovat průběžně.

## 2.6.1.1. Rozmístování jednotlivých částí třídy

- kód třídy doplněný o komentáře je většinou poměrně dlouhý
- abychom se v něm vyznali, je dobré dodržovat jednotnou strukturu částí třídy
- jednotlivé části třídy (zatím jen data a metody) se totiž dají vnitřně dále členit
- existuje šablona pro třídu – použije se po Nová třída / Standardní třída a má obsah:

```
/* UTF-8 encoding: Příliš žluťoučký kůň úpěl čábelské ódy. */

/*
 * Instance třídy {@code Sablona} představují ...
 *
 * @author jméno autora
 * @version 0.00.0000 - 20yy-mm-dd
 */
public class Sablona {
    //== KONSTANTNÍ ATRIBUTY TŘÍDY =====

    //== PROMĚNNÉ ATRIBUTY TŘÍDY =====

    //== STATICKÝ INICIALIZAČNÍ BLOK =====

    //== KONSTANTNÍ ATRIBUTY INSTANCÍ =====

    //== PROMĚNNÉ ATRIBUTY INSTANCÍ =====

    //######
    //== KONSTRUKTORY A TOVÁRNÍ METODY =====

    /**
     */
    public Sablona() {
    }

    //#####
    //== PUBLIC METODY TŘÍDY =====

    //== PRIVATE METODY TŘÍDY =====

    //== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCÍ =====

    //== PUBLIC METODY INSTANCÍ =====

    //== PRIVATE METODY INSTANCÍ =====
}
```

- tato šablona se z počátku zdá být příliš složitá, ale poskytuje dobrý základ pro rozmístování kódu
  - už jen to, že musíme trochu přemýšlet, do jaké části náš kód umístíme, znamená, že provedeme jeho analýzu

## ■ ukázka dosavadní třídy Strom (zde Komentovaný Strom)

```
/*
 * Instance třídy {@code KomentovanýStrom} představují ukázkovou třídu
 * pro výklad komentářů
 *
 * @author Pavel Herout
 * @version 2.00.000
 */
public class KomentovanýStrom
{
    //== KONSTANTNÍ ATRIBUTY TŘÍDY =====

    /** standardní barva kmene stromu */
    private static final Barva BARVA_KMENE = Barva.HNEDA;

    /** implicitní barva koruny stromu */
    private static final Barva IMPL_BARVA_KORUNY = Barva.ZELENA;

    /** barva koruny stromu na podzim */
    private static final Barva BARVA PODZIMU = Barva.ZLUTA;

    //== PROMĚNNÉ ATRIBUTY TŘÍDY =====

    /** Počet doposud vytvořených instancí. */
    private static int pocet = 0;

    //== STATICKÝ INICIALIZAČNÍ BLOK =====
    //== KONSTANTNÍ ATRIBUTY INSTANCI =====

    /** Pořadí kolikátá byla daná instance vytvořena v rámci třídy. */
    private final int PORADI = ++pocet;

    /** Název instance sestávající z názvu třídy a POŘADÍ instance */
    private final String NAZEV = "Strom_" + PORADI;

    /** koruna stromu */
    private final Elipsa koruna;
    /** kmen stromu */
    private final Obdelník kmen;

    //== PROMĚNNÉ ATRIBUTY INSTANCI =====

    /** x souřadnice levého horního rohu stromu
     * ve skutečnosti zbytečná, je použita jen pro ukázkou {@code this}
     */
    private int x;

    //#####
    //== KONSTRUKTORY A TOVÁRNÍ METODY =====
```

```

*****  

 * Implicitní konstruktor vytvoří v levém horním rohu plátna  

 * instanci širokou 100 bodů, vysokou 150 bodů  

 * s {@code BARVA_KMENE} a {@code IMPL_BARVA_KORUNY}  

 */  

public KomentovanyStrom() {  

    this(0, 0);  

}  

*****  

 * Vytvoří na definovaných souřadnicích (levý horní roh)  

 * instanci širokou 100 bodů, vysokou 150 bodů  

 * s {@code BARVA_KMENE} a {@code IMPL_BARVA_KORUNY}  

 *  

 * @param x x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna  

 * @param y y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna  

 */  

public KomentovanyStrom(int x, int y) {  

    this(x, y, IMPL_BARVA_KORUNY);  

}  

*****  

 * Vytvoří na definovaných souřadnicích (levý horní roh)  

 * instanci širokou 100 bodů, vysokou 150 bodů  

 * s {@code BARVA_KMENE} a zadanou barvou koruny  

 * jako vedlejší efekt uloží x-souřadnici - pouze pro ukázku použití {@code ►  

this}  

 *  

 * @param x x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna  

 * @param y y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna  

 * @param barvaKoruny barva koruny  

 */  

public KomentovanyStrom(int x, int y, Barva barvaKoruny) {  

    koruna = new Elipsa(x, y, 100, 100, barvaKoruny);  

    kmen = new Obdelnik(x + 45, y + 100, 10, 50, BARVA_KMENE);  

    this.x = x;  

}  

// #####  

//== PUBLIC METODY TŘÍDY =====  

//== PRIVATE METODY TŘÍDY =====  

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCÍ =====  

*****  

 * Vrátí x-ovou souřadnici pozice instance.  

 *  

 * @return x-ová souřadnice.  

 */  

public int getX() {  

    return this.x;
}

```

```

}

/******************
 * Vrátí y-ovou souřadnici pozice instance.
 *
 * @return y-ová souřadnice.
 */
public int getY() {
    return koruna.getY();
}

/******************
 * Vrátí název instance, tj. název její třídy následovaný POŘADÍM.
 *
 * @return Řetězec s názvem instance.
 */
public String getNazev() {
    return NAZEV;
}

/******************
 * Nastaví novou barvu koruny.
 *
 * @param nová Požadovaná nová barva.
 */
public void setBarvaKoruny(Barva barva) {
    koruna.setBarva(barva);
}

/******************
 * Vrátí výšku instance.
 *
 * @return Výška instance v bodech
 */
public int getVyska() {
    return koruna.getVyska() + kmen.getVyska();
}

/******************
 * Vrátí informaci o tom, zda má koruna podzimní korunu.
 * Pojmenování je nevhodné - ukázka chybné analýzy.
 *
 * @return {@code true}, pokud má koruna podzimní barvu, jinak {@code false}
 */
public boolean isPodzim() {
//    return koruna.getBarva() == BARVA PODZIMU;

//    Barva barva = koruna.getBarva();
//    boolean jePodzimni = barva.equals(BARVA PODZIMU);
//    return jePodzimni;
    return koruna.getBarva().equals(BARVA PODZIMU);
}

```

```

}

//== PUBLIC METODY INSTANCÍ =====
*****  

* Změní barvu koruny na podzimní barvu.  

*/
public void zezloutni() {
    setBarvaKoruny(BARVA PODZIMU);
}

*****  

* Změní postupně barvu koruny na podzimní barvu  

* <br/>
* nejprve použije KHAKI barvu a pak po půl vteřině finální ZLUTA
*/
public void postupneZezloutni() {
    setBarvaKoruny(Barva.KHAKI);
    IO.cekej(500);
    setBarvaKoruny(BARVA PODZIMU);
}

*****  

* Vykreslí obraz své instance na plátno.
*/
public void nakresli() {
    koruna.nakresli();
    kmen.nakresli();
}

*****  

* Smaže obraz své instance z plátna.
*/
public void smaz() {
    koruna.smaz();
    kmen.smaz();
}

*****  

* Převede instanci na řetězec obsahující název třídy, POŘADÍ instance,  

* její souřadnice, výšku a barvu koruny.  

*
* @return Řetězcová reprezentace dané instance.
*/
@Override
public String toString() {
    return getNazev() + ": x=" + getX() + ", y=" + getY() +
           ", výška=" + getVyska() +
           ", barva=" + koruna.getBarva().getNazev();
}

//== PRIVATE METODY INSTANCÍ =====
}

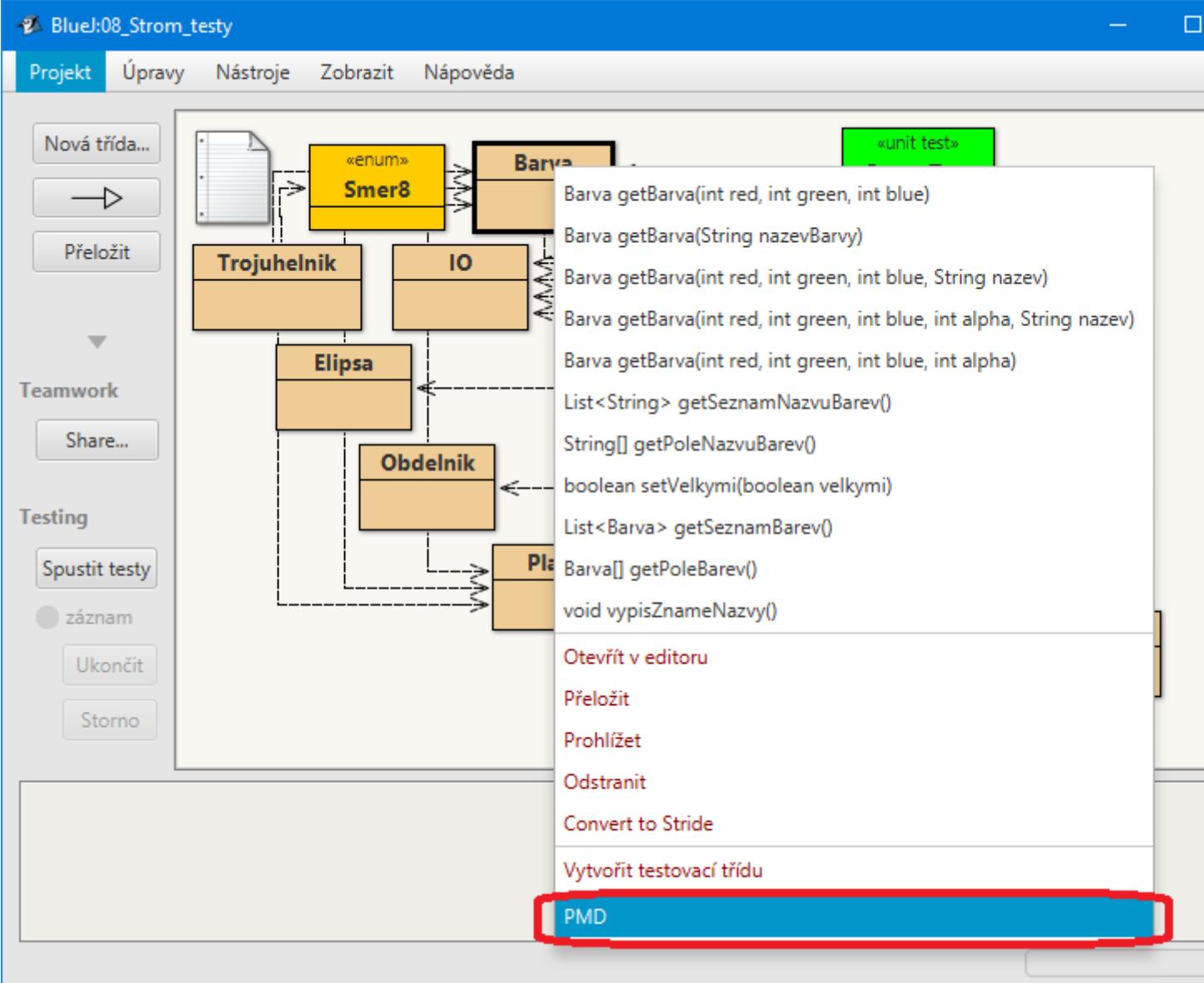
```

## Poznámka

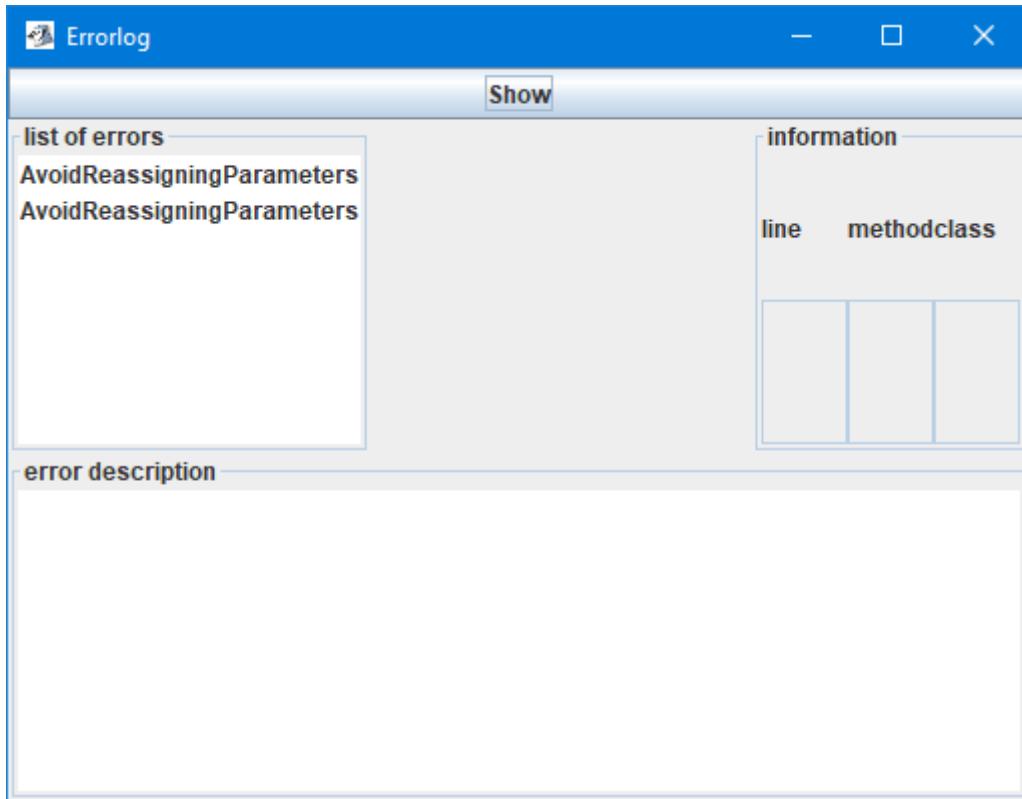
Používání této šablony není (narozdíl od dokumentování kódu) nutností. Je však vhodné si uvědomit, že nám pomáhá udržovat pořádek.

## 2.7. Kvalita kódu

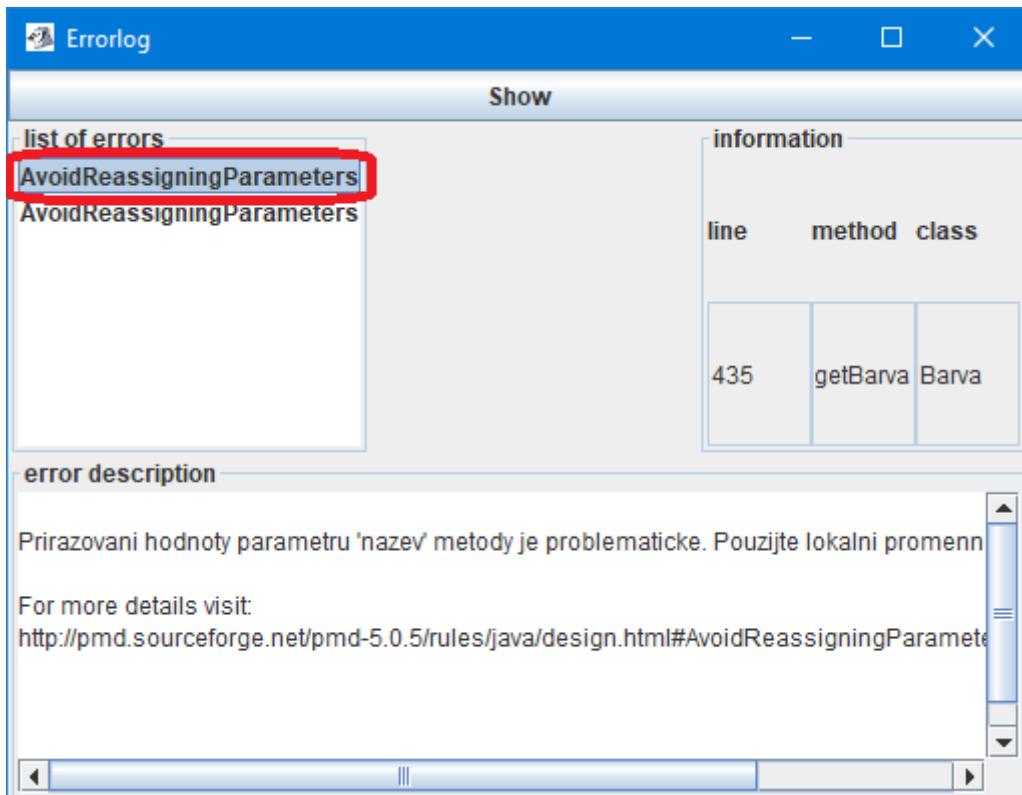
- statická analýza kódu
- poměrně nová, tzn. neznámá a málo používaná věc
- nezkoumáme, zda je kód funkční (na to jsou JUnit testy a další typy testů), ale zda je „hezký“
  - pro začátečníky ještě kontroverznější požadavek, než dokumentace
  - poměr užitečnosti / obtěžování = analogie s dokumentací zdrojového kódu
  - pokud kód není „hezký“, znamená to v budoucnu téměř jistě potenciální problém
- existuje několik podobných nástrojů (např. Checkstyle, FindBugs), my budeme využívat PMD
  - pmd.sourceforge.net
  - konfigurovatelný nástroj, který z počátku obtěžuje, ale ve svém důsledku výrazně pomáhá
  - dokáže zkонтrolovat řádově několik set známých problémů – počet lze v konfiguraci omezit
- PMD lze použít samostatně (z příkazové řádky), ale i zaintegrovat do BlueJ, Eclipse, ...
- integrace do BlueJ
  - do adresáře `BlueJ\lib\extensions` nahrát soubory
    - ◆ `PMDExtension.jar` – PMD plugin do BlueJ
    - ◆ `pmdrules.xml` – XML soubor s definicí kontrolovaných věcí a s českou návodou
  - pak z hlavního okénka BlueJ z místní nabídky třídy zvolit PMD / Zkontrolovat kód



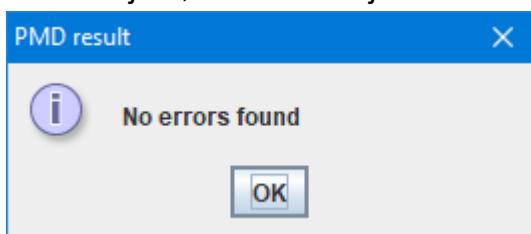
- dostaneme např.:



- a po rozkliknutí chyby dostaneme informaci o místě jejího výskytu a její stručný popis, např.:



- samozřejmě, ideální stav je:



■ PMD je velmi striktní – zásada: v reálném životě nemusíme odstraňovat všechny reportované problémy, ale ty, které ponecháme, musíme mít dobře zdůvodněné

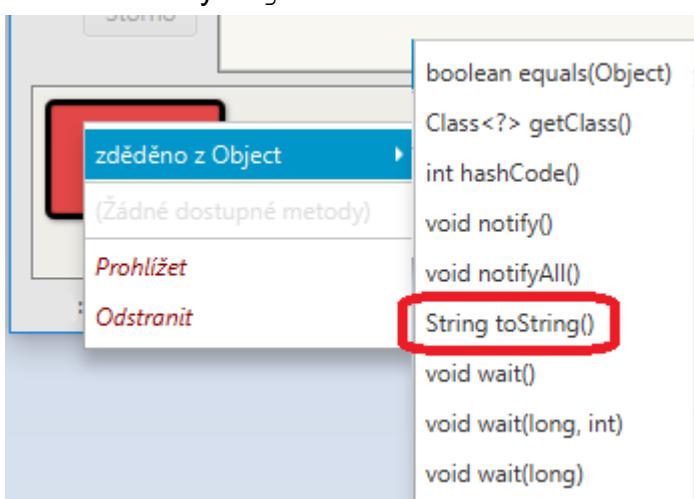
- pokud víme, že je hlášení problému zbytečné / nepodstatné, lze jej editací souboru `pmdrules.xml` potlačit
- pokud není soubor `pmdrules.xml` korektní, PMD není z BlueJ vůbec dostupné

## Varování

V předmětu KIV/OOP používaná konfigurace PMD kontroluje jen zcela základní věci. Protože je začleněna do kontrol validátoru, je nutné všechny hlášené problémy **ve svém zdrojovém kódu** odstranit, protože jinak úloha neprojde validací.

## 2.8. Společný předek Object

- všechny objekty v Javě mají společné vlastnosti, které jsou zahrnuté ve třídě `Object`
- jakákoli třída – ať knihovní nebo námi vytvořená
  - je dceřiná třída (třída potomka)
  - dědí od třídy `Object`



- `Object` představuje libovolný objekt, který může být v budoucnosti nahrazen objektem jiného typu
  - např. má-li metoda jako svůj parametr `Object` (ve třídě `IO` metoda `void zprava (Object text)`), může být tato metoda volána se skutečným parametrem typu libovolného objektu (`IO.zprava ("žetezec")` nebo `IO.zprava (Barva.MODRA)`)
- zděděné metody z `Object`
  - pokud nám nevyhovují, můžeme některé z nich překrýt, tj. napsat si vlastní verze
    - ◆ překrytí je odlišné od přetížení
    - ◆ v přetížení se metody jmenovaly stejně, ale měly rozdílné formální parametry
    - ◆ v překrytí musí hlavička metody (signatura) zcela souhlasit s hlavičkou původní metody
  - je jich celkem 9, prakticky v začátcích využijeme / překrýváme dvě:

- ◆ `String toString()` – vrací textový řetězec popisující stav třídy
    - v `Object` vrací pouze jméno třídy a hešovací kód (téměř nepoužitelné)
  - ◆ `boolean equals(Object o)` – porovnávací metoda, která vrací `true`, když jsou si objekty rovny
    - v `Object` je „přísná“ – objekty se rovnají, pouze jsou-li totožné, tzn. vůbec neuvažuje stavy (obsah atributů) objektů
    - podrobnosti viz později
- překrýváme-li metody, je vhodné před nimi uvést anotaci `@Override`
- informujeme tím překladač, že skutečně chceme provést akci překrytí
  - v případě překlepů nebo jiných omylů nás překladač upozorní, že naše metoda má odlišnou hlavičku a došlo by tedy v lepším případě pouze k přetížení metod
- při překrývání je nutné dodržet modifikátor přístupu `public`
- bez jeho uvedení je hlášena chyba překladu: *attempting to assing weaker access privileges; was public*

## 2.8.1. Řetězce a metoda `toString()`

### Poznámka

Úvod do řetězců viz [PPA1-138].

- texty v uvozovkách "Ahoj" jsou jedním z příkladů řetězců – třída `String`
- řetězce se dají spojovat do jednoho pomocí operátoru +
- ```
"Ahoj " + "lidi"
```
- Java má vlastnost, že automaticky převádí hodnoty primitivních datových typů na řetězce
- "Počet = " + pocet  
bude `Počet = 10`
- pro „hodnoty“ objektů platí totéž, a tato služba je zajišťována voláním metody `String toString()`, kterou každá třída dědí od třídy `Object`
- pokud není překryta, vrací jméno třídy, @ a hešovací kód třídy (později), např. `Strom@140fee`
- při překrývání se držíme konaktu, že metoda vrací řetězec, který z pohledu autora třídy co nejlépe popisuje stav instance
- „vrácený řetězec je stručnou a informativní reprezentací, které uživatel snadno porozumí“ – obecný kontrakt z Java Core API
  - jsou v něm vlastnosti třídy (atributy + další informace)

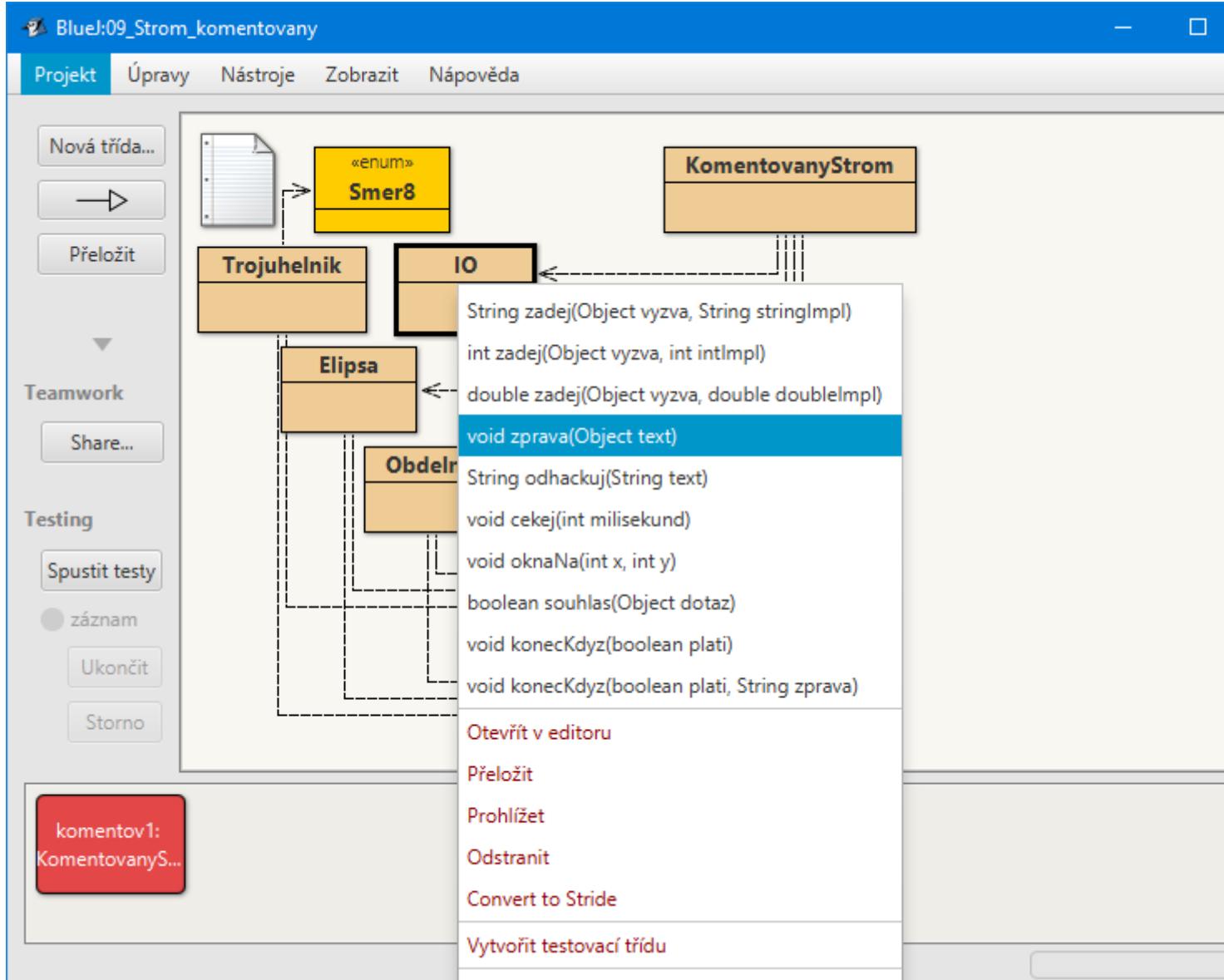
- velmi často řetězec začíná jménem třídy a pořadovým číslem instance – metoda `getNazev()`

```
/*
 * Převede instanci na řetězec obsahující název třídy, pořadí instance,
 * její souřadnice, výšku a barvu koruny.
 *
 * @return Řetězcová reprezentace dané instance.
 */
@Override
public String toString() {
    return getNazev() + ": x=" + getX() + ", y=" + getY() +
        ", výška=" + getVyska() +
        ", barva=" + koruna.getBarva().getNazev();
}
```

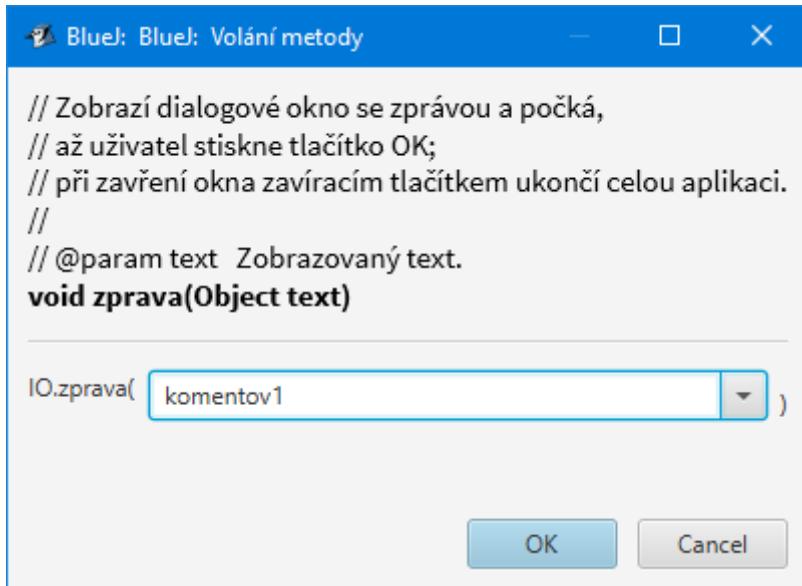
řetězec má např. obsah: Strom\_1: x=0, y=0, výška=150, barva=zelena

- s existencí metody `toString()` všichni počítají (je již v `Object`, takže určitě existuje)

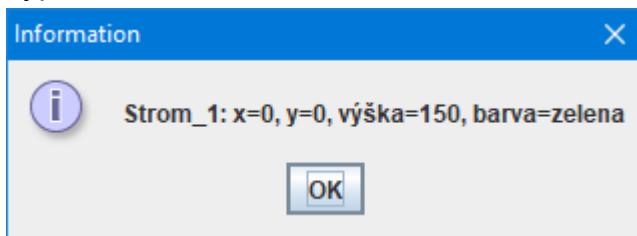
- v BlueJ ji můžeme zavolat např. pomocí metody `zprava()` ze třídy `IO`



- jako parametr zprávy dáme odkaz na náš objekt



♦ vypíše:



## 2.8.2. Metoda getClass ()

- vrací class-objekt třídy dané instance
  - class-objekt třídy je instance třídy Class
  - je v něm vždy skutečný typ třídy – bez ohledu na přetypování při dědění, použití rozhraní atd. – viz dále
- tento objekt lze využít k mnoha sofistikovaným akcím – za běhu programu lze získat úplnou informaci o dané třídě:
  - konstruktory
  - metody
  - modifikátory, atd.
- pro naše účely zatím postačí metody:
  - String getSimpleName () – vlastní název třídy (tj. bez balíčků) instance
  - String getName () – úplný název třídy instance včetně případných balíčků
- class-objekt třídy lze získat i jako statickou konstantu třídy Třída.class
- metoda getClass () má modifikátor final, takže ji nelze překrýt (např. toString () lze překrýt)
- příklad prázdné třídy Prázdná

```
public class Prázdná {  
}
```

■ zjištění informací o třídě:

```
public static void main(String[] args) {  
    Prázdná pr = new Prázdná();  
    System.out.println(pr.getClass().getSimpleName());  
  
    Class classPrZObjektu = pr.getClass();  
    Class classPrZeTřídy = Prázdná.class;  
    if (classPrZObjektu.equals(classPrZeTřídy)) {  
        System.out.println("Třídy " + classPrZObjektu +  
                           " a " + classPrZeTřídy +  
                           " jsou stejné");  
    }  
}
```

■ vypíše:

```
Prázdná  
Třídy class Prázdná a class Prázdná jsou stejné
```

# Kapitola 3. Návrhové vzory a rozhraní

## 3.1. Jednoduché návrhové vzory

- *design patterns*
- obdoba matematických vzorečků – časem prověřená řešení typických programátorských požadavků
  - předpisy, do kterých se „dosazují“ třídy a objekty
- výhody
  - rychlé řešení – nevymýslí se, pouze se použije
  - vyzkoušené řešení – malá pravděpodobnost, že uděláme chybu
  - řešení každý rozumí – zjednodušená komunikace mezi členy týmu a dokumentace
- prvotní zdroj GoF (*Gang of Four*) Gamla et al.: *Design Patterns*, 1995
- většina vzorů je jednoduchých, jen muselo někoho napadnout to sepsat
- v současné době je znalost NV naprostá nutnost pro profesionálního programátora
  - je výhodné je správně používat již od začátku programování

### Poznámka

Některé z dálé uváděných NV nepatří do základní skupiny NV z GoF, ale pro jejich obecnou známost a užitečnost se do NV počítají.

### 3.1.1. Knihovní třída (*Utility class*)

- „schránka“ na statické metody a atributy (zejména konstanty) – třída `IO` v BlueJ, `Math`, `System`, `Arrays` v Java Core API
  - typické použití je tam, kde se jako služba požaduje nějaká jednorázová transformace dat
    - ◆ např. výpočet hodnoty sinusu, výstup na konzoli atp.
- pro funkčnost nepotřebuje vytvářet žádné instance, veškerá činnost se odehraje pomocí statických metod a konstant
  - aby nešlo udělat instanci (pokud by to někoho napadlo), udělá se prázdný privátní konstruktor
  - navíc se třída označí jako `final`, aby bylo jasné (zejména pro překladač), že nejde zdědit
  - poskytování služeb (volání metod) je velmi rychlé – nemusí se před tím vytvářet instance
- tato třída se používá velmi často i v běžných projektech
  - typicky je umístěna někde v kořenovém balíku a jmene se `Konstanty.java`

- ukládají se do ní všechny konstanty platné pro celou aplikaci, např. jména fontů, adresáře, defaultní hodnoty apod.
- plus případné statické metody pracující s těmito konstantami

## ■ příklad

```
/**
 * Ukázka konstrukce knihovní třídy
 * @author P.Herout
 */
public final class KnihovniTrida {

    /**
     * privátní konstruktor, aby nebylo možné vytvořit instanci třídy
     */
    private KnihovniTrida() {}

    /** Ludolfovo číslo */
    public static final double PI = Math.PI;

    /**
     * Vypočítá obvod kruhu o zadaném poloměru
     *
     * @param polomér poloměr kruhu
     * @returns obvod kruhu
     */
    public static double vypočtiObvodKruhu(double poloměr) {
        return 2 * PI * poloměr;
    }

    /**
     * Vypočítá obsah kruhu o zadaném poloměru
     *
     * @param polomér poloměr kruhu
     * @returns obsah kruhu
     */
    public static double vypočtiObsahKruhu(double poloměr) {
        return PI * poloměr * poloměr;
    }

    ///////////////////////////////////////////////////
    /**
     * Jen pro testovací účely
     */
    public static void main(String[] args) {
        double r = 1.0;
        System.out.println("Obvod pro r = " + r + " je: " +
                           vypočtiObvodKruhu(r));
        System.out.println("Obsah pro r = " + r + " je: " +
                           vypočtiObsahKruhu(r));
    }
}
```

- použití v jakékoliv jiné třídě je:

```
double r = 1.0;
System.out.println("Obvod pro r = " + r + " je: " +
                    KnihovniTrida.vypočtiObvodKruhu(r));
System.out.println("Obsah pro r = " + r + " je: " +
                    KnihovniTrida.vypočtiObsahKruhu(r));
```

### 3.1.2. Statická tovární metoda (*Static factory method*)

- též se nazývá **jednoduchá tovární metoda** (*simple factory method*)
- statická metoda, která vrací odkaz na instanci své třídy
- používá se ke stejnemu účelu, jako konstruktor `new()`
- třída může znepřístupnit konstruktor pomocí `private`, ale narození od knihovní třídy dokáže / chce vytvářet své instance
- výhody:
  - může se jmenovat libovolně významově – `Osoba.getVysokastihla()` (narození od konstruktora, který má přesně vyhrazené jméno)
  - sama se rozhodne, zda vnitřně opravdu vytvoří novou instanci nebo použije už nějakou hotovou instanci ze svého vnitřního kontejneru (např. `Barva.getBarva()`)
  - může vracet objekt libovolného podtypu svého návratového typu – API může poskytovat objekty, aniž by zveřejňovalo jejich třídy
    - ◆ příklad – `java.util.Collections` má 20 pomocných implementací svých kolekcí (`SynchronizedMap`, `UnmodifiableMap`, ...)
    - ◆ zveřejnění všech by znamenalo velký a zbytečný nárůst API – stačí jedna třída (tj. `Collections`), která je umí pomocí statické tovární metody vytvářet
- nevýhody
  - tyto metody se nedají plnohodnotně dědít
    - ◆ to je ale spíše výhoda, protože dědění se obecně vyhýbáme a nahrazujeme je kompozicí (skládáním)
  - nejasné pojmenování – metodu nelze na první pohled odlišit od ostatních metod, ale typické názvy:
    - ◆ `getInstance()` – nejběžnější název
    - ◆ `getNázevTřídy()` – další používaná konvence: `Barva.getBarva()`
    - ◆ `valueOf()` – vrací instanci, která má „stejnou“ hodnotu, jako její parametry – typicky u `String`, kde `String.valueOf(123)` vrátí řetězec "123"
- příklad kódu: `Barva.getBarva(255, 255, 255)` nebo dále Jedináček

### 3.1.3. Jedináček (*Singleton*)

- třída s vždy jedinou instancí, např. kreslicí plátno, připojení do databáze, otevření I/O proudu
  - typický příklad Platno
- z nepřístupný konstruktor pomocí `private`
- pro získání odkazu používá statickou tovární metodu, která pokaždé vrací odkaz na stejnou instanci
  - odkazů může být samozřejmě více, ale vždy na jedinou instanci
- příklad

```
/**  
 * Ukázka konstrukce jedináčka Elvis  
 * @author P.Herout  
 */  
  
public class Elvis {  
  
    /** Jediná instance Elvis - opravdový Elvis je jen jeden */  
    private static final Elvis INSTANCE = new Elvis();  
  
    /*****  
     * Bezparametrický konstruktor - je volán pouze jednou.  
     */  
    private Elvis() {  
        // zde případné akce vytvářející zdokonalení božského Elvis  
    }  
  
    /**  
     * Statická tovární metoda  
     * @return jedinou instanci jedinečného Elvis  
     */  
    public static Elvis getElvis() {  
        return INSTANCE;  
    }  
  
    //////////////////////////////////////////////////////////////////  
    /**  
     * Příklad služeb, které Elvis poskytuje  
     * @param jménoPísňě jméno písni, kterou zazpívá  
     */  
    public void zpívej(String jménoPísňě) {  
        System.out.println("Originální Elvis zpívá: " + jménoPísňě);  
    }  
}
```

- použití

```
public class PouzitiElvise {  
    public static void main(String[] args) {
```

```

        Elvis idol = Elvis.getElvis();
        idol.zpivej("Heartbreak Hotel");
    }
}

```

### 3.1.4. Přepravka (*Messenger*)

- používáme, pokud chceme, aby metoda přijímala a nebo vracela více hodnot najednou
  - Java neumožňuje předávání parametrů metod odkazem – nelze je v metodě trvale měnit
  - předání více hodnot najednou je pohodlné i v případě vstupních (formálních) parametrů
- princip je, že použijeme třídu, která má tolik atributů, kolik potřebujeme předat hodnot
- atributy jsou `public` (`nikoliv private`) kvůli snadnějšímu přístupu – výjimka z pravidla
  - ale getry se jim ze zvyku příší
  - k atributům je přístup
    - ◆ `odkazNaPřepravku.jménoAtributu`
    - ◆ `odkazNaPřepravku.getJménoAtributu()`
- atributy jsou označeny jako `final` (konstanty), což znamená, že se jim smí přiřadit hodnota pouze jednou – typicky v konstruktoru

#### Poznámka

Přesto, že jsou `final`, označují se malými písmeny, ne jako běžné konstanty – x nikoliv X. Další výjimka z pravidla.

- tím se přepravka stává **neměnným objektem (immutable)**
- výhodou je, že se na hodnoty v Přepravce lze po celou dobu jejího života spolehnout – po nastavení je již nikdo nemůže změnit

```

/*****
 * Instance třídy {@code Pozice} představují přepravky uchovávající
 * informace o pozici objektu.
 * Proto jsou jejich atributy deklarovány jako veřejné konstanty.
 *
 * @author Rudolf PECINOVSKÝ
 * @version 1.09.2629 – 2011-01-03
 */
public class Pozice
{
//== KONSTANTNÍ ATRIBUTY INSTANCI =====
    /** Vodorovná souřadnice. */
    public final int x;

    /** Svislá souřadnice. */
    public final int y;
}

```

```

//#####
//== KONSTRUKTORY A TOVÁRNÍ METODY =====

/*********************  

 * Vytvoří přepravku uchovávající zadané souřadnice.  

 *  

 * @param x  Vodorovná souřadnice  

 * @param y  Svislá souřadnice  

 */  

public Pozice( int x, int y )  

{  

    this.x = x;  

    this.y = y;  

}

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCI =====

/*********************  

 * Vrátí uloženou velikost vodorovné souřadnice.  

 *  

 * @return Požadovaná hodnota  

 */  

public int getX()  

{  

    return x;  

}

/*********************  

 * Vrátí uloženou velikost svislé souřadnice.  

 *  

 * @return Požadovaná hodnota  

 */  

public int getY()  

{  

    return y;  

}

//== OSTATNÍ NESOUKROMÉ METODY INSTANCI =====

/*********************  

 * Vrátí informaci o tom, představuje-li instance zadána v parametru  

 * stejnou pozici.  

 *  

 * @param o  Testovaná instance  

 * @return Je-li zadána instance pozici se stejnými hodnotami atributů,  

 *         vrátí {@code true}, jinak vrátí {@code false}.  

 */  

@Override  

public boolean equals( Object o )  

{  

    if( ! (o instanceof Pozice) ) {  

        return false;           //=====>
    }
}

```

```

    }
    Pozice p = (Pozice)o;
    return (p.x == x) && (p.y == y);
}

/******************
 * Vrací textovou reprezentaci (podpis) dané instance
 * používanou především k ladicím účelům.
 *
 * @return Podpis dané instance
 */
@Override
public String toString()
{
    return "Pozice[x=" + x + ", y=" + y + "]";
}
}

```

- typické příklady použití přepravek v Java Core API – `java.awt.Point`, `java.awt.Dimension`

### 3.1.4.1. Použití Přepravky

- doplníme do `Strom` metody

```

/******************
 * Přemístí strom na jinou pozici - levý horní roh.
 *
 * @param x nová x-souřadnice
 * @param y nová y-souřadnice
 */
public void setPozice(int x, int y) {
    koruna.setPozice(x, y);
    kmen.setPozice(x + (koruna.getŠířka() - kmen.getŠířka()) / 2,
                    y + koruna.getVýška());
}

public Pozice getPozice() {
    return new Pozice(getX(), getY());
}

public void setPozice(Pozice p) {
    this.setPozice(p.x, p.y);
}

```

- `Strom` sice pozici má, ale uloženou ve dvou oddělených atributech `x` a `y`
  - ty jsou navíc atributy koruny, tj. Elipsy
- když někdo `Strom o Pozici požádá (getPozice())`, musí se nejdříve vytvořit její nová instance
- volání `this.setPozice(p.x, p.y);` znamená volání původní nyní přetížené metody své vlastní instance (proto nepovinné `this.`)

- při testování pozor na skutečnost, že pozice není souřadnice středu, ale levého horního rohu

```

public class StromTest {
    Strom zelenyStrom;
    Strom zlutyStrom;

    @Before
    public void setUp()
    {
        zelenyStrom = new Strom();
        zlutyStrom = new Strom(150, 150, Barva.ZLUTA);
        IO.zprava("Stromy vytvořeny");
    }

    /*****
     * Prohodí pozice zadaných stromů a nechá zkontořovat,
     * zda si stromy své pozice doopravdy vyměnily.
     * První strom je nutné nechat na konci překreslit,
     * protože druhý strom jej při přesunu vymaže.
     *
     * @param strom1 první strom
     * @param strom2 druhý strom
     */
    private void pomProhodPozice(Strom strom1, Strom strom2) {
        Pozice p1 = strom1.getPozice();
        strom1.setPozice(strom2.getPozice());
        strom2.setPozice(p1);
        strom1.nakresli();
    }

    @Test
    public void testProhodPozice() {
        pomProhodPozice(zelenyStrom, zlutyStrom);
    }
}

```

### 3.1.5. Výčtový typ (*Enum*)

- běžně se dosud používá jen jako typově zabezpečená náhrada symbolických konstant

- v Javě má tento návrhový vzor přímo klíčové slovo `enum`, které se používá místo `class`
- pro jednoduché výčty (kdy instance jsou celočíselné) stačí použít jen seznam názvů – využívá se přímo klíčového slova `enum`

```

/*****
 * Ukázka nejjednoduššího výčtového typu
 */
public enum SvetoveStrany {
    SEVER, VYCHOD, JIH, ZAPAD;
}

```

- tato jednoduchá deklarace již dává k dispozici defaultní metody:

- ◆ `String name()` – řetězec s názvem dané instance
- ◆ `int ordinal()` – pořadové číslo instance; první instance (zde SEVER) má pořadové číslo 0
- ◆ `String toString()` – název instance (stejně jako `name()`)

- použití pak je (viz též [PPA1-66]):

```
SvetoveStrany smer = SvetoveStrany.JIH;
System.out.println(smer.name());
System.out.println(smer.ordinal());
System.out.println(smer);
System.out.println(SvetoveStrany.SEVER);
```

- vypíše:

```
JIH
2
JIH
SEVER
```

- toto je ovšem poměrně velká degradace možností tohoto typu
- jedná se o plnohodnotnou třídu s mnoha výhodnými možnostmi
- zobecnění Jedináčka
- má konečný počet několika předem známých (vyjmenovaných) instancí
- dobrý příklad je `Smer8`, kdy s jednotlivými instancemi jsou svázány i doplňující metody

## Varování

Typy ve `Smer8` a `SvetoveStrany` jsou zcela rozdílné, byť jsou instance pojmenovány stejně.

- během chodu programu nelze vytvářet další instance
- opět má nepřístupný konstruktor (`private`)
- všechny instance jsou definovány jako veřejné statické atributy – odkazy získáváme přímo přes jméno třídy, např. `SvetoveStrany.JIH` (nebo `Smer8.ZAPAD`)
  - pro získání odkazů se nepoužívá statická tovární metoda
- příklad, kdy je s typem svázána instanční metoda
  - jméno instance (např. `JIH`) slouží jako volání konstruktora, takže parametr za ním je parametrem konstruktoru

```
/*
 * Instance výčtového typu {@code SvetoveStrany} představují
 * světové strany a jejich odpovídající směry na mapě
 */
```

```

*
* @author      Pavel Herout
* @version     2.00.000
*/
public enum SvetoveStrany {

//== HODNOTY VÝČTOVÉHO TYPU =====
    SEVER("nahoru"), VYCHOD("doprava"), JIH("dolů"), ZAPAD("doleva");

//== KONSTANTNÍ ATRIBUTY INSTANCI =====
    private final String smer;

//===== KONSTRUKTORY A TOVÁRNÍ METODY =====
    /**
     * privátní konstruktor
     */
    private SvetoveStrany(String smer) {
        this.smer = smer;
    }

//== OSTATNÍ NESOUKROMÉ METODY INSTANCI =====
    public String smerPohledu() {
        return "Když se na mapě dívám na " +
            name() +
            ", dívám se směrem " + smer;
    }

//== TESTOVACÍ METODY A TŘÍDY =====
    public static void main(String[] args) {
        System.out.println(SvetoveStrany.JIH.smerPohledu());
    }
}

```

- vypíše:

Když se na mapě dívám na JIH, dívám se směrem dolů

## 3.2. Rozhraní

- u instancí jedné třídy platí pravidlo, že jsou (z mateřské třídy) vybaveny sadou stejných metod a tedy umějí reagovat na stejné zprávy
- parametry zpráv musejí být přesně definovaného typu – to dosud nebyl problém
- jakmile budou parametry zpráv objekty našich tříd (dále „ovládané třídy“), pak pro třídu, která s nimi pracuje („ovládající třída“) by bylo nutné vytvářet stejné (překryté) metody, pouze s jiným typem parametru
  - přitom metody dělají principiálně totéž (např. posun) a liší se jen s čím (s obdélníkem, elipsou, ...)

```
public void posun(Obdelnik obd) { ... }
public void posun(Elipsa elip) { ... }
```

- další problém nastává ve chvíli, kdy chceme program rozšiřovat – pro každou novou třídu grafických objektů musíme do ovládající třídy dopsat novou metodu
- snahou je nalézt u ovládaných tříd takovou množinu metod, která je pro ně společná, např. informace o barvě, nakreslení se, skrytí se, změna barvy, posun, ...
- pak lze všechny ovládané třídy považovat za speciální případy nějakého obecnějšího typu
  - řeší se rozhraním a ovládací třída připravuje pouze jednu metodu pracující s obecnějším typem
- současné programy (třídy) mají dvě podoby (opakování z dřívějška)
  - vnější – reprezentovanou rozhraním
    - ◆ říká, co entita (program, třída, metoda, ...) umí a jak se s ním komunikuje
    - ◆ rozhraní pouze popisuje (slibuje), jak to bude vypadat / fungovat
    - ◆ rozhraní je souhrnná informace, kterou potřebuje uživatel pro využívání entity, jejíž vnitřek ho nezajímá
    - ◆ tato informace má být popsána v dokumentaci
  - vnitřní – reprezentovanou implementací
    - ◆ ta zajišťuje, aby entita splňovala sliby rozhraní
    - ◆ snahou je maximálně skrýt implementaci (`private` atributy apod.)
    - ◆ čím méně se o implementaci ví, tím jsou snazší její pozdější změny, ALE **rozhraní musí po svém zveřejnění zůstávat stejné**
    - ◆ jsou-li naveneck známé detaily implementace (`public` atributy, např. Přepravka), může je někdo použít a při změně implementace je třeba ošetřit, že to neovlivnilo všechna předchozí použití – obtížné, někdy nemožné
- toto dělení na dva pohledy důsledně uplatňujeme i ve svých programech, kde máme vše pod kontrolou a vše využíváme jen my sami
  - všechna porušení (zdůvodňovaná zjednodušeními) se dříve či později vymstí

### 3.2.1. Terminologie

- API – *Application Programming Interface*
  - aplikační programátorské rozhraní
  - rozhraní určené pro programátory, jejichž programy s danou aplikací komunikují
  - v Javě souhrn knihoven – Java Core API
  - popsáno důsledně pomocí Javadoc – signatura i kontrakt

## ■ GUI – *Graphical User Interface*

- grafické uživatelské rozhraní
- co uživatel aplikace vidí a přes co s programem komunikuje

## ■ CLI – *Command Line Interface*

- textové uživatelské rozhraní přes terminál příkazové řádky

## ■ rozhraní versus interface

- rozhraní je obecnější pojem – rozhraní = signatura + kontrakt
- interface – programová konstrukce signatury v Javě
- není-li nebezpečí záměny, je „rozhraní“ českým překladem interface

## 3.2.2. Konstrukce interface

### ■ pro možnost fyzického oddělení rozhraní a implementace ve zdrojovém kódu existuje v Javě konstrukce interface

- používá se jen pro popis celé třídy – místo class je interface
- signatura popisuje pouze hlavičky metod, nikoliv jejich těla (tj. implementaci)
  - ◆ je to „metoda bez implementace“
  - ◆ od Javy 8 je možnost zapsat i tělo metody – default – viz dále
- kontrakt je definován prostřednictvím dokumentačních komentářů
- interface je třeba přeložit, po překladu má vlastní soubor .class
- v diagramu tříd je doplněn stereotypem «interface»

### ■ interface je datový typ

- pokud zajistíme (jazykovou konstrukcí), že ovládané třídy budou tohoto datového typu, pak je vyřešen problém s opakujícími se přetíženými metodami (viz dříve posun())
- v ovládající třídě definujeme metody, které očekávají formální parametr typu rozhraní, a skutečný parametr bude instance konkrétní třídy splňující (implementující) rozhraní

### ■ název interface začíná konvencí písmenem I (IKresleny)

- pokud to je možné, nemělo by být názvem podstatné jméno (typické pro třídy) ani sloveso (typické pro metody), ale přídavné jméno
  - ◆ nejlépe takové, které vyjadřuje **schopnost** (IKreslitelný je lepší než IKresleny)
- ne vždy je to možné nebo vhodné nebo pochopitelné (viz dále IObdarovavatelny jako kuriózní případ), proto je konvence s písmenem I na začátku velmi dobrá, byť není používána v Java Core API

## ■ praktické poznámky

- všechny metody deklarované rozhraním jsou povinně `public`
  - ◆ protože neveřejné metody nejsou v rozhraní dovoleny, povoluje syntaxe modifikátor `public` nepsat
  - ◆ implementující třídy však `public` u svých metod uvádět musí => doporučuji jej psát – pak se dá z rozhraní snáze kopírovat
- všechny metody deklarované rozhraním jsou **abstraktní** (jsou to pouze deklarace bez implementace) (**Pozor:** Java 8 zavádí rozšíření – viz dále).
  - ◆ lze použít modifikátor `abstract`
  - ◆ nepoužívat, protože implementující třídy jej použít nesmí – po zkopirování je nutné jej vymazat
- před metodu v rozhraní dodat zakomentovanou anotaci `@Override`
  - ◆ po zkopirování do implementující třídy se dá lehce odkomentovat a překladač pak ohlídá, zda je metoda skutečně implementována

## ■ ukázka jednoduchého rozhraní

```
public interface IUkazkaRozhrani {  
  
    /**  
     * Vypíše na konzoli svůj stav  
     */  
    // @Override  
    public void vypis();  
}
```

- toto rozhraní ve svém kontraktu slibuje, že kdokoliv jej implementuje, bude umět vypsat na konzoli svůj stav
- signatura říká, že se tak stane zasláním zprávy `vypis()`

### Poznámka

Jméno `IUkazkaRozhrani` je zcela nevhodné, protože neříká nic o účelu. Vhodnější by bylo `IVypisovatelny`.

## 3.2.3. Implementace interface

### ■ třída se může přihlásit k tomu, že implementuje dané rozhraní

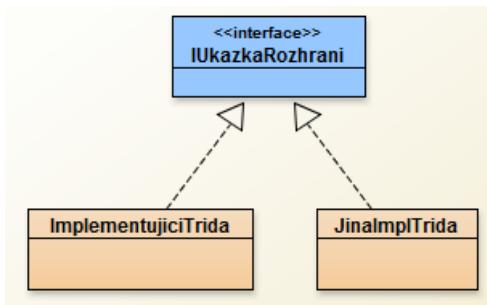
- přihlašuje se k tomu veřejně ve své hlavičce; tím se toto prohlášení stává součástí její signatury a překladač bude kontrolovat jeho naplnění
- instance třídy, která implementuje nějaké rozhraní, se mohou vydávat za instance daného rozhraní
- třída může implementovat několik rozhraní současně, její instance se pak mohou vydávat za instance kteréhokoliv z nich

- třída implementuje rozhraní, když implementuje všechny jeho metody a do své hlavičky uvede implements, např.

```
public class ImplementujiciTrida implements IUkazkaRozhrani {
```

- ukázka dvou jednoduchých tříd, které implementují rozhraní

- dodržení signatury zkонтroluje překladač
- dodržení kontraktu musí zajistit programátor a zkontovalovat tester – vypisuje něco smysluplného a vypisuje to na konzoli



- ImplementujiciTrida využívá standardních postupů a kontrakt plní uspokojivě

- metoda getClass() je ze třídy Object a metoda getSimpleName() vrací jméno třídy (viz dříve)

```
public class ImplementujiciTrida implements IUkazkaRozhrani {

    private static int pocet = 0;
    private final int PORADI = ++pocet;

    /**
     * Vypíše na konzoli svůj stav
     */
    @Override
    public void vypis() {
        System.out.println(this.toString());
    }

    public void ahoj() {
        System.out.println("Ahoj");
    }

    @Override
    public String toString() {
        return "Jsem " + PORADI + ". instance třídy " +
               this.getClass().getSimpleName();
    }
}
```

- JinaImplTrida použila cestu nejmenšího odporu, signaturu plní, ale o smysluplném kontraktu lze pochybovat

```

public class JinaImplTrida implements IUkazkaRozhrani {

    /**
     * Vypíše na konzoli svůj stav
     */
    @Override
    public void vypis() {
        System.out.println("Jsem nějaká instance nějaké třídy");
    }
}

```

- po vytvoření instancí (využije se defaultní bezparametrický konstruktor) a zaslání zprávy `vypis()` instance vypíší

```

Jsem 1. instance třídy ImplementujiciTrida
Jsem 2. instance třídy ImplementujiciTrida
Jsem nějaká instance nějaké třídy

```

- kdykoliv se hovoří o instanci rozhraní, hovoří se ve skutečnosti o instanci nějaké třídy, která dané rozhraní implementuje

### 3.2.4. Výhoda použití rozhraní

- výhody budou postupně přibývat
- reference na rozhraní může uchovávat odkaz na libovolnou instanci libovolné třídy splňující toto rozhraní
  - jinak řečeno: rozhraní umožňuje deklarovat požadované vlastnosti zpracovávaných objektů bez ohledu na to, čí instancí tyto objekty budou

```

ImplementujiciTrida it1 = new ImplementujiciTrida();
IUkazkaRozhrani ur1 = new ImplementujiciTrida();
IUkazkaRozhrani ur2 = new JinaImplTrida();
it1.vypis();
ur1.vypis();
ur2.vypis();

```

- toto prakticky znamená, že když od třídy potřebujeme jen službu slibovanou v rozhraní, stačí typovat vzniklé instance na toto rozhraní
  - ◆ významné sjednocení typu, který požaduje určitou službu
- pokud chceme, aby třída něco dělala, je možné tyto služby velmi přesně (signaturu i kontrakt) popsat pomocí rozhraní
  - učitel zadává úkoly, studenti je implementují – testy jsou pak snadné – učitel ověřuje jen kontrakt, signaturu ověří překladač
  - třída svými službami zapadá do nějakého většího celku

## 3.2.5. Značkovací rozhraní (*marker interface, tagging interface*)

- speciální případ rozhraní, které nemá žádné metody
  - tudíž třída, která jej implementuje, nemusí kromě změny své hlavičky dělat nic navíc
  - nelze tedy kontrolovat signaturu
- smysl značkovacího rozhraní je **připomenout**, že se má **splnit nějaký kontrakt**
- příklad – rozhraní `java.lang.Cloneable` jehož implementací:
  - se třída zavazuje plnit kontrakt pro metodu `clone()` zděděnou z `Object`
  - předpokládá se, že programátor, který napíše do hlavičky vytvářené třídy `implements Cloneable` nezapomene překrýt metodu `clone()`
  - ostatním povoluje vytvářet kopie jejich instancí klonováním

## 3.2.6. Funkční rozhraní (*functional interface*)

- rozhraní, které má pouze jednu metodu
  - přesněji řečeno jen jednu abstraktní metodu, další mohou být defaultní (viz dále)
- nejčastější použití – v souvislosti s lambda výrazy (viz dále)

## 3.2.7. Defaultní metody v rozhraní

- od Java 8
- v rozhraní mohou být metody, které mají tělo – jsou označeny klíčovým slovem `default`
  - v příkladu je doplněno dřívější rozhraní `IUkazkaRozhrani`

```
public interface IVypisovatelny {  
  
    //== SIGNATURY INSTANČNÍCH METOD =====  
    /**  
     * Vypíše na konzoli svůj stav  
     */  
    //  @Override  
    public void vypis();  
  
    //== DEFAULT METODY =====  
    /**  
     * Vypíše na konzoli svůj stav pomocí toString()  
     */  
    public default void defaultniVypis() {  
        System.out.println(this.toString());  
    }  
}
```

## ■ použití

```
public class A implements IVypisovatelny {  
    @Override  
    public void vypis() {  
        System.out.println("Jsem A");  
    }  
  
    public static void main(String[] args) {  
        A a = new A();  
        a.vypis();  
        a.defaultniVypis();  
    }  
}
```

vypíše:

```
Jsem A  
A@659e0bfd
```

## ■ defaultní metodu můžeme samozřejmě ve třídě znova implementovat

```
public class B implements IVypisovatelny {  
    @Override  
    public void vypis() {  
        System.out.println("Jsem B");  
    }  
  
    @Override  
    public void defaultniVypis() {  
        System.out.println("Jsem defaultni B");  
    }  
  
    public static void main(String[] args) {  
        B b = new B();  
        b.vypis();  
        b.defaultniVypis();  
    }  
}
```

vypíše

```
Jsem B  
Jsem defaultni B
```

## ■ někdy umíme již v rozhraní napsat defaultní metodu tak, že pravděpodobně nebude nutné ji znova implementovat

```
/* *****  
 * Instance rozhraní {@code IVypisovatelnyLepsi} představují objekty,  
 * které dokáží na konzoli vypsat svůj stav  
 * a navíc defaultně dokáží vypsat jméno třídy, která toto rozhraní  
 * implementuje  
 */
```

```

/*
 * @author Pavel Herout
 * @version 1.00.0000 - 2017-09-01
 */
public interface IVypisovatelnyLepsi {

    //== SIGNATURY INSTANČNÍCH METOD =====
    /**
     * Vypíše na konzoli svůj stav
     */
    // @Override
    public void vypis();

    //== DEFAULT METODY =====
    /**
     * Vypíše na konzoli jméno budoucí implementující třídy
     */
    public default void defaultniVypisJmenaTridy() {
        System.out.println("Třída " + this.getClass().getSimpleName());
    }
}

```

## použití

```

public class C implements IVypisovatelnyLepsi {
    @Override
    public void vypis() {
        System.out.println("Jsem C");
    }

    public static void main(String[] args) {
        C c = new C();
        c.vypis();
        c.defaultniVypisJmenaTridy();
    }
}

```

## vypíše

```
Jsem C
Třída C
```

## ■ rozumné důvody použití defaultních metod v rozhraní:

### 1. neinvazivní rozšíření již existujícího a používaného rozhraní

- můžeme přidat defaultní metody a nijak tím neovlivníme stávající třídy používající dřívější verzi rozhraní
  - ◆ pokud bychom do rozhraní přidali metody bez `default`, bylo by nutné implementaci těchto nových metod zajistit zpětně ve všech třídách, které původní rozhraní implementovaly – složité až nemožné

- výhoda je vidět v IVypisovatelnyLepsi, kdy každá třída, která toto rozhraní implementuje (zde třída C), má „zadarmo“ k dispozici službu výpisu svého jména

2. jednoduchá implementace metody, která by se stejně beze změny opakovala ve všech implementujících třídách (zde setPozice(Pozice pozice))

```
public interface IPosuvny {
    ►
    /*********************************************************************
     * Vrátí instanci třídy {@code Pozice} s aktuální pozicí instance.
     *
     * @return Instance třídy {@code Pozice} s aktuální pozicí instance
     */
    // @Override
    public Pozice getPozice();

    ►
    /*********************************************************************
     * Nastaví novou pozici instance.
     *
     * @param x Nově nastavovaná vodorovná (x-ová) souřadnice instance,
     *           x=0 má levý okraj plátna, souřadnice roste doprava
     * @param y Nově nastavovaná svislá (y-ová) souřadnice instance,
     *           y=0 má horní okraj plátna, souřadnice roste dolů
     */
    // @Override
    public void setPozice(int x, int y);

    ►
    /*********************************************************************
     * Nastaví novou pozici instance.
     *
     * @param pozice Nastavovaná pozice instance
     */
    public default void setPozice(Pozice pozice) {
        setPozice(pozice.x, pozice.y)
    }
}
```

### 3.2.8. Statické metody v rozhraní

- již v dřívějších verzích Javy mohla být součástí rozhraní i statická konstanta
- od Javy 8 může být v rozhraní i statická metoda s implementovaným tělem
- praktická použitelnost je sporná – nepřináší žádné podstatné výhody oproti návrhovému vzoru Knihovní třída
- ukázka rozhraní se statickými metodami

```
public interface IPrvociselnny {
    //== STATICKÉ KONSTANTY =====
    public static final int[] PRVOCISLA = {2, 3, 5, 7, 11, 13};
```

```

//== STATICKÉ METODY =====
public static int getPocetPrvocisel() {
    return PRVOCISLA.length;
}

public static int getPrvocisloVPoradi(int poradi) {
    if (poradi < getPocetPrvocisel()) {
        return PRVOCISLA[poradi - 1];
    }
    else {
        throw new IllegalArgumentException();
    }
}

//== SIGNATUREY INSTANČNÍCH METOD =====
/*
 * Zjistí, zda je libovolné číslo prvočíslem
 *
 * @param cislo libovolné číslo
 * @returns {@code true} pokud je {@code cislo} prvočíslem
 *          nebo {@code false} v ostatních případech
 */
//    @Override
public boolean isPrvocislo(int cislo);
}

```

## ■ implementace tohoto rozhraní

```

public class PraceSPrvocisly implements IPrvociseln {
    @Override
    public boolean isPrvocislo(int cislo) {
        boolean je = true;
        for (int i = cislo - 1; i > 1; i--) {
            if (cislo % i == 0) {
                je = false;
                break;
            }
        }
        return je;
    }

    public static void main(String[] args) {
        System.out.println("Pocet znamykh prvocisel = " + ►
IPrvociseln.getPocetPrvocisel());
        System.out.println("3. prvocislo v poradi = " + ►
IPrvociseln.getPrvocisloVPoradi(3));
        PraceSPrvocisly psp = new PraceSPrvocisly();
        System.out.println("27 je prvocislo: " + psp.isPrvocislo(27));
        System.out.println("29 je prvocislo: " + psp.isPrvocislo(29));
    }
}

```

```
}
```

vypíše:

```
Pocet znamych prvocisel = 6
3. prvocislo v poradi = 5
27 je prvocislo: false
29 je prvocislo: true
```

## 3.3. Návrhové vzory – pokračování

### 3.3.1. Motivace

- při presouvání stromu se stalo, že se nepřekreslovalo vždy správně – jednotlivé instance se přemazávaly a musely se dodatečně překreslovat
  - bylo by vhodné zajistit, aby se o plynulé vykreslování a hlavně překreslování při změně situace na plátně někdo postaral, aby to nemusely dělat vykreslené objekty jednotlivě
- dosud byla třída `Platno` pasivní objekt, na který si každý kreslil, jak se mu zachtělo
  - a současně byl každý zodpovědný za vše, co se na plátně událo – obtížná a špatně řešitelná situace – přemazávání objektů
- změníme kompletně filosofii – třída `SpravcePlatna` se chová jako manažer
  - dostane nějaké objekty do správy a dohlíží na to, aby byly všechny správně nakresleny – dvě aktivity:
    - ◆ vykreslit změněný objekt ve správnou chvíli
    - ◆ nechat překreslit ostatní objekty, které změna mohla ovlivnit (přemazání apod.)
  - když se dozví, že se něco změnilo, tak zařídí, aby obrázek na plátně odpovídal skutečnosti
  - sám ale nic nekreslí, kreslení pouze organizuje – objekty se vykreslují samy
  - kromě jiných vylepšení může vykreslovaný objekt metodou `SP.nekresli()` pozastavit svoje vykreslování (aby se nenakreslila koruna a pak někdy viditelně později kmen) a metodou `SP.vratKresli()` vykreslování obnovit (pokyn k překreslení)
    - ◆ `SP` je referenční proměnná na instanci jedináčka `SpravcePlatna`
- v celé implementaci tohoto kreslení jsou použity návrhové vzory Posluchač a Prostředník
  - Posluchač řeší úkol, **kdy** se mají vykreslit konkrétní objekty
  - Prostředník řeší úkol, **které** objekty se mají vykreslovat v závislosti na změně nějakého objektu
    - ◆ jak se má objekt vykreslit, může být vedlejší efekt Prostředníka

### 3.3.2. Posluchač (*Listener*)

- též **Pozorovatel** (*Observer*), též dvojice „**Vydavatel** (*Publisher*) – **Předplatitel** (*Subscriber*)“
- řeší typickou úlohu, kdy objekt čeká na nějakou událost, např. stisk klávesy, zaslání zprávy, na kterou má reagovat okamžitě
- dvojí způsob řešení:
  1. objekt neustále testuje, zda událost nastala – nevhodné
  2. objekt se zaregistrouje u zdroje událostí a nic nedělá; až událost nastane, zdroj mu pošle domluvenou zprávu – mnohem vhodnější
    - analogie prozvánění mobilem
- zdroj zpráv se označuje jako **Vysílač** nebo **Pozorovaný** nebo **Vydavatel**
  - tento návrhový vzor je tedy vždy dvojice **Vysílač-Posluchač**
  - terminologicky nejbližší reálnému životu je Vydavatel-Předplatitel, protože Předplatitelé se musejí u Vydavatele registrovat
- registrované objekty (instance různých tříd) posluchače musejí implementovat rozhraní, aby jim mohl zdroj vysílač posílat jednotnou zprávu – registrují se jako implementace rozhraní
- realizace tohoto návrhového vzoru viz později v dalších přednáškách
  - v Java Core API má tento návrhový vzor přímo předlohu – třída `Observable` a rozhraní `Observer`
- použití tohoto návrhového vzoru pro případ kreslení – třída `SpravcePlatna` je vysílač:
  1. definovat rozhraní se zprávou, kterou bude objekt vysílač posílat všem přihlášeným posluchačům – `IKresleny` nakresli (`Kreslitko`)

```
*****  
 * Rozhraní {@code IKresleny} musí implementovat všechny třídy, které ►  
 chtějí,  
 * aby jejich instance byly přijaty do správy instance {@link ►  
 SpravcePlatna},  
 * která vyžaduje, aby se na požádání uměly nakreslit prostřednictvím  
 * dodaného kreslítko.  
 * <p>  
 * Implementací tohoto rozhraní třída současně slibuje, že bude správce ►  
 plátна  
 * okamžitě informovat o jakýchkoliv změnách ve svém umístění a vzhledu,  
 * které chce promítнуть do své podoby na plátně. O tom, že se něco změnilo  
 * informuje správce plátна zavolením jeho metody  
 * {@link SpravcePlatna#překresli()}  
 */  
public interface IKresleny  
{  
    ►  
    *****  
        * Prostřednictvím dodaného kreslítko vykreslí obraz své instance.
```

```

    *
    * @param kreslitko Kreslitko, které nakreslí instanci
    */
//    @Override
    public void nakresli(Kreslitko kreslitko);
}

```

- Kreslitko není pro princip Vysílač-Posluchač důležité – je důležité proto, že odstínuje SpravcePlatna od vlastního kreslení
- aby objekty věděly, kam se mají vykreslit, využijí objekt třídy Kreslitko, který je jim předán jako parametr zasílané zprávy
- jeho instanci poskytuje jen SpravcePlatna
  - ◆ je to zabezpečení, že se bude kreslit jen tehdy, když to uzná SpravcePlatna za vhodné
  - ◆ k plátnu nesmí přistupovat nikdo, o kom SpravcePlatna jako Prostředník neví
- implementace Kreslitko není pro nás důležitá (je to záležitost grafiky)

## 2. implementovat toto rozhraní všemi posluchači, zde ve třídě Strom

- využívá se toho, že třídy Obdelník a Elipsa již implementují rozhraní IKresleny

```

*****
* Vykreslí obraz své instance na plátno.
*
* @param kreslitko objekt, jehož prostřednictvím se má instance nakreslit.
*/
public void nakresli(Kreslitko kreslitko) {
    koruna.nakresli(kreslitko);
    kmen.nakresli(kreslitko);
}

```

## 3. protože posluchači implementují stejné rozhraní, mohou se u SpravcePlatna zaregistrovat vykreslované různé objekty jako instance tohoto rozhraní

- to se provede metodou pridej (vykreslovanyObjekt), která je ze třídy SpravcePlatna
- pomocí této metody lze zaregistrovat všechny Posluchače u Vysílače (SpravcePlatna) – ukázka ze třídy Strom

```

*****
* Přihlásí instanci u aktivního plátna do jeho správy.
*/
public void zobraz() {
    SP.pridej( this );
}

```

## 4. kdykoliv dojde k očekávané události, vysílač SpravcePlatna všem zaregistrovaným pouze postupně posílá zprávu nakresli (aktualniKreslitko)

- jak si vytvoří aktuální kreslitko je jeho interní záležitostí

## Poznámka

Zde dochází k mírně nepřehledné situaci, kdy zdrojem události je grafický objekt, který se potřebuje vykreslit (nebo překreslit). Příjemcem zprávy je kromě ostatních grafických objektů tentýž grafický objekt, který od vysílače dostane výzvu (tj. zprávu) s povolením (tj. s objektem kreslítka) k vykreslení. Viz též následující návrhový vzor.

### 3.3.3. Prostředník (*Mediator*)

- používá se v případě, že má vzájemně komunikovat větší množství objektů
  - nakreslené obrazce navzájem přemazávají – nevědí o sobě
    - ◆ po změně jednoho by změněný měl dát zprávu všem ostatním, aby se překreslily
  - nelze, aby komunikoval každý s každým – mnoho vazeb – zdroj chyb, problémy při budoucích úpravách
    - ◆ každý obrazec by musel vědět o všech ostatních – velká složitost
- Prostředník je vytvářen jako jeden jediný objekt, který zprostředkuje vzájemné komunikace (telefonní ústředna)
  - objekty se obracejí pouze na Prostředníka, který je pak zodpovědný za předání zprávy adresátovi nebo adresátům
  - způsob registrace objektů je velmi podobný (někdy shodný) s registrací u Vysílače
- Prostředník pro svojí implementaci často využívá návrhový vzor Vysílač-Posluchač
  - objekt, který chce předat zprávu jiným, ji předá Vysílači, u kterého jsou jiní zaregistrovaní jako Posluchači
  - Vysílač jim zprávu předá
- Prostředník definuje formát zpráv, které je mu možno posílat
  - nejjednodušší formou je zpráva bez parametrů, která se zašle všem
  - zpráva ale může mít parametr (např. telefonní číslo, pozici, atd.) ze které může Prostředník s použitím další libovolně sofistikované logiky určit adresáta nebo omezenou skupinu adresátů
- používá se technika **vynucení závislosti** (*Dependency Injection*) – programujeme tak, aby objekty vyšší úrovně nezávisely na objektech nižší úrovně
  - Prostředník je objekt vyšší úrovně, komunikující objekty pak nižší úrovně
- příklad řešení
  - vytvoříme Prostředníka SpravcePlatna, ke kterému se každý nově vzniklý geometrický tvar přihlásí metodou `pridej(this)`
    - ◆ kdo bude chtít být nakreslen na plátně, tzn. potenciálně ovlivňovat situaci jiných na plátně, musí se na začátku přihlásit

- pokud jakýkoliv tvar změní pozici či podobu, informuje prostředníka metodou `SP.prekresli()`
- Prostředník zajistí **libovolně inteligentní** předání zprávy o vykreslení všem příslušným tvarům
  - ◆ např. pokud by zpráva obsahovala jako parametr `Pozici`, mohla by být předána jen objektům v blízkosti změněného objektu
  - ◆ v této implementaci je zpráva, aby se překreslily, posílána všem registrovaným objektům
  - ◆ součástí zprávy je servisní parametr – kreslitko

### 3.3.4. Služebník (Servant)

- přidání dodatečné funkcionality skupině tříd, aniž bychom museli tyto třídy měnit a dávat do nich stejný/podobný kód
- instance Služebníka obsluhují instance tříd požadujících novou funkčnost
- musí existovat dohoda mezi Služebníkem a obsluhovanými
  - obsluhovaní musí ve svém rozhraní popsat, co mají umět
  - Služebník má toto rozhraní jako typ parametru svých obsluhujících metod
  - požadovaná funkčnost obsluhovaných se zajistí tím, že se posílá zpráva Služebníkovi a reference na obsluhovanou instanci je skutečným parametrem této zprávy
- příklad – budeme chtít, aby se všechny tvary na plátně plynule posouvaly
  - rozhraní bude `IPosuvny` a využívá též přepravku `Pozice` – viz dříve

```

/*****
 * Instance rozhraní {@code IPosuvny} představují geometrické tvary,
 * které umějí prozradit a nastavit svoji pozici.
 *
 * @author Rudolf PECINOVSKÝ
 */
public interface IPosuvny extends IKresleny {
    //== SIGNATURY INSTANČNÍCH METOD =====
    /**
     * Vrátí instanci třídy {@code Pozice} s aktuální pozicí instance.
     *
     * @return Aktuální pozici v instanci třídy {@code Pozice}
     */
    //  @Override
    public Pozice getPozice();

    /**
     * Nastaví novou pozici instance.
     *
     * @param x Nově nastavovaná vodorovná (x-ová) souřadnice instance,
     *           x=0 má levý okraj plátna, souřadnice roste doprava
     * @param y Nově nastavovaná svislá (y-ová) souřadnice instance,
     *           y=0 má horní okraj plátna, souřadnice roste dolů
   */

```

```

        */
    //  @Override
    public void setPozice(int x, int y);

    //== DEFAULT METODY =====
    /***** Nastaví novou pozici instance.
    *
    * @param pozice    Nastavovaná pozice instance
    */
    public default void setPozice(Pozice pozice) {
        setPozice(pozice.x, pozice.y);
    }
}

```

- třída Presouvac je Služebník, který zajistí (na jednom místě kódu) implementaci plynulého přesunu, přičemž klíčové metody jsou presunO() a presunNa()

```

/*****
 * Třída {@code Přesouvac} slouží k plynulému přesouvání instancí tříd
 * implementujících rozhraní {@link IPosuvny}.
 *
 * @author Rudolf PECINOVSKÝ
 * @version 1.09.2629 - 2011-01-03
 */
public class Presouvac
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY =====
    /** Počet milisekund mezi dvěma překresleními objektu. */
    private static final int PERIODA = 50;

//== PROMĚNNÉ ATRIBUTY TŘÍDY =====
    private static int pocet = 0;

//== KONSTANTNÍ ATRIBUTY INSTANCÍ =====
    /**
     * Identifikační kód (rodné číslo) instance.
     */
    private final int ID = ++pocet;

    /**
     * Název sestávající z názvu třídy a pořadí instance
     */
    private final String nazev;

//== PROMĚNNÉ ATRIBUTY INSTANCÍ =====
    /**
     * Specifikuje rychlosť posunu objektu daným posunovačem.
     */
    private int rychlosť;

//== KONSTRUKTOŘE A TOVÁRNÍ METODY =====
    /*****
     * Vytvoří přesouvače, který bude přesouvat objekty rychlosťi 1.
     */
}

```

```

public Presouvac()
{
    this( 1 );
}

/***********************
 * Vytvoří přesouvače, který bude přesouvat objekty zadanou rychlostí.
 *
 * @param rychlost Rychlost, kterou bude přesouvač pohybovat
 *                 se svěřenými objekty.
 */
public Presouvac( int rychlost )
{
    if( rychlost <= 0 ) {
        throw new IllegalArgumentException(
            "Zadana rychlost musi byt kladna!" );
    }
    this.rychlost = rychlost;
    this.nazev    = getClass().getName() + "(ID=" + pocet +
                    ",rychlost=" + rychlost + ")";
}

//== OSTATNÍ NESOUKROMÉ METODY INSTANCI =====

/***********************
 * Metoda převádí instanci na řetězec - používá se pro účely ladění.
 * Vrácený řetězec obsahuje název třídy následovaný identifikačním číslem
 * dané instance, které určuje pořadí dané instance mezi instancemi této
 * třídy, a nastavenou rychlostí přesunu.
 *
 * @return Řetězec charakterizující instanci
 */
@Override
public String toString()
{
    return nazev;
}

/***********************
 * Plynule přesune zadaný objekt o zadaný počet obrazových bodů.
 *
 * @param doprava      Počet bodů, o než se objekt přesune doprava
 * @param dolu         Počet bodů, o než se objekt přesune dolů
 * @param objekt       Přesouvaný objekt
 */
public void presunO(int doprava, int dolu, IPosuvny objekt)
{
    double vzdalenost = Math.sqrt(doprava*doprava + dolu*dolu);
    int     kroku     = (int)(vzdalenost / rychlost);
    if (kroku == 0) {
        kroku = 1;
    }
}

```

```

        double dx = (doprava+.4) / kroku;
        double dy = (dolu +.4) / kroku;
        Pozice pozice = objekt.getPozice();
        double x = pozice.x + .4;
        double y = pozice.y + .4;

        for (int i=kroku; i > 0; i--) {
            x = x + dx;
            y = y + dy;
            objekt.setPozice( (int)x, (int)y );
            IO.cekej(PERIODA);
        }
    }

/*****
 * Plynule přesune zadaný objekt o zadaný počet obrazových bodů.
 *
 * @param posun Velikost posunu v jednotlivých směrech
 * @param objekt Přesouvaný objekt
 */
public void presunO(Pozice posun, IPosuvny objekt)
{
    presunO(posun.x, posun.y, objekt);
}

/*****
 * Plynule přesune zadaný objekt do požadované pozice.
 *
 * @param x      x-ova souřadnice požadované cílové pozice
 * @param y      y-ova souřadnice požadované cílové pozice
 * @param objekt Přesouvaný objekt
 */
public void presunNa(int x, int y, IPosuvny objekt)
{
    Pozice pozice = objekt.getPozice();
    presunO(x-pozice.x, y-pozice.y, objekt);
}

/*****
 * Plynule přesune zadaný objekt do požadované pozice.
 *
 * @param pozice Cílová pozice
 * @param objekt Přesouvaný objekt
 */
public void presunNa(Pozice pozice, IPosuvny objekt)
{
    presunNa(pozice.x, pozice.y, objekt);
}

```



- implementace IPosuvny u obsluhovaného Strom – viz též již dříve u Přepravky

**blok SP.nekresli(); { ... } SP.vratKresli(); zajistí vykreslení celého stromu najednou**

```
*****
 * Přemístí strom na jinou pozici - levý horní roh.
 *
 * @param x nová x-souřadnice
 * @param y nová y-souřadnice
 */
@Override
public void setPozice(int x, int y) {
    SP.nekresli();
    koruna.setPozice(x, y);
    kmen.setPozice(x + (koruna.getSirka() - kmen.getSirka()) / 2,
                    y + koruna.getVyska());
    } SP.vratKresli();
}

@Override
public Pozice getPozice() {
    return new Pozice(getX(), getY());
}
```

- implementace IPosuvny u obsluhovaného Obdelnik

```
public Pozice getPozice()
{
    return new Pozice( xPos, yPos );
}

public void setPozice(int x, int y)
{
    xPos = x;
    yPos = y;
    SP.prekresli();
}
```

- požadovaná funkčnost obsluhovaných se zajistí tím, že se posílá zpráva služebníkovi a obsluhovaná instance je skutečným parametrem této zprávy

```
@Test
public void testPresouvani() {
    Strom stromA = new Strom(100, 100);
    Strom stromB = new Strom();
    Strom stromC = new Strom(200, 0);
    stromA.zobraz();
    stromB.zobraz();
    stromC.zobraz();

    Obdelnik obd = new Obdelnik();
    obd.nakresli();
```

```

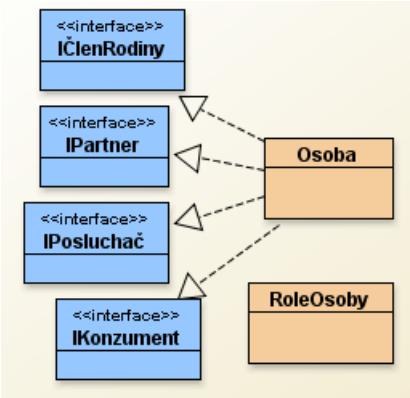
Presouvac presRychl_1 = new Presouvac();
presRychl_1.presunNa(150, 200, stromC);

Presouvac presRychl_5 = new Presouvac(5);
presRychl_5.presunO(200, 0, stromB);
presRychl_5.presunO(-100, -100, stromA);
presRychl_1.presunNa(200, 200, obd);
}

```

## 3.4. Třída implementuje více rozhraní

- při implementaci Služebníka je výhodné, aby poskytoval co nejužší služby (nic komplexního)
  - tím bude moci v budoucnu obsloužit mnohem větší počet zájemců než při komplexních službách
- snažíme se, aby v rozhraní bylo jen nutné minimum metod
- další metody společné obsluhovaným objektům dáváme do jiného či jiných rozhraní
- třída může implementovat kolik rozhraní, kolik chce – musí ale implementovat všechny metody všech rozhraní
  - to umožňuje mj. vytvářet specializované služebníky
- implementace více rozhraní (tj. chování navenek) je věc běžná ze života
  - doma člen rodiny (tj. dcera nebo syn)
  - ve škole posluchač(ka)
  - v menze konzument(ka)
  - na rande partner(ka)
  - každá role vyžaduje jiné schopnosti (např. uklidPoSobě(), pišSiPoznámky(), jezJídlo(), dejMiPusu(), ...)
- pokud objekt vystupuje jako instance nějakého rozhraní, pak mu lze zasílat jen zprávy z daného rozhraní (např. posluchač(ka) nemůže dostat zprávu dejMiPusu() )
  - patřičnost zpráv zkонтroluje překladač
- příklad



```

public interface IČlenRodiny {
    public void uklidPoSobě();
}
  
```

```

public interface IPosluchač {
    public void pišSiPoznámky();
}
  
```

```

public interface IKonzument {
    public void jezJídlo();
}
  
```

```

public interface IPartner {
    public void dejMiPusu();
}
  
```

```

public class Osoba implements IČlenRodiny, IPosluchač,
    IKonzument, IPartner {
  
```

```

@Override
public void uklidPoSobě() {
    System.out.println("uklizím");
}
  
```

```

@Override
public void pišSiPoznámky() {
    System.out.println("studuji");
}
  
```

```

@Override
public void jezJídlo() {
    System.out.println("jím");
}
  
```

```

@Override
public void dejMiPusu() {
    System.out.println("libám");
}
  
```

```

// není ze žádného rozhraní
public void přijmiDárek() {
    System.out.println("mám radost z dárku");
}
  
```

```

    }
}

public class RoleOsoby {
    public static void main(String[] args) {
        Osoba osoba = new Osoba();
        System.out.println("Jsem: " + osoba.getClass().getName() +
                           " a umím:");
        osoba.uklidPoSobě();
        osoba.pišSiPoznámky();
        osoba.jezJídlo();
        osoba.dejMiPusu();
        osoba.přijmiDárek();

        IPosluchač posluchač = new Osoba();
        System.out.println("Jsem: " + posluchač.getClass().getName() +
                           " a umím:");
        //      posluchač.dejMiPusu(); // nelze
    }
}

```

### 3.4.1. Přetypování

- s používáním různých rozhraní se stane, že máme odkaz na objekt uložený v referenční proměnné určitého typu, který ale neumožňuje využít jiných schopností odkazovaného objektu

```
posluchač.dejMiPusu(); // nelze
```

- přitom ale objekt tyto schopnosti má
- pak lze použít přetypování (typovou konverzi)
  - stejná konstrukce jako u primitivních datových prvků – [PPA1-26]
- přetypováním se mění typ odkazu – nemění odkazovaný objekt
- ten, kdo přetypovává, je zodpovědný za to, že objekt splňuje i vlastnosti nového typu

```
IPartner partner = (IPartner) posluchač;
```

- nesplňuje-li, bude za běhu vypsána chyba: `java.lang.ClassCastException`
- pro test, zda je možné přetypovat, použijeme operátor `instanceof`

```
boolean jeToTypIPartner = posluchač instanceof IPartner;
```

- vlastní přetypování tedy provedeme bezpečně až po testu s `instanceof`

```
if (posluchač instanceof IPartner) {
    IPartner partner = (IPartner) posluchač;
```

```
    partner.dejMiPusu();  
}
```

- potřebujeme-li službu jen dočasně, není nutné vytvářet referenční proměnnou – využijeme anonymní reference

```
if (posluchač instanceof IKonzument) {  
    ((IKonzument) posluchač).jezJídlo();  
}
```

# Kapitola 4. Datové typy, balíky, JAR

## 4.1. Metody třídy Object – pokračování

- pro další výklad je nutné znát další dvě metody, které každá třída dědí od třídy Object
  - boolean equals() – porovnává dva objekty na rovnost
  - int hashCode() – vypočte unikátní číslo, které může objekt dále reprezentovat
- obecný kontrakt zní, že pokud v naší třídě překryjeme jednu z těchto metod, musíme současně překrýt i druhou

### 4.1.1. Metoda equals()

- pět pravidel obecného kontraktu:

1. objekt se musí vždy rovnat sám sobě – reflexivnost

`x.equals(x)` je vždy true

těžko se poruší

2. objekty se musí rovnat křížem – symetričnost

`y.equals(x) == x.equals(y)`

porušit už lze

3. rovná-li se jeden objekt druhému a druhý třetímu, musí se rovnat i první třetímu – tranzitivita

jestliže `x.equals(y) == true` a `y.equals(z) == true` musí `x.equals(z) == true`

je reálné toto pravidlo porušit

4. jsou-li si dva objekty rovné, musí si být rovné tak dlouho, dokud u některého z nich nenastane změna – konzistentnost

upozorňuje na problém měnitelných objektů – viz dále

5. žádný objekt se nesmí rovnat null

`x.equals(null) == false`

pravidlo v sobě zahrnuje i nepřípustnost vyvolání výjimky `NullPointerException` – místo reference na porovnávaný objekt byla metodě `equals()` předána hodnota `null`

- kontrakt vypadá složitě, ale je poměrně snadné jej dodržet

- příklad pro přepravku `Pozice`

```
public class Pozice {  
    public final int x, y;
```

```
//... Konstruktor a jiné metody vynecháváme
```

```
public boolean equals(Object o) {  
    if (o == this) {  
        return true;  
    }  
    if (o instanceof Pozice == false) {  
        return false;  
    }  
    Pozice p = (Pozice) o;  
    return (x == p.x) && (y == p.y);  
}
```

- porovnáváme-li objekty pomocí operátoru `==` (nebo `!=`), zjistí se pouze to zda se jedná o shodné instance
- požadujeme-li test shody hodnot, používáme volání `equals(Object)`, která **může** zjistit stejnost hodnot
  - implicitní verze, která se dědí od `Object` pracuje zcela stejně jako operátor `==`
    - ◆ toto nastavení vyhovuje u odkazových objektových typů
      - pozor na bezmyšlenkové překrývání `equals()` – viz dále
      - ◆ u hodnotových typů (např. typu `Přepravka` apod. – viz dále) musíme definovat vlastní verzi `equals()`
- je na nás, abychom stanovili pravidla, kdy se objekty rovnají
  - typicky se rovnají, když se rovnají všechny jejich atributy

## 4.1.2. Metoda `hashCode()`

- metoda vrací vypočtené unikátní číslo, které může objekt dále jednoznačně a rychle reprezentovat
  - používá se pro výrazné zefektivnění práce s objekty v některých druzích kolekcí
  - využívá se rozptylových tabulek (*hash table*) – podrobně viz v PPA2
- tři pravidla obecného kontraktu:
  1. pro tentýž objekt musí `hashCode()` vracet vždy stejný `int`
  2. rozhodla-li `equals()`, že jsou si dva objekty rovny, musí `hashCode()` vrátit stejný `int`
  3. nejsou-li si objekty rovny podle `equals()`, mohou mít stejný hešovací kód
    - je to nevhodná implementace `hashCode()` – významně snižuje efektivitu programu
- způsob přípravy efektivní metody `hashCode()` bude ukázán později u kolekcí

## 4.2. Datové typy Javy

- Java pro zvýšení efektivnosti dělí datové typy na:

- primitivní
- objektové
  - ◆ odkazové
  - ◆ hodnotové
    - měnitelné / proměnné (*mutable*)
    - neměnitelné (*immutable*)

### 4.2.1. Primitivní datové typy

- podrobně viz [PPA1-19]

- datové typy jsou

- celočíselné
  - ◆ byte (8 bitů)
  - ◆ short (16 bitů)
  - ◆ int (32 bitů)
  - ◆ long (64 bitů)
- reálné
  - ◆ float (32 bitů)
  - ◆ double (64 bitů)
- znakové
  - ◆ char (16 bitů)
- booleovské
  - ◆ boolean – hodnoty true a false

- každý primitivní datový typ má svůj objektový obalový typ – viz dále

- paměť je primitivním datovým typům přidělena v okamžiku jejich deklarace a odebrána v okamžiku konce jejich života – viz podrobně dříve
  - např. formální parametry mají dobu života od vstupu do metody až do jejího ukončení
  - lokální proměnné od okamžiku deklarace do konce bloku, ve kterém jsou deklarovány

- v přidělování paměti se primitivní datové typy významně liší od objektových typů

## 4.2.2. Objektové datové typy

- paměť se přiděluje použitím operátoru `new` a inicializují se následným voláním konstruktoru
- k přidělené paměti máme přístup pouze pomocí odkazu (reference)
  - reference vzniká deklarací: `Obdelnik obd;`
- na jeden objekt (jedno místo v paměti) může odkazovat více odkazů
  - změna stavu objektu vyvolaná zasláním zprávy prostřednictvím jednoho odkazu je současně změnou stavu objektu odkazovaného druhým odkazem

```
Obdelnik o1, o2;
o1 = new Obdelnik();
o2 = o1;
o1.setSirka(200);
int sirka = o2.getSirka();
```

- podle účelu – nikoliv podle použitého klíčového slova v deklaraci – se objektové typy dále dělí na odkazové a hodnotové

### 4.2.2.1. Odkazové datové typy (*reference data types*)

- neuvažujeme o jejich hodnotách, objekt představuje sám sebe
  - nemá smysl hovořit o ekvivalenci dvou různých objektů
  - jsou vzájemně nezastupitelné
  - `equals()` se nepřekrývá, tj. zůstává „přísné“ nastavení děděné z `Object`
- červený obdélník  $10 \times 20$ , který je momentálně na pozici  $0,0$  není ekvivalentní jinému červenému obdélníku momentálně stejných rozměrů a na stejně pozici
  - oba dva tyto objekty se na plátně překrývají, ale vzhledem k tomu, že jsou odkazy, mohou změnit pozici, velikost, barvu, ...
- typicky jsou **měnitelné** (*mutable*) – k jejich atributům existují setry

### 4.2.2.2. Hodnotové datové typy (*value data types*)

- objekt zastupuje nějakou hodnotu
  - objekty se stejnou hodnotou se mohou vzájemně zastoupit => má smysl hovořit o jejich ekvivalenci
    - ◆ `equals()` se překrývá
  - často jsou také navzájem porovnatelné ve smyslu větší–menší, předchůdce–následník
    - ◆ viz dále `Comparable` u kolekcí
- červená barva na elipse je tatáž, jako červená barva na obdélníku

- přebarvení elipsy neznamená změnu v objektu Barva, ale použití jiného objektu Barva
- další příklady – řetězce String, obalové typy Integer, časy a datumy, cokoliv podle vzoru Přepravka
- typicky jsou **neměnitelné (immutable)** – jejich atributy jsou označeny modifikátorem `final` a neexistují k nim setry
  - hodnotu, která jim byla přiřazena „při narození“, uchovávají a poskytují až do své „smrti“
  - tím se splňuje další bod konaktu `equals()` – konzistentnost – pokud dva objekty porovnáme v jeden okamžik, musíme dostat stejný výsledek i při porovnání později
  - pokud existují metody, které mají měnit hodnotu objektu, musejí vracet **jiný objekt** s touto změněnou hodnotou
  - proměnné (*mutable*) hodnotové typy jsou pro program nebezpečné, a proto je používáme pouze výjimečně, máme-li pro jejich použití opravdu pádné důvody
- ukázka chování neměnitelného String, kdy jeho změnou vznikne vždy nový objekt a též ukázka rozdílu mezi chováním `==` a `equals()`

```
String sa = "Ahoj";
String sb = sa;
boolean rovno, equals;
sa += "!";
// sa="Ahoj !"
rovno = (sa == sb + "!");
// false
equals = sa.equals(sb + "!");
// true
System.out.println("Shoda instancí: " + rovno +
    "\nShoda hodnot: " + equals);
```

- ukázka vracení jiného objektu známé Pozice

```
public Pozice posunO(int x, int y) {
    return new Pozice(this.x + x, this.y + y);
}
```

### 4.2.2.3. Obalové datové typy (*wrapper data types*)

- jedná se o speciální případy neměnitelných hodnotových objektových typů
- každý primitivní datový typ má svůj objektový protějšek
  - důvodem je mj. skutečnost, že v mnoha případech nelze primitivní datový typ použít – typicky v kolekcích
  - protože se jedná o objekty, mohou kromě základního uchovávání hodnoty poskytovat množství užitečných metod (např. `parseInt(String s)` – převod String na primitivní datový typ) a konstant (`MAX_VALUE`)
    - ◆ mají mnohem rozsáhlejší možnosti operací (viz dříve)
- jména objektových typů

- byte – Byte
  - short – Short
  - long – Long
  - float – Float
  - double – Double
  - boolean – Boolean
  - int – Integer
  - char – Character
- samozřejmě, že tyto typy mají přetíženou `equals()`
- vytvářejí se buď pomocí konstruktoru (`new Integer("123")`) nebo pomocí statické tovární metody (`getInteger("123")`)

## Poznámka

Toto je také ukázka, že použití statické tovární metody nevylučuje současné použití konstruktoru.

- pro převod na primitivní datový typ mají metodu `xxxValue()` – např. `intValue()`
- pro převod řetězce na primitivní datový typ (nutná akce při vstupech z klávesnice či z textových souborů) používají statickou metodu `parseXXX()`, např.:  

```
int cislo = Integer.parseInt("123");
```

### 4.2.2.3.1. Automatické převody mezi primitivními datovými typy a jejich obalovými třídami

- též *autoboxing* a *unboxing*
- pokud intenzivně pracujeme s obalovými třídami, je otravná práce s vytvářením objektů a získáváním hodnot
- od JDK 1.5 to umí překladač zajistit automaticky, tj. provést přechod z primitivního datového typu na příslušnou obalovací třídu a naopak
  - této možnosti by se mělo využívat s rozmyslem a jen tam, kde nemůže dojít k nedorozumění
  - stále platí, že `int` a `Integer` jsou dva zcela rozdílné typy

```
Integer i1 = new Integer(5);
Integer i2 = 6;
int j1 = i1.intValue();
int j2 = i2;
```

## 4.2.3. Práce s řetězci

### ■ tři třídy

- `String` – neměnitelný řetězec – viz dříve a též [PPA1-138]
- `StringBuffer` – měnitelný řetězec zabezpečený pro používání ve vláknech
  - ◆ pomalejší
- `StringBuilder` – měnitelný řetězec

### ■ měnitelné řetězce používáme, pokud je řetězec často a postupně vytvářen (budován)

- vyhneme se neustálému vytváření a rušení objektů, což zdržuje

### ■ typické použití je, že řetězec postupně vytvoříme a na závěr práce jej převedeme na typ `String` pomocí metody `toString()`

### ■ třídy `StringBuffer` a `StringBuilder` (dále jen společně S-B) mají mnoho metod stejných jako `String`, např.: (nejsou uváděny formální parametry, protože metody jsou většinou přetížené):

- `char charAt()`
- `int indexOf()`
- `int length()`
- `String substring()`

### ■ navíc mají S-B třídy metody pro změnu řetězce, zejména:

- `append(typ)` – přidá řetězcovou reprezentaci typu na konec
- `insert(int pozice, typ)` – vloží řetězcovou reprezentaci typu od zadанé pozice – zbytek posune
- `delete(int start, int end)` – vypustí znaky od start včetně do end vyjma
- `deleteCharAt(int index)` – vypustí znak na indexu
- `setCharAt(int index, char ch)` – nahradí původní znak na indexu novým znakem
- `replace(int start, int end, String str)` – nahradí původní znaky na pozici novými znaky, velikost původního a nového se nemusí rovnat

### ■ kromě toho vracejí metody `append()` a `insert()` odkaz na svou instanci, takže se mohou zřetězovat

```
public String poleToString(String nadpis, Object[] pole) {  
    StringBuilder sb = new StringBuilder(nadpis);  
    sb.append("\n");  
    for(int i=0; i < pole.length; i++ ) {  
        sb.append(i).append(": ").append(pole[i]).append("\n");  
    }  
}
```

```
    return sb.toString();  
}
```

## 4.3. Praktické náležitosti Java programů

- nepotřebujeme při experimentech pomocí BlueJ, protože ten je supluje
- v reálném životě nutné

### 4.3.1. Hlavní třída

- metoda, která je (mimo prostředí BlueJ) spuštěna jako první, je:

```
public static void main(String[] args)
```

- kde `String[] args` je pole parametrů předávaných z příkazové řádky
- podrobnosti viz [PPA1-142]

- každá třída může mít metodu `main()`

- to bylo dříve využíváno pro testovací účely

- ◆ prakticky se ale `main()` do tříd už téměř nikdy nedává – třída se testuje pomocí JUnit testů umístěných v jiné třídě

- z hlediska přehlednosti celého projektu se vytváří speciální třída

- jmeneuje se typicky `Hlavni` nebo `Applikace`

- ◆ důrazně doporučuji, aby ve jméně této třídy nebyly použity akcenty – činilo by to problémy při budoucím vytváření JAR souboru

- má jen jedinou metodu a tou je `main()`

- ◆ toto pravidlo se občas porušuje a ve třídě ještě bývají statické metody pro
      - zpracování parametrů předaných z příkazové řádky, tj. `args`
      - metoda `help()` či `navod()`, která vypíše stručný návod k použití

- v metodě `main()` je typicky několik málo příkazů, které vytvoří instanci skutečně důležité třídy a zavolají její metodu

- velmi často je v `main()` pouze jeden řádek kódu

- příklad – třída `Krajina`

```
/* *****  
 * Instance třídy {@code Krajina} představují příklad použití  
 * tvarů a stromů  
 *  
 * @author Pavel Herout  
 * @version 2.00.000
```

```

*/ 
public class Krajina {

    //== KONSTANTNÍ ATRIBUTY TŘÍDY =====
    private static final SpravcePlatna SP = SpravcePlatna.getInstance();

    //== KONSTANTNÍ ATRIBUTY INSTANCI =====
    private final Strom strom;

    //== PROMĚNNÉ ATRIBUTY INSTANCI =====
    // nemohou být konstantní, protože jsou vytvářeny v metodě vytvořDomek()
    private Obdelnik zed;
    private Trojuhelnik strecha;

    //#####
    //== KONSTRUKTORY A TOVÁRNÍ METODY =====

    /*****
     * Vytvoří obrázek stromu a domku a zobrazí je na plátno
     */
    public Krajina() {
        strom = new Strom(50, 50);
        strom.zobraz();
        vytvorDomek();
    }

    //== OSTATNÍ NESOUKROMÉ METODY INSTANCI =====

    /*****
     * Zobrazí animaci jednoho dne ztvárněnou pohybem slunce
     * slunce se po vytvoření posune doleva z plátna
     * -- nelze jej vytvořit mimo plátno
     * pak se přesune přes celé plátno směrem doprava
     *
     * @param rychlost rychlosť přesouvání
     */
    public void jedenDen(int rychlost) {
        IO.zprava("Proběhne jeden den rychlosťí " + rychlost);
        Presouvac pres = new Presouvac(rychlosť);
        Elipsa slunce = new Elipsa(0, 5, 40, 40, Barva.ZLUTA);
        slunce.posunVpravo(-40);
        SP.pridej(slunce);
        pres.presunNa(500, 5, slunce);
    }

    //== SOUKROMÉ A POMOCNÉ METODY INSTANCI =====
    /*****
     * Vytvoří domek a zobrazí jej na plátno
     */
    private void vytvorDomek() {
        strecha = new Trojuhelnik(180, 50, 200, 60, Barva.CERVENA);
        zed = new Obdelnik(200, 110, 160, 90, Barva.SVETLESEDA);
        SP.pridej(strecha);
    }
}

```

```

        SP.pridej(zed);
    }
}

```

## ■ a možná třída Hlavni

```

/*
 * Třída {@code Hlavni} je hlavní třídou projektu,
 * který zobrazí krajinu a nechá projít jeden den
 *
 * @author Pavel Herout
 * @version 2.00.000
 */
public class Hlavni {

    private Hlavni() {
        // prazdny konstruktor
    }

    /*
     * Zpracuje jeden parametr příkazového řádku - rychlost posuvu
     * není-li zadán žádný parametr, vypíše návod
     *
     * @param args Parametry příkazového řádku
     * @returns zadanou rychlost
     */
    private static int zpracujParametry(String[] args) {
        if (args.length == 0) {
            návod();
        }
        return Integer.parseInt(args[0]);
    }

    /*
     * Vypíše návod k použití
     * a ukončí násilně běh programu pomocí System.exit(1);
     */
    public static void návod() {
        System.out.println("Spuštění:\n" +
                           "    java Hlavni rychlost\n" +
                           "        kde rychlost <1, 10>");
        System.exit(1);
    }

    /*
     * Metoda, prostřednictvím níž se spouští celá aplikace.
     *
     * @param args Parametry příkazového řádku
     */
    public static void main(String[] args) {
        int rychlost = zpracujParametry(args);
        new Krajina().jedenDen(rychlost);
    }
}

```

```
//      Krajina krajina = new Krajina();
//      krajina.jedenDen(rychlos);
}
}
```

### 4.3.1.1. Překlad a spuštění z příkazové řádky

- protože jsou zdrojové kódy v UTF-8 (a javac očekává windows-1250), musíme tuto skutečnost javac sdělit přepínačem -encoding UTF-8
- stačí zadat příkaz k překladu pouze hlavní třídy projektu (tj. souboru Hlavni.java), pokud javac zjistí, že nemá přeloženou nějakou potřebnou třídu, přeloží si ji
- spuštění je možné dvěma způsoby
  - v konzolovém režimu: java Hlavni 4  
aplikace se spustí spolu s konzolovým oknem, do něž se vypisují texty pro standardní a chybový výstup
  - v okenním režimu: javaw Hlavni 4  
otevře se pouze okno aplikace bez konzoly, standardní i chybový výstup jsou zahazovány – předpokládá se, že program vše vypisuje do oken

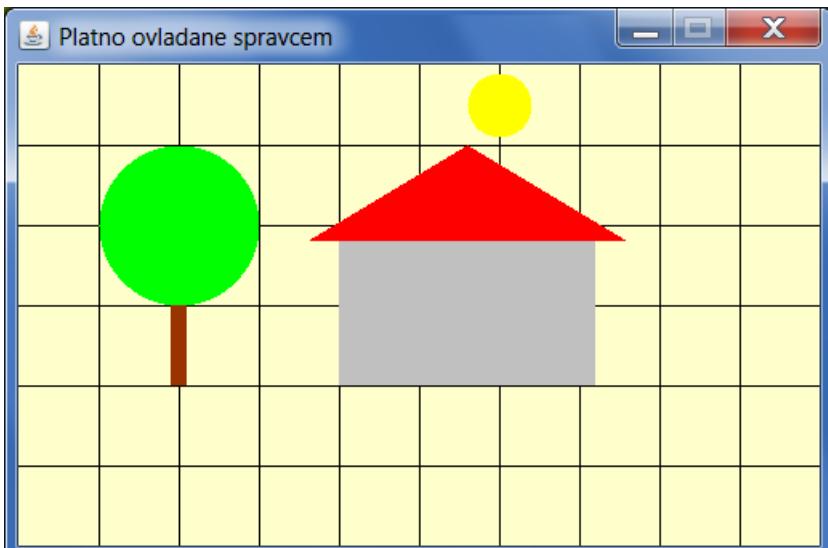
```
>javac -encoding UTF-8 Hlavni.java

>java Hlavni
Spuštění:
    java Hlavni rychlos
        kde rychlos <1, 10>

>java Hlavni 4

>javaw Hlavni 4
```

- výsledek práce programu

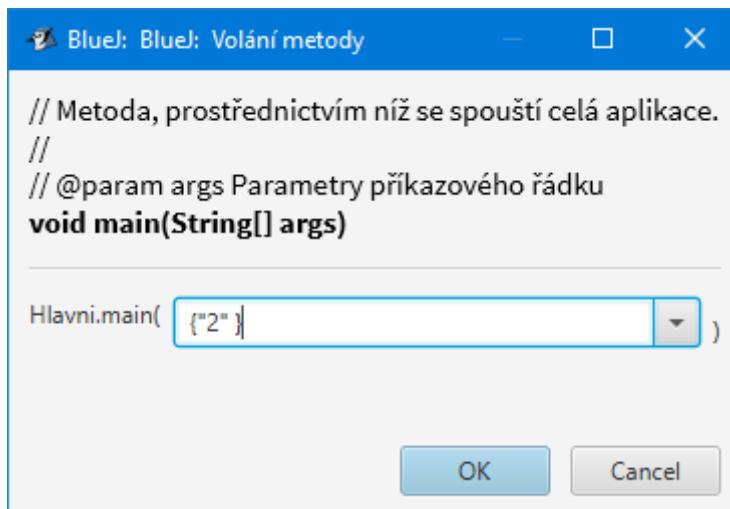


### 4.3.1.2. Spuštění z BlueJ

- použijeme příkaz z místní nabídky



- parametr příkazové řádky se zadá jako { "2" } tj. pole řetězců s jedním prvkem o hodnotě 2



### 4.3.2. JAR soubory

- aplikace většinou obsahuje více tříd, což znamená více .class souborů (předchozí příklad potřebuje 39 souborů)
- při přenosu programu na jiné místo lze snadno udělat chybu – nezkopírovat všechny soubory, změnit velikost písmen v názvu, změnit kódování názvů souborů apod.
- proto dává Java možnost využití JAR souborů (*Java ARchive*), ve kterých je vše zabalen do jednoho souboru
  - v Total Commander lze souboru JAR prohlížet klávesovou zkratkou Ctrl + PgDn
- soubor JAR je ZIP soubor – Pozor: nevytvářet jej pomocí zipovacích programů!
  - je doplněný o podadresář META-INF s textovým souborem MANIFEST.MF, který obsahuje informace o aplikaci
- povinné řádky v souboru MANIFEST.MF:

- Manifest-Version: 1.0 – verze manifestu je stále 1.0
- Created-By: 1.7.0\_06 (Oracle Corporation) – informace o verzi a dodavateli použité verze Javy
- Main-Class: Xyz – kde Xyz je úplný název (včetně případných balíků) hlavní třídy aplikace

## Poznámka

Není-li v JAR spustitelná aplikace, ale jenom o knihovna, řádek s uvedením hlavní třídy nemusí být uveden.

- v JAR souborech jsou názvy jednotlivých `.class` souborů v kódování UTF-8, tedy přenositelné
  - při rozbalení JAR souboru na konkrétní platformě se pak pro skutečné názvy souborů použije kódování platformy
  - problém je pouze s kódováním souboru `MANIFEST.MF`, tzn. se jménem hlavní třídy aplikace – proto by se v jejím názvu neměly objevovat akcenty

## Poznámka

Vytváření JAR souborů umožňují všechna vývojová prostředí. Další často používanou možností jsou nástroje Ant nebo Maven – viz KIV/UUR.

### 4.3.2.1. Vytváření JAR souboru z příkazové řádky

- k práci s JAR soubory slouží program `jar.exe`, který je součástí JDK
- má množství přepínačů, z nichž nám postačí:
  - `c` – vytvoření nového archivu
  - `m` – bude se vytvářet soubor `MANIFEST.MF` z informací uložených v následujícím textovém souboru to je v příkladu soubor `man.txt`, který snadno vytvoříme z příkazové řádky příkazem

```
echo Main-Class: Hlavni> man.txt
```

- `f` – určuje jméno souboru nového archivu – zde `jeden-den.jar`  
jméno nemusí být v žádném vztahu ke jménům Java tříd z projektu
- dále se na příkazové řádce udává, jaké soubory mají být součástí JAR – zde `*.class`  
často je zde `*.class *.java` – zahrne i zdrojové soubory

```
>echo Main-Class: Hlavni> man.txt
>jar cmf man.txt jeden-den.jar *.class
>java -jar jeden-den.jar
Spuštění:
    java Hlavni rychlosť
```

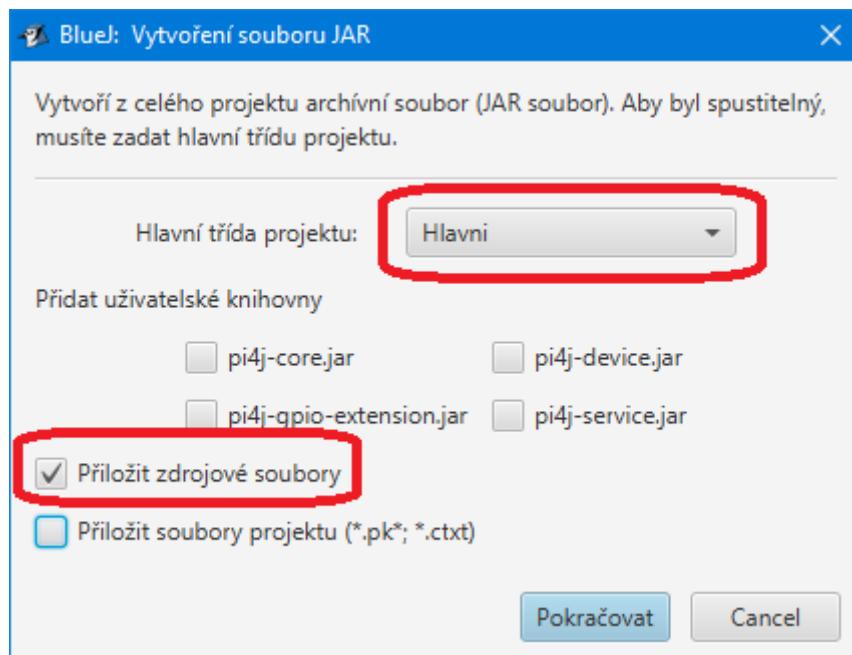
```
kde rychlost <1, 10>  
>java -jar jeden-den.jar 4
```

- spuštění je pomocí java, kterému se přepínačem -jar určí, že má spustit aplikaci z JAR souboru
  - java rozbalí JAR soubor a v souboru MANIFEST.MF zjistí, z jaké třídy má spustit metodu main()
- pokud potřebujeme JAR soubor rozbalit, použijeme příkaz

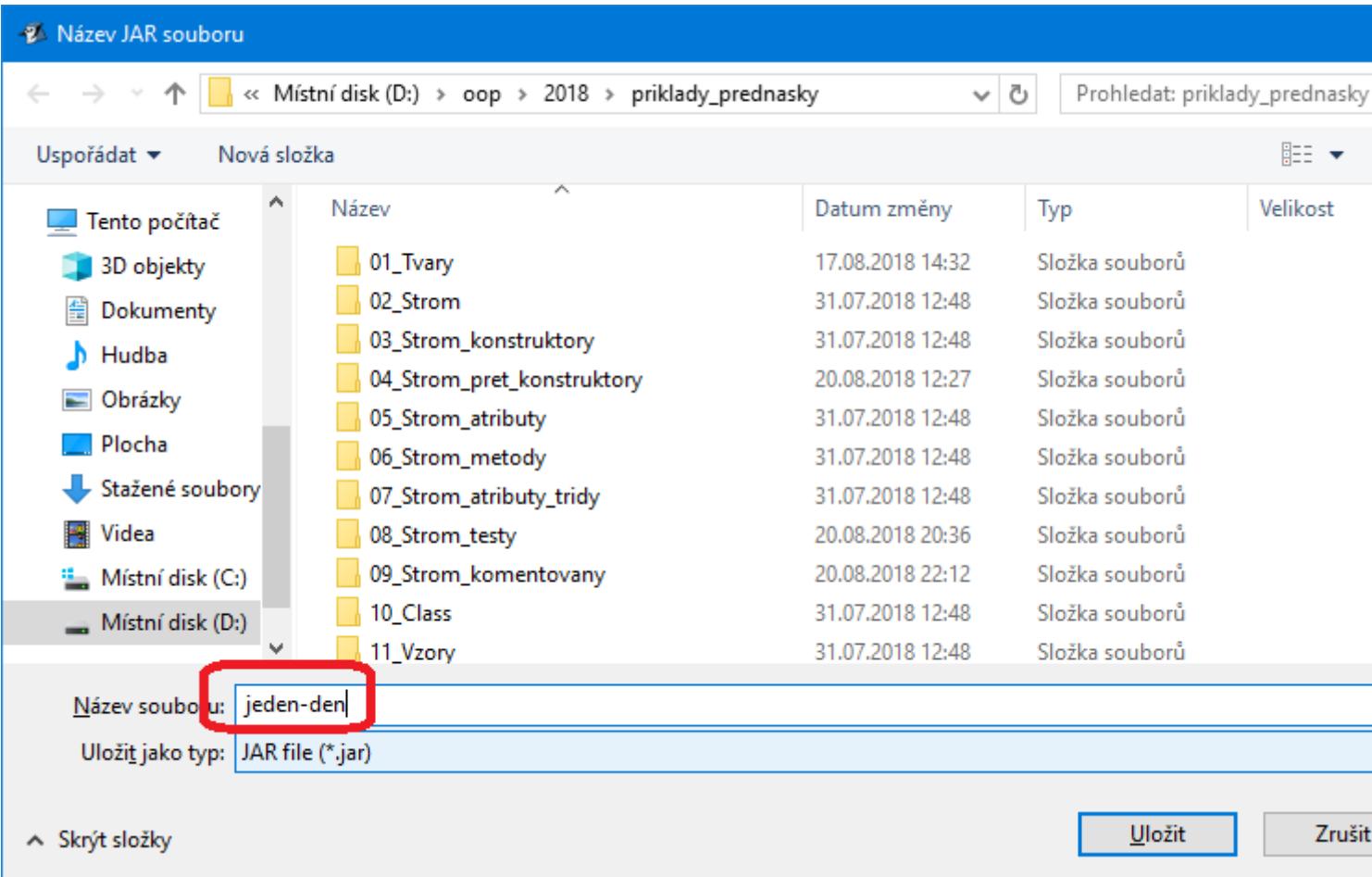
```
>jar xf jeden-den.jar
```

### 4.3.2.2. Vytvoření JAR z BlueJ

- použijeme Projekt / Vytvořit soubor JAR kde zvolíme třídu, ve které je main()



- pak je nutné zadat jméno výsledného JAR souboru (zde jeden-den)



- BlueJ sám zajistí vytvoření správného souboru MANIFEST.MF
  - do JAR souboru navíc přidá i podadresář doc s javadoc dokumentací
    - ◆ POZOR: Dokumentace ale musí být před tím vygenerována. To lze lehce provést tím, že se při editování libovolné třídy přepneme na zobrazení její dokumentace. To způsobí vygenerování kompletní dokumentace pro celý projekt.
- JAR soubor lze využít i k bezproblémovému přenosu zdrojových souborů na jinou platformu
  - tam použijeme příkaz Projekt / Otevřít Ne-BlueJ... a zadáme jméno JAR souboru
  - BlueJ vytvoří adresář se jménem JAR souboru (např. jeden-den) a do něj zkopiuje všechny zdrojové soubory

### 4.3.3. Balíky

- v rozsáhlých aplikacích (rozsáhlý je více než cca 10 tříd) nastávají problémy
  - souborů v adresáři je mnoho – lze se v nich špatně vyznat
  - možné konflikty jmen, zejména pokud spolupracuje více lidí – řešení **jmenné prostory**
    - ◆ v konkrétním jmenném prostoru si každý může pojmenovávat jak chce
    - ◆ první jmenný prostor byla pro nás třída – atributy jedné třídy se mohly jmenovat stejně, jako atributy jiné třídy

- obtížné vytváření knihoven jen z určitých tříd
- řešením se jeví použití balíků (*packages*) což je ekvivalent jmenných prostorů
- balíky je vhodné používat již od začátku práce – výjimkou jsou triviální projekty, které používají **defaultní balík**
- balík může obsahovat podbalíky
  - název balíku se skládá z názvu rodičovského balíku následovaného tečkou a vlastním názvem daného balíku
  - balík na vrcholu hierarchie se nazývá kořenový
  - např. `java.lang` a `java.util.zip` – u obou je kořenovým balíkem `java`
- při uložení přeložených souborů na disku musí umístění souborů v adresářích odpovídat jejich umístění v balíku
  - každému balíku je přiřazen jeho adresář
  - adresář se musí jmenovat přesně stejně jako daný balík (včetně velikosti písmen)
  - podbalíku daného balíku je přiřazen podadresář rodičovského adresáře
- jedna konkrétní třída může patřit jen do jediného balíku
- aplikace může využívat balíky z několika různých adresářů
  - typicky svoje balíky a balíky z Java Core API
  - často k nim přibývají ještě balíky knihoven od třetích stran
  - pro překlad i spuštění je nutné, aby byly všechny adresáře kořenových balíků uvedeny v systémové proměnné CLASSPATH
    - ◆ toto je největší problém při překladu a spuštění z příkazové řádky
    - ◆ vývojová prostředí splní tuto povinnost za nás
- kořenový adresář balíků by měl být odvozen dle internetové adresy výrobce podle vzoru `cz.zcu.fav.kiv.herout` nebo `cz.zcu.fav.kiv.jmenoprojektu`
  - v názvech se používají jen malá písmena
  - vlastní názvy balíků musí být významové
- balík by měl být „jednoúčelový“, tj. obsahovat pouze třídy a rozhraní přímo se vážící k řešení stejného problému
  - nelze přesněji stanovit, kolik tříd by mělo v balíku maximálně být – rozumný počet je asi 10, kdy je možné se ještě orientovat v diagramu tříd
  - Java Core API má balíky o jedné třídě (např. `javax.lang.model`) i o cca 150 třídách (`javax.swing`)

### 4.3.3.1. Použití balíků ve zdrojových souborech

- zdrojový kód každé třídy musí začínat příkazem `package hierarchie.balíků;`
  - před tímto příkazem smí být pouze komentáře a bílé znaky
- pokud není příkaz `package` uveden, patří daná třída do defaultního balíku
  - není to plnohodnotný balík, má omezení a nevýhody
  - používá se velmi sporadicky, např. pro drobné programy, prvotní ověření nápadů apod.
- název třídy pomocí plně kvalifikovaného jména je její název včetně celé hierarchie balíků
  - tento název nelze s ničím zaměnit
  - např. `java.lang.String`
- uvnitř balíku se na třídu lze odvolávat jejím vlastním názvem bez kvalifikace názvem jejího balíku
  - bez kvalifikace se lze odvolávat i na třídy z balíku `java.lang`
- při posílání zprávy třídě z jiného balíku je třeba uvádět její plný název včetně kvalifikace balíkem

```
java.math.BigInteger bigInt = new java.math.BigInteger("123");
```
- tento způsob ale výrazně prodlužuje kód a znepřehledňuje jej, proto se používají importy

### 4.3.3.2. Importy balíků

- příkaz `import` musí být na samém začátku zdrojového kódu – před ním je jen `package` a komentáře
- má dvě verze:
  - `import java.io.File;` – importuje jednu konkrétní třídu `File`
  - `import java.io.*;` – importuje všechny třídy balíku `java.io` (včetně třídy `File`), nikoliv však případné podbalíky
    - ◆ jednodušší a pohodlnější způsob, který se však nepoužívá
    - ◆ moderní IDE dokáží vygenerovat seznam importů prvního typu – výhodou je, že v programu pracujeme jen se skutečně potřebnými třídami
- ukázka zkrácení kódu

```
import java.math.BigInteger;  
...  
BigInteger bigInt = new BigInteger("123");
```

- od JDK 1.5 existuje možnost **statického importu**

- možnost importovat i statické členy jiných tříd a používat je pak bez kvalifikace

- ♦ ve statických importech se jde o úroveň níže (členy třídy) než předchozí typ importu (celé třídy)
- má opět dvě verze:
  - ♦ `import static java.lang.Math.PI;` – importuje statický člen PI třídy Math
  - ♦ `import static java.lang.Math.*;` – importuje všechny statické členy třídy java.lang.Math
- statický import obecně znepřehledňuje program
  - ♦ používá se pouze u členů, které se používají velice často a u nichž nehrozí vyvolání dojmu, že se jedná o členy dané třídy
  - ♦ jsou typická použití, jedním z nich je `import static org.junit.Assert.*;`, který se používá v JUnit testech verze 4
    - umožní např. používat jednoduše statickou metodu `assertEquals()`

```
assertEquals(true, strom.isPodzem());
```

tu si v JUnit testech nikdo s ničím jiným nesplete

– bez statického importu by bylo nutné

```
Assert.assertEquals(true, strom.isPodzem());
```

- příklad bez statického importu

```
System.out.println(Math.PI);
double logE1 = Math.log(Math.E);
```

- se statickým importem

```
import static java.lang.Math.PI;
import static java.lang.Math.*;
...
System.out.println(PI);
double logE2 = log(E);
```

### 4.3.3.3. Přístupová práva

- k balíkům se váží i přístupová práva, která byla dosud
  - `public` – všichni z vnějšku to vidí
  - `private` – nikdo z vnějšku to nevidí
- pokud u třídy, atributu či metody neuvedeme modifikátor přístupového práva, bude brán jako **přátelský** (občas se označuje jako *package private*)
  - třídě bez `public` se říká **neveřejná třída**
    - ♦ její sporadická výhoda je, že může být uložena v jednom souboru společně s jinou třídou

- tento člen je přístupný všem třídám ze stejného balíku, zhruba jako by měl přístupové právo public
  - to, co možná vypadá jako výhoda, je velmi nebezpečná možnost popírající autorizovaný přístup, proto ji nepoužíváme
- detailní podrobnosti o přístupových právech viz Herout, P.: Učebnice jazyka Java str. 234

# Kapitola 5. Dědičnost

- dědění je vztah obecný–speciální
- potomek je víc specializovaný – má vlastnosti a schopnosti navíc
- instance potomka se může kdykoliv vydávat za instanci rodiče
- je mnoho způsobů, jak zajistit dědičnost

## 5.1. Typy dědění

### ■ dědění typů

- potomek dodrží všechny vlastnosti a schopnosti předka, tj. převezme jeho signaturu a dodrží jeho kontrakt
- o implementaci se musí postarat sám
- může se proto kdykoliv plnohodnotně vydávat za předka
- např.: třída implementující nějaké rozhraní nebo rozhraní dědí jiné rozhraní

### ■ dědění implementace

- potomek převezme od předka jeho implementaci
- implementace je již hotova – převzaté metody nemusí definovat sám
  - ◆ ale může je předefinovat (překrýt), tzn. napsat znovu jejich implementaci
  - ◆ nebezpečí: při přizpůsobování zděděných entit potřebám potomka není občas dodržen kontrakt předka
- např.: všechny třídy přebírají základní metody od třídy `Object` a často se překrývá metoda `toString()`

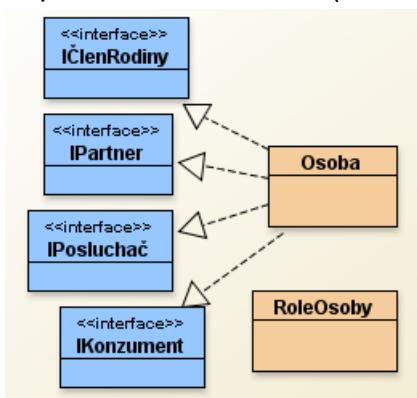
### ■ ještě se udává třetí typ dědění – **přirozené dědění též dědění podstaty**

- říká, jak chápeme vztah obecný–speciální bez (přímého) ohledu na programování
  - ◆ instance potomka je speciální případ instance předka, kdy většinou nerozšiřujeme, ale naopak omezujeme
    - kruh chápeme jako speciální případ elipsy
    - čtverec jako speciální případ obdélníku, který má obě dvě strany stejně dlouhé
- při dědění podstaty ale pravděpodobně později narazíme na problémy se správnou implementací takto pojaté dědičnosti
- *Liskov substitution principle* – potomek musí být kdykoli schopen zastoupit předka; přetypování na předka nesmí být v rozporu s logikou aplikace

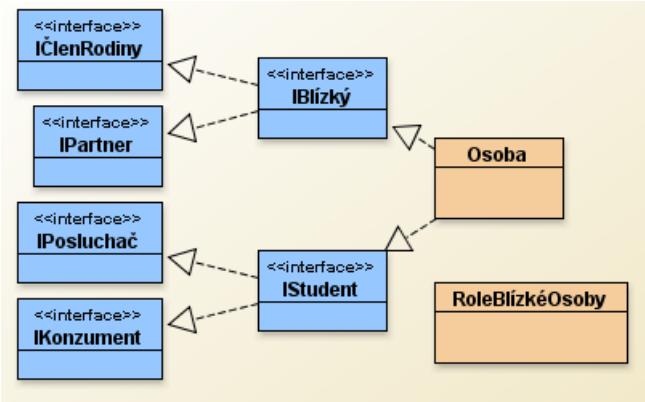
- ◆ dodržování tohoto principu znemožní vytvářet nepřirozené případy dědění, které by v budoucnu dělaly problémy
- ◆ někdy tyto případy nejsou na první pohled zřejmé – čtverec je speciální typ obdélníka
  - ale čtverec nemůže obdélník kdykoliv zastoupit, protože obdélníku lze nastavovat jakoukoliv stranu nezávisle na druhé
  - v tomto případě by se mělo použít skládání (kompozice) – viz dále
- ◆ nebo naopak – obdélník je speciální typ čtverce, kterému přibyla jedna strana navíc

## 5.2. Dědičnost rozhraní

- jedná se o dědičnost typů – co objekt umí
  - s tímto způsobem dědičnosti nejsou žádné skryté problémy
- implementaci rozhraní (nebo více rozhraní) třídou – viz dříve – považujeme za **dědění typů**



- implementovaná rozhraní počítáme mezi předky implementující třídy
- a naopak implementující třídy vystupují jako potomci svých implementovaných rozhraní
- přetypování potomka na předka provede překladač automaticky
  - ◆ instance třídy implementující rozhraní se může kdykoliv vydávat za instanci kteréhokoliv z implementovaných rozhraní
- máme-li množinu objektů se speciální vlastností (např. `přijmiDárek()`), je vhodné pro ně definovat speciální typ, který tuto vlastnost zahrnuje – můžeme:
  - definovat další rozhraní – `IObdarovávatelný` by byl pátým rozhraním
  - zdědit rozhraní `IČlenRodiny` a `IPartner` do nového rozhraní, které je spojuje – `IBlízký`
    - ◆ většinou zjednoduší vazby v programu – pokud by místo třídy `Osoba` existovaly dvě třídy `Muž` a `Žena`, pak se zjednoduší vztahy implementace
- příklad – `IBlízký` splňuje podmítku přirozeného dědění, protože dokáže zastoupit jak člena rodiny, tak i partnera
  - jeho specializací navíc je, že může dostávat dárky



- používáme klíčové slovo `extends` následované seznamem předků

```

/*
 * Instance rozhraní {@code IBlízký} představují
 * někoho, kdo je člen rodiny nebo partner
 * a navíc může přijmout dárek
 */
public interface IBlízký extends IČlenRodiny, IPartner {
    public void přijmiDárek();
}
  
```

- občas se vyskytne případ, že zděděné rozhraní nic nepřidává, pouze spojuje – je to další případ značkovacího rozhraní (viz dříve)

```

/*
 * Instance rozhraní {@code IStudent} představují studenta,
 * který je zároveň posluchačem a konzumentem v menze
 */
public interface IStudent extends IPosluchač, IKonzument {
    // prázdné
}
  
```

- ve třídě `Osoba` se zjednoduší hlavička, jinak zůstane vše stejné jako v předchozím příkladu

```

/*
 * Instance třídy {@code Osoba} představují osobu schopnou mnoha rolí
 */

public class Osoba implements IStudent, IBlízký {
    @Override
    public void uklidPoSobě() {
        System.out.println("uklízím");
    }

    @Override
    public void pišSiPoznámky() {
        System.out.println("studuju");
    }

    @Override
  
```

```

public void jezJidlo() {
    System.out.println("jím");
}

@Override
public void dejMiPusu() {
    System.out.println("líchám");
}

@Override
public void přijmiDárek() {
    System.out.println("mám radost z dárku");
}
}

```

- v příkladu použití je vidět, že `IBlízký` může skutečně zastoupit `IČlenRodiny` a `IPartner`
  - přetypování potomka na předka provede překladač automaticky

```

public class RoleBlízkéOsoby {
    public static void main(String[] args) {
        IBlízký příjemceDárku = new Osoba();
        příjemceDárku.přijmiDárek();
        příjemceDárku.uklidPoSobě();
        příjemceDárku.dejMiPusu();
    }
}

```

## 5.3. Skládání

- též kompozice
- toto není případ dědičnosti, ale případ, jak se ve velké většině případů dědičnosti implementace správně vyhnout
- podle Liskové principu musí být potomek kdykoli schopen zastoupit předka
  - někdy tyto případy nejsou na první pohled zřejmé – čtverec je speciální typ obdélníka, který má obě dvě strany stejně dlouhé
    - ◆ ale čtverec nemůže obdélník zastoupit, protože obdélníku lze nastavovat nezávisle na druhé jakoukoliv stranu
- při dědění tříd (viz později) dědíme implementaci, tj. kód, což **značně šetří psaní**
  - to je jeden z hlavních důvodů, proč se dědění tříd využívá nesprávně
- při skládání musíme znova napsat implementaci všech potřebných metod
  - v naprosté většině případů je implementace velmi jednoduchá a představuje jen jednu řádku volání (často stejně pojmenované) metody
  - to znamená, že se nejedná o příliš tvůrčí práci – opět to podporuje myšlenku dědění tříd

## Poznámka

Se skládáním jsme se již setkali v případu třídy Strom.

- příklad – Ctverec bude sestaven z pouze jednoho Obdélnika, který má 23 metod

- většinu z nich bychom měli napsat znovu
- jejich minimální počet je určen implementovanými rozhraními
- v této verzi Čtverec nebude všech 23 metod implementováno

```
*****  
 * Instance třídy {@code Ctverec} představují čtverce  
 *  
 * @author Pavel Herout  
 * @version 3.00.000  
 */  
public class Ctverec implements IKresleny, IPosuvny {  
  
//== KONSTANTNÍ ATRIBUTY TŘÍDY ►  
=====  
 /** Implicitní rozměr */  
 private static final int IMPLICITNI_ROZMER = 100;  
  
 /** Počáteční barva nakreslené instance v případě,  
 * kdy uživatel žádnou požadovanou barvu nezadá  
 */  
 public static final Barva IMPLICITNI_BARVA = Barva.ZLUTA;  
  
//== KONSTANTNÍ ATRIBUTY INSTANCÍ =====  
 /** skutečná reprezentace pomocí již existujícího tvaru */  
 private final Obdelnik obdelnik;  
  
//#####  
//== KONSTRUKTORY A TOVÁRNÍ METODY =====  
  
*****  
 * Vytvoří novou instanci se polohou [0, 0] a rozměrem 100  
 * a implicitní barvou.  
 */  
 public Ctverec() {  
     this(0, 0, IMPLICITNI_ROZMER);  
 }  
  
*****  
 * Vytvoří novou instanci se zadanou polohou a rozměrem  
 * a implicitní barvou.  
 *  
 * @param x      x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna  
 * @param y      y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna  
 * @param strana velikost strany vytvářené instance, strana >= 0  
 */  
 public Ctverec(int x, int y, int strana) {
```

```

    this( x, y, strana, IMPLICITNI_BARVA );
}

/********************* Vytvoří novou instanci se zadanými rozměrem, polohou a barvou. ****
 * @param x      x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna
 * @param y      y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna
 * @param strana velikost strany vytvářené instance, strana >= 0
 * @param barva   Barva vytvářené instance
 */
public Ctverec(int x, int y, int strana, Barva barva) {
    obdelnik = new Obdelnik(x, y, strana, strana, barva);
}

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCI =====
/********************* Vrátí x-ovou souřadnici pozice instance.
 * @return x-ová souřadnice.
 */
public int getX() {
    return obdelnik.getX();
}

/********************* Vrátí y-ovou souřadnici pozice instance.
 * @return y-ová souřadnice.
 */
public int getY() {
    return obdelnik.getY();
}

/********************* Vrátí velikost strany instance.
 * @return Velikost strany v bodech
 */
public int getStrana() {
    return obdelnik.getSirka();
}

/********************* Nastaví velikost strany instance.
 * @param strana   Velikost strany v bodech
 */
public void setStrana(int strana) {
    obdelnik.setRozmer(strana, strana);
}

```

```

* Vrátí barvu instance.
*
* @return Instance třídy Barva definující nastavenou barvu.
*/
public Barva getBarva() {
    return obdelnik.getBarva();
}

/***** Nastaví novou barvu instance.
*
* @param nova Požadovaná nová barva.
*/
public void setBarva(Barva nova) {
    obdelnik.setBarva(nova);
}

//== PŘEKRYTÉ METODY IMPLEMENTOVANÝCH ROZHRANÍ ►
=====

/***** Za pomoci dodaného kreslítko vykreslí obraz své instance
* na animační plátno.
*
* @param kreslitko Kreslitko, kterým se instance nakreslí na plátno. ▶
*/
@Override
public void nakresli(Kreslitko kreslitko) {
    obdelnik.nakresli(kreslitko);
}

►
/***** Vrátí instanci třídy Pozice s pozicí instance.
*
* @return Pozice s pozicí instance.
*/
@Override
public Pozice getPozice() {
    return new Pozice(getX(), getY());
}

/***** Nastaví novou pozici instance.
*
* @param x Nová x-ová pozice instance
* @param y Nová y-ová pozice instance
*/
@Override
public void setPozice(int x, int y) {
    obdelnik.setPozice(new Pozice(x, y));
}

```

```

//== NOVĚ ZAVEDENÉ METODY INSTANCI ►
=====

/*********************  

 * Přesune instanci o zadaný počet bodů vpravo,  

 * při záporné hodnotě parametru vlevo.  

 *  

 * @param vzdalenost Vzdálenost, o kterou se instance přesune.  

 */  

public void posunVpravo(int vzdalenost) {  

    obdelnik.posunVpravo(vzdalenost);  

}

//== PŘEKRYTÉ KONKRÉTNÍ METODY RODIČOVSKÉ TŘÍDY ►
=====

/*********************  

 * Převede instanci na řetězec. Používá se především při ladění.  

 *  

 * @return Řetězcová reprezentace dané instance.  

 */  

@Override  

public String toString()  

{  

    return this.getClass().getSimpleName() +  

        ": x=" + obdelnik.getX() + ", y=" + obdelnik.getY() +  

        ", strana=" + obdelnik.getSirka() +  

        ", barva=" + obdelnik.getBarva();  

}
}

```

## 5.4. Dědění tříd

- též **dědění implementace**
- při práci s objekty daného typu často odhalíme skupiny instancí se **speciálními**, avšak pro celou podskupinu společnými vlastnostmi
  - auta můžeme dělit na osobní, nákladní, dodávky, ...
  - osoby dělíme na muže a ženy
  - geometrické tvary můžeme dělit na elipsy, čtyřúhelníky a trojúhelníky
  - čtyřúhelníky můžeme dělit obdélníky, čtverce, kosodélníky, ...
- to, že je daný objekt členem speciální podskupiny, nijak neovlivňuje jeho členství v původní skupině
  - potomek se může kdykoliv vydávat za předka
  - např. čtverec se může vydávat za čtyřúhelník nebo za geometrický tvar

- někdy se postupuje obráceně – u řady různých druhů objektů nacházíme společné vlastnosti a definujeme pak společné skupiny
  - lidé, psi, delfíni patří do skupiny savců
  - auta, kola, vlaky, letadla, lodě apod. jsou dopravní prostředky
  - v objektově orientovaných programech je vše považováno za objekt
- různé terminologie obecný–speciální, které vyjadřují totéž
  - rodičovský typ – dceřiný typ
  - rodičovský typ – typ potomka
  - předek – potomek
  - bázový typ – odvozený typ
  - nadtyp – podtyp
  - nadřída – podřída

## 5.4.1. Principy dědičnosti implementace

- potomek převeze instanci předka jako svůj podobjekt a současně převeze i jeho rozhraní
  - tím potomek získá již hotovou (mnohdy značně rozsáhlou) implementaci některých potřebných atributů, metod a typů
- překladač zabezpečí automatické převzetí signatury, programátor má na starosti dodržení příslušného konaktu
- tři důvody, proč se používá dědičnost implementace (důvody se často překrývají):
  1. specializace – Auto versus OsobníAuto
    - případně je předkem **abstraktní třída** – viz dále
  2. překrývání metod a polymorfismus – existuje více potomků, kteří mají lehce odlišné chování na některé zprávy
    - společný předek Student, potomci Bakalář a Magistr a zpráva kvalifikačníPráce()
    - ◆ u prvního je to bakalářka, u druhého diplomka
  3. znovupoužití kódu – nejméně dobrý důvod – Ctverec versus Obdélník
    - autoři potomků dědících implementaci se často soustředí pouze na to, aby získali implementaci „zdarma“, a zcela ignorují nutnost dodržení konaktu a invariantů předka
      - ◆ správný (ale pracný) postup – viz příklad výše se skládáním
      - ◆ důsledek: instance potomka se nemůže plnohodnotně vydávat za instanci předka – poruší tak Liskové princip a konzistenci programu

- při rozšiřování programu pak vede k nelogickým konstrukcím

- nebezpečí – navržený program funguje, ale při změnách vyvstávají zásadní problémy

- nevhodné použití dědičnosti implementace

- obdélník je potomek bodu

- ◆ obdélník zdědí souřadnice jednoho rohu a přidá souřadnice protilehlého rohu

- kruh je potomek bodu

- ◆ kruh je implementován jako bod doplněný o poloměr

- kruhová výseč je potomek kruhu

- ◆ výseč je implementována jako kruh doplněný o úhel výseče

- kvádr je potomek obdélníka

- ◆ obdélník doplněný o výšku

- ◆ kvádr má šest obdélníků a při pozdější potřebě např. obarvit strany by došlo ke kuriózní situaci, že jedna strana je již implicitní

- z Java Core API `java.util.Stack` je potomek `java.util.Vector`

- ◆ zásobník má kontrakt jen přidat na vrchol a ubrat z vrcholu

- ◆ díky dědičnosti od `Vector`, ale zásobník může vkládat či vybírat zevnitř zásobníku – porušuje se obecný kontrakt

- ◆ tuto chybu již nelze odstranit, protože již bylo zveřejněno rozhraní a `Stack` se začal používat

- jsou dva používané způsoby (velmi často se kombinují), jak bránit používání:

- označení ve zdrojovém kódu `deprecated` (odmítaný, zavrhovaný)

- informace v javadoc dokumentaci: „*A more complete and consistent set of LIFO stack operations is provided by the Deque interface*“

- dědičnost tříd se použije v případě, že mezi třídami existuje vztah „*is-a*“ („je nějaký“)

- má-li být B potomkem A, pak si musíme kladně odpovědět na otázku: „Je každý B také A?“

- neodpovíme-li kladně, dědičnost NEpoužijeme

- při vhodném použití dědičnosti:

- vše, co nám z předka vyhovuje, to bez dalšího úsilí převezmeme

- co nám chybí, to dodáme

- co nám nevyhovuje (zejména metody), to překryjeme – viz dříve `toString()`

- jednonásobné dědění implementace v Javě

- v Javě lze dědit více typů – implementovat více rozhraní, dědit více rozhraní
- dědit rodičovskou třídu při dědění implementace lze ale pouze jednu
  - ◆ každá uživatelem definovaná třída má vždy právě jednoho předka
  - ◆ je jasný hierarchický strom, na jehož vrcholku je třída `Object`
  - ◆ vícenásobná dědičnost (možná např. v C++) přináší více problémů než výhod
    - navíc je možné ji v Javě pomocí rozhraní obejít

## 5.4.2. Realizace dědičnosti

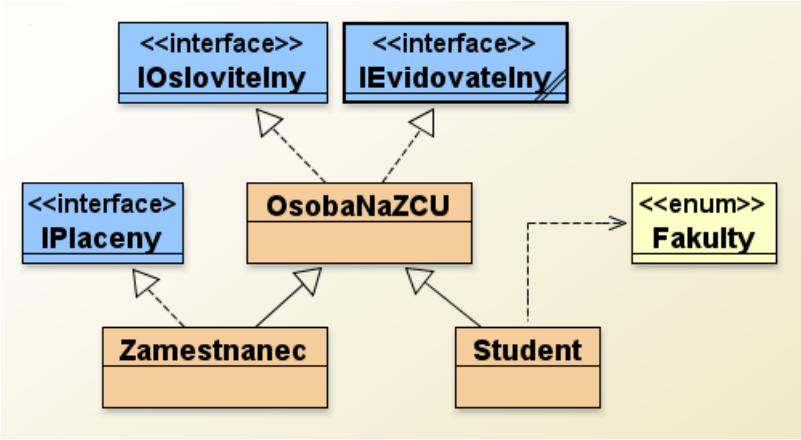
- odvození od rodičovské třídy deklaruje dceřiná třída v hlavičce za svým názvem klíčovým slovem `extends` následovaným názvem rodičovské třídy
- dědictví od třídy `Object` se uvádět nemusí
  - a naopak, nemá-li třída v hlavičce uvedeného předka, je přímým potomkem třídy `Object`
- případná deklarace implementace rozhraní se uvádí až za deklarací dědičnosti

```
public class Potomek extends Předek
    implements IRozhraní1, IRozhraní2 {
```

- modifikátor atributů a metod `protected` – „chráněný“
  - doplňuje dvojici `private` – `public`
  - atributy a metody s tímto modifikátorem jsou viditelné ve všech potomcích třídy (z libovolného balíku) a u všech tříd ze stejného balíku
  - pomocí `protected` označujeme to, co si myslíme, že by mohli případní potomci přímo využívat, ale před vnějším světem to má být skryto

### 5.4.2.1. Příklad bezproblémové dědičnosti

- v dědičnosti implementace mohou být mnohé záludnosti – viz později
  - následující příklad bude ukazovat typický jednoduchý bezproblémový případ
- cíl příkladu je ukázat
  - vše, co z rodičovské třídy vyhovuje, je v potomku ihned (bezpracně) dostupné
  - vše, co chybí, lze dodat
  - vše co nevyhovuje, lze opravit
  - vyřešení vztahu potomka a předka pomocí konstruktoru
  - situace, pokud předek či potomek navíc implementuje rozhraní
  - za co se může potomek vydávat



## ■ rozhraní IOslovitelny

```

/** Lze získat jméno */
public interface IOslovitelny {
    public String getJmeno();
}

```

## ■ rozhraní IEvidovatelny

```

/** Lze získat jednoznačný identifikátor */
public interface IEvidovatelny {
    public String getID();
}

```

## ■ třída OsobaNaZCU

```

/** rodičovská třída */
public class OsobaNaZCU implements IOslovitelny, IEvidovatelny {
    private final String jmeno;
    private final String osobniCislo;

    public OsobaNaZCU(String jmeno, String osobniCislo) {
        this.jmeno = jmeno;
        this.osobniCislo = osobniCislo;
    }

    @Override      // z Object
    public String toString() {
        String jmenoTridy = this.getClass().getSimpleName();
        return jmenoTridy + ": " + jmeno + ", ID=" + osobniCislo;
    }

    @Override      // z IOslovitelny
    public String getJmeno() {
        return jmeno;
    }

    @Override      // z IEvidovatelny
    public String getID() {

```

```

        return osobniCislo;
    }
}
```

## ■ možné testy nad třídou OsobaNaZCU

```

@Test
public void testKonstruktoru() {
    OsobaNaZCU os1 = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybné jméno: ", "Moudrý", os1.getJmeno());
    assertEquals("Chybné ID: ", "10321", os1.getID());

    OsobaNaZCU os2 = new OsobaNaZCU("Pilná", "A10B9999P");
    assertEquals("Chybné jméno: ", "Pilná", os2.getJmeno());
    assertEquals("Chybné ID: ", "A10B9999P", os2.getID());
}

@Test
public void testToString() {
    OsobaNaZCU os = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybná informace: ", "OsobaNaZCU: Moudrý, ID=10321",
                 os.toString());
}

@Test
public void testIOslovitelny() {
    IOslovitelny os = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybné jméno: ", "Moudrý", os.getJmeno());
}

@Test
public void testIEvidovatelny() {
    IEvidovatelny os = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybné ID: ", "10321", os.getID());
}
```

- je vidět, že instance třídy OsobaNaZCU se může vydávat za instanci

- ◆ OsobaNaZCU
- ◆ IOslovitelny
- ◆ IEvidovatelny

## ■ rozhraní IPlaceny

```

/** Lze získat informaci o výši platu */
public interface IPlaceny {
    public int getPlat();
}
```

## ■ třída Zamestnanec

```

public class Zamestnanec extends OsobaNaZCU implements IPlaceny {
    private int plat;

    public Zamestnanec(String jmeno, String osobniCislo, int plat) {
        super(jmeno, osobniCislo);
        setPlat(plat);
    }

    public void setPlat(int plat) {
        if (plat >= 0) {
            this.plat = plat;
        }
        else {
            throw new IllegalArgumentException("Plat < 0");
        }
    }

    @Override      // z IPlaceny
    public int getPlat() {
        return plat;
    }

    @Override      // z OsobaNaZCU
    public String toString() {
        String zakladniInfo = super.toString();
        return zakladniInfo + ", plat=" + plat;
    }
}

```

- aniž je to ve zdrojovém kódu uvedeno, dědí implementaci rozhraní IOslovitelny a IEvidovatelny
  - přidává atribut plat
  - protože ve třídě OsobaNaZCU není bezparametrický konstruktor, musí být v konstruktoru Zamestnanec pomocí super() volán konstruktor OsobaNaZCU(String jmeno, String osobniCislo)
  - v metodě setPlat(int plat) je ukázáno, jak řešit nevhodnost parametrů pomocí vyhození výjimky
    - ◆ to nemá s problematikou dědičnosti nic společného
  - implementace nového rozhraní IPlaceny je pomocí metody getPlat()
    - ◆ toto je přidaná zcela nová funkčnost potomka oproti předu
  - již ne zcela vyhovující metoda toString() je překryta, ale v jejím těle je pomocí super.toString() vyvolána metoda předka
    - ◆ ve skutečnosti je tedy metodě toString() rozšířena funkčnost
- možné testy nad třídou Zamestnanec

```

@Test
public void testToString() {
    OsobaNaZCU os = new Zamestnanec("Moudrý", "10321", 20000);

```

```

        assertEquals("Chybná informace: ", "Zamestnanec: Moudrý,
                      ID=10321, plat=20000", os.toString());
    }

@Test
public void testChybnyPlat() {
    IPlaceny os = new Zamestnanec("Moudrý", "10321", -100);
    assertEquals("Chybný plat: ", -100, os.getPlat());
}

@Test
public void testZmenaPlatu() {
    Zamestnanec os = new Zamestnanec("Moudrý", "10321", 20000);
    os.setPlat(30000);
    assertEquals("Chybný plat: ", 30000, os.getPlat());
}

@Test
public void testIPlaceny() {
    IPlaceny os = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybný plat: ", 20000, os.getPlat());
}

@Test
public void testIOslovitelny() {
    IOslovitelny os = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybné jméno: ", "Moudrý", os.getJmeno());
}

@Test
public void testIEvidovatelny() {
    IEvidovatelny os = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybné ID: ", "10321", os.getID());
}

```

- je vidět, že instance třídy `Zamestnanec` se může vydávat za instanci

- ◆ `OsobaNaZCU`
- ◆ `IPlaceny`
- ◆ `Zamestnanec`
- ◆ `IOslovitelny`
- ◆ `IEvidovatelny`

## ■ výčtový typ Fakulty

```

public enum Fakulty {
    FAV, FEK, FEL, FF, FPE, FPR, FST, FZS, FDU;
}

```

## ■ třída Student

```
public class Student extends OsobaNaZCU {  
    private Fakulty fakulta;  
  
    public Student(String jmeno, String osobniCislo, Fakulty fakulta) {  
        super(jmeno, osobniCislo);  
        this.fakulta = fakulta;  
    }  
  
    public Fakulty getFakulta() {  
        return fakulta;  
    }  
  
    @Override // z OsobaNaZCU  
    public String toString() {  
        String zakladniInfo = super.toString();  
        return zakladniInfo + ", fakulta=" + fakulta;  
    }  
}
```

- aniž je to uvedeno, dědí implementaci rozhraní IOslovitelny a IEvidovatelny
- přidává atribut fakulta výčtového typu
- protože ve třídě OsobaNaZCU není bezparametrický konstruktor, musí být v konstruktoru Student pomocí super() volán konstruktor OsobaNaZCU(String jmeno, String osobniCislo)
- nevyhovující metoda toString() je překryta, ale v jejím těle je pomocí super.toString() vyvolána metoda předka
  - ◆ ve skutečnosti je tedy metodě toString() rozšířena funkčnost

## ■ možné testy nad třídou Student

```
@Test  
public void testToString() {  
    OsobaNaZCU os = new Student("Pilná", "A10B9999P", Fakulty.FAV);  
    assertEquals("Chybná informace: ", "Student: Pilná, ID=A10B9999P,  
                fakulta=FAV", os.toString());  
}  
  
@Test  
public void testIOslovitelny() {  
    IOslovitelny os = new Student("Pilná", "A10B9999P", Fakulty.FAV);  
    assertEquals("Chybné jméno: ", "Pilná", os.getJmeno());  
}  
  
@Test  
public void testIEvidovatelny() {  
    IEvidovatelny os = new Student("Pilná", "A10B9999P", Fakulty.FAV);  
    assertEquals("Chybné ID: ", "A10B9999P", os.getID());  
}
```

```

@Test
public void testFakulty() {
    Student os = new Student("Pilná", "A10B9999P", Fakulty.FAV);
    assertEquals("Chybná fakulta: ", "FAV", os.getFakulta().toString());
}

```

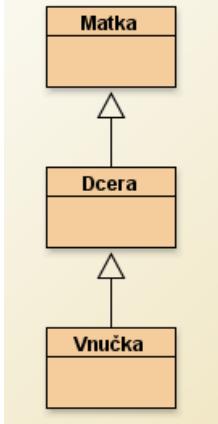
- je vidět, že instance třídy `Student` se může vydávat za instanci

- ◆ `OsobaNaZCU`
- ◆ `IOslovitelny`
- ◆ `IEvidovatelny`
- ◆ `Student`

### 5.4.3. Konstrukce třídy a spolupráce s nadtírdou

- aby bylo možno zabezpečit funkci všech (i soukromých) vazeb, obsahuje každý objekt dceřiné třídy jako svoji součást podobjekt své rodičovské třídy, který s sebou přináší požadovanou implementaci
- objekt dceřiné třídy se nemůže začít budovat dřív, než bude zcela vybudován jeho rodičovský podobjekt
- rodičovský podobjekt je vytvořen jako „soukromý atribut“ nazvaný `super` a pomocí tohoto identifikátoru mu lze zasílat zprávy
- je to něco podobného jako `this`

- umělý příklad pro ukázku vlastností (převzato od R.Pecinovského):



- v případě vytvoření instance `Vnučka` se v paměti vytvoří



- třída `Matka`

```

public class Matka
{
    private static int m_počet = 0;
    private int m_pořadí = ++m_počet;
    private String název = "Matka_" + m_počet;

    public static void zpráva( String text )
    {
        System.out.println( text + " (" + m_pořadí + " )");
    }

    public Matka()
    {
        System.out.println( "\nVytvářím " + m_pořadí +
            ". instanci třídy Matka " );
    }

    public Matka( String s )
    {
        System.out.println( "\nVytvářím " + m_pořadí +
            ". instanci třídy Matka " + s );
    }

    public void matka()
    {
        System.out.println( "\nMetoda matka() instance " + this +
            "\n Název podobjektu: " + název );
        soukromá();
        veřejná();
        System.out.println( název + ".matka - konec" );
    }

    public void veřejná()
    {
        System.out.println(" Třída Matka, metoda veřejná(): " +
            "\n      Podobjekt: " + název +
            "\n      Instance: " + this );
    }

    public void zprávy()
    {
        zpráva( "\nMatka - moje zpráva" );
        Matka .zpráva( "- Zpráva matky" );
        Dcera .zpráva( "- Zpráva dcery" );
        Vnučka.zpráva( "- Zpráva vnučky" );
    }

    private void soukromá()
    {
        System.out.println(" Třída Matka, metoda soukromá(): " +
            "\n      Podobjekt: " + název +
            "\n      Instance: " + this );
    }
}

```

```
    }  
}
```

## ■ třída Dcera

```
public class Dcera extends Matka  
{  
    private static int d_počet = 0;  
    private int d_pořadí = ++d_počet;  
    private String název = "Dcera_" + d_počet;  
  
    public static void zpráva( String text )  
    {  
        System.out.println( text + " (D)" );  
    }  
  
    public Dcera()  
    {  
        //      this( "" );  
        System.out.println( "Vytvářím " + d_pořadí +  
                           ". instanci třídy Dcera" );  
    }  
  
    public Dcera( String s )  
    {  
        //      super( "- pro dceru " + s );  
        System.out.println( "Vytvářím " + d_pořadí +  
                           ". instanci třídy Dcera " + s );  
    }  
  
    //  @Override  
    public void veřejná()  
    {  
        System.out.println(" Třída Dcera, metoda veřejná(): " +  
                            "\n      Podobjekt: " + název +  
                            "\n      Instance: " + this );  
    }  
  
    //  @Override  
    public void zprávy()  
    {  
        zpráva( "\nDcera - moje zpráva" );  
        Matka .zpráva( "- Zpráva matky" );  
        Dcera .zpráva( "- Zpráva dcery" );  
        Vnučka.zpráva( "- Zpráva vnučky" );  
    }  
  
    public void dcera()  
    {  
        System.out.println( "\nMetoda dcera() instance " + this +  
                            "\n      Název podobjektu: " + název );  
        soukromá();  
    }
```

```

        veřejná();
        System.out.println( název + ".dcera - konec");
    }

public void rodiče()
{
    System.out.println("Dcera - moje verze metody veřejná():");
    veřejná();
//    System.out.println("Dcera - rodičovská verze metody veřejná():");
//    super.veřejná();
}

private void soukromá()
{
    System.out.println(" Třída Dcera, metoda soukromá(): " +
                       "\n      Podobjekt: " + název +
                       "\n      Instance: " + this );
}
}

```

## ■ třída Vnučka

```

public class Vnučka extends Dcera
{
    private static int v_počet = 0;
    private int v_pořadí = ++v_počet;
    private String název = "Vnučka_" + v_počet;

//    public static void zpráva( String text ) - chybí

    public Vnučka()
    {
        System.out.println( "Vytvářím " + v_pořadí +
                           ". instanci třídy Vnučka " );
    }

    public Vnučka( String s )
    {
//        super( "- pro vnučku " + s );
        System.out.println( "Vytvářím " + v_pořadí +
                           ". instanci třídy Vnučka " + s );
    }

//    @Override
    public void veřejná()
    {
        System.out.println(" Třída Vnučka, metoda veřejná(): " +
                           "\n      Podobjekt: " + název +
                           "\n      Instance: " + this );
    }
}

```

```

//    @Override
public void zprávy()
{
    zpráva( "\nVnučka - moje zpráva" );
    Matka .zpráva( "- Zpráva matky" );
    Dcera .zpráva( "- Zpráva dcery" );
    Vnučka.zpráva( "- Zpráva vnučky" );
}

public void vnučka()
{
    System.out.println( "\nMetoda vnučka() instance " + this +
                        "\n Název podobjektu: " + název );
    soukromá();
    veřejná();
    System.out.println( název + ".vnučka - konec" );
}

//    @Override
public void rodiče()
{
    System.out.println("Vnučka - moje verze metody veřejná():");
    veřejná();
//    System.out.println("Vnučka - rodičovská verze metody veřejná():");
//    super.veřejná();
//    System.out.println("\nVnučka - rodičovská verze metody ►
rodiče():\n");
//    super.rodiče();
}

private void soukromá()
{
    System.out.println(" Třída Vnučka, metoda soukromá(): " +
                       "\n      Podobjekt: " + název +
                       "\n      Instance: " + this );
}
}

```

- při vytváření objektů příkazy – volá se implicitně bezparametrický konstruktor předka:

```

Matka matka = new Matka();
Dcera dcera = new Dcera();
Vnučka vnučka = new Vnučka();

```

se vypíše:

Vytvářím 1. instanci třídy Matka

Vytvářím 2. instanci třídy Matka

Vytvářím 1. instanci třídy Dcera

Vytvářím 3. instanci třídy Matka

```
Vytvářím 2. instanci třídy Dcera  
Vytvářím 1. instanci třídy Vnučka
```

což potvrzuje, že každý podobjekt má v sobě všechny objekty předků

### 5.4.3.1. Explicitní volání konstruktoru předka

- v předchozím případě byl při vytváření instance předka volán bezparametrický konstruktor
- i při dědění je možné využít přetížené konstruktory volané pomocí `this()`
- pro volání konstruktoru předka využijeme `super()`, které musí být v konstruktoru potomka uvedeno jako první
- po úpravě konstruktorů třídy `Dcera`

```
public Dcera()  
{  
    this( "" );  
    System.out.println( "Vytvářím " + d_pořadí +  
        ". instanci třídy Dcera " );  
}  
  
public Dcera( String s )  
{  
    super( "- pro dceru " + s );  
    System.out.println( "Vytvářím " + d_pořadí +  
        ". instanci třídy Dcera " + s );  
}
```

- po příkazu:

```
Dcera dcera = new Dcera();
```

se vypíše:

```
Vytvářím 1. instanci třídy Matka - pro dceru  
Vytvářím 1. instanci třídy Dcera
```

protože konstruktor `Dcera()` zavolal svým prvním příkazem pomocí `this("")` svůj přetížený konstruktor `Dcera( String s )`

- ten pomocí `super( "- pro dceru " + s )`; vyvolal konstruktor `Matka( String s )`

- celý příklad mj. ukazuje následující:

- pokud vytvoříme v předkovi konstruktor s parametry (a tím se nevytvoří implicitní bezparametrický konstruktor), musíme v konstruktoru potomka použít `super(parametry)`
- označíme-li v předkovi jeho konstruktory jako `private`, např. z důvodů použití statické tovární metody, znemožníme dědění této třídy

## 5.4.3.2. Dosažitelnost this

- Pozor: skrytý atribut `this` není dostupný před úplným vytvořením celé instance se všemi předky
- upravíme-li konstruktor z předchozího příkladu

```
public Dcera()
{
    //      this( "" );
    this( this.toString() );
}
```

vypíše překladač hlášení:

```
cannot reference this before supertype constructor has been called
```

- to znamená, že `this` ve vytvářené instanci lze plnohodnotně využívat, až po skončení práce konstruktoru třídy předka (po návratu ze `super()`)

## 5.4.3.3. Postup budování instance

- vytváření instance probíhá v následujících krocích
  1. konstruktory si postupně předávají zodpovědnost pomocí `this()`
  2. poslední konstruktor v řadě zavolá jako svůj první příkaz `super()` – buď explicitně (s parametry) nebo implicitně (pokud existuje bezparametrický)
  3. po návratu ze `super()` lze používat `this.` pro přístup k atributům a metodám
  4. postupně se provádí zdrojový kód konstruktoru a inicializují (i pomocí volání metod) jednotlivé atributy – dosud neinicializované mají nulové hodnoty

## 5.4.3.4. Využití statické tovární metody

- potřebujeme-li provést nějakou akci ještě před zavolením rodičovského konstruktoru
- statická tovární metoda je zcela běžná metoda, a proto na ni nejsou kladena omezení platná pro konstruktory
- její výhody (viz dříve) lze využít i pro konstrukci instancí zděděných tříd

## 5.4.4. Dědičnost a metody

- pokud není v potomkovi metoda překryta, je volána metoda bezprostředního předka
- implementaci rozhraní dědí potomek nezávisle na tom, jestli požadované metody zdědí již implementované, anebo se bude muset postarat o jejich implementaci sám

### 5.4.4.1. Statické metody

- třída Vnučka nemá metodu `public static void zpráva(String text)`

- metoda zprávy ve třídě Matka volá statické metody zpráva ()

```
public void zprávy()
{
    zpráva( "\nMatka - moje zpráva" );
    Matka .zpráva( "- Zpráva matky" );
    Dcera .zpráva( "- Zpráva dcery" );
    Vnučka.zpráva( "- Zpráva vnučky" );
}
```

při volání:

```
matka.zprávy();
```

se vypíše:

```
Matka - moje zpráva (M)
- Zpráva matky (M)
- Zpráva dcery (D)
- Zpráva vnučky (D)
```

je vidět, že třída Vnučka využívá metodu zpráva () od rodičovské Dcery – písmeno (D)

- nedoporučuje se v příbuzných třídách definovat stejně pojmenované statické metody – svádí to k různým falešným očekáváním

#### 5.4.4.2. Metody instancí

- ve zděděné třídě mohou být tři typy instančních metod

1. nově definované metody, jejichž signatura (tj. jméno a parametry) se v předcích nenachází
  - žádný problém – je to typ metod, se kterými jsme se dosud setkávali
2. zděděné metody od rodiče, které potomek používá, tak jak jsou
  - zjistí-li instance, že sama neimplementuje volanou metodu, pátrá u předků, a nalezne-li ji, vyvolá ji jako svoji metodu
  - to znamená, že tyto metody se volají jako předchozí metody
    - ◆ voláme-li tuto metodu ve vlastní třídě, můžeme využít i this, chceme-li
    - ◆ z vnějšku se metoda volá jako vnučka.dcera();
  - příklady již známých metod jsou getClass() a equals()
3. překryté zděděné metody, např. toString()

#### 5.4.4.3. Překryté zděděné metody

- rodič může zakázat překrývání svých metod uvedením modifikátoru final (ty mohou být současně public)
  - final též zajistí rychlejší vyvolávání metod

- pokud to neudělá, je taková metoda označovaná jako **virtuální** – potomek ji může překrýt
  - samozřejmě za předpokladu, že je pro něj metoda viditelná – v předkovi není `private`
    - ◆ když potomek definuje metodu se stejnou signaturou, jako má některá ze soukromých metod rodičovské třídy, je tato metoda považována za zcela novou metodu
- překrytá metoda musí mít stejnou signaturu – jako při implementaci rozhraní
  - stejně tak se před metodu uvádí anotace `@Override`, aby překladač dokázal zkontrolovat, že jde o překrytí, nikoliv o přetížení (stejné jméno, jiné parametry)
- kdykoliv v budoucnu někdo pošle objektu zprávu, jejíž zpracování má na starosti virtuální metoda, bude vždy zavolána metoda osloveného objektu, a to nezávisle na tom, za čí instanci se objekt v danou chvíli vydává (např. dočasně přetypovaná na rozhraní)
  - o vyvolané metodě rozhoduje virtuální stroj až za běhu aplikace
  - rozhoduje se podle tabulky virtuálních metod, která je skryta součástí třídy
- jakmile potomek překryje rodičovskou verzi metody, stane se překrytá verze pro okolí nedostupná
  - jedinou výjimkou je volání ze samotného potomka, který překrytou metodu předka dokáže vyvolat pomocí `super`.
    - ◆ to se používá poměrně často, kdy rodičovská metoda volaná pomocí `super`. něco předpřipraví a metoda potomka to dokončí / upřesní
    - ◆ pomocí `super`. můžeme volat metody pouze z přímého předka – není možné `super.super`.
- příklad pro třídu `Vnučka` – metody `veřejná()` a `rodiče()` jsou přetížené

```

@Override
public void veřejná()
{
    System.out.println(" Třída Vnučka, metoda veřejná(): " +
                       "\n      Podobjekt: " + název +
                       "\n      Instance: " + this );
}

@Override
public void rodiče()
{
    System.out.println("Vnučka - moje verze metody veřejná():");
    veřejná();
}

```

při volání

```
vnučka.rodiče();
```

vypíše

```
Vnučka - moje verze metody veřejná():
Třída Vnučka, metoda veřejná():
```

Podobjekt: Vnučka\_1  
Instance: Vnučka@ae3364

- upravíme-li metodu `rodiče()`, že budeme volat metodu veřejná z předka pomocí `super.veřejná()`

```
@Override  
public void rodiče()  
{  
    System.out.println("Vnučka - rodičovská verze metody veřejná()");  
    super.veřejná();  
}
```

při volání

```
vnučka.rodiče();
```

vypíše – ve výpisu je vidět, že:

- se skutečně zavolala metoda `Dcera` – Podobjekt: `Dcera_2`
- i když je volána metoda předka, jedná se o instanci `Vnučka`

Vnučka - rodičovská verze metody veřejná():  
Třída `Dcera`, metoda veřejná():  
Podobjekt: `Dcera_2`  
Instance: `Vnučka@1402d5a`

- upravíme-li metodu `rodiče()`, že budeme volat metodu `veřejná()` z předka pomocí `super.veřejná()`

```
@Override  
public void rodiče()  
{  
    System.out.println("\nVnučka - rodičovská verze metody rodiče():\n");  
    super.rodiče();  
}
```

při volání

```
vnučka.rodiče();
```

vypíše – ve výpisu je vidět, že i když je volána metoda předka, jedná se o instanci `Vnučka`, takže metoda `veřejná()` bude z instance `Vnučka`

Vnučka - rodičovská verze metody rodiče():  
  
Dcera - moje verze metody veřejná():  
Třída `Vnučka`, metoda veřejná():  
Podobjekt: `Vnučka_1`  
Instance: `Vnučka@6754d6`

## 5.4.5. Výhoda dědění

- v dříve uvedeném případě byl čtverec vytvořen skládáním z obdélníka
  - jednalo se o dva různé objekty, které nebylo možné vzájemně zaměnit – nemohly porušit Liskové princip
- pokud si řekneme, že se v aplikaci nebudeme nikdy pokoušet zaměňovat čtverec za obdélník, můžeme si děděním implementace výrazně ulehčit práci
- ve třídě `ZdedenyCtverec` je vhodné / nutné překrýt pouze jednu metodu z původního `Obdelnik` a to `setRozmer(int sirka, int vyska)`
- pak je nutné napsat konstruktory – to je ale třeba vždy
  - třída potomka by měla definovat konstruktor maximálně využívající co nejobecnější verzi rodičovského konstruktoru
    - ◆ a samozřejmě libovolný počet „jednodušších“ konstruktorů
  - pokud to neudělá, zkomplikuje přípravu případných vnoučat
- konstrukce nové třídy je výrazně kratší než v případě skládání

```
*****  
 * Instance třídy {@code ZdedenyCtverec} představují čtverce  
 *  
 * @author Pavel Herout  
 * @version 2.00.000  
 */  
public final class ZdedenyCtverec extends Obdelnik {  
  
//== KONSTANTNÍ ATRIBUTY TŘÍDY ======  
    /** Implicitní rozměr */  
    private static final int IMPLICITNI_ROZMER = 100;  
  
    /** Počáteční barva nakreslené instance v případě,  
     * kdy uživatel žádnou požadovanou barvu nezadá  
     */  
    public static final Barva IMPLICITNI_BARVA = Barva.ZLUTA;  
  
//#####  
//== KONSTRUKTORY A TOVÁRNÍ METODY ======  
  
*****  
    * Vytvoří novou instanci se polohou [0, 0] a rozměrem 100  
    * a implicitní barvou.  
    */  
    public ZdedenyCtverec() {  
        this(0, 0, IMPLICITNI_ROZMER);  
    }  
  
*****  
    * Vytvoří novou instanci se zadanou polohou a rozměrem
```

```

* a implicitní barvou.
*
* @param x      x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna
* @param y      y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna
* @param strana velikost strany vytvářené instance, strana >= 0
*/
public ZdedenyCtverec(int x, int y, int strana) {
    this( x, y, strana, IMPLICITNI_BARVA );
}

/******************
 * Vytvoří novou instanci se zadanými rozměrem, polohou a barvou.
 *
* @param x      x-ová souřadnice instance, x>=0, x=0 má levý okraj plátna
* @param y      y-ová souřadnice instance, y>=0, y=0 má horní okraj plátna
* @param strana velikost strany vytvářené instance, strana >= 0
* @param barva Barva vytvářené instance
*/
public ZdedenyCtverec(int x, int y, int strana, Barva barva) {
    super(x, y, strana, strana, barva);
}

//== PŘÍSTUPOVÉ METODY ATRIBUTU INSTANCÍ =====
/******************
 * Nastaví nové rozměry instance - oba parametry by měly být stejné.
 * Při rozdílných hodnotách se použije menší z nich.
 * Metoda musí být překryta, aby byl zajištěn kontrakt, že čtverec má
 * stejně dlouhé strany
 *
* @param sirka   Nově nastavovaná šířka; šířka>=0
* @param vyska   Nově nastavovaná výška; výška>=0
*/
@Override
public void setRozmer(int sirka, int vyska) {
    int strana = Math.min(sirka, vyska);
    super.setRozmer(strana, strana);
    super.setRozmer(Math.min(sirka, vyska));
}

// ****
// * Chybná verze - zacyklené volání
//
// * @param šířka   Nově nastavovaná šířka; šířka>=0
// * @param výška   Nově nastavovaná výška; výška>=0
// */
@Override
public void setRozmer(int sirka, int vyska) {
    super.setRozmer(Math.min(sirka, vyska));
}
}

```

- takto připravený čtverec má stejnou (lepší – implementuje všechny metody) funkčnost, jako čtverec vytvořený skládáním

### 5.4.5.1. Možné problémy při překrývání metod

- při konstrukci překryté metody `setRozmer(int sirka, int vyska)` lze udělat záludnou chybu
- ve třídě `Obdelnik` již existuje metoda `setRozmer(int rozmer)`, kterou by bylo zdánlivě možné ve třídě `ZdedenyCtverec` využít

```
@Override
public void setRozmer(int sirka, int vyska) {
    super.setRozmer(Math.min(sirka, vyska));
}
```

- překlad proběhne v pořádku, ale po pokusu o změnu rozměru je po chvíli vyvolána výjimka `java.lang.StackOverflowError`
- vysvětlení je takové, že v `Obdelnik` je

```
@Override
public void setRozmer(int rozmer)
{
    setRozmer( rozmer, rozmer );
}

@Override
public void setRozmer(int sirka, int vyska)
{
    if( (sirka < 0) || (vyska < 0) ) {
        throw new IllegalArgumentException(
            "Rozmery musí být nezáporné: sirka=" +
            sirka + ", vyska=" + vyska);
    }
    this.sirka = Math.max(1, sirka);
    this.vyska = Math.max(1, vyska);
    SP.prekresli();
}
```

- takže metoda `setRozmer(int rozmer)` volá metodu `setRozmer(int sirka, int vyska)`
- při volání se ale použije překrytá verze `setRozmer(int sirka, int vyska)` ze `ZdedenyCtverec`
  - protože virtuální stroj se vždy snaží volat metodu té instance, kterou ve skutečnosti je
- ta opět volá jednoparametrickou metodu `setRozmer(int rozmer)` z `Obdelnik` a tím dojde k zacyklení

### 5.4.6. Konečné třídy

- nechceme-li, aby bylo možné třídu zdědit, máme dvě možnosti

- privátní konstruktory – používáme u knihovních tříd
- označení třídy jako konečné pomocí `final`

```
public final class ZdedenyCtverec extends Obdelnik {
```

♦ to je v Java Core API použito např. pro `Class` nebo `String`

- důvody jsou bezpečnostní a rychlostní

## 5.4.7. Závěrečné poznámky o nevhodnosti dědičnosti implementace

- dědičnost porušuje zapouzdření a skrývání implementace

- nutí potomka, aby znal implementaci v předkovi – viz nekonečná smyčka při návrhu metody `setRozmer(int, int)`

- dědičnost zvyšuje vzájemnou provázanost tříd

- potřebuji-li modifikovat rodiče (najdu v něm chybu nebo modifikaci vynutila změna zadání), musím zkонтrolovat všechny potomky, že tato změna jejich chování nežádoucím způsobem neovlivnila
  - ♦ potomek je závislý nejenom na rozhraní rodiče, ale i na jeho implementaci

- dědičnost (raději) nepoužijeme v případě, že:

- umíme najít jiný rozumný způsob realizace potřebné funkčnosti, např. skládání
- potenciální potomek není speciálním případem předka, např. kvádr versus obdélník
- potomek je až příliš speciálním případem předka, např. čtverec versus obdélník

## 5.5. Abstraktní třída

- něco mezi rozhraním (`interface`) a třídou (`class`)

- má hlavičku `abstract class AJménoTřídy`

- vhodná konvence pro pojmenování je, že název třídy začíná písmenem `A`
- v UML diagramu tříd se jméno abstraktní třídy píše kurzivou *AJménoTřídy* a/nebo se použije stereotyp `<<abstract>>`

- na rozdíl od rozhraní může mít instanční atributy a může mít implementaci některých metod

- ty se neoznačují `default`

- zbývající metody mohou mít jen hlavičku, jako je to u rozhraní

- mezi jejich modifikátory však musí být uvedeno klíčové slovo `abstract`
- abstraktní metody nesmějí být soukromé (`private`), potomek by na ně neviděl a nemohl by je implementovat

- konkrétní metody v abstraktní třídě mohou ve svých definicích používat všechny deklarované metody včetně abstraktních, přestože tyto nejsou v době jejich definice ještě implementovány
- od abstraktní třídy nelze vytvořit instanci – musí se zdědit a doimplementovat
- výhoda – je to hodně předpřipravený polotovar
- za instance abstraktní třídy se vydávají instance jejích potomků
- příklad dopravních značek

- ADopravniZnacka v konstruktoru vytváří sloupek a pomocí volání abstraktní metody i ceduli
  - ◆ cedule je typu IKresleny, protože toto rozhraní implementují všechny základní tvary

```
public abstract class ADopravniZnacka implements IKresleny {

    //== KONSTANTNÍ ATRIBUTY TŘÍDY =====
    /** Barva sloupku */
    public static final Barva BARVA_SLOUPKU = Barva.SEDA;

    /** Aktivní plátno, které dohlíží na správné vykreslení instance. */
    private static final SpravcePlatna SP = SpravcePlatna.getInstance();

    //== KONSTANTNÍ ATRIBUTY INSTANCI =====
    private final Obdelnik sloupek;

    //== PROMĚNNÉ ATRIBUTY INSTANCI =====
    private IKresleny cedule = null;

    //== KONSTRUKTORY A TOVÁRNÍ METODY =====
    /**
     * vytvoří sloupek pod dopravní značkou a pak příslušnou ceduli
     * obě části zaregistrouje u správce plátna
     * @param x x-souřadnice
     * @param y y-souřadnice
     */
    public ADopravniZnacka(int x, int y) {
        sloupek = new Obdelnik(x + 40, y + 100, 20, 100, BARVA_SLOUPKU);
        SP.pridej(sloupek);
        cedule = vytvorCeduli(x, y);
        SP.pridej(cedule);
    }

    //== ABSTRAKTNÍ METODY =====
    /**
     * Vytvoří ceduli správného tvaru na požadovaném místě
     * @param x x-souřadnice
     * @param y y-souřadnice
     * @returns objekt cedule
     */
    protected abstract IKresleny vytvorCeduli(int x, int y);

    //== PŘEKRYTÉ METODY IMPLEMENTOVANÝCH ROZHRANÍ =====
}
```

```

 ****
 * Za pomoci dodaného kreslítka vykreslí obraz své instance
 * na animační plátno.
 *
 * @param kreslítko Kreslítko, kterým se instance nakreslí na plátno. ►
 */

@Override
public void nakresli(Kreslitko kreslitko) {
    sloupek.nakresli(kreslitko);
    cedule.nakresli(kreslitko);
}
}

```

- ZnackaZakazova v konstruktoru pouze volá konstruktor předka

◆ implementuje vytvorCeduli() – vytváří kruh

```

public class ZnackaZakazova extends ADopravniZnacka {
    public ZnackaZakazova(int x, int y) {
        super(x, y);
    }

    protected IKresleny vytvorCeduli(int x, int y) {
        return new Elipsa(x, y, 100, 100, Barva.CERVENA);
    }
}

```

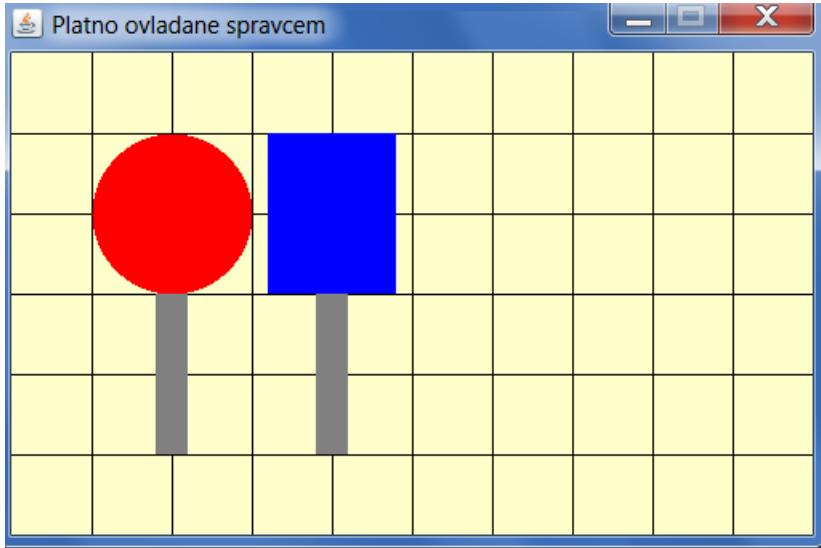
- ZnačkaInformativní – podobná předchozí, vytváří obdélník

```

public class ZnackaInformativni extends ADopravniZnacka {
    public ZnackaInformativni(int x, int y) {
        super(x, y);
    }

    protected IKresleny vytvorCeduli(int x, int y) {
        return new Obdelnik(x + 10, y, 80, 100, Barva.MODRA);
    }
}

```

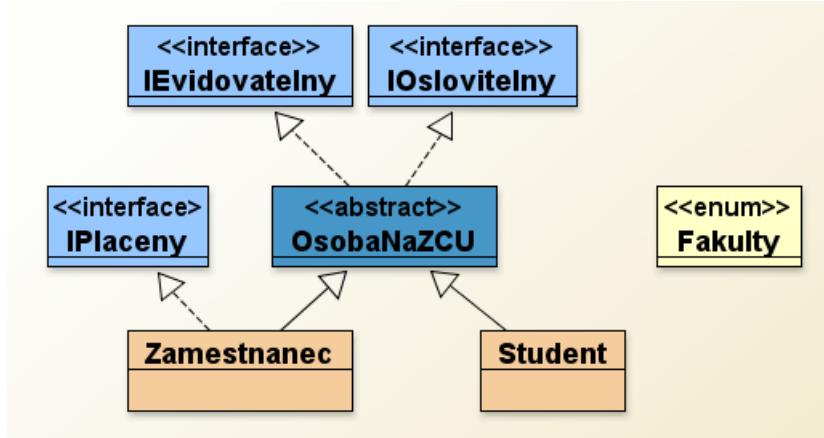


## 5.5.1. Speciální případ abstraktní třídy

- občas se setkáme se situací, kdy třída může mít všechny své metody implementovány, ale z logiky věci nemá smysl vytvářet její instance
  - např. OsobaNaZCU (viz dříve) nebude mít žádné instance, protože smysl mají až instance jejích potomků Zamestnanec nebo Student
  - pak lze snadno zajistit, aby bylo nutné třídu OsobaNaZCU zdědit
    - ◆ pouze v hlavičce třídy (nikoliv u jakékoli metody) se uvede klíčové slovo abstract

```
abstract public class OsobaNaZCU implements IOslovitelny, IEvidovatelny {
```

- ◆ vše ostatní zůstane stejné jako dříve



- toto je často používaný způsob a setkáme se s ním i v Java Core API, např.: java.lang.ClassLoader

## 5.5.2. Konstrukce abstraktní třídy z potomků

- můžeme se setkat se situací, kdy máme několik tříd se společnými vlastnostmi a začínáme postrádat jejich společného předka
  - důvodem je např. opakující se kód v těle těchto tříd

- např. třídy `Obdelnik`, `Elipsa` a `Trojuhelnik` mají mnoho společného, každý však musí mít jinou metodu `nakresli()`
- v této situaci s výhodou využijeme abstraktní třídu
  - definuje společného předka skupiny potomků
  - některé metody, které by měl tento předek „mít“, v něm nedokážeme implementovat, což vyřešíme tím, že je označíme za abstraktní
- doporučený postup, jak vytvořit společného rodiče
  1. projdeme všechny potenciální potomky vytvářeného rodiče a zjistíme, které metody mají společné – to jsou potenciální metody budoucího rodiče
    - tyto metody nemusí mít stejný kód, musí mít stejný účel (kontrakt)
  2. metody, které mají stejný či velmi podobný kód implementujeme v rodiči jako jejich společnou implementaci
  3. metody, jejichž implementace se vzájemně liší, ale mají stejný kontrakt, deklarujeme v rodiči jako abstraktní
  4. zjistíme, jaké atributy implementované metody potřebují, a deklarujeme je jako atributy rodiče
  5. definice atributů a metod, které jsme provedli v rodiči, v potomcích odstraníme

# Kapitola 6. Arrays, řazení, kolekce a genericita

## 6.1. Podpora práce s poli – třída Arrays

- pro ukládání více objektů stejného typu do paměti se používají datové struktury nazývané obecně **kontejnery**
  - kontejner má široký význam – označuje jakýkoliv objekt, který může uschovávat jiné objekty
    - ◆ **statické kontejnery** – jejich velikost je určena při jejich vzniku a je dále neměnná
      - návrhový vzor Přepravka
      - pole, podporované třídou `Arrays`
        - výhody – rychlosť, primitivní datové prvky i objekty
        - nevýhody – pevná velikost, malá podpora přídavnými funkcemi, menší úroveň bezpečnosti (synchronizace apod.)
    - ◆ **dynamické kontejnery** – jejich velikost je přizpůsobena okamžitým potřebám
      - kolekce = třídy z balíku *Collection Framework* – `java.util`
        - výhody a nevýhody víceméně opačné
        - používají se více než pole
        - viz dále

### 6.1.1. Možnosti třídy Arrays

- klasická pole mají jen jednu schopnost navíc – atribut `length`
- nemají žádné metody, např. pro seřazení pole
- `java.util.Arrays` – poskytuje existujícímu poli určité služby
- všechny metody jsou statické – zpracovávané pole se předává jako skutečný parametr
- např. existuje-li pole `abc`, pak jeho seřazení vyvoláme:
  1. správně `Arrays.sort(abc);`
  2. nesprávně `abc.sort();`
- Metody jsou (některé umí pracovat i s částí pole):
  - `equals()` – porovná dvě pole
  - `asList()` – převeďe pole na kolekci
  - `toString()` – převeďe pole na řetězec (velmi vhodné pro jednoduchý tisk pole)

- `fill()` – naplní část nebo celé pole konstantní hodnotou
- `sort()` – seřadí část nebo celé pole vzestupně
- `binarySearch()` – v poli seřazeném metodou `sort()` vrátí index prvku, který se shoduje se zadanou hodnotou
  - ◆ pokud v poli není prvek této hodnoty, vrací záporné číslo
  - ◆ toto číslo nemá konstantní hodnotu, ale v absolutní hodnotě představuje index do pole, kam by mohla být zadaná hodnota vložena
  - ◆ protože však 0 je platný index a -0 nelze použít, je záporná hodnota ještě zmenšena o 1

## Příklad 6.1. Použití metod třídy Arrays

```
import java.util.*;

public class ArraysPlneniSerazeniAVyhledavani {
    final static int POCET = 5;
    static int[] pole = new int[POCET];

    public static void main(String[] args) {
        Arrays.fill(pole, 3);
        System.out.println(Arrays.toString(pole));

        for (int i = 0; i < pole.length; i++) {
            pole[i] = (POCET - i) * 2;
        }
        System.out.println(Arrays.toString(pole));

        Arrays.sort(pole);
        System.out.println(Arrays.toString(pole));

        int k = Arrays.binarySearch(pole, 6);
        if (k >= 0)
            System.out.println("[" + k + "]=" + pole[k]);
    }
}
```

vypíše

```
[3, 3, 3, 3, 3]
[10, 8, 6, 4, 2]
[2, 4, 6, 8, 10]
[2]=6
```

## 6.2. Řazení objektů

### Poznámka

Tyto informace jsou mj. nutné pro pochopení jednoho typu (*Tree*) implementace kolekcí. Dále proto, že operace řazení se v kolekcích používá poměrně často.

- pro pole primitivních datových prvků jsou `sort()` a `binarySearch()` jednoduše použitelné
- totéž platí pro objekty knihovných tříd (`String`, `Integer` apod.), které mají jen jednu hodnotu (zjednodušeně řečeno)
- obsahuje-li pole obecné objekty, nemůže být bez upřesnění známo, jakým způsobem se budou objekty navzájem porovnávat mezi sebou
- dvě možnosti:
  1. přirozené řazení (*natural ordering*)
  2. absolutní řazení (*total ordering*)

## Výstraha

Všechny zde uváděné informace platí beze zbytku i pro kolekce!

### 6.2.1. Přirozené řazení (*natural ordering*)

- metody `sort()` a `binarySearch()` voláme stejně, jako v případě polí s primitivními datovými typy
- způsob porovnávání objektů musí být popsán ve třídě těchto objektů
  - nutno implementovat rozhraní `java.lang.Comparable<Typ>`, které má pouze jednu metodu `int compareTo(Typ t)`
    - ◆ `compareTo()` vrací `int` s hodnotou 0, pokud jsou objekty stejné, zápornou, pokud je parametr `t` menší, a kladnou, pokud je parametr `t` větší než objekt
    - ◆ metoda `compareTo()` musí být `public`
- rozhraní `Comparable` (a tudíž metodu `compareTo()`) implementují všechny obalové třídy primitivních datových typů i třída `String`
  - pro přirozené seřazení těchto typů nemusíme psát `compareTo()`, stačí zavolat metodu `Arrays.sort()`
- budeme-li řadit objekty libovolných tříd, implementací `Comparable` určíme, podle čeho budeme řadit

## Příklad 6.2. Přirozené řazení

Příklad objektu Osoba, který bude mít atributy vaha a vyska. Rozhraní Comparable ale má pouze jednu metodu compareTo() – nelze ji přetížit. Musíme si vybrat jeden atribut, podle kterého bude přirozené řazení probíhat.

```
import java.util.*;  
  
class Osoba implements Comparable<Osoba> {  
    int vyska;  
    double vaha;  
    String popis;  
  
    Osoba(int vyska, double vaha, String popis) {  
        this.vyska = vyska;  
        this.vaha = vaha;  
        this.popis = popis;  
    }  
  
    public int compareTo(Osoba os) {  
        int osVyska = os.vyska;  
        if (this.vyska > osVyska)  
            return +1;  
        else if (this.vyska == osVyska)  
            return 0;  
        else  
            return -1;  
    }  
  
    public String toString() {  
        return "vy = " + vyska +  
               ", va = " + vaha +  
               ", " + popis;  
    }  
}  
  
public class ArraysPrirozenaRazeninaObecnýchObjektu {  
    public static void main(String[] args) {  
        Osoba[] poleOsob = new Osoba[4];  
        poleOsob[0] = new Osoba(186, 82.5, "muz");  
        poleOsob[1] = new Osoba(172, 63.0, "zena");  
        poleOsob[2] = new Osoba(105, 26.1, "dite");  
        poleOsob[3] = new Osoba(116, 80.5, "obezní trpaslík");  
  
        Arrays.sort(poleOsob);  
  
        for (int i = 0; i < poleOsob.length; i++)  
            System.out.println("[ " + i + " ] " + poleOsob[i]);  
    }  
}
```

Vypíše:

```
[0] vy = 105, va = 26.1, dite
[1] vy = 116, va = 80.5, obezni trpaslik
[2] vy = 172, va = 63.0, zena
[3] vy = 186, va = 82.5, muz
```

- jedna z nevýhod přirozeného řazení – při použití třídy `Osoba` si nemůžeme vybírat, zda budeme řadit podle výšky či podle váhy nebo popisu
  - ve třídě `Osoba` určil její programátor, že `compareTo()` bude využívat atribut `vyska` – objekty této třídy lze přirozeným řazením řadit vždy jen podle výšky
- přirozené řazení používáme nejčastěji pro pole objektů s jednou hodnotou, podle které se samozřejmě (přirozeně) řadí
  - u tříd, které rozhraní `Comparable` neimplementují, nejde přirozené řazení použít

## 6.2.2. Absolutní řazení (*total ordering*)

- používáme přetížené verze metod `sort()` ze třídy `Arrays`, které mají jako poslední parametr objekt třídy, která implementuje rozhraní `java.util.Comparator`
  - využívá se návrhový vzor **Příkaz (Command)**
    - ◆ tento NV zabalí metodu do objektu, což umožňuje dynamickou výměnu používaných metod za běhu programu
    - ◆ objekt je často typován na rozhraní, které má (nejčastěji) jednu metodu
    - ◆ reference na tento objekt se pak předává jako skutečný parametr nějaké metody, pro kterou metoda v Příkazu provádí pomocnou službu
- při řazení máme absolutní kontrolu nad procesem řazení, bez ohledu na případné přednastavení řazených objektů vyjádřené metodou `compareTo()` z přirozeného řazení

`Comparator<Typ>` má dvě metody:

1. `boolean equals (Object obj)` – v naprosté většině případů neimplementujeme (dědí ji každá třída ze třídy `Object`)
  2. `int compare (Typ t1, Typ t2)` – platí stejná pravidla jako pro `compareTo()` z `java.lang.Comparable`, tj. vrací `int` s hodnotou 0, pokud jsou objekty stejné, zápornou, pokud je `t1` menší než `t2`, a kladnou v opačném případě
- metoda `compare()` musí být `public`

### Poznámka

Napíšeme-li metodu `compare()` nebo `compareTo()` podle zadání, bude se řadit vzestupně. Chceme-li sestupné řazení, **nikdy** to neřešíme obrácením znaménka návratové hodnoty! Lze použít `reverseOrder()` z `java.util.Collections`.

## Příklad 6.3. Ukázka absolutního řazení

V následujícím příkladě ponecháme zcela beze změny třídu `Osoba` z předchozího příkladu (včetně její metody `compareTo()`), což nám umožní použít i přirozené řazení – zde zakomentované).

Abychom mohli řadit jak podle výšky, podle váhy i podle popisu, napíšeme tři pomocné třídy `KomparatorOsobyPodleVysky`, jejichž anonymní objekty předáme jako druhý parametr metodě `sort()`.

Ve třídě `KomparatorOsobyPodlePopisu` můžeme bez problémů využít metodu `compareTo()` (z „konkurenčního“ řazení), protože ji třída `String` dává k dispozici.

V posledním řazení seřadíme s využitím `reverseOrder()` osoby podle výšky sestupně, přičemž se (vnitřně) používá metoda `compareTo()` ze třídy `Osoba`, nikoliv metoda `compare()` třídy `KomparatorOsobyPodleVysky`.

```
import java.util.*;  
  
class KomparatorOsobyPodleVysky implements Comparator<Osoba> {  
    public int compare(Osoba o1, Osoba o2) {  
        int v1 = o1.vyska;  
        int v2 = o2.vyska;  
        return v1 - v2;  
    }  
}  
  
class KomparatorOsobyPodleVahy implements Comparator<Osoba> {  
    public int compare(Osoba o1, Osoba o2) {  
        return (int) (o1.vaha - o2.vaha);  
    }  
}  
  
class KomparatorOsobyPodlePopisu implements Comparator<Osoba> {  
    public int compare(Osoba o1, Osoba o2) {  
        String s1 = o1.popis;  
        String s2 = o2.popis;  
        return s1.compareTo(s2);  
    }  
}  
  
public class OsobaAbsolutniRazení {  
    static Osoba[] poleOsob;  
  
    static void vypisOsob() {  
        for (int i = 0; i < poleOsob.length; i++)  
            System.out.println("[" + i + "] " + poleOsob[i].toString());  
    }  
  
    public static void main(String[] args) {  
        poleOsob = new Osoba[4];  
        poleOsob[0] = new Osoba(186, 82.5, "muz");  
        poleOsob[1] = new Osoba(172, 63.0, "zena");  
        poleOsob[2] = new Osoba(105, 26.1, "dite");  
        poleOsob[3] = new Osoba(116, 80.5, "obezni trpaslik");  
    }  
}
```

```

/* System.out.println("Prirozene razeni");
   Arrays.sort(poleOsob);
   vypisOsob();
*/
System.out.println("Absolutni razeni podle vahy");
Arrays.sort(poleOsob, new KomparatorOsobyPodleVahy());
vypisOsob();

System.out.println("Absolutni razeni podle popisu");
Arrays.sort(poleOsob, new KomparatorOsobyPodlePopisu());
vypisOsob();

System.out.println("Prirozene razeni podle vysky sestupne");
Arrays.sort(poleOsob, Collections.reverseOrder());
vypisOsob();
}
}

```

vypíše:

```

Absolutni razeni podle vahy
[0] vy = 105, va = 26.1, dite
[1] vy = 172, va = 63.0, zena
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 186, va = 82.5, muz
Absolutni razeni podle popisu
[0] vy = 105, va = 26.1, dite
[1] vy = 186, va = 82.5, muz
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 172, va = 63.0, zena
Prirozene razeni podle vysky sestupne
[0] vy = 186, va = 82.5, muz
[1] vy = 172, va = 63.0, zena
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 105, va = 26.1, dite

```

### 6.2.2.1. Binární vyhledávání pomocí metod absolutního řazení

- objekty tříd vzniklé implementací rozhraní `Comparator<Typ>` se po seřazení pole dají použít i pro binární vyhledávání metodou `binarySearch()`
  - `binarySearch()` je opět přetížena, takže poslední parametr jejího volání je objekt typu `Comparator`
- pokud je v poli více prvků (objektů) se stejnou hodnotou, není řečeno, který z nich je metodou `binarySearch()` nalezen, tzn. může být nalezen první z nich, ale také libovolný další

## 6.3. Kolekce a genericita – úvodní informace

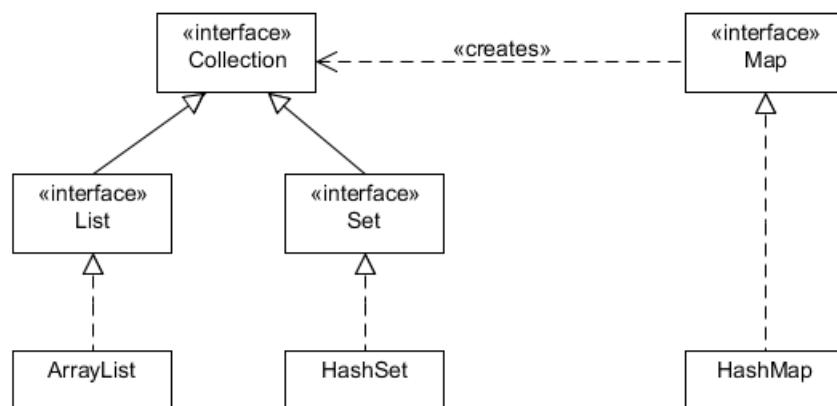
- objekty tříd z balíku `java.util` – *Java Collections Framework*
- slouží k uschovávání většího předem neznámého množství objektů libovolného typu

- „kolekce“ – dvě nejužívanější implementace – `ArrayList` a `HashSet` implementují rozhraní `java.util.Collection` (neplést si se třídou `java.util.Collections`)
- třetí významná implementace `HashMap` ale `java.util.Collection` neimplementuje

## Poznámka

Nepoužívat „staré dědictví“ `Vector`, `Hashtable` a `Dictionary`

- výhody (prakticky převažují nad nevýhodami)
  - zjednodušují program – vše je hotovo, lepší čitelnost a přehlednost
  - výrazně zrychlují vývoj programu a umožňují jeho vyladění
  - typ a počet uschovávaných objektů není omezen – uchovávají `Object`
- nevýhody
  - nelze vkládat přímo primitivní datové typy – použijeme obalovací třídy (`int` uložíme jako `Integer`)
  - většinou pomalejší než pole
- celá knihovna kolekcí velmi rozsáhlá a obsahuje ve své úplnosti (v JDK 1.8) 28 rozhraní, 8 abstraktních tříd a 31 tříd
  - na knihovnu se lze dívat ze tří pohledů:
    - ◆ **rozhraní** – abstraktní datové typy (např. `Set` = množina), které můžeme použít
    - ◆ **implementace** – konkrétní implementace rozhraní, kterou použijeme dle požadovaného účelu (např. `HashSet` nebo `TreeSet`)
    - ◆ **algoritmy** – metody poskytující služby nad kolekcemi, např. `sort()`, `reverse()`, většinou shodně pojmenované pro více kolekcí
- pokud ale nevyžadujeme žádné speciality a nezáleží nám (zpočátku) na maximální možné efektivnosti programu, potřebujeme pouze čtyři rozhraní a tři implementační třídy



### 1. List – rozhraní k seznamům – uspořádaná kolekce

#### **Seznam** – `ArrayList`

- nejvíce připomíná „klasické“ pole, ovšem s proměnnou délkou
- pro přístup k jednotlivým prvkům lze používat indexy, protože prvky jsou udržovány v určitém pořadí

## 2. Set – rozhraní k množinám – neuspořádaná kolekce

### Množina – HashSet

- obsahuje pouze unikátní prvky (nemá stejné prvky)
- pro přístup k jednotlivým prvkům nelze použít index, ale výhradně jen iterátor
- z hlediska efektivnosti implementace se pro přístup k prvkům množiny používá hešovací funkce – pro uživatele třídy HashSet naprosto skryto v implementaci (ale musí napsat metodu hashCode())

## 3. Map – rozhraní k mapám

### Mapa (asociativní pole) – HashMap

- uložení dvojcí objektů, které jsou ve vzájemném vztahu klíč-hodnota
- pomocí klíče dvojici vyhledáváme a zajímá nás nejen hodnota klíče, ale i hodnota prvku, se kterou je klíč svázán
- nejsou možné dva stejné klíče – při stejném klíči uschová naposledy vloženou hodnotu
- zjednodušený pohled – mapa je databáze o dvou sloupcích nebo dvojice ArrayListů

## Výstraha

- do JDK 1.5 byly všechny kolekce beztypové – do kolekce lze uložit cokoliv, ale při výběru musíme přetypovávat
- překlad beztypových kolekcí pod JDK 1.5 je možný, pouze se vypíše varovné hlášení:

```
Note: Some input files use unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

- od JDK 1.5 je možné kolekce typovat, což přináší větší bezpečnost kódu a větší komfort programátora (nemusí přetypovávat) – podrobnosti viz dále
- zápis dříve (TypovanaKolekce.java):

```
ArrayList arNetypovany = new ArrayList();  
arNetypovany.add(new Integer(1));  
arNetypovany.add(new Integer(2));  
arNetypovany.add("tri");  
for (Object objekt: arNetypovany) {  
    Integer i = (Integer) objekt;  
    System.out.print(i.intValue() + ", ");  
}
```

zápis od JDK 1.5:

```

ArrayList<Integer> arTypovany = new ArrayList<Integer>();
arTypovany.add(new Integer(1));
arTypovany.add(new Integer(2));
arTypovany.add("tri");      // chyba při komplaci
for (Integer cislo : arTypovany) {
    System.out.print(cislo.intValue() + ", ");
}

```

zápis od JDK 1.8:

```
ArrayList<Integer> arTypovany8 = new ArrayList<>();
```

tj. typ vkládaný do kolekce se při vytváření kolekce již neopakuje

- tento způsob bude dále výhradně používán

## Poznámka

Další vylepšení z JDK 1.5 viz později.

## Poznámka

Konstrukce `for()` pro procházení přes všechny prvky kolekce viz též [PPA1-97].

## Poznámka

Kolekce všech tří skupin lze snadno používat i pro více rozměrů, kdy např. prvky seznamu `ArrayList` mohou být buď další `ArrayList`, nebo množiny nebo mapy.

## Poznámka

Nemáme-li skutečně vážný důvod, používáme referenční proměnnou typu rozhraní, nikoliv typu implementace. Zdůvodnění viz dříve. Toto typování je nutností ve formálních parametrech našich metod, protože metody z kolekcí vracejí zásadně typ rozhraní, nikoliv typ implementace – viz později u `CeleCislo`. Další výhoda – v budoucnu je možná snadná změna implementace.

```

List<Integer> seznam = new ArrayList<>();
//      List<Integer> seznam = new LinkedList<>();
seznam.add(new Integer(1));
seznam.add(new Integer(2));
for (Integer cislo : seznam) {
    System.out.print(cislo.intValue() + ", ");
}

```

Pokud typujeme na co hierarchicky nejvyšší rozhraní, je tímto způsobem možná i změna typu kolekce. Pouze se vytvoří jiná implementace kolekce (ale vkládání a výběr prvků a průchod kolekcí zůstanou naprosto nezměněny, protože tyto služby jsou popsány kontraktem rozhraní).

```
Collection<Typ> c = new ArrayList<>();
```

nebo později:

```
Collection<Typ> c = new HashSet<>();
```

## Příklad 6.4.

Základní odlišnosti tří hlavních typů kolekcí. Do `List` lze uložit více stejných prvků, kdežto do `Set` ani do `Map` nikoliv. Obsahy všech kolekcí lze bez problémů tisknout.

```
import java.util.*;  
  
public class OdliisnostiKolekci {  
    public static void main(String[] args) {  
        List<String> seznam = new ArrayList<>();  
        seznam.add("prvni");  
        seznam.add("druhy");  
        seznam.add("prvni");  
        System.out.println("List: " + seznam);  
  
        Set<String> mnozina = new HashSet<>();  
        mnozina.add("prvni");  
        mnozina.add("druhy");  
        mnozina.add("prvni");  
        System.out.println("Set: " + mnozina);  
  
        Map<String, String> mapa = new HashMap<>();  
        mapa.put("prvni", "objekt");  
        mapa.put("druhy", "objekt");  
        mapa.put("prvni", "pivo");  
        System.out.println("Map: " + mapa);  
    }  
}
```

Vypíše:

```
List: [prvni, druhý, prvni]  
Set: [prvni, druhý]  
Map: {prvni=pivo, druhý=objekt}
```

## 6.4. Typové parametry a parametrizované typy

### Varování

Podrobně v KIV/PPA2.

- podobně jako metody mají své (formální) parametry, mohou mít parametry i objektové datové typy (rozhraní a třídy)
  - tyto datové typy se pak nazývají **parametrizované typy** nebo synonymně **generické typy**
  - jsou zavedeny od JDK 1.5
- parametry třídy (rozhraní) stanovují typy objektů, které budou dále ve třídě využívány
  - označují se jako **typové parametry**
  - třída se uvedením typového parametru specializuje jen (zejména) na práci s tímto typem

- ◆ překladač může provádět mnohem více kontrol
  - rozhoduje o přípustnosti skutečně použitého typu
- ◆ v přeloženém kódu již tyto typové informace nejsou
- jako typové parametry je možno použít pouze objektové typy (třídy nebo rozhraní) – nikoliv primitivní typy

■ typové parametry se uvádí ve špičatých závorkách za názvem třídy – např. `List<String>`

- v Java Core API je při deklaraci parametrizovaného typu je typový parametr často značen jako `<E>` („element“), např.:

```
public interface List<E>
```

- při použití je pak `E` nahrazeno skutečnou třídou nebo rozhraním – definujeme skutečný typ objektů

```
List<String> seznamRetezcu;
List<Integer> seznamCelychCisel;
List<Osoba> seznamOsob;
```

■ použití parametrizovaných typů je typické zejména v kolekcích, ale vyskytuje se i jinde

- např. rozhraní `java.lang.Comparable<Typ>`

■ můžeme vytvářet svoje parametrizované typy

- v začátcích používání OOP asi nevyužijeme, protože všechny potřebné typy jsou již hotové
- `E` je použito jako typ `<E>`, jako formální parametr metody `vloz(E prvek)` i jako návratová hodnota `E vyber()`

```
public class Zasobnik<E> {
    List<E> prvky = new ArrayList<>();

    public void vloz(E prvek) {
        prvky.add(0, prvek);
    }

    public boolean isPrazdny() {
        return (prvky.size() == 0);
    }

    public E vyber() {
        E vybirany = prvky.get(0);
        prvky.remove(0);
        return vybirany;
    }
}
```

- použití:

```
Zasobnik<Hruska> zas1;  
Zasobnik<Integer> zas2;
```

- občas potřebujeme, aby typové parametry vyhovovaly daným omezením – např. aby implementovaly nějaké rozhraní

```
public class PorovnavaciZasobnik<E extends Comparable<E>> {
```

- zde se i pro implementaci rozhraní používá klíčové slovo `extends`
- do tohoto zásobníku by šly ukládat jen třídy, které implementují rozhraní `Comparable`, např.:

```
PorovnavaciZasobnik<Hruska> zas1; // nelze  
PorovnavaciZasobnik<Integer> zas2;
```

◆ pro třídu `Hruska` hlásí překladač:

```
Bound mismatch: The type Hruska is not a valid substitute  
for the bounded parameter <E extends Comparable<E>>  
of the type PorovnavaciZasobnik<E>
```

## 6.4.1. Použití žolíků – *unbounded wildcard*

- slouží k řešení problémů způsobených omezeními dědičnosti parametrizovaných typů ve významu:
  - jakákoli třída: `<?>`
  - třída implementující rozhraní `R`, resp. potomek `R`: `<? extends R>` – viz dále
- typový parametr `<?>` se používá pouze v deklaracích tříd a metod, např.:
  - v API: `boolean removeAll(Collection<?> c)`
- nelze použít pro deklaraci: `List<?> seznam = new ArrayList<?>();`
- v našich programech zásadně nepoužívat – ztrácíme výhody typování

## Příklad 6.5.

```
public class TypovanaKolekceWildcard {  
    public static void main(String[] args) {  
        List<Object> seznam = new ArrayList<>();  
        seznam.add("jedna");  
        seznam.add(new Integer(2));  
        tisk(seznam);  
    }  
  
    public static void tisk(List<?> seznam) {  
        for (Object o : seznam) {  
            if (o instanceof String) {  
                System.out.println(o.toString());  
            }  
            if (o instanceof Integer) {  
                System.out.println(((Integer) o).intValue());  
            }  
        }  
    }  
}
```

## 6.4.2. Omezené využití žolíků – *bounded wildcard*

- stejně jako běžnou třídu lze omezit i použití žolíku
  - třída implementující rozhraní R, resp. potomek R: <? extends R>
- používá se i ve formálních parametrech metod
  - v API: boolean addAll(Collection<? extends T> c)
- nelze použít pro deklaraci:

```
List<? extends T> seznam = new ArrayList<? extends T>();
```

- jako bázový typ T lze samozřejmě použít i rozhraní
- v začátcích programování nepoužíváme

## Příklad 6.6.

```
interface Tisknutelny {  
    public void tiskni();  
}  
  
class A implements Tisknutelny {  
    public void tiskni() {  
        System.out.println("A");  
    }  
}  
  
class B extends A {  
    @Override  
    public void tiskni() {  
        System.out.println("B potomek A");  
    }  
}  
  
public class TypovanaKolekceOmezeniZolika {  
    public static void main(String[] args) {  
        List<A> seznam = new ArrayList<>();  
        seznam.add(new A());  
        seznam.add(new B());  
        tisk(seznam);  
    }  
  
    public static void tisk(List<? extends Tisknutelny> seznam) {  
        for (Tisknutelny tisknutelny : seznam) {  
            tisknutelny.tiskni();  
        }  
    }  
}
```

vypíše:

```
A  
B potomek A
```

## 6.5. Rozhraní Collection

■ základ seznamů a množin, definuje množství metod

- formální parametr `E` má význam „typově libovolný **element** kolekce“
  - ◆ jak je to v knihovně kolekcí implementačně zařízeno, nás nemusí zajímat

1. Metody pro plnění kolekce:

- `boolean add(E e)` – vložení jednoho prvku
- `boolean addAll(Collection <? extends E> c)` – vložení všech prvků, nacházejících se v jiné kolekci

## 2. Metody pro ubírání kolekce:

- `void clear()` – odstranění všech prvků z kolekce
- `boolean remove(E e)` – odstranění jednoho prvku
- `boolean removeAll(Collection <?> c)` – odstranění všech prvků, nacházejících se v jiné kolekci
- `boolean retainAll(Collection <?> c)` – ponechání pouze prvků, nacházejících se v jiné kolekci

## 3. Logické operace

- `int size()` – vrátí aktuální počet prvků kolekce
- `boolean isEmpty()` – test na prázdnou kolekci
- `boolean contains(E e)` – test, zda je daný prvek obsažen (alespoň jednou) v kolekci
- `boolean containsAll(Collection <?> c)` – test, zda jsou všechny prvky jiné kolekce obsaženy v kolekci

## 4. Převod kolekce na běžné pole

- `Object[] toArray()`

## 5. Získání přístupového objektu

- `Iterator <E> iterator()`

# 6.6. Rozhraní List

- přidává metody, které zavádějí možnost práce s prvky kolekce pomocí indexů:

## 1. Změny v kolekci:

- `void add(int index, E e)` – přidání prvku; prvky s vyšším indexem budou posunuty o jeden výše
- `E set(int index, E e)` – změna prvku na daném indexu; prvkům s vyšším indexem se indexy nemění
- `E remove(int index)` – odstranění prvku; prvky s vyšším indexem budou posunuty o jeden níže

## 2. Získání obsahu kolekce

- `E get(int index)` – vrátí prvek na daném indexu, ale současně jej ponechá v kolekci (rozdíl od `remove()`)
- `int indexOf(E e)` – vrátí index prvního nalezeného prvku, nebo -1, není-li prvek v kolekci
- `int lastIndexOf(E e)` – vrátí index posledního nalezeného prvku

- `List<E> subList(int startIndex, int endIndex)` – vrátí podseznam, ve kterém budou prvky od `startIndex` včetně do `endIndex-1`; Pozor, jedná se o mělkou kopii.
- rozhraní `Set` zděděné od `Collection` nepřidává žádné metody navíc

## 6.6.1. Implementace pomocí `ArrayList`

- nejpoužívanější implementace seznamu, na kterou přecházíme, přestávají-li nám stačit klasická pole
- kromě své velikosti, tj. aktuálního počtu prvků vraceného metodou `size()` má i kapacitu – důležité pro efektivitu implementace

## Příklad 6.7.

```
import java.util.*;  
  
public class ArrayListMetodyZCollection {  
    public static void tiskni(String jmeno, List<String> seznam) {  
        int vel = seznam.size();  
        System.out.print(jmeno + " (" + vel + ") : ");  
        for (int i = 0; i < vel; i++) {  
            System.out.print("[" + i + "]=" + seznam.get(i) + ", ");  
        }  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        List<String> sezn1 = new ArrayList<>();  
        System.out.println("seznam je prazdny: " + sezn1.isEmpty());  
        sezn1.add("prvni");  
        sezn1.add("druhy");  
        sezn1.add("prvni");  
        tiskni("seznam", sezn1);  
  
        System.out.println("\nPridavani a ubirani prvku");  
        List<String> sezn2 = new ArrayList<>(sezn1);  
        System.out.println("seznam je prazdny: " + sezn2.isEmpty());  
        sezn2.add("treti");  
        tiskni("seznam", sezn2);  
        sezn2.remove("prvni");  
        tiskni("seznam", sezn2);  
        sezn2.removeAll(sezn1);  
        tiskni("seznam", sezn2);  
        sezn2.addAll(sezn1);  
        tiskni("seznam", sezn2);  
        sezn2retainAll(sezn1);  
        tiskni("seznam", sezn2);  
  
        System.out.println("\nHledani prvku");  
        List<String> sezn3 = new ArrayList<>(sezn1);  
        sezn3.add("ctvrty");  
        System.out.println("seznam obsahuje 'paty': " + sezn3.contains("paty"));  
        System.out.println("seznam obsahuje seznam: " + sezn3.containsAll(sezn1));  
  
        System.out.println("\nPrevod na pole seznam");  
        String[] poleRetezcu = sezn3.toArray(new String[0]);  
        System.out.println(Arrays.toString(poleRetezcu));  
        System.out.println(Arrays.asList(poleRetezcu));  
  
        String[] hodnoty = {"jedna", "dva", "tri"};  
        List<String> sezn4 = new ArrayList<>();  
        for (int i = 0; i < hodnoty.length; i++) {  
            sezn4.add(hodnoty[i]);  
        }  
    }  
}
```

```

    }

List<String> sezn5 = new ArrayList<>(
    Arrays.asList(hodnoty));
tiskni("sezn4", sezn4);
tiskni("sezn5", sezn5);
}
}

```

Vypíše:

```

seznl je prazdny: true
seznl (3) : [0]=prvni, [1]=druhy, [2]=prvni,

Pridavani a ubirani prvku
seznl je prazdny: false
seznl (4) : [0]=prvni, [1]=druhy, [2]=prvni, [3]=treti,
seznl (3) : [0]=druhy, [1]=prvni, [2]=treti,
seznl (1) : [0]=treti,
seznl (4) : [0]=treti, [1]=prvni, [2]=druhy, [3]=prvni,
seznl (3) : [0]=prvni, [1]=druhy, [2]=prvni,

Hledani prvku
seznl obsahuje 'paty': false
seznl obsahuje sezn1: true

Prevod na pole sezn3
[prvni, druhy, prvni, ctvrty]
[prvni, druhy, prvni, ctvrty]
seznl (3) : [0]=jedna, [1]=dva, [2]=tri,
seznl (3) : [0]=jedna, [1]=dva, [2]=tri,

```

## Příklad 6.8.

Ve třídě `CeleCislo` je také překrytá metoda `toString()`, díky níž můžeme tisknout obsah celého seznamu najednou. Formálním parametrem metody `tiskni()` je objekt třídy `List`. To je nutné proto, že metoda `subList()` vrací rozhraní `List` nikoli implementaci `ArrayList`.

Na vráceném seznamu lze ukázat ještě něco, nač je třeba dát při práci s objekty pozor – kopie seznamu je mělká kopie což prakticky znamená, že pokud v kopii změníme hodnotu objektu, změní se tato i v originálu (nebo naopak). To je důvod, proč v kolekcích používáme téměř výhradně neměnné (*immutable*) objekty.

```
import java.util.*;  
  
class CeleCislo {  
    private int cislo;  
  
    CeleCislo(int i) { this.cislo = i; }  
  
    int getCislo() { return cislo; }  
  
    void setCislo(int i) { this.cislo = i; }  
  
    public String toString() { return ("'" + cislo); }  
}  
  
public class ArrayListVlastniTridaMetodyZList {  
    public static void tiskni(String jmeno, List<CeleCislo> li) {  
        int vel = li.size();  
        System.out.print(jmeno + " (" + vel + ") : ");  
        for (int i = 0; i < vel; i++) {  
            System.out.print("[ " + i + " ] = "  
                + li.get(i).getCislo() + ", ");  
        }  
        System.out.println();  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Vytvoreni seznamu");  
        List<CeleCislo> celySeznam = new ArrayList<>();  
        for (int i = 0; i < 5; i++) {  
            celySeznam.add(new CeleCislo(i + 10));  
        }  
        tiskni("celySeznam", celySeznam);  
        System.out.println("Tisk celeho seznamu: " + celySeznam);  
  
        System.out.println("Pridavani prvku");  
        celySeznam.add(2, new CeleCislo(77));  
        tiskni("celySeznam", celySeznam);  
        System.out.println("Vytvoreni podseznamu");  
        List<CeleCislo> podSeznam = celySeznam.subList(2, 5);  
        tiskni("podSeznam ", podSeznam);  
  
        celySeznam.get(3).setCislo(33);  
    }  
}
```

```

        tiskni("celySeznam", celySeznam);
        tiskni("podSeznam ", podSeznam);
    }
}

```

Vypíše:

```

Vytvoreni seznamu
celySeznam (5) : [0]=10, [1]=11, [2]=12, [3]=13, [4]=14,
Tisk celeho seznamu: [10, 11, 12, 13, 14]
Pridavani prvku
celySeznam (6) : [0]=10, [1]=11, [2]=77, [3]=12, [4]=13, [5]=14,
Vytvoreni podseznamu
podSeznam (3) : [0]=77, [1]=12, [2]=13,
celySeznam (6) : [0]=10, [1]=11, [2]=77, [3]=33, [4]=13, [5]=14,
podSeznam (3) : [0]=77, [1]=33, [2]=13,

```

## 6.7. Zajištění algoritmů – třída Collections

- pro běžné pole existuje třída `Arrays` se svými statickými metodami, které slouží pro realizaci algoritmů nad poli – vyplnění, seřazení a vyhledávání v poli
- pro třídy, které implementují rozhraní `Collection`, existuje třída `java.util.Collections`
  - poskytuje podobné metody jako `Arrays` a ještě mnohé další, většinou ale jen pro seznamy `List` nikoliv pro množiny `Set`
  - všechny metody jsou opět statické

Některé metody třídy `Collections`:

- vyplnění seznamu jednou (stejnou) hodnotou
  - `void fill(List<E> list, E e)`
- pro řazení lze opět použít oba již známé způsoby – **přirozené řazení** (`compareTo()` patřící třídě řazených objektů) nebo **absolutní řazení** (metoda `compare()` z vnějšího komparátoru)
  - `void sort(List<E> list)` – vzestupné přirozené řazení
  - `void sort(List<E> list, Comparator<E> c)` – absolutní řazení podle komparátoru
- v seřazeném seznamu lze rychle vyhledávat
  - `int binarySearch(List<E> list, E key)` – hledání s využitím `compareTo()`
  - `int binarySearch(List<E> list, E key, Comparator<E> c)` – hledání s pomocí externího komparátoru
- vyhledávání v neseřazeném seznamu (trvá ale mnohem delší dobu)
  - `int indexOf(E e)`

- nalezení prvku s minimální a maximální hodnotou v neseřazeném seznamu (opět lze použít obou typů porovnávání)
  - `E max(Collection<E> coll)`
  - `E max(Collection<E> coll, Comparator<E> comp)`
  - `E min(Collection<E> coll)`
  - `E min(Collection<E> coll, Comparator<E> comp)`
- otočení pořadí (většinou již seřazeného) seznamu
  - `void reverse(List<E> l)`
- pokud k řazení využíváme způsob přirozeného řazení, pak lze lehce změnit vzestupné pořadí na sestupné tím, že si necháme vygenerovat komparátor měnící pořadí
  - `Comparator<E> reverseOrder() nebo Comparator<E> reverseOrder(Comparator<E> comp)`
- „zamíchání“ seznamu – výhodné když jsou v seznamu jednoznačně definované prvky (např. pexeso) a my jen potřebujeme vždy jejich jiné pořadí
  - `void shuffle(List<E> list)`

## Příklad 6.9.

```
public class OsobaCollections {  
    public static void main(String[] args) {  
        List<Osoba> sez = new ArrayList<>();  
        sez.add(new Osoba(186, 82.5, "muz"));  
        sez.add(new Osoba(172, 63.0, "zena"));  
        sez.add(new Osoba(105, 26.1, "dite"));  
        sez.add(new Osoba(116, 80.5, "obezni trpaslik"));  
        System.out.println("Neserazeno: " + sez);  
  
        Collections.sort(sez, new KomparatorOsobyPodleVahy());  
        System.out.println("Absolutni razeni podle vahy: " + sez);  
  
        Collections.reverse(sez);  
        System.out.println("Podle vahy sestupne: " + sez);  
  
        Collections.sort(sez);  
        System.out.println("Prirozeno razeni podle vysky: " + sez);  
  
        Collections.shuffle(sez);  
        System.out.println("Zamichano: " + sez);  
  
        System.out.println("Nejvyssi:" + Collections.max(sez));  
        System.out.println("Nejlehci:" +  
            Collections.min(sez, new KomparatorOsobyPodleVahy()));  
  
        Collections.fill(sez, new Osoba(180, 75.0, "robot"));  
        System.out.println("Vyplneno: " + sez);  
    }  
}
```

Vypíše např.:

```
Neserazeno: [  
vy = 186, va = 82.5, muz,  
vy = 172, va = 63.0, zena,  
vy = 105, va = 26.1, dite,  
vy = 116, va = 80.5, obezni trpaslik]  
Absolutni razeni podle vahy: [  
vy = 105, va = 26.1, dite,  
vy = 172, va = 63.0, zena,  
vy = 116, va = 80.5, obezni trpaslik,  
vy = 186, va = 82.5, muz]  
Podle vahy sestupne: [  
vy = 186, va = 82.5, muz,  
vy = 116, va = 80.5, obezni trpaslik,  
vy = 172, va = 63.0, zena,  
vy = 105, va = 26.1, dite]  
Prirozeno razeni podle vysky: [  
vy = 105, va = 26.1, dite,  
vy = 116, va = 80.5, obezni trpaslik,  
vy = 172, va = 63.0, zena,
```

```

vy = 186, va = 82.5, muz]
Zamichano: [
vy = 105, va = 26.1, dite,
vy = 172, va = 63.0, zena,
vy = 116, va = 80.5, obezni trpaslik,
vy = 186, va = 82.5, muz]
Nejvyssi:
vy = 186, va = 82.5, muz
Nejlehci:
vy = 105, va = 26.1, dite
Vyplneno: [
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot,
vy = 180, va = 75.0, robot]

```

## 6.8. Postupný průchod kolekcí

- možný pomocí indexů – jen pro seznamy
- nebo iterátorů – obecně pro všechny kolekce
- od JDK 1.5 jsou iterátory dvou typů
  - původní objekt třídy `Iterator`
  - „*For-Each*“ pomocí klíčového slova `for` (zjednodušený iterátor)
- rychlosti průchodu indexací a iterátorem jsou stejné
- použijeme-li jednu implementaci kolekce (např. `ArrayList`) a budeme ji procházet pomocí iterátoru, můžeme někdy v budoucnu (např. z důvodu zvýšení rychlosti) snadno změnit tento typ kolekce za jiný
- pouze se vytvoří jiná kolekce, ale vkládání a výběr prvků a průchod kolekcí zůstanou naprosto nezměněny

```
Collection<Typ> c = new ArrayList<>();
```

nebo později:

```
Collection<Typ> c = new HashSet<>();
```

### 6.8.1. For-Each

- díky typovaným kolekcím nejjednodušší a nejpoužívanější
  - je to též bezpečnější konstrukce
- musí projít celou kolekcí od prvního do posledního prvku
  - to v naprosté většině případů chceme
- lze jej použít i na běžné pole (`TypovanaKolekceFor.java`)

## Příklad 6.10.

```
List<Integer> seznam = new ArrayList<>();
seznam.add(new Integer(1));
seznam.add(new Integer(2));
for (Integer prvek: seznam) {
    System.out.print(prvek.intValue() + ", ");
}

System.out.println("\nBezne pole");
int[] pole = {5, 6, 7, 8, 9};
for (int hodnota : pole) {
    System.out.print(hodnota + ", ");
}
```

Vypíše:

```
1, 2,
Bezne pole
5, 6, 7, 8, 9,
```

## 6.8.2. Iterátory

- jedná se o návrhový vzor **Iterátor (Iterator)**

- zprostředuje jednoduchý a sekvenční přístup k objektům uloženým v nějaké složitější datové struktuře, přičemž implementace této struktury je uživateli skryta
- iterátory mají v Javě silnou podporu – jsou odvozeny od rozhraní `java.util.Iterator<Typ>` (nepoužívat „starý“ `Enumeration`)
  - dají se použít pro všechny třídy, které implementují rozhraní `java.lang.Iterable` – to jsou všechny třídy kolekcí a mnohé další
- objekty, které toto rozhraní implementují, vrací rozhraní iterátoru metodou `Iterator<Typ> iterator()`
- v porovnání s *For-Each* používáme jen ve speciálních případech
  - nejčastěji přeskakování (vynechání) některých prvků nebo odstranění prvků z kolekce během průchodu kolekcí
- samotný iterátor umožňuje pouze tři aktivity:
  - `boolean hasNext()` – zjistí, zda v kolekci existuje ještě nějaký prvek
  - `Object next()` – přesune se na další prvek a vrátí jej
  - `void remove()` – zruší prvek odkazovaný předchozím `next()`, NEvrací rušený prvek

### Výstraha

Iterátor je jednopřůchodový – po průchodu pozbývá funkčnost. Pro případný další průchod se musí znova vygenerovat.

- kolekce odvozené od `List` dokáží metodou `listIterator()` vrátit objekt splňující rozhraní `ListIterator<Typ>`
- má navíc metody:
  1. umožňující pohyb od konce seznamu k jeho počátku
  2. změna prvku získaného předchozím `next()` nebo `previous()`
  3. získání indexu pomocí iterátoru
  4. přetížená metoda `listIterator(int zacIndex)` vrací iterátor fungující od uvedeného počátečního indexu

## Poznámka

Použití specializovaného iterátoru je třeba zvážit, protože můžeme přijít v budoucnu o možnost zaměňovat jednotlivé rozhraní (typy) kolekcí.

## Příklad 6.11.

Na dvou způsobech tisku je vidět, jak se lze iterátor využít cyklu `for` i `while`. U cyklu `while` se nezpracovávají první dvě hodnoty. Dále je vidět běžné využití překryté metody `toString()`.

```
import java.util.*;  
  
public class IteratorZakladniPouziti {  
    public static void main(String[] args) {  
        List<Integer> seznam = new ArrayList<>();  
        for (int i = 0; i < 10; i++) {  
            seznam.add(new Integer(i + 20));  
        }  
  
        for (Iterator<Integer> it = seznam.iterator(); it.hasNext(); ) {  
            System.out.print(it.next() + ", ");  
        }  
        System.out.println();  
  
        Iterator<Integer> it = seznam.iterator();  
        it.next();  
        it.next();  
        while (it.hasNext()) {  
            System.out.print(it.next() + ", ");  
        }  
    }  
}
```

Vypíše:

```
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,  
22, 23, 24, 25, 26, 27, 28, 29, ►
```

## 6.8.2.1. Změna kolekce při použití iterátoru

- během iterátor používání iterátoru se nesmí změnit procházená kolekce
  - v opačném případě je vyhozena výjimka `ConcurrentModificationException`
- jedinou možnou změnou kolekce je odstranění prvku, ale ne metodou kolekce, ale metodou `remove()` iterátoru
- je také třeba si uvědomit, že každé volání `next()` posunuje iterátor na další prvek kolekce
  - potřebujeme-li tedy s prvkem získaným pomocí `next()` pracovat opakovaně, je nutné vytvořit pomocnou referenční proměnnou

```
public class IteratorProblemy {  
    public static void main(String[] args) {  
        List<Integer> seznam = new ArrayList<>();  
        for (int i = 0; i < 11; i++) {  
            seznam.add(new Integer(i + 20));  
        }  
  
        int suma = 0;  
        for (Iterator<Integer> it = seznam.iterator(); it.hasNext(); ) {  
            // spravne - pomocna promenna  
            Integer prvek = it.next();  
            System.out.print(prvek + " ");  
            suma += prvek.intValue();  
  
            // chybne - dvakrát next()  
//            System.out.print(it.next() + ", ");  
//            suma += it.next().intValue();  
        }  
//        it.next(); // není přístupný - nelze udělat chybu  
        System.out.println(" = " + suma);  
  
        Iterator<Integer> it1 = seznam.iterator();  
//        Iterator<Integer> it2 = seznam.iterator();  
        it1.next();  
        it1.next();  
        while (it1.hasNext()) {  
            System.out.print(it1.next() + ", ");  
            it1.remove();  
        }  
  
        System.out.println("\nPo ubrani ze seznamu");  
        Iterator<Integer> it2 = seznam.iterator();  
        while (it2.hasNext()) {  
            System.out.print(it2.next() + ", ");  
        }  
        System.out.println();  
    }  
}
```

vypíše:

```
20 + 21 + 22 + 23 + 24 + 25 + 26 + 27 + 28 + 29 + 30 + = 275
22, 23, 24, 25, 26, 27, 28, 29, 30,
Po ubrani ze seznamu
20, 21, ►
```

## 6.9. Ochrana proti nekonzistenci dat

- kolekce obsahují zabudovanou automatickou ochranu proti případu, kdy se pokusíme změnit kolekci za současného používání pohledu (iterátor, podseznam, ...)
- ochrana způsobí vyvolání výjimky `ConcurrentModificationException`

### Příklad 6.12.

Výjimku vyvolá i akce s kolekcí v jednom a též procesu. Není tedy pravda, že pro vyhození této výjimky musí být do kolekce souběžný přístup z více procesů.

```
import java.util.*;
public class IteratorZmenaKolekce {
    public static void main(String[] args) {
        List<Integer> kolekce = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            kolekce.add(new Integer(i));
        }

        // Iterator<Integer> it = kolekce.iterator();
        // System.out.println(it.next());
        // kolekce.add(new Integer(20));
        // // zde vyhodi výjimku
        // System.out.println(it.next());

        for (Integer i : kolekce) {
            System.out.println(i);
            kolekce.add(new Integer(20));
        }
    }
}
```

Vypíše:

```
0
Exception in thread "main" java.util.ConcurrentModificationException
```

## 6.10. Výhodnost jednotlivých seznamů

- pokud se používají jen metody z rozhraní `List`, je záměna různých typů seznamů velice jednoduchá a zvýšení (nebo též snížení) výkonu může být ohromující
- čas v benchmarkách je vypisován v milisekundách a samozřejmě závisí na typu procesoru (Intel i7-3520, 2,9 GHz) a velikosti operační paměti (8 GB)
- byly testovány dvě třídy – běžný `ArrayList`, `LinkedList`

- velikost seznamu byla 100 000 prvků

|                                     | <b>ArrayList</b> | <b>LinkedList</b> |
|-------------------------------------|------------------|-------------------|
| naplnění                            | 16               | 10                |
| průchod indexací                    | 10               | 3579              |
| průchod iterátorem                  | 16               | 7                 |
| vypuštění poloviny indexací zezadu  | 171              | 1734              |
| vložení poloviny indexací           | 172              | 1857              |
| vypuštění poloviny indexací zepředu | 375              | 1834              |
| clear a naplnění                    | 10               | 11                |
| vypuštění poloviny iterátorem       | 375              | 10                |

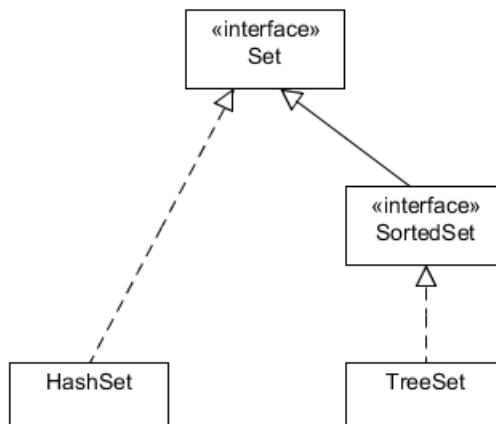
Jaké zajímavé závěry plynou z tabulky.

1. Průchod ArrayListu pomocí indexů a pomocí iterátoru je zcela srovnatelný, z čehož vyplývá, že je výhodnější používat iterátoru, protože to nám dává možnost budoucí záměny ArrayListu za jiný typ kolekce.
2. Tytéž výsledky – a tedy i závěry – jsou i u vypouštění prvků pomocí indexace a iterátoru (zepředu).
3. Jakákoli indexace v LinkedListu je extrémně pomalá.
4. Hromadné vypouštění nebo vkládání prvků do ArrayListu je časově velice náročné. U vypouštění prvků je LinkedList zhruba 30 krát výkonnější, a je tedy velmi vhodné jej pro tento typ aplikace použít. Na druhé straně je ale problémem LinkedListu vkládání, které je zhruba 10krát pomalejší než u ArrayListu. Vkládat doprostřed lze totiž pouze indexací, která je u LinkedListu extrémně pomalá (vkládání pomocí iterátoru vyvolá výjimku). Záleží ale také na tom, kam se vkládá. Pokud by bylo vkládání jen na konec či na začátek, pak by bylo jistě rychlejší.

# Kapitola 7. Kolekce – množiny a mapy

## 7.1. Množiny – rozhraní Set

- od rozhraní `java.util.Collection` je zděděno i rozhraní `java.util.Set`
  - představuje základ pro kolekce typu množiny
- množina se od seznamu liší tím, že umožňuje uschovávat pouze jeden prvek stejné hodnoty – všechny prvky množiny mají unikátní hodnotu
  - to vyžaduje, aby každý objekt vkládaný do množiny měl v závislosti na použité implementaci definovanou určitou metodu zajišťující test shody
- pro implementaci se používá nejčastěji třída `java.util.HashSet`
  - unikátnost je zajištěna tím, že každý objekt vkládaný do množiny musí mít definovány metody `equals()` a `hashCode()`
  - výhodou neseřazené množiny oproti neseřazenému seznamu je větší rychlosť vyhledání prvku ( $O(1)$ )
  - nevýhodou oproti seznamu je, že množina nezaručuje, že budou vložené prvky uchovávány v nějakém definovaném pořadí
- druhá konkurenční implementace je pomocí třídy `java.util.TreeSet` která implementuje `SortedSet`



- v `TreeSet` se průběžně udržují prvky seřazené (využívá stromovou strukturu)
  - unikátnost je zajištěna dvěma způsoby:
    1. každý objekt vkládaný do množiny musí implementovat rozhraní `Comparable` – přirozené řazení
    2. třídě `TreeSet` je v konstruktoru předán odkaz na existující `Comparator` – absolutní řazení
  - již v době vložení se vkládaný prvek zařadí do odpovídajícího pořadí vzhledem ke stávajícím prvkům
  - vkládání do `TreeSet` (i další operace) je pomalejší než do `HashSet`
    - ◆ výhoda `TreeSet` je v možnosti snadno získat nejmenší či největší prvek, případně podmnožinu původní kolekce

- protože `Set` je stejně jako `List` zděděno od `Collection`, obsahuje stejné metody, které již byly popsány u `Collection`
  - cokoliv, co mělo něco společného s indexy, které jsou běžné v seznamech, v množinách neexistuje
- jakýkoliv průchod kolekcí je možný pouze pomocí *For-Each* nebo iterátorů, jejichž princip je naprosto stejný jako u seznamů

## Příklad 7.1.

Ukázka, jak pro množiny fungují běžné metody, jako je vkládání, zjištění velikosti kolekce, vyhledávání prvku, vypouštění prvku, průchod kolekcí pomocí iterátoru a vymazání obsahu kolekce.

```
public class HashSetATreeSet {  
  
    public static void naplneniATisk(Set<String> mnozina) {  
        System.out.println(mnozina.getClass().getSimpleName());  
        mnozina.add("treti");  
        mnozina.add("druhy");  
        mnozina.add("prvni");  
        // pokus o vložení stejného prvku  
        if (mnozina.add("treti") == false) {  
            System.out.println("'treti' podruhé nevložen");  
        }  
        System.out.println(mnozina.size() + " " + mnozina);  
        for (String s: mnozina) {  
            System.out.print(s + ", ");  
        }  
        if (mnozina.contains("treti") == true) {  
            System.out.println("\n'treti' je v mnozine");  
        }  
        mnozina.remove("treti");  
        System.out.println(mnozina);  
        mnozina.clear();  
    }  
  
    public static void main(String[] args) {  
        naplneniATisk(new HashSet<String>());  
        naplneniATisk(new TreeSet<String>());  
    }  
}
```

Vypíše:

```
HashSet  
'treti' podruhé nevložen  
3 [prvni, treti, druhý]  
prvni, treti, druhý,  
'treti' je v mnozine  
[prvni, druhý]  
TreeSet  
'treti' podruhé nevložen  
3 [druhy, prvni, treti]  
druhy, prvni, treti,  
'treti' je v mnozine  
[druhy, prvni]
```

## 7.1.1. Práce s vlastní třídou v množině

- předchozí příklad využíval skutečnosti, že třída `String` (stejně jako obalovací třídy) má správně naprogramované metody `compareTo()` (pro `TreeSet`) a/nebo `equals()` a `hashCode()` (pro `HashSet`)

- máme-li však pracovat s objekty vlastních typů, musíme tyto tři metody implementovat a překrýt – bez toho bude program chodit chybně



## Příklad 7.2.

Do množiny budeme ukládat typ `Hruska`. Metoda `hashCode()` je zde jednoduchá. Využíváme přirozeného řazení, takže `Hruska` implementuje `Comparable`. Pozor na typ parametru v metodě `equals()`. Ten musí být typu `Object` (nikoliv `Hruska`), aby došlo k překrytí metody `equals()` ze třídy `Object` – použitím `Hruska` by došlo pouze k přetížení a program by pracoval chybně. Anotace `@Override` nás na tuto chybu upozorní.

Třída `Pocitadlo` umožňuje počítat, kolikrát byly volány metody `hashCode()`, `equals()` a `compareTo()`.

```
package kolekce;
import java.util.*;

class Pocitadlo {
    public static int e = 0;
    public static int h = 0;
    public static int c = 0;
}

class Hruska implements Comparable<Hruska> {
    private int cena;

    Hruska(int cena) { this.cena = cena; }

    public String toString() { return "" + cena; }

    @Override
    // public boolean equals(Hruska o) { // chyba
    public boolean equals(Object o) {
        Pocitadlo.e++;
        System.out.print(" " + Pocitadlo.e + "e ");

        if (o == this) {
            return true;
        }
        if (o instanceof Hruska == false) {
            return false;
        }
        boolean stejnaCena = (cena == ((Hruska) o).cena);
        return stejnaCena;
    }

    @Override
    public int hashCode() {
        Pocitadlo.h++;
        System.out.print(" " + Pocitadlo.h + "h ");

        return cena;
    }

    public int compareTo(Hruska h) {
        Pocitadlo.c++;
        System.out.print(" " + Pocitadlo.c + "c ");
    }
}
```



```

        int cenaPom = h.cena;
        if (this.cena > cenaPom)
            return (+1);
        else if (this.cena == cenaPom)
            return (0);
        else
            return (-1);
    }
}

public class HruskyVMnozine {
    static void praceSHruskami(Set<Hruska> mnozina) {
        System.out.println("\n" + mnozina.getClass().getSimpleName());
        for (int i = 30; i < 40; i++) {
            mnozina.add(new Hruska(i));
        }
        mnozina.add(new Hruska(35));

        System.out.print("\npocet: " + mnozina.size() + " [");
        for (Hruska h: mnozina) {
            System.out.print(h + ", ");
        }
        System.out.println("]");
    }

    public static void main(String[] args) {
        praceSHruskami(new HashSet<Hruska>());
        System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
                           + ", c=" + Pocitadlo.c);

        Pocitadlo.e = 0;
        Pocitadlo.h = 0;
        Pocitadlo.c = 0;
        praceSHruskami(new TreeSet<Hruska>());
        System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
                           + ", c=" + Pocitadlo.c);
    }
}

```

Vypíše:

```

HashSet
1h 2h 3h 4h 5h 6h 7h 8h 9h 10h 11h 1e
pocet: 10 [34, 35, 32, 33, 38, 39, 36, 37, 31, 30, ]
e=1, h=11, c=0

TreeSet
1c 2c 3c 4c 5c 6c 7c 8c 9c 10c 11c 12c 13c 14c 15c 16c 17c 18c 19c 20c 21c ►
22c 23c 24c 25c 26c 27c 28c 29c
pocet: 10 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, ]
e=0, h=0, c=29

```

Z výpisu pro HashSet je vidět, že metoda compareTo() není vůbec zapotřebí. Dále je vidět, že platí kontrakt mezi metodami equals() a hashCode(). Při pokusu o vložení dalšího prvku do množiny je nejprve volána hashCode() a v případě, že vrátí odlišnou hodnotu, od hodnot z již vložených objektů, není třeba volat equals(). V případě, že hashCode() vrátí stejnou hodnotu (dodatečně vkládaná Hruska (35) ), equals() definitivně rozhodne, zda se tento objekt rovná (=nevloží se) či nerovná (=vloží se). Vložené prvky jsou v množině v „náhodném“ pořadí.

Z výpisu pro TreeSet, je vidět, že veškeré porovnávání je provedeno pomocí compareTo(). Je také vidět, že vkládání do TreeSet je pomalejší, protože se zároveň vkládaný prvek zařazuje mezi již vložené. Vložené prvky jsou v množině v vzestupném pořadí.

Z ukázky vyplývá, že pokud připravujeme entitní třídu, která má být vkládána do množiny, měla by mít implementovány metody hashCode(), equals() a compareTo(). Tím později neomezujeme použitou implementaci kolekce množiny.

### 7.1.1.1. Vkládaná třída neimplementuje Comparable

- pokud vkládaná třída neimplementuje Comparable, nelze v této podobě TreeSet použít
  - po spuštění je vyhozena výjimka Hruska cannot be cast to java.lang.Comparable
- situace se řeší tím, že připravíme komparátor pro absolutní řazení a jeho instanci předáme jako parametr konstruktoru TreeSet
  - TreeSet bude pak používat pro práci s vloženými objekty pouze absolutní řazení
  - komparátor se typicky připravuje jako anonymní třída (viz dále), jejíž instance je přiřazena do statické konstanty, zde PODLE\_CENY

```
public static final Comparator<Hruska> PODLE_CENY =
        new Comparator<Hruska>() {
    public int compare(Hruska h1, Hruska h2) {
        Pocitadlo.c++;
        System.out.print(" " + Pocitadlo.c + "c ");

        if (h1.getCena() > h2.getCena())
            return (+1);
        else if (h1.getCena() == h2.getCena())
            return (0);
        else
            return (-1);
    }
};
```

- je ale možné připravit i novou třídu Porovnavac
  - ♦ zde je navíc použit trik pro zjednodušení porovnání, kdy dvě ceny jednoduše odečteme a vrátíme rozdíl

```
final class Porovnavac implements Comparator<Hruska> {
    public int compare(Hruska h1, Hruska h2) {
        Pocitadlo.c++;
        System.out.print(" " + Pocitadlo.c + "c ");
```

```

        return h1.getCena() - h2.getCena();
    }
}

```

z výpisu je vidět, že program pracuje stejně jako v předchozím případě

```

TreeSet
1c 2c 3c 4c 5c 6c 7c 8c 9c 10c 11c 12c 13c 14c 15c 16c 17c 18c 19c 20c 21c ▶
22c 23c 24c 25c 26c 27c 28c 29c
pocet: 10 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, ]
e=0, h=0, c=29

```

## ■ metoda `praceSHruskami()` zůstala nezměněna

```

import java.util.*;

class Pocitadlo {
    public static int e = 0;
    public static int h = 0;
    public static int c = 0;
}

class Hruska {
    private int cena;

    Hruska(int cena) { this.cena = cena; }

    public int getCena() { return cena; }

    public String toString() { return "" + cena; }
}

final class Porovnavac implements Comparator<Hruska> {
    public int compare(Hruska h1, Hruska h2) {
        Pocitadlo.c++;
        System.out.print(" " + Pocitadlo.c + "c ");

        return h1.getCena() - h2.getCena();
    }
}

public class HruskyComparator {

    public static final Comparator<Hruska> PODLE_CENY =
            new Comparator<Hruska>() {
        public int compare(Hruska h1, Hruska h2) {
            Pocitadlo.c++;
            System.out.print(" " + Pocitadlo.c + "c ");

            if (h1.getCena() > h2.getCena())
                return (+1);
            else if (h1.getCena() == h2.getCena())
                return 0;
            else
                return (-1);
        }
    }
}

```

```

        return (0);
    else
        return (-1);
}
};

static void praceSHruskami(Set<Hruska> mnozina) {
    System.out.println("\n" + mnozina.getClass().getSimpleName());
    for (int i = 30; i < 40; i++) {
        mnozina.add(new Hruska(i));
    }
    mnozina.add(new Hruska(35));

    System.out.print("\npocet: " + mnozina.size() + " [");
    for (Hruska h: mnozina) {
        System.out.print(h + ", ");
    }
    System.out.println("]");
}

public static void main(String[] args) {
//    praceSHruskami(new TreeSet<Hruska>(new Porovnavac()));
    praceSHruskami(new TreeSet<Hruska>(PODLE_CENY));
    System.out.println("e=" + Pocitadlo.e + ", h=" + Pocitadlo.h
        + ", c=" + Pocitadlo.c);
}
}
}

```

### 7.1.1.2. Vkládaná třída nepřekrývá `equals()` a `hashCode()`

- tyto dvě metody se dědí z `Object`, takže je možné i pro tuto třídu `Hruska` použít ihned `HashSet`

```

class Hruska {
    private int cena;

    Hruska(int cena) { this.cena = cena; }

    public int getCena() { return cena; }

    public String toString() { return "" + cena; }
}

```

```
praceSHruskami(new HashSet<Hruska>());
```

protože je však `equals()` naprogramována v `Object` na nejpřísnější porovnání, kdy jsou objekty stejné, pokud se jedná o stejně instance, bude do množiny chybně vložena již existující `Hruska(35)`

```
HashSet
pocet: 11 [36, 35, 34, 30, 32, 31, 35, 39, 37, 33, 38, ]
```

## 7.1.2. Problémy objektů v hešovacích třídách

- hešování (*hashing, rozptylové tabulky*) je jednou ze základních programovacích technik – podrobně viz v KIV/PPA2 a KIV/PT
  - pomáhá výrazně urychlit vkládání a výběr do/z kolekcí
  - implementační podrobnosti nás zatím nezajímají – ty jsou již vyřešeny ve třídě `HashSet`
  - podstatné pro využití této techniky je to, že vkládané objekty by měly poskytovat službu „reprezentuj se unikátním celým číslem“
  - v Javě se očekává, že tato služba bude realizována pomocí metody `int hashCode()`
  - pokud metoda nevrací pro různé objekty unikátní čísla, program funguje, ovšem mnohem méně efektivně – implementační vysvětlení viz ve zmíněných předmětech
    - ◆ vysvětlení z rozhraní je to, že při stejných hešovacích kódech se musí pro ověření shodnosti následně volat metoda `equals()`
    - ◆ ve skutečnosti musíme současně překrýt metody `hashCode()` a `equals()`
- častou chybou je, že hešovací kód už vypočítává metodu `hashCode()`, kterou každý objekt dědí od třídy `Object`
  - zdánlivě je vše v naprostém pořádku a nemusíme učinit naprosto nic a program bude přeložen správně
  - správně fungovat bude ale jen pro objekty všech obalovacích tříd primitivních datových typů (`Integer`, `Double` atp.) a třídy `String`
    - ◆ ty jsou pro svou jednoduchost často používány jako ukázka – problémy při přechodu na reálnou aplikaci

### 7.1.2.1. Metoda `equals()`

- pět pravidel obecného kontraktu (opakování z dřívějška)

1. objekt se musí vždy rovnat sám sobě – reflexivnost

`x.equals(x)` je vždy `true`

2. objekty se musí rovnat křížem – symetričnost

`y.equals(x) == x.equals(y)`

3. rovná-li se jeden objekt druhému a druhý třetímu, musí se rovnat i první třetímu – tranzitivita

`jestliže x.equals(y) == true a y.equals(z) == true musí x.equals(z) == true`

4. jsou-li si dva objekty rovné, musí si být rovné tak dlouho, dokud u některého z nich nenastane změna – konzistentnost

upozorňuje na problém měnitelných objektů

## 5. žádný objekt se nesmí rovnat null

```
x.equals(null) == false
```

pravidlo v sobě zahrnuje i nepřípustnost vyvolání výjimky `NullPointerException` – místo reference na porovnávaný objekt byla metodě `equals()` předána hodnota `null`

- při porovnávání primitivních hodnot typu `float` nebo `double` je vhodné tyto hodnoty převést na celočíselný typ pomocí metod obalovacích tříd `Float.floatToIntBits()` nebo `Double.doubleToLongBits()`
  - pak porovnáváme pomocí `==` typy `long` nebo `int`
  - vyhneme se případným problémům s okrajovými hodnotami typu `Float.NaN` apod., nepřesnému porovnávání „velmi podobných“ čísel atd.

## Příklad 7.3.

Například komparátor podle `double` atributu `vaha` by měl nejlépe vypadat takto:

```
class KomparatorOsobyPodleVahy implements  
Comparator<Osoba> {  
    public int compare(Osoba o1, Osoba o2) {  
        double v1 = o1.vaha;  
        double v2 = o2.vaha;  
        long lv1 = Double.doubleToLongBits(v1);  
        long lv2 = Double.doubleToLongBits(v2);  
        if (lv1 == lv2)  
            return 0;  
        if (v1 > v2)  
            return +1;  
        else  
            return -1;  
    }  
}
```

### 7.1.2.2. Metoda `hashCode()`

Tři pravidla obecného kontraktu:

1. pro tentýž objekt musí `hashCode()` vracet vždy stejný `int`
2. rozhodla-li `equals()`, že jsou si dva objekty rovny, musí `hashCode()` vrátit stejný `int`
3. nejsou-li si objekty rovny podle `equals()`, mohou mít stejný hešovací kód – nevhodná implementace `hashCode()` – významně snižuje efektivitu programu

### 7.1.2.3. Efektivní `hashCode()`

- nestejné hešovací kódy pro nestejné objekty
- měly by mít navíc rovnoměrné rozložení

Návod z knihy **Java efektivně**:

a. int pomocná proměnná vysledek inicializovaná číslem 17

```
int vysledek = 17;
```

b. pro každou významnou stavovou proměnnou objektu, která byla použita pro porovnávání v metodě equals() vypočteme vlastní hešovací kód a uložíme jej do pomocné proměnné pom

Výpočet v závislosti na typu stavové proměnné:

- boolean pom = sp ? 0 : 1;
- byte, char, short, int pom = (int) sp;
- long pom = (int) (sp ^ (sp >>> 32));
- float pom = Float.floatToIntBits(sp);
- double long l = Double.doubleToLongBits(sp);  
pom = (int) (l ^ (l >>> 32));

■ odkaz na objekt

```
pom = (sp == null) ? 0 : sp.hashCode();
```

- pro pole vypočteme pom postupně pro každý prvek pole

Po vypočtení pom touto hodnotou ovlivníme proměnnou vysledek

```
vysledek = 37 * vysledek + pom;
```

a pokračujeme s další stavovou proměnnou.

c. po ovlivnění vysledek všemi významnými stavovými proměnnými objektu je vrácen

d. zkонтrolujeme na příkladech, zda stejné objekty (z pohledu equals()) vracejí stejné hešovací kódy – pokud ne, je třeba zjistit proč a chybu opravit

## Poznámka

Eclipse umožňuje vygenerovat metody equals() a hashCode(). Efektivita vygenerované hashCode() (liší se konstantami prvočísel) je maličko nižší než u výše uvedené hashCode(). Tzn. vyplatí se použít tu z Eclipse, která je zadarmo.

## Příklad 7.4. Ukázka použití různých přístupů k hešování

Je použita třída `Osoba`, která má základní atributy typu `boolean`, `int`, `double` a `String`. Tato třída má připravenou metodu `equals()` přesně podle doporučení, ale nemá překrytu metodu `hashCode()`.

Od třídy `Osoba` jsou odvozeny tři další třídy, které teprve metodu `hashCode()` překrývají. Všechny tři jsou funkční a pokud je použijeme pro malé objemy dat (řádu tisíců), nepoznáme při běhu programu výkonnostní rozdíl.

Testovací program vždy připraví pole objektů zvolené třídy. Pak (v měřené části) uloží všechny objekty z tohoto pole do `HashSet`. V následujícím měřeném úseku postupně v množině všechny prvky vyhledá.

Třída `NevhodnaOsoba` vrací jako hešovací kód pouze atribut `vyska`. Protože však výška může být pouze v rozsahu 170 až 200, je k dispozici pouze 31 různých hešovacích kódů. To způsobí výrazný nárůst doby běhu programu na počtu vložených prvků. Je to z toho důvodu, že jednak skutečný rozdíl mezi prvky musí rozpoznat až metoda `equals()`, ale zejména proto, že hešovací tabulka se změnila na 31 lineárně zřetězených seznamů. (= implementační detail)

Mnohem lepší je třída `PrijatelnaOsoba`, kde jednoduchou úpravou, která představuje pouze vynášení číselných atributů třídy (`vyska * vaha`), získáme znatelný nárůst výkonnosti.

Třída `PerfektniOsoba` má metodu `hashCode()` připravenu přesně podle návodu. Při jejím použití zjistíme, že potřebný čas (zejména pro vyhledávání) narůstá s počtem prvků velmi málo.

V příkladu je použita navíc speciální třída `NemennaPerfektniOsoba`. Ta vychází z předpokladu, že hodnoty atributů se nebudou měnit a je tedy možné hešovací kód vypočítat jednou provždy v konstruktoru. Tato konstantní hodnota je pak vracena překrytou metodou `hashCode()`. Je třeba zdůraznit, že se nemění princip výpočtu hešovacího kódu – je stále „perfektní“.

```
import java.util.*;  
  
class Osoba {  
    // zakladni stavove atributy  
    protected boolean muz;  
    protected int vyska;  
    protected double vaha;  
    protected String jmeno;  
    // bitovy obraz vahy pro hashCode() a equals()  
    protected long longVaha; // odvozeny atribut  
    private static Random r = new Random();  
  
    Osoba() {  
        this.muz = r.nextBoolean();  
        this.vyska = 170 + r.nextInt(31);      // <170; 200>  
        this.vaha = 50 + 50 * r.nextDouble(); // <50; 100>  
        this.longVaha = Double.doubleToLongBits(this.vaha);  
        byte[] b = new byte[5];  
        for (int i = 0; i < 5; i++)  
            b[i] = (byte) ((r.nextInt(26) + (byte) 'a'));  
        jmeno = new String(b);  
    }  
  
    public String toString() {  
        return jmeno + ", " + (muz ? "muz " : "zena") + ", " + vyska + ", " + vaha;  
    }  
}
```

```

}

public boolean equals(Object o) {
    if (o == this)
        return true;
    if (o instanceof Osoba == false)
        return false;
    Osoba os = (Osoba) o;
    boolean bMuz = this.muz == os.muz;
    boolean bVyska = this.vyska == os.vyska;
    boolean bVaha = this.longVaha == os.longVaha;
    boolean bJmeno = this.jmeno.equals(os.jmeno);
    return bMuz && bVyska && bVaha && bJmeno;
}
}

class NevhodnaOsoba extends Osoba {
    public int hashCode() {
        return vyska;
    }
}

class PrijatelnaOsoba extends Osoba {
    public int hashCode() {
        return (int) (vyska * vaha);
    }
}

class PerfektniOsoba extends Osoba {
    public int hashCode() {
        int vysledek = 17;
        int pom;
        pom = this.muz ? 0 : 1;
        vysledek = 37 * vysledek + pom;
        pom = this.vyska;
        vysledek = 37 * vysledek + pom;
        long l = Double.doubleToLongBits(this.vaha);
        pom = (int) (l ^ (l >>> 32));
        vysledek = 37 * vysledek + pom;
        pom = this.jmeno.hashCode();
        vysledek = 37 * vysledek + pom;
        return vysledek;
    }
}

class NemennaPerfektniOsoba extends PerfektniOsoba {
    protected int hashKod;
    NemennaPerfektniOsoba() {
        super();
        hashKod = super.hashCode();
    }

    public int hashCode() {

```

```

        return hashKod;
    }
}

public class TypyHashCode {
    static int pocet;

    public static void main(String[] args) {
        if (args[0] != null)
            pocet = Integer.parseInt(args[0]);

        Osoba[] pole = new Osoba[pocet];

        for (int i = 0; i < pocet; i++) {
//            pole[i] = new NevhodnaOsoba();
//            pole[i] = new PrijatelnaOsoba();
            pole[i] = new PerfektniOsoba();
//            pole[i] = new NemennaPerfektniOsoba();
        }

        System.out.println(pole[0].getClass().getName());
        long zac = System.nanoTime();
        HashSet<Osoba> mnOsob = new HashSet<>(pocet);
        for (int i = 0; i < pocet; i++) {
            mnOsob.add(pole[i]);
        }
        long kon = System.nanoTime();
        System.out.print("Vlozeni: " + mnOsob.size() + " (" + pocet + ") ");
        System.out.println("cas = " + (kon - zac) / 1000000);

        zac = System.nanoTime();
        int n = 0;
        for (int i = pocet - 1; i >= 0; i--)
            if (mnOsob.contains(pole[i]) == true)
                n++;

        kon = System.nanoTime();
        System.out.print("Pristup: "+n+" (" + pocet + ") ");
        System.out.println("cas = " + (kon - zac) / 1000000);
    }
}

```

|               | <b>prvků</b> | <b>10_000</b> | <b>20_000</b> | <b>40_000</b> | <b>80_000</b> |
|---------------|--------------|---------------|---------------|---------------|---------------|
| NevhodnaOsoba | vložení      | 72            | 287           | 1245          | 7734          |
|               | přístup      | 53            | 220           | 1574          | 9152          |

| prvků                 |         | 10_000 | 80_000 | 500_000 | 1_000_000 |
|-----------------------|---------|--------|--------|---------|-----------|
| PrijatelnaOsoba       | vložení | 3      | 66     | 1531    | 6111      |
|                       | přístup | 3      | 51     | 1494    | 5075      |
| PerfektniOsoba        | vložení | 2      | 14     | 88      | 191       |
|                       | přístup | 1      | 9      | 43      | 89        |
| NemennaPerfektniOsoba | vložení | 1      | 13     | 86      | 175       |
|                       | přístup | 1      | 7      | 35      | 66        |

1. Hodnoty pro 10\_000 prvků v kolekci, kdy je čas přístupu k objektům libovolné z uvedených čtyř tříd prakticky neměřitelný, jsou důvodem, proč je správné přípravě metody `hashCode()` obecně věnována tak malá pozornost.
  2. U NevhodnaOsoba je od určitého počtu (zde cca 30\_000) přístup pomalejší než vkládání, což je dáno implementací hešovací tabulky (vysvětlení viz v KIV/PPA2).
  3. Na příkladu NemennaPerfektniOsoba je vidět, že pokud nechceme měnit stav vkládaného objektu, můžeme zvýšit rychlosť asi o jednu čtvrtinu.
  4. Na porovnání NemennaPerfektniOsoba, PrijatelnaOsoba a PerfektniOsoba je dobře vidět, že není nutné mít obavy z přehnané časové náročnosti výpočtu „perfektního“ hešovacího kódu. U PerfektniOsoba je sice výpočet zpomalen asi o čtvrtinu (oproti NemennaPerfektniOsoba), ale zůstává stále velmi malý. To se u PrijatelnaOsoba (s jednodušším výpočtem) zdáleka nedá říci (čas narůstá nelineárně z důvodů opakujících se hešovacích kódů).
- jak generované hešovací kódy splňují podmínu unikátnosti pro 1 milion **různých** objektů:
- NevhodnaOsoba – 31 rozdílných
  - PrijatelnaOsoba – 11\_493 rozdílných a 988\_507 shodných
  - PerfektniOsoba – 999\_894 rozdílných a pouze 106 shodných

### 7.1.3. Použití Collections

- ze všech metod tohoto rozhraní lze pro množiny použít pouze dvě (čtyři) metody pro nalezení největšího a nejmenšího prvku
- jedna dvojice metod používá přirozeného a druhá absolutního řazení
  - `E max(Collection<E> coll)`
  - `E max(Collection<E> coll, Comparator<E> comp)`
  - `E min(Collection<E> coll)`
  - `E min(Collection<E> coll, Comparator<E> comp)`

### 7.1.4. Rozhraní SortedSet

- TreeSet implementuje rozhraní SortedSet a uschovává prvky seřazené

## ■ Ize proto použít několik dalších metod

- E first() – vrací první prvek množiny
- E last() – vrací poslední prvek množiny
- SortedSet<E> headSet(E horniMez) – vrací podmnožinu prvků, které jsou uloženy před hraničním prvkem
- SortedSet<E> tailSet(E dolniMez) – vrací podmnožinu prvků, které jsou uloženy za hraničním prvkem, včetně tohoto prvku
- SortedSet<E> subSet(E dolniMez, E horniMez) – vrací podmnožinu prvků v zadaných mezích

```
SortedSet<String> treeset = new TreeSet<>();
treeset.add("ahoj");
treeset.add("akce");
treeset.add("brzo");
treeset.add("eso");
System.out.println("Pocty slov od jednotlivych pismen");
for (char ch = 'a'; ch <= 'e'; ch++) {
    String zac = new String(new char[] {ch});
    String kon = new String(new char[] {(char)(ch+1)});
    System.out.println(zac + ": " + treeset.subSet(zac, kon).size());
}
```

## 7.1.5. Množinové operace a triky

### ■ vynikající např. při práci se jmény souborů atd.

### ■ odstranění duplicit z ArrayList

```
public class EliminaceDuplicit {
    public static void main(String[] args) {
        Collection<String> d = new ArrayList<>();
        d.add("prvni");
        d.add("druhy");
        d.add("prvni");
        System.out.println("Duplicitni: " + d);
        Collection<String> nd = new ArrayList<>(
            new HashSet<>(d));
        System.out.println("NEduplicitni: " + nd);
    }
}
```

Vypíše:

```
Duplicitni: [prvni, druhý, prvni]
NEduplicitni: [druhy, prvni]
```

### ■ průniky, sjednocení apod.

```

Set<String> m1 = new TreeSet<>();
m1.add("1");
m1.add("2");
m1.add("3");
m1.add("4");
Set<String> m2 = new TreeSet<>();
m2.add("2");
m2.add("3");

if (m1.containsAll(m2) == true)
    System.out.println(m2 + " je podmnozinou " + m1);

m2.add("5");
Set<String> sjednoceni = new TreeSet<>(m1);
sjednoceni.addAll(m2);
System.out.println(sjednoceni + " je sjednocenim " + m1 + " a " + m2);
Set<String> prunik = new TreeSet<>(m1);
prunik.retainAll(m2);
System.out.println(prunik + " je prunikem " + m1 + " a " + m2);

Set<String> rozdil = new TreeSet<>(m1);
rozdil.removeAll(m2);
System.out.println(rozdil + " je rozdilem " + m1 + " a " + m2);

Set<String> symetrickyRozdil = new TreeSet<>(m1);
symetrickyRozdil.addAll(m2);
Set<String> tmp = new TreeSet<>(m1);
tmp.retainAll(m2);
symetrickyRozdil.removeAll(tmp);
System.out.println(symetrickyRozdil +
                    " je symetrickym rozdilem " + m1 + " a " + m2);

```

Vypíše:

```

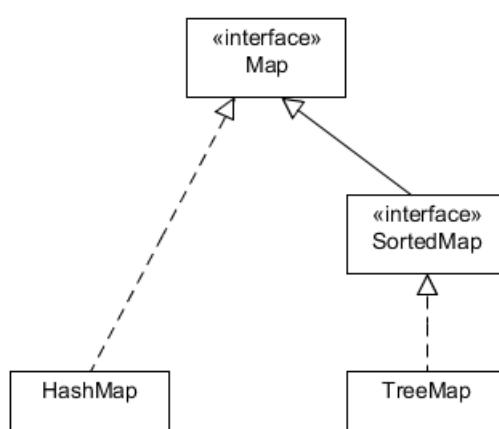
[2, 3] je podmnozinou [1, 2, 3, 4]
[1, 2, 3, 4, 5] je sjednocenim [1, 2, 3, 4] a [2, 3, 5]
[2, 3] je prunikem [1, 2, 3, 4] a [2, 3, 5]
[1, 4] je rozdilem [1, 2, 3, 4] a [2, 3, 5]
[1, 4, 5] je symetrickym rozdilem [1, 2, 3, 4] a [2, 3, 5]

```

## 7.2. Mapy – rozhraní Map

- též **slovníky** (*dictionary*) nebo **asociativní pole** (*associative array*)
- rozhraní Map nemá nic společného s rozhraním Collection, tj. mapy nejsou zaměnitelné za seznamy ani za množiny
  - z mapy ale lze získat seznam nebo množinu
- v mapě jeden prvek tvoří nedělitelná dvojice dvou objektů – klíče a hodnoty
  - pomocí klíče, který musí být neměnný a unikátní, se vyhledává hodnota

- hodnota může být proměnná (nevhodné) a může být duplicitní, tj. dva různé klíče mohou mít stejnou hodnotu
- struktura tříd a rozhraní odvozených od `Map` je ve zcela stejné strategii, jako u množiny



- třída `HashMap` je (opět) nejčastěji používaná „mapová“ třída
- v `TreeMap` jsou jednotlivé prvky seřazeny podle hodnoty klíče
  - `TreeMap` se používá (stejně jako `TreeSet`) méně – když potřebujeme mít prvky seřazené, nejčastěji z důvodu získání „podmapy“
- rozhraní `Map` umožňuje použít několika typů metod
  1. metody známé již ze seznamů a množin
    - `int size()` – vrací počet aktuálních prvků
    - `void clear()` – zruší všechny prvky
    - `boolean isEmpty()` – test na prázdnost
  2. vkládání a odstraňování prvků – prvkem se míní nedělitelná dvojice objektů – klíč (`key`) a hodnota (v metodách značená buď `value` nebo `entry`)
    - `V put(K key, V value)` – vložení prvku
    - `void putAll(Map<? extends K, ? extends V> m)` – vložení všech prvků z jiné mapy (mělká kopie)
    - `V remove(K key)` – odstranění prvku podle hodnoty klíče
  3. zjištění, zda je v množině buď objekt klíče nebo hodnoty
    - `boolean containsKey(K key)` – obsahuje klíč
    - `boolean containsValue(V value)` – obsahuje hodnotu
    - `V get(K key)` – podle klíče vrátí hodnotu – nejpoužívanější operace
  4. z mapy vytvoří množinu nebo seznam – musíme vybrat, zda to budou klíče či hodnoty (typické použití pro iterátory)

- Collection<V> values() – vrací hodnoty jako Collection (ne List nebo Set)
- Set<K> keySet() – vrací množinu klíčů
- Set<Map.Entry<K,V>> entrySet() – vrací množinu dvojic, prvky množiny jsou typu Map.Entry

```
public class ProchazeniMap {
    public static void main(String[] args) {
        Map<String, Integer> hm = new HashMap<String, Integer>();
        hm.put("prvni", 1);
        hm.put("druhy", 2);
        hm.put("treti", 3);

        // for-each
        Set<Map.Entry<String, Integer>> pomocnaMnozina = mapa.entrySet();
        for (Map.Entry<String, Integer> me: pomocnaMnozina) {
            System.out.print(me.getKey() + "=" + me.getValue() + ", ");
        }
        System.out.println();

        // for-each zkracene
        for (Map.Entry<String, Integer> me: mapa.entrySet()) {
            System.out.print(me.getKey() + "=" + me.getValue() + ", ");
        }
        System.out.println();

        // iterator
        for (Iterator<Map.Entry<String, Integer>>
             it = hm.entrySet().iterator();
             it.hasNext(); ) {
            Map.Entry<String, Integer> me = it.next();
//            it.remove();
            System.out.print(me.getKey() + "=" + me.getValue() + ", ");
        }
    }
}
```

Vypíše:

```
prvni=1, treti=3, druhý=2,
prvni=1, treti=3, druhý=2,
prvni=1, treti=3, druhý=2, ►
```

- u množiny hodnot (získaných pomocí values() ) získáváme mělké kopie – změna hodnoty prvku se projeví i v originální mapě

## Výstraha

Nelze měnit objekt hodnoty, lze měnit jen nastavení objektu představujícího hodnotu. To např. znamená, že pokud budeme mít mapu, ve které bude klíčem String se jménem člověka a hodnotou Integer s jeho váhou, nelze při ztlouštění nahradit tento Integer novým při zachování původního klíče (jména). Objektu typu Integer nelze měnit jeho hodnotu.

- chceme-li měnit hodnoty, máme tři základní možnosti:

1. zrušit celý prvek (jméno i váhu) a vložit nový (stejné jméno, jiná váha)
2. vložit nový prvek se stejným klíčem (stejné jméno, jiná váha)
3. zavést vlastní třídu Vaha, jejíž datový prvek lze měnit

## 7.2.1. Třída TreeMap

- používá se tehdy, potřebujeme-li mít klíče v mapě seřazené

- to není nutné z důvodů vyhledávání nějakého klíče – to stejně dobře (tj. rychle) poslouží i třída `HashMap`

- seřazení klíčů je nezbytné v tom případě, kdy potřebujeme získat z mapy:

- největší či nejmenší klíč
  - „podmapu“ v závislosti na hodnotě klíče

- `TreeMap` obsahuje všechny metody ze třídy `HashMap` a přidává k nim ještě metody:

- K `firstKey()` – vrací nejmenší (první v pořadí) klíč
  - K `lastKey()` – vrací největší (poslední v pořadí) klíč

- určitý komparátor pro absolutní řazení se zadá použitím přetíženého konstruktora `TreeMap(Comparator<K> comp)`

- použitím jen `TreeMap()`, bude použito přirozené řazení
    - ♦ je-li použito přirozené řazení, musí objekt klíče implementovat rozhraní `Comparable<K>`

- komparátor používaný v konkrétní `TreeMap` se dá zjistit metodou `Comparator<K> comparator()`

- vrací buď objekt použitého komparátoru (absolutní řazení) nebo `null` (přirozené řazení)

- metody z `TreeMap` vracející „podmapu“ vždy jako mělkou kopii

- `SortedMap<K, V> headMap(K doKlice)` – podmapa, kde klíče jsou ostře menší než daný klíč
  - `SortedMap<K, V> tailMap(K odKliceVcetne)` – podmapa, kde klíče jsou větší nebo rovny danému klíči
  - `SortedMap<K, V> subMap(K odKliceVcetne, K doKlice)` – podmapa „z prostředka“ stávající mapy

## Příklad 7.5. Nastavení default a uživatelských hodnot

Nastavení default a uživatelských hodnot – využívá toho, že vložení stejného klíče do mapy překryje původní hodnotu

```
public class NastaveniVMape {  
    private static String[] key =  
        {"pozadi", "popredi", "ramecek"};  
    private static String[] hodDef =  
        {"bila", "cerna", "modra"};  
    private static String[] hodUziv =  
        {null, "modra", "cervena"};  
  
    static Map<String, String> options(String[] hodnoty) {  
        Map<String, String> m = new HashMap<>();  
        for (int i = 0; i < key.length; i++) {  
            if (hodnoty[i] != null)  
                m.put(key[i], hodnoty[i]);  
        }  
        return m;  
    }  
  
    public static void main(String args[]) {  
        Map<String, String> defaultNastaveni = options(hodDef);  
        Map<String, String> uzivatelNastaveni = options(hodUziv);  
        Map<String, String> platneNastaveni =  
            new HashMap<>(defaultNastaveni);  
  
        platneNastaveni.putAll(uzivatelNastaveni);  
        System.out.println("Default: " + defaultNastaveni);  
        System.out.println("Uzivatel: " + uzivatelNastaveni);  
        System.out.println("Platne: " + platneNastaveni);  
    }  
}
```

Vypíše:

```
Default: {popredi=cerna, ramecek=modra, pozadi=bila}  
Uzivatel: {popredi=modra, ramecek=cervena}  
Platne: {popredi=modra, ramecek=cervena, pozadi=bila}
```

## Příklad 7.6. Zjištění frekvence slov

```
public class FrekvenceSlovPomociMapy {  
    public static void main(String args[]) {  
        String[] slova = {"lesni", "vily", "vence", "vily", "a",  
                          "psi", "z", "vily", "na", "ne", "vyli"};  
        Map<String, Integer> mapa = new TreeMap<>();  
        Integer ctnost;  
        for (int i = 0; i < slova.length; i++) {  
            int pom = 0;  
            if ((ctnost = mapa.get(slova[i])) != null) {  
                pom = ctnost.intValue();  
            }  
            mapa.put(slova[i], new Integer(pom + 1));  
        }  
  
        System.out.println("Nalezeno " + mapa.size() +  
                           " rozdílných slov");  
        System.out.println(mapa);  
    }  
}
```

Vypíše:

```
Nalezeno 9 rozdílných slov  
{a=1, lesni=1, na=1, ne=1, psi=1, vence=1, vily=3, vyli=1, z=1}
```

## 7.3. Složitější datové struktury

- dlouhou dobu u kolekcí vystačíme s vkládáním datového typu `String`, obalových datových typů a hodnotových neměnitelných tříd
- je třeba si ale uvědomit, že kolekce nijak typ vkládaných objektů neomezuje
  - do kolekce lze vložit jiná kolekce
  - tím lze vytvářet libovolně složité datové struktury
- typické příklady:
  - dvouozměrný seznam: `List<List<String>> dvourozmernySeznam;`

```
public static void dvourozmernySeznamRetezcu() {  
    List<List<String>> dvourozmernySeznam;  
  
    // postupne vytvareni  
    dvourozmernySeznam = new ArrayList<List<String>>();  
  
    for (int i = 0; i < 2; i++) {  
        dvourozmernySeznam.add(new ArrayList<>());  
        List<String> pomSeznam = dvourozmernySeznam.get(i);  
        for (int j = 0; j < 10; j++) {  
            String pom = "" + (i + 1) + "-" + (j + 1); // vzor 1-1  
            pomSeznam.add(pom);  
        }  
    }  
}
```

```

        pomSeznam.add(pom);
    }

    // pruchod iteratorem
    for (List<String> pomSeznam : dvouozmernySeznam) {
        System.out.println("\nRadka:");
        for (String prvek : pomSeznam) {
            System.out.print(prvek + ", ");
        }
    }
}

```

vypíše:

```

Radka:
1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, 1-9, 1-10,
Radka:
2-1, 2-2, 2-3, 2-4, 2-5, 2-6, 2-7, 2-8, 2-9, 2-10,

```

- **mapa množin:** Map<String, Set<String>> mapaMnozin;

množina je použita proto, že by se přítelkyně neměly opakovat a implementace HashSet protože na jejich pořadí nezáleží

```

public static void mapaMnozinRetezcu() {

    Map<String, Set<String>> mapaMnozin;

    // pomocna data
    String[] muzi = {"Karel", "Standa", "Honza"};
    String[] pritelkyne = {"Jana", "Hana", "Dana", "Anna", "Zina", "Dita"};
    Random r = new Random(2);

    // postupne vytvareni
    mapaMnozin = new HashMap<>();

    for (int i = 0; i < muzi.length; i++) {
        Set<String> mnozinaPritelkyn = new HashSet<>();

        int pocetPritelkyn = r.nextInt(pritelkyne.length) + 1;
        while (mnozinaPritelkyn.size() < pocetPritelkyn) {
            int poradi = r.nextInt(pritelkyne.length);
            mnozinaPritelkyn.add(pritelkyne[poradi]);
        }
        mapaMnozin.put(muzi[i], mnozinaPritelkyn);
    }
    System.out.println();

    // pruchod for-each
    for (Map.Entry<String, Set<String>> prvekMapy : mapaMnozin.entrySet()) {
        System.out.println("Muz: " + prvekMapy.getKey() + " a pritelkyne:");
        for (String prvekMnoziny : prvekMapy.getValue()) {

```

```
        System.out.print(prvekMnoziny + ", ");
    }
    System.out.println();
}
}
```

vypíše:

```
Muz: Standa a pritelkyne:  
Dana,  
Muz: Honza a pritelkyne:  
Dana, Hana, Anna,  
Muz: Karel a pritelkyne:  
Jana, Zina, Dana, Hana, Anna,
```



# Kapitola 8. UML

## 8.1. Základní informace



- *Unified Modeling Language* je sjednocený modelovací grafický jazyk pro vizualizaci, specifikaci a navrhování systémů s podílem SW
  - UML je navržen tak, aby svými výrazovými prostředky pokrýval celý životní cyklus vývoje výsledného systému s podílem SW, tj. všechny disciplíny – sběr požadavků, analýzu, návrh až po údržbu
  - zjednodušeně – formát dokumentace (např.) pro analýzu je UML
    - ◆ formát dokumentace pro kód je Javadoc
  - UML je „programovací“ jazyk analytika nezávislý na existujících programovacích jazycích
    - ◆ jazyk vizuálního modelování – „vizuální deklarativní jazyk“ – „technické kreslení“ pro softwarové inženýry
- důvody rozšíření – je to **standard**, umožňuje **jednotně** zapsat proces vývoje tak, aby mu rozuměli analytici, programátoři i zákazníci a pokrývá **celý životní cyklus** SW
  - ALE! Použití UML automaticky nezajišťuje projektu úspěch!
  - neobsahuje metodiku, jak analyzovat, specifikovat či navrhovat programové systémy
    - ◆ toto je klíčová odlišnost od předchozích podobných jazyků, které byly vždy nějak s metodikou svázané
- UML podporuje **objektově orientovaný přístup** k analýze, návrhu a popisu programových systémů
  - výhodný pro modelování objektově orientovaných systémů
    - ◆ Java si s UML „bezvadně“ rozumí a výrazně podporuje (někdy 1:1) některé modely
  - lze jím popsat i obchodní procesy a jiné nesoftwarové systémy
  - analýza v UML se dá použít i pro procedurální programování nebo pro návrh databází, ale není tam tak jednoznačný přechod do designu (do kódu)
  - UML může nahradit např. ERA modely z databází, vývojové diagramy, jazyky pro popis stavů apod.
- UML je de facto standard v SW průmyslu

## ■ historický pohled

- začátek vývoje 1994 sjednocením dosud používaných postupů a metodik pro OOA (OO analýza)
- společnost Rational Software na základě spolupráce tří hlavních metodik objektově orientované analýzy a návrhu (Booch, Rumbaugh, Jacobson)
- první uvolněná verze červen 1996
- konsorcium jazyka UML (DEC, HP, MS, Oracle, ...)
- verze 1.1 jazyka vytvořená konsorciem je v roce 1998 po dvou revizích přijata OMG (*Object Management Group*)
- poslední verze je 2.5.1 z prosince 2017 ([www.omg.org/spec/UML](http://www.omg.org/spec/UML))
  - ◆ předchozí verze byla 2.5 z června 2015

## ■ základní princip

- založen na „stavebních blocích“, což jsou grafické prvky (elementy, vztahy, diagramy), jejichž podoba je striktně předepsána a musí (měla by) se dodržovat
  - ◆ protože se správná podoba diagramů obecně nedá vynutit (neexistuje něco jako „referenční překladač“), např. BlueJ používá mnoho zjednodušení
    - to je ovšem chyba, která snižuje porozumění!
    - existují však „silné“ nástroje (Enterprise Architect, Magic Draw, ...), které striktně dodržují konvence
- má prostředky, které umožňují popis všech prvků jakkoliv složitého systému na všech úrovních a ve všech doménách (ale ne každý vše potřebuje!)
- vytváří se model systému, který v případě:
  - ◆ diagramu případů užití říká CO má systém dělat
  - ◆ ostatních diagramů JAK se to uvnitř dělá / vypadá

## ■ vytváření UML diagramů

- pro práci v UML je nutná jen tužka a papír (žádný „překladač“)
- různé jednoduché podpůrné nástroje „pouze“ ulehčují zápis jednotlivých modelů, případně jejich změny
  - ◆ existuje velké množství nástrojů různých kvalit a možností
  - ◆ my budeme využívat UMLet ([www.umlet.com](http://www.umlet.com))
    - ten ale patří mezi „jednoduší“ nástroje, tj. nevynucuje správné použití UML
    - to bude v případě KIV/OOP vynucováno validátorem

## Poznámka

Pozor při stahování – Google tam nabízí jiné produkty.



- profesionální CASE nástroje (špička – nástroje Rational Rose) vedou napříč procesním rámcem, tj. spojují s metodikou

## 8.1.1. Různé diagramy pro různé fáze vývoje projektu

- klíčem k rozumné vzájemné komunikaci mezi zákazníkem a tvůrcem systému je **abstrakce** = kladení důrazu na momentálně důležité informace a absenci nedůležitých
  - to, co je důležité a nedůležité, se v jednotlivých fázích projektu diametrálně mění
  - je nutné postupně použít více různých jednoduchých pohledů (diagramů) na systém
- UML má jeden metamodel, který definuje až 14 diagramů, kterými může být popsán konkrétní systém
  - diagramy jsou složeny z elementů UML, představujících jednotlivé stavební bloky
  - každý diagram představuje jiný pohled na informační systém a nemusejí být všechny použity
  - modelováním, tj. samotným vytvářením rozličných diagramů, bychom měli vytvářit jeden tentýž model, ale z různých úhlů pohledů
- rozdělení diagramů
  - strukturální diagramy (*structural diagrams*)
    - ◆ diagram tříd (*class diagram*)
    - ◆ diagram komponent (*component diagram*)
    - ◆ *composite structure diagram*
    - ◆ diagram nasazení (*deployment diagram*)
    - ◆ diagram balíčků (*package diagram*)
    - ◆ diagram objektů (též diagram instancí) (*object diagram*)
    - ◆ diagram profilů (*profile diagram*)
  - diagramy chování (*behavior diagrams*)
    - ◆ diagram případů užití (*use case diagram*)
    - ◆ diagram aktivit (*activity diagram*)
    - ◆ stavový diagram (*state machine diagram*)
  - diagramy interakce (*interaction diagrams*) (jsou specializací diagramů chování)
    - ◆ sekvenční diagram (*sequence diagram*)
    - ◆ diagram komunikace (*communication diagram*)
    - ◆ diagram interakcí (*interaction overview diagram*)
    - ◆ diagram časování (*timing diagram*)

## Poznámka

Toto dělení je základní dělení a stojí na něm specifikace UML. Lze se ale setkat s dalšími možnými způsoby dělení. Například na statické (= strukturální) a dynamické (= chování).

### ■ využití diagramů

- naprosté minimum pro školní příklady – diagram tříd (*class diagram*)
- vyžadované minimum pro netriviální projekty – diagram případů užití (*use case diagram*) a diagram tříd
  - ◆ diagramy případů užití jsou základem architektury UML a jsou tak vstupenkou k využití silných stránek UML.
- všechny ostatní diagramy se používají dle potřeby

## 8.2. Společné části diagramů

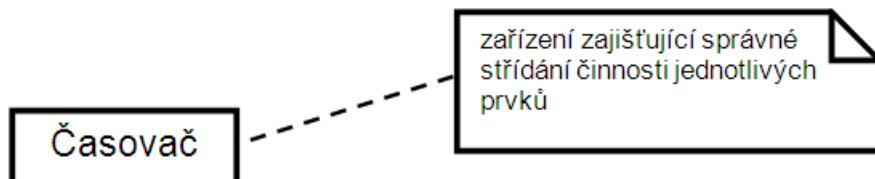
### ■ mohou se vyskytnout u jakémukoliv ze zmíněných diagramů nebo jejich částí

### Poznámka

Další příklady se budou vztahovat k modelu pračky prádla.

### 8.2.1. Poznámka (*note*)

- lze přiřadit k jakémukoliv elementu včetně šipky a spojovací čáry
- přidává vysvětlující text, čím více je **smysluplných** poznámek, tím lépe

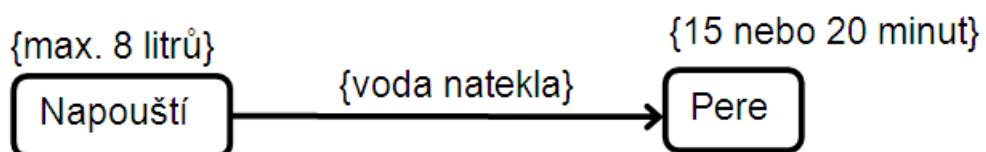


### 8.2.2. Stereotyp (*stereotype*)

- slouží k dodatečné specializaci elementů
- použijí se již existující prvky UML a vytvoří se z nich nové
  - jeden ze základních mechanismů rozširování UML
- používá se zápis <<stereotyp>> (francouzské uvozovky)
- některé stereotypy jsou již předefinovány
  - např. rozhraní, výčtový typ, abstraktní třída jsou speciální případy třídy

### 8.2.3. Omezení (*constraint*)

- druhý ze základních rozšiřovacích mechanismů UML
- lze přiřadit k jakémukoliv elementu
- je to často podmínka true/false, nebo výčty
- používá se zápis {omezení} (složené závorky)
- omezuje platnost některého stavebního bloku podmínkou ve formě definovaného výrazu

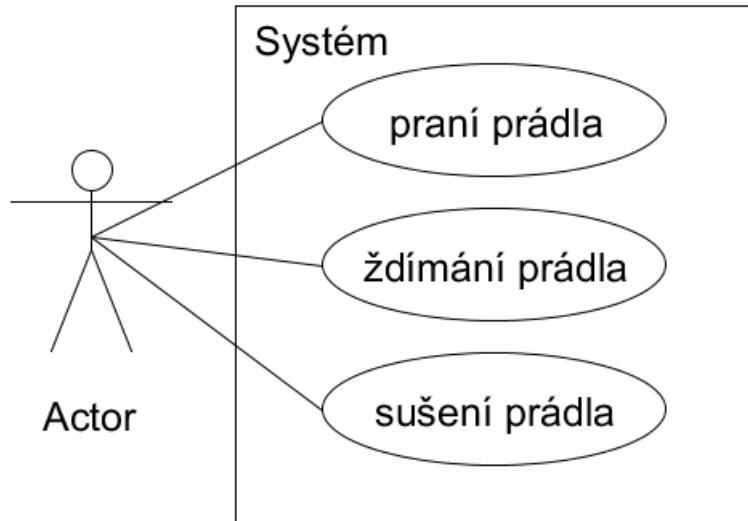


#### Poznámka

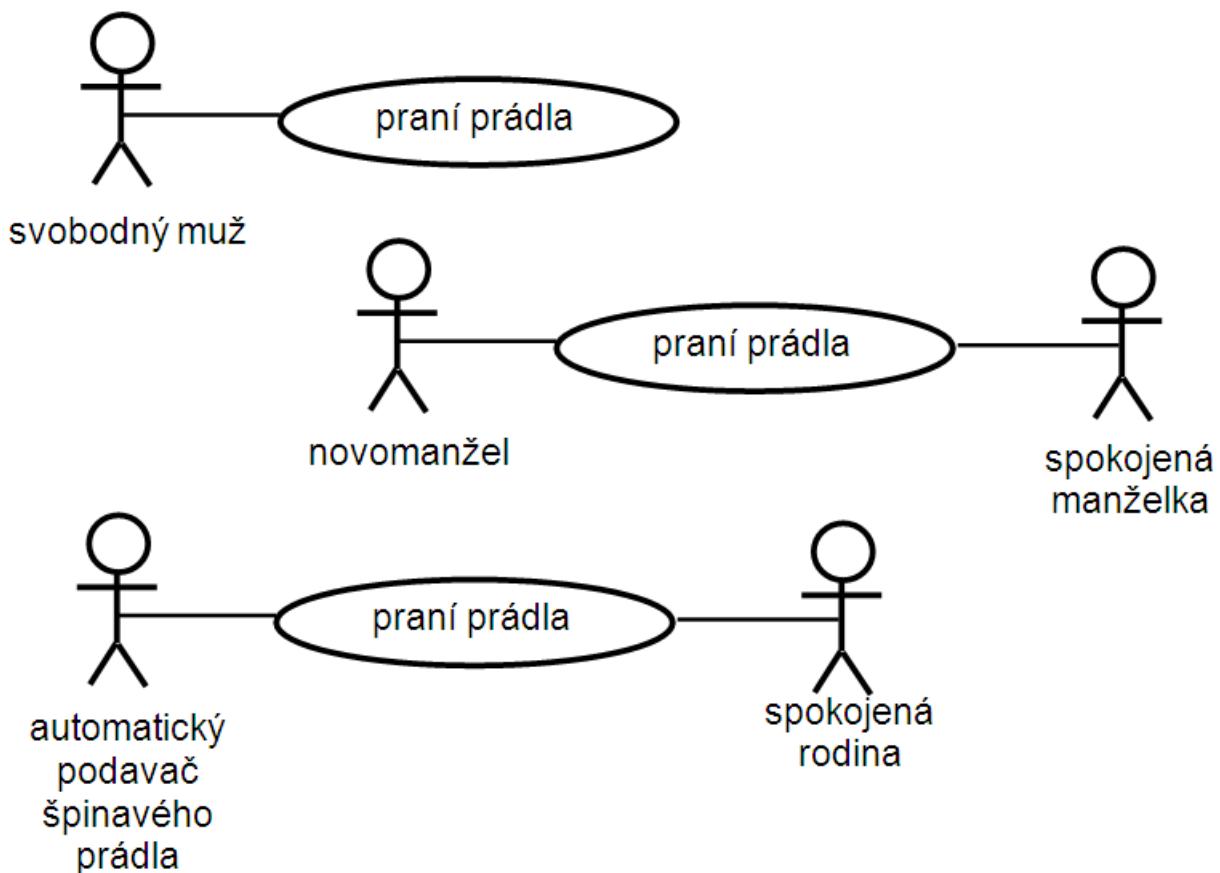
Toto je příklad na diagramu aktivit, který patří do skupiny diagramů chování.

### 8.3. Diagram případů užití

- slouží nejčastěji jako model požadavků na funkcionality s vytvořeného systému
  - je na pojmové úrovni, tj. srozumitelný všem, i netechnikům
- popisuje k čemu všemu se dá (informační) systém použít – mohou to být tisíce použití
  - zaznamenává vyčerpávajícím způsobem všechny funkce systému **z hlediska uživatele**
    - ◆ specifikuje chování systému (funkční požadavky), typické interakce mezi systémem a okolními aktory (uživatelé, jiné systémy, apod.)
- je-li hotový model případů užití, je popsán ucelený komplexní seznam funkčních požadavků na systém
  - víme vše, co má systém dělat a kdo (role) jej bude používat
  - lze provést kvalifikované odhady pracnosti a tím i nákladů
  - bez modelu případů užití se špatně hlídá celý projekt
- má dva základní elementy – „ovál“ pro případ užití (*use case*) a „panáčka“ pro **aktora** (účastníka, participant, aktéra, roli)



- vytvářený systém Pračka bude umožňovat uživateli pouze tři interakce (tj. případy užití) – praní, ždímnání a sušení
  - nic dalšího zákazník v době návrhu nepožaduje a proto nic dalšího nebude systém obsahovat
- aktor nemusí být člověk (např. jiný program, technologické zařízení, apod.), důležité je, že je vně systému (je externí), tj. neprogramuje se
  - aktor je ten, kdo systém obsluhuje a/nebo z něj má užitek
  - nalezením všech aktorů vymezíme hranice systému, tj. neprogramujeme pak něco, co nikdo nechtěl



- ukázka reálného případu užití

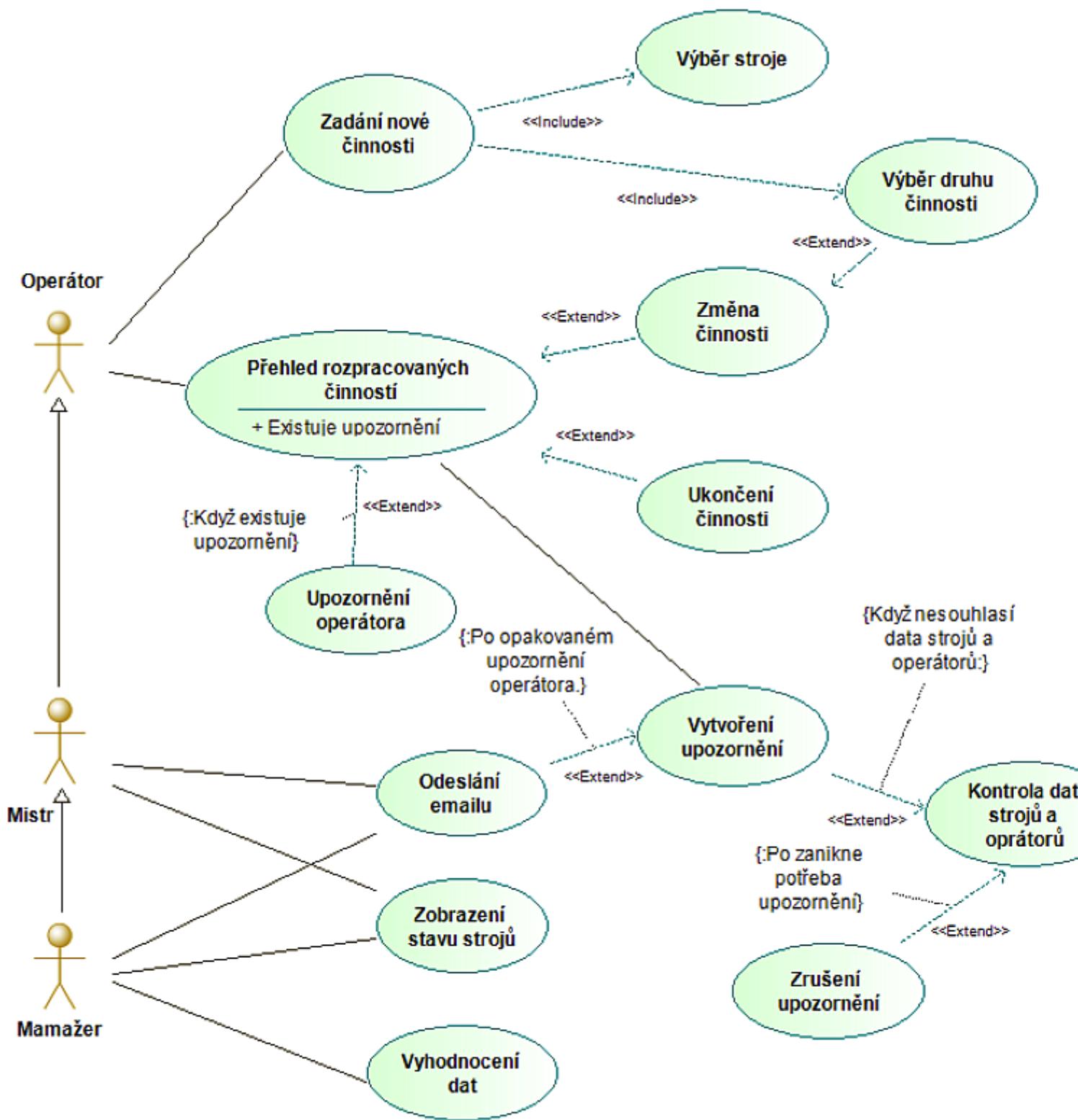


Diagram případů užití pro kontrolu produkce strojů a

## 8.4. Diagram tříd

- slouží k různým účelům, často pro analytický model vytvářeného systému ale i pro model návrhu (*design*)
  - druhé použití je dokumentační, kdy je již hotový systém velmi přesně popsán (na určité úrovni abstrakce)

♦ **toto bude použití v předmětu KIV/OOP**

- zaznamenává existenci tříd a vztahy mezi nimi (asociace, agregace, kompozice, dědění, implementace, ...)
  - analýza se věnuje zejména třídám aplikační logiky (méně už třídám GUI)
- BlueJ používá značně zjednodušený diagram tříd

### Poznámka

Další příklady se vztahují k příkladu využití polymorfismu – viz dále.

## 8.4.1. Způsoby zápisu třídy

- v nejjednodušší podobě je třída představována obdélníkem se svým jménem

SamoobsluznaRestaurace

- jedná-li se o abstraktní třídu, používá se kurziva a/nebo častěji stereotyp a kurziva

AHost

«abstract»  
AHost

- rozhraní, výčtové typy a JUnit testy jsou označeny příslušnými stereotypy

«interface»  
IHosteny

«enum»  
Fakulty

«test»  
RestauraceTest

- pokud chceme uvést, že třída náleží do nějakého balíku, použijeme názevbalíku::Třída

polymorfismus::Pokladni

- tento způsob se ale používá velmi zřídka, jen pokud chceme zdůraznit použití třídy z jiného balíku

- použití balíků

polymorfismus

«abstract»  
AHost

Pokladni

- u třídy lze uvést i atributy a metody, pak se obdélník třídy dělí vodorovně na tři části: jméno třídy, atributy, metody

| Třída                     |
|---------------------------|
| - privátní atributy: typ  |
| # protected atributy      |
| + public atributy         |
| + metoda(): návratový typ |

- atributy i metody mohou používat označení přístupových práv – pro private, # pro protected a + pro public
- typ atributu, typ formálního parametru nebo návratový typ metody se píše až za název oddělený :
  - ◆ zisk: int
  - ◆ zvysZisk(utrata: int): int
  - ◆ getSeznamHostu(): List<AHost>

- příklad kompletní informace

| SamoobsluznaRestaurace         |
|--------------------------------|
| -zisk: int                     |
| -seznamHostu: List<AHost>      |
| +getseznamHostu(): List<AHost> |
| +zvysZisk(utrata: int): int    |
| +nechHostyNajist(): void       |

- pokud atributy nebo metody chybějí (nebo nepovažujeme za nutné je uvádět – často u private), pak musí být příslušná část prázdná

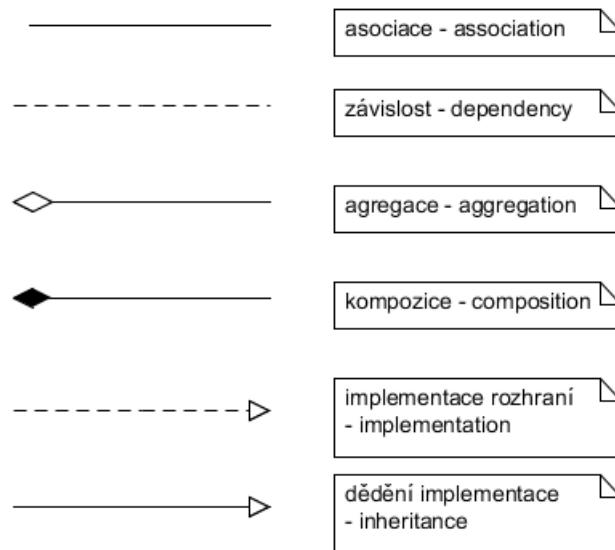
| Pokladni                                           |
|----------------------------------------------------|
|                                                    |
| +kasiruj(restaurace: SamoobsluznaRestaurace): void |

- jedná-li se o abstraktní metody, jsou zapsány kurzívou: konzumuje() a plati()

|                                                   |
|---------------------------------------------------|
| <b>«abstract»</b><br><b>AHost</b>                 |
| - jidlo: String                                   |
| - utrata: int                                     |
| + AHost()                                         |
| + objednavajidlo(jidlo: String, cena: int): AHost |
| + kdoJsem(): String                               |
| + konzumuje(): void                               |
| + plati(): int                                    |

## 8.4.2. Vazby tříd

- třídy jsou navzájem v mnoha vztazích, které se rozlišují typem spojovací čáry a případně i dodatečnými informacemi
- existuje šest základních typů vazeb tříd, lišících se (graficky) typem čáry a/nebo zakončením



- těchto šest typů je možné doplňovat dle potřeby šipkami a doplňujícími popisky – viz příklady u asociace

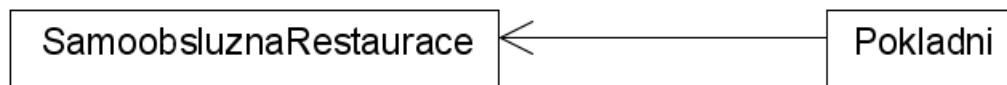
### ■ asociace (association)

- nejobecnější vazba – logická, analytická
  - ◆ použijeme v začátcích návrhu, kdy ještě neznáme implementační specifikaci pro třídy ve zjištěných vztazích
    - známe-li implementaci, jsou původní asociační vazby později (pro účely dokumentace systému – tj. případ KIV/OOP) nahrazovány přesnějšími vazbami
  - ◆ slouží pro popis vztahů – často se upřesňují role – viz dále
- vazba na začátku návrhu říká, že mezi dvěma třídami existuje blíže neurčený vztah, čili jedna třída **nějak** používá druhou (určuje „obvykle trvalou“ existenci spojení mezi instancemi)



- většinou je vztah pouze jednosměrný, což se naznačí jednoduchou (otevřenou) šipkou
  - ◆ šipka jde ve směru, odkud kam lze poslat zprávu – zde nějaká instance Pokladni může poslat zprávu do instance SamoobsluznaRestaurace

```
List<IHosteny> seznamHostu = restaurace.getSeznamHostu();
```

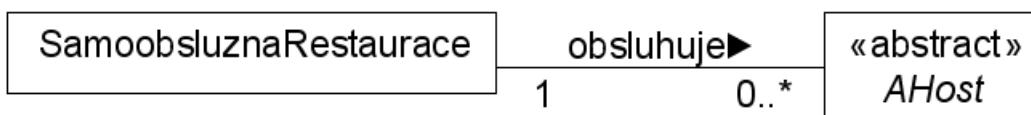


- ◆ také to znamená, že SamoobsluznaRestaurace vůbec nepotřebuje (neví o) existenci Pokladni
  - pokud by nebyla k dispozici třída Pokladni bylo by možné i přes to přeložit třídu SamoobsluznaRestaurace
- k asociaci lze dodat i slovní popis vzájemného vztahu (plná černá šipka určuje směr čtení), který může být i obousměrný



- k asociaci lze dodat také násobnost (multiplicita / kardinalita) vztahu

- ◆ určuje, kolik instancí třídy může (nebo musí) existovat



- ◆ SamoobsluznaRestaurace obsluhuje třídu AHost
- ◆ SamoobsluznaRestaurace bude existovat v právě jedné instanci, AHost bude existovat v žádné nebo ve více instancích

- možné multiplicity vztahu:

- ◆ 1 : 1 SamoobsluznaRestaurace obsluhuje právě jednoho AHost
- ◆ 1 : 0,1 SamoobsluznaRestaurace obsluhuje žádného nebo max. jednoho AHost
- ◆ 1 : \* SamoobsluznaRestaurace obsluhuje několik AHost
- ◆ 1 : 0..\* SamoobsluznaRestaurace obsluhuje žádného nebo několik AHost

- ◆ 1 : 3 Tříkolka má právě 3 Kola
- ◆ 1 : 1..5 Auto může přepravovat jednoho až pět Pasažér

## Poznámka

Multiplicita vztahu vede při implementaci většinou na použití nějakého kontejneru (možno i pole), neohraničená násobnost (\*) pak na kolekci.

## Poznámka

Všechny dále uváděné vazby už vyžadují znalost implementace. Jsou uváděny od nejvolnější vazby k nejtěsnější. Nahrazujeme jimi předchozí asociační vazby, pokud chceme diagram zpřesnit. **Konkrétní způsob nahrazování závisí na použité metodice** – dále budou uváděny nejčastější způsoby.

### ■ závislost (dependency)

- velmi „volná“ vazba – volná ve smyslu „nemusí být v diagramu vůbec uvedena“ ačkoliv existuje

## Poznámka

Pokud by byl UML diagram kvůli přemíře vazeb závislosti nepřehledný, pak je možné je (částečně) vynechat. Jeden se základních a dobrých principů modelování s UML je, že pokud něco není uvedeno v diagramech, neznamená to, že to nesmí být v modelu. Diagram by měl být především přehledný.

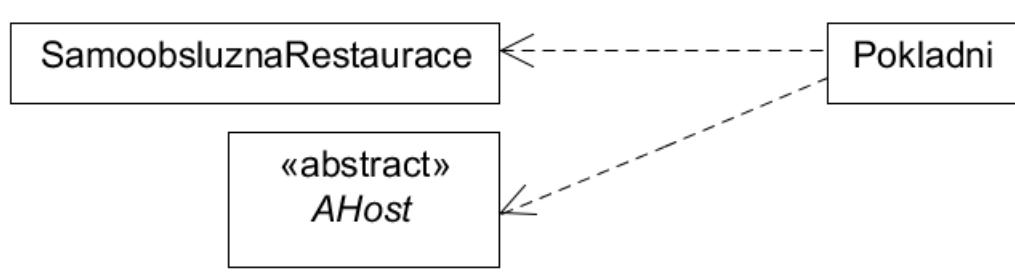
Na druhou stranu to ale ukazuje na špatnou analýzu (dekompozici) celé úlohy, protože se primárně snažíme o co nejmenší vzájemné provázání tříd.

- ◆ „analytik na ní většinou nepřijde, návrhář ji může specifikovat například s návrhovým vzorem a programátorovi tak nějak vyjde v kódu“
- ◆ nahrazuje původní (pokud vůbec existuje) asociační vazbu, je-li závislá třída ve třídě využívající použita jako:
  - typ formálního parametru v metodě
  - typ lokální proměnné (**Pozor:** ne atribut! – to je vazba agregace nebo kompozice)
  - adresát zasílání zprávy v metodě
  - návratový typ metody – viz též dále **Speciální případy vazeb**
  - „významný“ datový typ – viz dále **Speciální případy vazeb**
- ◆ typicky tedy využívající třída používá odkazy na již existující objekty závislé třídy (případně vytváří lokální objekty), aby zaslala zprávu závislé třídě
  - využívající třída může i vytvářet nové objekty závislé třídy, které poskytuje navenek
- v řešeném příkladu je:
  - ◆ objekt třídy SamoobsluznaRestaurace předán jako formální parametr metodě kasiruj() a tomuto objektu jsou poslány zprávy getSeznamHostu() a zvysZisk()

- ♦ typ `AHost` je typ lokální proměnné a je mu posílána zpráva `plati()`

```
public void kasiruj(SamoobsluznaRestaurace restaurace) {
    List<AHost> seznamHostu = restaurace.getSeznamHostu();

    for (AHost host : seznamHostu) {
        restaurace.zvysZisk(host.plati());
    }
}
```



## ■ agregace (aggregation)

- „těsná“ vazba – ve smyslu „nelze ji vynechat“
  - ♦ použijeme ji, pokud je závislá třída použita ve třídě využívající jako
    - typ atributu, kdy je tomuto atributu přiřazen odkaz na existující instanci, která typicky vzniká mimo využívající třídu
  - ♦ typicky tedy využívající třída trvale uchovává odkaz na existující objekty závislé třídy
    - a posílá jim zprávy z více svých metod
- důležité je, že závislá třída i její jednotlivé instance jsou (nebo mohou být) využívány více různými třídami, tj.:
  - ♦ může sloužit i více využívajícím třídám současně – využívající třída nevytváří a nevlastní instance závislé třídy
  - ♦ instance nevznikají současně s instance využívající třídy
    - mohou vzniknout dříve a pak je odkaz na ně předán nejčastěji do konstruktoru využívající třídy
    - mohou na ně existovat odkazy (agregace či asociace) z jiných využívajících tříd
  - ♦ instance většinou nezanikají se zánikem instance využívající třídy
- ve zdrojovém kódu má `SamoobsluznaRestaurace` atribut typu referenční proměnná na kolekci `<AHost>` (při multiplicitě `1:0..*`)
  - ♦ případně (ne zde) atribut typu referenční proměnná na typ `AHost` (při multiplicitě `1:1`)

```
public class SamoobsluznaRestaurace {
    private List<AHost> seznamHostu = new ArrayList<AHost>();
```

- ♦ ovšem jednotlivé instance `AHost` vznikají ve třídě `Hlavni` a do třídy `SamoobsluznaRestaurace` jsou předány jako skutečný parametr do kolekce `seznamHostu`

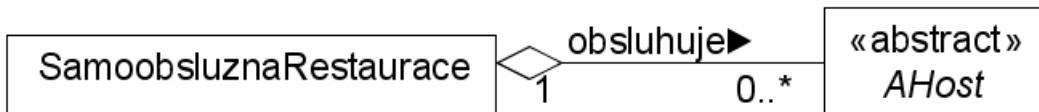
```

public class Hlavni {

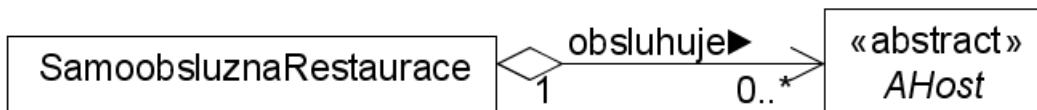
    public static void main(String[] args) {
        SamoobsluznaRestaurace restaurace = new SamoobsluznaRestaurace();

        List<AHost> seznamHostu = restaurace.getSeznamHostu();
        seznamHostu.add(new American().objednavajidlo("steak", 300));
    }
}

```



- se zánikem instance SamoobsluznaRestaurace nezaniká instance AHost
- opět je možné zdůraznit jednosměrnou závislost pomocí otevřené šipky
  - ◆ tím zdůrazňujeme, že AHost lze přeložit i bez existence SamoobsluznaRestaurace



## Poznámka

Mnemotechnická pomůcka – znak kosočtverce (diamant) je chápán jako „konektor“, tj. je umístěn u třídy využívající a třída závislá se do tohoto konektoru připojuje.

### ■ kompozice (composition) nebo též silná agregace

- těsná vazba
  - ◆ použijeme ji, pokud je závislá třída použita ve třídě využívající jako
    - typ atributu, kdy je tomuto atributu přiřazen odkaz instanci, která vzniká a je používána pouze ve využívající třídě
  - ◆ typicky tedy využívající třída trvale uchovává odkaz na **jí vytvořené a jí vlastněné** objekty závislé třídy a posílá jim zprávy z jedné nebo více svých metod
- závislá třída má vztah pouze k jedné využívající třídě, s ní vzniká (typicky v jejím konstruktoru) a zaniká
  - ◆ využívající (agregační) třída kompletně řídí celý životní cyklus agregované třídy
  - ◆ závislá třída **nemá samostatně smysl** – a takových tříd je velmi málo
- využívající třída má často stejně pojmenovaný atribut či kolekci atributů

```

abstract public class AHost implements IHosteny {
    private String jidlo;
    private int utrata;
    private Penezenka penezenka;
}

```

```

public AHost() {
    this.utrata = 0;
    this.penezenka = new Penezenka(1000);
}

```

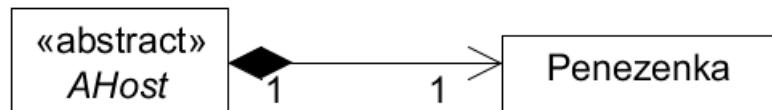


- AHost vlastní právě jednu Penezenka a každá Penezenka patří právě jednomu AHost
- neexistují žádné další (vnější mimo třídu AHost) reference na instanci Penezenka, tzn. nikdo jiný instanci Penezenka nepoužívá, protože mu není přístupná (nebo mu k ničemu není)

## Poznámka

Kompozice se vyskytují velmi zřídka. Musí pro ně být skutečně vážný důvod. Pokud se vyskytují, jsou nejčastěji realizovány pomocí **vnitřní třídy** – viz dále.

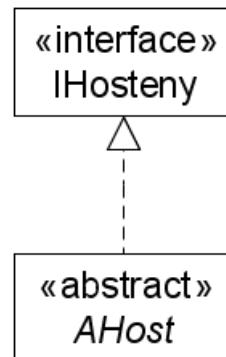
- chceme-li zdůraznit, že Penezenka je jednosměrně závislá na AHost, můžeme to zdůraznit otevřenou šipkou



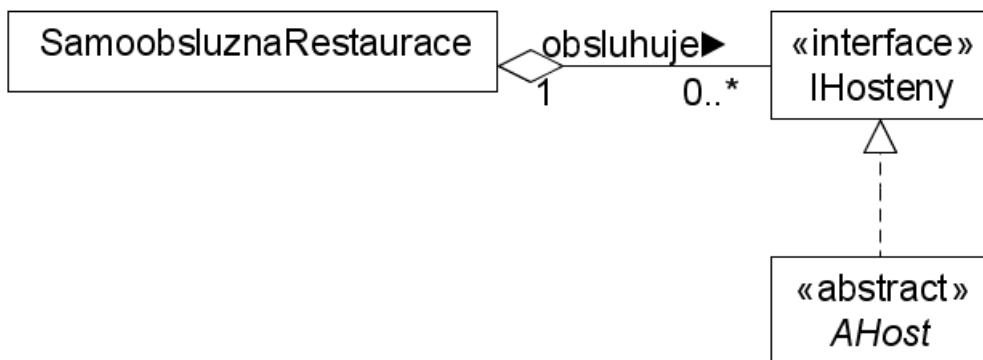
## ■ implementace rozhraní (*implementation*)

- velmi těsná vazba – narozdíl od předchozích vazeb, které byly reálně mezi instancemi, je tato vazba mezi třídami
  - ◆ použijeme ji, pokud třída nebo abstraktní třída implementuje rozhraní
    - v hlavičce třídy je uvedeno `implements`
- zde máme dvě možnosti, jak implementaci rozhraní zakreslit
  1. implementace několika málo rozhraní je klíčová pro celý program
    - použijeme přerušovanou čáru s uzavřenou šipkou
      - ◆ pokud je to možné, snažíme se, aby šipka směřovala vzhůru, tzn. umístíme rozhraní v diagramu nahoru

```
abstract public class AHost implements IHosteny {
```

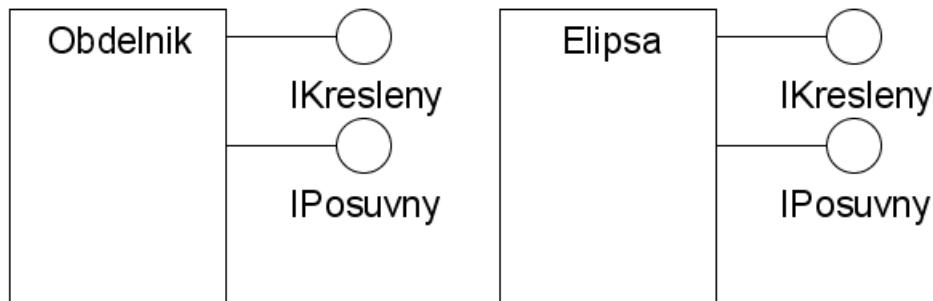


- pokud je rozhraní navíc využíváno jako typ v dalších třídách, např. SamoobsluznaRestaurace, pak se zakresluje jiným typem vazby



2. množství tříd implementuje množství rozhraní a vzájemné vazby by byly nepřehledné

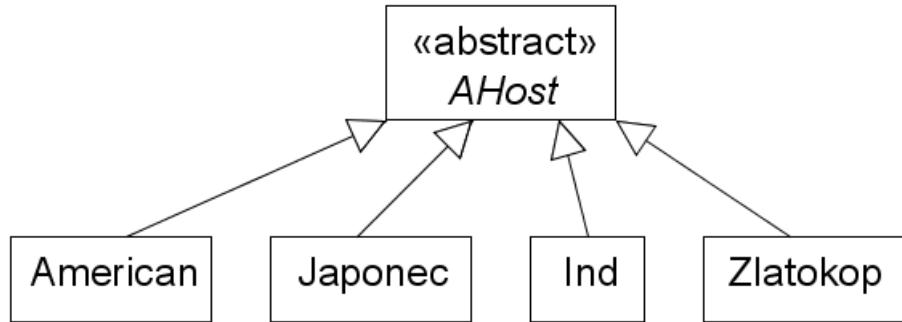
- demonstrujeme tím schopnosti objektů, které nejsou klíčové pro program



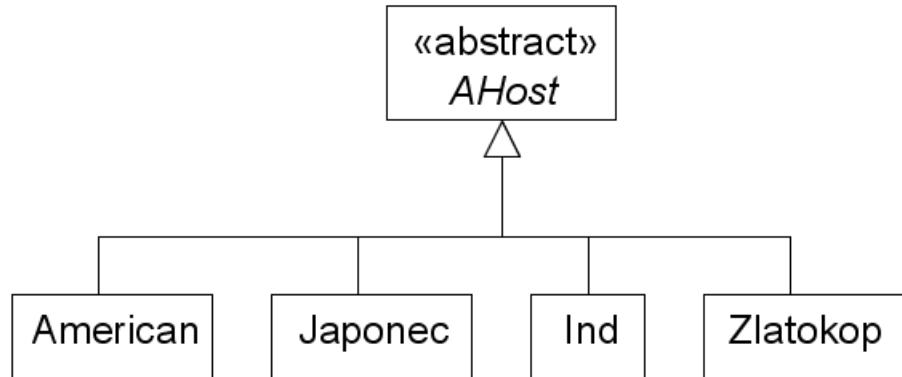
## ■ dědění (*inheritance*)

- velmi těsná vazba
  - ◆ použijeme ji, pokud třída dědí implementaci od jiné třídy nebo rozhraní dědí od jiného rozhraní
    - v hlavičce třídy nebo rozhraní je uvedeno extends
- používá se plná čára s prázdnou šípkou směrem od potomka k předkovi
  - ◆ pokud je to možné, snažíme se, aby šípka směřovala vzhůru, tzn. umístíme předka v diagramu nahoru

```
public class American extends AHost {
```

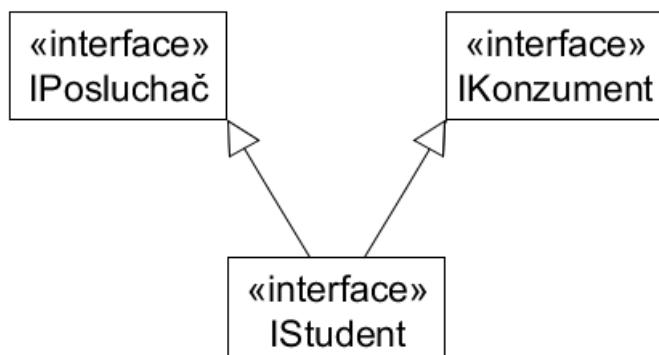


- v případě násobných dědičností, lze pro zvýšení přehlednosti použít



- tato vazba se používá i v případě dědění rozhraní

```
public interface IStudent extends IPosluchač, IKonzument {
```

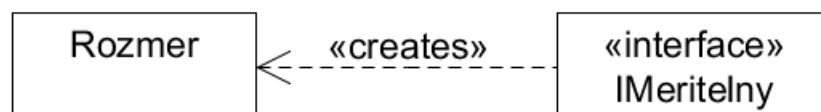


### 8.4.3. Speciální případy vazeb

- třída nebo rozhraní vytváří svojí metodou instanci (nebo instance) jiné třídy, které vrací jako návratový typ

- použijeme závislost se stereotypem <<creates>>

```
public interface IMeritelny {
    public Rozmer getRozmer();
```



- tato vazba má přednost před „normální“ závislostí (tj. lokální proměnná, formální parametr, zasílání zprávy)

## ■ třída využívá rozhraní jako formální parametr či lokální typ

- použijeme vztah závislost

```
public class Zvyraznovac {
    public Obdelnik zvyrazniPozadi(IZvyrazneny objekt, Barva barva) {
```



## ■ metoda využívá významným způsobem nějaký datový typ

- „významný způsob“ znamená, že z hlediska pochopení logiky programu je tato informace (tj. vztah k jiné třídě) podstatná

- použijeme vztah závislost

```
public final class Seznamka {
    public int realizujSchuzky(Pozice spolecnyOdchod, int dobaCekani) {
        ...
        if ((muz instanceof Superman) && (zena instanceof Superwoman)) {
```



- v příkladu není třída `Superman` ani návratovou hodnotou, ani formálním parametrem, ani lokálním datovým typem

- ♦ pro pochopení smyslu celého programu (dávají se dohromady páry superlidí – podrobně viz zadání DÚ-08) je ale klíčová

## ■ zakreslujeme pouze přímé vztahy

- třída `Rozmer` je využívána

- ♦ přímo – rozhraním `IMeritelny` (vytváří jej) a třídou `Zvyraznovac` (používá jej jako lokální proměnnou)

– vztahy je nutno zakreslit

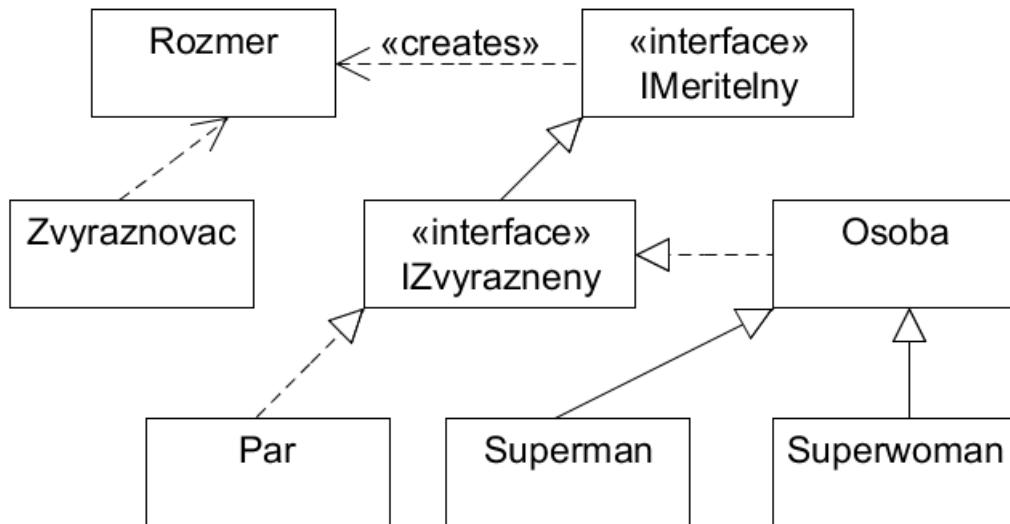
- ♦ nepřímo – třídami `Osoba`, `Par`, `Superman`, `Superwoman`

– tyto třídy používají `Rozmer` pouze v defaultní metodě `public Rozmer getRozmer();` z rozhraní `IMeritelny` a to je zděděno rozhraním `IZvyrazneny`

– vztah ke třídě Rozmer je již zakreslen nepřímo z:

- Osoba – implementační vazbou k IZvyrazneny
- Par – implementační vazbou k IZvyrazneny
- Superman, Superwoman – vazbou dědičnosti k Osoba

– nepřímé vztahy nekreslíme – to už vyžaduje dobrou znalost implementace, ale významně se zjednoduší počet vazeb a tím přehlednost diagramu



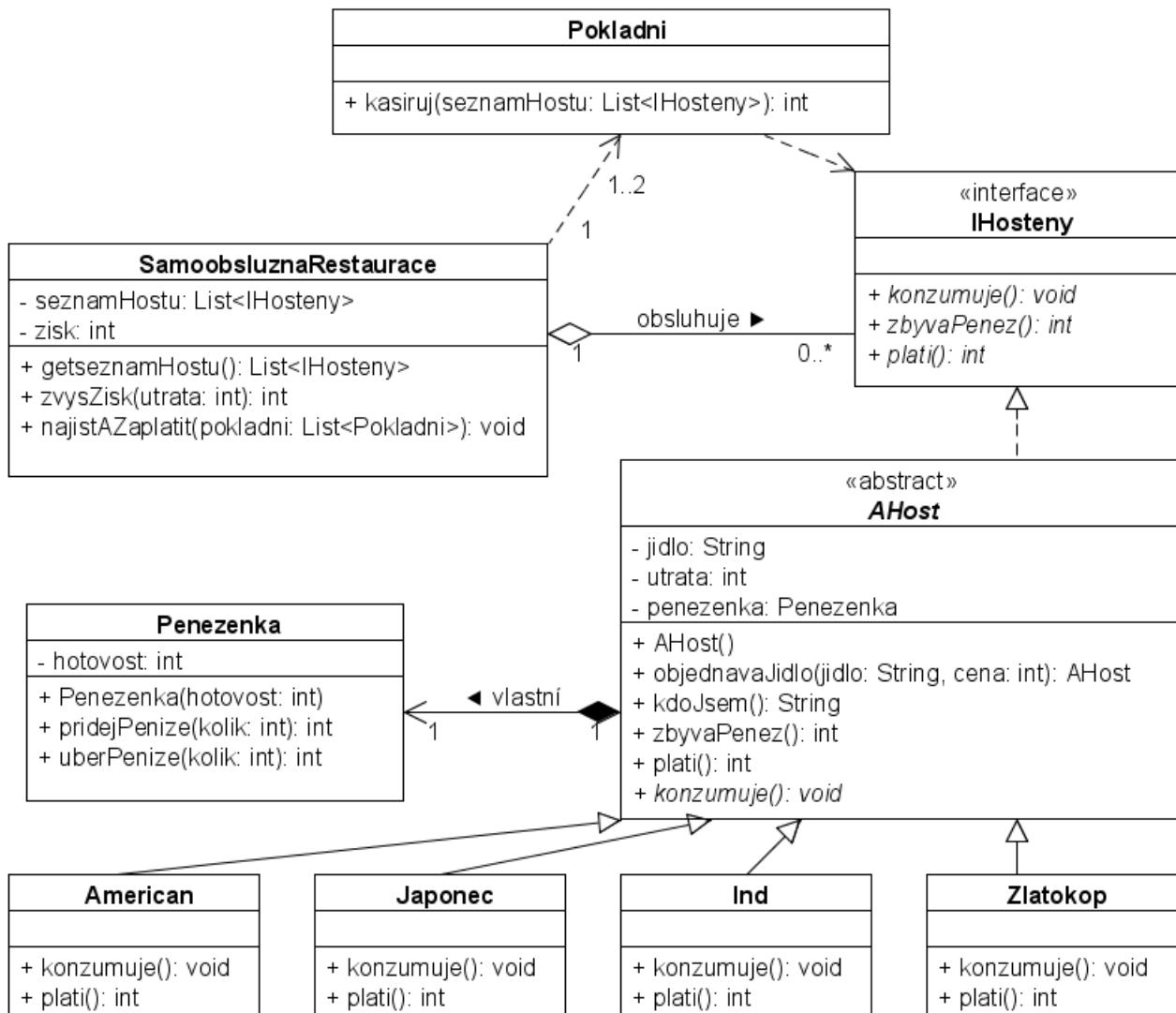
♦ zjednodušeně řečeno – pokud je vztah závislosti ve stejném směru, jako vztahy implementace a/nebo dědičnosti, nekreslíme závislost

– dojdeme-li po šipkách vyšších vazeb (Superwoman->Osoba->IZvyrazneny->IMeritelny->Rozmer) tam, kam by vedla přímá vazba závislosti (Superwoman->Rozmer), nemusíme kreslit závislost

## 8.5. Příklad na různé vztahy mezi třídami

■ příklad je modifikací příkladu uvedeného dále v polymorfismu

- snahou návrhu je, aby třídy SamoobsluznaRestaurace a Pokladni nebyly pokud možno provázaný těsnými vazbami
  - ◆ provázání bude provedeno v hlavní třídě



## ■ rozhraní IHosteny

```

public interface IHosteny {

    public void konzumuje();

    public int plati();

    public String zbyvaPenez();
}

```

## ■ třída AHost

```

abstract public class AHost implements IHosteny {
    private String jidlo;
    private int utrata;
    private Penezenka penezenka;

    public AHost() {
        this.utrata = 0;
        this.penezenka = new Penezenka(1000);
    }
}

```

```

public AHost objednavaJidlo(String jidlo, int cena) {
    this.jidlo = jidlo;
    this.utrata += cena;
    return this;
}

public String getJidlo() {
    return jidlo;
}

public int getUtrata() {
    return utrata;
}

abstract public void konzumuje();

public int plati() {
    penezenka.uberPenize(utrata);
    return utrata;
}

public String zbyvaPenez() {
    return kdoJsem() + ":" + penezenka.pridejPenize(0);
}

public String kdoJsem() {
    return this.getClass().getSimpleName();
}

@Override
public String toString() {
    return kdoJsem() + " ji " + jidlo + " ";
}
}

```

## ■ třída American

```

public class American extends AHost {

    public void konzumuje() {
        System.out.println(toString() + "plastikovym priborem");
    }

    public int plati() {
        super.plati();
        System.out.println(kdoJsem() + " plati utratu " + getUtrata()
            + " kreditni kartou");
        return getUtrata();
    }
}

```

■ podobně třídy Japonec, Ind a Zlatokop

■ třída SamoobsluznaRestaurace

```
public class SamoobsluznaRestaurace {  
    private List<IHosteny> seznamHostu = new ArrayList<>();  
    private int zisk = 0;  
  
    public List<IHosteny> getseznamHostu() {  
        return seznamHostu;  
    }  
  
    public int getZisk() {  
        return zisk;  
    }  
  
    public void najistAZaplatit(List<Pokladni> pokladni) {  
        for (IHosteny host : seznamHostu) {  
            host.konzumuje();  
        }  
        if (pokladni.size() == 1) {  
            zisk += pokladni.get(0).kasiruj(seznamHostu);  
        }  
        else {  
            zisk += pokladni.get(0).kasiruj(pоловинаHostu(true));  
            zisk += pokladni.get(1).kasiruj(pоловинаHostu(false));  
        }  
    }  
  
    private List<IHosteny> половинаHostu(boolean dolni) {  
        if (dolni) {  
            return seznamHostu.subList(0, seznamHostu.size() / 2);  
        }  
        else {  
            return seznamHostu.subList(seznamHostu.size() / 2, seznamHostu.size());  
        }  
    }  
}
```

■ třída Pokladni

```
public class Pokladni {  
    public int kasiruj(List<IHosteny> seznamHostu) {  
        int suma = 0;  
        for (IHosteny host : seznamHostu) {  
            suma += host.plati();  
        }  
        return suma;  
    }  
}
```

■ třída Hlavni

```

public class Hlavní {

    public static void main(String[] args) {
        SamoobsluznaRestaurace restaurace = new SamoobsluznaRestaurace();

        List<IHosteny> seznamHostu = restaurace.getseznamHostu();
        seznamHostu.add(new American().objednavajÍdlo("steak", 300));
        seznamHostu.add(new Japonec().objednavajÍdlo("susi", 200));
        seznamHostu.add(new Ind().objednavajÍdlo("capati", 80));
        seznamHostu.add(new Zlatokop().objednavajÍdlo("fazole", 50));

        List<Pokladni> seznamPokladnich = new ArrayList<>();
        seznamPokladnich.add(new Pokladni());
        seznamPokladnich.add(new Pokladni());
        restaurace.najistAZaplatit(seznamPokladnich);

        System.out.println("Celkova utrata: " + restaurace.getZisk());
        for (IHosteny host : seznamHostu) {
            System.out.println(host.zbyvaPenez());
        }
    }
}

```

která vypíše:

```

American ji steak plastikovym priborem
Japonec ji susi hulkami
Ind ji capati pravou rukou
Zlatokop ji fazole lzici
American plati utratu 300 kreditni kartou
Japonec plati utratu 200 sekem
Ind plati utratu 80 hotovosti
Zlatokop plati utratu 50 valouny
Celkova utrata: 630
American: 700
Japonec: 800
Ind: 920
Zlatokop: 950

```

## 8.6. Jak připravit diagram tříd snadno a rychle

- jedná se o rozsáhlejší aplikaci, která se skládá z deseti tříd a jednoho rozhraní

### 8.6.1. Zdrojové kódy

- z úsporných i názorných důvodů zde budou uváděny jen podstatné části kódů
  - třídy jsou seřazeny podle abecedy
  - úplné zdrojové kódy podobného příkladu naleznete v „9.1.2. Využití polymorfismu pomocí rozhraní“

```

abstract public class AHost implements IHosteny {
...
}

public class American extends AHost {
...
}

public interface IHosteny {
...
}

public class Ind extends AHost {
...
}

public class Japonec extends AHost {
...
}

public class Majitel implements IHosteny {
...
}

public class Pes implements IHosteny {
...
}

public class Pokladni {
    public void kasiruj(SamoobsluznaRestaurace restaurace) {
        List<AHost> seznamHostu = restaurace.getSeznamPlaticichHostu();

        for (AHost host : seznamHostu) {
            restaurace.zvysZisk(host.plati());
        }
    }
}

public class SamoobsluznaRestaurace {
    private List<IHosteny> seznamHostu = new ArrayList<>();
...
    public List<AHost> getSeznamPlaticichHostu() {
        List<AHost> pom = new ArrayList<>();
        for (IHosteny jedlik : seznamHostu) {
            if (jedlik instanceof AHost) {
                pom.add((AHost) jedlik);
            }
        }
        return pom;
    }
...
}

```

```

public class Vymahac implements IHosteny {
    ...
    public void berePenize(SamoobsluznaRestaurace restaurace) {
        int vypalne = 500;
        System.out.println("Vymaha penize: " + vypalne);
        restaurace.zvysZisk(-vypalne);
    }
}

public class Zlatokop extends AHost {
    ...
}

```

- je vidět, že k sestavení diagramu tříd stačí překvapivě málo informací ze zdrojového kódu

- většinou jsou to jen hlavičky tříd nebo rozhraní (nebo výčtových typů – zde není)
- dále atributy, které jsou referencemi na jiné třídy

```
private List<IHosteny> seznamHostu = new ArrayList<>();
```

- metody, u kterých se vyskytují reference na jiné třídy, jako:

- ◆ návratová hodnota, tj. instance j vytvářena

```
public List<AHost> getSeznamPlaticichHostu() {
```

- ◆ formální parametry

```
public void berePenize(SamoobsluznaRestaurace restaurace) {
```

- ◆ lokální proměnné

```
List<AHost> pom = new ArrayList<>();
```

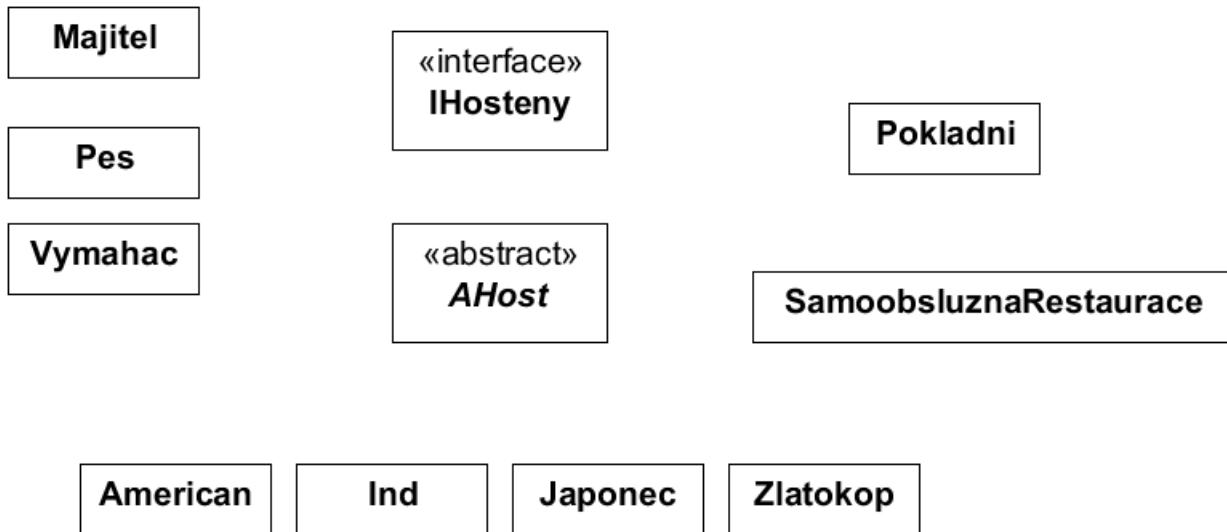
- vůbec nás nezajímají primitivní datové typy nebo reference na třídy z:

- Java Core API – považují se za obecně známé, tj. „samozřejmé“
  - ◆ jejich kreslení by snižovalo přehlednost diagramu a nepřinášelo by (jako „samozřejmé“) novou informaci
- použité externí knihovny (zde se nevyskytují, ale např. Obdelnik, Barva apod.)

## 8.6.2. Nakreslení tříd, rozhraní a výčtových typů

- je to mechanická záležitost – co .java soubor, to jeden element
- k názvům hned doplňujeme případné příslušné stereotypy
  - <<abstract>>

- <<interface>>
  - <<enum>>
- snažíme se o počáteční vhodné rozmístění objektů – nelze vždy dosáhnout na první pokus
- třída implementující rozhraní (`implements`) by měla být pod tímto rozhraním
  - třída, která vzniká děděním (`extends`) by měla být pod touto třídou
  - třídy, které mají něco společného, by měly být u sebe
    - ◆ společný předek – American, Ind, Japonec, Zlatokop
    - ◆ společná implementace rozhraní – Majitel, Pes, Vymahac
    - ◆ odkazují na sebe – Pokladni, SamoobsluznaRestaurace
- pokud to lze, objekty stejného typu by měly být stejně velké (široké)
- zde výjimka SamoobsluznaRestaurace, která má dlouhý název

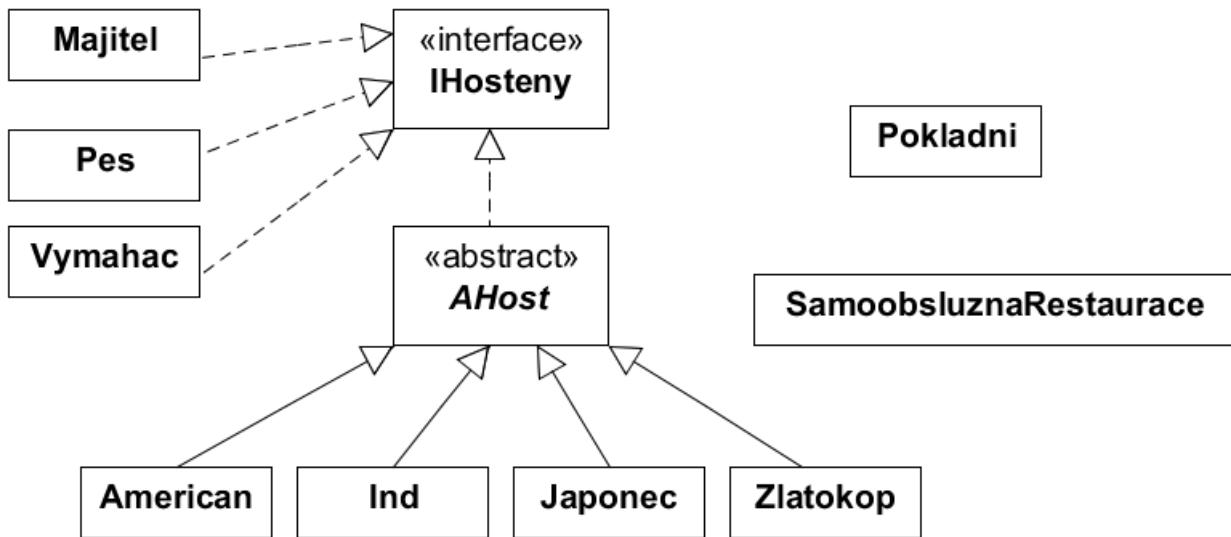


### 8.6.3. Nakreslení nejsilnějších vazeb

- nejprve kreslíme vazby:

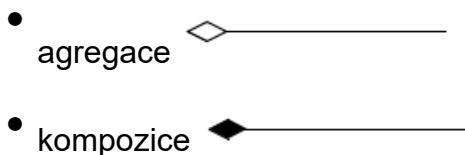
- dědičnosti →
- implementace →

- poznáme je z hlaviček tříd



## 8.6.4. Nakreslení vazeb agregací

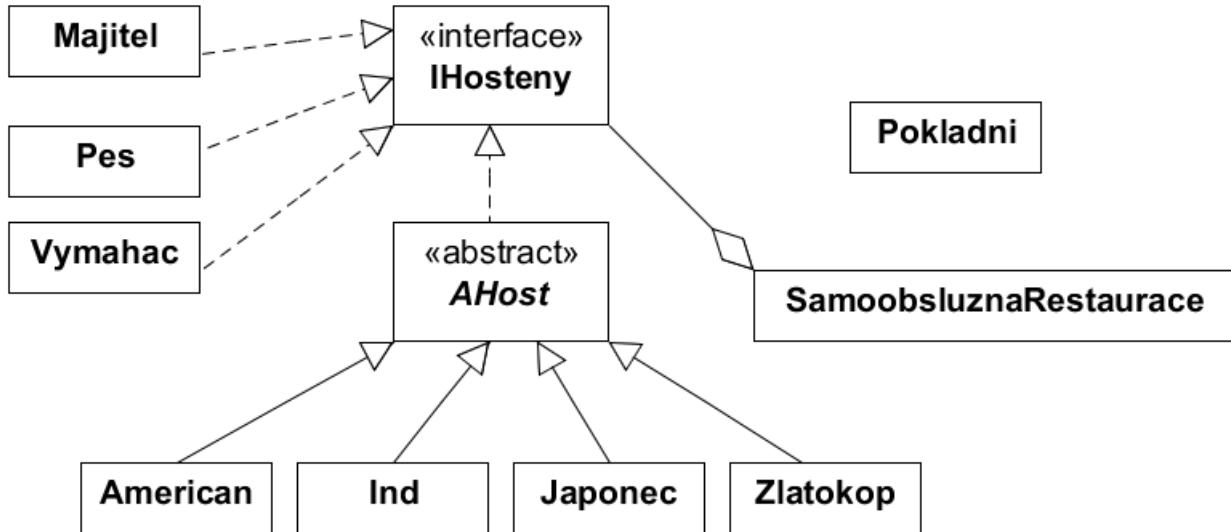
- v dalším kroku kreslíme vazby:



- poznáme je z atributů

```
private List<IHosteny> seznamHostu = new ArrayList<>();
```

- „diamant“ je „konektor“, do kterého se agregovaný objekt připojuje



## 8.6.5. Nakreslení vazeb závislostí

- jako poslední kreslíme vazby:

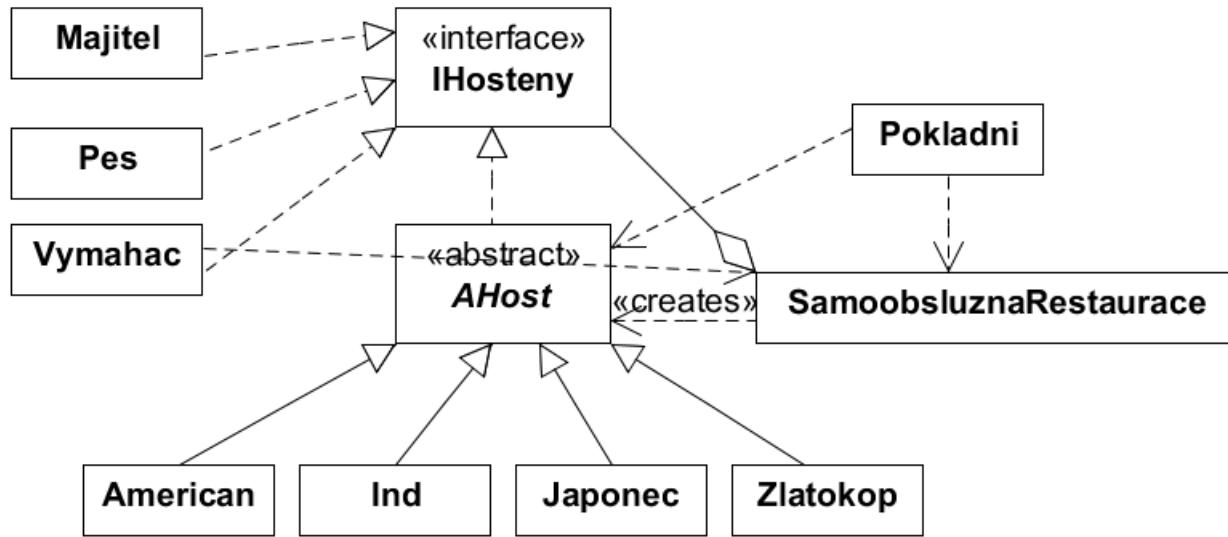
- závislostí s daným směrem ----->
- poznáme je z metod a jejich:
  - návratové hodnoty, pak ještě čáru doplňujeme stereotypem <<creates>>
  - formálních parametrů
  - lokálních proměnných
  - významných datových typů (zde není)
  - zasílání zpráv

## Poznámka

Návratová hodnota má přednost před formálním parametrem a/nebo lokální proměnnou či zasílání zpráv. To znamená, že závislosti nekreslíme dvakrát mezi týmiž dvěma objekty.

- závislost nekreslíme ani tam, kde již mezi třídami existuje některá ze silnějších vazeb
  - buď přímo
  - nebo nepřímo, přes několik tříd (viz „8.4.3. Speciální případy vazeb“)

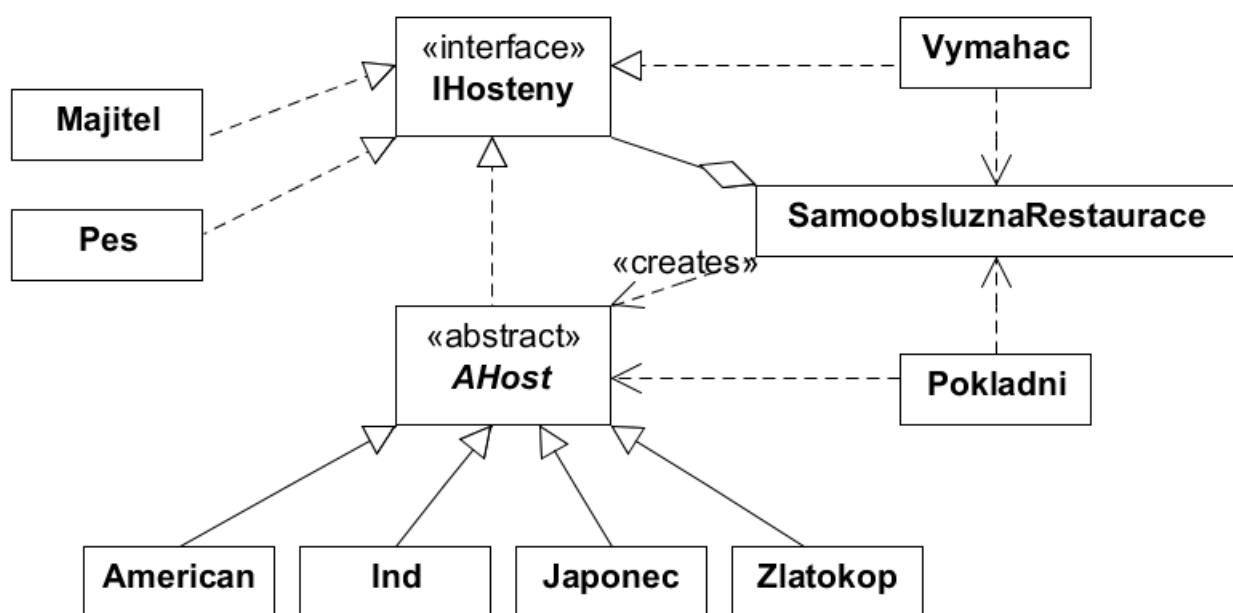
(zde není)



## 8.6.6. Přeskupení objektů

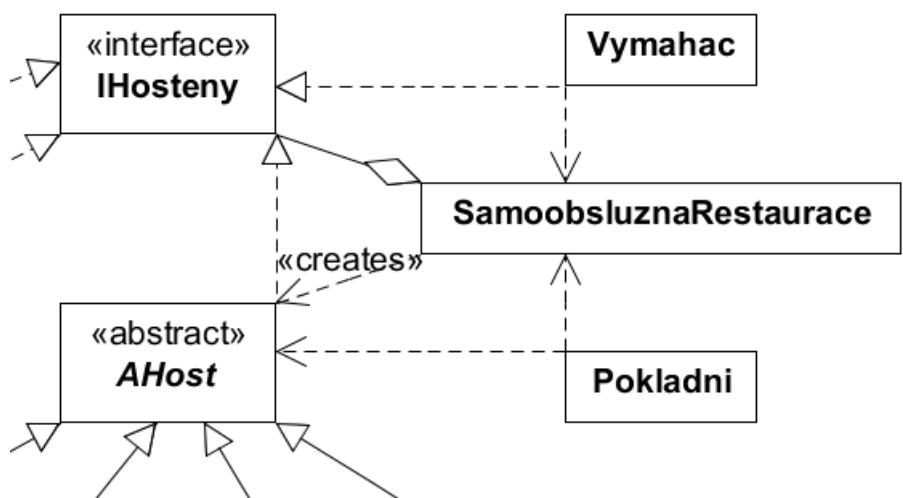
- po předchozím kroku máme nakresleny všechny objekty a všechny vazby mezi nimi
- následuje krok, kdy přeskupujeme objekty na finální místa
- snahou je, aby (prioritně):
  1. se nekřížily vazby

2. nadřízení (rozhraní, rodičovská třída) zůstaly fyzicky nad podřízenými (implementující třída, třída potomka)
3. aby diagram vypadal graficky hezky
  - symetrie
  - vazby pokud možno svislé a vodorovné
4. „společné“ třídy zůstaly u sebe



## Varování

Svislé a vodorovné čáry **nesmí** jít z rohů objektů.



## 8.6.7. Doplnění multiplicit

- u vazeb agregací (kompozic) doplňujeme multiplicity
  - zapisujeme vždy jen nejsilnější multiplicitu, např.:

```

public class SamoobsluznaRestaurace {
    private IHosteny praveObsluhovanyHost;
    private List<IHosteny> seznamHostu = new ArrayList<>();

```

♦ v tomto případě by měla být multiplicita

- 1:1 na praveObsluhovanyHost
- 1:0..\* na IHosteny

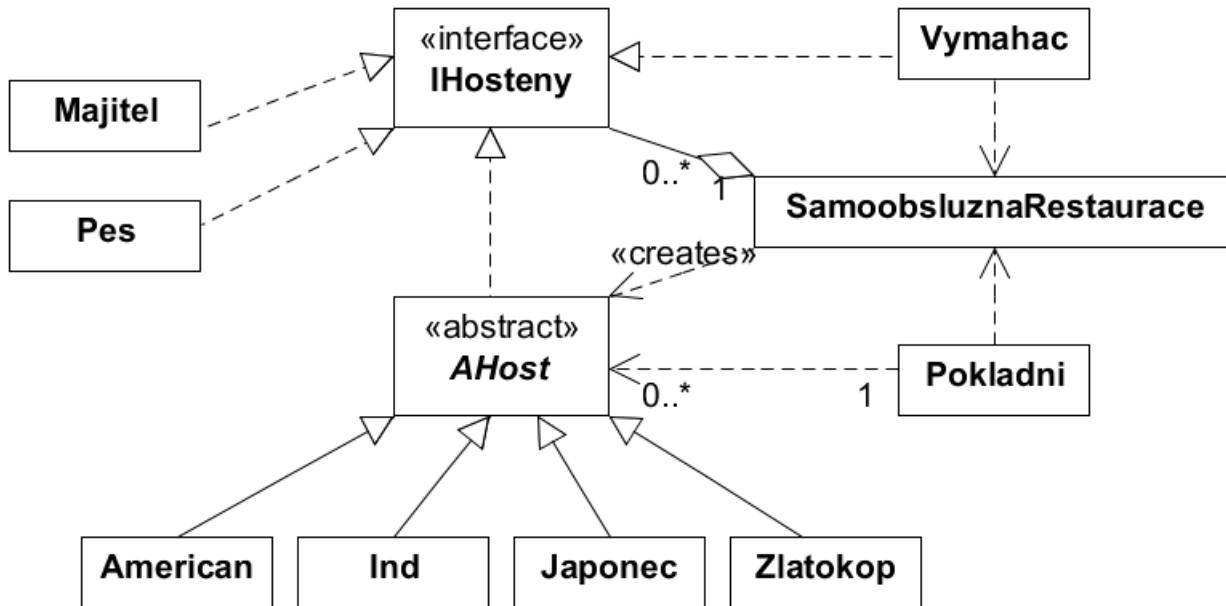
♦ zapisovali bychom ale jen „silnější“ multiplicitu 1:0..\* na IHosteny

■ tam, kde to zvýší srozumitelnost, doplňujeme multiplicity i u vazeb závislosti

zde Pokladni : AHost

- multiplicity 1:1 se téměř nikdy neuvádějí

zde Vymahac : SamoobsluznaRestaurace a Pokladni : SamoobsluznaRestaurace



# Kapitola 9. Polymorfismus, interní datové typy

## 9.1. Polymorfismus

- princip polymorfismu („vícečvarost“) byl zmíněn již při dědění
  - v okamžiku běhu kódu se zvolí metoda konkrétní instance, bez ohledu na to, jakého typu je referenční proměnná, pomocí které zasíláme instanci zprávu
  - toto chování je umožněno díky přetypování referenčních typů, překrývání metod v potomcích a pozdní vazbě – všechny pojmy viz dříve v dědění
- aby měl polymorfismus reálný smysl, musí (nejčastěji) existovat alespoň jedna další třída, která bude pracovat s větším počtem instancí různých potomků jedné třídy (nebo různých implementací jednoho rozhraní) a volat některé z metod, které jsou přetíženy v potomcích
  - typické použití je třída využívající kolekce
- v kolekci jsou uloženy různé instance potomků a v případě zpracování (typicky pomocí *For-Each*) nemusíme vůbec rozlišovat typ konkrétního potomka
  - zpracování je extrémně jednoduché

```
for (Předek instance : kolekce) {  
    instance.vykonejČinnost();  
}
```

- je to sjednocení práce s různými typy „příbuzných“ objektů
- pokud bychom v budoucnu rozšiřovali počet typů potomků, nebylo by vůbec nutné měnit výše uvedený kód jejich zpracování
  - ◆ stačí pouze, aby přidávaný potomek vhodně implementoval metodu `vykonejČinnost()`

- v roli Předek může být:

- rodičovská třída
- abstraktní třída
- rozhraní

### Poznámka

Dále uvedené grafické zápisy struktury tříd – tzv. **diagram tříd** – bude zapisován v UML notaci, která bude detailně vysvětlena později. V detailech se liší od grafického zápisu používaného v BlueJ.

### 9.1.1. Využití polymorfismu pomocí abstraktní třídy

- příklad bude ukazovat hosty v restauraci, kteří jedí různým způsobem a platí různými platidly
  - příklad nemá praktický význam, slouží jako ukázka polymorfismu

- UML diagram, ze kterého je patrné, že SamoobsluznaRestaurace nezná třídy American, Japonec, Ind a Zlatokop

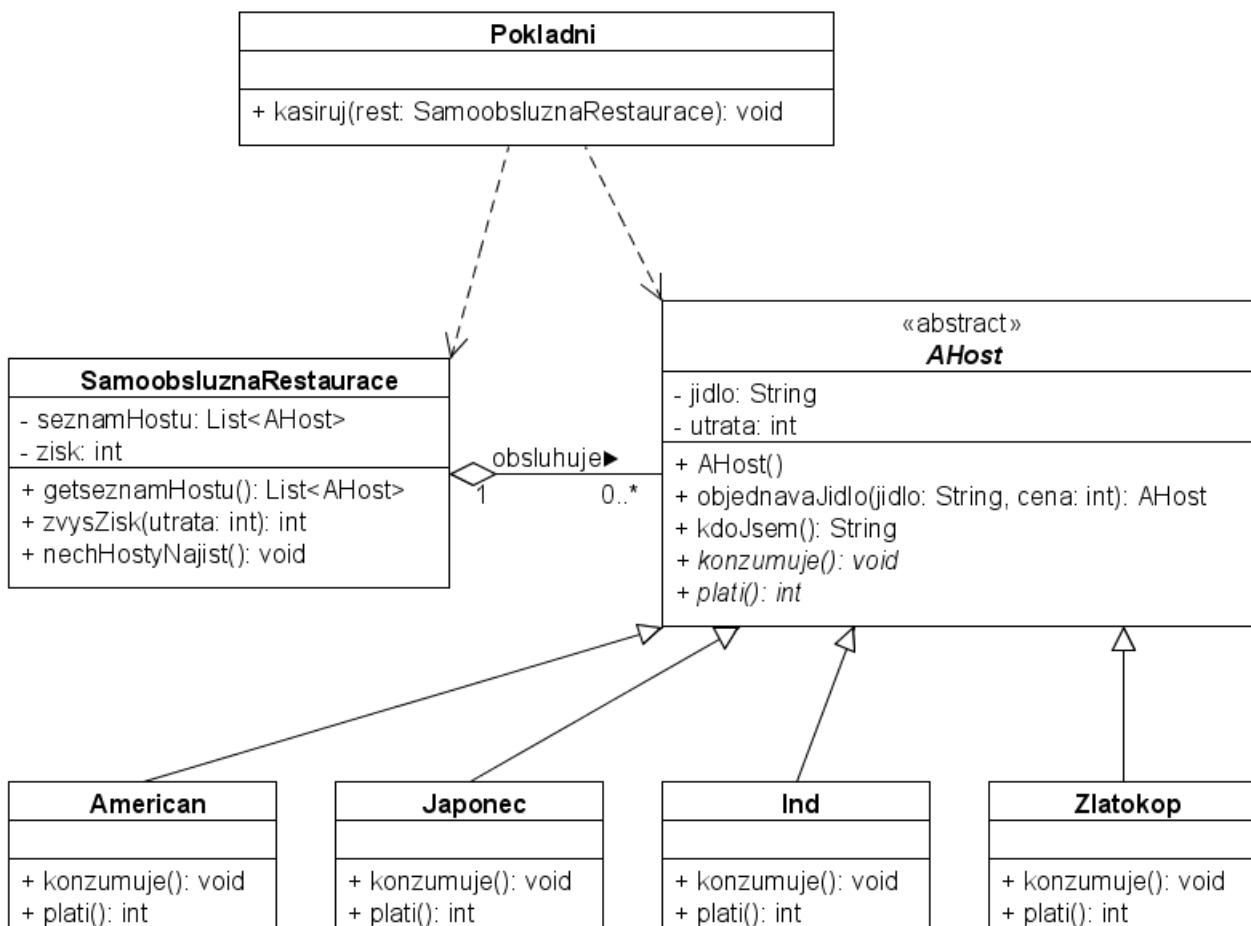
- místo nich pracuje pouze s jejich předkem AHost, která bude v metodě nechHostyNajist() volat metodu konzumuje()

```
public void nechHostyNajist() {
    for (AHost host : seznamHostu) {
        host.konzumuje();
    }
}
```

- ◆ ta je v jednotlivých třídách implementována způsobem vhodným pro konkrétního hosta
- podobně třída Pokladni bude v metodě kasiruj() volat metodu plati()

```
public void kasiruj(SamoobsluznaRestaurace restaurace) {
    List<AHost> seznamHostu = restaurace.getseznamHostu();

    for (AHost host : seznamHostu) {
        restaurace.zvysZisk(host.plati());
    }
}
```



- třída AHost

- metoda `objednavaJidlo()` vrací instanci sama sebe
  - ◆ je to podobný způsob, jako se používá např. u mnoha metod třídy `String`, aby šlo řetězit volání metod této třídy
  - ◆ zde je použit proto, aby nebylo nutné psát konstruktory ve zděděných třídách
    - využívá se v nich bezparametrický konstruktor `AHost()` – viz dále

```
abstract public class AHost {
    private String jidlo;
    private int utrata;

    public AHost() {
        this.utrata = 0;
    }

    public AHost objednavaJidlo(String jidlo, int cena) {
        this.jidlo = jidlo;
        this.utrata += cena;
        return this;
    }

    public String getJidlo() {
        return jidlo;
    }

    public int getUtrata() {
        return utrata;
    }

    abstract public void konzumuje();

    abstract public int plati();

    public String kdoJsem() {
        return this.getClass().getSimpleName();
    }

    @Override
    public String toString() {
        return kdoJsem() + " ji " + jidlo + " ";
    }
}
```

## ■ třída American

```
public class American extends AHost {

    public void konzumuje() {
        System.out.println(toString() + "priborem");
    }
}
```

```

public int plati() {
    System.out.println(kdoJsem() + " plati utratu " + getUtrata()
        + " kreditni kartou");
    return getUtrata();
}
}

```

## ■ třída Japonec

```

public class Japonec extends AHost {

    public void konzumuje() {
        System.out.println(toString() + "hulkami");
    }

    public int plati() {
        System.out.println(kdoJsem() + " plati utratu " + getUtrata()
            + " sekem");
        return getUtrata();
    }
}

```

## ■ podobně třídy Ind a Zlatokop

## ■ třída SamoobsluznaRestaurace

```

public class SamoobsluznaRestaurace {
    private List<AHost> seznamHostu = new ArrayList<>();
    private int zisk = 0;

    public List<AHost> getSeznamHostu() {
        return seznamHostu;
    }

    public void zvysZisk(int utrata) {
        zisk += utrata;
    }

    public int getZisk() {
        return zisk;
    }

    public void nechHostyNajist() {
        for (AHost host : seznamHostu) {
            host.konzumuje();
        }
    }
}

```

## ■ třída Pokladni

```

public class Pokladni {
    public void kasiruj(SamoobsluznaRestaurace restaurace) {
        List<AHost> seznamHostu = restaurace.getSeznamHostu();

        for (AHost host : seznamHostu) {
            restaurace.zvysZisk(host.plati());
        }
    }
}

```

## ■ třída Hlavní

```

public class Hlavní {

    public static void main(String[] args) {
        SamoobsluznaRestaurace restaurace = new SamoobsluznaRestaurace();

        List<AHost> seznamHostu = restaurace.getSeznamHostu();
        seznamHostu.add(new American().objednavajIdlo("steak", 300));
        seznamHostu.add(new Japonec().objednavajIdlo("susi", 200));
        seznamHostu.add(new Ind().objednavajIdlo("capati", 80));
        seznamHostu.add(new Zlatokop().objednavajIdlo("fazole", 50));
        seznamHostu.add(new Slon().objednavajIdlo("salat", 0));

        // AHost host = new Zlatokop();
        // host.objednavajIdlo("fazole", 50);
        // seznamHostu.add(host);

        restaurace.nechHostyNajist();
        new Pokladni().kasiruj(restaurace);

        System.out.println("Celkova utrata: " + restaurace.getZisk());
    }
}

```

Která vypíše:

```

American ji steak priborem
Japonec ji susi hulkami
Ind ji capati pravou rukou
Zlatokop ji fazole lzici
American plati utratu 300 kreditni kartou
Japonec plati utratu 200 sekem
Ind plati utratu 80 hotovosti
Zlatokop plati utratu 50 valouny
Celkova utrata: 630

```

## ■ pokud bychom v budoucnosti přidali další typ hosta, znamenalo by to připravit pouze třídu tohoto hosta

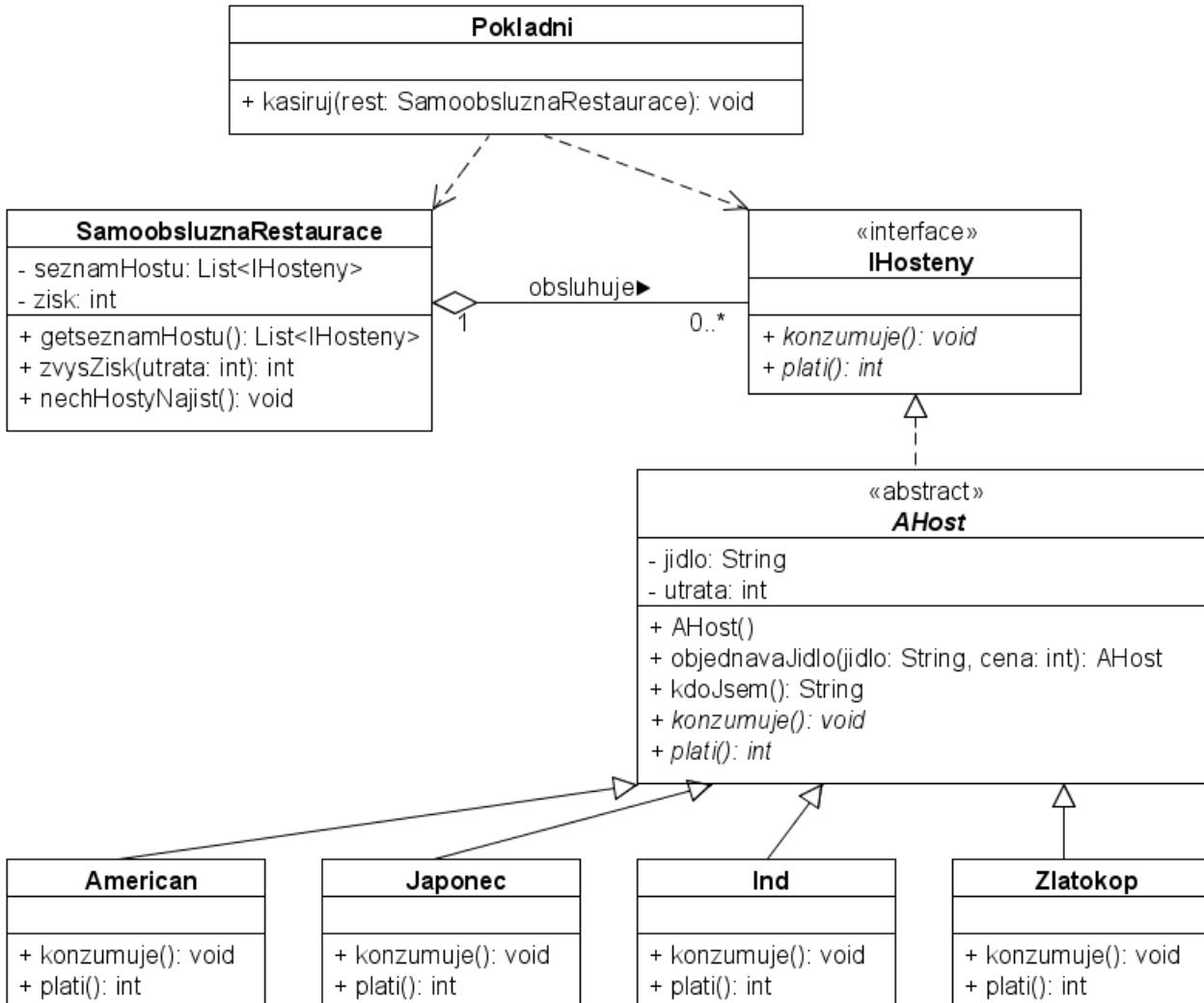
- třídy SamoobsluznaRestaurace a Pokladni by se nijak nezměnily

## 9.1.2. Využití polymorfismu pomocí rozhraní

- velmi podobné polymorfismu s dědičností
- z důvodů větší nezávislosti a pružnosti návrhu se obvykle navrhuje rozhraní, které definuje požadované chování (požadované metody)
- toto rozhraní pak implementuje abstraktní třída
- mnoho tříd z Java Core API je navrženo tímto způsobem, např. kolekce a mapy
- snahou návrhu je, aby třídy SamoobsluznaRestaurace a Pokladni nebyly pokud možno provázány těsnými vazbami
  - provázání bude provedeno v hlavní třídě
- v příkladu bude jen nepatrná změna oproti předchozímu:
  - zamění se AHost za IHosteny všude ve třídách SamoobsluznaRestaurace a Pokladni
  - třída AHost bude mít hlavičku

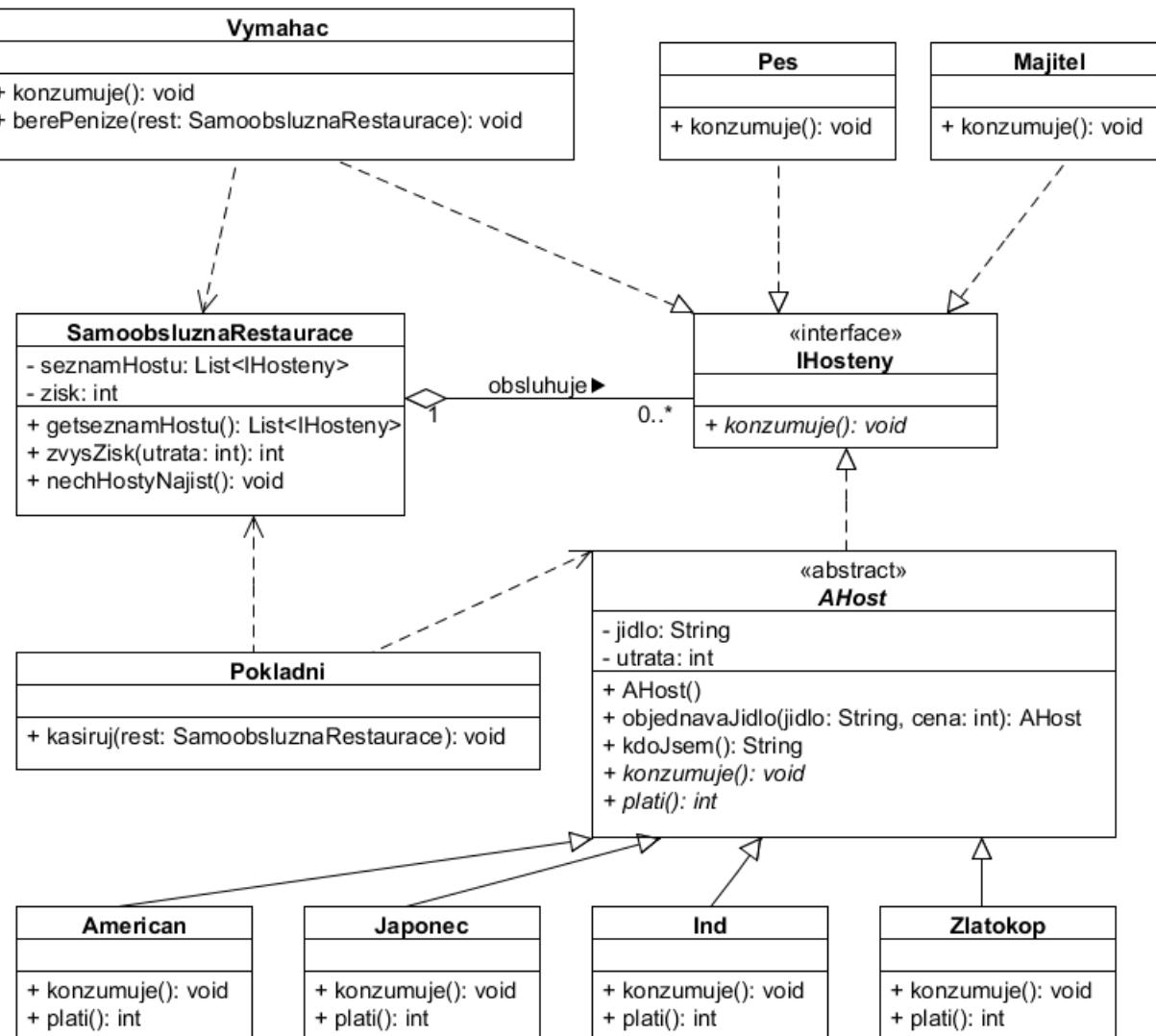
```
abstract public class AHost implements IHosteny {
```

- vše ostatní zůstane stejné



### 9.1.3. Využití rozhraní různými datovými typy

- předchozí případy vždy pracovaly jen s jednou implementací **IHosteny** – třídou **AHost**
  - zde jsou připraveny tři další, typově zcela navzájem odlišné třídy
  - rozhraní **IHosteny** muselo být zjednodušeno



## ■ změnilo se rozhraní IHosteny

```
public interface IHosteny {
    public void konzumuje();
}
```

## ■ nové datové typy

```
public class Majitel implements IHosteny {

    public void konzumuje() {
        System.out.println(getClass().getSimpleName()
                           + " ochutnava vsechno");
    }
}

public class Pes implements IHosteny {

    public void konzumuje() {
        System.out.println(getClass().getSimpleName()
                           + " zere zbytky");
    }
}
```

```

public class Vymahac implements IHosteny {

    public void konzumuje() {
        System.out.println(getClass().getSimpleName()
            + " ji jen to nejlepsi");
    }

    public void berePenize(SamoobsluznaRestaurace restaurace) {
        int vypalne = 500;
        System.out.println("Vymaha penize: " + vypalne);
        restaurace.zvysZisk(-vypalne);
    }
}

```

## ■ třída Hlavni

```

public final class Hlavni {

    private Hlavni() { /* prazdny konstruktor */ }

    public static void main(String[] args) {
        SamoobsluznaRestaurace restaurace = new SamoobsluznaRestaurace();

        List<IHosteny> seznamHostu = restaurace.getSeznamHostu();
        seznamHostu.add(new American().objednavajIdlo("steak", 300));
        seznamHostu.add(new Japonec().objednavajIdlo("susi", 200));
        seznamHostu.add(new Ind().objednavajIdlo("capati", 80));
        seznamHostu.add(new Zlatokop().objednavajIdlo("fazole", 50));

        seznamHostu.add(new Pes());
        seznamHostu.add(new Majitel());
        Vymahac vymahac = new Vymahac();
        seznamHostu.add(vymahac);

        restaurace.nechHostyNajist();
        new Pokladni().kasiruj(restaurace);

        System.out.println("Celkova utrata: " + restaurace.zvysZisk(0));
        vymahac.berePenize(restaurace);
        System.out.println("Celkovy zisk: " + restaurace.zvysZisk(0));
    }
}

```

## ■ vypíše

```

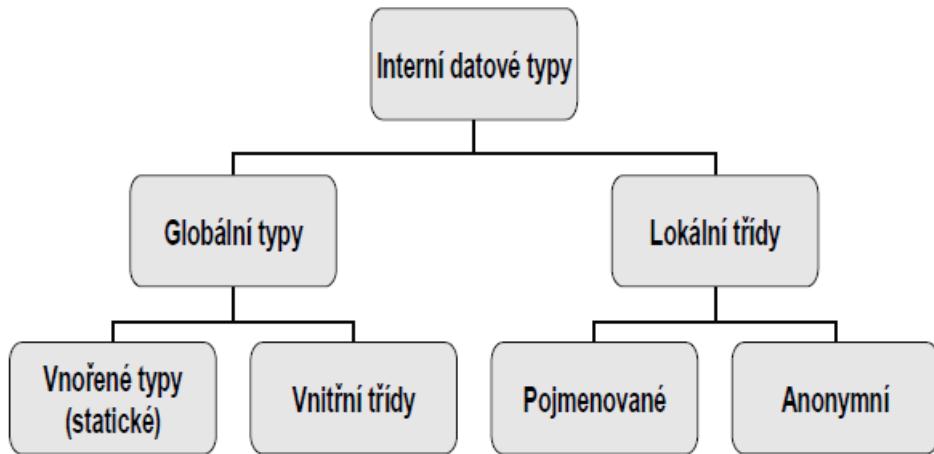
American ji steak plastikovym priborem
Japonec ji susi hulkami
Ind ji capati pravou rukou
Zlatokop ji fazole lzici
Pes zere zbytky
Majitel ochutnava vsechno
Vymahac ji jen to nejlepsi

```

```
American plati utratu 300 kreditni kartou  
Japonec plati utratu 200 sekem  
Ind plati utratu 80 hotovosti  
Zlatokop plati utratu 50 valouny  
Celkova utrata: 630  
Vymaha penize: 500  
Celkovy zisk: 130
```

## 9.2. Interní datové typy

- typy definované uvnitř jiných typů
  - není to parametrizování typů, jako např. `List<String>`
- dosud měla třída atributy a metody
  - mimo ně lze definovat interní datový typ „třídu ve třídě“
- lze je dělit na
  - globální – jsou ve vnější třídě umístěny mimo bloky kódu, tj. na úrovni atributů a metod
    - ◆ stejně jako atributy a metody mohou mít modifikátor `static`
      - mají `static` – **vnořené typy (nested, embedded, static inner)**
      - **vnitřní třídy (inner)**
    - ◆ mohou mít nastaven kterýkoliv modifikátor přístupu, tj. `public`, `protected`, `private`
    - ◆ přístup z vnějšku k `public` typům je jako k atributům či metodám, např.
  - java.awt.geom.Point2D.Double
  - lokální – jsou umístěny uvnitř metod a bloků kódu své vnější třídy
    - ◆ pojmenované (*named*)
    - ◆ anonymní (*anonymous*)
    - ◆ vlastnosti
      - mohou to být pouze třídy, nelze definovat lokální rozhraní
      - stejně jako lokální proměnné jsou zvenku nedosažitelné, ale
        - je možno je vypropagovat mimo daný blok
        - tam se vydávají za instance svého předka či implementovaného rozhraní
    - následující obrázek je převzat z knihy *Pecinovský, R.: Myslíme objektově v jazyku Java*



- každý interní datový typ má po překladu svůj `.class` soubor
  - jeho název je složen z názvu jeho vnější třídy následované znakem \$ (dolar) a názvem interní třídy
  - u anonymních tříd je místo jejich názvu přirozené číslo
  - překladem jedné vnější třídy tedy může vzniknout množství `.class` souborů, např. třída `IO.java` má:
    - ◆ `IO.class` – vnější třída
    - ◆ `IO$1.class` – anonymní lokální třída
    - ◆ `IO$ITester.class` – vnořené rozhraní vnořené třídy `Odhackuj`
    - ◆ `IO$Odhackuj.class` – vnořená třída
    - ◆ `IO$Oprava.class` – vnořená třída
    - ◆ `IO$Zpravodaj.class` – vnořená třída
- protože jsou interní datové typy umístěny uvnitř svých vnějších typů, vidí i na jejich soukromé (`private`) členy
  - to je jeden z hlavních důvodů používání těchto typů
  - v zásadě jsou tyto důvody použití:
    1. datový typ jenom pro interní použití v dané třídě
      - ◆ označíme jej většinou jako soukromý nebo lokální, takže o něm nikdo z vnějšku nebude vědět
      - ◆ typický případ je obsluha GUI prvků (tlačítka apod.) – viz dále
    2. datový typ je s vnější třídou úzce svázán (patří k sobě) a bez ní nemá opodstatnění – když vznikne, tak již existuje vnější třída
      - ◆ typicky je `public` a je tedy přístupný z vnějšku
      - ◆ je to extrémní příklad **silné aggregace** (kompozice) – viz dříve v UML

- ◆ typický případ je `java.util.Map.Entry<K, V>`

3. datový typ, který intenzivně využívá soukromé složky svého vnějšího datového typu – je poskytován navenek nejčastěji pomocí rozhraní

- ◆ typický případ je Iterátor nebo komparátor

- interní datové typy se používají překvapivě často (viz již dříve)
- při základních způsobech použití nepřinášejí žádné větší komplikace
- dále budou ukázky typického použití každé ze čtyř výše zmíněných kategorií
- podrobný rozbor kladů a záporů viz *Bloch, J.: Java efektivně*

## 9.2.1. Vnořené typy

- lze takto definovat třídu, rozhraní i výčtový typ – proto vnořené **typy**
- v podstatě se jedná jen o rozšíření jmenného prostoru
- vnořené typy můžeme definovat i uvnitř rozhraní (např. `java.util.Map.Entry`)
- příklad použití, kdy výčtový typ `Fakulty` je spojen se třídou `ZCU`

```
public class ZCU {
    final public static String NAZEV = "Zapadoceska univerzita v Plzni";

    public static enum Fakulty {
        FAV, FDU, FEK, FEL, FF, FPE, FPR, FST, FZS, UJP;
    }

    class Pouziti {
        public static void main(String[] args) {
            System.out.println("Pracuji na fakulte " + ZCU.Fakulty.FAV);
        }
    }
}
```

- další příklad viz dále

## 9.2.2. Vnitřní třídy

- jednou z možných ukázek použití je včlenění třídy `Par` do třídy `Rande`
  - tím se zjednoduší komunikace mezi těmito dvěma třídami a také se zvýší bezpečnost použití
    - ◆ instance `Par` může vzniknout až tehdy, když obě osoby přijdou na schůzku, tj. je zajištěno, že jsou fyzicky vedle sebe
      - konstruktor `Par()` je `private`, což dovoluje, aby byl vyvolán pouze ze třídy `Rande` nikoliv z vnějšku

- ♦ metody z Par mají přímý přístup k atributům Rande, tj. není třeba duplikovat muž a žena
- v příkladu budou vynechány nepodstatné metody ze třídy Rande

```

/*
 * Instance třídy {@code Rande} představují dvojice Osob,
 * které jsou na schůzce
 *
 * @author Pavel Herout
 * @version 3.00.000
 */
public class Rande {

//== KONSTANTNÍ ATRIBUTY TŘÍDY =====
    /** správce platna */
    private static final SprávcePlatna SP = SprávcePlatna.getInstance();

//== KONSTANTNÍ ATRIBUTY INSTANCI =====
    /** muž */
    private final Osoba muž;
    /** žena */
    private final Osoba žena;

    /** vyskovy posun muze, aby byl par na stejne dolni urovni */
    private final int yPosunMuze;
    /** vyskovy posun zeny, aby byl par na stejne dolni urovni */
    private final int yPosunŽeny;

    /** domácí pozice muže */
    private final Pozice domaMuž;
    /** domácí pozice ženy */
    private final Pozice domaŽena;

//##### KONSTRUKTORY A TOVÁRNÍ METODY =====
//=====
    /**
     * převeze instance muže a ženy a aktualizuje domácí pozice
     * zajistí zobrazení muže a ženy
     *
     * @param muž existující muž chystající se na rande
     * @param žena existující žena chystající se na rande
     */
    public Rande(Osoba muž, Osoba žena) {
        this.muž = muž;
        this.domaMuž = muž.getPozice();
        this.žena = žena;
        this.domaŽena = žena.getPozice();
        zajistitZobrazeníOsoby(muž);
        zajistitZobrazeníOsoby(žena);
        this.yPosunMuze = yPosunMuze();
        this.yPosunŽeny = yPosunŽeny();
    }
}

```

```

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCÍ =====
// vynecháno

//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ =====
/***
 * přesune jednotlivě muže a ženu na místo schůzky,
 * žena bude z pohledu muže vpravo
 * nohy musejí mít na stejném úrovni
 *
 * @param mistoSchuzky pozice místa schůzky
 * @param chuzeNaRande přesouvač
 *
 * @return vytvořený páru pro účely dalších přesunů
 */
public Par jdouNaRande(Pozice mistoSchuzky, Presouvac chuzeNaRande) {
    Pozice pM = new Pozice(mistoSchuzky.x - domaMuz.x + zena.getSirka(),
                           mistoSchuzky.y - domaMuz.y + yPosunMuze);
    chuzeNaRande.presunO(pM, muz);
    Pozice pZ = new Pozice(mistoSchuzky.x,
                           mistoSchuzky.y + yPosunZeny);
    chuzeNaRande.presunNa(pZ, zena);
Par par = new Par();
    return par;
}

/***
 * Přesune páru společně na zadáno místo
 *
 * @param par páru, který se dříve sešel a teď jde spolu
 * @param mistoVyletu pozice místa prvního výletu
 * @param chuzeSpolu přesouvač
 */
public void parJdeSpolecne(Par par, Pozice mistoVyletu, Presouvac chuzeSpolu) ▶
{
    chuzeSpolu.presunNa(mistoVyletu, par);
}

/***
 * přesune jednotlivce na jejich domácí pozice
 *
 * @param par páru, který se rozchází domů
 * @param chuzeDomu přesouvač
 */
public void jdouDomu(Par par, Presouvac chuzeDomu) {
    Pozice dZ = new Pozice(domaZena.x, domaZena.y - yPosunZeny);
    chuzeDomu.presunNa(dZ, par);
    chuzeDomu.presunNa(domaMuz, muz);
}

/***
 * Y posun muže, pokud je žena vyšší, jinak 0
 * @return posun v pixelech
 */

```

```

public int yPosunMuze() {
    int posun = Math.abs(muz.getVyska() - zena.getVyska());
    if (muz.getVyska() < zena.getVyska()) {
        return posun;
    }
    else {
        return 0;
    }
}

/**
 * Y posun ženy, pokud je muž vyšší, jinak 0
 * @return posun v pixelech
 */
public int yPosunZeny() {
    int posun = Math.abs(muz.getVyska() - zena.getVyska());
    if (muz.getVyska() > zena.getVyska()) {
        return posun;
    }
    else {
        return 0;
    }
}

//== INTERNÍ DATOVÉ TYPY =====
/*****************/
 * Instance třídy {@code Par} představují pář muže a ženy na schůzce
 * instance je použita proto, aby se oba spolu zobrazovali, přesouvali
 * a zvýrazňovali
 */
public class Par implements IZvyrazneny {

//== KONSTRUKTORY A TOVÁRNÍ METODY =====
/*****************/
 * Vytvoří pář z muže a ženy, kteří přišli na rande
 * nelze volat z vnějšku,
 * pář se vytvoří jedině pomocí Rande.jdouNaRande()
 *
 */
private Par() {
}

//== OSTATNÍ NESOUKROMÉ METODY INSTANCI =====
/*****************/
 * Prostřednictvím dodaného kreslítko vykreslí obraz své instance.
 *
 * @param kreslitko Kreslítko, které nakreslí instanci
 */
@Override
public void nakresli(Kreslitko kreslitko) {
    SP.nekresli();
    muz.nakresli(kreslitko);
    zena.nakresli(kreslitko);
}

```

```

        } SP.vratKresli();
    }

/********************* Vrátí instanci třídy {@code Pozice} s aktuální pozicí instance. ****
 * @return Instance třídy {@code Pozice} s aktuální pozicí instance
 */
@Override
public Pozice getPozice() {
    return new Pozice(zena.getX(), zena.getY() - yPosunZeny);
}

/********************* Nastavi novou pozici objektu. ****
 * @param x Nova x-ova pozice objektu
 * @param y Nova y-ova pozice objektu
 */
@Override
public void setPozice(int x, int y) {
    zena.setPozice(x, y + yPosunZeny);
    muz.setPozice(x + zena.getSirka(), y + yPosunMuze);
}

/********************* Vrátí šířku páru. ****
 * @return šířka páru
 */
@Override
public int getSirka() {
    return muz.getSirka() + zena.getSirka();
}

/********************* Vrátí výšku páru. ****
 * @return výška páru
 */
@Override
public int getVyska() {
    return Math.max(muz.getVyska(), zena.getVyska());
}
}
}

```

## ■ použití např. ve třídě RandeTest

```

@Test
public void testZvyrazneniParu() {
    Pozice pM = new Pozice(10, 10);
    Pozice pZ = new Pozice(400, 150);

```

```

Osoba muz = Osoba.getBeznyMuz(pM);
Osoba zena = Osoba.getBeznaZena(pZ);
rande = new Rande(muz, zena);

Presouvac chuzeNaRande = new Presouvac(25);
Pozice pS = new Pozice(150, 80);
Rande.Par par = rande.jdouNaRande(pS, chuzeNaRande);

rande.parJdeSpolecne(par, pS, chuzeNaRande);
assertEquals("Chybná šířka páru: ", 140, par.getSirka());
assertEquals("Chybná výška páru: ", 140, par.getVyska());
assertEquals("Chybná šířka páru: ", 140, par.getRozmer().sirka);
assertEquals("Chybná výška páru: ", 140, par.getRozmer().vyska);
Zvyraznovac zvyraznovac = new Zvyraznovac();
zvyraznovac.zvyrazniPozadi(par, Barva.STRIBRNA);
}

```

### 9.2.2.1. Použití jako třída obsluhující GUI

- toto je další typické použití (jako alternativa k lokálním anonymním třídám) – podrobnosti viz v KIV/UUR
  - využívá se návrhový vzor Vysílač–Posluchač
- tyto třídy jsou typicky private, protože mají smysl jen pro svoji vnější třídu

```

 JButton tlacitkoAhojBT = new JButton("Ahoj");
 tlacitkoAhojBT.addActionListener(new VnitrniTrida());
 ...
}

private class VnitrniTrida implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        napisLB.setText("Stisknuto Ahoj");
    }
}

```



- dvě omezení, na které brzy narazíme:
  - potřebujeme-li využít atribut this, musíme použít VnějšíTřída.this
  - nesmějí mít statické atributy a metody

### 9.2.3. Anonymní lokální

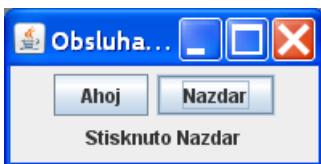
- třídy na jedno použití, na které se už nikdy nebudeme odvolávat, a proto nemusejí mít jméno

- pokud mají více než cca tři řádky výkonného kódu, jsou nepřehledné a měly by být nahrazeny pojmenovanou vnitřní třídou
- nesmějí mít statické atributy a metody
- typicky jsou to implementace rozhraní na místě

### 9.2.3.1. Použití jako třída obsluhující GUI

- typické použití (jako alternativa ke vnitřním třídám) – podrobnosti viz v KIV/UUR
  - vytváříme instanci rozhraní (`new ActionListener()`), které lokálně implementujeme

```
 JButton tlacitkoNazdarBT = new JButton("Nazdar");
 tlacitkoNazdarBT.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
         napisLB.setText("Stisknuto Nazdar");
     }
});
```



- v případě delšího obslužného kódu nebo více implementovaných metod jsou nepřehledné

### 9.2.3.2. Použití jako předpřipravené komparátory

- s tímto použitím jsme se setkali u absolutního řazení
- třída si implementací `Comparable` definuje své přirozené řazení, ale navíc může předpřipravit několik statických konstant pro předpokládaná absolutní řazení
  - v anonymní implementaci rozhraní `Comparator` lze používat přímý přístup k atributům vnější třídy `Osoba`

```
public class Osoba implements Comparable<Osoba> {
    private int vyska;
    private double vaha;
    private String popis;

    // konstanty komparátorů
    final public static Comparator<Osoba> KOMPARATOR_PODLE_VAHY =
        new Comparator<Osoba>() {
            public int compare(Osoba o1, Osoba o2) {
                return (int) (o1.vaha - o2.vaha);
            }
        };

    final public static Comparator<Osoba> KOMPARATOR_PODLE_POPISU =
        new Comparator<Osoba>() {
            public int compare(Osoba o1, Osoba o2) {
                String s1 = o1.popis;
```

```

        String s2 = o2.popis;
        return s1.compareTo(s2);
    }
};

Osoba(int vyska, double vaha, String popis) {
    this.vyska = vyska;
    this.vaha = vaha;
    this.popis = popis;
}

public int compareTo(Osoba os) {
    int osVyska = os.vyska;
    if (this.vyska > osVyska)
        return +1;
    else if (this.vyska == osVyska)
        return 0;
    else
        return -1;
}

public String toString() {
    return "vy = " + vyska + ", va = " + vaha + ", " +
    popis ;
}
}

```

## ■ příklad použití

```

public class Komparatory {
    public static void main(String[] args) {
        Osoba[] poleOsob = new Osoba[4];
        poleOsob[0] = new Osoba(186, 82.5, "muz");
        poleOsob[1] = new Osoba(172, 63.0, "zena");
        poleOsob[2] = new Osoba(105, 26.1, "dite");
        poleOsob[3] = new Osoba(116, 80.5, "obezni trpaslik");

//        Arrays.sort(poleOsob, Osoba.KOMPARATOR_PODLE_VAHY);
//        Arrays.sort(poleOsob, Osoba.KOMPARATOR_PODLE_POPISU);

        for (int i = 0; i < poleOsob.length; i++)
            System.out.println("[" + i + "] " + poleOsob[i]);
    }
}

```

## ■ vypíše

```

[0] vy = 105, va = 26.1, dite
[1] vy = 186, va = 82.5, muz
[2] vy = 116, va = 80.5, obezni trpaslik
[3] vy = 172, va = 63.0, zena

```

## 9.2.4. Pojmenované lokální

- není žádný běžný důvod, proč je používat

## 9.3. Metody s proměnným počtem parametrů

- někdy se nám hodí, aby metoda měla možnost použít více než jeden parametr, ale aby tyto parametry byly jednotným způsobem přístupné
- tyto situace se často řeší pomocí pole jako jednoho formálního parametru

```
int sumaPolem(int[] pole) {
```

- dají se ale řešit i proměnným počtem parametrů

- to je jednodušší zápis při volání metody, kdy skutečné parametry nejsou prvky pole
- při použití pole jako skutečného parametru by se do něj musely nejdříve uložit

- omezení metod s proměnným počtem parametrů

- parametry musejí být stejného typu
- proměnný počet parametrů musí být poslední formální parametr metody

- proměnný počet parametrů se zapíše jménem typu následovaným třemi tečkami a pak jménem formálního parametru

```
int sumaViceParametru(int... params) {
```

- skutečné parametry metody s proměnným počtem parametrů jsou pak do metody vnitřně předány jako pole, takže v ní lze použít všechny možnosti práce s polem, nejčastěji:

- skutečný počet parametrů – pomocí `length`
- přístup k jednotlivým parametrům – pomocí `[index]`
- zpracování všech parametrů – pomocí *For-Each*

```
public class PPP {  
    public static int sumaPolem(int[] pole) {  
        int suma = 0;  
        for (int p : pole) {  
            suma += p;  
        }  
        return suma;  
    }  
  
    public static int sumaViceParametru(int... params) {  
        int suma = 0;  
        for (int i = 0; i < params.length; i++) {  
            suma += params[i];  
        }  
        return suma;  
    }  
}
```

```
        }
        return suma;
    }

public static int sumaViceParametruFigl(int... params) {
    return sumaPolem(params);
}

public static void main(String[] args) {
    int[] poleA = {1, 2, 3, 4};
    System.out.println(sumaPolem(poleA));
    int p1 = 1;
    int p2 = 2;
    int p3 = 3;
    int p4 = 4;
    System.out.println(sumaViceParametru(p1, p2, p3, p4));
    System.out.println(sumaViceParametruFigl(p1, p2, p3, p4, p1, p2, p3));
}
}
```

## Poznámka

Známá metoda s proměnným počtem parametrů je `System.out.format()` [PPA1-36]



# Kapitola 10. Lambda výrazy a spol.

- lambda výrazy (*lambda expressions* nebo *lambdas*) jsou od Java 8
- podobné konstrukce se vyskytují i v jiných jazycích
  - Java se inspirovala a vzala si to dobré
- lambda výrazy jsou čím dál tím víc používány a jejich znalost je nezbytná
- výhody:
  - výrazně zjednodušují zápis **krátkých částí** kódu, např. oproti anonymním vnitřním třídám
    - ◆ hodí hlavně tam, kde je potřeba **předat jako skutečný parametr** akci, která se má s daty provést (obsluha tlačítka, výběr dat z kolekce)
      - tj. předává se kód, nikoliv data
  - mohou být deklarovány jako hodnoty proměnných typu funkčního rozhraní
  - mohou být předávány jako parametry metod
  - při použití v kolekcích umožňují využít paralelismus
- zdroje:
  - Java Tutorial
  - [www.lambdafaq.org](http://www.lambdafaq.org)

## 10.1. Funkční rozhraní

- *functional interface*
  - též *Single Abstract Method (SAM)*
- ### Poznámka

Opakování a rozšíření z dřívějška.

  - funkční rozhraní jsou klíčová pro použití lambda funkcí – jsou to jejich typy
  - je to rozhraní, které má jen jednu abstraktní metodu
    - může mít dále defaultní a/nebo statické metody, což není pro lambda výrazy podstatné
  - existuje-li jen jedna metoda, pak ji jednoznačně určuje už samotný název rozhraní
    - to lze dále zobecnit – je-li objekt typu funkčního rozhraní, pak disponuje pouze jednou metodou
  - aby mohl překladač ohlídat unikátnost metody, lze použít anotaci `@FunctionalInterface`

```
@FunctionalInterface  
public interface IFunkcniRozhrani {
```

```
    public int jedinaMetoda(int a, int b);  
}
```

- co se týče počtu formálních parametrů a typu návratové hodnoty, neklade funkční rozhraní žádné omezení

```
@FunctionalInterface  
public interface IFunkcniRozhraniVoid {  
    public void jedinaMetoda();  
}
```

- velmi často bývá funkční rozhraní generické, např.

```
@FunctionalInterface  
public interface IFunkcniRozhraniGenericko<T> {  
    public void jedinaMetoda(T typ);  
}
```

- pokud nechceme využívat lambda výrazy, lze funkční rozhraní používat v běžném smyslu, jako jakákoliv jiná rozhraní

## 10.1.1. Funkční rozhraní z Java Core API

- protože se očekává stále častější využívání funkčních rozhraní, jsou některé typově nejčastější již připraveny v balíku `java.util.function`
- balík obsahuje asi 70 rozhraní, ale ty jsou často seskupovány (po čtveřicích) následovně:
  - `Consumer` – generické rozhraní – `void accept(T typ)`
  - `IntConsumer` – rozhraní pro práci s typem `int` – `void accept(int value)`
  - `LongConsumer` – rozhraní pro práci s typem `long` – `void accept(long value)`
  - `DoubleConsumer` – rozhraní pro práci s typem `double` – `void accept(double value)`
- přehled skupin generických rozhraní (na jménu metody ve skutečnosti nezáleží – podstatný je počet formálních parametrů a návratový typ):
  - `Consumer` – `void accept(T typ)`
  - `BiConsumer` – `void accept(T typT, U typU)`
  - `Supplier` – `T get()`
  - `Predicate` – `boolean test(T typ)`
  - `Function` – `R apply(T typ)`
  - `UnaryOperator` – `T apply(T typ)`
- takže pokud bychom chtěli vytvářet vlastní funkční rozhraní, je vhodné zjistit, zda již v `java.util.function` neexistuje

- např. místo dříve uvedeného `IFunkcniRozhraniGenerické` by se použilo `Consumer`
- k těmto funkčním rozhraním je třeba přidat podobná rozhraní již známá z dřívějška
  - `java.lang.Comparable` – `int compareTo(T typ)`
  - `java.lang.Iterable` – `Iterator<T> iterator()`

## 10.2. Lambda výrazy

- *lambda expression*, též *lambdas* nebo *closure* (uzávěra)

### 10.2.1. Přechod z anonymních vnitřních tříd na lambda výrazy

- jak fungují lambda výrazy bude ukázáno na jednoduché vyhledávací aplikaci s využitím „evoluce“

#### 10.2.1.1. Vyhledávaná třída a pomocná třída

- budeme mít aplikaci, ve které bude existovat seznam osob
- nad tímto seznamem budeme **často** vyhledávat podle různých kriterií
- cílem akce je zapsat příkazy pro stanovení kriterií vyhledávání co nejjednodušeji
- třída `Osoba` – je to běžná třída, která nemá žádný vztah k lambda výrazům

```
public class Osoba {  
    public enum Pohlavi {  
        MUZ, ZENA;  
    }  
  
    private String jmeno;  
    private int vyska;  
    private Pohlavi pohlavi;  
  
    public Osoba(String jmeno, int vyska, Pohlavi pohlavi) {  
        this.jmeno = jmeno;  
        this.vyska = vyska;  
        this.pohlavi = pohlavi;  
    }  
  
    public String getJmeno() {  
        return jmeno;  
    }  
  
    public int getVyska() {  
        return vyska;  
    }  
  
    public Pohlavi getPohlavi() {  
        return pohlavi;  
    }  
}
```

```

}

@Override
public String toString() {
    return "[jmeno=" + jmeno + ", vyska=" + vyska + ", pohlavi="
           + pohlavi + "]";
}

public void vypis() {
    System.out.println(this);
}
}

```

- třída SeznamOsob – nemá žádný vztah k lambda výrazům, jedná se pouze o datový zdroj

```

import java.util.ArrayList;
import java.util.List;

public class SeznamOsob {
    private static final String[][] OSOBY = {
        {"Anna", "155", "Z"}, {"Jana", "165", "Z"}, {"Hana", "175", "Z"},
        {"Josef", "160", "M"}, {"Karel", "170", "M"}, {"Alois", "180", "M"},
    };

    public static List<Osoba> getSeznam() {
        List<Osoba> seznam = new ArrayList<>();
        for (String[] osoba : OSOBY) {
            int vyska = Integer.parseInt(osoba[1]);
            Osoba.Pohlavi pohlavi = osoba[2].startsWith("Z")
                ? Osoba.Pohlavi.ZENA : Osoba.Pohlavi.MUZ;
            seznam.add(new Osoba(osoba[0], vyska, pohlavi));
        }
        return seznam;
    }
}

```

## 10.2.1.2. Naivní vyhledávání

- pro hledání máme samostatnou specializovanou metodu zenyVyssiNez()

```

import java.util.List;

public class HledaniOsob_1 {

    public static void zenyVyssiNez(List<Osoba> seznam, int vyska) {
        for (Osoba o : seznam) {
            if (o.getPohlavi() == Osoba.Pohlavi.ZENA
                && o.getVyska() >= vyska) {
                o.vypis();
            }
        }
    }
}

```

```

public static void main(String[] args) {
    List<Osoba> seznam = SeznamOsob.getSeznam();
    zenyVyssiNez(seznam, 160);
}
}

```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

### 10.2.1.3. Rozhraní definuje vyhledávací kriterium

- protože vyhledávání bude více, bude vhodné napsat vyhledávací metodu univerzálněji (je přejmenována na `hledani()`) a vyhledávací kriterium specifikovat pomocí rozhraní

```

public interface IKriterium {
    public boolean vyhovuje(Osoba o);
}

```

- konkrétní vyhledávací kriterium je pak určeno třídou `ZenaVyssiNez160`

- může být vnější třída
- pravděpodobně by ale byla vnitřní třídou
- její nevýhodou je značné množství administrativního kódu, kterého je víc než výkonného kódu

```

class ZenaVyssiNez160 implements IKriterium {
    @Override
    public boolean vyhovuje(Osoba o) {
        return o.getPohlavi() == Osoba.Pohlavi.ZENA
            && o.getVyska() >= 160;
    }
}

```

- `import java.util.List;`

```

public class HledaniOsob_2 {

    public static void hledani(List<Osoba> seznam, IKriterium kriterium) {
        for (Osoba o : seznam) {
            if (kriterium.vyhovuje(o)) {
                o.vypis();
            }
        }
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        hledani(seznam, new ZenaVyssiNez160());
    }
}

```

```
    }  
}
```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]  
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

## 10.2.1.4. Použití anonymní třídy

- pro tak **krátký výkonný kód** se vyplatí použití anonymní vnitřní třídy jako skutečného parametru metody `hledani()`

```
import java.util.List;  
  
public class HledaniOsob_3 {  
  
    public static void hledani(List<Osoba> seznam, IKriterium kriterium) {  
        // ... nezměněno  
    }  
  
    public static void main(String[] args) {  
        List<Osoba> seznam = SeznamOsob.getSeznam();  
        hledani(seznam,  
            new IKriterium() {  
                public boolean vyhovuje(Osoba o) {  
                    return o.getPohlavi() == Osoba.Pohlavi.ZENA  
                        && o.getVyska() >= 160;  
                }  
            } );  
    }  
}
```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]  
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

## 10.2.1.5. Přechod z anonymní vnitřní třídy na lambda výraz

- protože je `IKriterium` funkční rozhraní, lze použít lambda výraz
  - ten výrazně zkracuje zápis implementace metody `vyhovuje()` z rozhraní `IKriterium`
  - ve skutečnosti není jméno metody `vyhovuje()` zapotřebí – ve funkčním rozhraní existuje jen jedna metoda
  - oproti anonymní třídě přibývá (`Osoba o`) ->
  - jinak se zapíše jen **výkonný kód jako druhý skutečný parametr metody `hledani()`** a to zcela stejně jako u anonymní vnitřní třídy

```

import java.util.List;

public class HledaniOsob_4 {

    public static void hledani(List<Osoba> seznam, IKriterium kriterium) {
        // ... nezměněno
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        hledani(seznam,
            (Osoba o) -> o.getPohlavi() == Osoba.Pohlavi.ZENA
                && o.getVyska() >= 160
        );
    }
}

```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

### 10.2.1.6. Standardní generické funkční rozhraní

- lze využít toho, že pro tak častou akci, jako je testování podmínky, je již v `java.util.function` předpřipraveno generické rozhraní `Predicate`
- pak
  - není nutné vytvářet naše vlastní rozhraní `IKriterium`
  - můžeme využít výhody generičnosti a dále zjednodušit lambda výraz
    - ◆ je dobré vidět, že jako druhý skutečný parametr metody `hledani()` se předává tělo metody `test()`

```

import java.util.List;
import java.util.function.Predicate;

public class HledaniOsob_5 {

    public static void hledani(List<Osoba> seznam,
                               Predicate<Osoba> kriterium) {
        for (Osoba o : seznam) {
            if (kriterium.test(o)) {
                o.vypis();
            }
        }
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        hledani(seznam,
            o -> o.getPohlavi() == Osoba.Pohlavi.ZENA
        );
    }
}

```

```

        && o.getVyska() >= 160
    );
}
}

```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

## 10.2.1.7. Přidána generická akce

- pokud v metodě `hledani()` použijeme jako další formální parametr funkční rozhraní `Consumer`, dostaneme možnost specifikovat v lambda výrazu výslednou akci provedenou po splnění kriteria
- při volání metody `hledani()` si pak můžeme vybrat akci, např. zda chceme vypisovat pomocí `vypis()` nebo pomocí `System.out.println()`

```

import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class HledaniOsob_6 {

    public static void hledani(List<Osoba> seznam,
                               Predicate<Osoba> kriterium,
                               Consumer<Osoba> akce) {
        for (Osoba o : seznam) {
            if (kriterium.test(o)) {
                akce.accept(o);
            }
        }
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        hledani(seznam,
                o -> o.getPohlavi() == Osoba.Pohlavi.ZENA
                    && o.getVyska() >= 160,
                o -> o.vypis()
//                o -> System.out.println(o)
        );
    }
}

```

vypíše:

```
[jmeno=Jana, vyska=165, pohlavi=ZENA]
[jmeno=Hana, vyska=175, pohlavi=ZENA]
```

## 10.2.1.8. Nejsme omezeni jen jedním typem

- ukázka, že metoda `hledani()` nemusí být ve svém těle omezena pouze jedním typem (`Osoba`)
- pomocí rozhraní `Function` lze získat jiný typ dat (zde `String`) a s ním dále pracovat

```
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class HledaniOsob_7 {

    public static void hledani(List<Osoba> seznam,
                               Predicate<Osoba> kriterium,
                               Function<Osoba, String> mapovani,
                               Consumer<String> akce) {
        for (Osoba o : seznam) {
            if (kriterium.test(o)) {
                String data = mapovani.apply(o);
                akce.accept(data);
            }
        }
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        hledani(seznam,
                o -> o.getPohlavi() == Osoba.Pohlavi.ZENA
                    && o.getVyska() >= 160,
                o -> o.getJmeno(),
                jmeno -> System.out.println(jmeno)
        );
    }
}
```

vypíše:

```
Jana
Hana
```

## 10.2.1.9. Vše generické

### Poznámka

Tento způsob zápisu určitě v začátcích programování nepoužijeme. Je uváděn jen jako ukázka, jak by se případně vytvářela vlastní agregovaná operace – viz též dále. A na tento způsob zápisu můžete také narazit v JavaDoc dokumentaci k Java Core API.

- byly zavedeny obecné generické typy `X`, `Y` a `seznam List<Osoba>` byl vyměněn za `Iterable<X>`

```

import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class HledaniOsob_8 {

    public static <X, Y> void hledani(Iterable<X> seznam,
                                         Predicate<X> kriterium,
                                         Function<X, Y> mapovani,
                                         Consumer<Y> akce) {
        for (X o : seznam) {
            if (kriterium.test(o)) {
                Y data = mapovani.apply(o);
                akce.accept(data);
            }
        }
    }

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        // nezměněno
        hledani(seznam,
                 o -> o.getPohlavi() == Osoba.Pohlavi.ZENA
                     && o.getVyska() >= 160,
                 o -> o.getJmeno(),
                 jmeno -> System.out.println(jmeno)
               );
    }
}

```

vypíše:

```
Jana
Hana
```

## 10.2.2. Syntaxe lambda výrazů

- hodnoty lambda výrazů se zapisují ve tvaru:

```

parametr -> výraz
(parametry) -> výraz
parametr -> { příkazy oddělené ; }
(parametry) -> { příkazy oddělené ; }
```

- vlevo od -> jsou vždy parametry
  - vpravo pak akce, která se má provést
- platí pravidla:
    - je-li vlevo jediný parametr, nemusí být v ( )

- ◆ naopak, není-li tam žádný parametr, jsou povinné prázdné ( )

- pro více parametrů je oddělovač čárka – viz dále

- umí-li překladač odvodit typ parametru, nemusí se typ uvádět

```
(Osoba o) -> o.getPohlavi() == Osoba.Pohlavi.ZENA
```

versus

```
o -> o.getPohlavi() == Osoba.Pohlavi.ZENA
```

- vyhodnocuje-li se vpravo nějaký výraz, nemusí být v { }

- ◆ hodnota lambda výrazu a typ hodnoty je pak dán výrazem na pravé straně

```
o.getPohlavi() == Osoba.Pohlavi.ZENA && o.getVyska() >= 160
```

- je možné použít příkaz return v { }

- ◆ hodnota a typ hodnoty lambda výrazu je dán tím, co se v return vrací

- pokud není return použit, předpokládá se, že výraz také bude mít hodnotu void

```
o -> {
    return o.getPohlavi() == Osoba.Pohlavi.ZENA
        && o.getVyska() >= 160;
}
```

- pokud je volána metoda s návratovým typem void, nemusí se dávat do { }

- ◆ kdyby metoda println() vracela třeba počet vypsaných znaků, složené závorky musí být

```
o -> System.out.println(o)
o -> o.vypis()
```

### 10.2.3. Lambda výrazy jako hodnoty funkčního rozhraní

- v předchozích příkladech byly lambda výrazy použity jako skutečné parametry metod

- příklad ukazuje:

- použití standardního funkčního rozhraní IntBinaryOperator typovaného na primitivní datový typ int
  - ◆ v předchozích příkladech byly dosud typem jen objekty
- uložení lambda výrazu do proměnné (scitani, odecitani)
- použití této proměnné jako skutečného parametru volání metody
- srovnání přehlednosti zápisu pomocí lambda výrazu a pomocí anonymní vnitřní třídy (nasobeni)

- začátečnický přístup (metoda `deleni()`), která sice vypadá jednoduše, ale typově je naprosto odlišná od všech ostatních
  - v případě rozšiřování funkčnosti by zde téměř jistě nastaly potíže

- import `java.util.function.IntBinaryOperator`;

```

public class Kalkulacka {

    public int pocetniUkon(int a, int b, IntBinaryOperator op) {
        return op.applyAsInt(a, b);
    }

    public int deleni(int a, int b) {
        return a / b;
    }

    public static void main(String[] args) {
        Kalkulacka k = new Kalkulacka();
        IntBinaryOperator scitani = (a, b) -> a + b;
        IntBinaryOperator odecitani = (a, b) -> a - b;

        IntBinaryOperator nasobeni = new IntBinaryOperator() {
            public int applyAsInt(int a, int b) {
                return a * b;
            }
        };

        System.out.println("3 + 5 = " + k.pocetniUkon(3, 5, scitani));
        System.out.println("5 - 3 = " + k.pocetniUkon(5, 3, odecitani));
        System.out.println("3 * 5 = " + k.pocetniUkon(3, 5, nasobeni));
        System.out.println("15 / 5 = " + k.deleni(15, 5));
    }
}

```

vypíše:

```

3 + 5 = 8
5 - 3 = 2
3 * 5 = 15
15 / 5 = 3

```

## 10.2.4. Lambda výrazy v GUI

- podrobnosti viz KIV/UUR
- velmi časté použití lambda výrazů je při obsluze událostí v GUI
  - výrazně zpřehledňuje kód – samozřejmě za předpokladu krátkého výkonného kódu
- obsluha stisku tlačítka s využitím anonymní vnitřní třídy

```
tlacitko.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Stisknuto!");
    }
});
```

- tatáž funkčnost s využitím lambda výrazu

```
tlacitko.setOnAction(
    event -> System.out.println("Stisknuto!")
);
```

## 10.3. Reference na metody

- díky technologii lambda výrazů lze předávat reference na již existující metody
  - to v důsledku potlačuje užitečnost návrhového vzoru Příkaz (*Command*), který dokáže totéž, ale musí předat referenci na metodu zabalenou do reference na objekt třídy
- pro referenci na metodu se používá ::

### 10.3.1. Princip reference

- základní princip je, že lambda výraz obsahuje pouze volání nějaké již existující metody, které pouze předá obdržené parametry
  - pak místo lambda výrazu stačí pouze uvést název této metody
- opět bude vysvětlen „evolučně“ na příkladu řazení řetězců
- pole řetězců představujících jména
  - jména jsou z „pedagogických důvodů“ naneštěstí zapsána s rozdílným použitím malých a velkých písmen

```
String[] poleJmen = {"anna", "Jana", "HANA"};
```

- známý příkaz pro tisk celého pole:

```
System.out.println("Nerazeno: " + Arrays.toString(poleJmen));
```

vypíše:

```
Nerazeno: [anna, Jana, HANA]
```

- pole lze seřadit metodou `Arrays.sort()`:
  - ta ale rozlišuje velká a malá písmena, takže výsledek nebude ideální

```
Arrays.sort(poleJmen);
System.out.println("Jen sort: " + Arrays.toString(poleJmen));
```

vypíše:

```
Jen sort: [HANA, Jana, anna]
```

■ metoda `Arrays.sort()` má přetíženou verzi, kdy druhým parametrem je `Comparator<String>`

- stačí jej implementovat pomocí anonymní vnitřní třídy a využít metodu `compareToIgnoreCase()` ze třídy `String`
- od této chvíle se již řadí dle očekávání

```
Arrays.sort(poleJmen,
            new Comparator<String>() {
                public int compare(String a, String b) {
                    return a.compareToIgnoreCase(b);
                }
            });
System.out.println("Anonymni comparator: " + Arrays.toString(poleJmen));
```

vypíše:

```
Anonymni comparator: [anna, HANA, Jana]
```

■ protože rozhraní `Comparator<String>` je funkční rozhraní, lze použít zjednodušený zápis pomocí lambda výrazu

```
Arrays.sort(poleJmen,
            (a, b) -> a.compareToIgnoreCase(b)
        );
System.out.println("Lambda vyraz: " + Arrays.toString(poleJmen));
```

vypíše:

```
Lambda vyraz: [anna, HANA, Jana]
```

■ protože je ale metoda `compareToIgnoreCase()` již ve třídě `String` napsaná (a samozřejmě implementuje rozhraní `Comparator<String>`), můžeme jednoduše na ni předat pouze referenci

- použijeme zápis `String::compareToIgnoreCase`
  - ◆ parametry `a, b` si překladač „domyslí“

```
Arrays.sort(poleJmen, String::compareToIgnoreCase);
System.out.println("Reference na metodu: " + Arrays.toString(poleJmen));
```

vypíše:

```
Reference na metodu: [anna, HANA, Jana]
```

## 10.3.2. Syntaxe zápisu referencí

### Poznámka

1. Seznam parametrů si překladač domyslí podle požadovaného typu cílového funkčního objektu.
2. Názvy metod píšeme bez () .

- statická nebo instanční metoda akce () definovaná ve třídě Třída

```
Třída::akce
```

příklad: String::compareToIgnoreCase

- instanční metoda akce () instance inst

```
inst::akce
```

- bezparametrický konstruktor třídy

- použití viz dále v „Agregované operace vracející kolekce“ a podrobnosti použití viz v Java Tutorial

```
Třída::new
```

- příklad je řazení objektů třídy Osoba známé z předchozích příkladů

- připravíme třídu RadiciKriterium, která neimplementuje žádné rozhraní

- má dvě metody, z nichž každá má dva formální parametry typu Osoba a vrací typ int
- metoda porovnejPodleVysky() je označena jako static, aby byly později vidět oba způsoby použití reference

```
public class RadiciKriterium {  
    public int porovnejPodleJmena(Osoba a, Osoba b) {  
        return a.getJmeno().compareTo(b.getJmeno());  
    }  
  
    public static int porovnejPodleVysky(Osoba a, Osoba b) {  
        return a.getVyska() - b.getVyska();  
    }  
}
```

- při použití si můžeme snadno vybrat, podle jakého kriteria budeme řadit

- rk::porovnejPodleJmena je reference z instance rk
- RadiciKriterium::porovnejPodleVysky je reference ze třídy
  - ◆ nešvar volání statické metody z referenční proměnné (rk.porovnejPodleVysky()), který Java umožňuje pomocí .., není pomocí :: možný

```

import java.util.Collections;
import java.util.List;

public class RazeniOsob {

    public static void main(String[] args) {
        List<Osoba> seznam = SeznamOsob.getSeznam();
        RadiciKriterium rk = new RadiciKriterium();

        Collections.sort(seznam, rk::porovnejPodleJmena);
        System.out.println("Jmeno: " + seznam);

        Collections.sort(seznam, RadiciKriterium::porovnejPodleVysky);
        System.out.println("Vyska: " + seznam);
    }
}

```

vypíše:

```

Jmeno: [[jmeno=Alois, vyska=180, pohlavi=MUZ], [jmeno=Anna, vyska=155, ►
pohlavi=ZENA], ...]
Vyska: [[jmeno=Anna, vyska=155, pohlavi=ZENA], [jmeno=Josef, vyska=160, ►
pohlavi=MUZ], ...]

```

## 10.4. Agregované operace (Aggregate Operations)

- provádí se nad kolekcemi (poli, I/O, ...)
- výrazné zjednodušení práce s kolekcemi
  - ◆ programu říkáme, co má dělat, nikoliv jak to má dělat
- umožňují automatický paralelismus
- základní myšlenka:
  - objekty uložené v kolekci potřebujeme zpracovat
  - typický způsob zpracování je:

```

pro všechny prvky kolekce
- aplikuj nějaké výběrové kriterium
-- prvky vyhovující kriteriu dále zpracuj

```

- ukázkový příklad na již známé kolekci Osoba

```

public static void main(String[] args) {
    List<Osoba> seznam = SeznamOsob.getSeznam();

    // klasicky zpusob vypisu

```

```

for (Osoba o : seznam) {
    if (o.getVyska() < 175) {
        System.out.print(o.getJmeno() + ", ");
    }
}

System.out.println();
// vypis pomocí agregovanych operaci
seznam
    .stream()
    .filter(o -> o.getVyska() < 175)
    .forEach(o -> System.out.print(o.getJmeno() + ", "));
}

```

vypíše:

```

Anna, Jana, Josef, Karel,
Anna, Jana, Josef, Karel,

```

■ pro zpracování se zavádějí dva pojmy (konstrukce)

1. *pipeline* (roura)

- název se nikde nepoužívá jako klíčové slovo
- je to sekvence agregovaných operací začínající zdrojem
  - ◆ v předchozím příkladě se *pipeline* skládá ze zdroje `stream()` a ze dvou agregovaných operací – `filter()` a `forEach()`
- typické části pipeline
  - a. **zdroj (source)** – nejčastěji kolekce, ale též pole nebo I/O kanál
    - ◆ zdroj se musí vyskytovat
    - ◆ produkuje `stream` – viz dále
  - b. **průběžné (intermediate) operace** – nejčastěji filtry
    - ◆ každá průběžná operace produkuje nový `stream`
      - to umožňuje jednoduše zřetězovat další operace
    - ◆ nemusí být přítomna žádná průběžná operace
      - to v případě, že chceme zpracovat všechny prvky původního `streamu`
  - c. **koncová (terminal) operace** – zakončuje činnost *pipeline* a vrací výsledek celého předchozího řetězu operací
    - ◆ výstupní hodnotou může být:
      - kolekce

- jednotlivý objekt
- primitivní datový typ (často suma, průměr nebo počet)
- void – v případě např. tisku (viz výše) – typicky po použití operace `forEach()`

## 2. *stream* (datovod)

- název se používá jako název metody, která vytváří počáteční zdroj
- je to posloupnost elementů (*pipeline* je posloupnost operací nad elementy)
  - ◆ není to kolekce – dopředu nealokuje paměť pro všechny elementy
- je zodpovědný za předávání dat v rámci *pipeline*
- lze jej vytvořit jako sekvenční nebo paralelní
- existují čtyři typy *streamu*:
  - ◆ `Stream<T>` – generický *stream* pro objekty
  - ◆ `IntStream` – *stream* pro primitivní datové prvky typu `int`
  - ◆ `LongStream` – *stream* pro primitivní datové prvky typu `long`
  - ◆ `DoubleStream` – *stream* pro primitivní datové prvky typu `double`

### ■ agregované operace a *streamy* nalezneme v

- `java.util.stream`
- `java.util.stream.Collectors`

## 10.4.1. Vytvoření *streamu*

### ■ z kolekcí používáme defaultní metody z rozhraní `java.util.Collection`:

- `stream()` – pro sekvenční *stream*
- `parallelStream()` – pro (potenciálně) paralelní *stream*

### Poznámka

Před používáním paralelního *streamu* je vhodné si přečíst lekci *Parallelism* z Java Tutorial.

### ■ z polí metody z knihovní třídy `java.util.Arrays`

- pro celé pole:
  - ◆ `stream(T[] array)` – pro pole objektů
  - ◆ `stream(int[] array)`, `stream(long[] array)`, `stream(double[] array)` – pro pole primitivních datových prvků
- pro výsek pole:

- ◆ stream(T[] array, int startInclusive, int endExclusive) – pro výsek z pole objektů
- ◆ dtto pro int, long a double
- vytvořené *streamy* jsou sekvenční
  - ◆ na paralelní je lze převést pomocí např. `java.util.stream.IntStream.parallel()`

## 10.4.2. Průběžné operace (*intermediate*)

- je jich několik – viz `java.util.stream.Stream<T>` např.:
  - `limit(long n)` – zpracuje pouze prvních `n` elementů, ostatní vynechá
  - `skip(long n)` – přeskočí prvních `n` elementů a zpracovává až ty následující
  - `map()`, `mapToInt()`, `mapToDouble()` – mapuje jeden *stream* na druhý (příklady viz dále)
- dále budou uvedeny příklady dvou nejčastějších operací

### 10.4.2.1. Filtrování streamu

- provádí se metodou `filter()`
  - jejím parametrem je buď lambda výraz, který musí vracet `boolean`, nebo reference na příslušnou metodu vracející `boolean`
- příklad použití

```
seznam
    .stream()
    .filter(o -> o.getVyska() < 175)
```

### 10.4.2.2. Řazení

- řazení metodou `java.util.stream.Stream<T>.sorted()` dává obě známé možnosti
  - `sorted()` – přirozené řazení
  - `sorted(Comparator<T> comparator)` – absolutní řazení
- příklad na seřazení seznamu s využitím komparátoru pro obrácené pořadí řazení
  - vysvětlení `collect(Collectors.toList())` viz dále

```
public static void main(String[] args) {
    List<Integer> seznam = new ArrayList<>(MAX);

    // naplnení
    for (int i = 1; i <= MAX; i++) {
        seznam.add(Integer.valueOf(i));
    }
}
```

```

        System.out.println(seznam);

        Comparator<Integer> vzestupne = Integer::compare;
        Comparator<Integer> reversed = vzestupne.reversed();

        List<Integer> serazenySeznam = seznam
            .stream()
            .sorted(reversed)
            .collect(Collectors.toList());

        System.out.println(serazenySeznam);
    }
}

```

vypíše:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

## 10.4.3. Koncové redukční agregované operace

- koncové operace (např. `average()`, `sum()`, `min()`, `max()`, `count()`, ...), které z původního *streamu* vytvoří jednu hodnotu
  - např. v `java.util.stream.IntStream`
  - speciálním případem je: `findFirst()` – vrátí první element kolekce
- před jejich použitím je někdy třeba převést *stream* na příslušný `IntStream`, `DoubleStream`, `LongStream`
  - to se provede metodou např. `mapToInt()` jejímž parametrem je reference (nebo lambda výraz) vracející `int` hodnotu
- příklad na výpočet průměrné výšky mužů

```

List<Osoba> seznam = SeznamOsob.getSeznam();

double prumernaVyska = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .mapToInt(Osoba::getVyska)
    .average()
    .getAsDouble();

System.out.println("Prumerna vyska muzu = " + prumernaVyska);
}

```

vypíše:

```
Prumerna vyska muzu = 170.0
```

- příklad na zjištění počtu mužů

```

long pocetMuzu = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .count();

System.out.println("Pocet muzu = " + pocetMuzu);

```

vypíše:

```
Pocet muzu = 3
```

■ příklad na nalezení výšky nejvyššího muže

```

int nejvyssiMuz = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .mapToInt(Osoba::getVyska)
    .max()
    .getAsInt();

System.out.println("Vyska nejvyssiho muze = " + nejvyssiMuz);

```

vypíše:

```
Vyska nejvyssiho muze = 180
```

■ volání metod `getAsDouble()` nebo `getAsInt()` je nutné, protože převádí vnitřní typ `Optional` (viz dále) na příslušný primitivní datový typ

### Poznámka

Pokud nám nestačí předpřipravené redukční operace, lze si napsat vlastní s využitím `java.util.stream.Stream<T>.reduce()`. Pokud bychom ale nepotřebovali paralelismus, pak by zřejmě přehlednějším způsobem bylo „klasické“ procházení kolekcí.

## 10.4.4. Koncové agregované operace vracející kolekce

■ potřebujeme-li jako návratovou hodnotu ze zpracování `streamu` získat kolekci, použijeme jako koncovou operaci metodu `collect()`:

- jejím parametrem může být:

- ◆ `Collectors.toList()` – seznam
- ◆ `Collectors.toSet()` – množina
- ◆ `Collectors.toCollection(TreeSet::new)` – množina `TreeSet`
  - zde je použita reference na bezparametrický konstruktor

■ příklad na získání seznamu mužů

```

List<Osoba> seznamMuzu = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .collect(Collectors.toList());
System.out.println("Seznam muzu = " + seznamMuzu);

```

vypíše:

```

Seznam muzu = [[jmeno=Josef, vyska=160, pohlavi=MUZ],
[jmeno=Karel, vyska=170, pohlavi=MUZ],
[jmeno=Alois, vyska=180, pohlavi=MUZ]]

```

### ■ příklad na získání množiny jmen mužů

```

Set<String> mnozinaJmenMuzu = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .map(o -> o.getJmeno())
    .collect(Collectors.toSet());

```

```
System.out.println("Mnozina jmen muzu = " + mnozinaJmenMuzu);
```

vypíše:

```
Mnozina jmen muzu = [Alois, Josef, Karel]
```

### ■ velmi užitečnou možností je použití metody `collect()` s parametrem `Collectors.groupingBy()`

- umožní vytvářet mapy podle zvoleného kriteria

### ■ příklad na vytvoření mapy osob s klíčem Pohlavi

```

Map<Osoba.Pohlavi, List<Osoba>> osobyPodlePohlavi = seznam
    .stream()
    .collect(Collectors.groupingBy(Osoba::getPohlavi));

```

```
System.out.println("Seznam osob podle pohlavi = " + osobyPodlePohlavi);
```

vypíše:

```

Seznam osob podle pohlavi =
{MUZ=[[jmeno=Josef, vyska=160, pohlavi=MUZ],
[jmeno=Karel, vyska=170, pohlavi=MUZ],
[jmeno=Alois, vyska=180, pohlavi=MUZ]],
ZENA=[[jmeno=Anna, vyska=155, pohlavi=ZENA],
[jmeno=Jana, vyska=165, pohlavi=ZENA],
[jmeno=Hana, vyska=175, pohlavi=ZENA]]}

```

### ■ příklad na mapy jmen osob s klíčem Pohlavi

```

Map<Osoba.Pohlavi, List<String>> jmenaPodlePohlavi = seznam
    .stream()

```

```

.collect(
    Collectors.groupingBy(
        Osoba::getPohlavi,
        Collectors.mapping(
            Osoba::getJmeno,
            Collectors.toList())));
}

System.out.println("Seznam jmen podle pohlavi = " + jmenaPodlePohlavi);

```

vypíše:

```
Seznam jmen podle pohlavi = {ZENA=[Anna, Jana, Hana],
                               MUZ=[Josef, Karel, Alois]}
```

- pomocí metody `collect()` s parametrem `Collectors.partitioningBy()` lze provést prahování
  - znamená to vytvořit mapu, jejímž klíčem bude `Boolean`, tj. nevyhovující versus vyhovující zvolenému kriteriu
- příklad na získání mapy osob s klíčem malí/velcí, kdy malí < 175

```

Map<Boolean, List<Osoba>> maliVelci = seznam
    .stream()
    .collect(Collectors.partitioningBy(o -> o.getVyska() >= 175));

System.out.println("Mali a velci = " + maliVelci);

```

vypíše:

```
Mali a velci = {false=[[jmeno=Anna, vyska=155, pohlavi=ZENA],
                      [jmeno=Jana, vyska=165, pohlavi=ZENA],
                      [jmeno=Josef, vyska=160, pohlavi=MUZ],
                      [jmeno=Karel, vyska=170, pohlavi=MUZ]],
               true=[[jmeno=Hana, vyska=175, pohlavi=ZENA],
                      [jmeno=Alois, vyska=180, pohlavi=MUZ]]}
```

- posledním užitečným parametrem metody `collect()` je `Collectors.joining()`
  - umožní spojit výstup do jednoho řetězce a ještě navíc zvolit vhodný oddělovač podřetězců
- příklad na získání řetězce se jmény všech mužů s oddělovačem „:“

```

String jmenaMuzu = seznam
    .stream()
    .filter(o -> o.getPohlavi() == Osoba.Pohlavi.MUZ)
    .map(Osoba::getJmeno)
    .collect(Collectors.joining(":"));

System.out.println("Jmena muzu = " + jmenaMuzu);

```

vypíše:

```
Jmena muzu = Josef: Karel: Alois
```

# 10.5. Třída Optional

- nabývá na důležitosti při používání agregovaných operací, kde je typické zřetězené volání:

```
List<Integer> serazenySeznam =  
seznam.stream().sorted(reversed).collect(Collectors.toList());
```

- jestliže jakákoli část výrazu vrátí null (ve významu „prázdný“), měla by být v následující zřetězené operaci vyhozena známá výjimka `java.lang.NullPointerException`
- bezpečný zápis předchozího výrazu by tedy měl být:

```
List<Integer> serazenySeznam = null;  
Stream<Integer> s1 = seznam.stream();  
if (s1 != null) {  
    Stream<Integer> s2 = s1.sorted(reversed);  
    if (s2 != null) {  
        serazenySeznam = s2.collect(Collectors.toList());  
    }  
}
```

což samozřejmě prodlužuje kód a výrazně snižuje čitelnost programu

- Java 8 možný výskyt null reference umožňuje řešit pomocí nové třídy `java.util.Optional<T>`
  - její verze pro primitivní datové typy `OptionalInt`, `OptionalLong` a `OptionalDouble`
- princip je jednoduchý – jedná se o obalovací třídu, která používá místo null speciální prázdnou hodnotu (`Optional.empty()`)
  - na tento princip je nabalen množství dalších metod, které dovolují např.:
    - ◆ nastavit prázdnou hodnotu – `empty()`
    - ◆ nastavit neprázdnou či prázdnou (pomocí null) hodnotu – `ofNullable()`
    - ◆ nastavit neprázdnou hodnotu – `of()`
    - ◆ testovat, zda je neprázdná hodnota – `isPresent()`
    - ◆ provést akci, pokud je neprázdná hodnota – `ifPresent()`
    - ◆ získat neprázdnou hodnotu nebo (v případě prázdné hodnoty) defaultní hodnotu – `orElse()`
- třída `Optional` (a její verze) není samospasitelný zázrak
  - měla by být použita jen pro referenční proměnné, které jsou volitelné, tj. nemusí být nastaveny
  - doporučované použití je jako návratová hodnota, kdy lze zajistit, že se nikdy nevrátí null
  - nejsou (zatím) obecně známé idiomy použití, viz též:
    - ◆ „*Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional!*“
    - ◆ [http://www.tutorialspoint.com/java8/java8\\_optional\\_class.htm](http://www.tutorialspoint.com/java8/java8_optional_class.htm)

## Poznámka

Třídu `Optional` nemusíme aktivně používat, ale musíme ji znát. Je totiž široce používána v agregovaných operacích. Viz např. dříve metody `getAsInt()` a `getAsDouble()` v „Koncové redukční agregované operace“.

### 10.5.1. Ukázka možného použití `Optional`

- třída `Predmet` má jako volitelné dva atributy

- `nazev` – může být zadán v konstruktoru
  - ◆ není-li zadán, je prázdný a metoda `getNazev()` s tím počítá a jednoduše (pomocí `orElse()`) vrací informaci `NENI_K_DISPOZICI`
    - takže navenek se jeví jako téměř (viz dále skutečný parametr konstruktoru) jako běžný `String`
- `znamka` – při vytvoření objektu je vždy prázdná (není parametr v konstruktoru)
  - ◆ je možné ji z vnějšku nastavit jako `OptionalInt`
  - ◆ vrací se jako `OptionalInt`, aby bylo vidět, jak lze s touto hodnotou dále zacházet
- přičemž metoda `toString()` vrací originální typy všech tří atributů

```
import java.util.Optional;
import java.util.OptionalInt;

public class Predmet {
    private final static String NENI_K_DISPOZICI = "N/A";

    private String zkratka;
    private Optional<String> nazev;
    private OptionalInt znamka;

    public Predmet() {
        this(null, Optional.empty());
    }

    public Predmet(String zkratka, Optional<String> nazev) {
        this.nazev = nazev;
        this.zkratka = zkratka;
        znamka = OptionalInt.empty();
    }

    public String getNazev() {
        return nazev.orElse(NENI_K_DISPOZICI);
    }

    public String getZkratka() {
        return zkratka;
    }
}
```

```

public OptionalInt getZnamka() {
    return znamka;
}

public void setZnamka(OptionalInt znamka) {
    this.znamka = znamka;
}

@Override
public String toString() {
    return "[zkratka=" + zkratka + ", nazev=" + nazev
           + ", znamka=" + znamka + "]";
}
}

```

■ třída Pouziti ukazuje tři různé případy použití

- prázdný předmět má všechny tři atributy prázdné
  - ◆ srovnajte přehlednost informace null u zkratky a N/A u názvu
  - ◆ u známky je rozhodně lepší testovat prázdnou hodnotu pomocí `isPresent()`, než si vymýšlet nějakou int hodnotu, která by prázdnou hodnotu představovala
    - nabízely by se např. hodnoty 0, -1, -`Integer.MAX_VALUE`

```

Predmet prazdny = new Predmet();
System.out.println("prazdny: " + prazdny);
System.out.println("prazdny.zkratka: " + prazdny.getZkratka());
System.out.println("prazdny.nazev: " + prazdny.getNazev());
if (prazdny.getZnamka().isPresent() == false) {
    System.out.println("prazdny.znamka: " + "neni nastavena");
}

```

vypíše:

```

prazdny: [zkratka=null, nazev=Optional.empty, znamka=OptionalInt.empty]
prazdny.zkratka: null
prazdny.nazev: N/A
prazdny.znamka: neni nastavena

```

- částečně vyplněný předmět (`bezNazvuOOP`) má vyplněnou pouze zkratku předmětu
  - ◆ pro nastavení skutečného parametru `nazev` v konstruktoru je použita metoda `Optional.ofNullable(null)`
    - ta má tu vlastnost, že v případě hodnoty null připraví prázdnou (`Optional.empty()`) hodnotu
  - ◆ pokusíme-li se získat hodnotu známky z její prázdné hodnoty (metodou `getAsInt()`), je vyhozena výjimka `java.util.NoSuchElementException: No value present`

```

Optional<String> bezNazvu = Optional.ofNullable(null);
Predmet bezNazvuOOP = new Predmet("KIV/OOP", bezNazvu);
System.out.println("bezNazvuOOP: " + bezNazvuOOP);
int znamka = bezNazvuOOP.getZnamka().getAsInt();

```

vypíše:

```

bezNazvuOOP: [zkratka=KIV/OOP, nazev(Optional.empty, ►
znamka=OptionalInt.empty]
Exception in thread "main" java.util.NoSuchElementException: No value present

```

- kompletně vyplněný předmět (`maNazevOOP`) má vyplněny všechny tři atributy
  - ◆ pro nastavení skutečného parametru `nazev` v konstruktoru je opět použita metoda `Optional.ofNullable()`, tentokrát však se skutečným názvem předmětu
  - ◆ známka je nastavena pomocí `OptionalInt.of(1)`, kdy skutečný parametr musí být platná `int` hodnota
  - ◆ ve výpisu je dobře vidět, že `Optional` je skutečně obalovací třída
    - získat skutečnou hodnotu známky je nutné pomocí `getAsInt()`
  - ◆ poslední příkaz ukazuje rozdíl použití `isPresent()` a `ifPresent()`
    - vyhneme se extrakci hodnoty z obalovací třídy

```

Optional<String> maNazev = Optional.ofNullable("Objektové orientované ►
programování");
Predmet maNazevOOP = new Predmet("KIV/OOP", maNazev);
maNazevOOP.setZnamka(OptionalInt.of(1));
System.out.println("maNazevOOP: " + maNazevOOP);
System.out.println("maNazevOOP.nazev: " + maNazevOOP.getNazev());
if (maNazevOOP.getZnamka().isPresent() == true) {
    System.out.println("maNazevOOP.znamka: "
                       + maNazevOOP.getZnamka().getAsInt());
}
maNazevOOP.getZnamka().ifPresent(System.out::println);

```

vypíše:

```

maNazevOOP: [zkratka=KIV/OOP, nazev=Optional[Objektové orientované ►
programování], znamka=OptionalInt[1]]
maNazevOOP.nazev: Objektové orientované programování
maNazevOOP.znamka: 1
1

```

## 10.5.2. Ukázka „skrytí“ `Optional` ve třídě `Predmet`

- předchozí příklad ukazoval různé přístupy k použití třídy `Optional`
- přesto, že byla třída `Optional` použita, měl uživatel třídy `Predmet` komplikace s nastavováním či manipulací atributů

- ukázka předchozího komplikovaného nastavování jména předmětu

```
Optional<String> maNazev = Optional.ofNullable("Objektove orientovane ►
programovani");
Predmet maNazevOOP = new Predmet("KIV/OOP", maNazev);
```

- ukázka předchozího komplikovaného získání známky z předmětu

```
if (maNazevOOP.getZnamka().isPresent() == true) {
    int znamka = maNazevOOP.getZnamka().getAsInt();
}
```

- v reálném případě by však bylo žádoucí **zcela** „odstínit“ budoucího uživatele třídy `Predmet` od komplikací s nastavováním či manipulací s typem `Optional`

- třída `Predmet` bude důsledně:

- používat typ `Optional` pro všechny atributy
- zabezpečovat všechny operace s `Optional` interně, takže se navenek zdá, že se pracuje s „běžnými“ datovými typy

```
package optional.lepsi;

import java.util.Optional;
import java.util.OptionalInt;

public class Predmet {
    private final static String NENI_K_DISPOZICI = "N/A";
    private final static int ZNAMKA_NENI_K_DISPOZICI = 0;

    private Optional<String> zkratka;
    private Optional<String> nazev;
    private OptionalInt znamka;

    public Predmet(String zkratka) {
        this(zkratka, "");
    }

    public Predmet(String zkratka, String nazev) {
        this.zkratka = transformString(zkratka);
        this.nazev = transformString(nazev);
        this.znamka = OptionalInt.empty();
    }

    public String getZkratka() {
        return zkratka.orElse(NENI_K_DISPOZICI);
    }

    public String getNazev() {
        return nazev.orElse(NENI_K_DISPOZICI);
    }
}
```

```

public int getZnamka() {
    return znamka.orElse(ZNAMKA_NENI_K_DISPOZICI);
}

public void setNazev(String nazev) {
    this.nazev = transformString(nazev);
}

public void setZnamka(int znamka) {
    this.znamka = transformInt(znamka);
}

@Override
public String toString() {
    return "[zkratka=" + getZkratka() + ", nazev=" + getNazev()
           + ", znamka=" + getZnamka() + "]";
}

private Optional<String> transformString(String str) {
    Optional<String> oStr = Optional.empty();
    if (str != null) {
        if (str.isBlank() == false) {
            oStr = Optional.of(str.trim());
        }
    }
    return oStr;
}

private OptionalInt transformInt(int cislo) {
    OptionalInt oCislo = OptionalInt.empty();
    if (cislo > ZNAMKA_NENI_K_DISPOZICI) {
        oCislo = OptionalInt.of(cislo);
    }
    return oCislo;
}
}

```

- metoda `toString()` volá metody `get...()`, ve kterých je pomocí `orElse()` důsledně vracena zvolená náhrada hodnoty „`empty`“
- metoda `Optional<String> transformString(String str)` zajišťuje nastavení `Optional<String>`:
  - ◆ na `Optional.empty()`, pokud je hodnota parametru `str` buď `null` nebo prázdný řetězec nebo řetězec jen s bílými znaky
  - ◆ na skutečnou hodnotu parametru `str` „obalenou“ do typu `Optional<String>`
- metoda `OptionalInt transformInt(int cislo)` zajišťuje nastavení `OptionalInt`:
  - ◆ na `OptionalInt.empty()`, pokud je hodnota parametru `cislo` buď 0 nebo záporné číslo
  - ◆ na skutečnou hodnotu parametru `cislo` „obalenou“ do typu `OptionalInt`

## ■ třída VylepsenePouziti ukazuje různé případy použití

- ukázka možnosti nastavení názvu předmětu pomocí použití null a současně výpis pomocí `toString()`

```
Predmet bezNazvuNull = new Predmet("KIV/OOP", null);
System.out.println("bezNazvuNull: " + bezNazvuNull);
```

◆ vypíše:

```
bezNazvuNull: [zkratka=KIV/OOP, nazev=N/A, znamka=0]
```

- ukázka možnosti nastavení názvu předmětu pomocí použití řetězce s bílými znaky a následně nastavení názvu předmětu a použití „getrů“

```
Predmet bezNazvuEmpty = new Predmet("KIV/OOP", " ");
System.out.println("bezNazvuEmpty: " + bezNazvuEmpty);
System.out.println("bezNazvuEmpty.nazev: " + bezNazvuEmpty.getNazev());
System.out.println("bezNazvuEmpty.znamka: " + bezNazvuEmpty.getZnamka());
bezNazvuEmpty.setNazev("Objektove orientovane programovani");
System.out.println("bezNazvuEmpty: " + bezNazvuEmpty);
System.out.println("bezNazvuEmpty.nazev: " + bezNazvuEmpty.getNazev());
```

◆ vypíše:

```
bezNazvuEmpty: [zkratka=KIV/OOP, nazev=N/A, znamka=0]
bezNazvuEmpty.nazev: N/A
bezNazvuEmpty.znamka: 0
bezNazvuEmpty: [zkratka=KIV/OOP, nazev=Objektove orientovane programovani, ►
znamka=0]
bezNazvuEmpty.nazev: Objektove orientovane programovani
```

- ukázka možnosti nastavení názvu předmětu reálnou hodnotou a následně použití „setru“ pro nastavení známky

```
Predmet maNazevOOP = new Predmet("KIV/OOP", "Objektove orientovane ►
programovani");
maNazevOOP.setZnamka(-1);
System.out.println("maNazevOOP: " + maNazevOOP);
maNazevOOP.setZnamka(2);
System.out.println("maNazevOOP.zkratka: " + maNazevOOP.getZkratka());
System.out.println("maNazevOOP.nazev: " + maNazevOOP.getNazev());
System.out.println("maNazevOOP.znamka: " + maNazevOOP.getZnamka());
```

◆ vypíše:

```
maNazevOOP: [zkratka=KIV/OOP, nazev=Objektove orientovane ►
programovani, znamka=0]
maNazevOOP.zkratka: KIV/OOP
maNazevOOP.nazev: Objektove orientovane programovani
maNazevOOP.znamka: 2
```

# Kapitola 11. Reflexe, invokace a anotace v Javě

## 11.1. Reflexe

- třídy a datové typy z `java.lang.reflect`
- dovolují o prvcích třídy získat z jejich signatur veškerou informaci
  - prvky třídy jsou míněny:
    - ◆ atributy (zde *field*), konstruktory, metody, anotace
  - informací je míněno:
    - ◆ použité modifikátory – `public`, `static`, `final`, ...
    - ◆ typy formálních parametrů a návratových hodnot
- zkoumaná třída je předána pouze jménem, takže vůbec nemusíme znát její zdrojový kód ani kontrakt
  - využíváme metodu `Class<T> Class.forName(String jménoBalíku.jménoTřídy)`
- reflexe se používá:
  - ve vývojových nástrojích
  - pro speciální účely, typicky testování
- příklad `UkazkovaTrida` obsahuje všechny typické základní prvky třídy (jejich logika není podstatná)

```
package reflexe;

public class UkazkovaTrida {
    public static final String STATICKA_KONSTANTA = "obsah staticke konstanty";
    private static int statickaPromenna = 3;

    private final double INSTANCNI_KONSTANTA = 3.14;
    private boolean instancniPromenna;

    public UkazkovaTrida() {
        this(true);
    }

    private UkazkovaTrida(boolean nastaveni) {
        this.instancniPromenna = nastaveni;
    }

    public static void statickaMetodaBezParametru() {
        System.out.println("STATICKA_KONSTANTA = " + STATICKA_KONSTANTA);
    }

    public static int statickaMetodaSParametrem(int nasobek) {
        statickaPromenna = statickaPromenna * nasobek;
    }
}
```

```

        System.out.println("statickaPromenna = " + statickaPromenna);
        return statickaPromenna;
    }

    public boolean instancniMetodaBezParametru() {
        instancniPromenna = instancniPromenna ? false : true;
        System.out.println("instancniPromenna = " + instancniPromenna);
        return instancniPromenna;
    }

    void instancniMetodaSParametrem(int nasobek) {
        System.out.println("INSTANCNI_KONSTANTA = "
                           + INSTANCNI_KONSTANTA * nasobek);
    }

    // jen pro ověření funkčnosti
    public static void main(String[] args) {
        UkazkovaTrida utTrue = new UkazkovaTrida(false);
        utTrue.instancniMetodaBezParametru();
        utTrue.instancniMetodaSParametrem(1);

        UkazkovaTrida utFalse = new UkazkovaTrida();
        utFalse.instancniMetodaBezParametru();

        statickaMetodaBezParametru();
        statickaMetodaSParametrem(1);
    }
}

```

po spuštění vypíše:

```

instancniPromenna = true
INSTANCNI_KONSTANTA = 3.14
instancniPromenna = false
STATICKA_KONSTANTA = obsah staticke konstanty
statickaPromenna = 3

```

■ třída `Reflexe` zkoumá nejjednodušším způsobem obsah třídy `reflexe.UkazkovaTrida`

## Poznámka

Všechny výjimky jsou pro jednoduchost řešeny deklarací.

```

package reflexe;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Reflexe {

    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> zkoumanaClass = Class.forName("reflexe.UkazkovaTrida");
    }
}

```

```

Field[] atributy = zkoumanaClass.getDeclaredFields();
Constructor<?>[] konstruktory = zkoumanaClass.getDeclaredConstructors();
Method[] metody = zkoumanaClass.getDeclaredMethods();

System.out.println("Atributy:");
for (Field atribut : atributy) {
    System.out.println(atribut);
}

System.out.println("\nKonstruktory:");
for (Constructor<?> konstruktor : konstruktory) {
    System.out.println(konstruktor);
}

System.out.println("\nMetody:");
for (Method metoda : metody) {
    System.out.println(metoda);
}
}
}

```

po spuštění vypíše:

```

Atributy:
public static final java.lang.String reflexe.UkazkovaTrida.STATICKA_KONSTANTA
private static int reflexe.UkazkovaTrida.statickaPromenna
private final double reflexe.UkazkovaTrida.INSTANCEKONSTANTA
private boolean reflexe.UkazkovaTrida.instancniPromenna

Konstruktory:
public reflexe.UkazkovaTrida()
private reflexe.UkazkovaTrida(boolean)

Metody:
public static void reflexe.UkazkovaTrida.statickaMetodaBezParametru()
public static int reflexe.UkazkovaTrida.statickaMetodaSParametrem(int)
public boolean reflexe.UkazkovaTrida.instancniMetodaBezParametru()
void reflexe.UkazkovaTrida.instancniMetodaSParametrem(int)
public static void reflexe.UkazkovaTrida.main(java.lang.String[])

```

- pro podrobnější zkoumání je ve třídách `Field`, `Constructor` a `Method` k dispozici množství metod pro sofistikovanou analýzu
- modifikátory všech tří uvedených se získají metodou `int getModifiers()`
  - vracené int číslo lze analyzovat pomocí statických metod třídy `Modifier` – `boolean isXYZ(int modifikátory)`
    - ◆ `isAbstract(int mod)`, `isFinal(int mod)`
    - ◆ `isPrivate(int mod)`, `isProtected(int mod)`, `isPublic(int mod)`
    - ◆ `isStatic(int mod)`

- příklad použití viz dále

## 11.2. Invokace

- je to nepřímé (anonymní) vyvolání konstruktorů nebo metod, které neznáme jménem nebo neznáme jejich parametry
  - informaci o nich jsme získali díky reflexi
- pro jednodušší způsob invokace využíváme metody `invoke()` ze třídy `Method`
  - složitější způsob je možný prostředky balíku `java.lang.invoke` (zde nebude použito)
- při vyvolávání metod je třeba určit, zda jsou **statické**
  - pak je vyvolání jednoduché – první parametr metody `invoke()` je `null`
- v případě **instančních** metod musíme nejdříve vytvořit instanci pomocí volání konstruktoru metodou `newInstance(Object... initargs)` třídy `Constructor`
  - pro jednodušší případy stačí takto vytvořenou instanci typovat na třídu `Object`
  - referenční proměnná na vytvořenou instanci je pak prvním parametrem metody `invoke()`
- v příkladu budou vyvolány všechny metody (kromě `main()`) z výše uvedené třídy `UkazkovaTrida`

```
package reflexe;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class AnonymniVolani {

    public static void main(String[] args) throws Exception {
        Class<?> zkoumanaClass = Class.forName("reflexe.UkazkovaTrida");
        Method[] metody = zkoumanaClass.getDeclaredMethods();

        System.out.println("Staticke metody");
        for (Method metoda : metody) {
            int modifikatory = metoda.getModifiers();
            if (Modifier.isStatic(modifikatory) == true) {
                if (metoda.getName().equals("main")) {
                    // nechci spoustet main()
                    continue;
                }
                System.out.println("Volani metody: " + metoda.getName());
                if (metoda.getParameterTypes().length == 0) {
                    // staticka bez parametru
                    metoda.invoke(null);
                }
                else {
                    // staticka s parametry
                    metoda.invoke(null, 2);
                }
            }
        }
    }
}
```

```
        }
    }
}

System.out.println("\nInstancni metody");
Constructor<?> konstruktor = zkoumanaClass.getDeclaredConstructor();
Object instance = konstruktor.newInstance();
for (Method metoda : metody) {
    int modifikatory = metoda.getModifiers();
    if (Modifier.isStatic(modifikatory) == false) {
        System.out.println("Volani metody: " + metoda.getName());
        if (metoda.getParameterTypes().length == 0) {
            // instancni bez parametru
            metoda.invoke(instance);
        }
        else {
            // instancni s parametry
            metoda.invoke(instance, 2);
        }
    }
}
```

po spuštění vypíše:

```
Staticke metody
Volani metody: statickaMetodaBezParametru
STATICKA_KONSTANTA = obsah staticke konstanty
Volani metody: statickaMetodaSParametrem
statickaPromenna = 6

Instancni metody
Volani metody: instancniMetodaBezParametru
instancniPromenna = false
Volani metody: instancniMetodaSParametrem
INSTANCNI_KONSTANTA = 6.28
```

### **11.3. Anotace**

### **11.3.1. Úvodní informace**

- nová konstrukce jazyka od 5.0
  - občas se označují termínem **metadata** – dávají datům ve zdrojovém kódu přidaný význam
    - přidání anotací neovlivňuje chování programu (pokud anotace nechce využívat)
  - značky vkládané do zdrojového kódu a určené pro různé nástroje, které s programem pracují
  - etapy zpracování/využití anotací

- pouze ve zdrojovém kódu – javadoc, javac
- přenáší se do .class souboru a
  - ◆ slouží pro instalaci aplikace
  - ◆ využijí se za běhu programu
- anotace jsou uvozeny znakem @
  - tentýž znak slouží pro prvky JavaDoc komentáře (@param, @returns, ...) – to nejsou anotace
  - anotace nejsou v komentářových závorkách
- anotace jsou typem modifikátorů – umísťují se před deklarace
  - podle konvencí jsou před všemi ostatními modifikátory
- anotace lze použít v podstatě na každý element programu – tam, kde lze použít nějaký modifikátor
  - třída
  - rozhraní
  - atribut
  - konstruktor
  - metoda
  - lokální proměnná
  - balík – zde se anotace zapisují do speciálního souboru package-info.java
- anotace mohou mít parametry – příklad používá známou anotaci @Test z JUnit testů
  - @Test
  - @Test (expected=IllegalArgumentException.class)
  - @Test (timeout=100)
- anotace se překládají do .class souborů

### 11.3.2. Anotace z java.lang

- @Deprecated
  - označený prvek (nejčastěji metoda nebo třída) je považován za „zavržený“ a neměl by se používat
    - ◆ v programu / v knihovně zůstává pouze pro udržení zpětné kompatibility
  - překladač je povinen vydat *warning* u každého použití takto označeného prvku
  - příklad java.lang.String.getBytes()

```

public class AnotaceZJavaAPI {
    // @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        byte[] pole = new byte[10];
        "ahoj".getBytes(0, 2, pole, 0);
    }
}

```

po překladu vypíše:

```

Note: AnotaceZJavaAPI.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

```

- pokud musíme použít *deprecated* prvek, lze *warning* potlačit použitím anotace

```
@SuppressWarnings ("deprecation")
```

#### ■ @Override

- bezparametrická anotace
- označuje metodu, která má překrýt stejnou metodu z rodičovské třídy
  - ◆ pokud se tak nestane, např. anotací označená metoda se v potomkovi pouze přetěžuje, překladač ohlásí chybu
- použití viz dříve

#### ■ @SuppressWarnings

- anotace s parametrem, kterým je seznam čárkami oddělených textových řetězců specifikující varování, která má překladač potlačit
- pokud je varování potlačeno, nedá se znova (v téže třídě) nastavit
  - ◆ např. používáme-li `@SuppressWarnings` pouze pro jednu metodu, uvedeme jej bezprostředně před deklarací této metody, nikoliv před deklarací třídy

#### ■ @SafeVarargs

- bezparametrická anotace
- ujištění, že metoda používající proměnný počet parametrů neobsahuje potenciálně nebezpečné operace

### 11.3.3. Metaanotace

- pro vytváření anotací potřebujeme znát **metaanotace** z `java.lang.annotation`
  - nové anotace jsou deklarovány pomocí nich
- `@Documented`

- bezparametrická metaanotace
- zajišťuje, že jí označená anotace (např. `@Deprecated`) bude uvedena v JavaDoc dokumentaci u prvků označených `@Deprecated`
  - ◆ tj. označení „prosákne“ do JavaDoc dokumentace

#### ■ `@Retention`

- jednoparametrická metaanotace
- určuje, jak dlouho bude uchována informace z použité anotace
  - ◆ to souvisí s tím, kdy bude anotace využita
- má tři možné hodnoty svého jediného parametru `@Retention(value=hodnota)`
  - ◆ SOURCE – pouze pro programy pracující se zdrojovým kódem
    - do `.class` se nedostane
    - příklad použití u anotace: `@Override`
  - ◆ CLASS – je využitelná např. jen pro instalaci
    - je uložena v `.class` souboru
    - do běžící instance se nedostane
  - ◆ RUNTIME – lze ji využít za běhu programu nejčastěji pomocí reflexe
    - je uložena v `.class` souboru
    - virtuální stroj ji zavede do paměti jako součást třídy
    - příklad použití u anotace: `@Deprecated`
- tyto konstanty jsou uvedeny ve výčtovém typu `java.lang.annotation.RetentionPolicy`
- hodnotu je možno zapsat dvěma způsoby:

```
@Retention(value=RUNTIME)
```

nebo

```
@Retention(RUNTIME)
```

#### ■ `@Target`

- jednoparametrická metaanotace
- určuje, jaký prvek může být anotací ve své deklaraci označen
- možné hodnoty jediného parametru `@Target(value=hodnota)`
  - ◆ PACKAGE – balík

- ◆ TYPE – třída, rozhraní, výčtový typ a jiná anotace
  - ◆ FIELD – atribut
  - ◆ CONSTRUCTOR – konstruktor
  - ◆ METHOD – metoda
  - ◆ PARAMETER – formální parametr metody (nemá praktický význam)
  - ◆ LOCAL\_VARIABLE – lokální proměnná (nemá praktický význam)
  - ◆ ANNOTATION\_TYPE – vše výše uvedené dohromady
- tyto konstanty jsou uvedeny ve výčtovém typu `java.lang.annotation ElementType`
  - chceme-li použít vytvářenou anotaci pro všechny prvky, `@Target` vůbec neuvedeme
  - hodnotu lze opět zapsat dvěma způsoby – s a bez `value`
  - chceme-li výčet prvků, použijeme např.:

```
@Target (value={CONSTRUCTOR, FIELD, METHOD, PARAMETER, TYPE})
```

nebo ve stejném významu:

```
@Target ({CONSTRUCTOR, FIELD, METHOD, PARAMETER, TYPE})
```

## 11.3.4. Využití anotací v programu pomocí reflexe

- využívá se metod z `java.lang.reflect.AnnotatedElement`, které jsou děděny do Method atp.
  - pro jednoduché případy se značkovací anotací stačí pouze `isAnnotationPresent()`
- třída `AnotaceZJavaAPI` používá anotaci `@Deprecated`, která je využitená i za běhu programu
  - označuje jí metodu `prevod()`

```
package anotace;

import java.util.Arrays;

public class AnotaceZJavaAPI {

    // @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        byte[] pole = prevod();
        System.out.println(Arrays.toString(pole));
    }

    @Deprecated
    public static byte[] prevod() {
        byte[] pole = new byte[10];
        "ahoj".getBytes(0, 2, pole, 0);
    }
}
```

```
        return pole;
    }
}
```

- při anonymním spouštění lze zjistit, zda je metoda označena jako `@Deprecated`, a rozhodnout o jejím případném spuštění

## Poznámka

Výjimky jsou pro jednoduchost opět řešeny deklarací.

```
package anotace;

import java.lang.reflect.Method;
import java.util.Arrays;

public class ReflexeAnotace {

    public static void main(String[] args) throws Exception {
        Class<?> zkoumanaClass = Class.forName("anotace.AnotaceZJavaAPI");
        Method[] metody = zkoumanaClass.getDeclaredMethods();

        for (Method metoda : metody) {
            if (metoda.isAnnotationPresent(Deprecated.class)) {
                System.out.println("Metoda " + metoda.getName()
                    + "() je deprecaded, ale spustim ji");
                byte[] pole = (byte[]) metoda.invoke(null);
                System.out.println(Arrays.toString(pole));
            }
        }
    }
}
```

po spuštění vypíše:

```
Metoda prevod() je deprecaded, ale spustim ji
[97, 104, 0, 0, 0, 0, 0, 0, 0]
```

## 11.3.5. Praktický příklad využití anotací – JUnit testy

- JUnit testy využívají pro označení testovací metody anotaci `@Test`
- ta je nadefinovaná (přibližně) jako:

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface Test { }
```

- známé použití testů

- mezi testovacími metodami je `beznaMetoda()` s podobnou signaturou, ale bez anotace `@Test`

- nebude při spouštění testů brána v úvahu

```
package anotace;

import org.junit.Test;
import static org.junit.Assert.*;

public class TestovaciTrida {
    @Test
    public void testMetodaSpravne() {
        System.out.println("Testovaci metoda - spravne");
        assertEquals("Spravne - projde", true, true);
    }

    @Test
    public void testMetodaChybne() {
        System.out.println("Testovaci metoda - chybne");
        assertEquals("Chybne - neprojde", false, true);
    }

    // zde záměrně chybí @Test
    public void beznaMetoda() {
        System.out.println("Bezna metoda bez anotace @Test");
        assertEquals("Chybne - neprojde", false, true);
    }
}
```

## ■ princip spouštění testovacích metod ve třídě

```
package anotace;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import org.junit.Test;

public class JUnitTestySpusteniPrincip {

    public static void main(String[] args) throws Exception {
        int proslo = 0;
        int chybne = 0;
        Class<?> zkoumanaClass = Class.forName("anotace.TestovaciTrida");
        Method[] metody = zkoumanaClass.getDeclaredMethods();

        Constructor<?> konstruktor = zkoumanaClass.getDeclaredConstructor();
        Object instance = konstruktor.newInstance();
        for (Method metoda : metody) {
            if (metoda.isAnnotationPresent(Test.class)) {
                try {
                    metoda.invoke(instance);
                    proslo++;
                } catch (Exception ex) {
                    System.out.println("Test metody " + metoda.getName()
                        + " selhal z důvodu\n" + ex.getCause());
                    chybne++;
                }
            }
        }
    }
}
```

```

        chybne++;
    }
}
System.out.println("Proslo: " + proslo + ", selhalo: " + chybne);
}
}

```

po spuštění vypíše:

```

Testovaci metoda - spravne
Testovaci metoda - chybne
Test metody testMetodaChybne() selhal z duvodu
java.lang.AssertionError: Chybne - neprojde expected:<false> but was:<true>
Proslo: 1, selhalo: 1

```

## 11.3.6. Vytváření vlastních anotací

- deklarace anotací se podobají deklaracím rozhraní
- obecné schéma je:

```

metaanotace
modifikátory @interface JménoAnotace
{
    deklarace parametru;
    . . .
    deklarace parametru;
}

```

- metaanotace jsou dříve uvedené metaanotace, typicky `@Documented` a `@Retention`
  - ◆ `@Target` se většinou neudává a pak je anotaci možné použít pro označení libovolného prvku
  - ◆ není nutné uvádět ani zbývající, ale je to vhodné
- jako modifikátor přístupového práva je nejčastěji `public` – může být i `public abstract`
- deklarace parametrů (tj. při použití anotace její skutečné parametry) jsou nepovinné
  - ◆ pokud jsou použity vypadají buď

```
typ jménoParametru();
```

nebo

```
typ jménoParametru() default implicitníHodnota;
```

- ◆ typ parametru může být jen:

- primitivní datový typ (`int, double, ...`)
- `String`

- Class
- výčtový typ
- jiná anotace
- pole výše uvedených typů

## Varování

I když parametr vypadá při deklaraci jako metoda, při použití anotace se jako její skutečný parametr zadává bez kulatých závorek.

### 11.3.6.1. Bezparametrická (značkovací) anotace

- nemá deklarace parametrů
- typický vzhled:

```
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface BezparametrickaAnotace { }
```

- zkrácený zápis se stejnou funkčností:

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;

@Documented
@Retention(RUNTIME)
public @interface BezparametrickaAnotace { }
```

- použití viz výše u ukázky JUnit testů

### 11.3.6.2. Anotace s jedním parametrem

- anotaci lze nastavit její parametr tak, že jeho jméno i s jeho hodnotou uvedeme v kulatých závorkách, např. @Test(timeout=100)
- používáme-li v anotaci jen jeden parametr, nazveme jej `value`
  - to nám při použití umožní napsat dvojici `value=hodnota`, ale jen `hodnota`
  - viz použití u metaanotací u bezparametrické anotace

## Varování

Hodnota parametru nesmí být null.

- typický vzhled:

### Poznámka

Typ `String` není podmínkou, může to být kterýkoliv z povolených typů.

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;

@Documented
@Retention(RUNTIME)
public @interface AnotaceJedenParametr {
    String value();
}
```

- příklad jednoparametrické anotace s defaultní hodnotou

- jako default hodnota je použit řetězec "anotace metody", protože lze předpokládat, že anotovaných metod bude nejvíce – viz dále použití
- pak při použití anotace není vůbec nutné uvádět parametr

```
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;

@Documented
@Retention(RUNTIME)
public @interface AnotaceJedenParametrDefault {
    String value() default "anotace metody";
}
```

- příklad třídy anotované oběma typy jednoparametrických anotací

- u metod `getCislo()` a `setCislo()` má různě zapsaná anotace stejný význam – viz výpis

```
package anotace;

@AnotaceJedenParametr("anotace tridy")
public class PouzitiJednoparametrickaAnotace {

    @AnotaceJedenParametr("anotace atributu")
    public int cislo;

    @AnotaceJedenParametr("anotace konstruktoru")
    public PouzitiJednoparametrickaAnotace(int cislo) {
        this.cislo = cislo;
    }
}
```

```

@AnotaceJedenParametrDefault
public int getCislo() {
    return cislo;
}

@AnotaceJedenParametrDefault(value="anotace metody")
public void setCislo(int cislo) {
    this.cislo = cislo;
}
}

```

## ■ výpis anotací

```

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Arrays;

public class VypisAnotaci {

    public static void main(String[] args) throws Exception {
        Class<?> zkoumanaClass =
            Class.forName("anotace.PouzitiJednoparametrickaAnotace");
        System.out.println("Trida : "
            + Arrays.toString(zkoumanaClass.getAnnotations()));

        System.out.println("\nAtributy:");
        for (Field atr : zkoumanaClass.getDeclaredFields()) {
            if (atr.getAnnotations().length > 0) {
                System.out.println(atr.getName() + ": "
                    + Arrays.toString(atr.getAnnotations()));
            }
        }

        System.out.println("\nKonstruktory:");
        for (Constructor<?> con : zkoumanaClass.getDeclaredConstructors()) {
            if (con.getAnnotations().length > 0) {
                System.out.println(Arrays.toString(con.getAnnotations()));
            }
        }

        System.out.println("\nMetody:");
        for (Method met : zkoumanaClass.getDeclaredMethods()) {
            if (met.getAnnotations().length > 0) {
                System.out.println(met.getName() + ": "
                    + Arrays.toString(met.getAnnotations()));
            }
        }
    }
}

```

po spuštění vypíše:

```
Trida: [@anotace.AnotaceJedenParametr(value=anotace tridy)]
```

Atributy:

```
cislo: [@anotace.AnotaceJedenParametr(value=anotace atributu)]
```

Konstruktory:

```
[@anotace.AnotaceJedenParametr(value=anotace konstruktoru)]
```

Metody:

```
getCislo: [@anotace.AnotaceJedenParametrDefault(value=anotace metody)]
```

```
setCislo: [@anotace.AnotaceJedenParametrDefault(value=anotace metody)]
```

### 11.3.6.3. Anotace s více parametry

- je podobná předchozí jednoparametrické anotaci, ale použijeme více deklarových parametrů (samořejmě vhodně pojmenovaných)
- příklad bude ukazovat anotaci `@PodminkySpusteniTestu`, která bude použita pro doplnění informace v JUnit testech
  - touto anotací budeme chtít navíc (tj. kromě původního spuštění JUnit testu) zajistit
    - ◆ vynucení pořadí prováděných testů – pokud výsledky nějakého testu závisejí na předchozích testech
      - to je většinou chyba v návrhu testů, ale někdy se vynucení pořadí nelze vyvarovat
    - ◆ rozlišení, zda budou testy spuštěny lokálně na desktopu nebo na validátoru
      - validátor může např. některé testy vynechat, protože jsou časově náročné atp.
      - použijeme nový výčtový typ (`NIKDE` má význam zejména pro ladění)

```
/**  
 * Definuje místo, kde bude test spuštěn  
 */  
public enum MistoSpusteni {  
    DESKTOP, VALIDATOR, NIKDE;  
}
```

- zdrojový kód anotace

- typ parametru `misto()` musí být pole, aby bylo možno v default deklaraci použít dvojici `DESKTOP` a `VALIDATOR`

#### Poznámka

Toto je použito pro ukázku, jak se pracuje s více hodnotami u jednoho prvku. Mnohem jednoduší (viz dále složitosti v programu), by bylo zavedení další hodnoty výčtového typu `MistoSpusteni.VSUNE`.

```
import java.lang.annotation.Documented;  
import java.lang.annotation.Retention;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```

@Documented
@Retention(RUNTIME)
@Target(METHOD)
public @interface PodminkySpusteniTestu {
    /**
     * Definuje pořadí v jakém budou jednotlivé testy spouštěny
     * pořadí jsou přirozená čísla od 1 (=spuštěno vždy jako první)
     * default hodnota 0 znamená, že na pořadí nezáleží,
     * ale tento test bude spuštěn až pro proběhnutí
     * všech testů se zadáným pořadím
     */
    int poradi() default 0;

    /**
     * Definuje, kde bude test spuštěn
     * hodnota MistoSpousteni.NIKDE má význam zejména pro ladění
     */
    MistoSpousteni[] misto() default
        {MistoSpousteni.DESKTOP, MistoSpousteni.VALIDATOR};
}

```

## Poznámka

Toto je ukázka netriviálního praktického příkladu. Je třeba si ale uvědomit, že anotace mají mnohem více sofistikovaných možností, které zde pro jednoduchost nebudou uváděny.

- testovaná třída Osoba je velmi jednoduchá a na principu anotací vůbec nezávisí

```

package anotace;

/**
 * Ukázka testované třídy, kdy záleží na pořadí testů
 */
public class Osoba {

    /** Počet doposud vytvořených instancí */
    private static int pocet = 0;

    /** pořadí kolikátá byla daná instance vytvořena v rámci třídy */
    private final int PORADI = ++pocet;

    /** Název sestávající z názvu třídy a PORADI instance */
    private String nazev = null;

    /** pohlavi "muz" nebo "zena" */
    private String pohlavi = null;

    public Osoba(String pohlavi) {
        this.nazev = this.getClass().getSimpleName() + "_" + PORADI;
        this.pohlavi = pohlavi;
    }
}

```

```

public String getPohlavi() {
    return pohlavi;
}

@Override
public String toString() {
    return nazev;
}
}

```

- třída testů OsobaTest využívá známé možnosti JUnit testů a navíc možnosti anotování pomocí @PodminkySpusteniTestu

- testy budou spouštěny v pořadí:

- ◆ testToString()
- ◆ testZmenaNazvu()
- ◆ testPohlavi()

- na DESKTOP budou spouštěny testy:

- ◆ testToString() – defaultní nastavení
- ◆ testZmenaNazvu()

- na VALIDATOR budou spouštěny testy:

- ◆ testToString() – defaultní nastavení
- ◆ testPohlavi()

```

package anotace;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class OsobaTest {

    @PodminkySpusteniTestu(poradi=0,misto=MistoSpousteni.VALIDATOR)
    @Test
    public void testPohlavi() {
        Osoba o = new Osoba("zena");
        assertEquals("Chybne pohlavi", "zena", o.getPohlavi());
    }

    @PodminkySpusteniTestu(poradi=1)
    @Test
    public void testToString() {
        Osoba o = new Osoba("zena");
        assertEquals("Chybný název: ", "Osoba_1", o.toString());
    }
}

```

```

@PodminkySpusteniTestu(poradi=2,misto=MistoSpousteni.DESKTOP)
@Test
public void testZmenaNazvu() {
    Osoba o = new Osoba("zena");
    assertEquals("Chybný název: ", "Osoba_2", o.toString());
}
}

```

## 11.3.7. Použití parametrů anotací

### 11.3.7.1. Metody z rozhraní `java.lang.reflect.AnnotatedElement`

- toto rozhraní implementují všechny třídy známé z reflexe, tj. `Field`, `Constructor`, `Method`, atd.
- to znamená, že se k informacím o anotacích dostáváme podobným způsobem, jako k informacím o ty- pech, modifikátorech apod.
- v případě, že známe název anotace (tj. téměř vždy), využíváme následujících metod:

- `boolean isAnnotationPresent(Class<? extends Annotation>)` např.:

```
metoda.isAnnotationPresent(PodminkySpusteniTestu.class)
```

- `<T extends Annotation> T getAnnotation(Class<T> annotationClass)` např.:

```
PodminkySpusteniTestu pst =
    metoda.getAnnotation(PodminkySpusteniTestu.class);
```

- pokud máme referenci na instanci konkrétní anotace (zde `pst`), můžeme nadní volat metody v ní deklarované, abychom získali konkrétní hodnoty parametrů, např.:

```
int index = pst.poradi();
```

nebo

```
MistoSpousteni[] ms = pst.misto();
```

### 11.3.7.2. Dokončení příkladu s upřesněním JUnit testů

- třída `JUnitTestySpusteniOsoba` využívá parametrů anotace `@PodminkySpusteniTestu` dvěma způsoby:
  - nejdříve použije prvek `poradi`, pomocí něhož seřadí metody v požadovaném pořadí spouštění
  - pak použije prvek `misto`, pomocí něhož spouští seřazené metody buď na `DESKTOP` nebo na `VALIDATOR`

#### Poznámka

Spouštění na obou místech v jednom programu nemá praktický význam. Díky němu dojde k selhání testu. To ovšem ale na druhé straně ukazuje, že JUnit testy fungují dle předpokladu.

```

package anotace;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import org.junit.Test;

public class JUnitTestySpusteniOsoba {

    private static Method[] serazeniMetodPodlePoradi(Method[] metody) {
        Method[] poradi = new Method[metody.length];

        // zarazeni metod s poradi != 0
        int nejvyssiIndex = 0;
        for (Method met : metody) {
            if (met.isAnnotationPresent(PodminkySpusteniTestu.class)) {
                PodminkySpusteniTestu pst =
                    met.getAnnotation(PodminkySpusteniTestu.class);
                int index = pst.poradi() - 1;
                if (index >= 0) {
                    poradi[index] = met;
                    if (nejvyssiIndex < index) {
                        nejvyssiIndex = index;
                    }
                }
            }
        }

        // zarazeni metod s poradi == 0 za vsechny jiz zarazene
        for (Method met : metody) {
            if (met.isAnnotationPresent(PodminkySpusteniTestu.class)) {
                PodminkySpusteniTestu pst =
                    met.getAnnotation(PodminkySpusteniTestu.class);
                int index = pst.poradi();
                if (index == 0) {
                    nejvyssiIndex++;
                    poradi[nejvyssiIndex] = met;
                }
            }
        }
    }

    return poradi;
}

private static void spusteniTestu(Method[] metody, MistoSpousteni misto,
                                  Object instance) {
    System.out.println("Spoustim testy na: " + misto.name());
    int proslo = 0;
    int chybne = 0;
    for (Method met : metody) {
        if (met.isAnnotationPresent(Test.class) == false) {
            continue;
        }
    }
}

```

```

        if (met.isAnnotationPresent(PodminkySpusteniTestu.class) == false) {
            continue;
        }
        PodminkySpusteniTestu pst =
                met.getAnnotation(PodminkySpusteniTestu.class);
        MistoSpousteni[] ms = pst.misto();
        // pro anotaci misto={MistoSpousteni.DESKTOP, MistoSpousteni.VALIDATOR}
        if (ms[0] == misto || ms[ms.length - 1] == misto) {
            System.out.print(" - Test metody " + met.getName() + "()");
            try {
                met.invoke(instance);
                System.out.println("OK");
                proslo++;
            } catch (Exception ex) {
                System.out.println("selhal z duvodu\n\t" + ex.getCause());
                chybne++;
            }
        }
    }
    System.out.println("Proslo: " + proslo + ", selhalo: " + chybne);
}

public static void main(String[] args) throws Exception {
    Class<?> zkoumanaClass = Class.forName("anotace.OsobaTest");

    Method[] serazeneMetody =
            serazeniMetodPodlePoradi(zkoumanaClass.getDeclaredMethods());

    Constructor<?> konstruktor = zkoumanaClass.getDeclaredConstructor();
    Object instance = konstruktor.newInstance();
    spusteniTestu(serazeneMetody, MistoSpousteni.DESKTOP, instance);
    spusteniTestu(serazeneMetody, MistoSpousteni.VALIDATOR, instance);
}
}

```

■ po spuštění vypíše:

```

Spoustim testy na: DESKTOP
- Test metody testToString() OK
- Test metody testZmenaNazvu() OK
Proslo: 2, selhalo: 0

Spoustim testy na: VALIDATOR
- Test metody testToString() selhal z duvodu
    org.junit.ComparisonFailure:
        Chybný název: expected:<Osoba_[1]> but was:<Osoba_[3]>
- Test metody testPohlavi() OK
Proslo: 1, selhalo: 1

```

■ z výpisu je vidět, že na DESKTOP i na VALIDATOR proběhly pouze určené testy a to v definovaném pořadí

- selhání testu `testToString()` na `VALIDATOR` je způsobeno dřívějším během testů z `DESKTOP`
- pokud spustíme pouze testy na `VALIDATOR`, proběhnou správně, respektive neselžou

```
Spoustim testy na: VALIDATOR
- Test metody testToString() OK
- Test metody testPohlavi() OK
Proslo: 2, selhalo: 0
```

### 11.3.7.3. Využití kolekcí pro refactoring třídy JUnitTestySpusteniOsoba

- metody `serazeniMetodPodlePoradi()` a `spusteniTestu()` z předchozí třídy `JUnitTestySpusteniOsoba` nebyly příliš elegantní
  - navíc jsou manipulace s indexy v poli náchylné k chybám
- bude ukázáno lepší (elegantnější) řešení obou zmíněných metod s naprosto stejnou funkčností celého programu
  - má demonstrovat praktické využití kolekcí

#### Varování

Práce s anotacemi bude zcela stejná jako v předchozím příkladě.

- třída `JUnitTestySpusteniOsobaLepsi`

```
package anotace;

import org.junit.Test;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.util.*;

public class JUnitTestySpusteniOsobaLepsi {

    private static Collection<Method> serazeniMetodPodlePoradi(Method[] metody) ▶
    {
        int posledni = metody.length;
        Map<Integer, Method> serazeno = new TreeMap<>();

        for (Method metoda : metody) {
            if (metoda.isAnnotationPresent(PodminkySpusteniTestu.class)) {
                int poradi = ▶
                metoda.getAnnotation(PodminkySpusteniTestu.class).poradi();
                if (poradi == 0) {
                    poradi = ++posledni;
                }
                serazeno.put(poradi, metoda);
            }
        }
    }
}
```

```

        return serazeno.values();
    }

private static void spusteniTestu(Collection<Method> metody, MistoSpousteni ►
misto,
                                Object instance) {
    System.out.println("Spoustim testy na: " + misto.name());
    int proslo = 0;
    int chybne = 0;
    for (Method met : metody) {
        if (met.isAnnotationPresent(Test.class) == false) {
            continue;
        }
        if (met.isAnnotationPresent(PodminkySpusteniTestu.class) == false) {
            continue;
        }
        PodminkySpusteniTestu pst = ►
met.getAnnotation(PodminkySpusteniTestu.class);
        Set<MistoSpousteni> mistoSet = new ►
HashSet<>(Arrays.asList(pst.misto()));
        if (mistoSet.contains(misto) == true) {
            System.out.print(" - Test metody " + met.getName() + "() ");
            try {
                met.invoke(instance);
                System.out.println("OK");
                proslo++;
            } catch (Exception ex) {
                System.out.println("selhal z duvodu\n\t" + ex.getCause());
                chybne++;
            }
        }
    }
    System.out.println("Proslo: " + proslo + ", selhalo: " + chybne);
}

public static void main(String[] args) throws Exception {
    Class<?> zkoumanaClass = Class.forName("anotace.OsobaTest");

    Collection<Method> serazeneMetody =
        serazeniMetodPodlePoradi(zkoumanaClass.getDeclaredMethods());

    Constructor<?> konstruktor = zkoumanaClass.getDeclaredConstructor();
    Object instance = konstruktor.newInstance();
    spusteniTestu(serazeneMetody, MistoSpousteni.DESKTOP, instance);
    spusteniTestu(serazeneMetody, MistoSpousteni.VALIDATOR, instance);
}
}

```

■ **metoda** Collection<Method> serazeniMetodPodlePoradi(Method[] metody)

- využívá skutečnosti, že TreeMap<Integer, Method> řadí vkládané dvojice podle klíče

- ♦ pokud je klíčem hodnota poradi z parametru anotace, pak pro
  - poradi  $\geq 1$  stačí dvojici {poradi, metoda} přímo vložit
  - poradi = 0 je nutné hodnotu pořadí změnit na hodnotu posledního použitého indexu
    - na velikost této hodnoty nezáleží, stačí pouze ujištění, že bude větší, než je počet všech metod ze vstupního pole metod
- na závěr se z metody vrací kolekce metod, které byly v mapě seřazeny jako dvojice podle svého klíče
  - ♦ klíč už je však nadále zbytečný, stačí jen seřazené metody

## Poznámka

Tento postup je také bezpečnější, protože hodnoty parametru anotace poradi nemusejí být po sobě jdoucí čísla. Ale největší hodnota nesmí být větší než počet metod v poli.

- metoda spusteniTestu(Collection<Method> metody, MistoSpousteni misto, Object instance)
  - převede pole získané voláním `pst.misto()` na množinu
  - v množině se metodou `contains()` snadno zjistí, zda obsahuje prvek `misto`

## Poznámka

Postup je opět bezpečnější, protože převedením pole na množinu se odstraní případné duplicitní hodnoty. Takže při opakovaném zadání stejného místa v parametru anotace bude program stejně fungovat.

# Kapitola 12. Návrhové vzory

## 12.1. Základní informace

### ■ dobré zdroje informací:

- [en.wikipedia.org/wiki/Software\\_design\\_pattern](http://en.wikipedia.org/wiki/Software_design_pattern)
- [javadesign-patterns.blogspot.cz](http://javadesign-patterns.blogspot.cz)

### ■ obdoba matematických vzorečků – časem prověřená řešení typických programátorských požadavků

- předpisy, do kterých se „dosazují“ třídy a objekty

### ■ výhody

- rychlé řešení – nevymýslí se, pouze se použije
- vyzkoušené řešení – malá pravděpodobnost, že uděláme chybu
- řešení každý rozumí – zjednodušená komunikace mezi členy týmu a dokumentace

### ■ prvotní zdroj GoF (*Gang of Four*)

■ Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995

■ v knize je 23 základních vzorů, které se dělí do tří hlavních skupin

- **konstrukční vzory (creational)**

- ◆ abstraktní továrna (*Abstract factory*)
- ◆ staviteľ (*Builder*)
- ◆ tovární metoda (*Factory method*)
- ◆ prototyp (*Prototype*)
- ◆ jedináček (*Singleton*)

- **strukturální vzory (structural)**

- ◆ adaptér (*Adapter*)
- ◆ most (*Bridge*)
- ◆ kompozit (*Composite*)
- ◆ dekorátor (*Decorator*)
- ◆ fasáda (*Facade*)
- ◆ muší váha (*Flyweight*)

- ◆ zástupce (*Proxy*)

- **vzory chování (behavioral)**

- ◆ řetěz odpovědnosti (*Chain of responsibility*)

- ◆ příkaz (*Command*)

- ◆ interpret (*Interpreter*)

- ◆ iterátor (*Iterator*)

- ◆ prostředník (*Mediator*)

- ◆ memento (*Memento*)

- ◆ pozorovatel (*Observer*)

- ◆ stav (*State*)

- ◆ strategie (*Strategy*)

- ◆ šablonová metoda (*Template method*)

- ◆ návštěvník (*Visitor*)

- postupem času začaly přibývat další a další vzory i kategorie, např.:

- **Concurrency patterns** – *Lock, Monitor, Read write lock, Scheduler, Thread pool, ...*

- **Architectural patterns** – *Model–View–Controller (MVC), n–tier, Data access object (DAO), Data transfer object (DTO), ...*

- časem se objevily i různé variace těchto vzorů

- lze nalézt i několik velmi **principiálně** různých implementací jednoho a téhož vzoru

- užitečné může být dělení vzorů dle knihy: Pecinovský, R.: Návrhové vzory – 33 vzorových postupů pro objektové programování, Computer Press, 2007,

- **ovlivnění počtu instancí**

- ◆ knihovní třída (*Library class*)

- ◆ jedináček (*Singleton*)

- ◆ výčtový typ (*Enumerated type*)

- ◆ fond (*Pool*)

- ◆ muší váha (*Flyweight*)

- **skrývání (implementace)**

- ◆ zástupce (*Proxy*)

- ◆ příkaz (*Command*)

- ◆ iterátor (*Iterator*)
- ◆ stav (*State*)
- ◆ šablonová metoda (*Template method*)

- **optimalizace rozhraní**

- ◆ fasáda (*Facade*)
- ◆ adaptér (*Adapter*)
- ◆ kompozit (*Composite*)

- **univerzální vytváření**

- ◆ tovární metoda (*Factory method*)
- ◆ prototyp (*Prototype*)
- ◆ stavitel (*Builder*)
- ◆ abstraktní továrna (*Abstract factory*)

- **zjednodušení programu**

- ◆ dekorátor (*Decorator*)
- ◆ řetěz odpovědnosti (*Chain of responsibility*)
- ◆ pozorovatel (*Observer*)
- ◆ prostředník (*Mediator*)

- **zvýšení přizpůsobitelnosti**

- ◆ most (*Bridge*)
- ◆ strategie (*Strategy*)
- ◆ návštěvník (*Visitor*)
- ◆ memento (*Memento*)
- ◆ interpret (*Interpreter*)

## Varování

Nevhodné použití návrhových vzorů může zbytečně zvýšit složitost úlohy.

## 12.2. Konstrukční vzory

### 12.2.1. Stavitel (*Builder*)

- v případě velmi složité instance umožňuje její postupné přehledné vytváření

- vytváříme jen to, co nás skutečně zajímá, ostatní je default
- navíc mohou být jednotlivé části velmi složité a proto je vytváří vždy specializovaná metoda

ZnalostJazyka je pouze výčtový typ

```
public enum ZnalostJazyka {
    ZADNA, A1, A2, B1, B2, C1, C2, RODILY_MLUVCI;
}
```

StudentJazykoveSkoly musí mít nainicializovány všechny atributy (znalosti všech uvedených jazyků)

- pro malý počet jazyků by šel jejich seznam předat jako parametry konstruktoru
- v případě, že by jazyková škola vyučovala desítky jazyků, byl by konstruktor nepřehledný
- postupná konstrukce je umožněna díky vnořené třídě (má přístup k atributům vnější třídy) Builder
  - ta svojí metodou build() vytvoří a vrátí objekt své vnější třídy StudentJazykoveSkoly
    - ◆ vyvolá konstruktor vnější třídy, ve kterém se pouze překopíruje aktuální nastavení ze třídy Builder
  - konstruktor třídy Builder má jen jeden parametr – řetězec jméno studenta, které musí být vždy zadáno
  - specializované metody (cestina(), ...) umožní samostatnou inicializaci každého jednotlivého jazyka
    - ◆ vždy vrací objekt třídy Builder, což pak umožňuje zřetězení jednotlivých jazyků

```
public class StudentJazykoveSkoly {
    private final String jmeno; // musí být vyplněno
    private ZnalostJazyka cestina;
    private ZnalostJazyka anglictina;
    private ZnalostJazyka nemcina;
    private ZnalostJazyka rustina;

    private StudentJazykoveSkoly(Builder builder) {
        this.jmeno = builder.jmeno;
        this.cestina = builder.cestina;
        this.anglictina = builder.anglictina;
        this.nemcina = builder.nemcina;
        this.rustina = builder.rustina;
    }

    @Override
    public String toString() {
        return jmeno + " [cestina=" + cestina
            + ", anglictina=" + anglictina + ", nemcina=" + nemcina
            + ", rustina=" + rustina + "]";
    }

    public static class Builder {
        public final String jmeno;
        public ZnalostJazyka cestina = ZnalostJazyka.ZADNA;
        public ZnalostJazyka anglictina = ZnalostJazyka.ZADNA;
```

```

public ZnalostJazyka nemcina = ZnalostJazyka.ZADNA;
public ZnalostJazyka rustina = ZnalostJazyka.ZADNA;

// vytvoření vnější třídy
public StudentJazykoveSkoly build() {
    return new StudentJazykoveSkoly(this);
}

// konstruktor pro povinné atributy
public Builder(String jmeno) {
    this.jmeno = jmeno;
}

public Builder cestina(ZnalostJazyka uroven) {
    this.cestina = uroven;
    return this;
}

public Builder anglictina(ZnalostJazyka uroven) {
    this.anglictina = uroven;
    return this;
}

public Builder nemcina(ZnalostJazyka uroven) {
    this.nemcina = uroven;
    return this;
}

public Builder rustina(ZnalostJazyka uroven) {
    this.rustina = uroven;
    return this;
}
}
}
}

```

v aplikaci se jednotliví studenti vytvářejí postupným skládáním jen významových jazyků

- celý proces je završen voláním metody `build()`

```

public class Aplikace {

    public static void main(String[] args) {
        StudentJazykoveSkoly pavel = new StudentJazykoveSkoly.Builder("Pavel")
            .cestina(ZnalostJazyka.RODILY_MLUVCI)
            .anglictina(ZnalostJazyka.B2)
            .rustina(ZnalostJazyka.B2)
            .build();

        StudentJazykoveSkoly keith = new StudentJazykoveSkoly.Builder("Keith")
            .anglictina(ZnalostJazyka.RODILY_MLUVCI)
            .nemcina(ZnalostJazyka.C1)
            .rustina(ZnalostJazyka.ZADNA)
            .cestina(ZnalostJazyka.A1)
    }
}

```

```

        .build();

    System.out.println(pavel);
    System.out.println(keith);
}
}

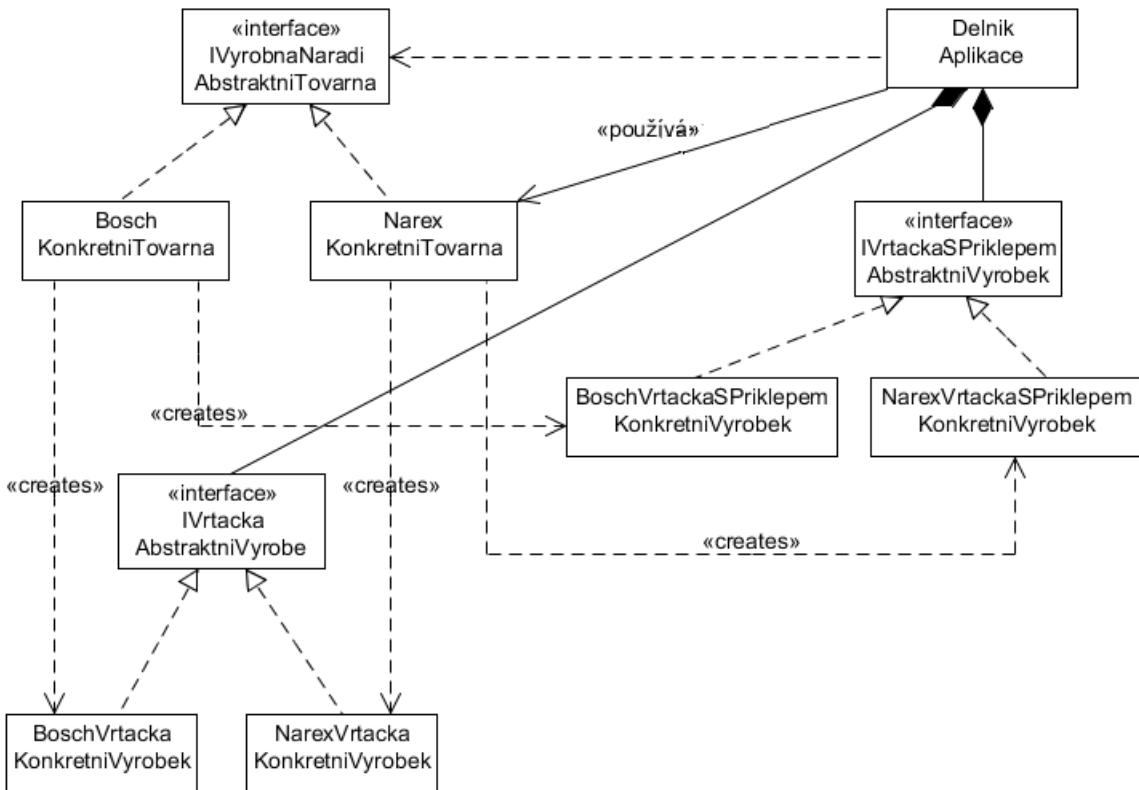
```

- vypíše:

```
Pavel [cestina=RODILY_MLUVCI, anglictina=B2, nemcina=ZADNA, rustina=B2]
Keith [cestina=A1, anglictina=RODILY_MLUVCI, nemcina=C1, rustina=ZADNA]
```

## 12.2.2. Abstraktní továrna (*Abstract factory*)

- definuje rozhraní pro tvorbu celé rodiny souvisejících nebo závislých objektů a tím odděluje klienta od vlastního procesu vytváření objektů
- abstraktní továrna umožňuje jednoduše ovlivnit chování celého systému
- existuje X výrobců a každý z nich vyrábí Y různých (na sobě nezávislých) výrobků
  - výrobek Y1 od firmy A má **stejnou funkčnost**, jako výrobek Y1 od firmy B
    - ◆ může mít jiný vzhled, odlišný způsob práce, apod.
- cílem je, aby uživatel (aplikace) používala všechny výrobky právě od jednoho výrobce
  - a bylo možné jednoduše (jedním příkazem) stanovit, který výrobce to bude
  - po výběru výrobce jsou všechny výrobky vytvořeny automaticky
- příklad ze života – změna celého GUI (k vidění např. v UMLet)



- tím, že **Delnik použije vazbu <<používá>> na (firmu) Narex, vytvoří si NarexVrtacku a NarexVrtackuSPriklepem**
  - je to jeho jediná vazba na konkrétní třídu
  - všechny další vazby vedou na rozhraní
- pokud by vazbu <<používá>> přemístil na (firmu) Bosch, vytvoří si nářadí od této firmy

**IVyrobnaNaradi\_AbstraktniTovarna** určuje, že bude produkovat dva typy (představované rozhraními) různých abstraktních výrobků – **IVrtacka\_AbstraktniVyrobek** a **IVrtackaSPriklepem\_AbstraktniVyrobek**

```

public interface IVyrobnaNaradi_AbstraktniTovarna {
    public IVrtacka_AbstraktniVyrobek vyrobVrtacku();
    public IVrtackaSPriklepem_AbstraktniVyrobek vyrobVrtackuSPriklepem();
}

```

**IVrtacka\_AbstraktniVyrobek** definuje schopnosti prvního abstraktního výrobku

```

public interface IVrtacka_AbstraktniVyrobek {
    public void vrta();
}

```

**IVrtackaSPriklepem\_AbstraktniVyrobek** definuje schopnosti druhého abstraktního výrobku

```

public interface IVrtackaSPriklepem_AbstraktniVyrobek {
    public void vrta();
    public void porovnaniCen(IVrtacka_AbstraktniVyrobek abstraktniVrtacka);
}

```

Bosch\_KonkretniTovarna **zajistí výrobu dvou konkrétních typů výrobků** – BoschVrtacka\_KonkretniVyrobek a BoschVrtackaSPriklepem\_KonkretniVyrobek

```
public class Bosch_KonkretniTovarna implements IVyrobnaNaradi_AbstraktniTovarna{  
    @Override  
    public IVrtacka_AbstraktniVyrobek vyrobVrtacku() {  
        return new BoschVrtacka_KonkretniVyrobek();  
    }  
  
    @Override  
    public IVrtackaSPriklepem_AbstraktniVyrobek vyrobVrtackuSPriklepem() {  
        return new BoschVrtackaSPriklepem_KonkretniVyrobek();  
    }  
}
```

BoschVrtacka\_KonkretniVyrobek **zajistí, že konkrétní výrobek má požadované schopnosti (z rozhraní IVrtacka\_AbstraktniVyrobek)**

```
public class BoschVrtacka_KonkretniVyrobek implements  
        IVrtacka_AbstraktniVyrobek {  
    @Override  
    public void vrta() {  
        System.out.println(getClass().getSimpleName() + " a vrta.");  
    }  
}
```

**dtto pro** BoschVrtackaSPriklepem\_KonkretniVyrobek

```
public class BoschVrtackaSPriklepem_KonkretniVyrobek implements  
        IVrtackaSPriklepem_AbstraktniVyrobek {  
    @Override  
    public void vrta() {  
        System.out.println(getClass().getSimpleName() + " a vrta s priklepem.");  
    }  
  
    @Override  
    public void porovnaniCen(IVrtacka_AbstraktniVyrobek abstraktniVrtacka) {  
        System.out.println(getClass().getSimpleName() + " je dražší než "  
                + abstraktniVrtacka.getClass().getSimpleName());  
    }  
}
```

Narex\_KonkretniTovarna **je jako** Bosch\_KonkretniTovarna

```
public class Narex_KonkretniTovarna implements IVyrobnaNaradi_AbstraktniTovarna{  
    @Override  
    public IVrtacka_AbstraktniVyrobek vyrobVrtacku() {  
        return new NarexVrtacka_KonkretniVyrobek();  
    }  
  
    @Override  
    public IVrtackaSPriklepem_AbstraktniVyrobek vyrobVrtackuSPriklepem() {  
        return new NarexVrtackaSPriklepem_KonkretniVyrobek();  
    }  
}
```

```
}
```

dtto

```
public class NarexVrtacka_KonkretniVyrobek implements  
    IVrtacka_AbstraktniVyrobek {  
  
    @Override  
    public void vrta() {  
        System.out.println(getClass().getSimpleName() + " a vrta.");  
    }  
}
```

dtto

```
public class NarexVrtackaSPriklepem_KonkretniVyrobek implements  
    IVrtackaSPriklepem_AbstraktniVyrobek {  
  
    @Override  
    public void vrta() {  
        System.out.println(getClass().getSimpleName() + " a vrta s priklepem.");  
    }  
  
    @Override  
    public void porovnaniCen(IVrtacka_AbstraktniVyrobek abstraktniVrtacka) {  
        System.out.println(getClass().getSimpleName() + " je dražší než "  
                           + abstraktniVrtacka.getClass().getSimpleName());  
    }  
}
```

Delnik\_Aplikace ví, že bude potřebovat nějakou vrtačku (IVrtacka\_AbstraktniVyrobek) a nějakou vrtačku s příklepem (IVrtackaSPriklepem\_AbstraktniVyrobek)

- obě mu vyrobí továrna v jeho konstruktoru
- jaká to bude továrna (Bosch nebo Narex), se určí skutečným parametrem konstruktoru
  - žádný jiný vztah ke konkrétní továrně či konkrétním výrobkům dělník nemá

```
public class Delnik_Aplikace {  
    IVrtacka_AbstraktniVyrobek vrtacka;  
    IVrtackaSPriklepem_AbstraktniVyrobek priklepovka;  
  
    public Delnik_Aplikace(IVyrobnaNaradi_AbstraktniTovarna tovarna)  
    {  
        vrtacka = tovarna.vyrobVrtacku();  
        priklepovka = tovarna.vyrobVrtackuSPriklepem();  
    }  
  
    public static void main(String[] args) {  
//    Delnik_Aplikace delnik = new Delnik_Aplikace(new Bosch_KonkretniTovarna());  
    Delnik_Aplikace delnik = new Delnik_Aplikace(new Narex_KonkretniTovarna());  
    System.out.println("Delnik používá: ");  
    delnik.vrtacka.vrta();  
    }
```

```

        delnik.priklepovka.vrta();
        System.out.println("Delnik zaplatil: ");
        delnik.priklepovka.porovnaniCen(delnik.vrtacka);
    }
}

```

- vypíše:

```

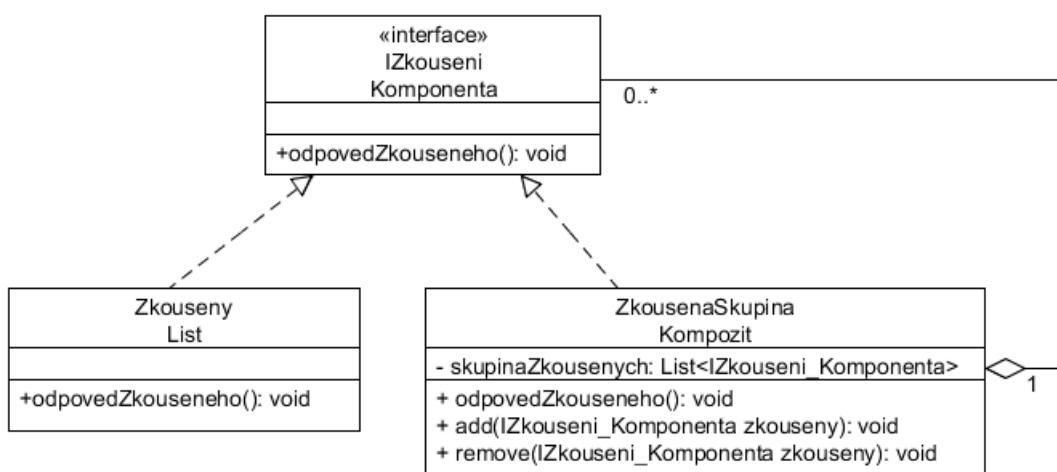
Delnik pouziva:
NarexVrtacka_KonkretniVyrobek a vrta.
NarexVrtackaSPriklepem_KonkretniVyrobek a vrta s priklepem.
Delnik zaplatil:
NarexVrtackaSPriklepem_KonkretniVyrobek je drazsi nez ►
NarexVrtacka_KonkretniVyrobek

```

## 12.3. Strukturální vzory

### 12.3.1. Kompozit (*Composite*)

- sjednocuje typy používaných objektů a umožňuje tak jednotné zpracování každého z nich nezávisle na tom, jedná-li se o koncový (dále nedělitelný) objekt nebo o objekt složený z jiných objektů
- příklady z Java Core API
  - `java.io.File` – pro soubory i pro adresáře (složky)
  - `java.awt.Component` a `java.awt.Container`



`IZkouseni_Komponenta` definuje schopnosti kompozitu

```

public interface IZkouseni_Komponenta {
    public void odpovedZkouseneho();
}

```

`Zkouseny_List` jako konečný prvek implementuje schopnosti kompozitu

```

public class Zkouseny_List implements IZkouseni_Komponenta {
    private static final Random nahodnyTip = new Random(1);
    private static final int POCET_ODPOVEDI = 4;

    private final String jmeno;

    public Zkouseny_List(String jmeno) {
        this.jmeno = jmeno;
    }

    @Override
    public void odpovedZkouseneho() {
        System.out.println(jmeno + "> "
            + (char) (nahodnyTip.nextInt(POCET_ODPOVEDI) + 'a'));
    }
}

```

ZkousenaSkupina\_Komposit má kontejner na jednotlivé kompozity skupinaZkousenych

- vykonávání schopností kompozitu (`odpovedZkouseneho()`) postupně deleguje na jednotlivé prvky kontejneru
- má metody pro přidání (`add(IZkouseni_Komponenta zkouseny)`) a ubrání (`remove(IZkouseni_Komponenta zkouseny)`) kompozitu z kontejneru

```

public class ZkousenaSkupina_Komposit implements IZkouseni_Komponenta {
    private final List<IZkouseni_Komponenta> skupinaZkousenych =
        new ArrayList<IZkouseni_Komponenta>();

    @Override
    public void odpovedZkouseneho() {
        for (IZkouseni_Komponenta zkouseny : skupinaZkousenych) {
            zkouseny.odpovedZkouseneho();
        }
    }

    // přidání zkoušeného do skupiny
    public void add(IZkouseni_Komponenta zkouseny) {
        skupinaZkousenych.add(zkouseny);
    }

    // odebrání zkoušeného ze skupiny
    public void remove(IZkouseni_Komponenta zkouseny) {
        skupinaZkousenych.remove(zkouseny);
    }
}

```

Aplikace může vytvářet instance kompozitů, jak listů (`Zkouseny_List`), tak i seznamů (`ZkousenaSkupina_Komposit`) a libovolně je seskupovat

- případně skupiny měnit

```

public class Aplikace {

```

```

public static void main(String[] args) {
    // inicializace zkoušených studentů
    Zkouseny_List studentka1 = new Zkouseny_List("Alena");
    Zkouseny_List studentka2 = new Zkouseny_List("Hana");
    Zkouseny_List student3 = new Zkouseny_List("Pavel");
    Zkouseny_List student4 = new Zkouseny_List("Petr");
    Zkouseny_List studentZahranicni = new Zkouseny_List("John");

    // inicializace zkouškových skupin
    ZkousenaSkupina_Kompozit zeny = new ZkousenaSkupina_Kompozit();
    ZkousenaSkupina_Kompozit muzi = new ZkousenaSkupina_Kompozit();
    ZkousenaSkupina_Kompozit vsichni = new ZkousenaSkupina_Kompozit();

    // přidání studentů do skupin
    zeny.add(studentka1);
    zeny.add(studentka2);

    muzi.add(student3);
    muzi.add(student4);
    vsichni.add(studentZahranicni);

    // vytvoření finální skupiny
    vsichni.add(zeny);
    vsichni.add(muzi);

    // vyzkoušení celé skupiny
    vsichni.odpovedZkouseneho();
    // přezkoušení jedné studentky
    studentka1.odpovedZkouseneho();
    // přezkoušení všech mužů
    vsichni.remove(zeny);
    vsichni.odpovedZkouseneho();
}
}

```

■ vypíše:

```

John> c
Alena> a
Hana> b
Pavel> b
Petr> a
Alena> a
John> b
Pavel> c
Petr> d

```

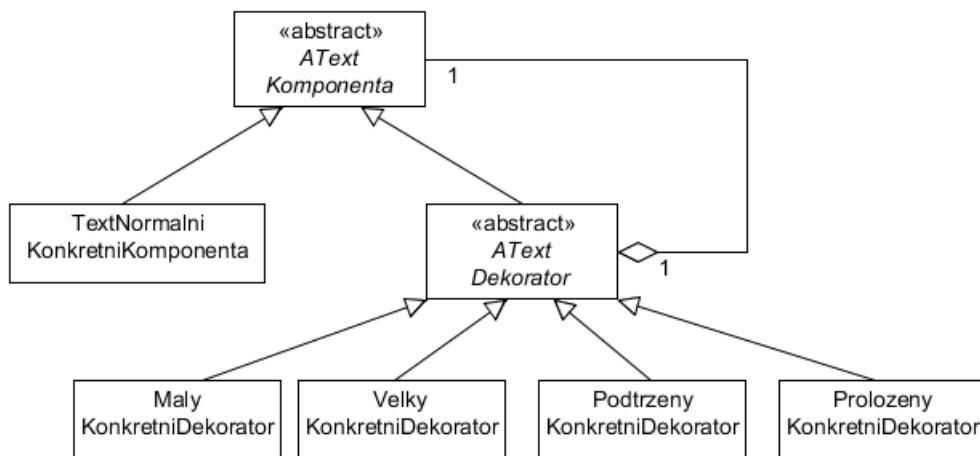
## 12.3.2. Dekorátor (*Decorator*)

- přidává další funkcionality k objektu tím, že objekt „zabalí“ do jiného objektu, který má na starosti pouze tuto přidanou funkcionality; zbytek činnosti deleguje na „zabalený“ objekt

- tím umožňuje přidávat funkčnost dynamicky, ale nezvyšovat počet tříd kombinacemi jejich možností
  - ◆ počet kombinací by jinak zvyšoval počet tříd kombinatoricky
- další funkcionality se stejně snadno přidávají bez ovlivnění stávajících tříd
- příklad z Java Core API – vše, co je nabalenou např. na `java.io.Reader` – `BufferedReader`, `FileReader`, `StringReader`, `InputStreamReader` atd.

```
BufferedReader bfr = new BufferedReader(
    new FileReader(
        new File(celeJmenoSouboru) ));
```

- další vhodné použití je v případě plánovaných postupných úprav a rozšiřování



```
public abstract class AText_Komponenta {
    public abstract String textProVypis();
}
```

`TextNormalni_KonkretniKomponenta` jako jediná má v sobě data, se kterými se bude pracovat – `poskytovanyText`

```
public class TextNormalni_KonkretniKomponenta extends AText_Komponenta {
    private String poskytovanyText;

    public TextNormalni_KonkretniKomponenta(String poskytovanyText) {
        this.poskytovanyText = poskytovanyText;
    }

    @Override
    public String textProVypis() {
        return poskytovanyText;
    }
}
```

`AText_Dekorator` má atribut `dekorovanyText`, kterým se odkazuje na `AText_Komponenta`, tj. na skutečný text

```

public abstract class AText_Dekorator extends AText_Komponenta {
    protected final AText_Komponenta dekorovanyText;

    public AText_Dekorator (AText_Komponenta dekorovanyText) {
        this.dekorovanyText = dekorovanyText;
    }

    public String textProVypis() {
        return dekorovanyText.textProVypis();
    }
}

```

„dekorování“, tj. přidání funkčnosti je pomocí super.textProVypis().toLowerCase()

```

public class Maly_KonkretniDekorator extends AText_Dekorator {
    public Maly_KonkretniDekorator(AText_Komponenta dekorovanyText) {
        super(dekorovanyText);
    }

    @Override
    public String textProVypis() {
        return super.textProVypis().toLowerCase() + " (maly) ";
    }
}

```

```

public class Velky_KonkretniDekorator extends AText_Dekorator {
    public Velky_KonkretniDekorator(AText_Komponenta dekorovanyText) {
        super(dekorovanyText);
    }

    @Override
    public String textProVypis() {
        return super.textProVypis().toUpperCase() + " (VELKY) ";
    }
}

```

```

public class Podtrzeny_KonkretniDekorator extends AText_Dekorator {
    public Podtrzeny_KonkretniDekorator(AText_Komponenta dekorovanyText) {
        super(dekorovanyText);
    }

    @Override
    public String textProVypis() {
        StringBuilder sb = new StringBuilder("\n");
        String pomocny = super.textProVypis();

        for (int i = 0; i < pomocny.length(); i++) {
            sb.append("=");
        }

        return pomocny + sb.toString() + " (podtrzeny) ";
    }
}

```

```

public class Prolozeny_KonkretniDekorator extends AText_Dekorator {
    public Prolozeny_KonkretniDekorator(AText_Komponenta dekorovanyText) {
        super(dekorovanyText);
    }

    @Override
    public String textProVypis() {
        StringBuilder sb = new StringBuilder();
        String pomocny = super.textProVypis();

        for (int i = 0; i < pomocny.length(); i++) {
            sb.append(pomocny.charAt(i) + " ");
        }

        return sb.toString() + " (p r o l o z e n y) ";
    }
}

```

Aplikace může vytvářet stejně jako

- instance nedekorovaného textu – text
- instance jednotlivých dekorátorů – textP
- instance kombinací dekorátorů – textVP nebo textPoM

```

public class Aplikace {
    public static final void main(String[] args) {
        AText_Komponenta text =
            new TextNormalni_KonkretniKomponenta("Poskytovany TEXT");

        System.out.println(text.textProVypis());

        AText_Komponenta textP = new Prolozeny_KonkretniDekorator(text);
        System.out.println(textP.textProVypis());

        AText_Komponenta textVP = new Velky_KonkretniDekorator(
            new Prolozeny_KonkretniDekorator(text));
        System.out.println(textVP.textProVypis());

        AText_Komponenta textPoM = new Podtrzeny_KonkretniDekorator(
            new Maly_KonkretniDekorator(text));
        System.out.println(textPoM.textProVypis());
    }
}

```

- vypíše:

```

Poskytovany TEXT
P o s k y t o v a n y     T E X T     (p r o l o z e n y)
P O S K Y T O V A N Y     T E X T     (P R O L O Z E N Y)     (VELKY)
poskytovany text (maly)
===== (podtrzeny)

```

## 12.3.3. Adaptér (Adapter)

- tento návrhový vzor má více možností použití
- v souvislosti s výše uvedenými GUI rozhraními se při programování GUI setkáme velmi rychle s jednou z nich

### Poznámka

Následující text nevysvětluje jemnosti v používání GUI, ale principy, kterými se tyto akce programují.

- pro ošetřování akcí způsobených pohybem myši je třeba implementovat rozhraní `java.awt.event.MouseMotionListener`
  - toto rozhraní má celkem dvě metody
    - ◆ `void mouseDragged(MouseEvent e)` – tažení myší
    - ◆ `void mouseMoved(MouseEvent e)` – přesun myší
- pokud budeme klasickým způsobem pomocí vnitřní třídy obsluhovat jen tažení myší (pohyb myši nás nezajímá), pak musíme implementovat obě metody
  - ale metoda `mouseMoved()` bude mít prázdné tělo, což je lehce matoucí

```
...
tlacitkoAhojBT.addMouseMotionListener(new ObsluhaMysi());
...

private class ObsluhaMysi implements MouseMotionListener {
    public void mouseDragged(MouseEvent e) {
        napisLB.setText("Tahnu");
    }
    public void mouseMoved(MouseEvent e) {
        // zadna akce
    }
}
```

- místo toho se často používá způsob, kdy pro každé podobné rozhraní (s více než jednou metodou) je v Java Core API připravena třída adaptéru implementující toto rozhraní
  - ta všechny metody implementuje jako prázdné
  - ve vnitřní třídě pak neimplementujeme rozhraní (= nutnost psát všechny metody z rozhraní), ale dědíme adaptér a překryjeme pouze metody, které nás zajímají

```
private class ObsluhaMysiAdapter extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
        napisLB.setText("Tahnu");
    }
}
```

### 12.3.3.1. Adaptér je vnořený typ vnějšího rozhraní

- pro výše popisovaný způsob by bylo výhodnější, kdyby byl příslušný adaptér s rozhraním spojen
  - toho se dá dosáhnout využitím vnořeného typu – viz též dříve
- pro obsluhu GUI se toto nepoužívá, ale může to být výhodné pro námi připravovaná rozhraní
- příklad je umělý, napodobující předchozí příklad s GUI
- rozhraní IPohybyMysi má dvě metody tazeniMysi() a pohybMysi()
  - dále obsahuje vnořenou (static) třídu Adapter, která toto rozhraní implementuje
    - ◆ výjimka java.lang.UnsupportedOperationException je častý způsob, jak předpřipravit kód, který není zcela funkční, aby se v budoucnu nezapomnělo tuto funkčnost dodat
    - ◆ metoda bez implementace proběhne bez jakéhokoliv varování

```
public interface IPohybyMysi {  
    public void tazeniMysi();  
    public void pohybMysi();  
  
    /**  
     * Vnorena trida implementujici vnejsi rozhrani  
     */  
    public static class Adapter implements IPohybyMysi {  
        public void tazeniMysi() {  
            // zadna akce  
        }  
        public void pohybMysi() {  
            throw new UnsupportedOperationException("neimplementovano");  
        }  
    }  
}
```

- třída Pouziti ukazuje oba způsoby využití rozhraní

- výhodou je, že třída adaptéra je neoddělitelně spojena se třídou rozhraní
  - ◆ to představuje bezpečnější kód než v případě GUI, kdy se adaptér shodoval s rozhraním jen částí jména (MouseMotionAdapter a MouseMotionListener)

```
public class Pouziti {  
    IPohybyMysi zpusob1 = new ImplementaceRozhrani();  
    IPohybyMysi zpusob2 = new DedeniAdapteru();  
  
    public Pouziti() {}  
  
    private class ImplementaceRozhrani implements IPohybyMysi {  
        public void tazeniMysi() {  
            System.out.println("Tazeni - rozhrani");  
        }  
    }  
}
```

```

    }
    public void pohybMysi() {
        // zadna akce, ale metoda musi byt uvedena jako prazdna
    }
}

private class DedeniAdapteru extends IPohybyMysi.Adapter {
    public void tazeniMysi() {
        System.out.println("Tazeni - adapter");
    }
}
}

```

■ **třída Hlavni**

```

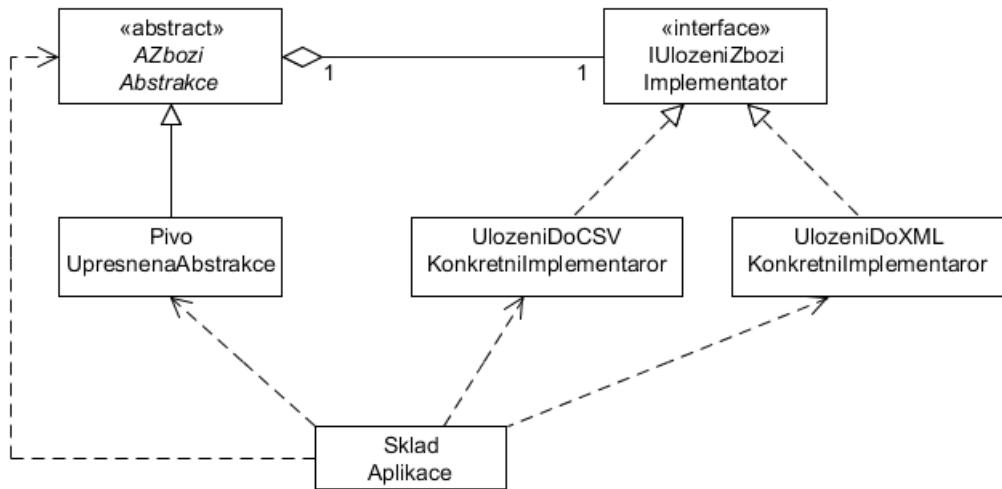
public class Hlavni {
    public static void main(String[] args) {
        Pouziti p = new Pouziti();
        p.zpusob1.tazeniMysi();
        p.zpusob2.tazeniMysi();
    }
}

```

## 12.3.4. Most (*Bridge*)

- aplikace komunikuje se třídou, která je pouze zprostředkovatelem nabízené služby

- skutečnou službu má na starosti jiná třída
  - ◆ třída využívající službu ji neumí nebo nechce zajistit
  - ◆ jiným častým případem je, že službu od začátku vytváří někdo jiný – týmová práce
- typicky je to služba, která je nějak implementačně závislá a proto se může v budoucnu měnit
  - ◆ pak se snadno vymění původní třída za jinou, bez ovlivnění zbývajících tříd
- nebo mají příbuzné třídy jednu (či více) služeb odlišných



`IUlozeniZbozi_Impementator` popisuje externě zajišťovanou službu

```
public interface IUlozeniZbozi_Impementator {
    public void uloz(String nazev, double cena);
}
```

`AZbozi_Abstrakce` je předek tříd, které budou službu využívat

- využívaná služba (`uloz()`) se může jmenovat jinak – zde `ulozZbozi()`
- ostatní služby si třída může zajišťovat vlastními prostředky – zde `zmenCenu()`

```
public abstract class AZbozi_Abstrakce {
    protected IUlozeniZbozi_Impementator ulozeniZbozi;

    public AZbozi_Abstrakce(IUlozeniZbozi_Impementator ulozeniZbozi) {
        this.ulozeniZbozi = ulozeniZbozi;
    }

    public abstract void ulozZbozi(); // implementačně závislé
    public abstract void zmenCenu(double procenta); // implementačně NEzávislé
}
```

`Pivo_UpresnenaAbstrakce` stanoví v konstruktoru (*dependency injection*), kdo bude skutečně externí službu zajišťovat

- tuto službu pak volá přes atribut ze své rodičovské třídy – `ulozeniZbozi.uloz(nazev, cena)`;

```
public class Pivo_UpresnenaAbstrakce extends AZbozi_Abstrakce {
    private String nazev;
    private double cena;

    public Pivo_UpresnenaAbstrakce(String nazev, double cena,
                                    IUlozeniZbozi_Impementator ulozeniZbozi) {
        super(ulozeniZbozi);
        this.nazev = nazev;
        this.cena = cena;
    }
}
```

```

// implementačně závislé
@Override
public void ulozZbozi() {
    ulozeniZbozi.uloz(nazev, cena);
}

// implementačně NEzávislé
@Override
public void zmenCenu(double procenta) {
    cena *= procenta;
}
}

```

**UlozeniDoCSV\_KonkretniImplementaror – první možná implementace služby – uložení do formátu CSV**

```

public class UlozeniDoCSV_KonkretniImplementaror implements
    IUlozeniZbozi_Implementator {
    @Override
    public void uloz(String nazev, double cena) {
        System.out.println(nazev + ";" + cena);
    }
}

```

**UlozeniDoXML\_KonkretniImplementaror – druhá možná implementace služby – uložení do formátu XML**

```

public class UlozeniDoXML_KonkretniImplementaror implements
    IUlozeniZbozi_Implementator {
    @Override
    public void uloz(String nazev, double cena) {
        System.out.println("<zbozi>");
        System.out.println("  <nazev>" + nazev + "</nazev>");
        System.out.println("  <cena>" + cena + "</cena>");
        System.out.println("</zbozi>");
    }
}

```

**Sklad\_Aplikace využívá možnosti, že každé z piv bude mít jinou externí službu uložení**

```

public class Sklad_Aplikace {
    public static void main(String[] args) {
        AZbozi_Abstrakce[] zbozi = new AZbozi_Abstrakce[] {
            new Pivo_UpresnenaAbstrakce("Gambrinus", 10,
                new UlozeniDoXML_KonkretniImplementaror()),
            new Pivo_UpresnenaAbstrakce("Prazdroj", 20,
                new UlozeniDoCSV_KonkretniImplementaror()),
        };

        for (AZbozi_Abstrakce kus : zbozi) {
            kus.zmenCenu(1.5);
            kus.ulozZbozi();
        }
    }
}

```

```
}
```

- vypíše:

```
<zbozi>
  <nazev>Gambrinus</nazev>
  <cena>15.0</cena>
</zbozi>
Prazdroj; 30.0
```

## 12.4. Vzory chování

### 12.4.1. Příkaz (*Command*)

#### Varování

Význam tohoto návrhového vzoru se snižuje s rozvojem lambda výrazů, které jej v jednoduchých případech velmi efektivně nahrazují.

- zabalí metodu (službu) do objektu, takže s ní lze pracovat jako s daty
  - typicky předat jako skutečný parametr metody
- zabalení umožňuje dynamickou výměnu používaných metod i za běhu programu
- příklady z Java Core API
  - `java.util.Arrays` – `sort()`, `binarySearch()`
  - `java.util.TreeSet`, `java.util.TreeMap`

#### IStylVypisu – definice poskytované služby

```
public interface IStylVypisu {
    public void vypisStylem(String slovo);
}
```

Styl\_Prikaz – předpřipravené služby jako statické konstanty (STANDARDNI, VELKY, MALY, PROLOZENE) implementované pomocí anonymních vnitřních tříd

```
public class Styl_Prikaz {

    // konstanty různých stylů výpisu
    public static final IStylVypisu STANDARDNI = new IStylVypisu() {
        public void vypisStylem(String slovo) {
            System.out.print(slovo);
        }
    };

    public static final IStylVypisu VELKY = new IStylVypisu() {
        public void vypisStylem(String slovo) {
            System.out.print(slovo.toUpperCase());
        }
    };
}
```

```

    }

}

public static final IStylVypisu MALY = new IStylVypisu() {
    public void vypisStylem(String slovo) {
        System.out.print(slovo.toLowerCase());
    }
};

public static final IStylVypisu PROLOZENE = new IStylVypisu() {
    public void vypisStylem(String slovo) {
        for (int i = 0; i < slovo.length(); i++) {
            System.out.print(slovo.charAt(i) + " ");
        }
    }
};

}
}

```

služba implementovaná samostatnou třídou StylPrikazVelbloud

- jiný způsob definici služby

```

public class StylPrikazVelbloud implements IStylVypisu {

    public void vypisStylem(String slovo) {
        String pomocne = slovo.toUpperCase();
        for (int i = 0; i < pomocne.length(); i++) {
            char znak = pomocne.charAt(i);
            znak = (i % 2 == 0) ? znak : Character.toLowerCase(znak);
            System.out.print(znak);
        }
    }
}

```

VyuzivaStyl je jedna z mnoha možných tříd využívající nabízené služby

- pro nejčastěji používaný styl výpisu (tj. STANDARDNI) má pro zjednodušení přetíženou metodu

- srovnej Arrays.sort(pole) – přirozené řazení a Arrays.sort(pole, komparator) – abso-lutní řazení

```

public class VyuzivaStyl {

    public static void vypisVeStylu(String[] slova) {
        vypisVeStylu(slova, Styl_Prikaz.STANDARDNI);
    }

    public static void vypisVeStylu(String[] slova, IStylVypisu styl) {
        System.out.print("[");
        for (String slovo : slova) {
            styl.vypisStylem(slovo + " ");
        }
        System.out.println("]");
    }
}

```

```
}
```

Aplikace – ukázka použití různých zápisů jako:

- předpřipravených konstant ze Styl\_Prikaz (VELKY, PROLOZENE)
- instance třídy StylPrikazVelbloud

```
public class Aplikace {  
  
    public static void main(String[] args) {  
        String[] slova = {"Kočka", "leze", "dírou",  
                          "pes", "oknem",  
                          "nebude-li", "pršet", "nezmoknem" };  
  
        System.out.println(Arrays.toString(slova));  
        VyuzivaStyl.vypisVeStylu(slova);  
        VyuzivaStyl.vypisVeStylu(slova, Styl_Prikaz.VELKY);  
        VyuzivaStyl.vypisVeStylu(slova, Styl_Prikaz.PROLOZENE);  
        VyuzivaStyl.vypisVeStylu(slova, new StylPrikazVelbloud());  
    }  
}
```

- vypíše:

```
[Kočka, leze, dírou, pes, oknem, nebude-li, pršet, nezmoknem]  
[Kočka leze dírou pes oknem nebude-li pršet nezmoknem ]  
[KOČKA LEZE DÍROU PES OKNEM NEBUDE-LI PRŠET NEZMOKNEM ]  
[K o č k a   l e z e   d í r o u   p e s   o k n e m   n e b u d e - l i  
p r š e t   n e z m o k n e m   ]  
[KočkA LeZe DíRoU PeS OkNeM NeBuDe-lI PrŠeT NeZmOkNeM ]
```

## 12.4.2. Pozorovatel (*Observer*)

- zavádí vztah 1 : N (pozorovaný : pozorovatelé) mezi objekty reagujícími na změnu stavu pozorovaného nebo na jím sledované události
  - pozorovatelé se u pozorovaného jednorázově přihlásí
  - pozorovaný je průběžně informuje
  - často je k dispozici i možnost odhlášení pozorovatele ze seznamu
- komunikace je **jednosměrná**
- termín **událostmi řízené programování**
- jiné názvy vzoru – Posluchač (*Listener*), Vydavatel–Předplatitel (*Publisher–Subscriber*)
- příklady z Java Core API:
  - obsluha všech událostí z GUI

- dvojice `java.util.Observer` a `java.util.Observable`
- pro základní využití výhod tohoto návrhového vzoru používáme knihovní dvojici `Observer`–`Observable`
- logicky chybné (prohozené) pojmenování
- Pozorovaný musí dědit `Observable` (**nevýhoda**)
- pak má k dispozici:

- `setChanged()` – nastala událost, o které se vyplatí informovat pozorovatele
- `notifyObservers(Object data)` – informování o události, kde `data` jsou možné upřesnění reportované události
  - ◆ nechceme-li upřesňovat, pak `null`

```
import java.util.Observable;

public class Pozorovany extends Observable {

    public void posliNovouZpravu(String novaZprava) {
        setChanged();
        notifyObservers(novaZprava);
    }
}
```

`Zapisovatel_Pozorovatel` implementuje jedinou metodu z `Observer` – `update(Observable obs, Object data)`

- `obs` – umožňuje posluchači identifikovat zdroj události – principiálně může pozorovat více zdrojů
- `data` – upřesňující zpráva

```
import java.util.Observable;
import java.util.Observer;

public class Zapisovatel_Pozorovatel implements Observer {
    @Override
    public void update(Observable obs, Object data) {
        System.out.println("Zapisuji zprávu: " + data);
    }
}
```

`Archivator_Pozorovatel` další typ pozorovatele

```
import java.util.Observable;
import java.util.Observer;

public class Archivator_Pozorovatel implements Observer {
    @Override
    public void update(Observable obs, Object data) {
        System.out.println("Archivuji zprávu: " + data);
    }
}
```

## Informator\_Pozorovatel další typ pozorovatele

```
import java.util.Observable;
import java.util.Observer;

public class Informator_Pozorovatel implements Observer {
    @Override
    public void update(Observable obs, Object data) {
        System.out.println("Informuji o zprávě: " + data);
    }
}

public class PredavaniNovychZprav_Aplikace {

    public static void main(String[] args) {
        Pozorovany vysilac = new Pozorovany();

        // vytváření a registrace posluchačů
        vysilac.addObserver(new Zapisovatel_Pozorovatel());
        vysilac.addObserver(new Informator_Pozorovatel());
        vysilac.addObserver(new Archivator_Pozorovatel());

        // zaslání zprávy všem registrovaným
        vysilac.posliNovouZpravu("Začínáme");
        vysilac.posliNovouZpravu("Pokračujeme");
        vysilac.posliNovouZpravu("Končíme");
    }
}
```

### ■ vypíše:

```
Archivuji zprávu: Začínáme
Informuji o zprávě: Začínáme
Zapisuji zprávu: Začínáme
Archivuji zprávu: Pokračujeme
Informuji o zprávě: Pokračujeme
Zapisuji zprávu: Pokračujeme
Archivuji zprávu: Končíme
Informuji o zprávě: Končíme
Zapisuji zprávu: Končíme
```

## 12.4.3. Prostředník (*Mediator*)

- odstraňuje vzájemné vazby mezi množinou navzájem komunikujících objektů tím, že zavede objekt, který bude prostředníkem mezi komunikujícími objekty (telefonní ústředna)
  - tím se ruší vzájemná přímá závislost mezi jednotlivými objekty
- komunikace je **obousměrná**
- často se implementuje pomocí návrhového vzoru Pozorovatel

Prostrednik využije všech možností `java.util.Observable`

```

import java.util.Observable;

public class Prostrednik extends Observable {

    public void predejZpravu(String zprava) {
        setChanged();
        notifyObservers(zprava);
    }
}

```

GrafickyObjekt využívá možností `java.util.Observer`

- `kdoJsem` – je unikátní pojmenování (ID) příjemce
- příslušný prostředník je předán v konstruktoru (*dependency injection*)
- `update()` – přijímá zprávy od prostředníka a dál je filtruje, zda je zpráva určená pro něj
- `posliZpravu()` – posílá svoji novou zprávu prostředníkovi a ten zajistí její distribuci všem registrovaným

```

import java.util.Observable;
import java.util.Observer;

public class GrafickyObjekt implements Observer {
    private Prostrednik prostrednik;
    private String kdoJsem;

    public GrafickyObjekt(Prostrednik prostrednik, String kdoJsem) {
        this.prostrednik = prostrednik;
        this.kdoJsem = kdoJsem;
    }

    @Override
    public void update(Observable obs, Object novaZprava) {
        String zprava = (String) novaZprava;
        String prijemce = zprava.substring(0, 1).toUpperCase();
        if (kdoJsem.equals(prijemce) || prijemce.equals("*")) {
            System.out.println(kdoJsem + " děkuje za zprávu: " + zprava.substring(1));
        }
    }

    // komu = "*" -> vsem
    public void posliZpravu(String komu, String zprava) {
        prostrednik.predejZpravu(komu + zprava);
    }
}

public class PredavaniZprav_Aplikace {

    public static void main(String[] args) {
        Prostrednik prostrednik = new Prostrednik();
        GrafickyObjekt a = new GrafickyObjekt(prostrednik, "A");
        GrafickyObjekt b = new GrafickyObjekt(prostrednik, "B");
        GrafickyObjekt c = new GrafickyObjekt(prostrednik, "C");
    }
}

```

```

// vytváření a registrace posluchačů
prostrednik.addObserver(a);
prostrednik.addObserver(b);
prostrednik.addObserver(c);

// zaslání zprávy všem registrovaným
a.posliZpravu("b", "Přejdi do pozadí");
a.posliZpravu("*", "Překreslete se");
b.posliZpravu("a", "Nezobrazuji se");
c.posliZpravu("b", "Překresli se");
}
}

```

■ vypíše:

```

B děkuje za zprávu: Přejdi do pozadí
C děkuje za zprávu: Překreslete se
B děkuje za zprávu: Překreslete se
A děkuje za zprávu: Překreslete se
A děkuje za zprávu: Nezobrazuji se
B děkuje za zprávu: Překresli se

```

## 12.4.4. Stav (State)

- zavedením několika vnitřních stavů řeší výrazný rozdíl mezi chováním objektu v jeho navenek různých stavech
  - změnu stavu objektu řeší záměnou objektu reprezentujícího stav
  - objekt pak vypadá, jako by měnil svoji třídu
- bez znalosti tohoto návrhového vzoru se podobná úloha řeší pomocí (rozsáhlých) příkazů switch a/nebo větví if–else–if
- tím, že se používají (vnitřní) „jednostavové“ třídy, je jejich kód výrazně jednodušší
  - další možné stavy se pak přidávají velmi snadno a pouze na jednom místě třídy – nikoliv v mnoha jejích metodách
- příklad ze života – kurzor myši

`AVnitrniStav` společný předek vnitřních stavů

- zde se stav bude lišit jen rozdílným způsobem výpisu

```

public abstract class AVnitrniStav {
    public void vypisNazevVnitrnihostavu() {
        System.out.print(getClass().getSimpleName() + "> ");
    }

    public abstract void vypis(String retezec);
}

```

Maly\_VnitriStav – jeden z možných vnitřních stavů

- při důkladnějším skrývání implementace by to byla vnitřní třída vnějšího stavu

```
public class Maly_VnitriStav extends AVnitriStav {  
    @Override  
    public void vypis(String retezec) {  
        vypisNazevVnitrihoStavu();  
        System.out.println(retezec.toLowerCase());  
    }  
}
```

dtto

```
public class Velky_VnitriStav extends AVnitriStav {  
    @Override  
    public void vypis(String retezec) {  
        vypisNazevVnitrihoStavu();  
        System.out.println(retezec.toUpperCase());  
    }  
}
```

dtto

```
public class Velbloud_VnitriStav extends AVnitriStav {  
    @Override  
    public void vypis(String retezec) {  
        vypisNazevVnitrihoStavu();  
        String pomocne = retezec.toUpperCase();  
        for (int i = 0; i < pomocne.length(); i++) {  
            char znak = pomocne.charAt(i);  
            znak = (i % 2 == 0) ? znak : Character.toLowerCase(znak);  
            System.out.print(znak);  
        }  
        System.out.println();  
    }  
}
```

VnejsiStav má v sobě jako konstanty všechny dostupné vnitřní stavы

- aktualniVnitriStav – může se zvnějšku měnit pomocí setru
- jeho metoda vypis(String retezec) jen deleguje pravomoc na metodu z aktuálního vnitřního stavu

```
public class VnejsiStav {  
    public static final AVnitriStav MALY = new Maly_VnitriStav();  
    public static final AVnitriStav VELKY = new Velky_VnitriStav();  
    public static final AVnitriStav VELBLOUD = new Velbloud_VnitriStav();  
  
    private AVnitriStav aktualniVnitriStav = MALY;  
  
    public void setStav(AVnitriStav novyVnitriStav) {  
        this.aktualniVnitriStav = novyVnitriStav;
```

```

}

public void vypis(String retezec) {
    aktualniVnitrniStav.vypis(retezec);
}
}
```

ExterniUdalost třída, která podle vnějších podmínek mění vnější stav

- změna stavu je na základě délky vypisovaného řetězce
- tato třída není vždy nutná – VnejsiStav by se mohl měnit i sám

```

public class ExterniUdalost {
    private VnejsiStav stav;

    public ExterniUdalost(VnejsiStav stav) {
        this.stav = stav;
    }

    public void zmenVnitrniStav(String retezec) {
        int delka = retezec.length();
        if (delka < 4) {
            stav.setStav(VnejsiStav.ELKY);
        }
        else if (delka < 7) {
            stav.setStav(VnejsiStav.MALY);
        }
        else {
            stav.setStav(VnejsiStav.ELBLOUD);
        }
    }
}

public class Aplikace {

    public static void main(String[] args) {
        VnejsiStav stav = new VnejsiStav();
        ExterniUdalost eu = new ExterniUdalost(stav);

        String[] retezce = {"Jedna", "Dva", "Dvanact", "Osm", "Jedenact" };
        for (String retezec : retezce) {
            eu.zmenVnitrniStav(retezec);
            stav.vypis(retezec);
        }
    }
}
```

- vypíše:

```

Maly_VnitrniStav> jedna
Velky_VnitrniStav> DVA
Velbloud_VnitrniStav> DvAnAct
```

Velky\_Vnitrnistav> OSM

Velbloud\_Vnitrnistav> JeDeNaCt

