# Přednášky z OOP

# Obsah

# 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 [skr-9] znamená učební text Herout, P.: **Počítače a programování 1**, str. 9
    - učební text je dostupný na Courseware PPA1 i OOP ve formě PDF

## 1.1.1. Výhody Java

- výhod je mnoho, z našeho pohledu stačí:
  - přenositelná (platformně nezávislá)
  - 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`
- viz též [skr-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

- pojem virtuální stroj – nadstavba nad platformu
  - jedná se o interpretr mezijazyka – `.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*
  - vývoj tzv. RIA aplikací – *Rich Internet Applications* – uživatelské rozhraní
  - konkurent Adobe Flash a Microsoft Silverlight

## 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
  - 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 OO programování
- `www.bluej.org`
- popis instalace verze 3.0.5 je na CourseWare
- 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`
- 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. PMD

- nejedná se o vývojové prostředí, ale o nástroj pro zvyšování kvality zdrojového kódu
- `pmd.sourceforge.net`
- lze použít samostatně (případně přes JJ), 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 rozsáhlejších 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 naprogramovat
  - 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 téměř 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

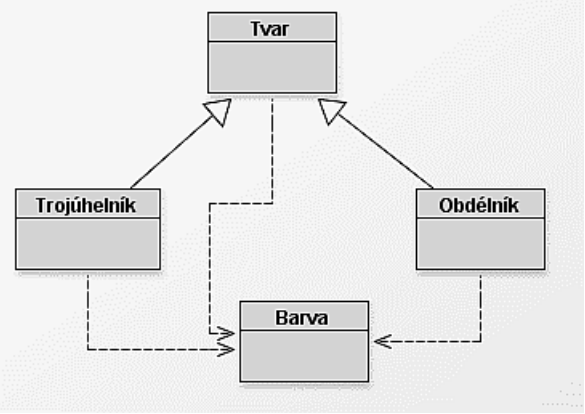
- strukturovaný program = posloupnost příkazů
  - analýza: vymýšlejí se postupy
  - stavební kameny: procedury/funkce; (proměnné)
  - výsledek: většinou samostatný program
- objektově orientovaný program = množina tříd a jejich instancí, které si posílají zprávy
  - analýza: definují se účastníci a jejich 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 13 diagramů (verze 2.0)
- 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; JUnit proto zavádí následující technologii (kterou přejímá BlueJ):
    - testovaná sada objektů označovaná jako **testovací přípravek** a 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

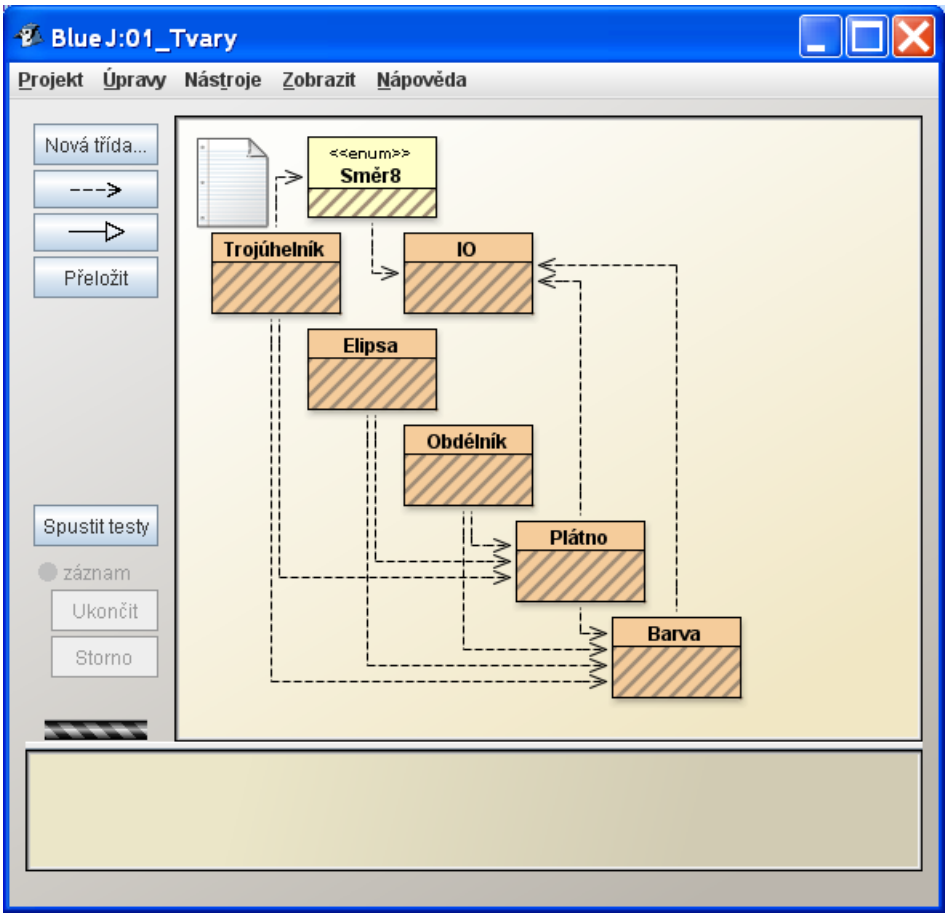
- každý program je model reálného či virtuálního světa, který sestává z objektů
- programovací jazyk musí podporovat konstrukce, které umí s objekty pracovat
- objekt je vše, co lze označit podstatným jménem, např.
  - „skutečné“ objekty (auto, pes, obdélník, ...)
  - vlastnosti (velikost, barva, směr, …)
  - 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 požádání 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 něco provést
  - zpráva má vždy adresáta
  - v aktivitě, kterou objekt vykonává jako reakci na zprávu, je skryt algoritmus její činnosti
    - 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 (někdo 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 – `Směr8` – 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
  - další soubory (`.class` a `.ctxt`) a adresář (`doc`) se vytvářejí při překladu či generování dokumentace a je možné je po uzavření projektu smazat

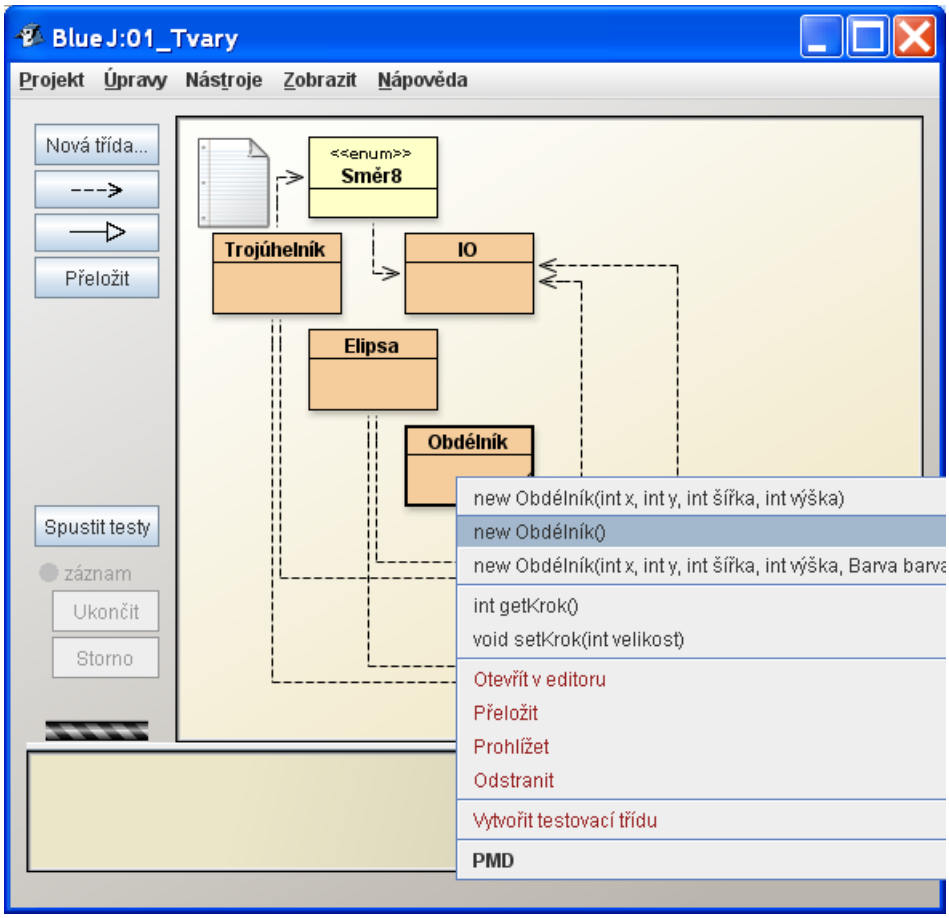
    ### Varování

    Pokud je projekt otevřen, neměnit z vnějšku jeho soubory!
- nevýhoda BlueJ – velké množství oken

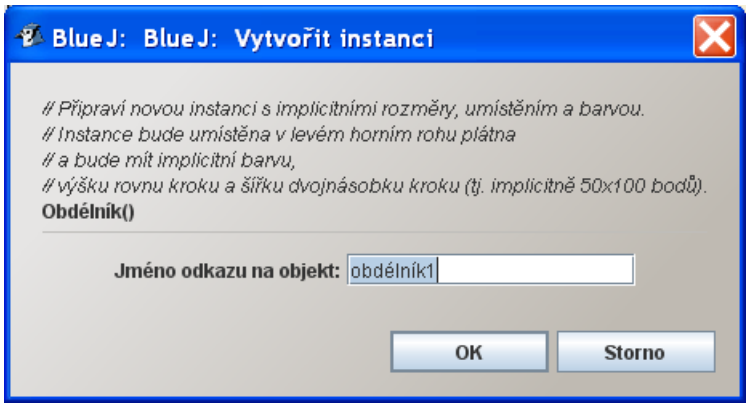
## 1.2.2.1. Zasílání zpráv

### Poznámka

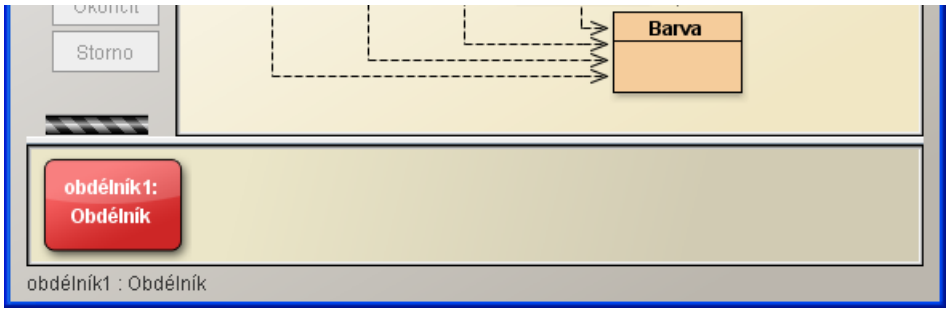
Podrobnosti o zprávách viz souhrnně dále.



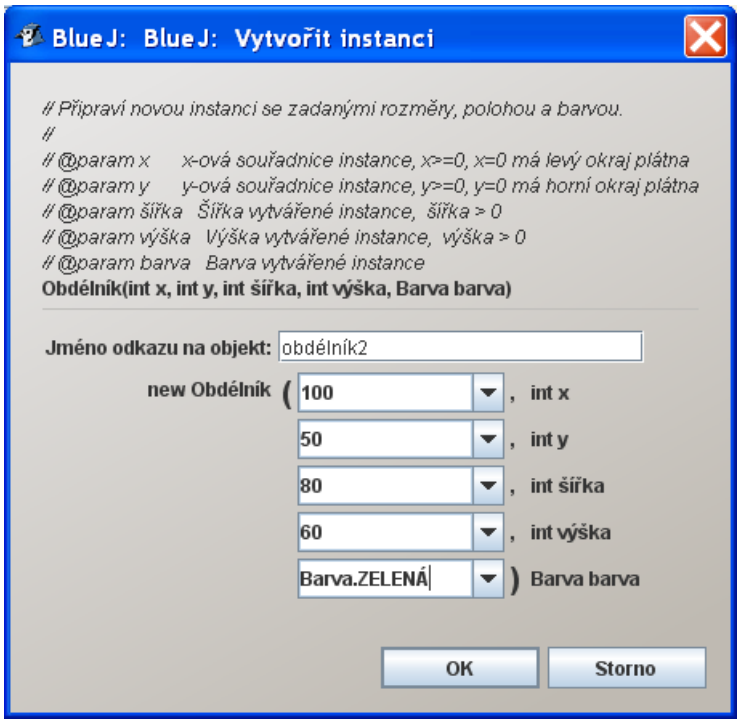

- v interaktivním režimu lze zaslat zprávu třídě zadáním příkazu z její místní nabídky
- typicky vytváříme novou instanci pomocí `new` (volání konstruktoru)
  - je vidět i možné přetížení – tři konstruktory vytvářející `Obdélník`, kdy parametry upřesňují zasílanou zprávu
  - k instanci máme přístup vždy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



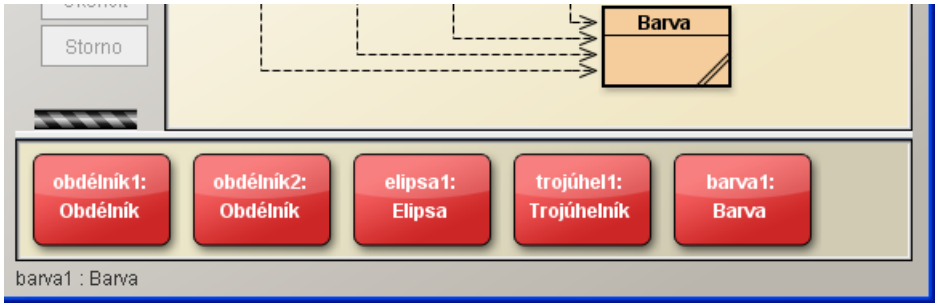

- pravidla pro identifikátory v Javě [skr-24]:
  - identifikátor je libovolná posloupnost písmen, číslic a _ nezačínající číslicí
  - třída – první velké písmeno `Obdélník`, `ZavodniAuto`
  - proměnná – první malé písmeno `obdélník1`, `mojeCerveneZavodniAuto`
  - balík – všechna malá písmena `auta`
  - 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



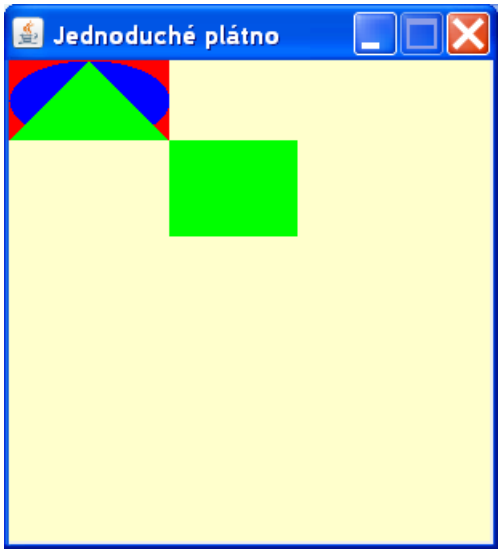

- odkazů lze vytvořit libovolné množství
  - pokud má konstruktor parametry, BlueJ je umožní zadat v dialogovém boxu



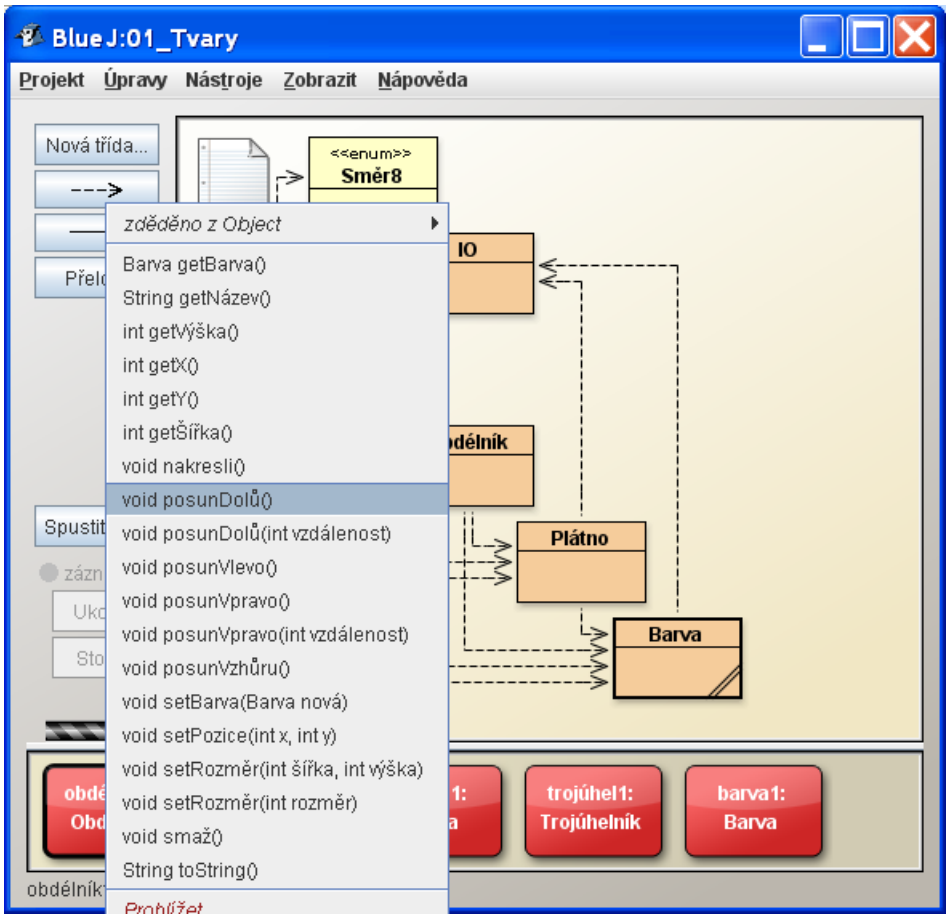

- výsledek po použití několika konstruktorů



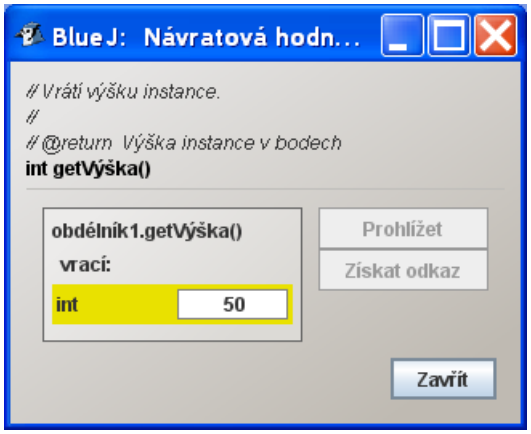

- tato aplikace umí zobrazit některé vzniklé objekty na plátno



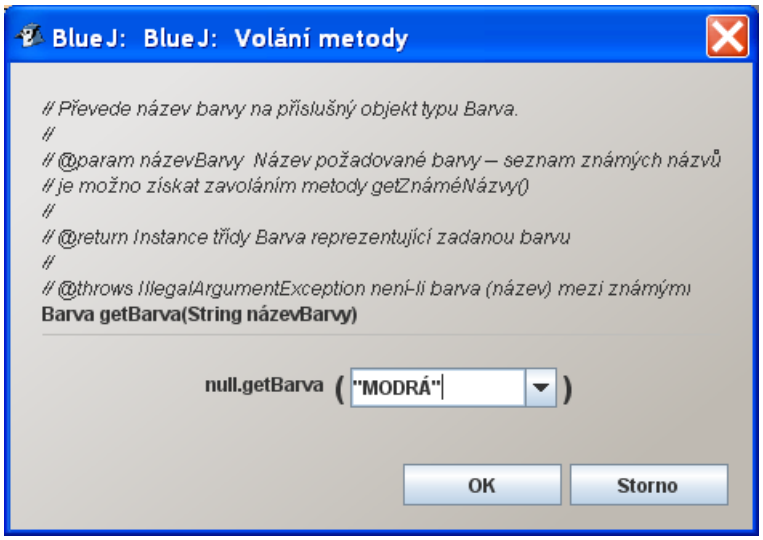

- existujícím instancím je možné interaktivně zasílat zprávy, opět z místní nabídky



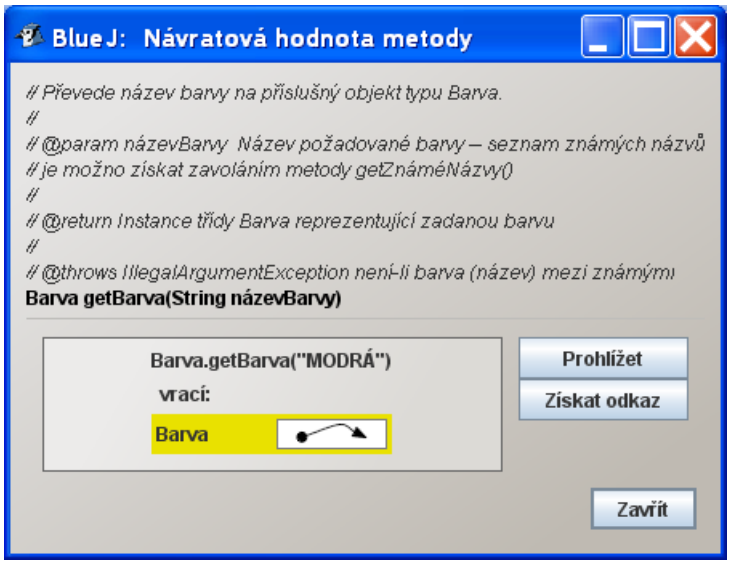

- některé metody vracejí hodnotu, kterou BlueJ zobrazí v dialogovém boxu

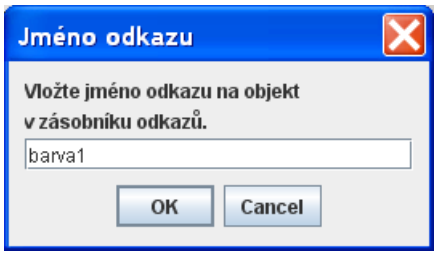



- třída `Barva` je speciální třída, jejíž instance se nezískávají pomocí konstruktoru, ale voláním jednoduché tovární metody (např. `Barva getBarva(String názevBarvy)`) – Pozor: uvozovky nutné `"MODRÁ"`!




- 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 často `void`
    - není-li `void`, pak často chybová zpráva (akce se podařila / nepodařila)

- vrací informaci o svém vnitřním stavu – obdélník má červenou barvu
  - ♦ návratová hodnota číslo nebo odkaz na jiný objekt
- mění vnitřní stav – obdélník se posunul dolů (změnil svoji y-souřadnici)
  - ♦ návratová hodnota často `void`
- vytvářejí nové objekty – `new()` (volání konstruktoru) nebo `getBarva()` (jednoduchá tovární metoda)
  - ♦ návratová hodnota 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 `Obdélník.getKrok()`
- nad objektem – jméno proměnné, ve které je odkaz `obdélník1.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
- množina přípustných hodnot, např. čísla od 1 do 100 nebo barvy červená a modrá
- 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 [skr-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 ± 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 `Obdélník`
  - 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
  - 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
- `String` (řetězec) je objektový typ, ale často používaný
  - má některé vlastnosti primitivních typů, např. nepotřebuje konstruktor `new()`
  - hodnoty instance `String` jsou v uvozovkách `"MODRÁ"`
    - bez uvozovek `MODRÁ` 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 nástrojů – neměnit
- `get` a `is` nemají parametry
- `set` nevracejí hodnotu (jsou `void`)

### Poznámka

Existuje možnost, že setr vrací původní hodnotu – používá se v případech, kdy se na hodnotu dané vlastnosti ptáme spíše výjimečně – příklad vkládací metody do kolekcí `put()`

### 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
- 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 barva1 = Barva.getBarva("khaki")` ten následně předávat `obdelnik.setBarva(barva1)`
  - nebo vytvořit v parametru nový objekt, např. `obdelnik.setBarva(Barva.getBarva("khaki"))`
  - nebo použít již hotový objekt `obdelnik.setBarva(Barva.KHAKI)` – viz dále

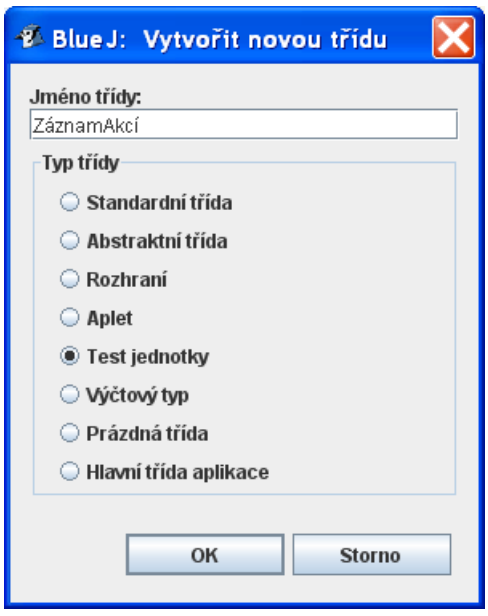
## 1.2.4. Zautomatizování interaktivního režimu BlueJ

- 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.

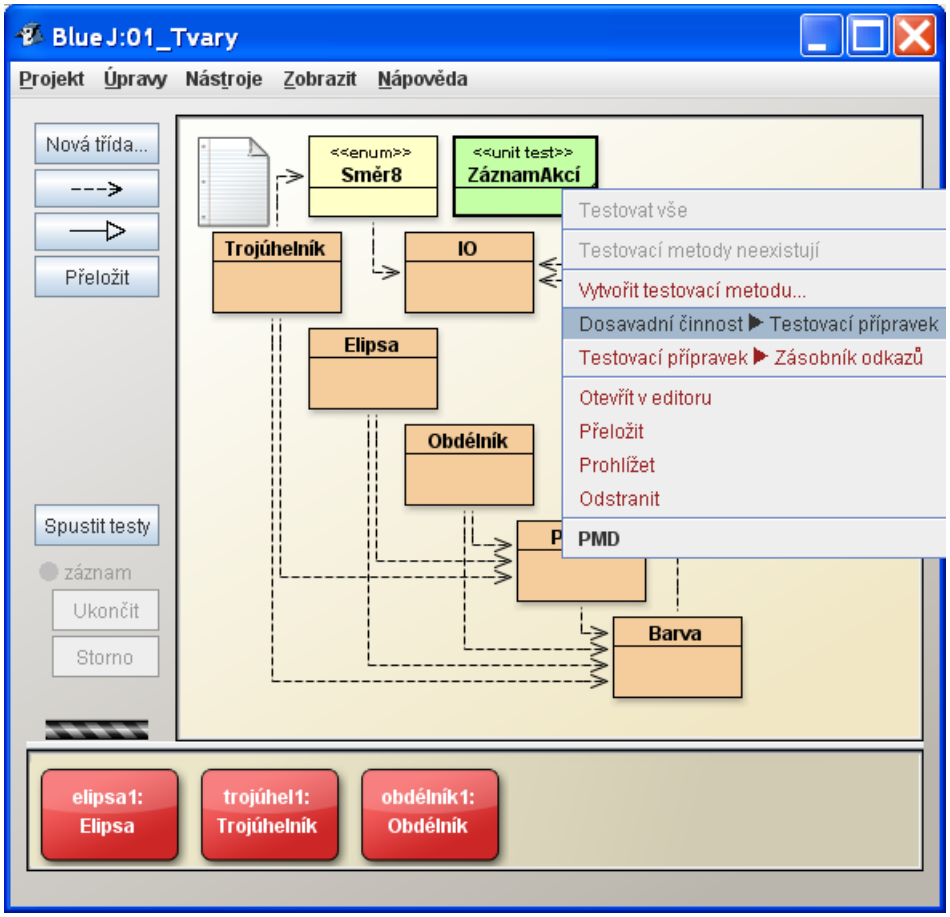
- BlueJ od verze 3.0.5 využívá JUnit 4.2 (s anotacemi)
- nejprve je nutno vytvořit testovací třídu: Úpravy/Nová třída



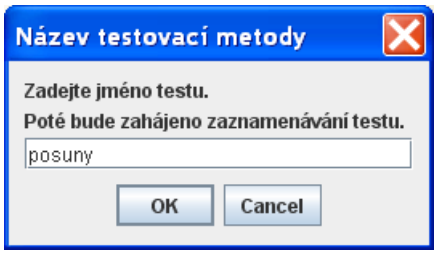

  - na pojmenování nezáleží, typicky je v jejím jménu slovo `Test`, ale protože ji teď budeme používat jen pro záznam akcí, je jméno `ZáznamAkcí`

- v diagramu tříd má stereotyp `<<unit test>>`

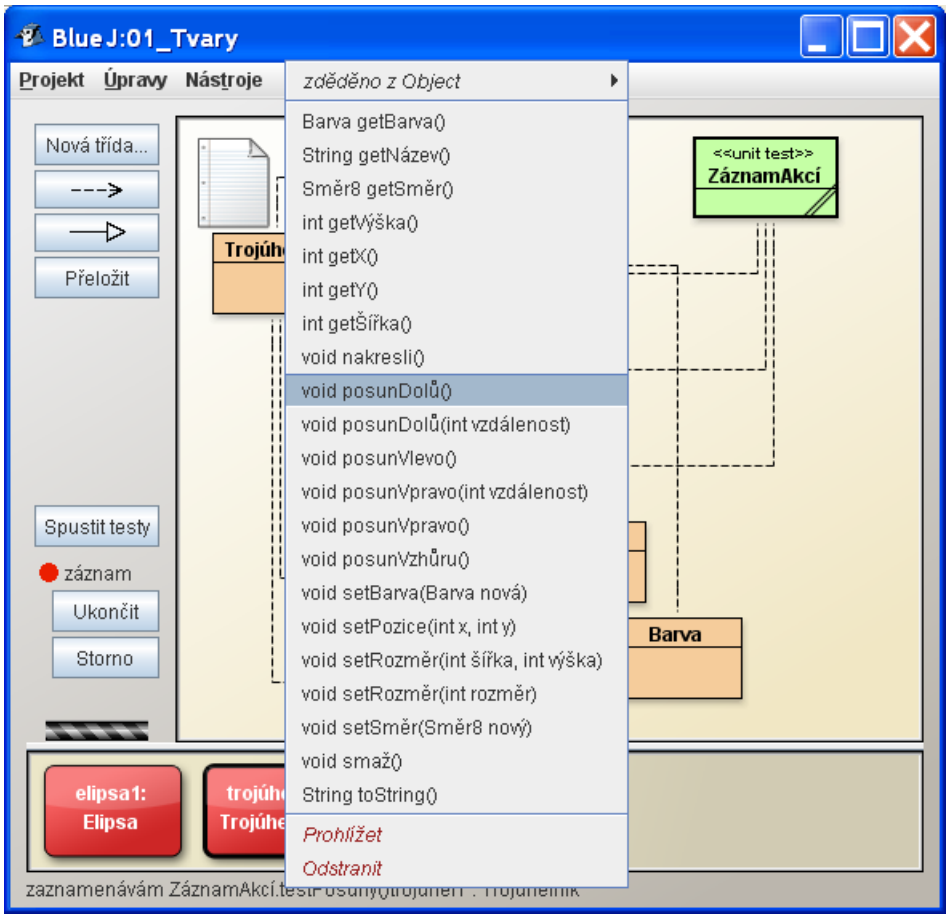
- pak je vhodné: Nástroje/Restartovat virtuální stroj
- využíváme dvou možností záznamu aktivit (obě z místní nabídky)




- **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




- spouštění zaznamenaných akcí
  - Testovací přípravek -> Zásobník odkazů
  - výběr jména testovací metody – vždy před svojí činností aktualizuje testovací přípravek
- oprava zaznamenaných akcí
  - otevřít třídu `ZáznamAkcí` v editoru – a tam editovat (čísla, jména proměnných, ...)
    - testovací přípravek

```
public class ZáznamAkcí
{
  private Elipsa elipsa1;
  private Trojúhelník trojúhel1;
  private Obdélník obdélník1;

  @Before
  public void setUp()
  {
    elipsa1 = new Elipsa();
    trojúhel1 = new Trojúhelník();
    obdélník1 = new Obdélník();
  }
```

    - testovací metoda

```
@Test
public void posuny()
{
  elipsa1.posunDolů();
  elipsa1.posunDolů();
  trojúhel1.posunDolů();
  obdélník1.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 vždy skrytě využívají konstruktor
- instance se vytváří dvoukrokově
  1. začne virtuální stroj, který zajistí přidělení potřebné paměti – má za úkol 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 parametrů)
```

  - seznam parametrů – specifikuje podmínky vzniku 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.ZELENÁ);
    new Obdélník(45, 100, 10, 50, Barva.HNĚDÁ);
  }
}
```

- vytvoření nové instance

```
new Strom();
```

### Poznámka

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

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

Jakmile uživatel jakýkoliv konstruktor vytvoří, překladač již nic nedoplňuje.

# 2.2.1. Přetížené konstruktory

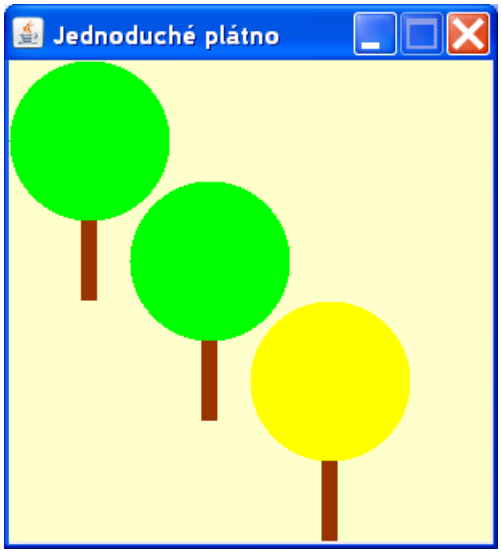
- velmi často existuje více verzí konstruktoru jedné třídy – jsou přetížené
- musí se lišit alespoň jedním z:
  - počtem parametrů
  - typy parametrů
  - pořadím parametrů

```
public class Strom {
  public Strom() {
    new Elipsa(0, 0, 100, 100, Barva.ZELENÁ);
    new Obdélník(45, 100, 10, 50, Barva.HNĚDÁ);
  }

  public Strom(int x, int y) {
    new Elipsa(x, y, 100, 100, Barva.ZELENÁ);
```

```
    new Obdélník(x + 45, y + 100, 10, 50, Barva.HNĚDÁ);
  }

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

- jako skutečné parametry se zadávají hodnoty, na jejichž základě objekt modifikuje svoji reakci na zprávu
  - typ předávané hodnoty musí přesně odpovídat deklarovanému typu daného parametru
  - jako hodnoty parametrů objektových typů se předávají odkazy na instance těchto typů (tříd)

```
new Strom();
new Strom(75, 75);
new Strom(150, 150, Barva.ŽLUTÁ);
```




## 2.2.1.1. Využití `this`

- každá instance má skrytý parametr `this`, který odkazuje na ni samu
  - pomocí konstrukce `this.` lze přistupovat k libovolným částem třídy
  - 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.ZELENÁ);
  }

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

- předávání řízení lze hezky ukázat pomocí debuggeru BlueJ

# 2.3. Atributy

- proměnné či konstanty popisující stav objektu (uchovávají hodnoty potřebné pro správnou funkci programu)
  - v JDK a v BlueJ jsou označované jako *fields*
- každá třída definuje, jaké bude mít atributy a ty pak přeberou 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.ČERNÁ`)
    - pro použití `public` musí být skutečně vážný důvod
  - `private` (soukromý) – přístupný jen pro vlastní třídu/instanci
    - typické nastavení, čtou se zvnějšku 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 atributech 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.KHAKI`
  - `obdelnik.setBarva(Barva.KHAKI)`
  - to je rozdíl, kdy dříve bylo nutné požádat o vrácení nové barvy `Barva.getBarva(String název)`, tento odkaz uložit do pomocné proměnné `khaki` a tu pak použít při nastavování `obdelnik.setBarva(khaki)`
- příklady třídních a instančních atributů

```
//Instanční proměnná
private int velikost;

//Inicializovaná instanční proměnná
private int počet = 0;

//Instanční konstanta
private final int MAX = 10;

//Incializovaná třídní proměnná
private static Barva implicitníBarva = Barva.ČERNÁ;

//Incializovaná třídní konstanta
private static final Barva IMPLICITNÍ_BARVA = Barva.ČERNÁ;
```

- příklady inicializací v deklaraci (platí pro třídní i instanční, pro proměnné i konstanty):
  - přiřazením předem známé hodnoty

```
private int šířka = 100;
```

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

```
private Barva barvaKmene = Barva.HNĚDÁ;
private int výška = šířka;
```

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

```
private Obdélník kmen = new Obdélník(0, 0, šířka, výška, barvaKmene);
private Elipsa koruna = new Elipsa();
```

- přiřazením hodnoty vrácené nějakou metodou:

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

- přiřazením hodnoty výrazu:

```
private String implicitníJméno = koruna.getBarva().getNázev();
private int výška = šířka * 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 by se dala 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 beze zbytku pravidlo, že každý atribut má getr a setr
  - skryté atributy, např. u `Strom` atributy `koruna` a `kmen`

# 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 atribut;
  static int třídníAtribut;

  void metoda(int formálníParametr) {
    int lokálníProměnná = formálníParametr;
    this.atribut = 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

- přístupová práva (`private`, `public`) nemají smysl

## 2.3.3.2. Inicializace

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

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

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

  void setAtribut(int par) {
    this.atribut = 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.atribut = 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 pomocí setru
- 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
- lokální proměnné
  - běžně měníme pomocí přiřazovacích příkazů

```
void metoda(int formálníParametr) {
  int lokálníProměnná = formálníParametr / 123;
  this.atribut = 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, 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.HNĚDÁ;
  private static final Barva IMPL_BARVA_KORUNY = Barva.ZELENÁ;
  private static final Barva BARVA_PODZIMU = Barva.ŽLUTÁ;
  private final Elipsa koruna;
```

```
  private final Obdélník 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 Obdélník(x + 45, y + 100, 10, 50, BARVA_KMENE);
    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 [skr-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()`, `getVýška()`
  - změnit vlastnosti objektu – setry – `zežloutni()`, `setBarvaKoruny()`
  - provádět další činnosti spojené s objektem, např. jeho vykreslení – `nakresli()`, `smaž()`
- ke statickým atributům mohou přistupovat pouze statické metody
  - instanční metody mohou ke všem atributům

```
  public int getX() {
    return this.x;
  }

  public int getY() {
    return koruna.getY();
  }
```

```
public int getVýška() {
  return koruna.getVýška() + kmen.getVýška();
}

public boolean isPodzim() {
  return koruna.getBarva() == BARVA_PODZIMU;
}

public void zežloutni() {
  setBarvaKoruny(BARVA_PODZIMU);
}

public void postupněŽežloutni() {
  setBarvaKoruny(Barva.KHAKI);
  IO.čekej(500);
  setBarvaKoruny(BARVA_PODZIMU);
}

public void setBarvaKoruny(Barva barva) {
  koruna.setBarva(barva);
}

public void nakresli() {
  koruna.nakresli();
  kmen.nakresli();
}

public void smaž() {
  koruna.smaž();
  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

```
public boolean isPodzim() {
  Barva barva = koruna.getBarva();
  boolean jePodzimní = barva.equals(BARVA_PODZIMU);
  return jePodzimní;

  /* 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 stejné jako:

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

## 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 počet = 0;

/** Pořadí kolikátá byla daná instance vytvořena v rámci třídy. */
private final int POŘADÍ = ++počet;
```

```
  /** Název sestávající z názvu třídy a pořadí instance */
  private final String NÁZEV = "Strom_" + POŘADÍ;

  public String getNázev() {
    return NÁZEV;
  }
```

## 2.4.3.1. Inicializace atributů třídy

- pokud nelze atribut třídy inicializovat jednoduchým příkazem 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, 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

      ```
      jednoducheCislo = STATICKA_KONSTANTA * 3;
      ```

    - 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 záleží)
  - používá se zásadně pouze jeden
  - 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 inicializační výrazy se uvádějí v konstruktoru

# 2.4.4. Odložená inicializace (*lazy initialization*)

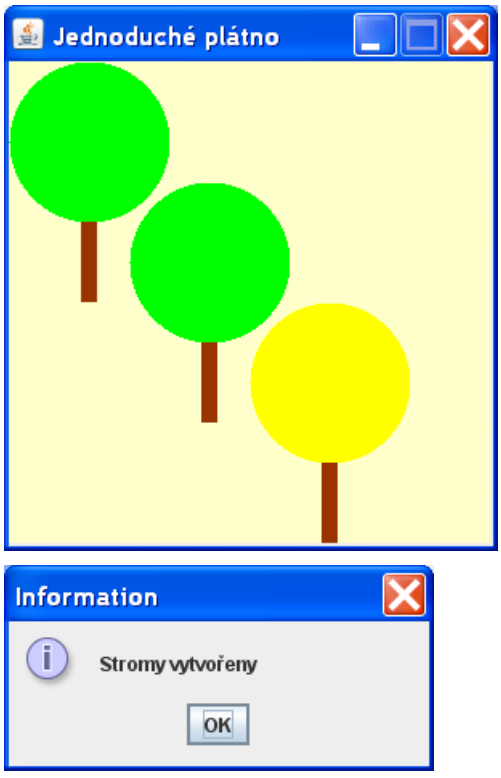
- občas má třída instanční či statický atribut s hodnotou, jejíž zjišťování je náročné a přitom ji nikdo ani nemusí využít
- tuto hodnotu nenastavujeme v konstruktoru, ale v metodě `get()`, která ji jednou vypočte a vrátí a navíc ji uloží do atributu
- při každém dalším volání vrací hodnotu atributu
- doporučuje se, aby atribut byl objekt (v případě primitivních datových typů obalová třída – zde `Integer`), aby se ještě nenastavený atribut snadno poznal podle hodnoty `null`

```
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;
  }

}
```

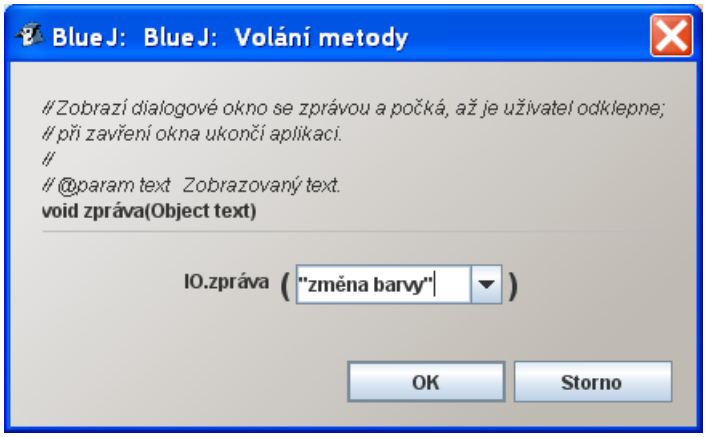
# 2.4.5. Využití testů v BlueJ

- 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.zpráva()`

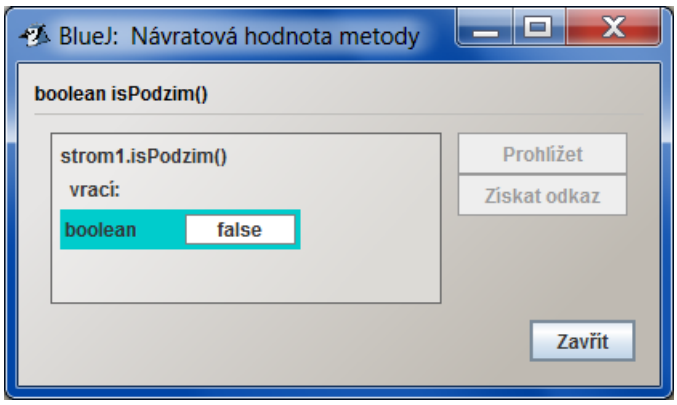
```
@Before
public void setUp()
{
  strom1 = new Strom();
  strom2 = new Strom(100, 100);
  strom3 = new Strom(200, 150, Barva.ŽLUTÁ);
  IO.zpráva("Stromy vytvořeny");
}
```



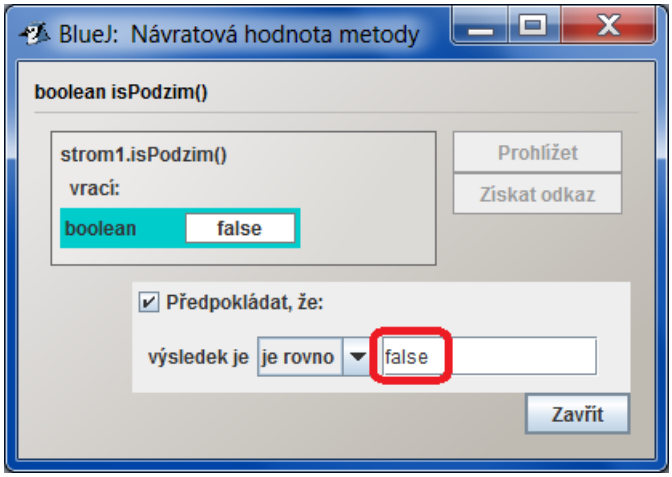

- podobnou zprávu je vhodné vložit i mezi jednotlivé příkazy testovací metody – `IO.zpráva("změna barvy");` buď interaktivně, nebo později úpravou kódu



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



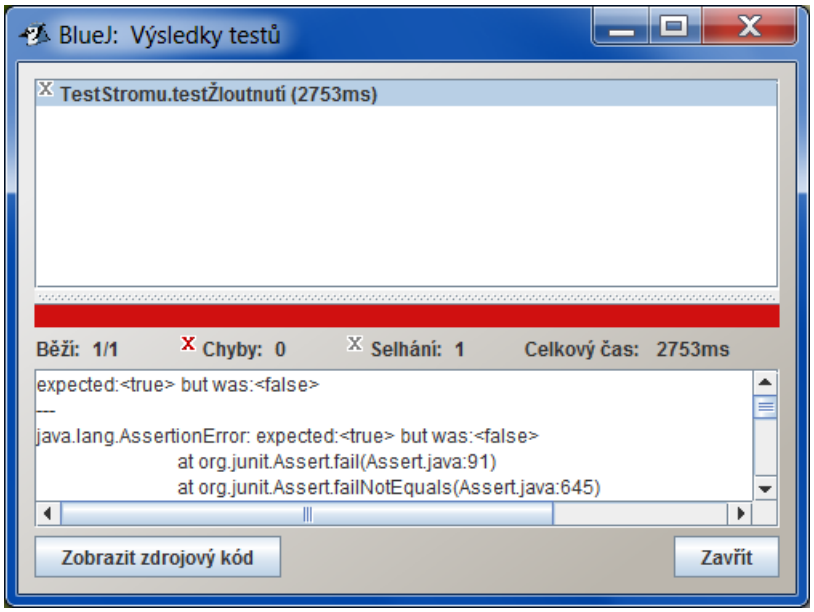

- voláme-li při tutéž metodu při záznamu testů, zobrazí BlueJ okénko s aktuální návratovou hodnotou a možným vyhodnocením testu



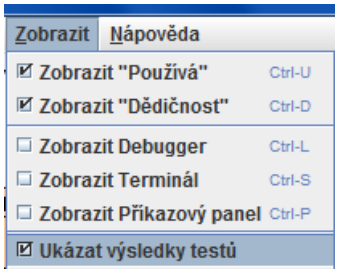

- očekáváme-li, že hodnota bude `false` (strom je zelený), doplníme tuto hodnotu a v testu bude následně ověřena – generovaný kód je `assertEquals(false, strom1.isPodzim());`

```
@Test
public void testŽloutnutí()
{
  assertEquals(false, strom1.isPodzim());
  strom1.zežloutni();
  assertEquals(true, strom1.isPodzim());
  IO.zpráva("změna barvy");
  strom2.postupněZežloutni();
  IO.zpráva("změna barvy");
  assertEquals(true, strom3.isPodzim());
}
```

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



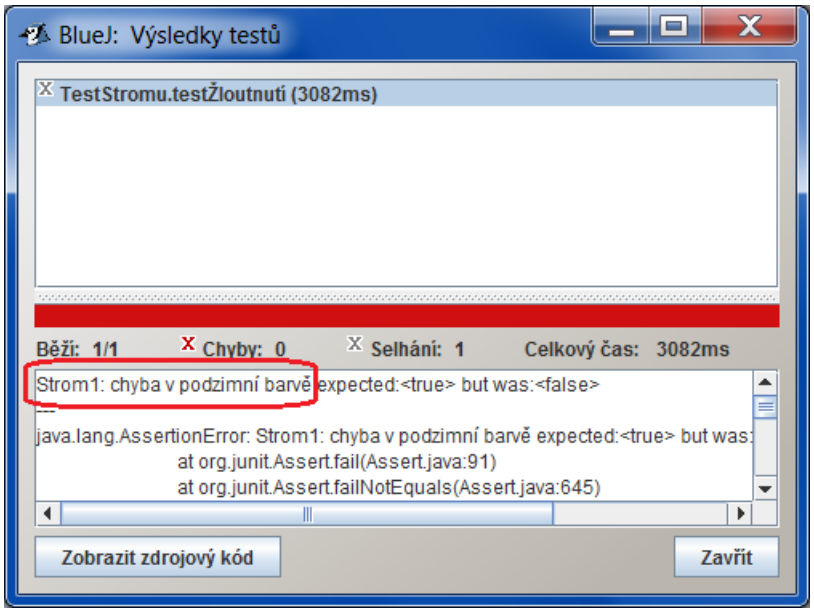

- musíme mít ale před tím v hlavním okně BlueJ zapnuto zobrazování výsledků testů




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

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

  - zadáme-li špatnou hodnotu (zde `true`), 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 testSouřadniceX() {
  assertEquals(75, strom2.getX());
}
```

  - opět je vhodné doplnit chybovým výpisem

```
@Test
public void testSouřadniceX() {
  assertEquals("Chybná 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 `žloutnutíJednohoStromu()` je `private`, protože je pomocná a není před ní anotace `@Test`

```
private void žloutnutíJednohoStromu(KomentovanýStrom strom) {
  strom.zežloutni();
  assertEquals(true, strom.isPodzim());
  IO.zpráva("změna barvy");
}
```

```
@Test
public void testŽloutnutíVšech() {
  žloutnutíJednohoStromu(strom1);
  žloutnutíJednohoStromu(strom2);
  žloutnutíJednohoStromu(strom3);
}
```

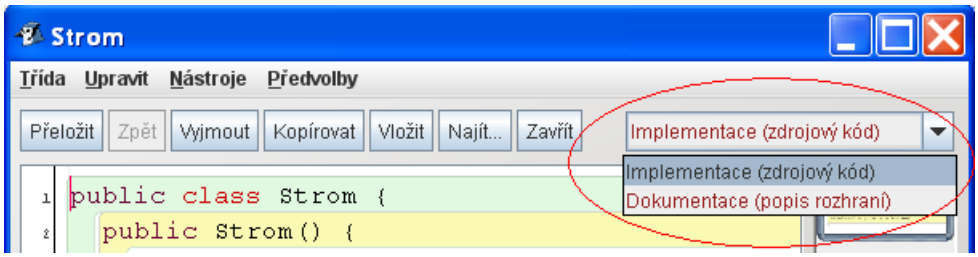
- v BlueJ lze samostatně testovat každou třídu – místní nabídka "Vytvořit testovací třídu"
  - je sdružena s testovanou třídou

# 2.5. Rozhraní versus implementace

- současné programy (třídy) mají dvě podoby
  - vnější – reprezentovanou rozhraním
    - co entita (program, třída, metoda, ...) umí a jak se s ním komunikuje
    - rozhraní pouze popisuje (slibuje), jak to bude vypadat
    - rozhraní je souhrnná informace, kterou potřebuje uživatel pro využívání entity, jejíž vnitřek ho nezajímá
    - tato informace má být popsána 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 zveřejnění zůstávat stejné
    - jsou-li navenek známé detaily 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
- pro možnost fyzického oddělení rozhraní a implementace ve zdrojovém kódu existuje v Javě konstrukce `interface` (rozhraní) – viz později

### Poznámka

BlueJ ideálně využívá možností zobrazit u třídy pohled na implementaci (zdrojový kód) a na rozhraní (dokumentaci).




# 2.5.1. Signatura versus kontrakt

- rozhraní můžeme rozdělit na dvě složky:
  - signatura – charakteristiky, které může při použití zkontrolovat překladač
    - dostupnost dané entity (`public`, `private`, …) a její název
    - u metod navíc seznam typů parametrů a typ návratové hodnoty
    - atributů jejich typ, ...
  - kontrakt – specifikuje informace, které překladač zkontrolovat nedokáže
    - podmínky, které je třeba dodržet, např. 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 kontraktu by měl být uveden v dokumentačních komentářích
- příklad pro metodu

```
/*****************************************************************************
 * Změní postupně barvu koruny na podzimní barvu.
 */
public void postupněZežloutni() {
  setBarvaKoruny(Barva.KHAKI);
  IO.čekej(500);
  setBarvaKoruny(BARVA_PODZIMU);
}
```

  - co říká rozhraní–signatura:
    - jmenuje se `postupněZežloutni`
    - nemá žádné parametry
    - nic nevrací
  - 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 `čekej()` třídy `IO`, které předá počet 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 co a proč příkaz či metoda či atribut dělá 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 vraťBarva()
//    {
//        setBarva(barva);
//    }
```

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

```
/*
    public void vraťBarva()
    {
        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 integrovaný, takže pohled na dokumentaci je přímo přístupný z editoru (druhý pohled je implementace)
  - 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 [skr-32] a [skr-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>`
- 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` – metodou vyhazovaná výjimka
- `{@code název}` – název entity, který bude 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 si uživatel mohl libovolně přidávat
 *  vlastní barvy.
 *
 * @author   Rudolf PECINOVSKÝ
 * @version  3.00.001
 */
public class Barva {
```

■ komentování atributu

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

■ komentování metody

```
/*****************************************************************************
 * Existuje-li zadaná barva mezi známými, vrátí ji; v opačném případě
 * vytvoří novou barvu se zadanými parametry a vrátí odkaz na ni.
 *
 * @param red       Velikost červené složky
 * @param green     Velikost zelené složky
 * @param blue      Velikost modré složky
 * @param název     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 název )
  {
    return getBarva( red, green, blue, 0xFF, název );
  }
```

### Poznámka

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

## 2.6.1.1. Rozmísťování 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
- R.Pecinovský připravil šablonu pro třídu – použije se po Nová třída / Standardní třída a má obsah:

```
/* Soubor je ulozen v kodovani UTF-8.
 * Kontrola kódování: Příliš žluťoučký kůň úpěl ďábelské ódy. */


/**********************************************************
 * Instance třídy {@code Šablona} představují ...
 *
 * @author    jméno autora
 * @version   0.00.000
 */
public class Šablona
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================
//== PROMĚNNÉ ATRIBUTY TŘÍDY ==================================
//== STATICKÝ INICIALIZAČNÍ BLOK - STATICKÝ KONSTRUKTOR ========
//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================
//== PROMĚNNÉ ATRIBUTY INSTANCÍ ===============================
//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ TŘÍDY =======================
//== OSTATNÍ NESOUKROMÉ METODY TŘÍDY ==========================

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

    /**********************************************************
     *
     */
    public Šablona()
    {
    }



//== ABSTRAKTNÍ METODY ========================================
//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ INSTANCÍ ====================
//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ =======================
//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ==========================
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ =======================
//== INTERNÍ DATOVÉ TYPY ======================================
//== TESTY A METODA MAIN ======================================
```

```
//
//     /**********************************************************
//      * Testovací metoda.
//      */
//     public static void test()
//     {
//         Šablona instance = new Šablona();
//     }
//     /** @param args Parametry příkazového řádku - nepoužívané. */
//     public static void main(String[] args)  {  test();  }
}
```

- tato šablona se z počátku zdá být příliš složitá, ale poskytuje dobrý základ pro rozmísťování kódu
  - už jen to, že musíme trochu přemýšlet, do jaké části náš kód umístíme, znamená, že provedeme jeho analýzu
- ukázka dosavadní třídy `Strom` (zde `KomentovanýStrom`)

```
/**************************************************************************
 * Instance třídy {@code KomentovanýStrom} představují ukázkovou třídu pro
 * ukázku komentářů
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
public class KomentovanýStrom
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

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

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

  /** barva koruny stromu na podzim */
  private static final Barva BARVA_PODZIMU = Barva.ŽLUTÁ;


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

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


//== STATICKÝ INICIALIZAČNÍ BLOK - STATICKÝ KONSTRUKTOR =======================
//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================================

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

  /** Název sestávající z názvu třídy a pořadí instance */
```

```
  private final String NÁZEV  = "Strom_" + POŘADÍ;

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

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

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

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ TŘÍDY =====================================
//== OSTATNÍ NESOUKROMÉ METODY TŘÍDY ========================================

//############################################################################
//== 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 KomentovanýStrom() {
    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 KomentovanýStrom(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 KomentovanýStrom(int x, int y, Barva barvaKoruny) {
```

```
    koruna = new Elipsa(x, y, 100, 100, barvaKoruny);
    kmen = new Obdélník(x + 45, y + 100, 10, 50, BARVA_KMENE);
    this.x = x;
  }


//== ABSTRAKTNÍ METODY ==========================================================
//== 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 getNázev(){
    return NÁZEV;
  }

  /*****************************************************************************
   * 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 getVýška() {
    return koruna.getVýška() + kmen.getVýška();
  }
```

```
  /**************************************************************************
   * 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;
  }

//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ ================================

  /**************************************************************************
   * Změní barvu koruny na podzimní barvu.
   */
  public void zežloutni() {
    setBarvaKoruny(BARVA_PODZIMU);
  }

  /**************************************************************************
   * Změní postupně barvu koruny na podzimní barvu.
   */
  public void postupněZežloutni() {
    setBarvaKoruny(Barva.KHAKI);
    IO.čekej(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 smaž() {
    koruna.smaž();
    kmen.smaž();
  }


//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ====================================
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ =================================
//== INTERNÍ DATOVÉ TYPY ================================================
//== TESTY A METODA MAIN ================================================
//
//    /**************************************************************************
```

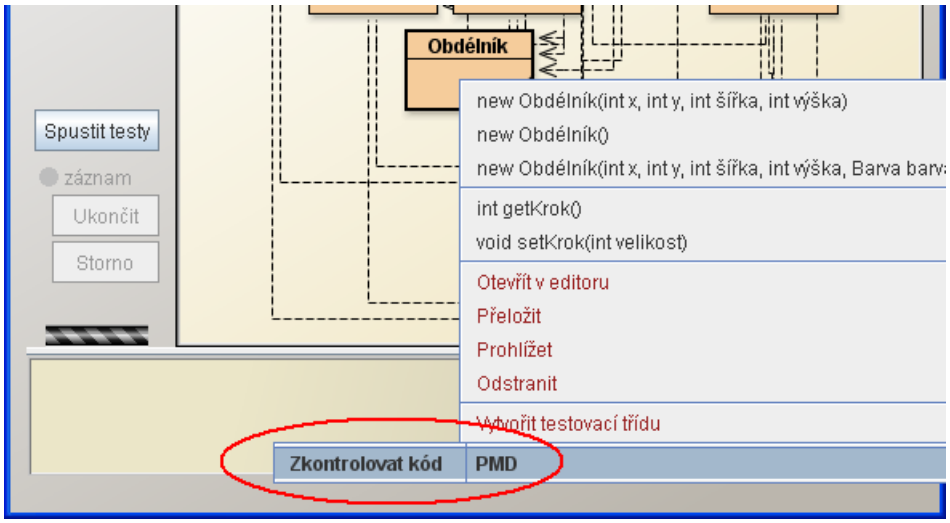
```
//       * Testovací metoda.
//       */
//      public static void test()
//      {
//          KomentovanýStrom instance = new KomentovanýStrom();
//      }
//      /** @param args Parametry příkazového řádku - nepoužívané. */
//      public static void main(String[] args)  {  test();  }
}
```

### Poznámka

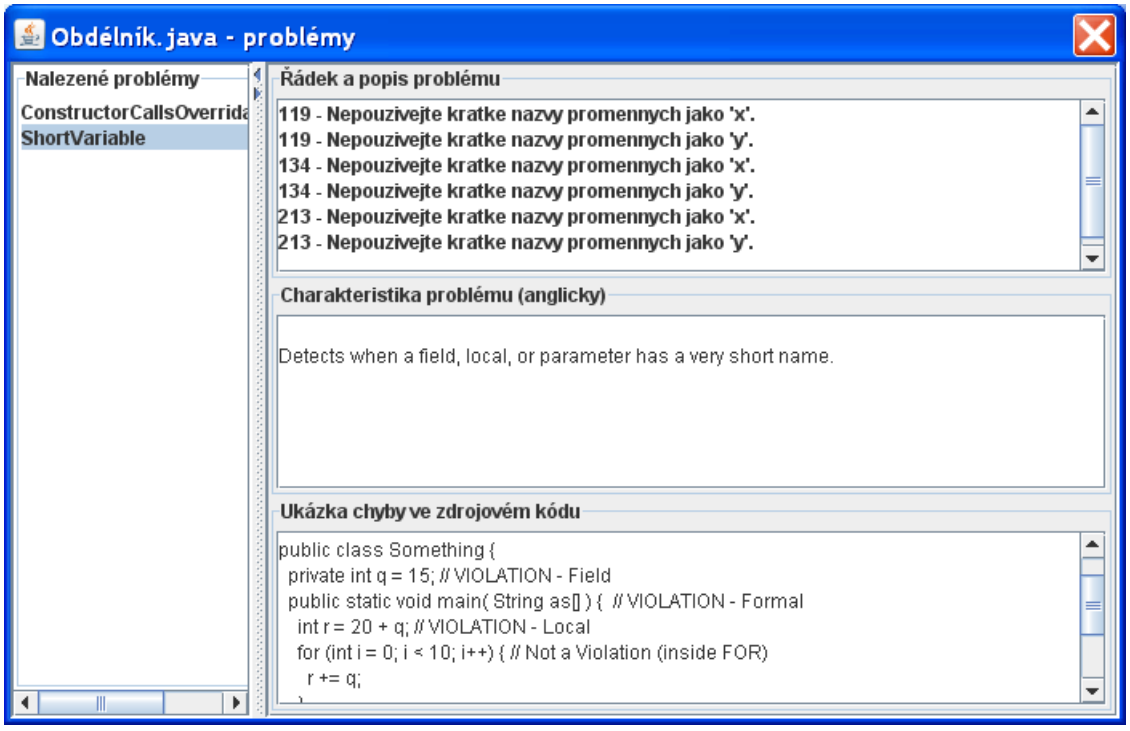
Používání této šablony není (narozdíl od dokumentování kódu) nutností. Je však vhodné si uvědomit, že nám pomáhá udržovat pořádek.

# 2.7. Kvalita kódu

- poměrně nová, tzn. neznámá a málo používaná věc
- nezkoumáme, zda je kód funkční (na to jsou JUnit testy), ale zda je „hezký“
  - pro začátečníky ještě kontroverznější požadavek, než dokumentace
  - poměr užitečnost / 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 (co zkratka znamená, neví ani autoři ;-)
  - `pmd.sourceforge.net`
  - konfigurovatelný nástroj, který z počátku obtěžuje, ale ve svém důsledku výrazně pomáhá
  - dokáže zkontrolovat řádově několik set problémů – počet lze v konfiguraci omezit
- PMD lze použít samostatně (případně přes WWW pomocí JavaJudge), 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 nápovědou
  - pak z hlavního okénka BlueJ z místní nabídky třídy zvolit PMD / Zkontrolovat kód



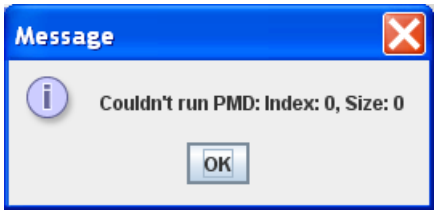

- dostaneme např.:




- PMD je velmi striktní – zásada: nemusíme odstraňovat všechny reportované problémy, ale ty, které ponecháme musíme mít dobře zdůvodněné
  - pokud víme, že je hlášení problému zbytečné / nepodstatné (např. ShortVariable), lze jej editací souboru `pmdrules.xml` potlačit

## Varování

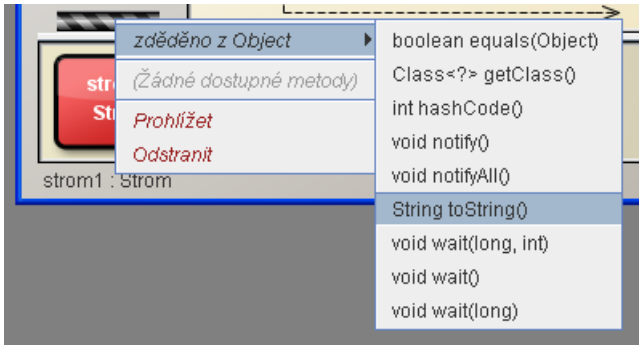
Aktuální konfigurace PMD má problém s příliš dlouhými (více než 20 znaků) identifikátory. Pak hlásí kryptografickou zprávu (opět lze řešit editací `pmdrules.xml`):




Pokud není soubor `pmdrules.xml` korektní, PMD není z BlueJ vůbec dostupné.

# 2.8. Společný předek `Object`

- všechny objekty v Javě mají společné vlastnosti, které jsou zahrnuté ve třídě `Object`
  - jsou to dceřiné třídy (třídy potomků)
  - tyto třídy z `Object` dědí




- `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 zpráva(Object text)`), může být tato metoda volána se skutečným parametrem typu libovolného objektu (`zpráva("řetězec")`)
- zděděné metody z `Object`
  - pokud nám nevyhovují, můžeme je 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 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`
  - informuje 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 [skr-138].

- texty v uvozovkách `"Ahoj"` jsou jedním z příkladů řetězců – třída `String`
  - řetězce se dají spojovat do jednoho pomocí operátoru `+`

```
"Ahoj " + "lidi"
```

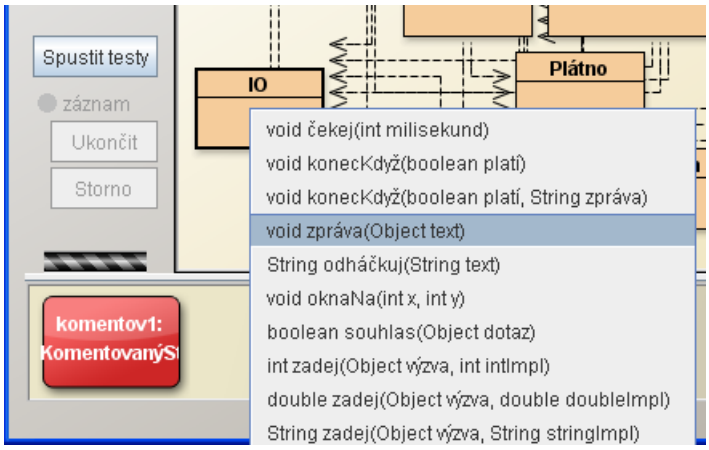
- Java má vlastnost, že automaticky převádí hodnoty primitivních datových typů na řetězce
  - `"Počet = " + počet`

    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 kontraktu, 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 `getNázev()`

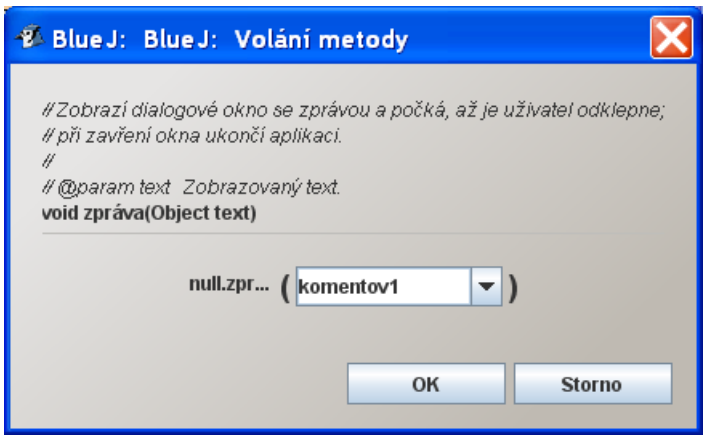
```
/**************************************************************************
 * Převede instanci na řetězec obsahující název třídy, pořadí instance,
 * její souřadnice, výšku a barvu koruny.
 *
 * @return Řetězcová reprezentace dané instance.
 */
@Override
public String toString() {
  return getNázev()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

  řetězec má např. obsah: `Strom_1: x=0, y=0, výška=150, barva=zelená`
- s existencí metody `toString()` všichni počítají (je již v `Object`, takže určitě existuje)

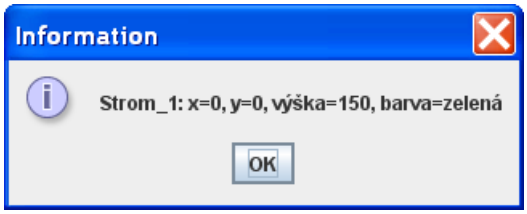
- v BlueJ ji můžeme zavolat např. pomocí metody `zpráva()` ze třídy `IO`




- 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 – nevymýšlí 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 dále 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
- 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, a ukládají se do ní všechny konstanty platné pro celou aplikaci, např. jména fontů, adresáře, defaultní hodnoty apod.
- 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 poloměr poloměr kruhu
   * @returns obvod kruhu
   */
  public static double vypočtiObvodKruhu(double poloměr) {
    return 2 * PI * poloměr;
  }

  /**
   * Vypočítá obsah kruhu o zadaném poloměru
   *
   * @param poloměr 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** (*simply factory method*)
- statická metoda, která vrací odkaz na instanci své třídy
- třída může znepřístupnit konstruktor pomocí `private`
- používá se ke stejnému účelu, jako konstruktor `new()`
- výhody:
  - může se jmenovat libovolně významově – `Osoba.getVysokáŠtíhlá()` (narozdíl od konstruktoru, který má přesně vyhrazené jméno)
  - sama se rozhodne, zda vnitřně opravdu vytvoří novou instanci nebo použije už nějakou hotovou ze svého vnitřního kontejneru
  - 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 vytvářet
- nevýhody
  - tyto třídy se nedají dědit
    - to je ale spíše výhoda, protože dědění se obecně vyhýbáme a nahrazujeme je kompozicí (skládáním)
  - nejasné pojmenování – 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 `Plátno`
- znepřístupňuje konstruktor pomocí `private`
- pro získání odkazu používá statickou tovární metodu, která pokaždé vrací odkaz na stejnou instanci
  - odkazů může být samozřejmě víc, ale vždy na jedinou instanci

- příklad

```
/**
 * Ukázka konstrukce jedináčka Elvise
 * @author P.Herout
 */

public class Elvis {

  /** Jediná instance Elvise - 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 Elvise
  }

  /**
   * Statická tovární metoda
   * @return jedinou instanci jedinečného Elvise
   */
  public static Elvis getElvis() {
    return INSTANCE;
  }

////////////////////////////////////////////////////////////////
  /**
   * Příklad služeb, které Elvis poskytuje
   * @param jménoPísně jméno písně, kterou zazpívá
   */
  public void zpívej(String jménoPísně) {
     System.out.println("Originální Elvis zpívá: " + jménoPísně);
  }
}
```

- použití

```
public class PouzitiElvise {
  public static void main(String[] args) {
    Elvis idol = Elvis.getElvis();
    idol.zpívej("Heartbreak Hotel");
  }
}
```

# 3.1.4. Přepravka (*Messenger*)

- používáme, pokud chceme, aby metoda 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 píší
  - 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`.

  - 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.
 *
 * @author   Rudolf PECINOVSKÝ
 * @version  3.00.002
 */
public class Pozice
{
//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================================

    /** 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 INSTANCÍ ==========================================

    /**************************************************************************
     * 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;
    }
}
```

- 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

```
/**************************************************************************
 * Prohodí pozice zadaných stromů a nechá zkontrolovat,
 * zda si stromy své pozice doopravdy vyměnily.
 * První strom je nutné nechat na konci překreslit,
 * protože druhý strom jej při přesunu vymaže.
 *
 * @param s1 První strom
 * @param s2 Druhý strom
 */
private void pomProhoďPozice(Strom s1, Strom s2) {
  Pozice p1 = s1.getPozice();
  s1.setPozice(s2.getPozice());
  s2.setPozice(p1);
  s1.nakresli();
}

public void testProhoďPozice() {
  pomProhoďPozice(zelenýStrom, žlutýStrom);
}
```

# 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 SvětovéStrany {
  SEVER, VÝCHOD, JIH, ZÁPAD;
}
```

  - tato jednoduchá deklarace již dává k dispozici metody:
    - `String name()` – řetězec s názvem dané instance

- ♦ `int ordinal()` – pořadové číslo instance; první instance má pořadové číslo 0
- ♦ `String toString()` – název instance (stejně jako `name()`)

- použití pak je (viz též [skr-66]):

```
SvětovéStrany směr = SvětovéStrany.JIH;
System.out.println(směr.name());
System.out.println(směr.ordinal());
System.out.println(směr);
System.out.println(SvětovéStrany.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 `Směr8`, kdy s jednotlivými instancemi jsou svázány i doplňující metody

### Varování

Typy ve `Směr8` a `SvětovéStrany` 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. `SvětovéStrany.JIH` (nebo `Barva.KHAKI`)
  - pro získání odkazů se nepoužívá statická tovární metoda
- příklad, kdy je s typem svázána instanční metoda
  - jméno instance (např. `JIH`) slouží jako volání konstruktoru, takže parametr za ním je parametrem konstruktoru

```
/**********************************************************
 * Instance výčtového typu {@code SvětovéStrany} představují
 * světové strany a jejich odpovídající směry na mapě
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
```

```
public enum SvětovéStrany {

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

//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==========================
   private final String směr;

//###########################################################
//== KONSTRUKTORY A TOVÁRNÍ METODY ==========================
  /**
   * privátní konstruktor
   */
  private SvětovéStrany(String směr) {
    this.směr = směr;
  }

//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ =====================
  public String směrPohledu() {
    return "Když se na mapě dívám na " +
           name() +
           ", dívám se směrem " + směr;
  }

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

- 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 posuň(Obdélník obd) { ... }
public void posuň(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
    - 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
    - jak je zajištěno, 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 zůstávat stejné
    - jsou-li navenek 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

- 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 „třída bez implementace“
  - 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 `posuň()`)
  - definujeme metody, které očekávají (formální) parametr typu rozhraní a skutečný parametr bude instance konkrétní třídy splňující (implementující) rozhraní
- název `interface` začíná konvencí písmenem `I` (`IKreslený`)
  - 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 `IKreslený`)
  - ne vždy je to možné nebo vhodné nebo pochopitelné (viz dále `IObdarovávatelný` jako kuriózní případ), proto je konvence s písmenem `I` na začátku velmi dobrá, byť není používána v Java Core API
- rozhraní může deklarovat pouze metody instancí – nelze definovat metody s atributem `static`
- 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)
  - lze použít modifikátor `abstract`
  - nepoužívat, protože implementující třídy jej použít nesmí – po zkopírování je nutné jej vymazat
- před metodu v rozhraní dodat zakomentovanou anotaci `@Override`
  - po zkopírování do implementující třídy se dá lehce odkomentovat a překladač pak ohlídá, zda je metoda skutečně implementována

■ ukázka jednoduchého rozhraní

```
public interface IUkázkaRozhraní {

 /**
  * Vypíše na konzoli svůj stav
  */
//  @Override
  public void vypiš();
}
```

■ 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 `vypiš()`

### Poznámka

Jméno `IUkázkaRozhraní` je zcela nevhodné, protože neříká nic o účelu. Vhodnější by bylo `IVypisovatelný`.

# 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

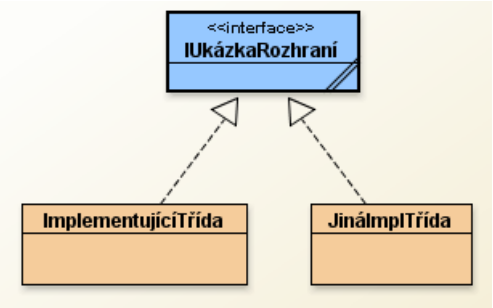
■ třída implementuje rozhraní, když implementuje všechny jeho metody a do své hlavičky uvede `implements`, např.

```
public class ImplementujícíTřída implements IUkázkaRozhraní {
```

■ v BlueJ se dá hlavička implementace zajistit (vygenerovat) pomocí trojúhelníkové šipky tažené ve směru od třídy k rozhraní

- import třídy Úpravy / Nová třída ze souboru

- ukázka dvou jednoduchých tříd, které implementují rozhraní
  - dodržení signatury zkontroluje překladač
  - dodržení kontraktu musí zajistit programátor a zkontrolovat tester – vypisuje něco smysluplného a vypisuje to na konzoli




- `ImplementujícíTřída` využívá standardních postupů a kontrakt plní uspokojivě
  - metoda `getClass()` je ze třídy `Object` a metoda `getSimpleName()` vrací jméno třídy (viz dříve)

```
public class ImplementujícíTřída implements IUkázkaRozhraní {

  private static int počet = 0;
  private final int POŘADÍ = ++počet;

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

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

- `JináImplTřída` šla cestou nejmenšího odporu, signaturu plní, ale o smysluplném kontraktu lze pochybovat

```
public class JináImplTřída implements IUkázkaRozhraní {

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

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

```
Jsem 1. instance třídy ImplementujícíTřída
Jsem 2. instance třídy ImplementujícíTřída
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

```
ImplementujícíTřída it1 = new ImplementujícíTřída();
IUkázkaRozhraní ur1 = new ImplementujícíTřída();
IUkázkaRozhraní ur2 = new JináImplTřída();
it1.vypiš();
ur1.vypiš();
ur2.vypiš();
```

  - 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 překladač
  - třída svými službami zapadá do nějakého většího celku

# 3.2.5. Značkovací rozhraní

- 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 jejích instancí klonováním

# 3.3. Návrhové vzory – pokračování

## 3.3.1. Motivace

- při přesouvání 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 `Plátno` 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 `SprávcePlátna` 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 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.vraťKresli()` vykreslování obnovit (pokyn k překreslení)
    - `SP` je referenční proměnná na instanci jedináčka `SprávcePlátna`
- 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*), **Předplatitel** (*Subscriber*)
- řeší typickou úlohu, kdy objekt čeká na nějakou událost (např. stisk klávesy)
- dvojí způsob řešení:

1. objekt neustále testuje, zda událost nastala – nevhodné
2. objekt se zaregistruje 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 rozhraní
- realizace tohoto návrhového vzoru viz později
  - v Java Core API má NV přímo předlohu – třída `Observable` a rozhraní `Observer`
- použití tohoto návrhového vzoru pro případ kreslení – třída `SprávcePlátna` je vysílač:

1. definovat rozhraní se zprávou, kterou bude objekt vysílač posílat všem přihlášeným posluchačům – `IKreslený nakresli(Kreslítko)`

```
public interface IKreslený
{
  /**************************************************************
   * Za pomoci dodaného kreslítka vykreslí obraz své instance
   * na animační plátno.
   *
   * @param kreslítko Kreslítko, kterým se instance nakreslí na plátno
   */
  void nakresli( Kreslítko kreslítko );
}
```

   - `Kreslítko` není pro princip Vysílač-Posluchač důležité – je důležité proto, že odstiňuje `SprávcePlátna` od vlastního kreslení
   - aby objekty věděly kam se mají vykreslit, využijí objektu třídy `Kreslítko`, který je jim předán jako parametr zasílané zprávy
   - jeho instanci poskytuje jen `SprávcePlátna`
     - je to zabezpečení, že se bude kreslit jen tehdy, když to uzná `SprávcePlátna` za vhodné
     - k plátnu nesmí přistupovat nikdo, o kom `SprávcePlátna` jako Prostředník neví
   - implementace `Kreslítko` není pro nás důležitá
2. implementovat toto rozhraní všemi posluchači, zde ve třídě `Strom`
   - využívá se toho, že třídy `Obdélník` a `Elipsa` již implementují rozhraní `IKreslený`

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

3. protože posluchači implementují stejné rozhraní, mohou se vykreslované různé objekty jako instance tohoto rozhraní u `SprávcePlátna` zaregistrovat
   - to se provede metodou `přidej(vykreslovanýObjekt)`, která je ze třídy `SprávcePlátna`
   - pomocí této metody lze zaregistrovat všechny Posluchače u Vysílače (`SprávcePlátna`) – ukázka ze třídy `Strom`

```
/*************************************************************
 * Přihlásí instanci u aktivního plátna do jeho správy.
 */
public void zobraz()
{
  SprávcePlátna.getInstance().přidej( this );
}
```

4. kdykoliv dojde k očekávané události, vysílač `SprávcePlátna` všem zaregistrovaným pouze postupně posílá zprávu `nakresli(aktuálníKreslítko)`
   - jak si vytvoří `aktuálníKreslítko` 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 vytvoří jeden 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 `SprávcePlátna`, ke kterému se každý nově vzniklý geometrický tvar přihlásí metodou `přidej( 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.překresli()`
  - 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 objektům
    - součástí zprávy je servisní parametr – `kreslítko`

# 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 `IPosuvný` a využívá též přepravku `Pozice` – viz dříve

```
/**********************************************************************
 * Rozhraní {@code IPosuvný} definuje povinou sadu metod, jež musí být
 * poskytovány objekty, které mají byt schopny posunu po animovaném plátně.
 *
 * @author     Rudolf Pecinovsky
 * @version    1.01, 28.1.2003
 */
public interface IPosuvný
{
//== DEKLAROVANÉ METODY ================================================

    /**********************************************************************
     * Vrátí aktuální pozici pposuvného objektu (většinou pozici levého
     * horního rohu opsaného obdélníku) jako instanci třídy {@code Pozice}.
     *
     * @return Akutální pozice objektu.
     */
     public Pozice getPozice();


    /**********************************************************************
     * Přesune posuvný objekt do nové pozice.
     *
     * @param pozice  Nová pozice objektu.
     */
    public void setPozice( Pozice pozice );


    /**********************************************************************
     * Nastaví novou pozici objektu.
     *
     * @param x   Nová x-ová pozice objektu
     * @param y   Nová y-ová pozice objektu
     */
    public void setPozice(int x, int y);
}
```

- třída `Přesouvač` je Služebník, který zajistí (na jednom místě kódu) implementaci plynulého přesunu, přičemž klíčové metody jsou `přesuňO()` a `přesuňNa()`

```
/**********************************************************************
 * Třída {@code Přesouvač} slouží k pohybu s instancemi tříd
 * implementujicich rozhraní {@link IPosuvný}.
 *
 * @author     Rudolf Pecinovský
 * @version    2.01, duben 2004
```

```
 */
public class Přesouvač
{
//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================================

    /** Tento atribut je tu pouze pro zjednodušení psaní. */
    private static final SprávcePlátna SP = SprávcePlátna.getInstance();

    /** Oočet milisekund mezi dvěma překresleními objektu. */
    private static final int PERIODA = 50;


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

    /** Počet vytvořených instancí */
    private static int počet = 0;


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

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


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

    /** Specifikuje rychlost posunu objektu daným posunovačem. */
    private int rychlost;


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

    /**************************************************************************
     * Vytvoří přesouvače, který bude přesouvat objekty rychlosti 1.
     */
    public Přesouvač()
    {
        this( 1 );
    }


    /**************************************************************************
     * Vytvoří přesouvače, který bude přesouvat objekty zadanou rychlostí.
     *
     * @param rychlost Rychlost, kterou bude přesouvač pohybovat
     *                 se svěřenými objekty.
     */
    public Přesouvač( int rychlost )
    {
        if( rychlost <= 0 ) {
            throw new IllegalArgumentException(
                "Zadaná rychlost musí byt nezáporná!" );
```

```
        }
        this.rychlost = rychlost;
        this.název    = getClass().getName() + "(ID=" + ++počet +
                        ",rychlost=" + rychlost + ")";
    }

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

    /**************************************************************************
     *
     * @return     Řetězec informující o vnitřním stavu instance
     */
    @Override
    public String toString()
    {
        return název;
    }

//== NOVĚ ZAVEDENÉ METODY INSTANCÍ ============================================

    /**************************************************************************
     * Přesune zadaný objekt o požadovaný počet bodů.
     *
     * @param objekt  Přesouvaný objekt
     * @param doprava Počet bodů, o než se objekt přesune doprava
     * @param dolů    Počet bodů, o než se objekt přesune dolů
     */
    public void přesunO( IPosuvný objekt, int doprava, int dolů )
    {
        double vzdálenost = Math.sqrt(doprava*doprava + dolů*dolů);
        int    kroků      = (int)(vzdálenost / rychlost);
        double dx = (doprava+.4) / kroků;
        double dy = (dolů   +.4) / kroků;
        Pozice p  = objekt.getPozice();
        double x  = p.getX() + .4;
        double y  = p.getY() + .4;

        for(int i=kroků;   i > 0;   i-- )
        {
            x = x + dx;
            y = y + dy;
            SP.nekresli(); {
                objekt.setPozice( (int)x, (int)y );
                SP.překresli();
            } SP.vraťKresli();
            IO.čekej(PERIODA);
        }
    }


    /**************************************************************************
     * Přesune zadaný objekt o požadovaný počet bodů.
     *
```

```
     * @param objekt  Přesouvaný objekt
     * @param posun   Počet bodů, o než se objekt přesune doprava a dolů
     *                uložený v přepravce
     */
    public void přesunO( IPosuvný objekt, Pozice posun )
    {
        přesunO( objekt, posun.x, posun.y );
    }


   /*************************************************************************
     * Přesune zadaný objekt do požadované pozice.
     *
     * @param objekt  Přesouvaný objekt
     * @param x       x-ova souřadnice požadované cílové pozice
     * @param y       y-ova souřadnice požadované cílové pozice
     */
    public void přesunNa( IPosuvný objekt, int x, int y )
    {
        Pozice p = objekt.getPozice();
        přesunO( objekt, x-p.x, y-p.y );
    }


   /*************************************************************************
     * Přesune zadaný objekt do požadované pozice.
     *
     * @param objekt  Přesouvaný objekt
     * @param pozice  Požadované cílové pozice.
     */
    public void přesunNa( IPosuvný objekt, Pozice pozice )
    {
        přesunNa( objekt, pozice.x, pozice.y );
    }
}
```

- implementace `IPosuvný` u obsluhovaného `Strom` – viz též již dříve u Přepravky

  blok `SP.nekresli(); { ... } SP.vraťKresli();` zajistí vykreslení celého stromu najednou

```
  /*************************************************************************
   * 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) {
    SP.nekresli(); {
      koruna.setPozice(x, y);
      kmen.setPozice(x + (koruna.getŠířka() - kmen.getŠířka()) / 2,
                     y + koruna.getVýška());
    } SP.vraťKresli();
  }
```

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

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

- implementace `IPosuvný` u obsluhovaného `Obdélník`

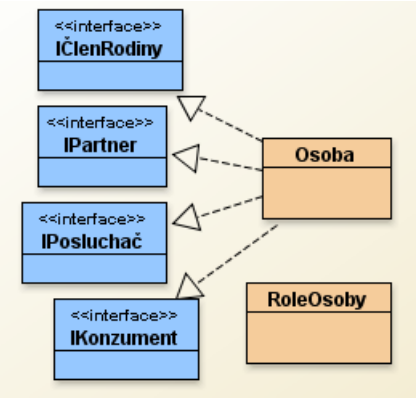
```
  public Pozice getPozice()
  {
      return new Pozice( xPos, yPos );
  }

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

  public void setPozice(Pozice pozice)
  {
      setPozice( pozice.x, pozice.y );
  }
```

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

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

  Přesouvač přesRychl_1 = new Přesouvač();
  přesRychl_1.přesunNa(stromC, 150, 200);

  Přesouvač přesRychl_5 = new Přesouvač(5);
  přesRychl_5.přesunO(stromB, 200, 0);
  přesRychl_5.přesunO(stromA, -100, -100);
}
```

# 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 tolik 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. `ukliďPoSobě()`, `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 zkontroluje překladač
- příklad




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

```
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 {
  public void ukliďPoSobě() {
    System.out.println("uklízím");
  }

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

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

  public void dejMiPusu() {
    System.out.println("líbám");
  }

  // 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.ukliďPoSobě();
    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 – [skr-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 jde 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 vynechávám

  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 rovnost 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 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 [skr-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: `Obdélník 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

```
Obdélník o1, o2;
o1 = new Obdélník();
o2 = o1;
o1.setŠířka(200);
int šířka = o2.getŠířka();
```

- 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
- červený obdélník 10 x 20, který je momentálně na pozici 0,0 není ekvivalentní jinému červenému obdélníku momentálně stejných rozměrů a na stejné pozici
  - oba dva tyto objekty se na plátně překrývají, ale vzápětí 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 kontraktu `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 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 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 číslo = 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 [skr-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 maji `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 zadané 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 svojí 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 [skr-142]
- každá třída může mít metodu `main()`
  - to lze využít pro testovací účely – v BlueJ v šabloně standardní třídy je zakomentována
  - prakticky se ale `main()` do tříd téměř nikdy nedává – třída se testuje pomocí JUnit testů
- z hlediska přehlednosti celého projektu se vytváří speciální třída
  - jmenuje se typicky `Hlavni` nebo `Aplikace`
    - 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 `návod()`, 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   1.00.000
 */
public class Krajina {

//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================
```

```
  private static final SprávcePlátna SP = SprávcePlátna.getInstance();

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

//== PROMĚNNÉ ATRIBUTY INSTANCÍ ================================
  // nemohou být konstantní, protože jsou vytvářeny v metodě vytvořDomek()
  private Obdélník zeď;
  private Trojúhelník střecha;

//== PŘÍSTUPOVÉ METODY VLASTNOSTÍ TŘÍDY ========================
//== OSTATNÍ NESOUKROMÉ METODY TŘÍDY ===========================

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

  /**********************************************************
   * Vytvoří obrázek stromu a domku a zobrazí je na plátno
   */
  public Krajina() {
    strom = new Strom(50, 50, 100, 100);
    strom.zobraz();
    vytvořDomek();
  }

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

  /**********************************************************
   * 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 rychlost přesouvání
   */
  public void jedenDen(int rychlost) {
    IO.zpráva("Proběhne jeden den rychlostí " + rychlost);
    Přesouvač přes = new Přesouvač(rychlost);
    Elipsa slunce = new Elipsa(0, 5, 40, 40, Barva.ŽLUTÁ);
    slunce.posunVpravo(-40);
    SP.přidej(slunce);
    přes.přesunNa(slunce, 300, 5);
  }

//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ ========================
//== SOUKROMÉ A POMOCNÉ METODY TŘÍDY ===========================
//== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ========================
  /**********************************************************
   * Vytvoří domek a zobrazí jej na plátno
   */
  private void vytvořDomek() {
    střecha = new Trojúhelník(180, 50, 120, 50, Barva.ČERVENÁ);
```

```
    zeď = new Obdélník(200, 100, 80, 50, Barva.SVĚTLEŠEDÁ);
    SP.přidej(střecha);
    SP.přidej(zeď);
  }
}
```

- a možná třída `Hlavni`

```
/*************************************************************
 * Třída {@code Hlavni} je hlavní třídou projektu,
 * který zobrazí krajinu a nechá projít jeden den
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
public class Hlavni {

  /***********************************************************
   * Zpracuje jeden parametr příkazového řádku - rychlost posuvu
   * není-li zadán žádný parametr, vypíše návod
   *
   * @param args Parametry příkazového řádku
   * @returns zadanou rychlost
   */
  public 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 Hlavní 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(rychlost);
```

```
    }
}
```

## 4.3.1.1. Překlad a spuštění z příkazové řádky

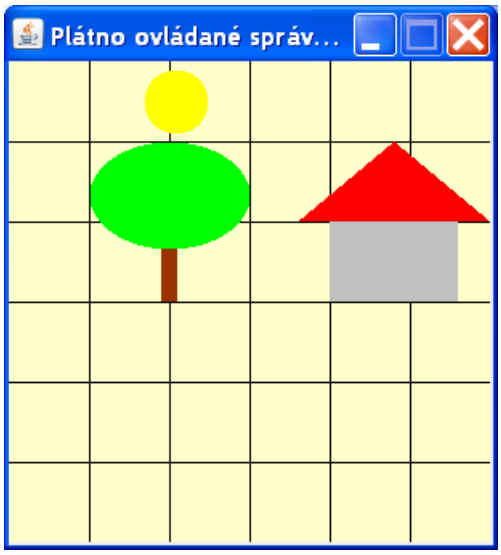
- protože jsou zdrojové kódy v UTF-8 (a `javac` očekává windows-1250), musíme tuto skutečnost `javac` sdělit přepínačem `-encoding UTF-8`
- stačí zadat příkaz k překladu pouze hlavní třídy projektu (tj. souboru `Hlavni.java`), pokud `javac` zjistí, že nemá přeloženou nějakou potřebnou třídu, přeloží si ji
- výpis návodu na konzoli je v nesprávném kódování (`SpuÜtýnÝ:`) – to je možné ošetřit v programu – viz KIV/JXT
- spuštění je možné dvěma způsoby
  - v konzolovém režimu: `java Hlavni 4`

    aplikace se spustí spolu s konzolovým oknem, do nějž 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

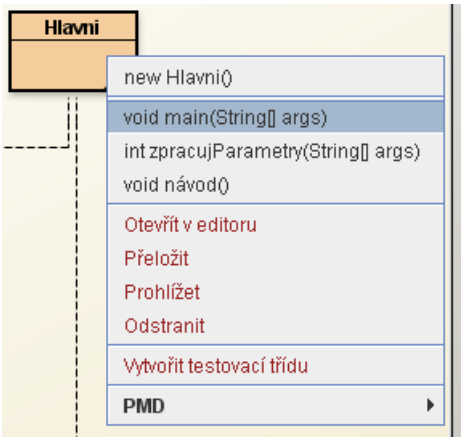
```
>javac -encoding UTF-8 Hlavni.java

>java Hlavni
SpuÜtýnÝ:
  java Hlavni rychlost
    kde rychlost <1, 10>

>java Hlavni 4

>javaw Hlavni 4
```

- výsledek práce programu

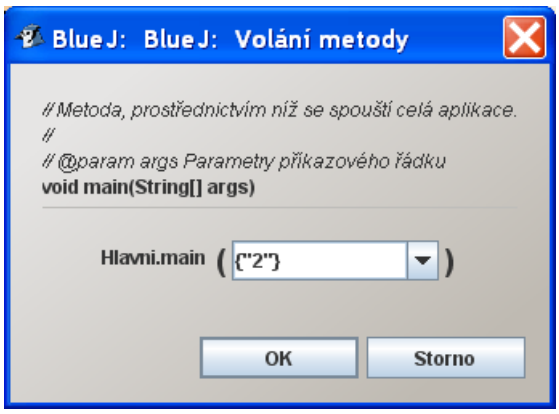


## 4.3.1.2. Spuštění z BlueJ

- 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 28 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 zabaleno 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: 1.6.0_27 (Sun Microsystems Inc.)` – 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 přenositelně
  - 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 je nástroj Ant – viz KIV/JXT nebo 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

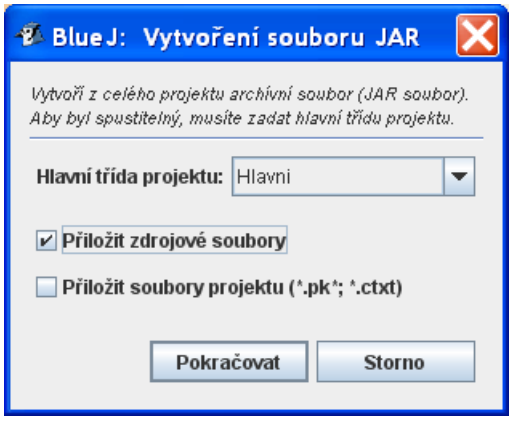
```
>echo Main-Class: Hlavni> man.txt

>jar cmf man.txt jeden-den.jar *.class

>java -jar jeden-den.jar
SpuÜtýnÝ:
  java Hlavni rychlost
    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




- BlueJ sám zajistí vytvoření správného souboru `MANIFEST.MF`
- 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 zkopíruje 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`, `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 tuto povinnost splní 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.jmeno_projektu`
  - 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.isPodzim());
    ```

    tu si v JUnit testech nikdo s ničím jiným nesplete
  - bez statického importu by bylo nutné

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

- 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

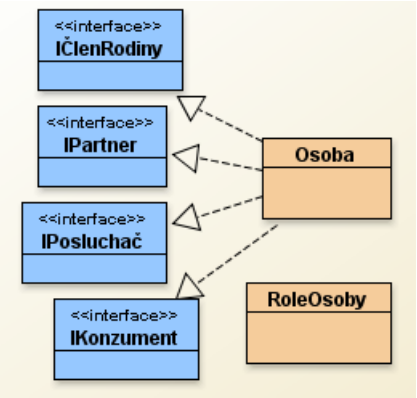
# 5.1. Typy dědění

- **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 (přepsat)
    - nebezpečí: při přizpůsobování zděděných entit potřebám potomka není občas dodržen kontrakt předka
  - např.: všechny třídy přebírají základní metody od třídy `Object`
- 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

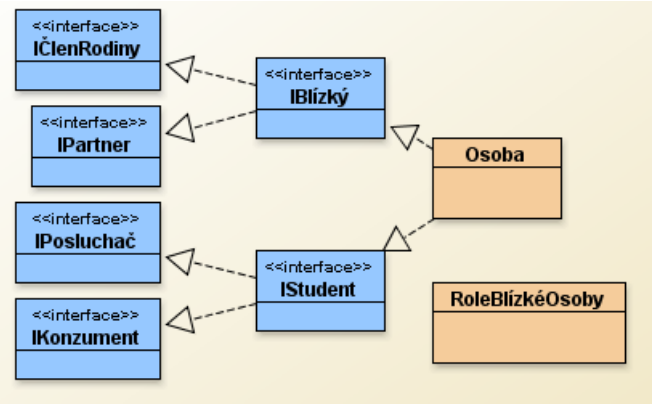
- ♦ někdy tyto případy nejsou na první pohled zřejmé – čtverec je speciální typ obdélníka
  - – ale čtverec nemůže obdélník kdykoliv zastoupit, protože obdélníku lze nastavovat jakoukoliv stranu nezávisle na druhé
  - – v tomto případě by se mělo použít skládání (kompozice) – viz dále
- ♦ nebo naopak – 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 zjednodušuje vazby v programu – pokud by místo třídy `Osoba` existovaly dvě třídy `Muž` a `Žena`, pak se zjednoduší vztahy implementace
- příklad – `IBlízký` splňuje podmínku 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 {
  public void ukliďPoSobě() {
    System.out.println("uklízím");
  }

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

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

```
  public void dejMiPusu() {
    System.out.println("líbám");
  }

  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.ukliďPoSobě();
    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 znovu napsat implementaci všech potřebných metod
  - v naprosté většině případů je implementace velmi jednoduchá a představuje jen jedno 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 – `Čtverec` bude sestaven z pouze jednoho `Obdélníka`, 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 Čtverec} představují čtverce
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
public class Čtverec implements IKreslený, IPosuvný {

//== KONSTANTNÍ ATRIBUTY TŘÍDY ====================================
  /** Implicitní rozměr */
  private static final int IMPLICITNÍ_ROZMĚR = 100;

  /** Počáteční barva nakreslené instance v případě,
   *  kdy uživatel žádnou požadovanou barvu nezadá
   */
  public static final Barva IMPLICITNÍ_BARVA = Barva.ŽLUTÁ;

//== KONSTANTNÍ ATRIBUTY INSTANCÍ =============================
  /** skutečná reprezentace pomocí již existujícího tvaru */
  private final Obdélník obdélník;

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

  /**********************************************************
   * Vytvoří novou instanci se polohou [0, 0] a rozměrem 100
   * a implicitní barvou.
   */
  public Čtverec() {
    this(0, 0, IMPLICITNÍ_ROZMĚR);
  }

  /**********************************************************
   * 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 Čtverec(int x, int y, int strana) {
    this(x, y, strana, IMPLICITNÍ_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 Čtverec(int x, int y, int strana, Barva barva) {
    obdélník = new Obdélník(x, y, strana, strana, barva);
  }

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

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

  /**************************************************************************
   * Vrátí instanci třídy Pozice s pozicí instance.
   *
   * @return   Pozice s pozicí instance.
   */
  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
   */
  public void setPozice(int x, int y) {
    obdélník.setPozice(new Pozice(x, y));
  }

  /**************************************************************************
   * Nastaví novou pozici instance.
   *
   * @param pozice   Nová pozice instance
```

```
   */
  public void setPozice(Pozice pozice) {
    obdélník.setPozice( pozice.x, pozice.y );
  }


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

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

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

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

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

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

   */
  public void nakresli(Kreslítko kreslítko) {
    obdélník.nakresli(kreslítko);
```

```
  }

//== NOVĚ ZAVEDENÉ METODY INSTANCÍ ============================

  /**************************************************************************
   * Přesune instanci o zadaný počet bodů vpravo,
   * při záporné hodnotě parametru vlevo.
   *
   * @param vzdálenost Vzdálenost, o kterou se instance přesune.
   */
  public void posunVpravo(int vzdálenost) {
    obdélník.posunVpravo(vzdálenost);
  }

//== 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=" + obdélník.getX() + ", y=" + obdélník.getY() +
             ", strana=" + obdélník.getŠířka() +
             ", barva=" + obdélník.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
  - nadtřída – podtřída

# 5.4.1. Principy dědičnosti implementace

- potomek převezme instanci předka jako svůj podobjekt a současně převezme 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 kontraktu
- tři důvody, proč se používá dědičnost (důvody se často překrývají):
  1. specializace – `Auto` x `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 `závěrečnáPráce()`
       - u prvního je to bakalářka, u druhého diplomka
  3. znovupoužití kódu – nejméně dobrý důvod – `Čtverec` x `Obdélník`
     - autoři potomků dědících implementaci se často soustředí pouze na to, aby získali implementaci „zdarma“,a zcela ignorují nutnost dodržení kontraktu a invariantů předka
       - správný (ale pracný) postup – viz příklad výše se skládáním
       - důsledek: instance potomka se nemůže plnohodnotně vydávat za instanci předka – porušuje 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, jak bránit používání:
        - označení `deprecated` (odmítaný, zavrhovaný)
        - informace v 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, nepoužijeme dědičnost
- 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 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

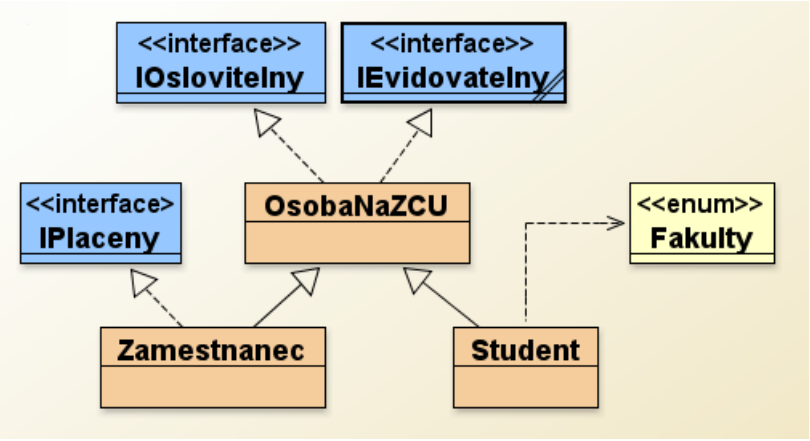
- odvození od rodičovské třídy deklaruje dceřiná třída v hlavičce za svým názvem klíčovým slovem `extends` následovaným názvem rodičovské třídy
- dědictví od třídy `Object` se uvádět nemusí a naopak, nemá-li třída v hlavičce uvedeného předka, je přímým potomkem třídy `Object`
- případná deklarace implementace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

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

- modifikátor `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 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 os1 = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybná informace: ",
            "OsobaNaZCU: Moudrý, ID=10321", os1.toString());
  }

  @Test
  public void testIOslovitelny() {
    IOslovitelny os1 = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybné jméno: ", "Moudrý", os1.getJmeno());
  }

  @Test
  public void testIEvidovatelny() {
    IEvidovatelny os1 = new OsobaNaZCU("Moudrý", "10321");
    assertEquals("Chybné ID: ", "10321", os1.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 uvedeno, dědí implementaci rozhraní `IOslovitelny` a `IEvidovatelny`
- přidává atribut `plat`
- protože ve třídě `OsobaNaZCU` není bezparametrický konstruktor, musí být v konstruktoru `Zamestnanec` být 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 předku
- 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 `Zamestnanec`

```
  public void testToString() {
    OsobaNaZCU os1 = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybná informace: ",
```

```
      "Zamestnanec: Moudrý, ID=10321, plat=20000", os1.toString());
  }

  public void testChybnyPlat() {
    IPlaceny os1 = new Zamestnanec("Moudrý", "10321", -100);
    assertEquals("Chybný plat: ", -100, os1.getPlat());
  }

  public void testZmenaPlatu() {
    Zamestnanec os1 = new Zamestnanec("Moudrý", "10321", 20000);
    os1.setPlat(30000);
    assertEquals("Chybný plat: ", 30000, os1.getPlat());
  }

  public void testIPlaceny() {
    IPlaceny os1 = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybný plat: ", 20000, os1.getPlat());
  }

  public void testIOslovitelny() {
    IOslovitelny os1 = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybné jméno: ", "Moudrý", os1.getJmeno());
  }

  public void testIEvidovatelny() {
    IEvidovatelny os1 = new Zamestnanec("Moudrý", "10321", 20000);
    assertEquals("Chybné ID: ", "10321", os1.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, UJP, UUD;
}
```

- 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` být 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`

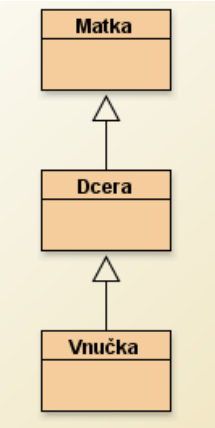
```
  public void testToString() {
    OsobaNaZCU os2 = new Student("Pilná", "A10B9999P", Fakulty.FAV);
    assertEquals("Chybná informace: ",
          "Student: Pilná, ID=A10B9999P, fakulta=FAV", os2.toString());
  }

  public void testIOslovitelny() {
    IOslovitelny os2 = new Student("Pilná", "A10B9999P", Fakulty.FAV);
    assertEquals("Chybné jméno: ", "Pilná", os2.getJmeno());
  }

  public void testIEvidovatelny() {
    IEvidovatelny os2 = new Student("Pilná", "A10B9999P", Fakulty.FAV);
    assertEquals("Chybné ID: ", "A10B9999P", os2.getID());
  }

  public void testFakulty() {
    Student os2 = new Student("Pilná", "A10B9999P", Fakulty.FAV);
    assertEquals("Chybná fakulta: ", "FEL", os2.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 nadtřídou

- 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)" );
    }
```

```
    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 zavoláním 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 nemá 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 skrytou součástí třídy

■ jakmile potomek překryje rodičovskou verzi metody, stane se překrytá verze pro okolí nedostupná

- jedinou výjimkou je samotný potomek, 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
  - ♦ 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 výrazně ulehčit práci
- ve třídě `ZděděnýČtverec` je vhodné / nutné překrýt pouze jednu metodu z původního `Obdélník` a to `setRozměr(int šířka, int výška)`
- pak je nutné napsat konstruktory – to je ale třeba vždy
  - třída potomka by měla definovat konstruktor maximálně využívající co nejobecnější verzi rodičovského konstruktoru
  - pokud to neudělá, zkomplikuje přípravu případných vnoučat

- konstrukce nové třídy je výrazně kratší než v případě skládání

```
/*****************************************************************
 * Instance třídy {@code ZděděnýČtverec} představují čtverce
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
public class ZděděnýČtverec extends Obdélník {

//== KONSTANTNÍ ATRIBUTY TŘÍDY ==================================
  /** Implicitní rozměr */
  private static final int IMPLICITNÍ_ROZMĚR = 100;

  /** Počáteční barva nakreslené instance v případě,
   *  kdy uživatel žádnou požadovanou barvu nezadá
   */
  public static final Barva IMPLICITNÍ_BARVA = Barva.ŽLUTÁ;

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

  /*****************************************************************
   * Vytvoří novou instanci se polohou [0, 0] a rozměrem 100
   * a implicitní barvou.
   */
  public ZděděnýČtverec() {
    this(0, 0, IMPLICITNÍ_ROZMĚR);
  }

  /*****************************************************************
   * 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 ZděděnýČtverec(int x, int y, int strana) {
    this( x, y, strana, IMPLICITNÍ_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 ZděděnýČtverec(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 šířka    Nově nastavovaná šířka; šířka>=0
   * @param výška    Nově nastavovaná výška; výška>=0
   */
  @Override
  public void setRozměr(int šířka, int výška) {
    int strana = Math.min(šířka, výška);
    super.setRozměr(strana, strana);
  }
}
```

- 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 `setRozměr(int šířka, int výška)` lze udělat záludnou chybu
- ve třídě `Obdélník` již existuje metoda `setRozměr(int rozměr)`, kterou by bylo zdánlivě možné ve třídě `ZděděnýČtverec` využít

```
  @Override
  public void setRozměr(int šířka, int výška) {
    super.setRozměr(Math.min(šířka, výška));
  }
```

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

```
    public void setRozměr(int rozměr)
    {
        setRozměr( rozměr, rozměr );
    }

    public void setRozměr(int šířka, int výška)
    {
        if( (šířka < 0) || (výška < 0) ) {
            throw new IllegalArgumentException(
                "Rozměry musí byt nezáporné: šířka="+šířka + ",
                                             výška="+výška);
        }
        this.šířka = šířka;
        this.výška = výška;
```

```
        SP.překresli();
    }
```

- takže metoda `setRozměr(int rozměr)` volá metodu `setRozměr(int šířka, int výška)`
- při volání se ale použije překrytá verze `setRozměr(int šířka, int výška)` ze `ZděděnýČtverec`
  - protože se virtuální stroj vždy snaží volat metodu té instance, kterou je
- ta opět volá jednoparametrickou metodu `setRozměr(int rozměr)` z `Obdélník` 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 pomocí `final` jako konečné

    ```
    public final class ZděděnýČtverec extends Obdélník {
    ```

    - 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

- dědičnost porušuje zapouzdření a skrývání implementace
  - nutí potomka, aby znal implementaci v předkovi – viz nekonečná smyčka při návrhu metody `setRozměr(int,int)`
- dědičnost zvyšuje vzájemnou provázanost tříd
  - potřebuji-li modifikovat rodiče (najdu v něm chybu nebo v reakci na změnu zadání), musím zkontrolovat 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 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* nebo se použije stereotyp *<<abstract>>*

- na rozdíl od rozhraní může mít instanční atributy a může mít implementaci některých metod
- zbývající metody mohou mít jen hlavičku, jako je to u rozhraní
  - mezi modifikátory však musí být uvedeno klíčové slovo `abstract`
  - abstraktní metody nesmějí být soukromé, potomek by na ně neviděl a nemohl by je implementovat
  - konkrétní metody 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
  - `ADopravníZnačka` v konstruktoru vytváří sloupek a pomocí volání abstraktní metody i ceduli
    - cedule je typu `IKreslený`, protože toto rozhraní implementují všechny základní tvary

```
public abstract class ADopravníZnačka implements IKreslený {

//== KONSTANTNÍ ATRIBUTY TŘÍDY ================================
  public static final Barva BARVA_SLOUPKU = Barva.ŠEDÁ;
  private static final SprávcePlátna SP = SprávcePlátna.getInstance();

//== KONSTANTNÍ ATRIBUTY INSTANCÍ ==============================
  private final Obdélník sloupek;

//== PROMĚNNÉ ATRIBUTY INSTANCÍ ▶
==============================================
  private IKreslený cedule = null;

//== KONSTRUKTORY A TOVÁRNÍ METODY =============================
  /**********************************************************
   * vytvoří sloupek pod dopravní značkou a pak příslušnou ceduli
   * obě části zaregistruje u správce plátna
   */
  public ADopravníZnačka(int x, int y) {
    sloupek = new Obdélník(x + 40, y + 100, 20, 100, BARVA_SLOUPKU);
    SP.přidej(sloupek);
    cedule = vytvořCeduli(x, y);
    SP.přidej(cedule);
  }

//== ABSTRAKTNÍ METODY =========================================
  /**********************************************************
```

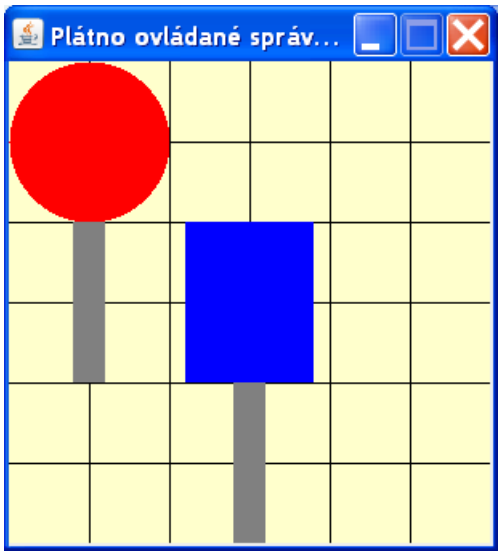
```
   * Vytvoří ceduli správného tvaru na požadovaném místě
   * @param x x-souřadnice
   * @param y y-souřadnice
   * @returns objekt cedule
   */
  protected abstract IKreslený vytvořCeduli(int x, int y);

//== 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(Kreslítko kreslítko) {
    sloupek.nakresli(kreslítko);
    cedule.nakresli(kreslítko);
  }
}
```

- `ZnačkaZákazová` v konstruktoru pouze volá konstruktor předka
  - implementuje `vytvořCeduli()` – vytváří kruh

```
public class ZnačkaZákazová extends ADopravníZnačka {
  public ZnačkaZákazová(int x, int y) {
    super(x, y);
  }

  protected IKreslený vytvořCeduli(int x, int y) {
    return new Elipsa(x, y, 100, 100, Barva.ČERVENÁ);
  }
}
```

- `ZnačkaInformativní` – podobná předchozí, vytváří obdélník

```
public class ZnačkaInformativní extends ADopravníZnačka {
  public ZnačkaInformativní(int x, int y) {
    super(x, y);
  }

  protected IKreslený vytvořCeduli(int x, int y) {
    return new Obdélník(x + 10, y, 80, 100, Barva.MODRÁ);
  }
}
```

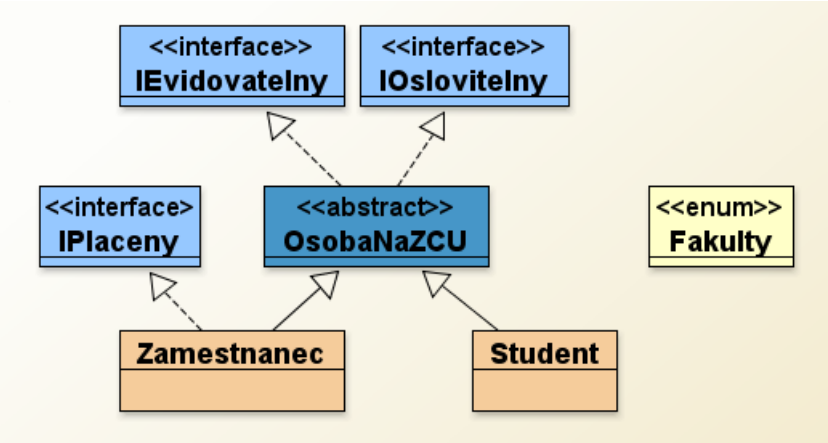



# 5.5.1. Speciální případ abstraktní třídy

- 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 jakékoliv 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 – rychlost, 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 `Array.sort(abc);`
  2. nesprávně `abc.sort();`
- Metody jsou (některé umí pracovat i s částí pole):
  - `equals()` – porovná dvě pole
  - `asList()` – převede pole na kolekci
  - `toString()` – převede 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 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 knihovních 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 ArraysPrirozeneRazeniObecnychObjektu {
  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);

    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` využívá `compareTo()` 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 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! Pro `compareTo()` 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 `KomparatorOsobyPodleXyz`, 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 OsobaAbsolutniRazeni {
  static Osoba[] poleOsob;

  static void vypisOsoby() {
    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);
    vypisOsoby();
*/
    System.out.println("Absolutni razeni podle vahy");
    Arrays.sort(poleOsob, new KomparatorOsobyPodleVahy());
    vypisOsoby();

    System.out.println("Absolutni razeni podle popisu");
    Arrays.sort(poleOsob, new KomparatorOsobyPodlePopisu());
    vypisOsoby();

    System.out.println("Prirozene razeni podle vysky sestupne");
    Arrays.sort(poleOsob, Collections.reverseOrder());
    vypisOsoby();
  }
}
```

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

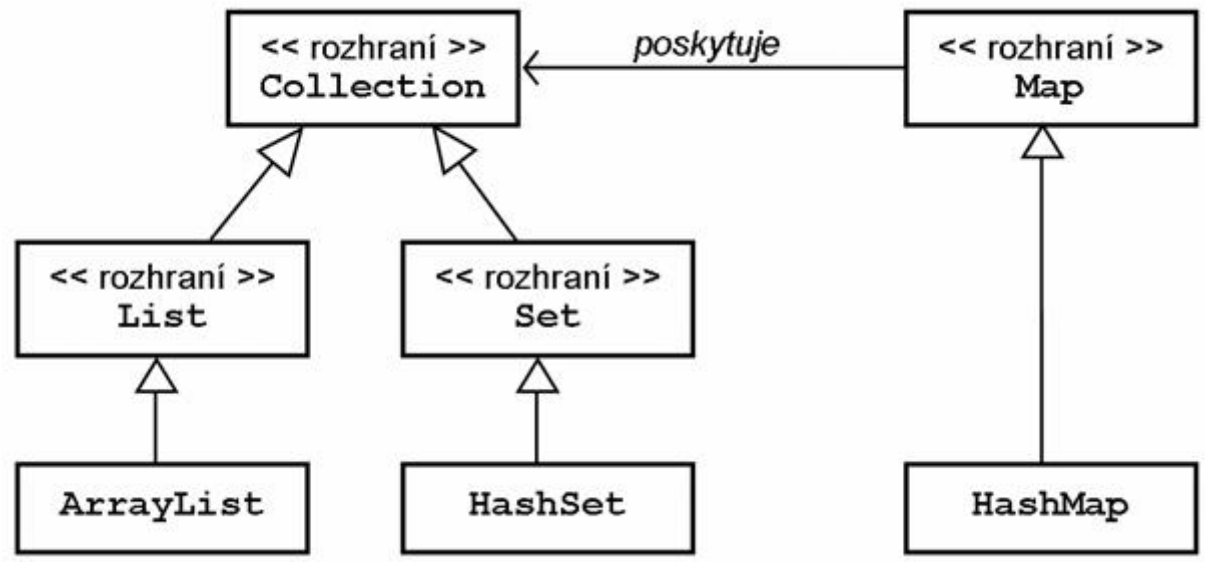
- objekty tříd z balíku `java.util` – *Java Collections Framework*
- slouží k uschovávání většího předem neznámého množství objektů libovolného typu

- „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 1.6) 13 rozhraní, 8 abstraktních tříd a 16 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 dvojic objektů, které jsou ve vzájemném vztahu klíč-hodnota
   - pomocí klíče prvek vyhledáváme, ale 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 `ArrayList`ů

## 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 kompilaci
for (Integer cislo : arTypovany) {
  System.out.print(cislo.intValue() + ", ");
}
```

## Poznámka

Další vylepšení z JDK 1.5 viz později.

## Poznámka

Konstrukce `for()` pro procházení přes všechny prvky kolekce viz též [skr-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<Integer>();
//    List<Integer> seznam = new LinkedList<Integer>();
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<Typ>();
```

nebo později:

```
Collection<Typ> c = new HashSet<Typ>();
```

### 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 OdlisnostiKolekci {
  public static void main(String[] args) {
    List<String> seznam = new ArrayList<String>();
    seznam.add("prvni");
    seznam.add("druhy");
    seznam.add("prvni");
    System.out.println("List: " + seznam);

    Set<String> mnozina = new HashSet<String>();
    mnozina.add("prvni");
    mnozina.add("druhy");
    mnozina.add("prvni");
    System.out.println("Set:  " + mnozina);

    Map<String, String> mapa = new HashMap<String, String>();
    mapa.put("prvni", "objekt");
    mapa.put("druhy", "objekt");
    mapa.put("prvni", "pivo");
    System.out.println("Map:  " + mapa);
  }
}
```

Vypíše:

```
List: [prvni, druhy, prvni]
Set:  [prvni, druhy]
Map:  {prvni=pivo, druhy=objekt}
```

# 6.4. Typové parametry a parametrizované typy

- 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 vyskytují 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<E>();

  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:
  - jakákoliv 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<Object>();
    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<A>();
    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<String>();
    System.out.println("sezn1 je prazdny: " + sezn1.isEmpty());
    sezn1.add("prvni");
    sezn1.add("druhy");
    sezn1.add("prvni");
    tiskni("sezn1", sezn1);

    System.out.println("\nPridavani a ubirani prvku");
    List<String> sezn2 = new ArrayList<String>(sezn1);
    sezn2.add("treti");
    tiskni("sezn2", sezn2);
    sezn2.remove("prvni");
    tiskni("sezn2", sezn2);
    sezn2.removeAll(sezn1);
    tiskni("sezn2", sezn2);
    sezn2.addAll(sezn1);
    tiskni("sezn2", sezn2);
    sezn2.retainAll(sezn1);
    tiskni("sezn2", sezn2);

    System.out.println("\nHledani prvku");
    List<String> sezn3 = new ArrayList<String>(sezn1);
    sezn3.add("ctvrty");
    System.out.println("sezn3 obsahuje 'paty': "
                       + sezn3.contains("paty"));
    System.out.println("sezn3 obsahuje sezn1: "
                       + sezn3.containsAll(sezn1));

    System.out.println("\nPrevod na pole sezn3");
    String[] poleRetezcu = (String[]) sezn3.toArray(new String[0]);
    System.out.println(Arrays.asList(poleRetezcu));
  }
}
```

Vypíše:

```
sezn1 je prazdny: true
sezn1 (3) : [0]=prvni, [1]=druhy, [2]=prvni,
```

```
Pridavani a ubirani prvku
sezn2 (4) : [0]=prvni, [1]=druhy, [2]=prvni, [3]=treti,
sezn2 (3) : [0]=druhy, [1]=prvni, [2]=treti,
sezn2 (1) : [0]=treti,
sezn2 (4) : [0]=treti, [1]=prvni, [2]=druhy, [3]=prvni,
sezn2 (3) : [0]=prvni, [1]=druhy, [2]=prvni,

Hledani prvku
sezn3 obsahuje 'paty': false
sezn3 obsahuje sezn1: true

Prevod na pole sezn3
[prvni, druhy, prvni, ctvrty]
```

## 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<CeleCislo>();
    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 dva 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()`
- „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<Osoba>();
    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("Prirozene 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]
Prirozene 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 jeden typ kolekce (např. `List`) a budeme jej procházet pomocí iterátoru, můžeme někdy v budoucnu (např. z důvodů 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<Typ>();
  ```

  nebo později:

  ```
  Collection<Typ> c = new HashSet<Typ>();
  ```

# 6.8.1. For-Each

- díky typovaným kolekcím nejjednodušší a nejpoužívanější
  - je to též bezpečnější konstrukce
- musí projít celou kolekci 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<Integer>();
    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*)
  - zprostředkuje 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 kolekcí 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 jednoprůchodový – po průchodu pozbývá funkčnost. Pro případný další průchod se musí znovu 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 hrušky. Dále je vidět běžné využití překryté metody `toString()`.

```
import java.util.*;

class Hruska {
  private int cena;
  Hruska(int cena) { this.cena = cena; }
  public String toString() { return "" + cena; }
  public void tisk() { System.out.print(cena + ", "); }
}

public class IteratorZakladniPouziti {
  public static void main(String[] args) {
    List<Hruska> kosHrusek = new ArrayList<Hruska>();
    for (int i = 0;  i < 10;  i++) {
      kosHrusek.add(new Hruska(i + 20));
    }

    for (Iterator<Hruska> it = kosHrusek.iterator(); it.hasNext(); ) {
      System.out.print(it.next() + ", ");
    }
    System.out.println();

    Iterator<Hruska> it = kosHrusek.iterator();
    it.next();
    it.next();
    while (it.hasNext()) {
      it.next().tisk();
    }
    System.out.println();
  }
}
```

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<Hruska> kosHrusek = new ArrayList<Hruska>();
    for (int i = 0;  i < 11;  i++) {
      kosHrusek.add(new Hruska(i + 20));
    }

    for (Iterator<Hruska> it = kosHrusek.iterator();
                                          it.hasNext(); ) {
      Hruska h = it.next();
      System.out.print(h + ", ");
      h.tisk();
    }
//    it.next();    // neni pristupny - nelze udelat chybu
    System.out.println();

    Iterator<Hruska> it1 = kosHrusek.iterator();
//    Iterator<Hruska> it2 = kosHrusek.iterator();
    it1.next();
    it1.next();
    while (it1.hasNext()) {
      it1.next().tisk();
      it1.remove();
    }

    System.out.println("\nPo ubrani hrusek");
    Iterator<Hruska> it2 = kosHrusek.iterator();
    while (it2.hasNext()) {
      it2.next().tisk();
    }
    System.out.println();
  }
}
```

vypíše:

```
20, 20, 21, 21, 22, 22, 23, 23, 24, 24, 25, 25, 26, 26, 27, 27, 28, 28, 29, ►
29, 30, 30,
22, 23, 24, 25, 26, 27, 28, 29, 30,
Po ubrani hrusek
20, 21,
```

# 6.9. 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 (Athlon, 1,4 GHz) a velikosti operační paměti (512 MB)
- byly testovány dvě třídy – běžný `ArrayList`, `LinkedList`
- velikost seznamu byla 100 000 prvků

| | ArrayList | LinkedList |
|---|---|---|
| naplnění | 241 | 391 |
| průchod indexací | 10 | 552 454 |
| průchod iterátorem | 20 | 30 |
| vypuštění poloviny indexací zezadu | 5 477 | 178 107 |
| vložení poloviny indexací | 5 719 | 174 841 |
| vypuštění poloviny indexací zepředu | 57 723 | 175 743 |
| clear a naplnění | 100 | 360 |
| vypuštění poloviny iterátorem | 57 713 | 40 |

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. Jakákoliv indexace v `LinkedListu` je extrémně pomalá.

4. Hromadné vypouštění nebo vkládání prvků do `ArrayListu` je časově velice náročné. U vypouštění prvků je `LinkedList` zhruba 1000 krát výkonnější, a je tedy velmi vhodné jej pro tento typ aplikace použít. Na druhé straně je ale problémem `LinkedListu` vkládání, které je zhruba 6krát 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.

# 6.10. 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 témž 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<Integer>();
    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 vyjimku
//    System.out.println(it.next());

    for (Integer i : kolekce) {
      System.out.println(i);
      kolekce.add(new Integer(20));
    }
  }
}
```

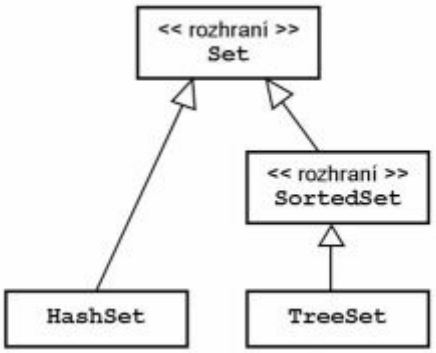
Vypíše:

```
0
Exception in thread "main" java.util.ConcurrentModificationException
```

# 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 definovánu 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 rychlost vyhledání prvku (O1)
  - 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 vlozeni stejneho prvku
    if (mnozina.add("treti") == false) {
      System.out.println("'treti' podruhe nevlozen");
    }
    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' podruhe nevlozen
3 [prvni, treti, druhy]
prvni, treti, druhy,
'treti' je v mnozine
[prvni, druhy]
TreeSet
'treti' podruhe nevlozen
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 již známý 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 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), která 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 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);
    }
  };

  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 stejné 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 vypočítává metoda `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 svojí 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. zkontrolujeme 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 připravenu 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 vynásobení 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<Osoba>(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);
  }
}
```

| | prvků | 1000 | 10000 | 20000 | 30000 |
|---|---|---|---|---|---|
| NevhodnaOso-ba | vložení | 20 | 1392 | 7050 | 17185 |
| | přístup | 10 | 1351 | 7000 | 16904 |

| | prvků | 1000 | 10000 | 100000 | 500000 |
|---|---|---|---|---|---|
| `PrijatelnaOsoba` | vložení | 10 | 51 | 1542 | 25066 |
| | přístup | 10 | 20 | 771 | 18106 |
| `PerfektniOsoba` | vložení | 10 | 40 | 961 | 8051 |
| | přístup | 10 | 20 | 130 | 651 |
| `NemennaPerfektniOsoba` | vložení | 10 | 40 | 871 | 7761 |
| | přístup | 0 | 10 | 80 | 411 |

1. Hodnoty pro 1000 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. Na příkladu `NemennaPerfektniOsoba` je vidět, že pokud nechceme měnit stav vkládaného objektu, můžeme zvýšit rychlost až o jednu třetinu.

3. 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 o třetinu (oproti `NemennaPerfektniOsoba`), ale zůstává stále velmi malý. To se u `PrijatelnaOsoba` (s jednodušším výpočtem) zdaleka nedá říci (čas narůstá nelineárně).

- jak generované hešovací kódy splňují podmínku unikátnosti pro 100 tisíc různých objektů:
  - NevhodnaOsoba – 31 rozdílných
  - PrijatelnaOsoba – 11147 rozdílných a 88853 shodných
  - PerfektniOsoba – 99998 rozdílných a pouze 2 shodné

## 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
- lze 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<String>();
   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<String>();
    d.add("prvni");
    d.add("druhy");
    d.add("prvni");
    System.out.println("Duplicitni:   " + d);
    Collection<String> nd = new ArrayList<String>(
                            new HashSet<String>(d));
    System.out.println("NEduplicitni: " + nd);
  }
}
```

Vypíše:

```
Duplicitni:   [prvni, druhy, prvni]
NEduplicitni: [druhy, prvni]
```

- průniky, sjednocení apod.

```
Set<String> m1 = new HashSet<String>();
m1.add("1");
m1.add("2");
m1.add("3");
m1.add("4");
Set<String> m2 = new HashSet<String>();
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 HashSet<String>(m1);
sjednoceni.addAll(m2);
System.out.println(sjednoceni + " je sjednocenim " + m1 + " a " + m2);
Set<String> prunik = new HashSet<String>(m1);
prunik.retainAll(m2);
System.out.println(prunik+ " je prunikem "+ m1+" a "+ m2);

Set<String> rozdil = new HashSet<String>(m1);
rozdil.removeAll(m2);
System.out.println(rozdil+ " je rozdilem "+ m1+ " a "+m2);

Set<String> symetrickyRozdil = new HashSet<String>(m1);
symetrickyRozdil.addAll(m2);
Set<String> tmp = new HashSet<String>(m1);
tmp.retainAll(m2);
symetrickyRozdil.removeAll(tmp);
System.out.println(symetrickyRozdil +
                     " je symetrickym rozdilem " + m1 + " a " + m2);
```

Vypíše:

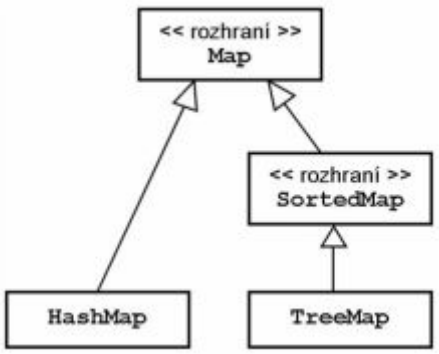
```
[3, 2] je podmnozinou [3, 2, 4, 1]
[3, 5, 2, 4, 1] je sjednocenim [3, 2, 4, 1] a [3, 5, 2]
[3, 2] je prunikem [3, 2, 4, 1] a [3, 5, 2]
[4, 1] je rozdilem [3, 2, 4, 1] a [3, 5, 2]
[5, 4, 1] je symetrickym rozdilem [3, 2, 4, 1] a [3, 5, 2]
```

# 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 je neměnný a unikátní, se vyhledává hodnota

- hodnota je proměnná 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 důvodů 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>> s = hm.entrySet();
    for (Map.Entry<String, Integer> me: s) {
      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, druhy=2,
prvni=1, treti=3, druhy=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 ztloustnutí 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 konstruktoru `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<String, String>();
    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<String, String>(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[] s = {"lesni", "vily", "vence", "vily", "a",
                  "psi", "z", "vily", "na", "ne", "vyli"};
    Map<String, Integer> m = new TreeMap<String, Integer>();
    Integer c;
    for (int i = 0;  i < s.length;  i++) {
      int cet = 0;
      if ((c = m.get(s[i])) != null) {
        cet = c.intValue();
      }
      m.put(s[i], cet + 1);
    }

    System.out.println("Nalezeno " + m.size() +
                                    " rozdilnych slov");
    System.out.println(m);
  }
}
```

Vypíše:

```
Nalezeno 9 rozdilnych 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 neomezují
  - do kolekce lze vložit jiná kolekce
  - tím lze vytvářet libovolně složité datové struktury
- typické příklady:
  - dvourozměrný 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<String>());
    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);
      }
    }

    // pruchod iteratorem
    for (List<String> pomSeznam : dvourozmernySeznam) {
      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 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<String, Set<String>>();

    for (int i = 0;  i < muzi.length;  i++) {
      Set<String> mnozinaPritelkyn = new HashSet<String>();

      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);
    }

    // pruchod iteratorem
   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. Polymorfismus, UML, interní datové typy

# 8.1. Polymorfismus

- princip polymorfismu („vícetvarost“) 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 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 instance potomků a v případě zpracování (typicky pomocí *For-Each*) nemusíme vůbec rozlišovat konkrétního potomka
  - zpracování je extrémně jednoduché

```
for (Předek instance : kolekce) {
 instance.vykonejČinnost();
}
```

  - je to sjednocení práce s různými typy 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.

## 8.1.1. 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`

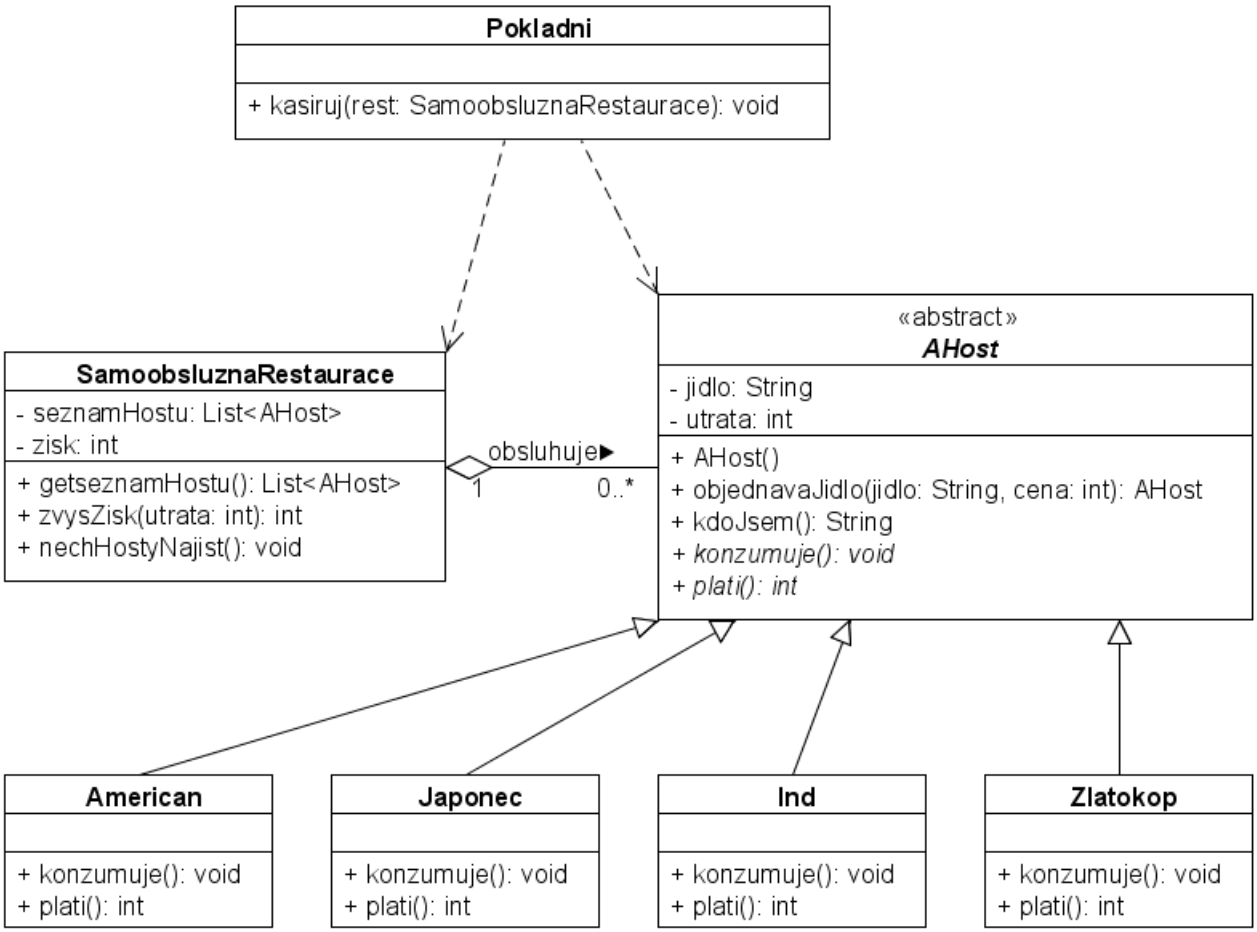
- místo nich pracuje pouze s jejich předkem `AHost`, která bude v metodě `nechHostyNajist()` volat metodu `konzumuje()`

```
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<AHost>();
  private int zisk = 0;

  public List<AHost> getseznamHostu() {
    return seznamHostu;
  }

  public int zvysZisk(int utrata) {
    zisk += utrata;
    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 `Hlavni`

```
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));
    seznamHostu.add(new Japonec().objednavaJidlo("susi", 200));
    seznamHostu.add(new Ind().objednavaJidlo("capati", 80));
    seznamHostu.add(new Zlatokop().objednavaJidlo("fazole", 50));

    restaurace.nechHostyNajist();
    new Pokladni().kasiruj(restaurace);

    System.out.println("Celkova utrata: " + restaurace.zvysZisk(0));
  }
}
```

  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

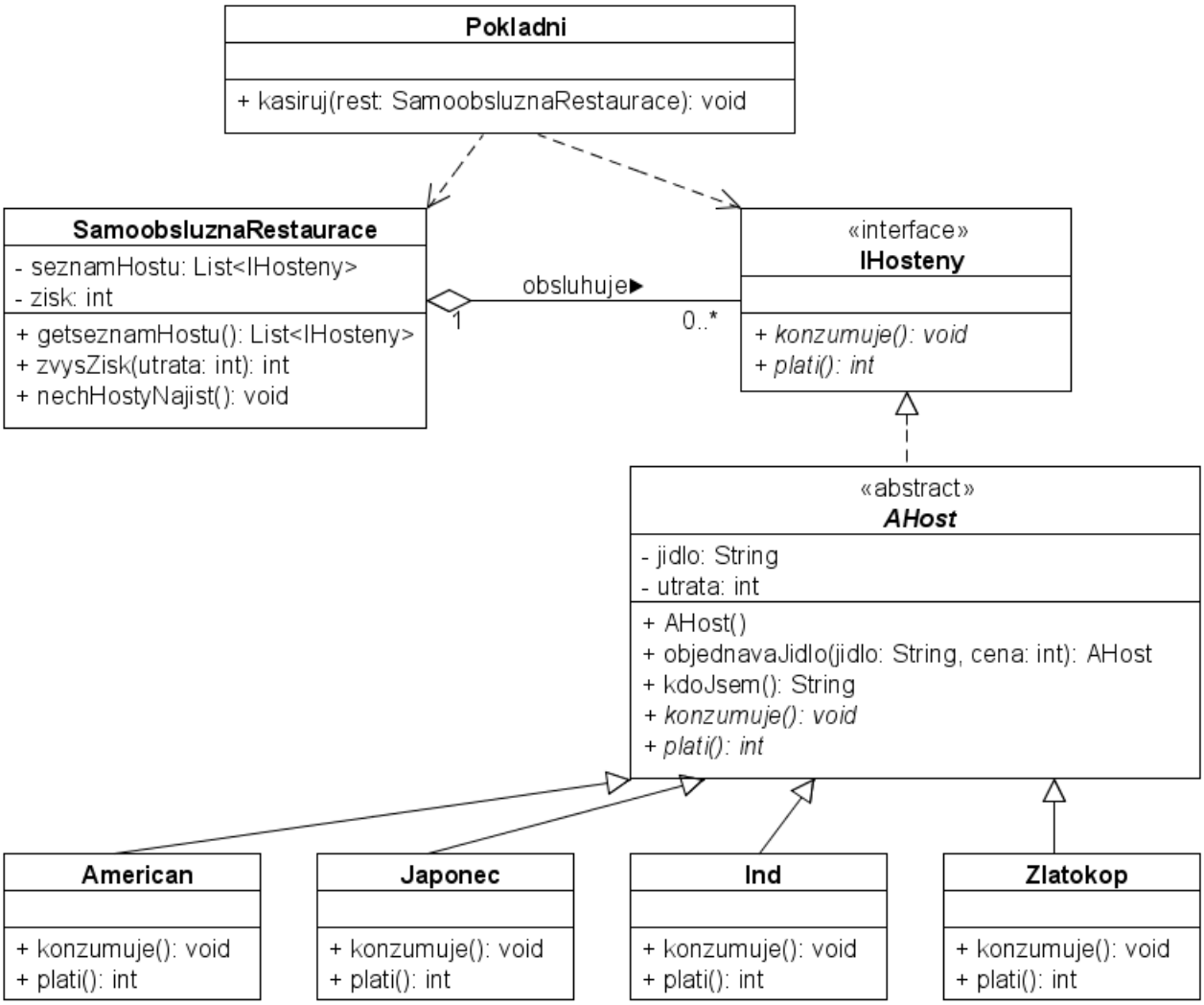
# 8.1.2. Využití polymorfismu pomocí rozhraní

- 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
- v příkladu bude jen nepatrná změna zamění se `AHost` za `IHosteny` ve třídách `SamoobsluznaRestaurace` a `Pokladni` a třída `AHost` bude mít hlavičku

```
abstract public class AHost implements IHosteny {
```

  - vše ostatní zůstane stejné




# 8.2. UML

- *Unified Modeling Language* je grafický jazyk pro vizualizaci, specifikaci a navrhování programových systémů
  - formát dokumentace pro analýzu je UML
    - formát dokumentace pro kód je Javadoc
  - UML je „programovací“ jazyk analytika nezávislý na existujících programovacích jazycích
    - jazyk vizuálního modelování
- důvod rozšíření – umožňuje **jednotně** zapsat proces vývoje tak, aby mu rozuměli analytici, programátoři i zákazníci
  - ALE! Použití UML automaticky nezajišťuje projektu úspěch!
  - neobsahuje metodiku, jak analyzovat, specifikovat či navrhovat programové systémy
- 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 metodiků 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 2.3 z května 2010 (`www.omg.org`)
- základní princip
  - založen na grafických prvcích – diagramech, jejichž podoba je striktně předepsána a musí (měla by) se dodržovat
    - protože se správná podoba diagramů nedá vynutit, např. BlueJ používá mnoho zjednodušení

- 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 říká CO má systém dělat, nikoliv JAK bude implementován

■ vytváření UML diagramů

- pro práci v UML je nutná jen tužka a papír (žádný „překladač“)
- různé podpůrné nástroje „pouze“ ulehčují zápis jednotlivých modelů, případně jejich změny
  - ♦ existuje velké množství nástrojů různých kvalit a možností
  - ♦ špička – nástroje Rational Rose
  - ♦ my budeme využívat UMLet (`www.umlet.com`)

    **Poznámka**

    Pozor při stahování – Google tam nabízí jiné produkty.

# 8.2.1. 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 použít více různých jednoduchých pohledů (modelů) na systém

■ UML definuje 14 modelů zapsaných pomocí diagramů

- modely jsou složeny z elementů UML, představujících jednotlivé stavební díly
- každý model představuje jiný pohled na informační systém a nemusejí být všechny použity
- slova **model** a **diagram** zde představují synonyma, častěji se používá diagram

■ 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*)
  - ♦ sekvenční diagram (*sequence diagram*)
  - ♦ diagram komunikace (*communication diagram*)
  - ♦ diagram interakcí (*interaction overview diagram*)
  - ♦ diagram časování (*timing diagram*)

**Poznámka**

Toto dělení je pouze jedno z mnoha možných, se kterými se lze setkat. Další způsoby dělení jsou např. na statické a dynamické, na implementační a analytické atd.

- 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
  - všechny ostatní diagramy se používají dle potřeby

# 8.2.2. Společné části diagramů

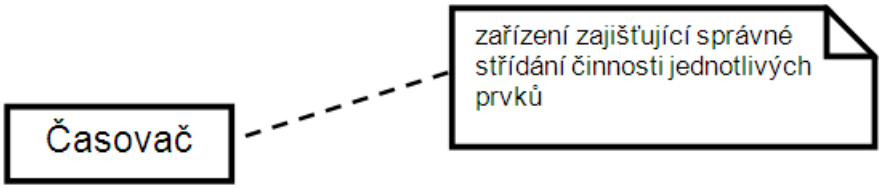
- mohou se vyskytnout u jakéhokoliv ze zmíněných modelů nebo jejich částí

**Poznámka**

Další příklady se budou vztahovat k modelu pračky prádla.

## 8.2.2.1. Poznámka (*note*)

- lze přiřadit k jakémukoliv elementu včetně šipky a spojovací čáry
- přidává vysvětlující text, čím více je **smysluplných** poznámek, tím lépe



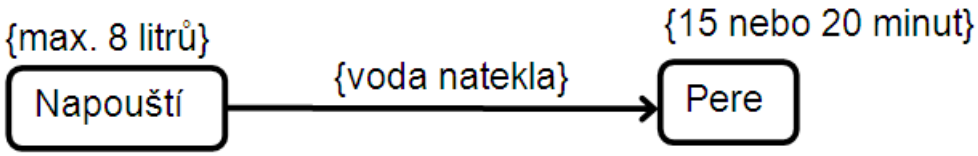

## 8.2.2.2. Stereotyp (*stereotype*)

- slouží k dodatečné kategorizaci objektů
- použijí se již existující prvky UML a vytvoří se z nich nové

- používá se zápis `<<stereotyp>>` (francouzské uvozovky)
- některé stereotypy jsou již předefinovány
  - např. rozhraní, výčtový typ, abstraktní třída jsou speciální případy třídy

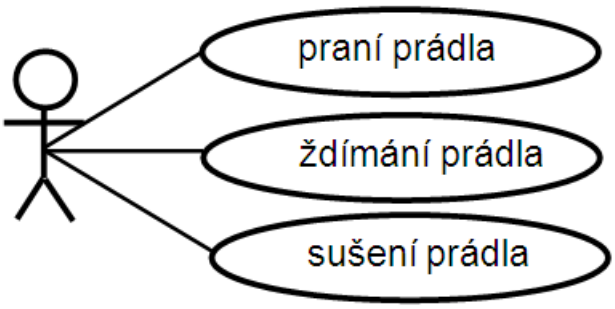

### 8.2.2.3. Omezení (*constraint*)

- lze přiřadit k jakémukoliv elementu
- je to často podmínka true/false, nebo výčty
- používá se zápis `{omezení}` (složené závorky)
- není-li omezení splněno, model se stává nekonzistentní

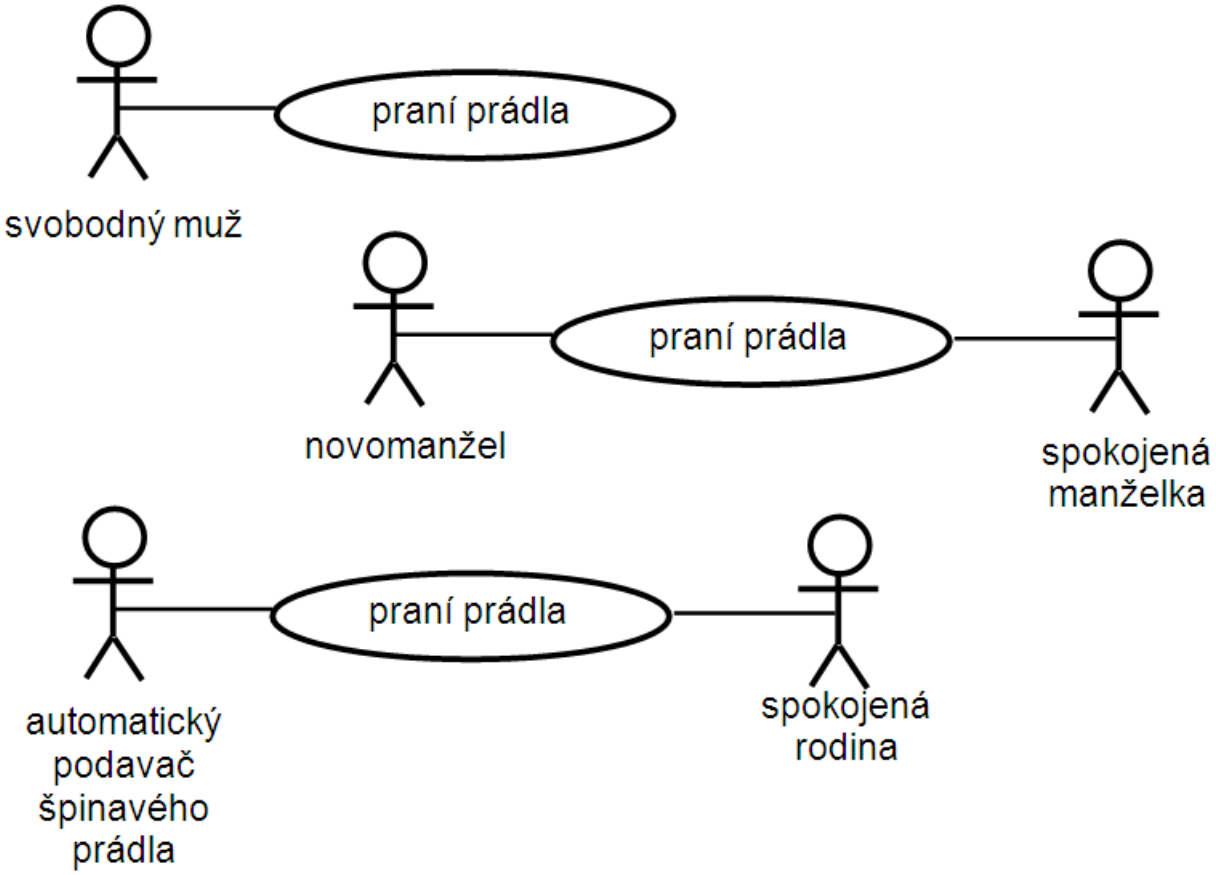



## 8.2.3. Diagram případů užití

- slouží pro analytický model vytvářeného systému
  - je na pojmové úrovni, tj. srozumitelný všem
- popisuje k čemu všemu se dá (informační) systém použít – mohou to být tisíce použití
  - zaznamenává vyčerpávajícím způsobem všechny funkce systému z hlediska uživatele
- je-li hotový model případů užití (popsaný diagramem případů užití), je popsán informační 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, participanta, aktéra, roli)




- z případů užití je zřejmé, že vytvářený systém Pračka bude umožňovat uživateli pouze tři aktivity – praní, ždímání 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)
  - 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




# 8.2.4. Diagram tříd

- slouží pro analytický model vytvářeného systému ale i pro model návrhu (*design*)
- zaznamenává existenci tříd a vztahy mezi nimi (asociace, agregace, kompozice, dědění, implementace, …)
  - analýza se věnuje zejména třídám aplikační logiky (málokdy třídám GUI)

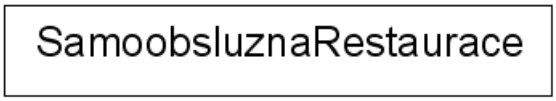
- BlueJ používá značně zjednodušený diagram tříd

**Poznámka**

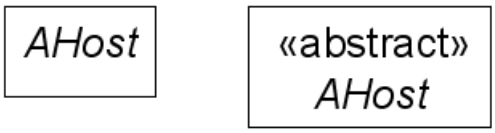
Další příklady se vztahují k příkladu využití polymorfismu – viz dříve.

## 8.2.4.1. Způsoby zápisu třídy

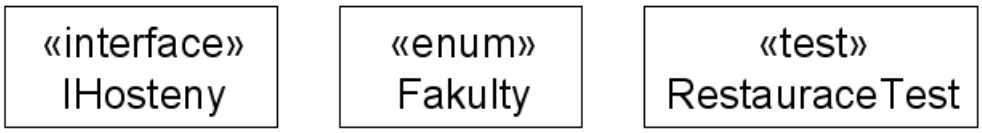
- v nejjednodušší podobě je třída představována obdélníkem se svým jménem



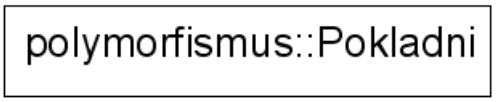

- jedná-li se o abstraktní třídu, používá se kurzíva a/nebo častěji stereotyp




- rozhraní, výčtové typy a JUnit testy jsou označeny příslušnými stereotypy

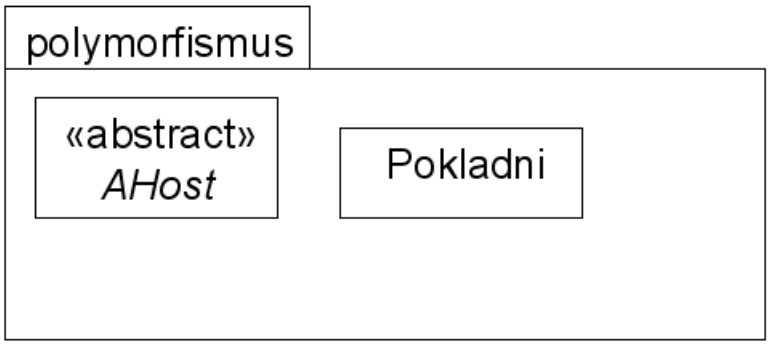



- pokud chceme uvést, že třída náleží do nějakého balíku, použijeme `názevbalíku::Třída`

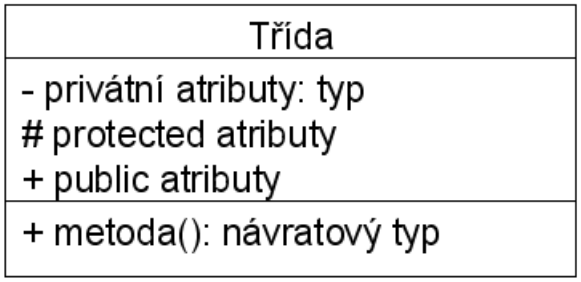



  - tento způsob se ale používá velmi zřídka, jen pokud chceme zdůraznit použití třídy z jiného balíku

- použití balíků

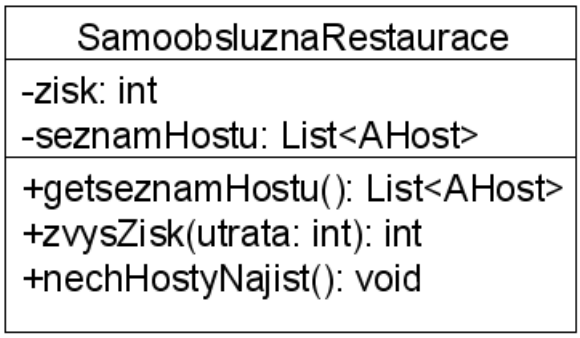



- u třídy lze uvést i atributy a metody, pak se obdélník třídy dělí vodorovně na tři části: jméno třídy, atributy, metody

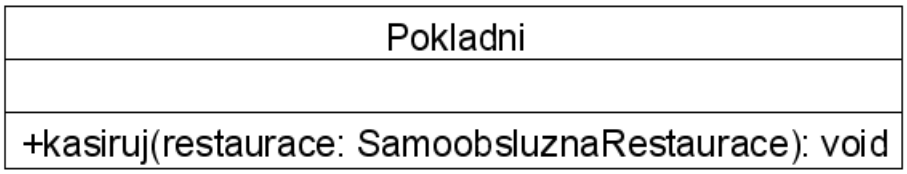



- atributy i metody mohou používat označení přístupových práv – pro `private`, `#` pro `protected` a `+` pro `public`
- typ atributu, typ formálního parametru nebo návratový typ metody se píše až za název oddělený :
  - `zisk: int`
  - `zvysZisk(utrata: int)`
  - `getSeznamHostu(): List<AHost>`

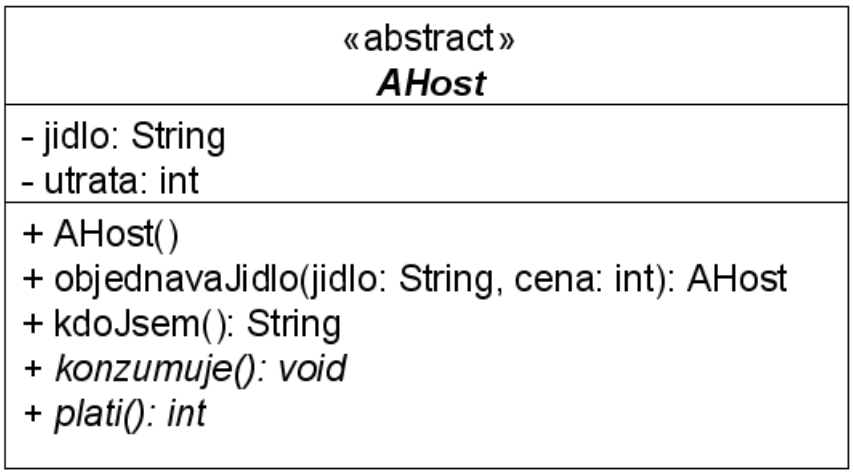
■ příklad kompletní informace




■ pokud atributy nebo metody chybějí (nebo nepovažujeme za nutné je uvádět – často u `private`), pak musí být příslušná část prázdná




■ jedná-li se o abstraktní metody, jsou zapsány kurzívou: `konzumuje()` a `plati()`

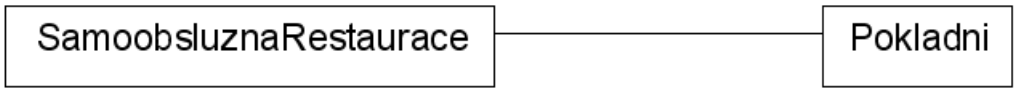



## 8.2.4.2. Vazby tříd

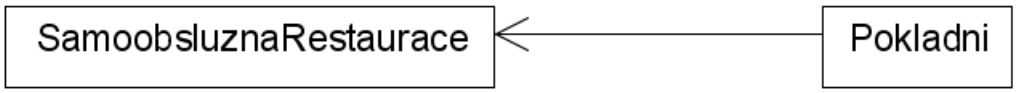
- třídy jsou navzájem v mnoha vztazích, které se rozlišují typem spojovací čáry a případně i dodatečnými informacemi

> **Poznámka**
>
> Nebudou zde uvedeny všechny možnosti, pouze ty nejčastěji využívané. Další možnosti lze vidět v souboru `vztahy.zip`.

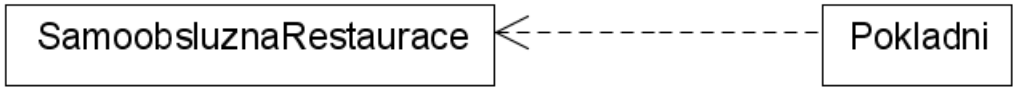
- **asociace**
  - říká, že mezi dvěma třídami existuje blíže neurčený vztah, nejčastěji jedna třída nějak používá druhou




  - někdy je vztah pouze jednosměrný, což se naznačí šipkou




    - zde to znamená, že `SamoobsluznaRestaurace` vůbec nepotřebuje (neví o) existenci `Pokladni`
      - v řešeném příkladu je objekt třídy `SamoobsluznaRestaurace` předán jako parametr metody `kasiruj()`
        - v tomto typu vztahu nemá `Pokladni` jako atribut referenční proměnnou na `SamoobsluznaRestaurace`
      - v tomto případě se vztah označuje jako **závislost** (*dependency*) a je zakreslen

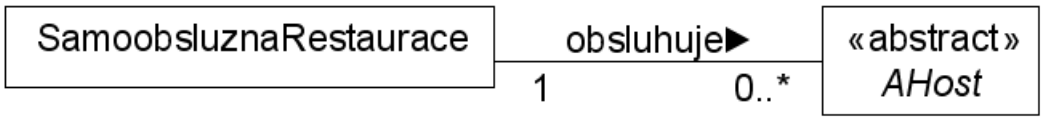


### Poznámka

Tento typ šipky (otevřená) se používá jako doplňující informace i v dále uváděných vazbách.

- k asociaci lze dodat i vzájemný vztah, který může být i obousměrný


- k asociaci lze dodat také násobnost (kardinalitu) vztahu




  - ♦ `SamoobsluznaRestaurace` obsluhuje třídu `AHost`
  - ♦ `SamoobsluznaRestaurace` bude existovat v jedné instanci, `AHost` bude existovat v žádné nebo ve více instancích
- možné kardinality 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`

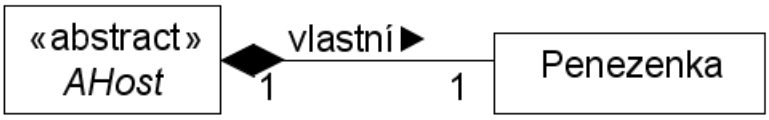
### Poznámka

Vícenásobnost vztahu vede při implementaci většinou na použití nějakého kontejneru (možno i pole), neohraničená násobnost (`*`) pak na kolekci.

■ **agregace**

- zkonkretizování asociace
- dvě možnosti dle vztahu agregované třídy k partnerské třídě

1. **kompozice** („složenina“) nebo **silná agregace**
   - ♦ třída má vztah pouze k jedné třídě a většinou s ní vzniká a zaniká, samostatně nemá smysl
   - ♦ vlastnící třída má často stejně pojmenovaný atribut



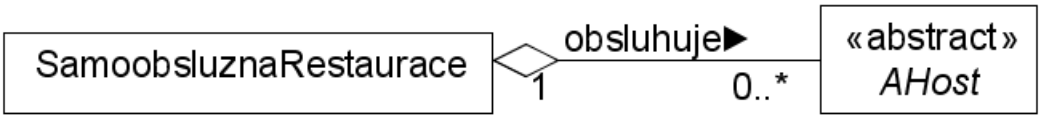

- `AHost` vlastní právě jednu `Penezenka`
  - ve zdrojovém kódu má `AHost` atribut typu referenční proměnná pravděpodobně pojmenovaný `penezenka`
- instance `Penezenka` vzniká v konstruktoru `AHost`, případně je do konstruktoru předána
- neexistují žádné další (vnější mimo třídu `AHost`) reference na instanci `Penezenka`, tzn. nikdo jiný `Penezenka` nepoužívá, protože mu není přístupná

### Poznámka

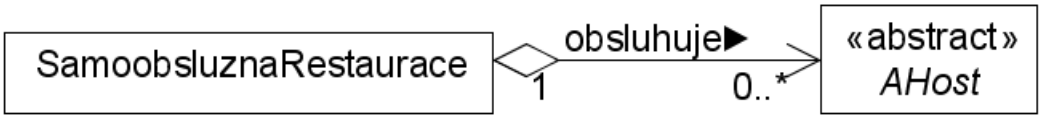
Silné agregace se vyskytují velmi zřídka. Musí pro ně být skutečně vážný důvod. Pokud se vyskytují, jsou nejčastěji realizovány pomocí **vnitřní třídy** – viz dále.

- chceme-li zdůraznit, že `Penezenka` nemá žádnou informaci o `AHost`, můžeme to zdůraznit otevřenou šipkou

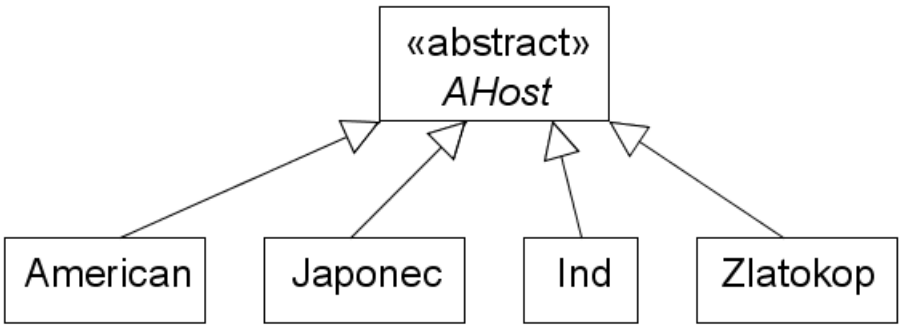

2. **agregace**
   - třída je (nebo může být) využívána více třídami
   - vzniká nezávisle na vzniku těchto tříd, odkazy na ní jsou do nich předány nejčastěji voláním metod
   - ve zdrojovém kódu má `SamoobsluznaRestaurace` atribut typu referenční proměnná na typ `AHost` (při kardinalitě 1:1) nebo na kolekci `<AHost>` (při kardinalitě 1:0..*)



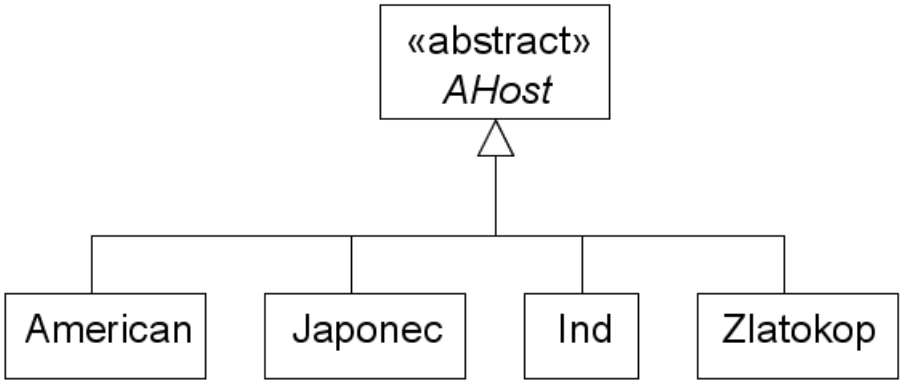

- `SamoobsluznaRestaurace` nevytváří ani nevlastní `AHost`, pouze jej používá
  - stejně tak `AHost` používá i `Pokladni`
- seznam `AHost` v `SamoobsluznaRestaurace` je naplněn tím, kdo `SamoobsluznaRestaurace` používá (zde `Hlavni`)
- se zánikem `SamoobsluznaRestaurace` nezaniká `AHost`
- opět je možné zdůraznit jednosměrnou závislost pomocí otevřené šipky




- **dědění**
  - používá se plná čára s prázdnou šipkou od potomka k předkovi
    - pokud je to možné, snažíme se, aby šipka směřovala vzhůru

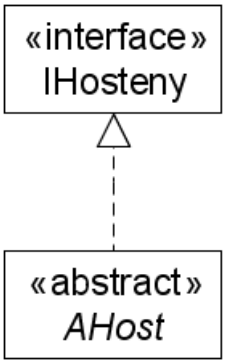



  - v případě násobných dědičností, lze pro zvýšení přehlednosti použít

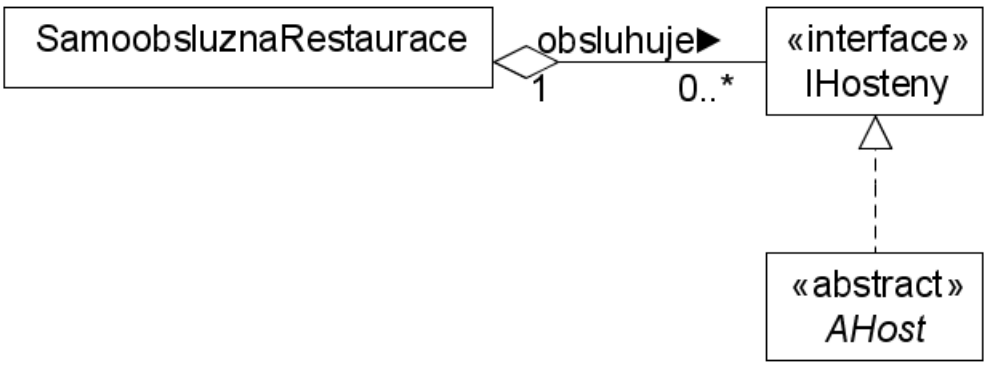



- **implementace rozhraní** – zde máme dvě možnosti

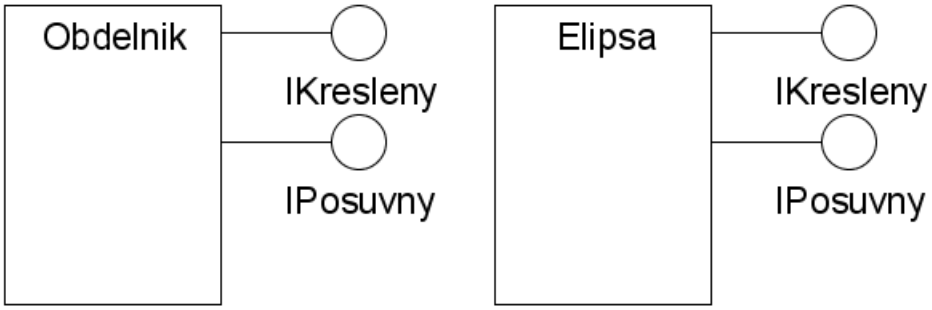
1. implementace několika málo rozhraní je klíčová pro celý program
   - použijeme přerušovanou čáru s uzavřenou šipkou




   - např. rozhraní je využíváno jako typ v dalších třídách, např. `SamoobsluznaRestaurace`

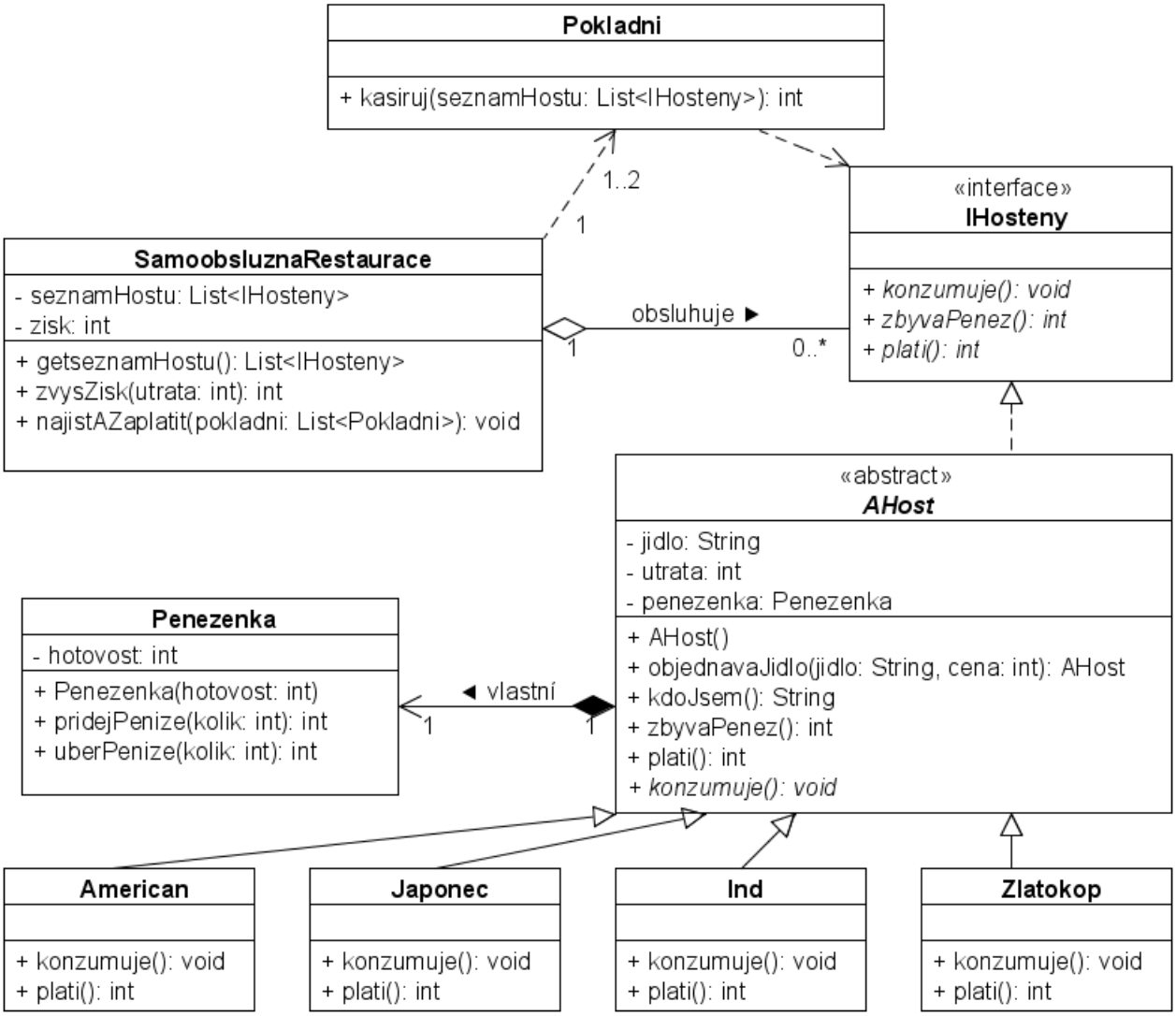



2. množství tříd implementuje množství rozhraní a vzájemné vazby by byly nepřehledné
   - demonstrujeme tím schopnosti objektů, které nejsou klíčové pro program




# 8.2.5. Příklad na různé vztahy mezi třídami

- příklad je modifikací příkladu uvedeného dříve v polymorfismu
  - snahou návrhu je, aby třídy `SamoobsluznaRestaurace` a `Pokladni` nebyly pokud možno provázány těsnými vazbami
    - provázání bude provedeno v hlavní třídě




- třída `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() + "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<IHosteny>();
  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(polovinaHostu(true));
      zisk += pokladni.get(1).kasiruj(polovinaHostu(false));
    }
  }

  private List<IHosteny> polovinaHostu(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 Hlavni {

  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));

    List<Pokladni> seznamPokladnich = new ArrayList<Pokladni>();
    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 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
```

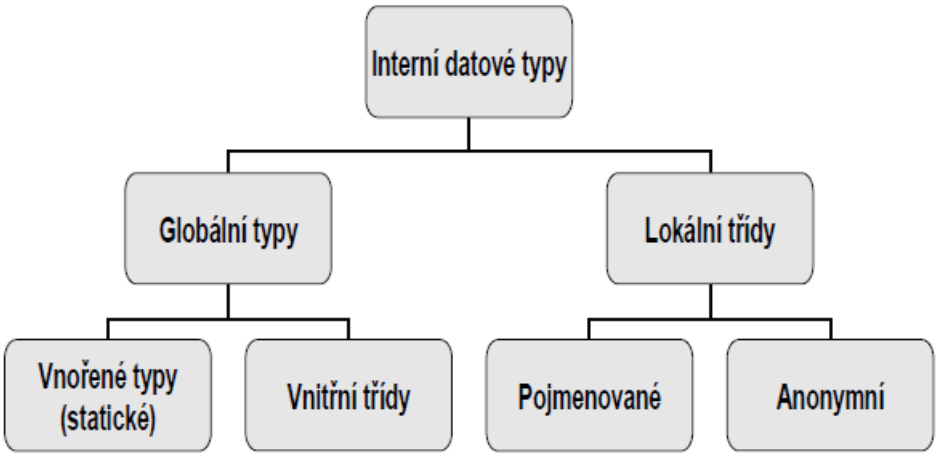
# 8.3. Interní datové typy

- 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$1.class` – anonymní lokální třída
- `IO$Odháčkuj.class` – vnořená třída
- `IO$Odháčkuj$1.class` – první anonymní lokální třída vnořené třídy `Odháčkuj`
- `IO$Odháčkuj$2.class` – druhá anonymní lokální třída vnořené třídy `Odháčkuj`
- `IO$Odháčkuj$INormalizer.class` – vnořené rozhraní vnořené třídy `Odháčkuj`

### Poznámka

Takto pojaté zanořování je extrémní případ, který si může dovolit použít jen skutečný odborník. Běžně se setkáváme pouze s jednou úrovní zanoření.

- 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 (tlačítko 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 agregace** – 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*

# 8.3.1. 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, FEK, FEL, FF, FPE, FPR, FST, FZS, UJP, UUD;
  }

}

class Pouziti {
  public static void main(String[] args) {
    System.out.println("Pracuji na fakulte " + ZCU.Fakulty.FAV);
  }
}
```

- další příklad viz dále

# 8.3.2. 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 `muz` a `zena`
  - v příkladu budou vynechány nepodstatné metody ze třídy `Rande`

```
/*****************************************************************
 * Instance třídy {@code Rande} představují dvojice Osob,
 * které jsou na schůzce
 *
 * @author    Pavel Herout
 * @version   1.00.000
 */
public class Rande {

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

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

  /** muž */
```

```
  private final Osoba muz;
  /** žena */
  private final Osoba zena;

  /** domácí pozice */
  private final Pozice domaMuz;
  private final Pozice domaZena;

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

  /*************************************************************
   * vytvoří instance muže a ženy na domácích pozicích
   * zajistí jejich zobrazení
   *
   * @param domaMuz domácí pozice muže
   * @param domaZena domácí pozice ženy
   */
  public Rande(Pozice domaMuz, Pozice domaZena) {
    muz = Osoba.getBeznyMuz(domaMuz);
    this.domaMuz = domaMuz;
    zena = new Osoba(domaZena, Pohlavi.ZENA);
    this.domaZena = new Pozice(domaZena.x, domaZena.getY());
    muz.zobraz();
    zena.zobraz();
  }


//== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ ======================
  /**
   * z dvojice na jednom místě vytvoří pár a přesune jej na zadané místo
   *
   * @param mistoVyletu pozice místa prvního výletu
   * @param chuzeSpolu přesouvač
   * @return vytvořený pár pro účely dalších přesunů
   */
  public Par parJdeSpolecne(Pozice mistoVyletu, Presouvac chuzeSpolu) {
    Par par = new Par();
    SP.pridej(par);
    chuzeSpolu.presunNa(par, mistoVyletu);
    return par;
  }

  /**
   * přesune pár na místo schůzky, žena je z pohledu muže vpravo
   *
   * @param mistoVyletu pozice místa prvního výletu
   * @param chuzeSpolu přesouvač
   */
  public void parPokracujeSpolecne(Par par, Pozice dalsiMistoVyletu, Presouvac ▶
chuzeSpolu) {
    chuzeSpolu.presunNa(par, dalsiMistoVyletu);
  }
```

```
  public void jdouDomu(Par par, Presouvac chuzeDomu) {
    chuzeDomu.presunNa(par, domaZena);
    chuzeDomu.presunNa(muz, domaMuz);
  }

//== INTERNÍ DATOVÉ TYPY ============================================
  public class Par implements IKresleny, IPosuvny {

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

    /**
     * pár vzniká jen metodou parJdeSpolecne()
     */
    private Par() {
    }

  //== OSTATNÍ NESOUKROMÉ METODY INSTANCÍ =====================

    /**
     * vykresli instanci
     */
    @Override
    public void nakresli(Kreslitko kreslitko) {
      SP.nekresli(); {
        muz.nakresli(kreslitko);
        zena.nakresli(kreslitko);
      } SP.vratKresli();
    }


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


   /**************************************************************************
     * Presune posuvny objekt do nove pozice.
     *
     * @param pozice  Nova pozice objektu.
     */
    @Override
    public void setPozice(Pozice pozice) {
      setPozice(pozice.x, pozice.y);
    }


   /**************************************************************************
     * 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);
      muz.setPozice(x + zena.getSirka(), y);
    }
  }
}
```

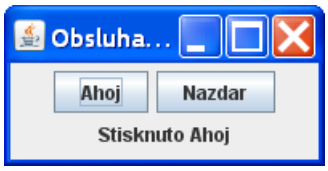
- použití např. ve třídě `Hlavni`

```
    Pozice pM = new Pozice(10, 10);
    Pozice pZ = new Pozice(400, 150);
    Rande rande = new Rande(pM, pZ);

    Presouvac chuzeNaRande = new Presouvac(rychlostNaRande);
    Pozice pS = new Pozice(150, 80);
    rande.jdouNaRande(pS, chuzeNaRande);

    Pozice pV = new Pozice(10, 10);
    Presouvac chuzeSpolu = new Presouvac(rychlostSpolu);
    Rande.Par par = rande.parJdeSpolecne(pV, chuzeSpolu);

    Pozice pD = new Pozice(350, 10);
    rande.parPokracujeSpolecne(par, pD, chuzeSpolu);
    Pozice pK = new Pozice(10, 150);
    rande.parPokracujeSpolecne(par, pK, chuzeSpolu);

    Presouvac chuzeDomu = new Presouvac(rychlostDomu);
    rande.jdouDomu(par, chuzeDomu);
```

## 8.3.2.1. 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

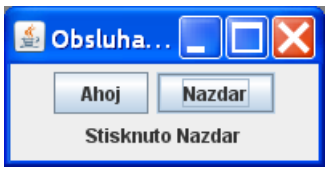
# 8.3.3. Anonymní lokální

- třídy na jedno použití, na které se už nikdy nebudeme odvolávat, a proto nemusejí mít jméno
- nesmějí mít statické atributy a metody
- typicky jsou to implementace rozhraní na místě

## 8.3.3.1. 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

## 8.3.3.2. 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
```

## 8.3.4. Pojmenované lokální

- není žádný běžný důvod, proč je používat

# 8.4. Návrhový vzor 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 myši
- 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");
    }
  }
```

# 8.4.1. 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 adaptéru 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();
  }
}
```

# 8.5. 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;
  }

  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

Typická metoda s proměnným počtem parametrů je `System.out.format()` [skr-36]

# Kapitola 9. Automatizované testování

# 9.1. Základní pojmy

- testy **psané** programátory vlastního kódu ve stejném jazyce, jako je zdrojový kód
  - nikoliv automaticky generované
  - „automatizované testování“ znamená, že testy jsou automatizovaně spouštěné a vyhodnocované
  - též „programátorské testy“
- kromě programátorských testů se používají ještě (podrobnosti viz v KIV/ZSWI)
  - testy uživatelského rozhraní
    - testuje funkčnost z hlediska uživatele
    - nezávislé na použitém programovacím jazyce a kódu programu
  - akceptační testy – testy pro kontrolu, že program vyhovuje dané specifikaci
  - zátěžové testy
  - testy bezpečnosti, atd.
- testování je celé rozsáhlé odvětví SW průmyslu
- existuje „vývoj programů řízený testy“ (*Test-Driven Development* – TDD)
  - hodně speciální způsob vývoje programů – kniha Beck, K.: **Programování řízené testy**, Grada
    - princip je, že nejdříve napíšeme jednoduchý test, který má budoucí kód splnit a až potom píšeme tento kód
      - kód pak upravujeme tak dlouho, dokud nevyhoví testu
      - pak zesložitíme test nebo napíšeme další test
      - tento cyklus se opakuje tak dlouho, dokud nejsme srozsahem schopností testované třídy spokojeni
      - pak píšeme jednoduchý test pro další třídu
  - TDD závisí na použití xUnit testů, ale naopak to neplatí
    - xUnit testy lze použít nezávisle na způsobu, jakým byl program vytvořen
- vždy je třeba mít na paměti, že jakékoliv úspěšné testy ještě nedokazují absolutní správnost programu

### Poznámka

Dále se budeme zabývat jen programátorskými testy, tj. „test“ = „programátorský test“

# 9.1.1. Psychologické aspekty testů

## 9.1.1.1. Pseudozdůvodnění pro nepoužívání testů

- testy psané programátory zdržují, tzn. prodražují vývoj
- přitom jsou zbytečné, protože
  - naši programátoři chyby nedělají (manažerský pohled)
    - ♦ nesmysl, chyby dělá každý
  - máme testery, kteří na chybu přijdou
    - ♦ ale testeři testují na zcela jiné úrovni
      - – v naprosté většině funkčnost uživatelského rozhraní
      - – tester má malé (nebo žádné) možnosti otestovat jiné vrstvy kódu (datovou, aplikační)
      - – i kdyby měl, musel by opakovaně ručně testovat to, co lze otestovat automaticky

## 9.1.1.2. Proč psát automatické testy

- důvěra v kód, která se dá kdykoliv **automaticky** ověřit – srovnej dále *Jiné způsoby testování*
  - testy jsou kdykoliv jednoznačně opakovatelné
- bezpečný refaktoring kódu
  - refaktoring je „uklízení“ zdrojového kódu bez změny dosavadní funkcionality
    - ♦ nejedná se o vylepšování funkčnosti kódu, ale vylepšení „krásy“ (přehlednosti) zdrojového kódu
  - při změnách vidíme, že jiným způsobem napsané části kódu neporušily původní funkčnost
- budoucí změny – opravy chyb, rozšiřování funkčnosti
  - pak víme, že změna (tj. vylepšení) nezhoršila původní funkčnost
- testy jsou nezbytné při vývoji v týmu, kdy jeden závisí na druhém
  - pak přijme jeho část díla, až když vyhoví testům
- nutí k lepšímu návrhu kódu (využívání rozhraní, krátké jednoúčelové metody), protože takový kód se lépe testuje

## 9.1.1.3. Jiné způsoby testování

- testovat lze pomocí debuggeru nebo pomocných výpisů na obrazovku
- nevýhody:
  - u výpisů na obrazovku – testovací kód je součástí zdrojového kódu

- u debuggeru – potřebujeme RAD a breakpointery nastavujeme ručně

- základní společná nevýhoda – výsledky testů se musí vyhodnocovat ručně
  - vyhodnocování nelze zautomatizovat

### 9.1.1.4. Testování je opačný pohled na programování

- při psaní testu máme opačný cíl než při psaní programu
  - snažíme se program „poškodit“, místo abychom se ho snažili „zprovoznit“
    - psychologický problém – programátor často podvědomě vynechává testy oblastí, o kterých tuší, že by mohly způsobit problémy

## 9.1.2. Druhy testů

- testů je více druhů a vzájemně se doplňují a rozšiřují

1. **JUnit testy** (obecně xUnit)
   - nejnižší úroveň, která má dvě podúrovně
     a. testy izolovaných částí víceméně bez vzájemných souvislostí – JUnit
     b. testy jednoduchých souvislostí – JUnit + *mock* objekty
2. **lehké integrační testy**
   - test hierarchického volání komponent
   - testuje se provázanost vrstev systému
     - přístup do datové vrstvy (nejčastěji DB) se simuluje, případně se DB nahradí vestavěnou („paměťovou“) DB (např. HSQLDB)
   - JUnit + SpringTestCase
3. **těžké integrační testy**
   - test komponent v reálném prostředí, kde bude systém nasazen
     - nad reálnou DB
   - JUnit + Cactus

**Poznámka**

Je vidět, že JUnit je vždy základ.

## 9.1.3. Další pojmy

### 9.1.3.1. Pokrytí kódu (*code coverage*)

- existují pomocné nástroje, které automaticky zjistí, jaká část zdrojového kódu je pokryta testy

- není dobré tuto možnost přehánět (udělat z ní základní cíl)
  - pak to dopadá tím způsobem, že je snaha o 100% pokrytí na úkor např. důkladných testů okrajových podmínek
  - důvod, proč se 100% pokrytí stává základním cílem, je jednoduchý
    - pokrytí lze lehce automaticky ověřit na rozdíl od skutečně významového testování

## 9.1.3.2. Legacy code

- v současné době to znamená „kód bez testů“
  - dříve skutečně ve významu „zděděný kód“, který již není dále podporovaný (např. MS-DOS)
- zkušenost – zapojit testy do již existujícího projektu bez testů (tj. do *legacy code*) je problematické
  - je to ale nutné udělat, pokud chceme projekt rozšiřovat či upravovat – viz výše
    - u většího SW je dodání testů práce na několik člověkoměsíců
  - určitě průběžně psát testy pro všechny nové věci, které se do projektu přidávají
  - při psaní kódu tedy **průběžně**
    - píšeme Javadoc komentáře
    - ověřujeme kvalitu kódu pomocí PMD (nebo podobných nástrojů)
    - píšeme JUnit testy

## 9.1.3.3. Dělení Java tříd podle účelu

- **entitní třídy** (též **objekty** nebo **datové třídy**)
  - hodně dat (atributů) + getry a setry – typicky hodnotové datové typy a velká část odkazových datových typů
  - `toString()`, `equals()`, `hashCode()`, apod.
  - málo nebo žádná aplikační logika
  - z hlediska logiky celého programu **nezávislé na dalších** (nic nevolají)
    - pokud jsou složeny z jiných tříd, pak se jedná o tentýž typ tříd (`Osoba` je složena z `Elipsa` a `Obdelnik`)
- **doménové třídy** (občas též **komponenty**)
  - málo nebo žádná data (tzn. bez getrů a setrů)
    - pro data si vytvářejí objekty entitních tříd
  - výrazná převaha aplikační logiky
    - závislé na dalších doménových třídách

        - hierarchické volání ve stromové struktuře (tj. nadřízený podřízeného)
    - často realizované jako návrhový vzor Jedináček

### 9.1.3.4. Kontrakt

- vzájemná domluvená souvislost mezi určitými částmi programu
    - tato domluva nemusí být na první pohled zjevná
        - kontraktem je například i vztah mezi `equals()` a `hashCode()`
- kontraktem je také míněn i algoritmus, který zajišťuje jen jedna metoda (kontrakt metody)

### 9.1.3.5. Testovací případ (*test case*)

- jeden dále nedělitelný test
- celý test se sestává z mnoha (desítek, stovek, tisíců) testovacích případů
    - např. DU-08 obsahuje celkem 39 testovacích případů

# 9.2. JUnit testy

- užitečné odkazy
    - `www.junit.org`
    - `junit.sourceforge.net/doc/testinfected/testing.htm`
    - `junit.sourceforge.net/doc/cookstour/cookstour.htm` – starší verze JUnit, ale užitečné principy
    - `junit.sourceforge.net/doc/faq/faq.htm`
    - `www.exubero.com/junit/antipatterns.html` – velmi přehledně, co a jak nedělat
    - `www-128.ibm.com/developerworks/opensource/library/os-junit/?ca=dgr-lnxw07JUnite` – další „antivzory“

## 9.2.1. Úvodní informace

- základ je knihovna xUnit
    - tvůrce Kent Beck nejprve pro Smalltalk
    - existuje ve variantách pro různé programovací jazyky
        - pro Javu existuje JUnit
            - tvůrci Kent Beck a Erich Gamma
- JUnit je velmi rozšířený způsob psaní a spouštění testů

- de facto průmyslový standard
- v současné době je součástí většiny RAD (Eclipse, ...), které poskytují testům všemožnou podporu
  - generování kostry testů, grafická nadstavba, hromadné spouštění, atd.

- v současnosti (září 2010) verze 4.8
  - verze 4.x má pouze textový spouštěč (*TestRunner*)
    - grafický byl do verze 3.x
    - v současné době zajišťují RAD mnohem komfortnější grafickou nadstavbu
- verze 3.x, která je ve starších testech stále ještě vidět, má stejné filosofické principy použití, ale jinou praktickou realizaci
  - 3.x využívá dědění, zatímco verze 4.x používá anotace zavedené do Javy od JDK 1.5

**Poznámka**

V dalších příkladech se bude používat verze 4.7.

# 9.2.2. Základní informace o použití

- testování funkčnosti malých jednotek kódu
  - typicky entitní třídy a metody v nich
  - komponentové třídy lze testovat v omezeném rozsahu
    - pro větší rozsah testování se používají nadstavby nad JUnit – viz dále *mock* objekty
  - testy mají být co nejjednodušší, aby jeden test vždy pokrýval jen jednu testovanou oblast
- důvod tohoto způsobu testování
  - ujistit se, že jednotlivé základní stavební kameny „skládanky“ fungují, jak mají
- testují se `public` a `protected` metody
  - nikoliv `private` – není k nim přístupové právo
- ověřujeme základní kontrakty tříd nebo metod – podrobně viz dále

# 9.2.3. Princip použití

- použití bude ukazováno na příkladech

## 9.2.3.1. Ukázka jednoduché entitní třídy pro testování

- třída `HodnoceniPredmetu` slouží pro záznam známky z předmětu
  - je napsaná velmi zjednodušeně

- lze vytvořit objekt třídy pouze s názvem předmětu
  - pak je známka nastavena na hodnotu `DOSUD_NEHODNOCENO`
- metoda `setZnamka(int znamka)` testuje, zda je nastavovaná známka v povoleném rozsahu
  - není-li, je vyhozena *run-time* výjimka `IllegalArgumentException`
- metoda `isNehodnoceno()` vrací `true` v případě, kdy předmět ještě nebyl hodnocen

```
/**
 * Třída pro hodnocení předmětu
 */
public class HodnoceniPredmetu {
  private String nazev;
  private int znamka;

  // konstanty pro známkování
  final public static int VYBORNE = 1;
  final public static int NEDOSTATECNE = 5;
  final public static int DOSUD_NEHODNOCENO = 0;
  final public static String DOSUD_NEHODNOCENO_SLOVY = "N/A";

  public HodnoceniPredmetu(String nazev, int znamka) {
    this.nazev = nazev;
    setZnamka(znamka);
  }

  public HodnoceniPredmetu(String nazev) {
    this.nazev = nazev;
    this.znamka = DOSUD_NEHODNOCENO;
  }

  public String getNazev() {
    return nazev;
  }

  public int getZnamka() {
    return znamka;
  }

  /**
   * Nastavuje známku v rozsahu od VYBORNE do NEDOSTATECNE
   * @param znamka nastavovaná známka
   * @throws IllegalArgumentException pokud je známka mimo rozsah
   */
  public void setZnamka(int znamka) throws IllegalArgumentException {
    if (znamka >= VYBORNE  &&  znamka <= NEDOSTATECNE) {
      this.znamka = znamka;
    }
    else {
      throw new IllegalArgumentException("" + znamka);
    }
  }
```

```
  /**
   * Zjistí, zda je předmět dosud nehodnocen
   * @return true, pokud je předmět nehodnocen
   */
  public boolean isNehodnoceno() {
    return (znamka == DOSUD_NEHODNOCENO);
  }

  @Override
  public String toString() {
    if (isNehodnoceno() == true) {
      return nazev + ": " + DOSUD_NEHODNOCENO_SLOVY + "\n";
    }
    else {
      return nazev + ": " + znamka + "\n";
    }
  }

}
```

**Poznámka**

V tomto okamžiku bychom pravděpodobně začali psát metodu `main()` někde umístěnou, abychom ověřili, zda je třída `HodnoceniPredmetu` správně. Tato metoda `main()` by byla jednoúčelová. Místo ní můžeme začít psát regulérní testovací případy, které poslouží lépe a navíc budou moci být trvalým doplňkem této třídy.

## 9.2.3.2. Umístění JUnit

- JUnit se stáhne jako jeden `.jar` soubor
  - zde `junit-4.7.jar`, který je umístěn např. na: `C:\Program Files\Java\junit\`
  - tento soubor musí být na *ClassPath* (viz dále)

## 9.2.3.3. První testovací případ

- testovací případy jsou typicky uloženy ve třídě, která se jmenuje *`TestovanáTřída`*`Test` (zde `HodnoceniPredmetuTest`)

```
import org.junit.Test;
import static org.junit.Assert.*;

public class HodnoceniPredmetuTest {

  @Test
  public void testSetZnamkaFunkcnost() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    assertEquals(1, predmet.getZnamka());
  }
}
```

### Poznámka

Zdrojové kódy JUnit testů často obsahují „magická čísla“ (zde `1`), což není považováno za nedokonalost kódu.

- význam jednotlivých částí kódu:
  - `import org.junit.Test;`
    - umožní používat anotaci `@Test` – viz dále
  - `import static org.junit.Assert.*;`
    - tzv. „statický import“, který dovoluje používat statické metody třídy bez úvodního jména třídy, tedy
      - `assertEquals()` místo `Assert.assertEquals()`
  - `@Test`
    - anotace, která způsobí, že následující metoda může být zařazena jako testovací případ a pomocí JUnit spouštěna
  - `public void testSetZnamkaFunkcnost()`
    - hlavička metody, která představuje jeden testovací případ
    - musí být typu `public void`
      - často se používá i `public final void`
    - název metody je odvozen
      - typicky začíná `test`
        - je to dobrý zvyk, který zůstal ze starších verzí JUnit
      - pokračuje názvem testované metody `SetZnamka`
      - často končí pojmenováním toho, co se testuje, případně pořadovým číslem testovacího případu (viz dále)
        - je běžné, že jedna metoda je testována více testovacími případy
        - v tomto uvedeném příkladu se testuje obecná funkčnost
        - další testy mohou být např. na horní a dolní meze – viz dále
    - metoda musí být bez formálních parametrů
    - v těle metody by měla být alespoň jedna – typicky **právě jedna** – metoda třídy `Assert`
      - toto pravidlo se občas rozumně porušuje, dále uvidíme test bez *assert* metody nebo více *assert* metod v jednom testovacím případě
  - `HodnoceniPredmetu predmet = new...`
    - vytvoření objektu testované třídy – běžný Java kód

- `predmet.setZnamka(1);`
  - nastavení známky – běžný Java kód
- `assertEquals(1, predmet.getZnamka());`
  - vlastní test s využitím metody `assertEquals()` třídy `Assert` z JUnit
  - první skutečný parametr je **očekávaná** hodnota
  - druhý skutečný parametr je **aktuální** hodnota objektu `predmet`
  - pokud se budou hodnoty obou parametrů rovnat, tento testovací případ proběhne úspěšně
  - testy píšeme „pozitivně“, tj. tak, jak očekáváme, že bude program správně fungovat

## 9.2.3.4. Spuštění testů z příkazové řádky

### Poznámka

Metoda označená anotací `@Test` je spuštěna a v případě, že nevyhoví (metoda `assertXY()` hlásí problém), je vyhozena výjimka. Běh bez výjimek znamená, že testy prošly správně.

- soubory `HodnoceniPredmetu.java` a `HodnoceniPredmetuTest.java` jsou zcela dostačující pro spuštění JUnit testu, ačkoliv neobsahují metodu `main()`
  - metoda `main()` je uložena ve třídě `org.junit.runner.JUnitCore`, která je součástí souboru `junit-4.7.jar`
- překlad obou souborů se provede příkazem:

```
javac -cp "C:\Program Files\Java\junit\junit-4.7.jar";. *.java
```

- spuštění testů se provede příkazem (vše na jedné řádce):

```
java -cp "C:\Program Files\Java\junit\junit-4.7.jar";.
     org.junit.runner.JUnitCore HodnoceniPredmetuTest
```

- při úspěšném průběhu testu se vypíše např.:

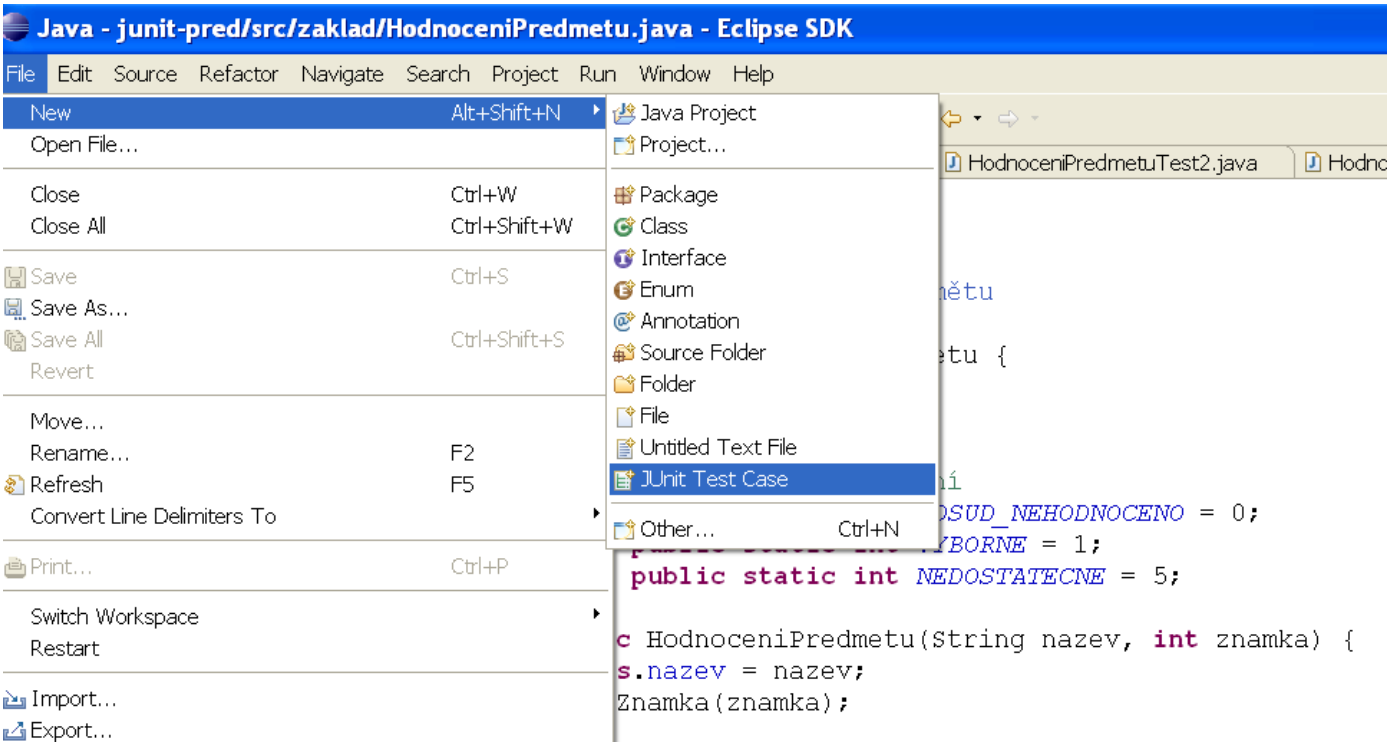
```
JUnit version 4.7
.
Time: 0,02
OK (1 test)
```

## 9.2.3.5. Použití testů z Eclipse

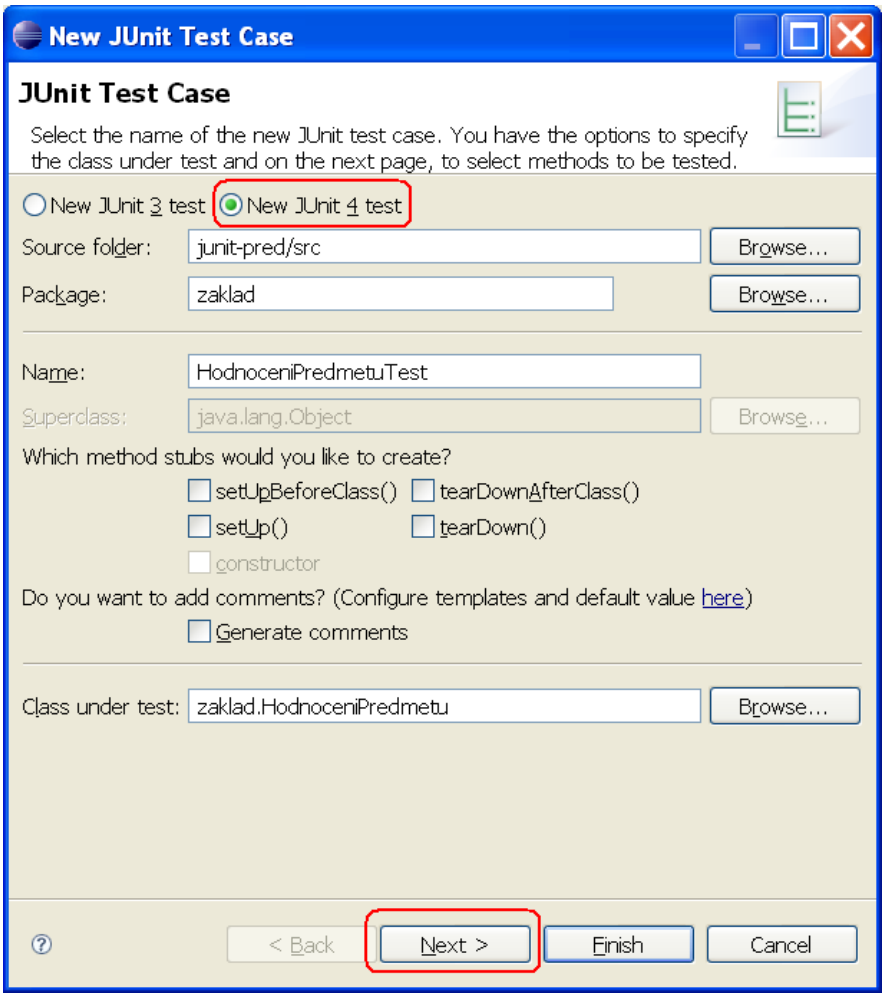
- příprava souboru testovacích případů a spouštění testů je v Eclipse (a jiných RAD) maximálně zautomatizováno

1. v editoru vybereme soubor se třídou, pro kterou chceme vytvářet testovací případy (zde `HodnoceniPredmetu.java`)

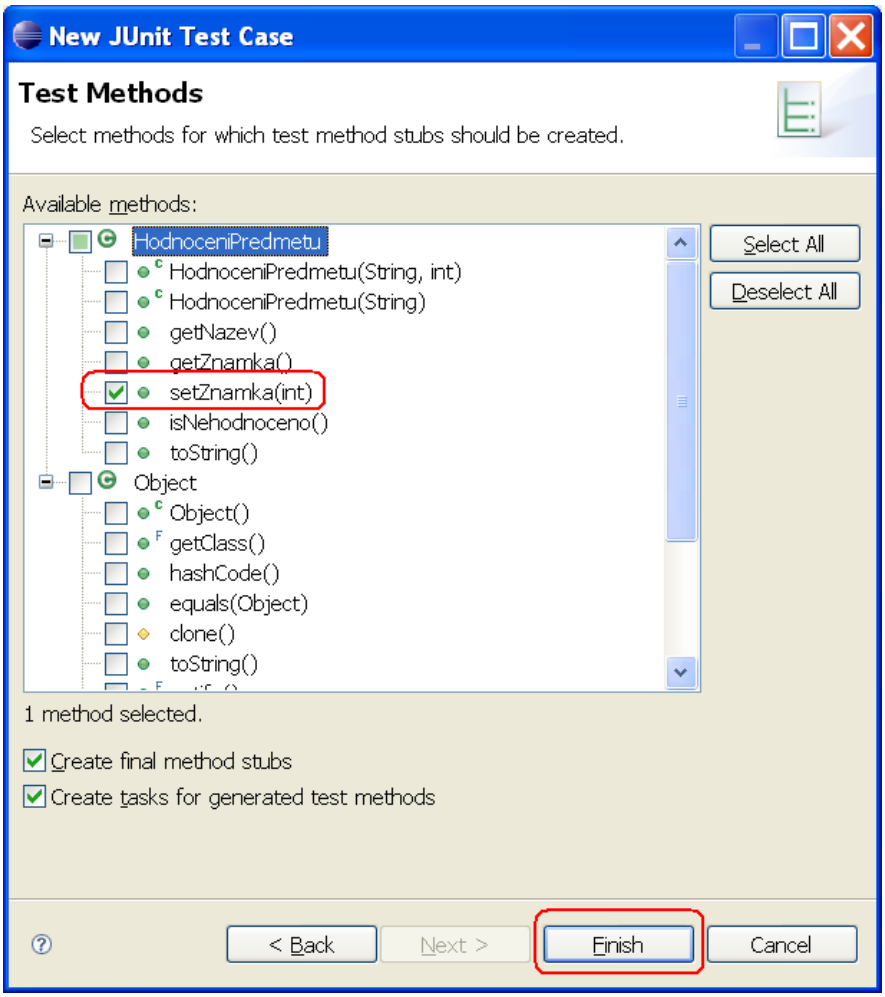
2. Z menu zvolíme *File / New / JUnit Test Case*




3. V dialogovém okně zvolíme přepínač *New JUnit 4 test* a pak *Next >*




4. V dialogovém okně zaškrtneme metodu `setZnamka(int)`, pro kterou chceme vygenerovat testovací případ a stiskneme *Finish*



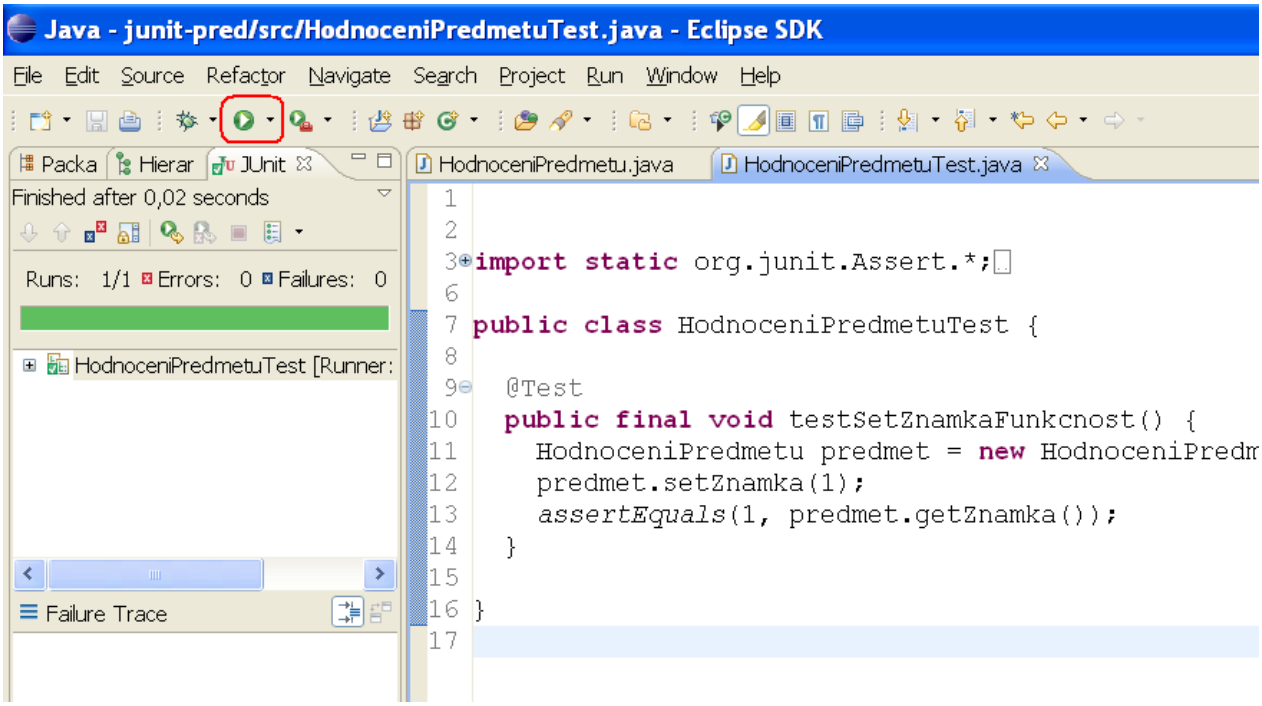

5. Eclipse vygeneruje nový soubor `HodnoceniPredmetuTest.java` s obsahem

```
import org.junit.Test;
import static org.junit.Assert.*;

public class HodnoceniPredmetuTest {

  @Test
  public final void testSetZnamka() {
    fail("Not yet implemented"); // TODO
  }

}
```

6. V tomto souboru doplníme název metody na `testSetZnamkaFunkcnost()` a pak i tělo metody na (viz též výše)

```
  @Test
  public void testSetZnamkaFunkcnost() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    assertEquals(1, predmet.getZnamka());
  }
```

7. Spustíme běh programu zelenou ikonou běhu a vlevo se objeví zelený ukazatel průběhu, který znamená, že testy proběhly úspěšně




- soubor testovacích případů se tak dá připravit pro každou třídu projektu

## 9.2.3.6. JUnit test odhalil chybu

- jedná se o neúspěšný průběh testu
  - platí zásada, že pokud je v celém testu alespoň jeden testovací případ chybně (zde v ukázce je zatím pouze jeden), je celý test chybně
  - chybný testovací případ je ohlášen vyhozením výjimky `java.lang.AssertionError`
- například pro metodu, ve které je změněna očekávaná hodnota na `2`

```
@Test
public void testSetZnamkaFunkčnost() {
  HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
  predmet.setZnamka(1);
  assertEquals(2, predmet.getZnamka());
}
```

- při spuštění z příkazové řádky se vypíše:

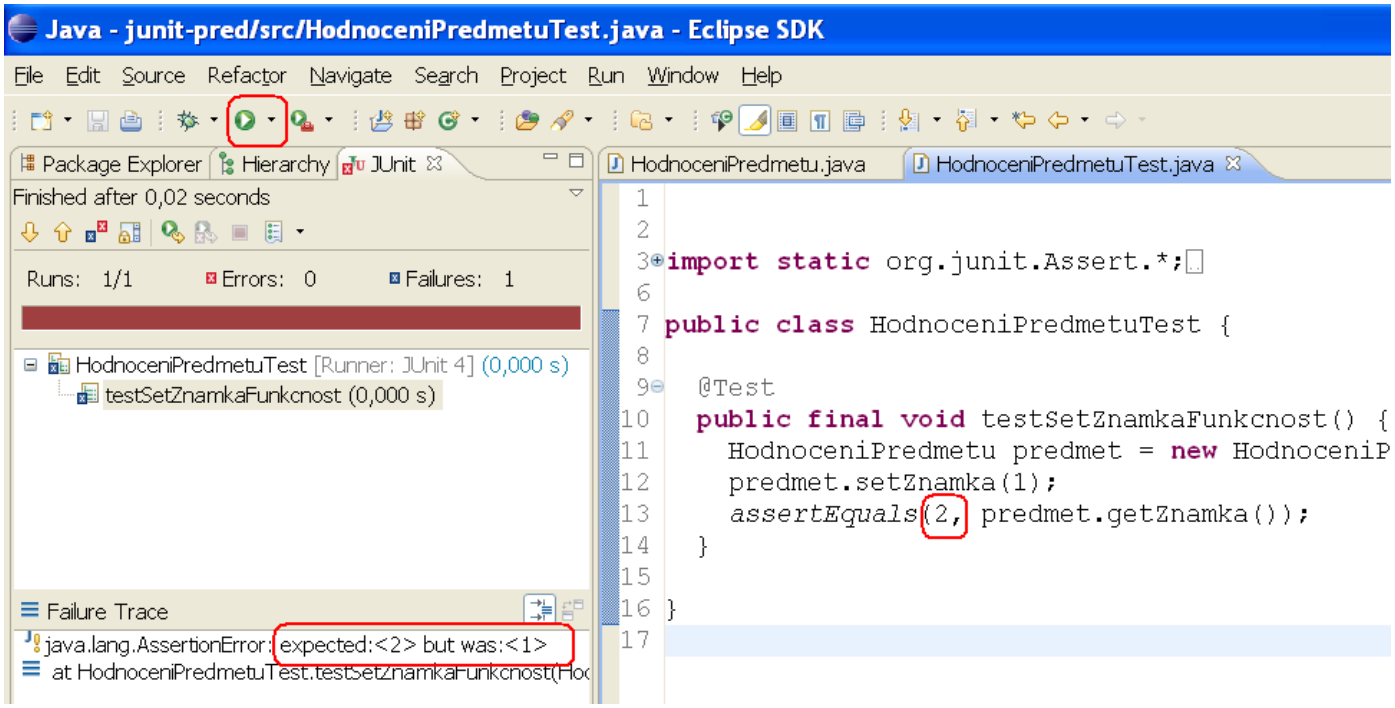
```
JUnit version 4.7
.E
Time: 0
There was 1 failure:
1) testSetZnamkaFunkčnost(HodnoceniPredmetuTest)
java.lang.AssertionError: expected:<2> but was:<1>
        at org.junit.Assert.fail(Assert.java:91)
        ...
```

- zde je podstatná informace:

```
java.lang.AssertionError: expected:<2> but was:<1>
```

  ze které vidíme, že test selhal proto, že očekávaná hodnota `2` se liší od poskytnuté hodnoty `1`

- při spuštění s Eclipse dostaneme červený průběh testu a vlevo dole opět vidíme důvod selhání testu




# 9.2.4. Pokročilé možnosti při psaní testovacích případů

- JUnit nám dává široké možnosti při vytváření testovacích případů
  - máme k dispozici tři základní sady možností:
    1. nastavení anotace `@Test`
    2. použití různých `assertXY()` metod ze třídy `org.junit.Assert`
    3. definování akcí, které mají běžet před spuštěním celého testu a i jednotlivých testovacích případů a po jejich ukončení
- je zcela běžné, že testovací třída obsahuje více testovacích případů – podrobně viz dále

## 9.2.4.1. Parametry anotace `@Test`

- základním úkolem anotace `@Test` je informovat JUnit o existenci testovacího příkladu
- ve většině testovacích případů je `@Test` použita v základním tvaru
- pro speciální případy je možnost `@Test` modifikovat dvěma dodatečnými volitelnými parametry:
  1. **testování očekávaných výjimek**

```
@Test(expected=jméno_výjimky.class)
```

- test projde, pokud je během něho vyhozena tato konkrétní výjimka (případně její rodič)
  - neprojde v ostatních případech, tj. vyhození jiné výjimky nebo nevyhození výjimky vůbec
- používá se pro testování metod, které pracují s výjimkami, např. metoda `setZnamka()`

```
/**
 * Nastavuje známku v rozsahu od VYBORNE do NEDOSTATECNE
 * @param znamka nastavovaná známka
 * @throws IllegalArgumentException pokud je známka mimo rozsah
 */
public void setZnamka(int znamka) throws IllegalArgumentException {
  if (znamka >= VYBORNE  &&  znamka <= NEDOSTATECNE) {
    this.znamka = znamka;
  }
  else {
    throw new IllegalArgumentException("" + znamka);
  }
}
```

- v příkladu jsou uvedeny čtyři testovací případy:

### Poznámka

V případu 3. a 4. je porušeno pravidlo, že testujeme „pozitivně“. Tyto testovací případy jsou napsány tak, aby bylo možné ověřit, jak vypadá selhání testu. Proto jejich název končí na `Chyba`. Případy 1. a 2. z tohoto důvodu končí na `OK`, byť je to zbytečné (pozitivní testování) a běžně toto označení nepoužíváme.

- **// 1.** `testSetZnamkaVyjimkaOK()`
  - testuje přímo výjimku `IllegalArgumentException`, která je metodou vyhazována při nesprávné hodnotě parametru `znamka` (zde hodnota `8`)
    - test projde
- **// 2.** `testSetZnamkaVyjimkaRodicOK()`
  - testuje rodičovskou výjimku `RuntimeException`, která je metodou vyhazována při nesprávné hodnotě parametru `znamka` (zde hodnota `8`)
    - test projde
- **// 3.** `testSetZnamkaVyjimkaChyba()`
  - testuje přímo výjimku `IllegalArgumentException`, která ale zde není metodou vyhazována, protože hodnota parametru `znamka` je správná (zde hodnota `1`)
    - test neprojde, je vypsáno:

```
java.lang.AssertionError:
Expected exception: java.lang.IllegalArgumentException
```

- **// 4.** `testSetZnamkaJinaVyjimkaChyba()`

– testuje úplně jinou výjimku `NullPointerException`, která ale není metodou vyhazována, ačkoliv je nesprávná hodnota parametru `znamka` (zde hodnota `8`)

• test neprojde, je vypsáno:

```
java.lang.Exception:
Unexpected exception, expected<java.lang.NullPointerException>
                    but was<java.lang.IllegalArgumentException>
```

```
import org.junit.Test;

public class HodnoceniPredmetuTestVyjimka {

// 1.
  @Test(expected=IllegalArgumentException.class)
  public final void testSetZnamkaVyjimkaOK() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(8);
  }

// 2.
  @Test(expected=RuntimeException.class)
  public final void testSetZnamkaVyjimkaRodicOK() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(8);
  }

// 3.
  @Test(expected=IllegalArgumentException.class)
  public final void testSetZnamkaVyjimkaChyba() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
  }

// 4.
  @Test(expected=NullPointerException.class)
  public final void testSetZnamkaJinaVyjimkaChyba() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(8);
  }
}
```

### Poznámka

Všimněte si, že při použití parametru `expected` nepoužíváme v těle testovacího případu metody třídy `org.junit.Assert`.

2. **testování maximální povolené doby běhu programu**

```
@Test(timeout=délka_trvání_v_milisec)
```

• test projde, pokud testovací případ nemá delší dobu trvání, než je uvedené maximum

- používá se k testování metod, jejichž doba trvání může činit problémy, případně je podezření na nekonečný cyklus
- v příkladu jsou uvedeny tři testovací případy:
  - **// 1.** `testSetZnamkaDobaOK()`
    - testuje, zda úsek testovaného kódu proběhne rychleji než za 100 milisec
      - test projde
      - je to typická ukázka běžného testovacího případu s metodou `assertEquals()`, která má navíc dodatečnou „časovou pojistku“ proti eventuelní nadměrné délce trvání
  - **// 2.** `testSetZnamkaDobaChyba()`
    - testuje, zda v kódu není nekonečný cyklus
      - test neprojde, je vypsáno:

        ```
        java.lang.Exception: test timed out after 100 milliseconds
        ```

      - zde uvedené použití pro `while(true)` je naivní, v reálném případě by se použilo pro testování metody, kde jsou málo přehledné vnořené cykly

        ### Poznámka

        Všimněte si, že není nutné v těle testovacího případu použít metody třídy `org.junit.Assert`.
  - **// 3.** `testSetZnamkaDobaAssertChyba()`
    - ukázka, jak funguje kombinace metody `assertEquals()` s „časovou pojistkou“
      - test neprojde, je vypsáno:

        ```
        java.lang.AssertionError: expected:<2> but was:<1>
        ```

      - pokud by však bylo testováno `assertEquals(1, predmet.getZnamka());` test by neprošel, protože by vypršel čas, a bylo by vypsáno:

        ```
        java.lang.Exception: test timed out after 100 milliseconds
        ```

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class HodnoceniPredmetuTestDoba {

// 1.
  @Test(timeout=100)
  public final void testSetZnamkaDobaOK() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
```

```
    predmet.setZnamka(1);
    assertEquals(1, predmet.getZnamka());
  }

// 2.
  @Test(timeout=100)
  public final void testSetZnamkaDobaChyba() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    while (true) {
       predmet.setZnamka(1);
    }
  }

// 3.
  @Test(timeout=100)
  public final void testSetZnamkaDobaAssertChyba() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    while (true) {
       predmet.setZnamka(1);
       assertEquals(2, predmet.getZnamka());
    }
  }
}
```

## 9.2.4.2. Testovací metody z `org.junit.Assert`

- třída `org.junit.Assert` dává k dispozici množství *assert* metod
  - v případě selhání (tj. *assert* nevyhověl) generují výjimku `org.junit.AssertionError`
    - v případě úspěchu se neděje nic, což znamená, že testovací případ prošel úspěšně
- metody jsou přetížené pro primitivní datové typy `long` a `double` (ostatní primitivní datové typy se dají bez problémů použít díky implicitním konverzím), např. pro typ `int`:
  - `assertEquals(long očekávanáHodnota, long skutečnáHodnota)`
  - `assertEquals(1, predmet.getZnamka());`
- každá metoda má přetíženou verzi pro generování výjimky `AssertionError` se zadanou chybovou zprávou
  - do zprávy můžeme přidat dodatečnou informaci typu `String` – doporučuje se používat
    - pro některé *assert* metody je tato možnost velmi důležitá – viz dále příklad s `assertTrue()`
    - ale i pro ostatní *assert* metody je užitečná – viz následující příklad
  - informace o tom, proč aserce neprošla, je často vypsána i při použití *assert* metody bez zprávy

```
public class HodnoceniPredmetuTestZprava {

  @Test
```

```
  public void testSetZnamkaBezZpravy() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    assertEquals(2, predmet.getZnamka());
  }

  @Test
  public void testSetZnamkaSeZpravou() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    assertEquals("Vracena hodnota znamky je chybna: ",
                 2, predmet.getZnamka());
  }
}
```

vypíše:

```
1) testSetZnamkaBezZpravy(HodnoceniPredmetuTestZprava)
java.lang.AssertionError: expected:<2> but was:<1>

2) testSetZnamkaSeZpravou(HodnoceniPredmetuTestZprava)
java.lang.AssertionError:
   Vracena hodnota znamky je chybna: expected:<2> but was:<1>
```

- *assert* metody lze dělit do pěti základních typů:

1. **testy rovnosti**

```
assertEquals(datový_typ očekávanáHodnota, datový_typ skutečnáHodnota)
```

- základní a nejčastější typ *assert* metody
- rovnají-li se hodnoty, test prošel
- pro porovnání `double` hodnot mají verzi s tolerancí přesnosti

```
assertEquals(double očekávanáHodnota, double skutečnáHodnota,
             double epsilon)
```

### Poznámka

Metoda `assertEquals()` pro porovnání `double` bez přesnosti je *deprecated*.

příklad na vyhovující a nevyhovující přesnost

```
public class TestDouble {

  @Test
  public void presnostOK() {
    assertEquals(Math.PI, 3.14, 0.01);
  }

  @Test
  public void presnostChyba() {
```

```
    assertEquals(Math.PI, 3.14, 0.0001);
  }
}
```

pro druhý testovací případ vypíše

```
java.lang.AssertionError: expected:<3.141592653589793> but was:<3.14>
```

- testy rovnosti lze použít i pro porovnávání objektů
  - ♦ pak se prakticky testuje správnost metody `equals()`
  - ♦ tyto testy jsou jiné, než testy shodnosti referenčních proměnných `assertSame()` – viz dále
  - ♦ rovnají-li se hodnoty referenčních proměnných, tzn. jedná-li se o tentýž objekt, test prošel

2. **testy logické hodnoty**

```
assertFalse(boolean skutečnáLogickáHodnota)
assertTrue(boolean skutečnáLogickáHodnota)
```

- často jsou (nevhodně) nahrazovány předchozím testem

```
assertEquals(true/false, skutečnáLogickáHodnota)
```

- příklad na vyhovující a nevyhovující test logické hodnoty

```
public class HodnoceniPredmetuTestBoolean {

  @Test
  public final void testIsNehodnocenoOK() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    assertTrue(predmet.isNehodnoceno());
  }

  @Test
  public final void testIsNehodnocenoChyba() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    String zprava = "Predmet je hodnocen: " +
                    predmet.isNehodnoceno();
    assertTrue(zprava, predmet.isNehodnoceno());
  }
}
```

pro druhý testovací případ, kdy využijeme možnosti výpisu zprávy, vypíše

```
java.lang.AssertionError: Predmet je hodnocen: false
```

bez výpisu zprávy se vypíše jen málo konkrétní informace:

```
java.lang.AssertionError:
```

3. **testy obsahu referenčních proměnných**

```
assertNull(Object object)
assertNotNull(Object object)
```

- výhodné při komplikovaných vytvářeních objektů, kdy si nejsme jisti, zda je objekt vytvořen či nikoliv

4. **testy shodnosti referenčních proměnných**

```
assertSame(Object očekávaný, Object skutečný)
assertNotSame(Object očekávaný, Object skutečný)
```

- Pozor! nejedná se o test shodnosti objektů, tj. testy `assertEquals()` – viz dříve
- rovnají-li se hodnoty referenčních proměnných, tzn. jedná-li se o tentýž objekt, test prošel

  příklad na vyhovující a nevyhovující shodu

```
public class TestSame {

  @Test
  public void testSameOK() {
    Integer i1 = new Integer(5);
    Integer i2 = i1;
    assertSame(i1, i2);
  }

  @Test
  public void testSameChyba() {
    Integer i1 = new Integer(5);
    Integer i2 = new Integer(5);
    assertSame(i1, i2);
  }
}
```

  pro druhý testovací případ, kdy se jedná o rozdílné objekty se stejnou hodnotou, vypíše

```
java.lang.AssertionError: expected same:<5> was not:<5>
```

5. **testy shodnosti celých polí**

```
assertArrayEquals(datový_typ očekávanéPole, datový_typ skutečnéPole)
```

- lze použít pro pole primitivních datových typů i pole objektů
  - ♦ pro pole objektů se porovnávají hodnoty objektů, tj. správnost metody `equals()`
- rovnají-li se délky obou polí a současně i hodnoty všech prvků, test prošel

  příklad na vyhovující a nevyhovující testy polí

```
public class TestArray {

  @Test
  public void testpoleOK() {
```

```
    Integer[] pole1 = new Integer[5];
    Integer[] pole2 = new Integer[5];
    for (int i = 0; i < pole1.length; i++) {
      pole1[i] = new Integer(i + 1);
      pole2[i] = new Integer(i + 1);
    }
    assertArrayEquals(pole1, pole2);
  }

  @Test
  public void testpoleDelkaChyba() {
    int[] pole1 = new int[5];
    int[] pole2 = new int[6];
    for (int i = 0; i < pole1.length; i++) {
      pole1[i] = i + 1;
      pole2[i] = i + 1;
    }
    assertArrayEquals(pole1, pole2);
  }

  @Test
  public void testpoleHodnotaChyba() {
    int[] pole1 = new int[5];
    int[] pole2 = new int[5];
    for (int i = 0; i < pole1.length; i++) {
      pole1[i] = i + 1;
      pole2[i] = i;
    }
    assertArrayEquals(pole1, pole2);
  }
}
```

pro druhý testovací případ vypíše

```
java.lang.AssertionError:
    array lengths differed, expected.length=5 actual.length=6
```

pro třetí testovací případ vypíše

```
arrays first differed at element [0]; expected:<1> but was:<0>
```

## 9.2.4.3. Akce před a po spuštění testovacích případů

- pro testování jedné třídy nebo i jedné metody používáme běžně více testovacích případů
  - pak se často opakuje kód nastavení podmínek pro testovací případ, např.:

```
public class HodnoceniPredmetuTestOpakovane {

  @Test
  public void testSetZnamka() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
```

```
    assertEquals(1, predmet.getZnamka());
  }

  @Test
  public final void testIsNehodnoceno() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
    assertFalse(predmet.isNehodnoceno());
  }
}
```

kdy se v obou testovacích případech (dokonce i v testech rozdílných metod) opakují příkazy

```
HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
predmet.setZnamka(1);
```

- JUnit umožňuje toto opakované nastavování (*test fixture*) zjednodušit pomocí čtyř anotací

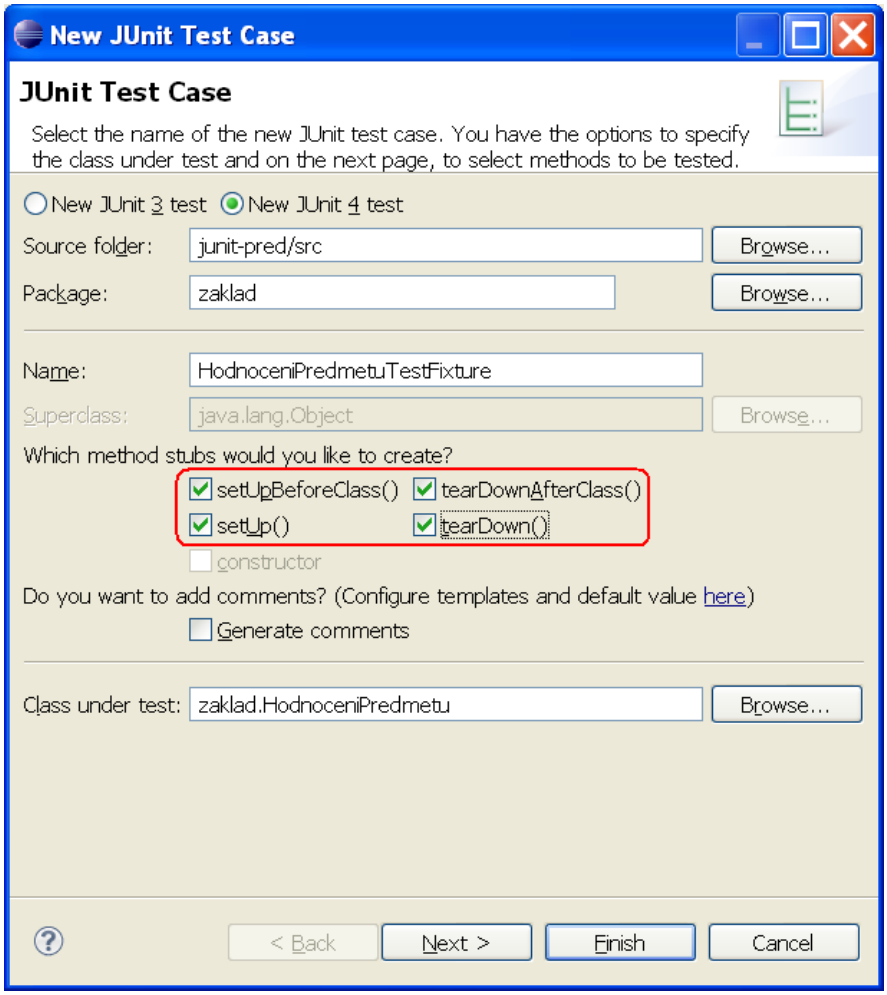
### Varování

Na jménech následujících metod by nemělo nezáležet. Prakticky ale např. Eclipse vyžaduje doporučená jména.

1. `@BeforeClass`
   - uvádí metodu typu `public static void`, typicky
     - `public static void setUpBeforeClass()`
   - provede se jednou před spuštěním všech testovacích případů této třídy
     - výhodné pro nastavení všech případných statických proměnných
2. `@AfterClass`
   - uvádí metodu typu `public static void`, typicky
     - `public static void tearDownAfterClass()`
   - provede se jednou po skončení všech testovacích případů této třídy
     - výhodné pro závěrečný „úklid“ nebo pro statistiku testu
3. `@Before`
   - uvádí metodu typu `public void`, typicky
     - `public void setUp()`
   - provede se před spuštěním každého testovacího případu
     - výhodné pro opakované nastavení podmínek jednoho testovacího případu
4. `@After`

- uvádí metodu typu `public void`, typicky
  - `public void tearDown()`
- provede se po ukončení každého testovacího případu
  - výhodné pro průběžný „úklid“

- při použití Eclipse lze kostry (*stubs*) těchto metod nechat automaticky vygenerovat (v dialogovém okénku se nepoužívají jména anotací)




- ukázka využití jednotného nastavování – příklad má stejnou funkčnost jako předchozí příklad

```
public class HodnoceniPredmetuTestFixture {
  HodnoceniPredmetu predmet;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
    System.out.println("Před všemi testovacími případy");
  }

  @AfterClass
  public static void tearDownAfterClass() throws Exception {
    System.out.println("Po všech testovacích případech");
  }

  @Before
```

```
  public void setUp() throws Exception {
    System.out.println("Před každým testovacím případem");
    predmet = new HodnoceniPredmetu("Matika");
    predmet.setZnamka(1);
  }

  @After
  public void tearDown() throws Exception {
    System.out.println("Po každém testovacím případu");
    predmet = null;
  }

  @Test
  public void testSetZnamkaOK() {
    System.out.println("1. testovací případ");
    assertEquals(1, predmet.getZnamka());
  }

  @Test
  public final void testIsNehodnocenoOK() {
    System.out.println("2. testovací případ");
    assertFalse(predmet.isNehodnoceno());
  }
}
```

testy proběhnou správně, na obrazovku se vypíše

```
Před všemi testovacími případy
Před každým testovacím případem
1. testovací případ
Po každém testovacím případu
Před každým testovacím případem
2. testovací případ
Po každém testovacím případu
Po všech testovacích případech
```

# 9.2.5. Dobré rady při vytváření testovacích případů

## 9.2.5.1. Obecné doporučení, co testovat

- ověřujeme základní kontrakty tříd nebo metod a v nich zejména
  - testy všech větví zpracování
  - testy speciálních případů
  - komplikované úseky kódu
  - okrajové případy
  - chybná vstupní data
  - vyhazované výjimky

- testují se `public` a `protected` metody
  - nikoliv `private` – není k nim přístupové právo
- navíc je vhodné převést na JUnit test:
  - cokoliv, co si kdykoliv zkusíte otestovat ručně, tzn. napíšete si pro to pomocnou metodu `main()`
  - všechny testovací a ladící výpisy
  - testy použité k verifikaci chyb a jejich oprav
    - pokud zákazník najde chybu, nejdříve napište JUnit test, který tuto chybu reprodukuje
      - až poté chybu najděte ve zdrojovém kódu a opravte
- naopak typicky netestujeme getry a setry – v případě, že jsou jednoduché
- v testovacích případech neodchytáváme výjimky
  - nechávají se být, aby shodily test
    - druhý testovací případ, kdy je výjimka odchycena, je chybně napsaný
      - test neselže (tj. proběhne „správně“), i když vznikne výjimka

```
public class TestNeocekavaneVyjimky {

  @Test
  public void testVyjimkyOK() {
    ArrayList<Integer> seznam = new ArrayList<Integer>();
    Integer i = seznam.get(0);
  }

@Test
  public void testVyjimkyChybne() {
    try {
      ArrayList<Integer> seznam = new ArrayList<Integer>();
      Integer i = seznam.get(0);
    } catch (Exception e) {
    }
  }
}
```

## 9.2.5.2. Organizace více testovacích případů

- je zcela běžné, že testovací třída (*test suite*) obsahuje více testovacích případů (*test case*)
  - testují se všechny potřebné (viz dále) metody třídy a konstruktory
  - navíc se každá metoda či konstruktor testují pomocí více testovacích případů, což je výhodné:
    - testovací metody mohou být dostatečně jednoduché
    - mohou se vždy zaměřit jen na jednu testovanou vlastnost

- dohromady jsou ale všechny testovací případy uloženy v jednom `.java` souboru
  - ♦ spuštění testů znamená spuštění všech testů dané třídy

■ lze mít víc tříd testů na jednu třídu kódu

- pozor ale, že je to často signál, že třída má víc různých zodpovědností
- pak existuje několik `.java` souborů testů, které je nutno postupně všechny spustit – viz dále

## 9.2.5.3. Ukázka testů entitní třídy

■ třída `HodnoceniPredmetu`, která je v balíku `skutecnytest`

```
package skutecnytest;

/**
 * Třída pro hodnocení předmětu
 */
public class HodnoceniPredmetu {
  private String nazev;
  private int znamka;

  // konstanty pro známkování
  final public static int VYBORNE = 1;
  final public static int NEDOSTATECNE = 5;
  final public static int DOSUD_NEHODNOCENO = 0;
  final public static String DOSUD_NEHODNOCENO_SLOVY = "N/A";

  public HodnoceniPredmetu(String nazev, int znamka) {
    this.nazev = nazev;
    setZnamka(znamka);
  }

  public HodnoceniPredmetu(String nazev) {
    this.nazev = nazev;
    this.znamka = DOSUD_NEHODNOCENO;
  }

  public String getNazev() {
    return nazev;
  }

  public int getZnamka() {
    return znamka;
  }

  /**
   * Nastavuje známku v rozsahu od VYBORNE do NEDOSTATECNE
   * @param znamka nastavovaná známka
   * @throws IllegalArgumentException pokud je známka mimo rozsah
   */
  public void setZnamka(int znamka) throws IllegalArgumentException {
    if (znamka >= VYBORNE  &&  znamka <= NEDOSTATECNE) {
```

```
      this.znamka = znamka;
    }
    else {
      throw new IllegalArgumentException("" + znamka);
    }
  }

  /**
   * Zjistí, zda je předmět dosud nehodnocen
   * @return true, pokud je předmět nehodnocen
   */
  public boolean isNehodnoceno() {
    return (znamka == DOSUD_NEHODNOCENO);
  }

  @Override
  public String toString() {
    if (isNehodnoceno() == true) {
      return nazev + ": " + DOSUD_NEHODNOCENO_SLOVY + "\n";
    }
    else {
      return nazev + ": " + znamka + "\n";
    }
  }

}
```

- může být otestována následujícími testovacími případy
  - jsou rozděleny do tří souborů (*test suite*), aby bylo později vidět, jak je možné spustit všechny tři najednou
    - zde rozdělení do více souborů nemá praktický smysl, protože testovacích případů je relativně málo
    - význam dělení do více souborů by byl nejspíše při potřebě mít různé metody `setUp()`
  - všechny testovací případy jsou pozitivní (projdou) – test neselže
- třída `HodnoceniPredmetuTestKonstruktor`
  - testuje správnou funkci konstruktorů
  - první testovací případ je víceméně zbytečný, protože se testuje pouze správnost přiřazovacího příkazu
  - další dva testy již testují kontrakt

```
package skutecnytest;

import static org.junit.Assert.*;
import org.junit.*;

public class HodnoceniPredmetuTestKonstruktor {

  @Test
```

```
  public final void testHodnoceniPredmetuNazev() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika", 5);
    assertEquals("Matika", predmet.getNazev());
  }

  @Test
  public final void testHodnoceniPredmetuZnamka() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika", 5);
    assertEquals(5, predmet.getZnamka());
  }

  @Test
  public final void testHodnoceniPredmetuZnamkaNevyplnena() {
    HodnoceniPredmetu predmet = new HodnoceniPredmetu("Matika");
    assertEquals(HodnoceniPredmetu.DOSUD_NEHODNOCENO, predmet.getZnamka());
  }
}
```

- třída `HodnoceniPredmetuTestZnamka`
  - testuje všechny možnosti metody `setZnamka()`

```
package skutecnytest;

import static org.junit.Assert.assertEquals;
import org.junit.*;

public class HodnoceniPredmetuTestZnamka {
  HodnoceniPredmetu predmet;

  @Before
  public void setUp() throws Exception {
    predmet = new HodnoceniPredmetu("Matika");
  }

  @Test
  public final void testSetZnamkaVMezich() {
    predmet.setZnamka(1);
    assertEquals(1, predmet.getZnamka());
    predmet.setZnamka(5);
    assertEquals(5, predmet.getZnamka());
  }

  @Test(expected=IllegalArgumentException.class)
  public final void testSetZnamkaMimoMezeDolni() {
    predmet.setZnamka(0);
  }

  @Test(expected=IllegalArgumentException.class)
  public final void testSetZnamkaMimoMezeHorni() {
    predmet.setZnamka(6);
  }
}
```

- třída `HodnoceniPredmetuTestMetody`
  - testuje metodu `isNehodnoceno()` v jednom testovacím případu oba stavy (`true`, `false`)
    - tento způsob není vhodný – smyslem JUnit testů je testovat co nejvíce jednoduchých příkladů, tj. jeden *assert* v jednom testovacím případu
  - testuje metodu `toString()` ve dvou nezávislých případech – vhodnější než předchozí přístup

```
package skutecnytest;

import static org.junit.Assert.*;
import org.junit.*;

public class HodnoceniPredmetuTestMetody {
  final static String NAZEV_PREDMETU = "Matika";
  HodnoceniPredmetu predmet;

  @Before
  public void setUp() throws Exception {
    predmet = new HodnoceniPredmetu(NAZEV_PREDMETU);
  }

  @Test
  public final void testIsNehodnoceno() {
    assertTrue(predmet.isNehodnoceno());
    predmet.setZnamka(HodnoceniPredmetu.VYBORNE);
    assertFalse(predmet.isNehodnoceno());
  }

  @Test
  public final void testToStringNehodnoceno() {
    String vysledek = NAZEV_PREDMETU + ": "
            + HodnoceniPredmetu.DOSUD_NEHODNOCENO_SLOVY + "\n";
    assertEquals(vysledek, predmet.toString());
  }

  @Test
  public final void testToStringHodnoceno() {
    predmet.setZnamka(HodnoceniPredmetu.NEDOSTATECNE);
    String vysledek = NAZEV_PREDMETU + ": "
            + HodnoceniPredmetu.NEDOSTATECNE + "\n";
    assertEquals(vysledek, predmet.toString());
  }
}
```

# 9.2.6. Praktické záležitosti

## 9.2.6.1. Kam se soubory testů fyzicky umísťují

- soubory lze umístit do téhož adresáře (tj. i balíku) jako testovanou třídu

```
src
   skutecnytest
      HodnoceniPredmetu.java
      HodnoceniPredmetuTestKonstruktor.java
      HodnoceniPredmetuTestMetody.java
      HodnoceniPredmetuTestZnamka.java
```

- je to jednoduchý způsob, který např. implicitně používá Eclipse
  - ♦ činí ale problémy při závěrečné distribuci programů bez JUnit testů

■ proto se soubory umísťují do jiného adresáře než testovaná třída, ale do stejného balíku

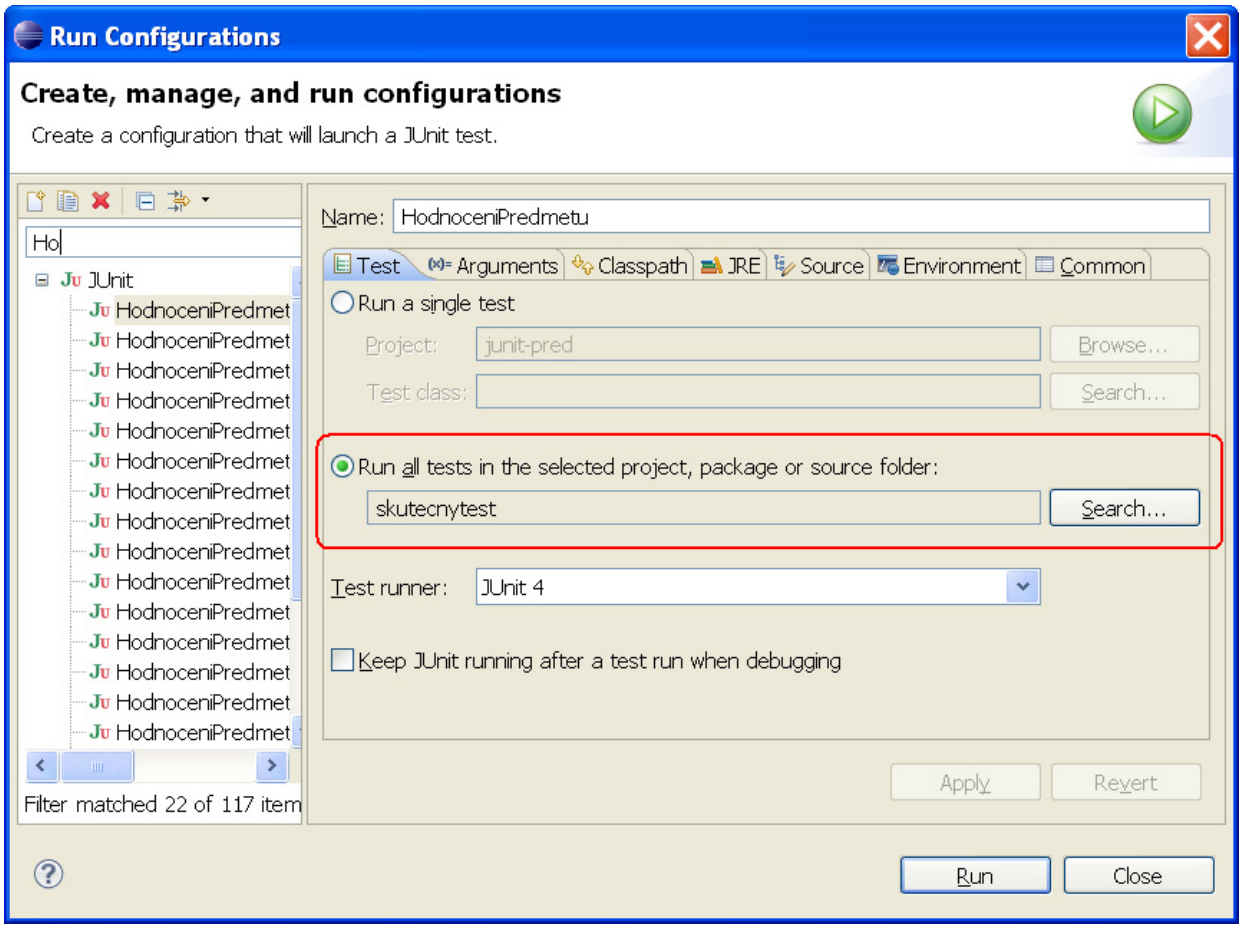
- v Eclipse příkaz **File / New / Folder**

```
src
   skutecnytest
      HodnoceniPredmetu.java

test
   skutecnytest
      HodnoceniPredmetuTestKonstruktor.java
      HodnoceniPredmetuTestMetody.java
      HodnoceniPredmetuTestZnamka.java
```

- je to o trochu více komplikovaný způsob, ale jen při počátečním nastavení

## 9.2.6.2. Jak se spouští najednou testy z více testovacích souborů

■ v Eclipse zvolíme *Run / Run Configuration* a v něm vybereme *Run all test*:




- z příkazové řádky musíme vyjmenovat jednotlivé třídy s testovacími případy (každý příkaz je jen na jedné řádce)

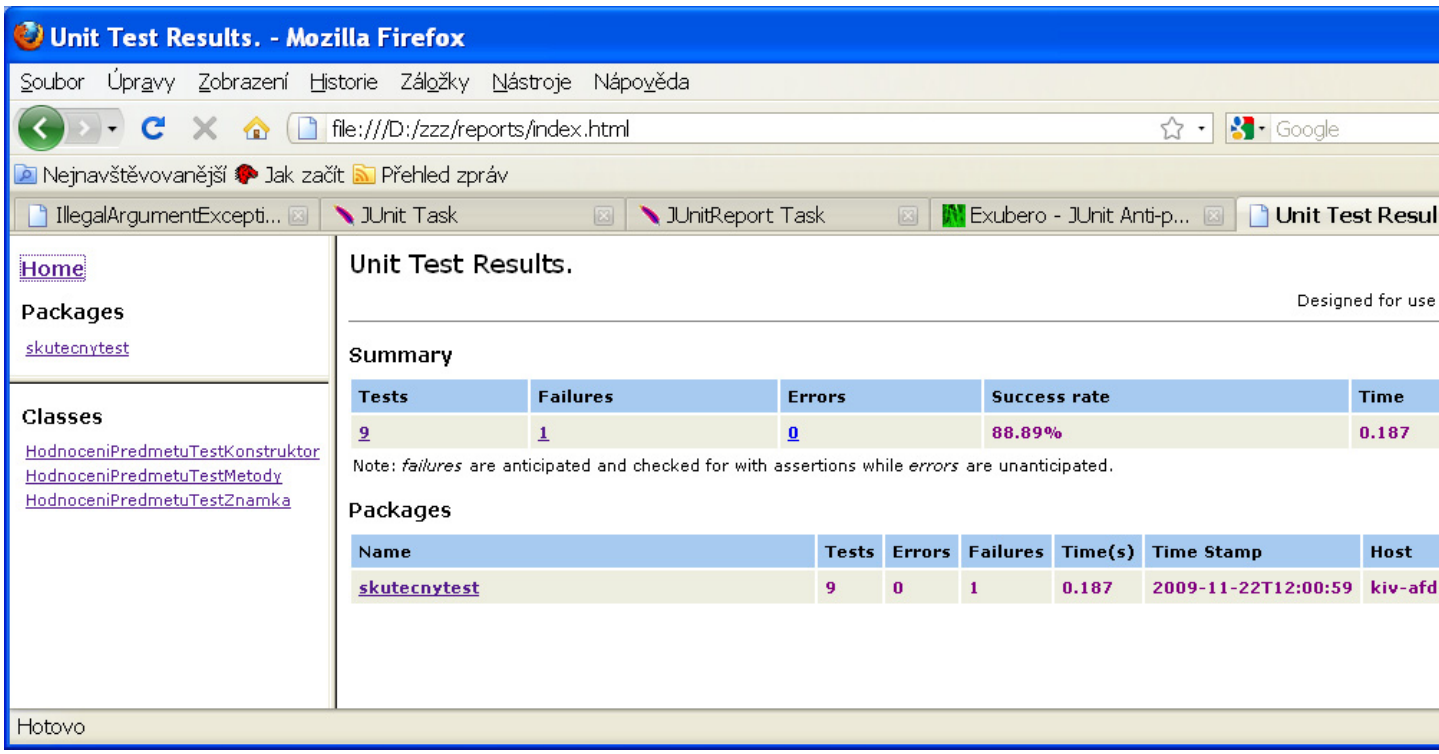
```
D:\zzz>javac -cp "C:\Program Files\Java\junit\junit-4.7.jar";.
       -d . skutecnytest/*.java

D:\zzz>java -cp "C:\Program Files\Java\junit\junit-4.7.jar";.
       org.junit.runner.JUnitCore
       skutecnytest.HodnoceniPredmetuTestKonstruktor
       skutecnytest.HodnoceniPredmetuTestZnamka
       skutecnytest.HodnoceniPredmetuTestMetody

JUnit version 4.7
.........
Time: 0,02

OK (9 tests)
```

## 9.2.6.3. Spouštění z Ant

- pokud vyvíjíme program v nějakém RAD (např. Eclipse), má význam provádět testy již během vývoje
- pro větší projekty, kdy je testů mnoho, je ale vhodné testy dále automatizovat
  - pro tuto činnost lze s úspěchem využít Ant a jeho dva tasky
    - `<junit>` – provádí vlastní testy

http://ant.apache.org/manual/OptionalTasks/junit.html

- `<junitreport>` – výsledky testů transformuje do přehledných HTML formulářů




- pro použití JUnit testů v Ant je třeba zajistit následující činnosti:

1. nastavení konstant a vlastností

```
<property name="test.adr" value="D:/zzz/" />
<property name="junit.path"
          value="C:/Program Files/Java/junit/junit-4.7.jar" />

<property name="test.lib" value="${test.adr}lib" />
<property name="test.class" value="${test.adr}class" />
<property name="test.reports" value="${test.adr}reports" />
```

2. nastavení ClassPath

   - `.class` soubory testovaného programu jsou zabaleny do JAR souboru v adresáři `${test.lib}` nebo jako `.class` soubory v adresáři `${test.class}` případně v jeho podadresářích, pokud je použit balík

```
<path id="test.classpath">
  <pathelement location="${test.class}" />
  <pathelement location="${junit.path}" />
  <fileset dir="${test.lib}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

3. spuštění tasku `<junit>`

- `.class` soubory JUnit testů jsou v adresáři `${test.class}` případně v jeho podadresářích, pokud je použit balík
- pokud jsou v adresáři `${test.class}` ještě další `.class` soubory, musí mít soubory JUnit testů ve svém názvu řetězec `Test`

```
<junit fork="yes" printsummary="yes" haltonfailure="no">
  <batchtest fork="yes" todir="${test.reports}">
    <fileset dir="${test.class}">
      <include name="**/*Test*.class" />
    </fileset>
  </batchtest>
  <formatter type="xml" usefile="true" />
  <classpath refid="test.classpath" />
</junit>
```

4. spuštění transformace výsledků do HTML

```
<junitreport todir="${test.reports}">
  <fileset dir="${test.reports}">
    <include name="TEST-*.xml" />
  </fileset>
  <report todir="${test.reports}" />
</junitreport>
```

- po dokončení transformace použijeme soubor `index.html` z adresáře `${test.reports}`

■ pokud nám stačí jen testy spustit a výsledek si prohlédnout na konzoli, bude mít `<junit>` tento obsah

```
<junit fork="yes" printsummary="no" haltonfailure="no">
  <batchtest fork="yes" todir="${test.reports}">
    <fileset dir="${test.class}">
      <include name="**/*Test*.class" />
    </fileset>
  </batchtest>
  <formatter type="plain" usefile="false" />
  <classpath refid="test.classpath" />
</junit>
```

výpis má pak tvar např.:

```
[junit] Testsuite: skutecnytest.HodnoceniPredmetuTestKonstruktor
[junit] Tests run: 3, Failures: 1, Errors: 0, Time elapsed: 0,02 sec
[junit]
[junit] Testcase: testHodnoceniPredmetuNazev took 0,02 sec
[junit] Testcase: testHodnoceniPredmetuZnamka took 0 sec
[junit]     FAILED
[junit] expected:<2> but was:<5>
   ...
```

## 9.2.6.4. Spouštění z Java programu

■ testy lze spustit z našeho programu

- tato možnost nemá zdánlivě žádný smysl, protože
  - při vývoji pomocí RAD spouštíme testy průběžně prostředky RAD a o výsledku jsme dostatečně dobře informováni
  - při rozsáhlejších testech využijeme možností Antu a jeho reportů, které jen těžko vizuálně vylepšíme
- přesto má smysl spouštět testy programově, např. v případě, že potřebujeme jednoduchý výsledek testu typu **prošel / neprošel**
  - to bychom využili v případě, kdy práci přebíráme a zajímá nás jen celková funkčnost, ne případná místa výskytů chyb

■ testy spouštíme pomocí metody `org.junit.runner.JUnitCore.runClasses()`

- předáme jí jako seznam skutečných parametrů názvy `.class` souborů jednotlivých testovacích tříd oddělených čárkami
- metoda vrací výsledek typu `org.junit.runner.Result`
  - API viz `http://junit.sourceforge.net/javadoc_40/org/junit/runner/Result.html`
  - nejdůležitější jsou metody:
    - `wasSuccessful()` – vrací `true` v případě, že všechny testovací případy proběhly úspěšně
    - `getFailureCount()` – vrací počet neúspěšných testovacích případů
    - `getFailures()` – vrací `List<Failure>` se záznamem jednotlivých neúspěšných testovacích případů, který lze využít pro podrobnější výpisy
      - to je ale většinou zbytečné, protokoly Antu poskytnou lepší službu

■ program se třídou `main()`, který spustí všechny tři výše uvedené testovací soubory

```
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class SpusteniTestu {

  public static void main(String[] args) throws AssertionError {
    Result result = org.junit.runner.JUnitCore.runClasses(
          HodnoceniPredmetuTestKonstruktor.class,
          HodnoceniPredmetuTestMetody.class,
          HodnoceniPredmetuTestZnamka.class);

    System.out.println("Uspesnost testu: " + result.wasSuccessful());

    if (result.wasSuccessful() == false) {
      System.out.println("Pocet chyb: " + result.getFailureCount());
      System.out.println("Seznam chyb: ");
      for (Failure fail: result.getFailures()) {
        System.out.print(fail.getDescription().getClassName() + ".");
        System.out.print(fail.getDescription().getMethodName() + ": ");
        System.out.println(fail.getMessage());
```

```
      }
    }
  }

}
```

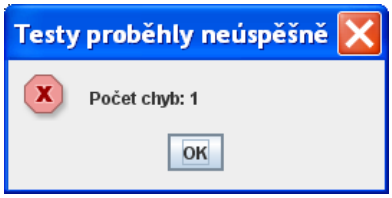
- po spuštění se vypíše např.:

```
Uspesnost testu: false
Pocet chyb: 1
Seznam chyb:
skutecnytest.HodnoceniPredmetuTestKonstruktor.
  testHodnoceniPredmetuZnamka: expected:<2> but was:<5>
```

- pokud ve výše uvedeném programu (soubor `SpusteniTestuGUI.java`) změníme část popisující výsledek testů na

```
  if (result.wasSuccessful() == false) {
    JOptionPane.showMessageDialog(
        null,
        "Počet chyb: " + result.getFailureCount(),
        "Testy proběhly neúspěšně",
        JOptionPane.ERROR_MESSAGE);
  }
```

- dostaneme např.:




## 9.2.6.5. Spouštění testů z `.jar`

- je-li možné spouštět testy programově – viz předchozí část, je možné je spustit i z `.jar` souboru
- to má tu výhodu, že v jednom `.jar` souboru může být kompletní, funkční a spustitelný uživatelský program
  - zde soubory `HodnoceniPredmetu.class` a `Hlavni.class` v souboru `Hlavni.jar`

```
package skutecnytest;

public class Hlavni {
  public static void main(String[] args) {
    HodnoceniPredmetu matika = new HodnoceniPredmetu("Matika");
    matika.setZnamka(2);
    System.out.println("Hlavni program");
    System.out.println(matika.toString());
```

```
    }
}
```

- spuštění z podadresáře `lib` bez jakýchkoliv testů

```
D:\zzz>java -jar .\lib\Hlavni.jar
```

- vypíše:

```
Hlavni program
Matika: 2
```

- ve druhém `.jar` souboru (zde `JUnitTest.jar`) jsou všechny testy včetně spouštěcího programu (zde `SpusteniTestu`) , které v případě potřeby dokážeme spustit
  - oba `.jar` soubory jsou uloženy v podadresáři `lib` a budou přidány do *ClassPath*
- pak je možné spustit testy z příkazové řádky bez rozbalování `.jar` souborů

```
D:\zzz>java -cp
  "C:\Program Files\Java\junit\junit-4.7.jar";
  .\lib\JUnitTest.jar;
  .\lib\Hlavni.jar;.
  skutecnytest.SpusteniTestu
```

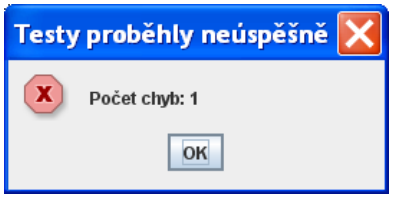
- vypíše:

```
Uspesnost testu: false
Pocet chyb: 1
Seznam chyb:
skutecnytest.HodnoceniPredmetuTestKonstruktor
.testHodnoceniPredmetuZnamka: expected:<2> but was:<5>
```

- respektive pro spouštěcí soubor `SpusteniTestuGUI.java`

```
D:\zzz>java -cp
  "C:\Program Files\Java\junit\junit-4.7.jar";
  .\lib\JUnitTest.jar;
  .\lib\Hlavni.jar;.
  skutecnytest.SpusteniTestuGUI
```

- dostaneme



# 9.3. Testování doménové třídy

- předchozí příklady ukázaly základní možnost použití JUnit testů
  - pro testování entitních tříd naprosto postačující
- dostaneme-li se ale o „stupeň výše“, k testování doménových tříd (komponent), je použití jen JUnit testů nedostatečné
  - doménové třídy se vyznačují mj. závislostmi – např. prezentační vrstva závisí na aplikační logice a ta závisí na datové vrstvě (DAO – *Data Access Object*)
- je třeba mít metodiku a nástroje na testování doménových tříd, které obsahují závislosti
  - jdou proti sobě dvě věci:
    1. díky vzájemným závislostem musíme mít stále komplikovanější testy zahrnující více doménových tříd najednou
    2. požadavek, aby testovací případ prověřil pokud možno jen jednu funkcionalitu kódu, aby se snadno odhalilo místo chyby
- tato metodika a nástroje naštěstí existují
  - je možné provádět testy izolovaných doménových tříd v podobném duchu, jako testy entitních tříd

## 9.3.1. Ukázka doménové třídy

- tato třída bude navazovat na předchozí entitní třídu `HodnoceniPredmetu`
  - poskytuje službu výpočtu průměru známek z daného předmětu – metoda `prumerZPredmetu()`
- třída je napsaná pro testovací účely nevhodně a postupem času ji budeme upravovat

```
import java.util.List;

public class StatistikaHodnoceniNevhodna {
  private SpravaDatHodnoceni spravaDat;

  public StatistikaHodnoceniNevhodna() {
    this.spravaDat = new SpravaDatHodnoceni();
  }

  /**
   * Vypočítá průměr známek z jednoho konkrétního předmětu
   * @param nazev název předmětu
   * @return průměr známek
   */
  public double prumerZPredmetu(String nazev) {
    List<HodnoceniPredmetu> vybrane;
    vybrane = spravaDat.getJedenPredmet(nazev);
    double suma = 0;
    for (HodnoceniPredmetu hod : vybrane) {
      suma += hod.getZnamka();
```

```
    }
    double prumer = suma / vybrane.size();
    return prumer;
  }
}
```

- třída `StatistikaHodnoceniNevhodna` potřebuje ke své činnosti objekt třídy `SpravaDatHodnoceni`
  - ten si vytváří v konstruktoru
- chceme-li testovat metodu `prumerZPredmetu()`, pak je problém v tom, že musíme mít **současně** k dispozici i třídu `SpravaDatHodnoceni` a pravděpodobně nějaký zdroj dat (soubor, databázi, apod.)
  - tím se poměrně jednoduchý test jedinné metody značně komplikuje („rozplizává“)
  - navíc nás v tomto okamžiku nezajímá třída `SpravaDatHodnoceni` a její metoda `getJedenPredmet()`
    - za její správnost ručí někdo jiný, který ji také testuje
    - tuto metodu prostě považujeme za správnou
    - třída nemusí být vůbec k dispozici
- problém ovšem je, jak pomocí příkazu `spravaDat.getJedenPredmet(nazev);` uvedeného v metodě `prumerZPredmetu()` dostat taková testovací data, která nyní potřebujeme pro testy činnosti této metody

# 9.3.2. Principy řešení problému „rozplizlého“ testování

- filosofickým řešením výše zmíněného problému je použití *mock* objektu (atrapa, náhrada, imitace, přeneseně též maketa)
  - pomocí něj nasimulovat existenci objektu třídy `SpravaDatHodnoceni` tak, aby vrátil předem připravená data
  - pak bychom skutečně testovali pouze jeden kontrakt a to správnou funkci metody `prumerZPredmetu()`
    - „odřízneme“ se zcela od třídy `SpravaDatHodnoceni`
    - při testech metody `prumerZPredmetu()` tedy nebudeme spouštět skutečnou metodu `getJedenPredmet()`, ale její náhražku reprezentovanou *mock* objektem
      - technologicky je to ale nutné zajistit tak, aby nebyly třeba žádné zásahy do zdrojového kódu metody `prumerZPredmetu()`

## 9.3.2.1. Implementace *mock* objektů

- existuje několik frameworků, které využívají *mock* objekty
- dělí se na dvě skupiny podle způsobu jejich implementace
  - využívající proxy – častější a populárnější

- ♦ proxy objekt je objekt používaný (dočasně) na místě skutečného objektu
  - – imituje skutečný objekt, na kterém náš kód závisí
  - – problém, jak a kde proxy objekt vytvořit, řeší framework
- využívající *class remapping* – framework **jmockit**

### 9.3.2.2. Návrhový vzor *Dependency injection* (DI)

- tento návrhový vzor se výborně hodí pro „rozvolnění závislostí“ mezi doménovými třídami – podrobnosti viz v předmětu KIV/ASWI
  - programové konstrukce, které naopak závislosti zvyšují, jsou typicky
    - ♦ jakékoliv `static` prvky
    - ♦ používání operátoru `new`
- pro náš účel zde využijeme techniky z DI nazvané *Constructor Injection*, kdy je doménové třídě předán odkaz na jinou doménovou třídu v konstruktoru
  - takto jsou předávány **povinné** závislosti
  - nepovinné závislosti (v tomto jednoduchém případu nejsou) by se předaly pomocí setrů

### 9.3.2.3. *Business interface*

- pojem z EJB (*Enterprise Java Bean*)
- rozhraní, které popisuje všechny `public` metody doménové třídy, která má být nahrazena
  - pak lze snadno připravit zástupný (*proxy*) objekt

## 9.3.3. Ukázka upravené doménové třídy

- nejprve si připravíme rozhraní `BISpravaDatHodnoceni` (`BI` znamená *Business interface*)

```
import java.util.List;

public interface BISpravaDatHodnoceni {
  public List<HodnoceniPredmetu> getJedenPredmet(String nazev);
  public List<HodnoceniPredmetu> getVsechnaHodnoceni();
  public int getPocetVsechHodnoceni();
}
```

**Poznámka**

Metody `getVsechnaHodnoceni()` a `getPocetVsechHodnoceni()` využijeme později.

- ve třídě `StatistikaHodnoceni` provedeme dvě změny dle předchozích doporučení

1. proměnná `spravaDat` bude typu rozhraní `BISpravaDatHodnoceni`
   - místo dřívějšího typu třídy `SpravaDatHodnoceni`

2. v konstruktoru `StatistikaHodnoceni()` se předá objektu této třídy odkaz na již existující objekt

- tyto dvě změny dovolí, aby požadovaná záměna objektu skutečné třídy `SpravaDatHodnoceni` za *mock* objekt mohla být vnucena z vnějšku, tj. z testovací třídy
  - ♦ skutečnou implementaci třídy `SpravaDatHodnoceni` vůbec nemusíme znát

**Poznámka**

Metoda `prumerZPredmetu()` zůstala nezměněna.

```
import java.util.List;

public class StatistikaHodnoceni {
  private BISpravaDatHodnoceni spravaDat;

  public StatistikaHodnoceni(BISpravaDatHodnoceni spravaDat) {
    this.spravaDat = spravaDat;
  }

  public double prumerZPredmetu(String nazev) {
    List<HodnoceniPredmetu> vybrane;
    vybrane = spravaDat.getJedenPredmet(nazev);
    double suma = 0;
    for (HodnoceniPredmetu hod : vybrane) {
      suma += hod.getZnamka();
    }
    double prumer = suma / vybrane.size();
    return prumer;
  }
}
```

**Poznámka**

Třída `StatistikaHodnoceni` sama nevytváří závisející třídu, ale nechává si ji pomocí DI vložit od někoho jiného. Kontrolu tedy přebírá někdo jiný – to je další návrhový vzor *Inversion of Control*, který úzce souvisí s DI.

# 9.3.4. Knihovna EasyMock

- řeší všechny technické a administrativní záležitosti s *mock* objekty v návaznosti na JUnit
  - je to podpůrná knihovna k JUnit
- `www.easymock.org`
  - ke stažení `easymock-2.5.2.zip` – v prosinci 2009 verze 2.5.2
  - po rozbalení do `C:\Program Files\Java\junit\easymock` budeme využívat soubor `easymock-2.5.2.jar`
    - ♦ samozřejmě je nutné jej uložit na *ClassPath*
  - součástí instalace je i stručný návod k použití

- využívá principu *bussines interface*
  - za **běhu testu** vytvoří implementaci tohoto rozhraní pomocí mechanismu třídy `java.lang.reflect.Proxy`
- má širokou škálu možností, zde budou uvedeny jen základní principy použití

## 9.3.4.1. První testovací případ

- soubor vytváříme jako běžný JUnit test (viz výše) např. pomocí Eclipse
- celý kód třídy:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

import org.easymock.EasyMock;

import java.util.ArrayList;
import java.util.List;

public class StatistikaHodnoceniTest {

  @Test
  public void testPrumerZPredmetu() {
    // příprava dat pro mock objekt
    List<HodnoceniPredmetu> testovaciData =
                        new ArrayList<HodnoceniPredmetu>();
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    testovaciData.add(new HodnoceniPredmetu("Matika", 3));
//    testovaciData.add(new HodnoceniPredmetu("Matika", 5));

    // příprava mock objektu
    BISpravaDatHodnoceni spravaDatAtrapa =
          EasyMock.createStrictMock("atrapaSD",
                                    BISpravaDatHodnoceni.class);
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
                         .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    // vlastní JUnit test
    StatistikaHodnoceni sh = new StatistikaHodnoceni(spravaDatAtrapa);
    double prumer = sh.prumerZPredmetu("Matika");
    assertEquals(2.0, prumer, 0.1);

    // ukončení práce s mock objektem
    EasyMock.verify(spravaDatAtrapa);
  }

}
```

- vlastní testovací případ (metoda `testPrumerZPredmetu()`) má čtyři části

1. příprava dat pro budoucí *mock* objekt
   - zde připravíme zcela konkrétní testovací data, která bude *mock* objekt vracet po zavolání metody `getJedenPredmet()`
2. příprava *mock* objektu
   - vytvoření objektu

```
BISpravaDatHodnoceni spravaDatAtrapa =
          EasyMock.createStrictMock("atrapaSD",
                                    BISpravaDatHodnoceni.class);
```

     - ♦ využívá se typování na *bussines interface* `BISpravaDatHodnoceni`
     - ♦ první skutečný parametr je řetězec `"atrapaSD"` a je to pojmenování tohoto objektu
       - pojmenování je důležité při případných chybových hlášeních
     - ♦ druhý parametr je odkaz na *bussines interface*, který využije `java.lang.reflect.Proxy`
   - nahrání dat do objektu

```
EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
                                .andReturn(testovaciData);
```

     - ♦ jako odezvu na budoucí volání `getJedenPredmet("Matika")` vrátí objekt `testovaciData`
     - ♦ data se do *mock* objektu skutečně sekvenčně nahrávají jako záznamy (jako na magnetofonovou pásku)
       - ukázku sekvence záznamů viz dále
   - ukončení nahrávání dat

```
EasyMock.replay(spravaDatAtrapa);
```

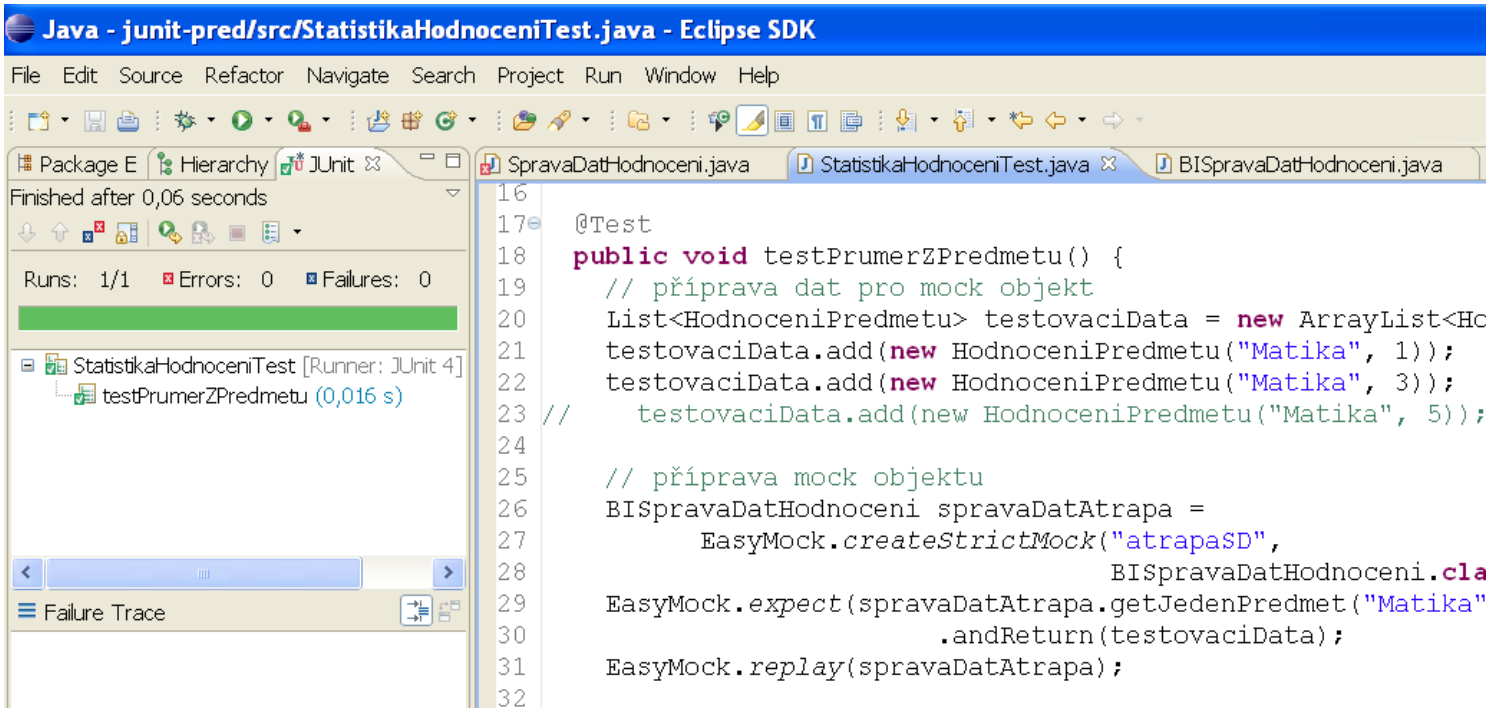
     - ♦ od této chvíle je *mock* objekt připraven k použití
     - ♦ tento příkaz je nutný, bez něj je po spuštění testu vyhozena výjimka `NullPointerAssignment`
3. vlastní JUnit test
   - při vytváření objektu třídy `StatistikaHodnoceni` je mu podstrčen odkaz na *mock* objekt `spravaDatAtrapa`
   - dále následuje běžný JUnit *assert*
4. ukončení práce s *mock* objektem
   - ověřuje, zda byl *mock* objekt skutečně zcela použit, tj. byly využity všechny (zde jen jeden) záznamy
     - ♦ pokud ne, vyhazuje výjimku

■ postup práce je tedy následující sled kroků:

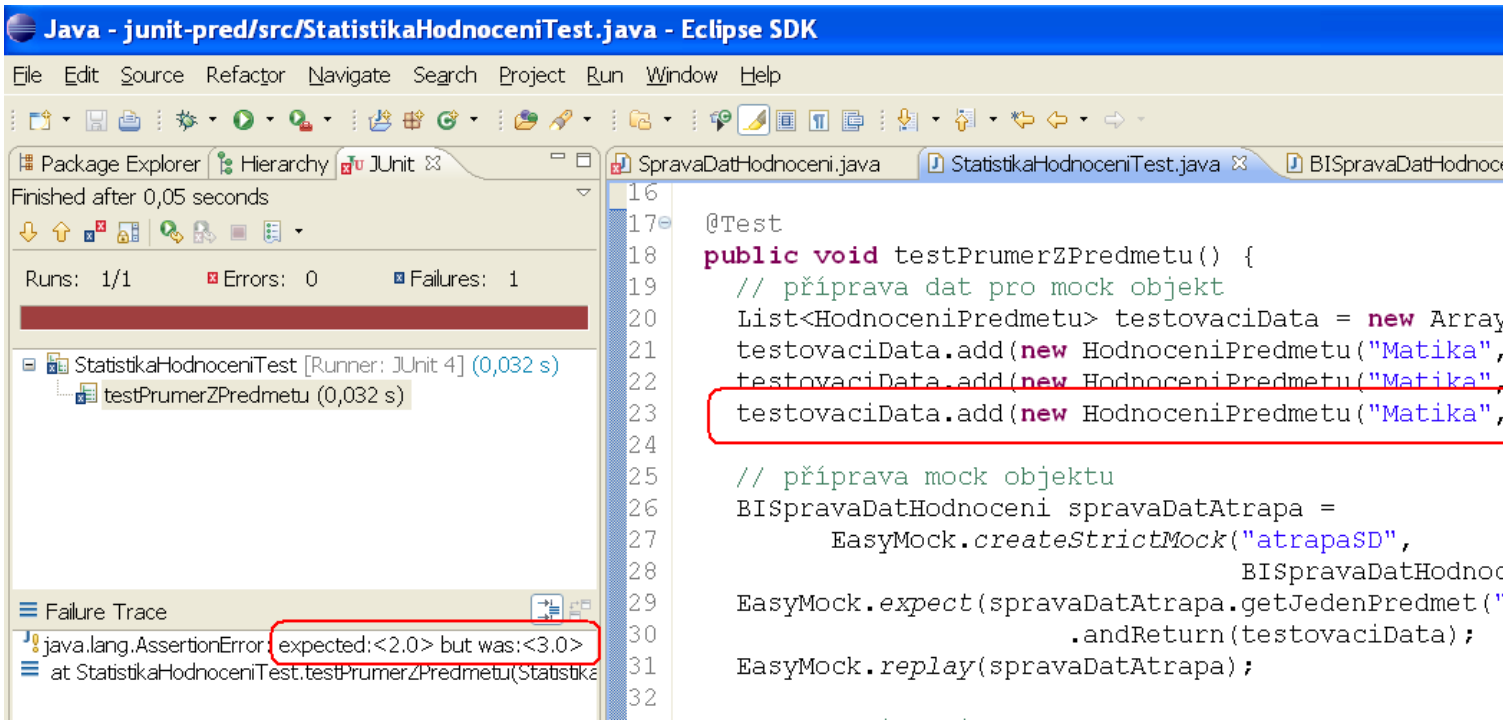
1. vytvořit *mock* objekt pomocí statické tovární metody `EasyMock.createStrictMock()`

2. ve fázi recordingu nahrát do *mock* objektu záznamy, pro konkrétní volání `EasyMock.expect()`
3. přepnout *mock* objekt do fáze přehrávání `EasyMock.replay()`
4. v testované doménové třídě nahradit původní objekt připraveným *mock* objektem `StatistikaHodnoceni(spravaDatAtrapa);`
5. zavolat testovanou metodu v níž se volá metoda, kterou simuluje *mock* objekt `sh.prumerZPredmetu("Matika");`
6. ověřit výsledek JUnit testu `assertEquals(2.0, prumer, 0.1);`
7. zkontrolovat, zda byly z *mock* objektu použity všechny záznamy `EasyMock.verify()`

- po spuštění test proběhne správně




- pokud změníme testovací data (odkomentujeme přidání dalšího hodnocení), dostaneme:




## 9.3.4.2. Využití `@Before` a `@After`

- ve skutečnosti by předchozí případ pravděpodobně využíval možnosti oddělení úvodních a ukončujících akcí od skutečného testu
  - to ale nijak nemění principy práce s *mock* objekty

```
public class StatistikaHodnoceniTestRozdeleni {
  private BISpravaDatHodnoceni spravaDatAtrapa;

  @Before
  public void setUp() throws Exception {
    // příprava dat pro mock objekt
    List<HodnoceniPredmetu> testovaciData =
                        new ArrayList<HodnoceniPredmetu>();
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    testovaciData.add(new HodnoceniPredmetu("Matika", 3));

    // příprava mock objektu
    spravaDatAtrapa = EasyMock.createStrictMock("atrapaSD",
                                    BISpravaDatHodnoceni.class);
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
                        .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);
  }

  @After
  public void tearDown() throws Exception {
    // ukončení práce s mock objektem
    EasyMock.verify(spravaDatAtrapa);
  }

  @Test
```

```
  public void testPrumerZPredmetu() {
    StatistikaHodnoceni sh = new StatistikaHodnoceni(spravaDatAtrapa);
    double prumer = sh.prumerZPredmetu("Matika");
    assertEquals(2.0, prumer, 0.1);
  }
}
```

## 9.3.4.3. Využití více záznamů v *mock* objektu

- vytvoříme třídu `StatistikaHodnoceniJina`
  - v metodě `prumerZeVsechPredmetu()` se volají postupně dvě různé metody ze třídy `SpravaDatHodnoceni`

> **Poznámka**
>
> Volání metody `getPocetVsechHodnoceni()` nemá logický smysl, protože počet hodnocení lze samozřejmě zjistit z velikosti vráceného seznamu. Volání je zde použito pro ukázku druhého volání v pořadí.

```
public class StatistikaHodnoceniJina {
  private BISpravaDatHodnoceni spravaDat;

  public StatistikaHodnoceniJina(BISpravaDatHodnoceni spravaDat) {
    this.spravaDat = spravaDat;
  }

  public double prumerZeVsechPredmetu() {
    List<HodnoceniPredmetu> vsechny;
    vsechny = spravaDat.getVsechnaHodnoceni();
    double suma = 0;
    for (HodnoceniPredmetu hod : vsechny) {
      suma += hod.getZnamka();
    }
    int pocet = spravaDat.getPocetVsechHodnoceni();
    double prumer = suma / pocet;
    return prumer;
  }
}
```

- pro tuto třídu vytvoříme testovací třídu `StatistikaHodnoceniJinaTestDveVolani`
  - v metodě `setUp()` jsou do *mock* objektu nahrány dva záznamy
    - tyto záznamy musejí být v daném pořadí budoucího volání
      - prohodí-li se pořadí, test neprojde a je vyhozena výjimka

```
Unexpected method call atrapaSD.getVsechnaHodnoceni():...
```

    - pokud by nemělo záležet na pořadí, musí se použít místo `EasyMock.createStrictMock()` `EasyMock.createMock()`

– to ale není dobrý způsob, protože ověření pořadí volání je další způsob testu

```
public class StatistikaHodnoceniJinaTestDveVolani {
  private BISpravaDatHodnoceni spravaDatAtrapa;

  @Before
  public void setUp() throws Exception {
    // příprava dat pro mock objekt
    List<HodnoceniPredmetu> testovaciData =
                        new ArrayList<HodnoceniPredmetu>();
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    testovaciData.add(new HodnoceniPredmetu("Fyzika", 3));

    // příprava mock objektu
    spravaDatAtrapa = EasyMock.createStrictMock("atrapaSD",
                                    BISpravaDatHodnoceni.class);
    // záznam 1
    EasyMock.expect(spravaDatAtrapa.getVsechnaHodnoceni())
                        .andReturn(testovaciData);
    // záznam 2
    EasyMock.expect(spravaDatAtrapa.getPocetVsechHodnoceni())
                        .andReturn(2);
    EasyMock.replay(spravaDatAtrapa);
  }

  @After
  public void tearDown() throws Exception {
    EasyMock.verify(spravaDatAtrapa);
  }

  @Test
  public void testPrumerZPredmetu() {
    StatistikaHodnoceniJina shj =
            new StatistikaHodnoceniJina(spravaDatAtrapa);
    double prumer = shj.prumerZeVsechPredmetu();
    assertEquals(2.0, prumer, 0.1);
  }
}
```

## 9.3.4.4. Více testovacích případů

- skutečná třída `StatistikaHodnoceniSpravna` má v metodě `prumerZPredmetu()` ošetřeny všechny okrajové podmínky včetně toho, že hodnocení může být nevyplněno

```
import java.util.List;

public class StatistikaHodnoceniSpravna {
  private BISpravaDatHodnoceni spravaDat;

  public StatistikaHodnoceniSpravna(BISpravaDatHodnoceni spravaDat) {
    this.spravaDat = spravaDat;
  }
```

```
  /**
   * Počítá průměr z konkrétních předmětů
   * @param nazev název předmětu
   * @return vypočtený průměr, nebo 0, pokud nešlo průměr vypočíst
   */
  public double prumerZPredmetu(String nazev) {
    List<HodnoceniPredmetu> vybrane;
    vybrane = spravaDat.getJedenPredmet(nazev);
    if (vybrane.size() == 0) {
      return 0.0;
    }

    double suma = 0;
    int pocetHodnocenych = 0;
    for (HodnoceniPredmetu hod : vybrane) {
      if (hod.getZnamka() != HodnoceniPredmetu.DOSUD_NEHODNOCENO) {
        suma += hod.getZnamka();
        pocetHodnocenych++;
      }
    }
    if (pocetHodnocenych == 0) {
      return 0.0;
    }
    else {
      double prumer = suma / pocetHodnocenych;
      return prumer;
    }
  }
}
```

- třída `StatistikaHodnoceniTestSpravna` má celkem 5 testovacích případů a využívá s výhodou akce před a po (`@Before` a `@After`)
  - všimněte si, že v `@Before` je i vytvoření objektu třídy `StatistikaHodnoceniTestSpravna` s dosud nedokončeným *mock* objektem
    - do toho se nahrají záznamy až v jednotlivých testovacích případech
    - stále platí, že *mock* objekt lze použít až po `EasyMock.replay()`
  - jednotlivé testovací případy testují vždy jeden kontrakt, takže při výskytu chyby lze snadno nalézt důvod

```
import static org.junit.Assert.assertEquals;

import java.util.ArrayList;
import java.util.List;

import org.easymock.EasyMock;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```
public class StatistikaHodnoceniTestSpravna {
  private BISpravaDatHodnoceni spravaDatAtrapa;
  private List<HodnoceniPredmetu> testovaciData;
  private StatistikaHodnoceniSpravna shs;

  @Before
  public void setUp() throws Exception {
    testovaciData = new ArrayList<HodnoceniPredmetu>();
    spravaDatAtrapa = EasyMock.createStrictMock("atrapaSD",
                                BISpravaDatHodnoceni.class);
    shs = new StatistikaHodnoceniSpravna(spravaDatAtrapa);
  }

  @After
  public void tearDown() throws Exception {
    EasyMock.verify(spravaDatAtrapa);
  }

  // žádný předmět
  @Test
  public void testPrumerZPredmetuZadnyPredmet() {
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
            .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    double prumer = shs.prumerZPredmetu("Matika");
    assertEquals(0.0, prumer, 0.1);
  }

  // nehodnocený předmět
  @Test
  public void testPrumerZPredmetuNehodnocenyPredmet() {
    testovaciData.add(new HodnoceniPredmetu("Matika"));
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
            .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    double prumer = shs.prumerZPredmetu("Matika");
    assertEquals(0.0, prumer, 0.1);
  }

  // nehodnocený předmět a hodnocený předmět
  @Test
  public void testPrumerZPredmetuNehodnocenyAHodnoceny() {
    testovaciData.add(new HodnoceniPredmetu("Matika"));
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
            .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    double prumer = shs.prumerZPredmetu("Matika");
    assertEquals(1.0, prumer, 0.1);
```

```
  }

  // nehodnocený předmět a dva hodnocené předměty
  @Test
  public void testPrumerZPredmetuNehodnocenyAHodnocene() {
    testovaciData.add(new HodnoceniPredmetu("Matika"));
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    testovaciData.add(new HodnoceniPredmetu("Matika", 3));
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
              .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    double prumer = shs.prumerZPredmetu("Matika");
    assertEquals(2.0, prumer, 0.1);
  }

  // dva hodnocené předměty
  @Test
  public void testPrumerZPredmetuHodnocene() {
    testovaciData.add(new HodnoceniPredmetu("Matika", 1));
    testovaciData.add(new HodnoceniPredmetu("Matika", 3));
    EasyMock.expect(spravaDatAtrapa.getJedenPredmet("Matika"))
              .andReturn(testovaciData);
    EasyMock.replay(spravaDatAtrapa);

    double prumer = shs.prumerZPredmetu("Matika");
    assertEquals(2.0, prumer, 0.1);
  }
}
```

## 9.3.4.5. Spouštění z Ant

- protože jsou testy pomocí *mock* objektů nadstavbou JUnit testů, dají se spouštět z Antu zcela stejným způsobem
  - jedinou věcí, kterou je nutno učinit, je doplnění `easymock-2.5.2.jar` do *ClassPath*

```
<property name="junit.path"
          value="C:/Program Files/Java/junit/junit-4.7.jar" />
<property name="mock.path"
          value="C:/Program Files/Java/easymock/easymock-2.5.2.jar" />

<path id="test.classpath">
  <pathelement location="${test.class}" />
  <pathelement location="${junit.path}" />
  <pathelement location="${mock.path}" />
  <fileset dir="${test.lib}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

  - pak funguje spouštění a generování reportů, samozřejmě i pro více testovacích tříd