ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA PEDAGOGICKÁ
KATEDRA VÝPOČETNÍ A DIDAKTICKÉ TECHNIKY

Objektová paradigmata programovacího jazyka Java

BAKALÁŘSKÁ PRÁCE

Jakub Nejdl
Přírodovědná studia, Informatika se zaměřením na vzdělání

Vedoucí práce: Dr. Ing. Jiří Toman

Plzeň, 2014

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně s použitím uvedené literatury a zdrojů informací.

V Plzni, 15. dubna 2014

.....

vlastnoruční podpis

PODĚKOVÁNÍ

Rád bych zde poděkoval vedoucímu mé práce Dr. Ing. Jiřímu Tomanovi za jeho vedení, věcné připomínky a rady, díky kterým jsem byl schopen zkompletovat tuto bakalářskou práci.

ZDE SE NACHÁZÍ ORIGINÁL ZADÁNÍ KVALIFIKAČNÍ PRÁCE.

ABSTRAKT

Práce je věnována problematice objektově orientovaného programování, v programovacím jazyce Java.
V teoretické části jsou definována základní paradigmata objektově orientovaného programování. Základními paradigmaty zpracovanými v této bakalářské práci jsou: Objekty, Dědičnost, Zapouzdření a Polymorfismus.
Teoretická část je doplněná o ukázky částí zdrojových kódů. Programy, z nichž vycházejí zdrojové kódy, byly odladěny ve vývojovém prostředí BlueJ a jsou jako přílohy součástí bakalářské práce.
Cílem této práce je co nejsrozumitelněji vysvětlit a popsat základy objektově orientovaného programování.

OBSAH

ÚVOD

Paradigmata objektově orientovaného programování jsou v dnešní době nejčastěji používanými programovými konstrukcemi. Používají se při vývoji aplikací různého rozsahu. Jsou specifické tím, že programové konstrukce, nejsou vždy triviální a jejich účel není vždy patrný na první pohled.

Programovací jazyk Java, je jedním z nejpoužívanějších programovacích jazyků. Jeho syntaxe je považována za poměrně snadno zvládnutelnou a tak je velmi často používán pro výuku objektově orientovaného programování. Dříve byl místo Javy, pro účely výuky programování hojně využíván programovací jazyk Pascal (jeho objektové rozšíření Delphi). V dnešní době však z různých důvodů získávají při programování rozsáhlejších aplikací jak ve firemní, tak školní sféře dominantní zastoupení právě Java a jazyk C.

Publikací, které se zabývají výukou objektově orientovaného programováním v různých jazycích, je mnoho. Ať už se jedná o publikace, zřejmě nejznámějšího českého pedagoga objektových paradigmat, pana **Ing. Rudolfa Pecinovského, CSc.** (*Myslíme objektově v jazyku Java, Java 8 – Učebnice objektové architektury pro začátečníky, OOP – Naučte se myslet a programovat objektově* atd.), nebo o rozšířené publikace zaměstnance Západočeské univerzity **Doc. Ing. Pavla Herouta, Ph.D.** (*Učebnice jazyka Java, Učebnice jazyka C*). Všechny publikace těchto dvou pedagogů jsou velmi kvalitně zpracovány a širokou veřejností kladně hodnoceny.

Cílem mé bakalářské práce, na rozdíl od výše uvedených publikací, není obsáhnout celou problematiku objektově orientovaného programování, ani probrat celou syntaxi programovacího jazyka Java, ale zaměřit se na nejzákladnější konstrukce, které dělají objektově orientované programování tak populárním a profitují ho pro vývoj moderních aplikací.

Všechny programové konstrukce v této práci budou dle zadání prezentovány v programovacím jazyce Java a doplněny o snímky grafického rozhraní vývojového prostředí BlueJ, které bylo pro výuku paradigmat objektově orientovaného programování v jazyce Java vytvořeno ze známého vývojového prostředí NetBeans.

Téma, paradigmata objektově orientovaného programování jsem si vybral, protože mě při studiu na Západočeské univerzitě velice zaujalo. Při vytváření semestrálních prací, ve kterých bylo nutné tyto paradigmata využívat, jsem si uvědomil nekonečné možnosti jejich využití. Proto, když se naskytla příležitost vypracovat svou bakalářskou práci na toto téma, jsem dlouho neváhal.

1 PROGRAMOVACÍ JAZYK JAVA

1.1 HISTORIE

Programovací jazyk Java se za krátkou dobu své existence dokázal rozšířit do celé řady elektronických zařízení. Předchůdcem Javy je jazyk C, od kterého částečně přebral jeho syntaxi. Na rozdíl od svého předchůdce je Java považována za snadněji zvládnutelný a *„čistší“* programovací jazyk. Dnes je po programovacím jazyku C, druhým nejrozšířenějším programovacím jazykem. Počáteční motivací pro jeho vytvoření bylo vytvoření jazyka vhodného pro tvorbu programového vybavení elektronických spotřebičů, tedy jazyka, nezávislém na počítačových platformách. Předchůdce programovacího jazyka Java, projekt ***Oak***, začal roku 1991 vyvíjet James Gosling O.C., Ph.D. a jeho tým, pod záštitou firmy ***Sun Microsystems*** . Téhož roku byl poprvé představen originální návrh tohoto programovacího jazyka a implementován jeho první překladač a virtuální stroj. Projektu se ze začátku příliš nedařilo. Kolem roku 1993 si však firma Sun uvědomila vzrůstající důležitost World Wide Webu a možnosti využít tento programovací jazyk k programování webových aplikací. Roku 1994 byl tento programovací jazyk přenesen do počítačů a v této době také získal své pojmenování ***Java*** (horká káva). V roce 1995 byla Java poprvé oficiálně představena. Již v té době byl zřejmý její velký potenciál. Díky rostoucímu významu a zejména obchodnímu zájmu využít Internet se začala Java rozšiřovat. Velkým mezníkem ve vývoji Javy se stalo v roce 1995 zařazení její podpory do, v té době, velice populárního prohlížeče ***Netscape Navigator 2.0***. Od té doby se Java velice rozšířila (mobilní telefony, počítače, servery, rozsáhlé firemní apliace atd.). Roku 2007, firma Sun uvolnila zdrojové kódy Javy a ta je od té doby vyvíjena jako open source [1] programovací jazyk. Roku 2010 Javu od firmy Sun odkoupila firma Oracle. (1)

Jedná se o vyspělý objektově orientovaný programovací jazyk, obsahující všechny vlastnosti, vyžadované v moderním objektově orientovaném programování.

[1] ***Open source*** *– Java je vyvíjena otevřeně. Tzn., že její komponenty jsou legálně zdarma dostupné na stránkách firmy Oracle.*

1.2 SILNÉ STRÁNKY JAZYKA

Silnou stránkou jazyka Java je jeho multiplatformnost (běží na většině hlavních platformách hardwaru a operačních systémů), kdy zdrojový kód je přeložen do spustitelného ***mezikódu (byte code – česky bajtkódu)***, který lze následně spouštět na libovolném stroji, který obsahuje runtime prostředí ***Java Virtual Machine*** (dále již pouze JVM).

Přenositelnost spustitelného kódu není a ani nemůže být 100% (hlavně z technických důvodů zapříčiněných rozhraním). Z tohoto důvodu bylo vyvinuto několik edicí Javy, které se vzájemně liší v drobných rozdílech a rozšířeních. Mezi další výhody programovacího jazyka Java patří jeho vysoká bezpečnost vůči hardwaru, na kterém běží JVM, daná překladem programu do bajtkódu a implementováním bezpečnostních mechanismů hlídajících chování programu (práce se zdroji atd.). Tato vlastnost profituje Javu pro použití na vývoj kritických aplikací. Java je tzv. „silně“ objektově orientovaný jazyk. (2)

Navíc je považována za jednoduchý jazyk. To znamená, že její syntaxe není složitá. (3)

Zároveň je velice robustní, což může být považováno za výhodu i nevýhodu současně. Výhodou je, že neumožňuje některé programátorské chyby, jako nižší programovací jazyky [2], např. při správě paměti. Na druhou stranu to, že neumožňuje uživateli spravovat paměť, může někdo považovat za nevýhodu.

1.3 SLABÉ STRÁNKY JAZYKA

Mezi velké nevýhody tohoto programovacího jazyka patří jeho rychlost, kterou hodně ovlivňuje překlad v runtime prostředí JVM. V dnešní době se však tato nevýhoda dá kompenzovat použitím specializovaných překladačů na urychlení překladu v cílovém prostředí. Např.: ***Java just-in-time*** atd.. Just-in-time kompilace je rozšíření JVM o myšlenku dynamické kompilace kódu. Jedná se o překlad bajtkódu v okamžiku, kdy se vykonává, aniž by se ukládala do interní ***cache*** [3] paměti. Tímto procesem odpadá většina

[2] ***Nižší programovací jazyky****, jsou jazyky, jejichž instrukce jsou stejné (nebo velmi podobné) jako příkazy procesoru.*

[3] ***Cache memories (česky „mezipaměti****“) jsou vyrovnávací paměti používané k vyrovnávání rychlostí mezi rychlými a pomalými zařízeními. Např. mezi procesorem a operační pamětí.*

předchozích nevýhod systému. Zároveň však odpadá i několik výhod (např. výhoda sdílení kódu). Současně má tento proces vysoké požadavky na rychlost převodu, což znamená, že optimalizace nemohou být výpočetně náročné. Proti kompilaci přímo do nativního kódu místo do bajtkódu hovoří zejména narušení filozofie jediné kompilace a následného libovolného spouštění na různých platformách. I tímto se dnes zabývají různé projekty např***. GNU Compilers Collection***. (4)

1.4 ZPRACOVÁNÍ PROGRAMU V JAZYCE JAVA

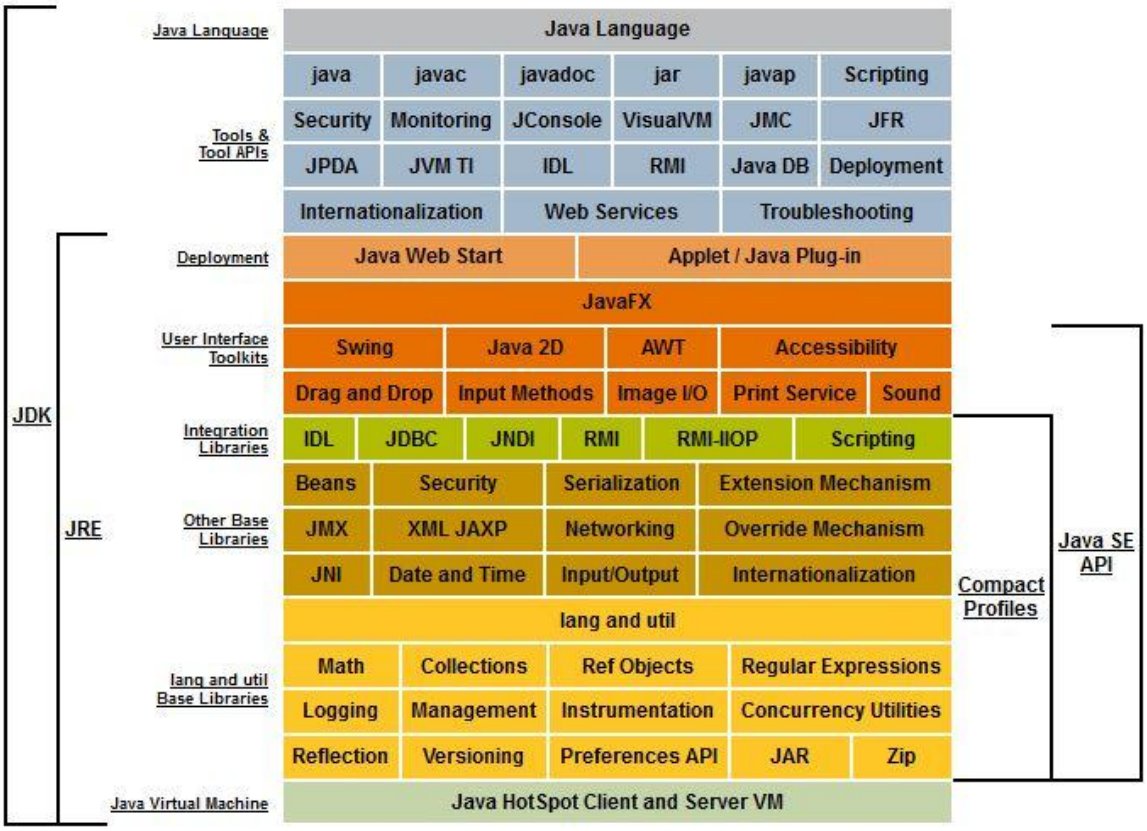
Ke zpracování programu v jazyce Java je nutné nainstalovat ***překladač*** (kompilátor) a ***interpret*** jazyka. Překladač umožní překlad kódu (algoritmů) z vývojového prostředí do bajtkódu a interpret tento přeložený kód vykoná. Překladač je obsažen v komplexním balíku komponent ***JDK*** (Java Development Kit). Součástí JDK je také runtime prostředí **JRE** (Java Runtime Enviroment), které obsahuje interpret jazyka ***JVM***. JDK je mnoho verzí, jelikož Java a její balíčky prochází neustálou aktualizací ze strany firmy Oracle. Nejnovější verzi JDK je možno jako open source stáhnout ze stránek firmy Oracle. (2)

Standardní postup při zpracování programu v Javě je, že program prochází pěti fázemi, ve kterých využívá různých nástrojů:

- **Editováním**
  - vývojové prostředí (BlueJ, NetBeans, Eclipse atd.)
  - editování kódu, vytváření implementací ve vývojových prostředích
- **Kompilací (překladem**)
  - po zkompilování zdrojového kódu kompilátorem, získáme spustitelný bajtkód
- **Zavedením**
- **Ověřováním (verifikací)**
- **Prováděním**
  - JVM - provádění kódu

Čtyři z těchto fází jsou běžné i v ostatních programovacích jazycích. Fáze ověřování je specifická zejména pro Javu. Dochází při ní k verifikaci bajtkódu. Umožňuje tak dosáhnout

velmi vysoké bezpečnosti spuštěného programu a tudíž i záruku konzistence zdrojů, se kterými program pracuje. (2)



Obrázek 1: Komponenty platformy Java (5)

- ***JDK*** – obsahuje komponenty od ***Java Language***, až po ***Java Virtual Machine***
  - komplexní balík, obsahující komponenty potřebné pro vývoj a běh programu
- ***JRE***– obsahuje komponenty od ***Deployment Technologies***, až po ***Lang and util Base Libraries***
  - obsahuje komponenty potřebné pro běh programu (ne pro vývoj)

1.5 VÝVOJOVÉ PROSTŘEDÍ BLUEJ

Vývojové prostředí BlueJ nástroj vyvíjený pro několik platforem (Windows, Linux, Mac), určený pro výuku OOP v jazyce Java. Demonstrace OOP budou v této bakalářské práci prezentovány právě přes ***Graphical User Interface*** [4] (dále jen GUI) prostředí BlueJ.

[4] ***Graphical User Interface*** – *(česky Grafické Uživatelské Rozhraní) – je uživatelské rozhraní programu. Přes toto rozhraní zadává uživatel požadavky (vstupy) a program přes něj prezentuje své výsledky (výstupy).*

Vývojové prostředí BlueJ bylo vyvinuto tak, aby bylo co nejjednodušší. Programátor se tedy nemusí rozptylovat širokou škálou možností nastavení a může se plně soustředit na vývoj programů. (6)

Kromě psaní kódu, umožňuje pozorovat:

- závislosti mezi jednotlivými třídami (diagram tříd)
- vytvořené objekty v paměti, jejich data a metody

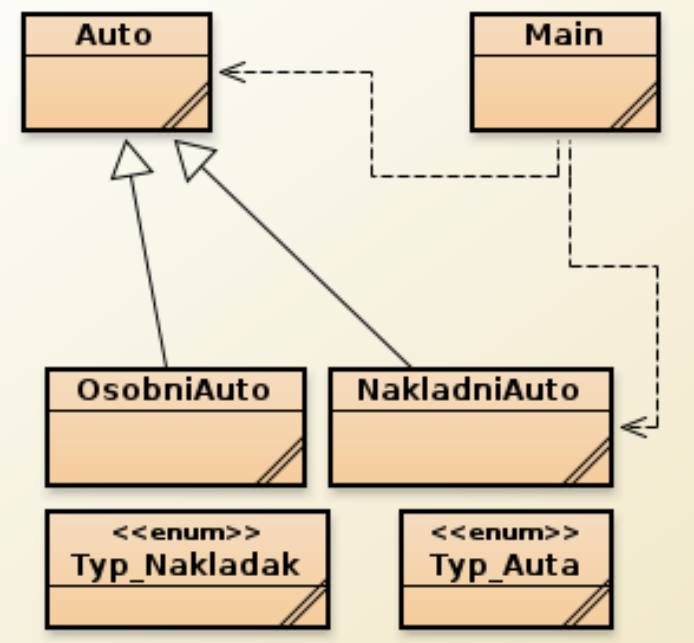
Zároveň nabízí nástroj pro generování dokumentace jednotlivých programů, ve formátu ***.html***.

1.5.1 UML V PROSTŘEDÍ BLUEJ

Objektově orientované programování klade vyšší nároky na programátora. A to nejen při psaní kódu, ale zejména při návrhu a analýze vytvářeného programu. Pro účel grafického znázornění vyvíjené aplikace byl vytvořen modelovací jazyk ***UML*** (Unified Modeling Language). UML slouží ke grafickému modelování objektů a popisu konstrukcí reálného světa, převedených do světa informačních systémů. (7)

Pomocí UML, můžeme vytvořit např.:

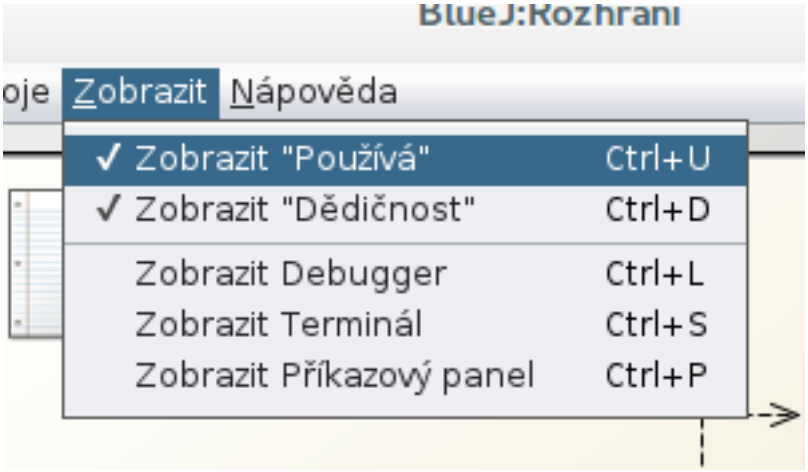
- **diagram tříd**
  - znázorňuje jednotlivé třídy a vztahy mezi nimi (asociace, rozhraní, balíčky atd.)
  - nejzákladnější diagram, často vytvářen na začátku projektu
- **stavový diagram**
  - znázorňuje stavy objektu a jejich přechody
  - používají se především pro popis chování objektu
- **diagram nasazení**
  - ukazuje rozložení komponent na různém hardwaru (různých strojích)
- **diagram případu použití**
- **diagram komponent**
- **diagram spolupráce**
- **diagram aktivit** a další (7)



Obrázek 2: UML ve vývojovém prostředí BlueJ

Zjednodušený UML diagram tříd nabízí také vývojové prostředí BlueJ.

V tomto zjednodušeném UML diagramu v sobě nemají třídy zapsány metody, ani jednotlivé proměnné. Nenalezneme zde ani specifikátory přístupu jednotlivých metod a proměnných. Můžeme zde však pozorovat závislosti (dědění / implementování) mezi jednotlivými třídami nebo rozhraními.



Obrázek 3: Systémové menu v prostředí BlueJ

V systémovém menu (záložka ***Zobrazit***) můžeme povolit zobrazení vztahů mezi třídami na úrovni používání. To, že jedna třída používá druhou, znamená, že v těle třídy dochází volání metod, nebo jsou zde použity proměnné z druhé třídy.

2 OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

Tématem této bakalářské práce jsou paradigmata objektově orientovaného programování (dále jen OOP). Jazyky, jež umožňují tyto paradigmata vytvářet, nazýváme objektově orientované programovací jazyky. Jedním z nich je programovací jazyk Java. Mezi další patří např.: Perl, Object Pascal, C++, C#, PHP a další.

OOP nám má ulehčit vytváření programů, které jsou buď projekcí reálného světa, nebo simulací virtuálního světa námi vymyšleného. Čím blíže se těmto světům přiblížíme, tím lepší bude výsledný program. Všechny tyto světy, jsou ve skutečnosti tvořeny objekty. U OOP tomu není jinak a hlavní roli zde také hrají objekty (viz. Objekty obecně, kap. Objekty). (8)

Aby byl programovací jazyk objektově orientovaný, musí umožňovat:

- **vytváření objektů**
- **dědičnost**
  - dědění vlastností
- **zapouzdření dat**
  - kontrola přístupu k datům
- **polymorfismus (mnohotvarost)**

2.1 OBJEKTY

Objekty jsou definovány množinou ***dat*** a množinou ***funkcí*** (v Javě je nazýváme metody), které jsou schopny vykonávat.

Množina dat objektu (proměnné) reprezentuje jeho ***stav*** *(state).* Ve skutečnosti se jedná o vnitřní paměť, která je přidělená objektu a která je z vnějšku objektu nepřístupná. Tato paměť umožňuje objektu pamatovat si stav, ve kterém se zrovna nachází.

Množina *funkcí (metod)* vyjadřuje jeho ***chování*** *(behavior)*. Funkce jsou části programu, které vykonávají nějakou specializovanou činnost nad vnitřní pamětí (nad daty).

Další vlastností objektu, která ho definuje, je schopnost přijmout a zpracovat zprávy z vnějšku. Jedná se o požadavek o provedení *„služby“* jinému objektu (např. objekt obdrží zprávu, že má provést výpis na konzoli). Pokud objekt obdržené zprávě *rozumí* a umí na ní

zareagovat (má implementované metody, které na zprávu reagují), tak tak učiní. Pokud však obdržené zprávě *nerozumí*, není na ní schopen jakkoliv reagovat. (9)

Komunikační protokol objektu je v objektově orientovaném programování často skloňován. V různých programovacích jazycích, může být realizován jinak.

Programátor může využívat předdefinované objekty, které čerpá z externích knihoven, popřípadě může vytvářet vlastní objekty. Při analýze se programátor snaží popsat objekty, které v rámci programu mají vzniknout. Snaží se popsat jejich požadované vlastnosti (proměnné) a chování (metody). (10)

Klasický program vytvářený technikou objektově orientovaného programování vytváří velké množství objektů.

2.1.1 TŘÍDY

Třída je základním paradigmatem objektově orientovaného programování. V těle třídy jsou implementovány metody, proměnné, konstruktory (viz. kap.: Konstruktory) a vnořené třídy. Slouží jako šablona pro vytváření libovolného počtu objektů se stejnými vlastnostmi. To znamená, že objekt je konkrétní instancí třídy s vlastnostmi a počátečním stavem, definovaným pomocí metod a proměnných třídy. Stačí tedy nadefinovat jednu třídu pro vytváření objektů, kterých následně můžeme vytvořit kolik chceme a nebudeme se v jejich definici opakovat. Pouze odkážeme, že objekt je instancí dané třídy, ve které je zahrnuta jeho definice. Třída jako taková má alokovanou paměť pouze pro proměnné třídy (viz. kap. Proměnné třídy). Paměť pro instanci objektu je alokována až při jeho vytvoření (viz. kap. Vytvoření objektu). Každý spustitelný program vytvořený v jazyce Java obsahuje vždy minimálně jednu třídu. (11)

Tříd může být několik druhů:

- **Základní druh třídy**
    - Lze z ní vytvořit instanci, může obsahovat vnořené třídy.[5]
    - Každý spustitelný program musí obsahovat minimálně jednu.

[5] ***Instance*** *– je konkrétní datový objekt umístěný v paměti počítače.*

- **Vnořené třídy**
  - Jsou implementovány uvnitř jiné třídy / rozhraní.
  - K proměnným / metodám vnořené třídy nemá vnější třída přístup. Naopak vnořená třída má neomezený přístup k proměnným / metodám třídy vnější.
  - Na vnořenou třídu nelze deklarovat referenční proměnnou. (2)
- **Abstraktní třídy** (viz. kap.Abstraktní třídy)
  - Instanci abstraktní třídy operátorem ***new*** nelze vytvořit.

Deklarace třídy

Deklarace třídy je umístěna v hlavičce programu (nejedná-li se o speciální druh třídy – ***vnořenou třídu***), hned pod pojmenováním balíčku, ve kterém je umístěna a pod importem knihoven, které používá.

Formální zápis třídy vypadá takto (viz. příloha: Vytváření třídy):

```
public class Trida(){
   // tělo třídy
}
```

- **public** – specifikátor přístupu (viz. kap.: Zapouzdření)
- **class –** klíčové slovo, které deklaruje, že se jedná o třídu
- **Trida** – identifikátor třídy (její jméno),
  - Používáme jména popisující význam třídy, začínající velkým písmenem.
  - Pokud použijeme více slov, každé z nich by mělo začínat velkým písmenem.
  - Identifikátor by zároveň neměl být moc dlouhý ani moc krátký.

2.1.2 METODY

Metody jsou části programu, které mají za úkol reagovat na obdržené zprávy tím, že provedou nějakou specializovanou činnost, na kterou jsou implementovány. Mají za úkol co nejvíce přiblížit chování objektů reálnému světu. Ve skutečnosti v podstatě předurčují chování objektu.

Metody můžeme dělit z několika hledisek:

- **podle toho, jestli navrací hodnotu:**
  - s návratovou hodnotou
  - bez návratové hodnoty
- **podle toho, jestli jim předáváme parametry:**
  - s parametry
  - bez parametrů
- **podle toho, jestli náleží třídě nebo instanci:**
  - Metody třídy
  - Metody instance

Každý spustitelný program napsaný v programovacím jazyce Java obsahuje jednu, nebo více metod. Voláme je buď se seznamem parametrů (**metody s parametry**), kterými předáváme hodnoty potřebné pro výpočet, nebo je voláme bez parametrů (**metody bez parametrů**). Metody mohou mít dále návratovou hodnotu (**metody s návratovou hodnotou**). Typ návratové hodnoty je vždy uveden v hlavičce metody. Tyto metody mají svůj kód ukončen klíčovým slovem ***return***. Pokud žádnou hodnotu nevrací, je v hlavičce metody na místě typu návratové hodnoty slovo ***void*** (**metody bez návratové hodnoty).**

Příklad formálního zápisu metody vypadá takto (viz. příloha: Metody třídy / instance):

```
public static int metoda(int parametr){
            return parametr - 3;
}
```

- **public** – specifikátor přístupu (viz. kap.: Zapouzdření)
- **static –** pro statickou metodu (viz. kap.: Metody třídy)
- **int –** návratový typ metody (dále: String, double, boolean atd.)
- **metoda** – identifikátor metody
  - Vždy by měl začínat malým písmenem.
  - Pokud je identifikátor metody víceslovný, jsou za sebe slova zapsána bez mezer a zároveň druhé a všechna slova (kromě prvního) začínají velkým písmenem.
- **int parametr** – parametr metody – může jich být libovolné množství a mohou být různých typů (např. i typu Object)

- **return parametr – 3;** - návratová hodnota - na konci metody je použito klíčové slovo ***return***, za kterým je hodnota, kterou metoda navrací

Metody třídy

Metody třídy (uvozené slovem ***static***) jsou společné pro všechny instance a mohou přistupovat pouze k proměnným třídy (viz. kap.: Proměnné třídy). Tyto metody jsou nezávislé na instanci třídy. To znamená, že pro jejich použití není potřeba vytvořit žádnou instanci třídy. Na druhou stranu je možné tyto metody z instance třídy přímo volat.

Nejčastěji se statické metody využívají při matematických výpočtech, popřípadě při textových výpisech do konzole.

Příklad metody třídy (viz. příloha: Metody třídy / instance):

```
public static int soucet(int a, int b){
   return a + b;
}
```

Metody instance

Metody instance definují chování objektu. Tyto metody jsou z vnějšku objektu neviditelné (nelze k nim přistupovat). Voláme je přes referenci na daný objekt. Mohou přistupovat jak k proměnným třídy, tak k proměnným dané instance.

Příklad metody instance (viz. příloha: Metody třídy / instance):

```
public int getPocet(){
   return pocet;
}
```

Speciálním typem metod jsou:

- **Metoda main**
- **Konstruktory**

Metoda main

Speciální typ metody, který musí obsahovat každý program v Javě, má-li být spustitelný. Metoda ***main*** totiž obsahuje spustitelný kód programu. Nejčastěji v ní vytváříme instance tříd. Po vykonání kódu, který je v ***main*** implementován, program skončí.

Formální zápis metody ***main:***

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

Main musí být vždy definována tímto způsobem, aby byla jednoznačně rozeznána překladačem. Obsahuje parametr pojmenovaný ***args*** „pole Stringů“, který můžeme používat pro předávání parametrů spouštěnému programu.

Příklad spuštění ***main*** se zadanými parametry včetně ošetření výjimek (viz. příloha: Metoda main):

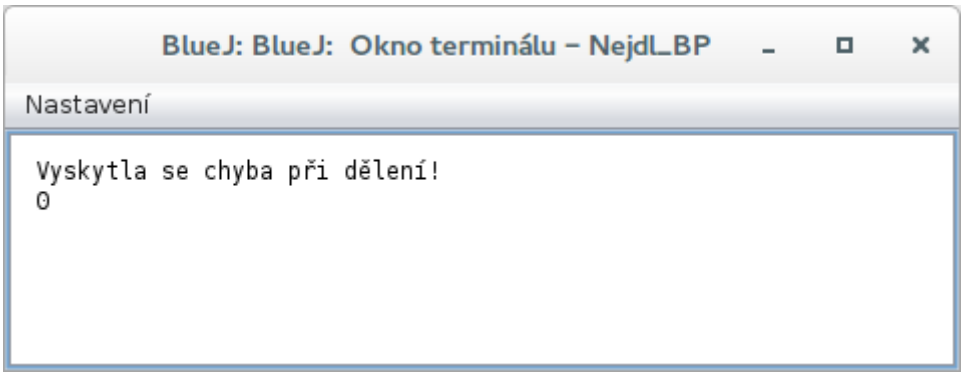
```
private static double a,b,vysledek;
public static void main(String[] args){
        try{
            a = Double.parseDouble(args[0]);
            b = Double.parseDouble(args[1]);
            vysledek = a / b;
            System.out.println(Double.toString(vysledek));
        }catch (Exception e){
            System.out.println("Vyskytla se chyba při dělení!");
            System.out.println(e.getMessage());
        }
}
```

- Při spuštění programu zadáme jako parametry dvě čísla typu ***double.***

Obrázek 4: Zadání parametrů při spuštění programu

- V bloku ***try*** je umístěn potenciálně nebezpečný kód (v našem případě se jedná o parsování indexů pole *Stringů* a dělení dvou čísel). Jedná se o bezpečnostní mechanizmus, který nazýváme ***výjimky***. Výjimkami rozumíme výjimečný stav, který může vzniknout při běhu programu. Část kódu, u kterého je pravděpodobný vznik výjimečného stavu, ošetřujeme pomocí konstrukce ***try – catch*** popřípadě ***finally***. Nebezpečný kód je umístěn v bloku ***try***.
- Pokud se při vykonávání tohoto kódu vyskytne chyba, skočí program do části kódu, který *odchytává* výjimky (***catch***) a text s identifikátorem výjimky se vypíše do konzole (např. pokud nezadáme žádné parametry při spuštění programu). Blok

finally je nepovinný. Obsahuje příkazy, které se vykonají vždy, nezávisle na tom, jestli došlo k výjimečnému stavu nebo ne.



Obrázek 5: Chybové hlášení. Výpis identifikátoru výjimky

- Pokud jsou zadány správné parametry, vypočítá se podíl dvou zadaných čísel, výsledek se vypíše do konzole a ***main*** vlákno skončí – tím skončí celý program.

VÝSTUP PO SPUŠTĚNÍ MAIN SE ZADANÝMI PARAMETRY (5.12, 8.54):
```
0.599316159250587
```

2.1.3 PROMĚNNÉ

Proměnné představují v programovacích jazycích části paměti, které používáme k uložení dat.

Java rozlišuje dva základní typy proměnných:

- **Referenční proměnné**
  - uchovávají referenční datové typy
- **Primární proměnné**
  - uchovávají hodnotové datové typy
  - např. int, String, boolean, double atd.

Proměnné také dále dělíme podle toho, jestli uchovávají data třídy nebo instance:

- **Proměnné třídy**
- **Proměnné instance**

Z hlediska paradigmat objektově orientovaného programování jsou nejdůležitější Proměnné třídy, Proměnné instance a Referenční proměnné.

Referenční proměnné

Činnost referenčních proměnných úzce souvisí se správou paměti. Jelikož se v Javě s pamětí nepracuje přímo, jako například v C, nahrazují referenční proměnné **pointery** [6] z nízko-úrovňových programovacích jazyků.

Neuchovávají data ve smyslu např. jednoho čísla, jednoho znaku atd.. Obsahují totiž referenci do paměti. Referencí do paměti se rozumí adresa místa v paměti, kde je uložen začátek datové struktury objektu (proměnné referenčního datového typu). (12)

Příklad přiřazení reference referenční proměnné, je v kapitole Vytváření objektů třídy.

Proměnné třídy

Proměnné třídy jsou podobně jako metody třídy uvozené slovem ***static***. Nazýváme je tedy statickými proměnnými. Podobně jako u statických metod, jsou statické proměnné nezávislé na instanci. Neuchovávají tedy data objektu, nýbrž data třídy. Mají globální platnost a mohou s nimi pracovat jak metody třídy, tak metody instance, mají-li k nim přístupová práva. Ve všech objektech mají stejnou hodnotu.

Pro proměnné třídy, alokuje program paměť ihned po jeho spuštění. To znamená, že nečeká na vytvoření instance třídy.

Příklad implementace proměnné třídy (viz. příloha: Proměnné třídy / instance):

```
private static String promennaTridy = „Ahoj“;
```

- **private** – specifikátor přístupu (viz. kap. Zapouzdření)
- **static** – deklaruje, že se jedná o proměnnou třídy
- **String** – datový typ proměnné
- **promennaTridy** – identifikátor proměnné
    - Pro identifikátory proměnných, platí stejná nepsaná pravidla, jako pro identifikátory metod.

[6] ***Pointery*** *– (česky Ukazatele) jsou proměnné, do kterých se ukládají adresy paměťových bloků, ve kterých je uložen libovolný typ dat. Pointery se používají např. v programovacím jazyce C.*

Proměnné instance

Proměnné instance, na rozdíl od proměnných třídy, uchovávají data jednotlivých objektů. Hodnota těchto proměnných může být pro každý objekt jiná. Přistupujeme k nim přes referenci na daný objekt (viz. kap.: Referenční proměnné) a při jejich implementaci se neuvádí klíčové slovo ***static***.

Alokace paměti pro proměnné instance proběhne při vytváření objektu. Jejich inicializaci lze provést v konstruktoru objektu (viz. kap. Konstruktory, Klíčové slovo this) , což se u proměnných tříd důrazně nedoporučuje, protože by se jejich hodnota změnila vždy při vytváření nového objektu. Hodnoty těchto proměnných jsou pro každý objekt jedinečné.

Příklad implementace proměnné instance (viz. příloha: Proměnné třídy / instance):

```
private String promennaInstance;
```

- **private** – specifikátor přístupu (viz. kap.: Zapouzdření)
- **String** – datový typ proměnné
- **promennaInstance** – identifikátor proměnné
    - Pro identifikátory proměnných, platí stejná nepsaná pravidla, jako pro identifikátory metod.

2.1.4 ŽIVOTNÍ CYKLUS OBJEKTU

Každý objekt prochází určitým životním cyklem. Na začátku životního cyklu se objektu alokuje paměť pro proměnné. Po zavolání konstruktoru se do těchto proměnných dosadí hodnoty. Po provedení činnosti, na kterou byl objekt implementován a vytvořen, je objekt ukončen. V tu chvíli mu Garbage Collector (viz. kap.: Ukonční (rušení) objetku) odebere alokované zdroje, které jsou poté dostupné pro další činnost programu. (13)

Vytváření objektů třídy

K vytvoření objektu, slouží operátor ***new***.

Formální zápis syntaxe vytvoření objektu (viz. příloha: Vytváření objektu třídy):

```
new Reference();
```

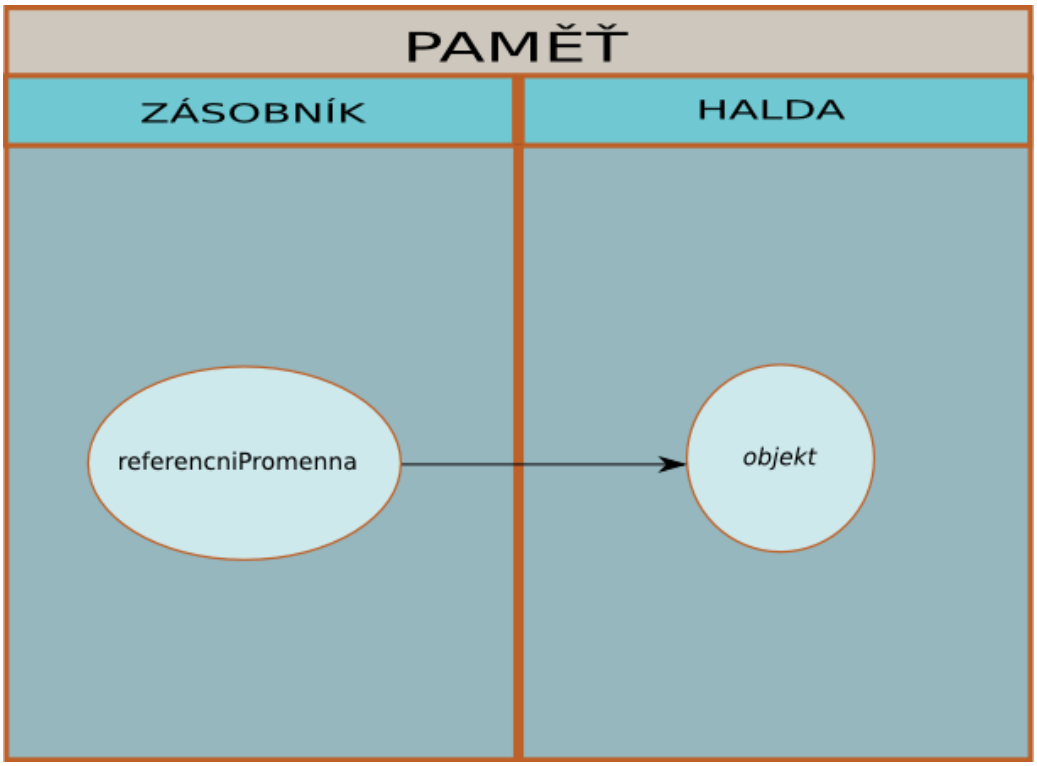
- **new** – operátor alokující paměť pro objekt
- **Reference()** - volání konstruktoru, který se jmenuje stejně jako třída (viz. kap.: Konstruktory)

Operátor ***new*** dále vrací referenci na objekt. Reference se poté předá referenční proměnné. (12)

Příklad navrácení reference do paměti (viz. příloha: Vytváření objektu třídy):

```
Reference referencniPromenna = new Reference();
```

- **Reference**– identifikátor třídy
- **referencniPromenna** – referenční proměnná, obsahující referenci do paměti, kde se nachází objekt
- **new** – operátor vracející referenci do paměti
- **Reference()** – identifikátor konstruktoru (viz. kap.: Konstruktory)



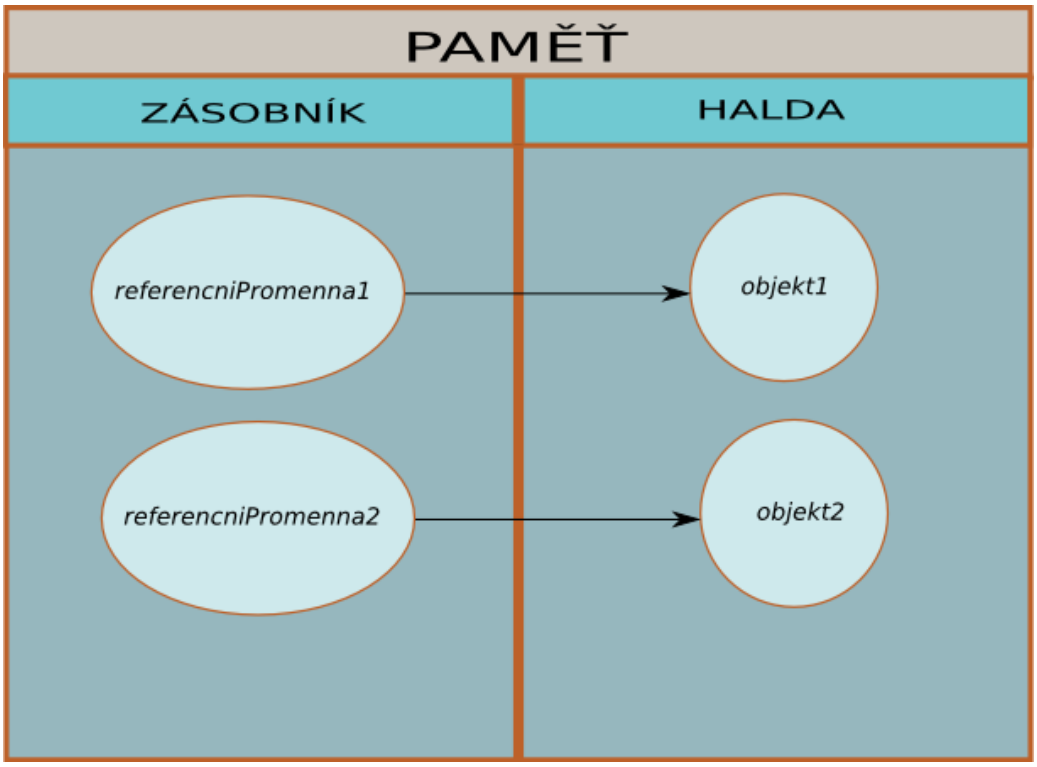
Obrázek 6: Referenční proměnná odkazující na objekt (12)

- **Halda** - část paměti, používaná pro alokaci zdrojů objektu
- **Zásobník** – část paměti, ve které jsou umístěny referenční proměnné, ukazující na počáteční adresu alokované paměti objektu

Jedna referenční proměnná, může ukazovat pouze na jeden objekt. Na druhou stranu na jeden objekt může ukazovat více referenčních proměnných. K tomu může dojít následujícím způsobem:

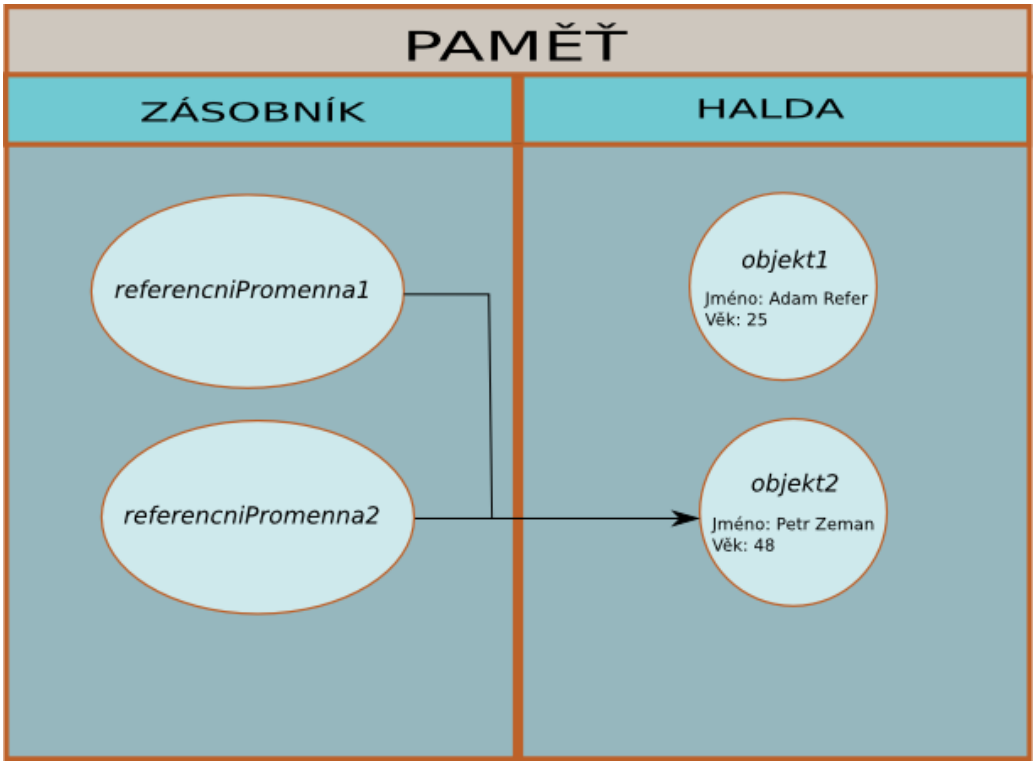
```
Reference referencniPromenna1 = new Reference(String jmeno, int vek);
//Objekt1
Reference referencniPromenna2 = new Reference(String jmeno, int vek);
//Objekt2
referencniPromenna1 = referencniPromenna2;
```

V první fázi se v paměti vytvoří dva objekty. Na každý objekt ukazuje jeho referenční proměnná.



Obrázek 7: Dvě referenční proměnné. Každá odkazuje na jiný objekt. (12)

Poté však proběhne přiřazení ***referencniPromenna1*** = ***referencniPromenna2***. Důsledkem tohoto přiřazení bude, že ***referenciPromenna1*** ukazuje do paměti na místo, kde je umístěn ***objekt2***. (12)



Obrázek 8: Dvě referenční proměnné, odkazující na stejný objekt. (12)

To znamená, že nyní máme dvě referenční proměnné odkazující na stejný objekt. Přes obě referenční proměnné můžeme k tomuto objektu přistupovat (volat jeho metody a proměnné). ***Objekt1*** teď však nemá na sebe žádnou referenci (žádná referenční proměnná na něj neodkazuje). S objektem už se nedá pracovat a tak dojde k jeho odstranění z paměti (viz. kap. Ukonční (rušení) objetku). (12)

Např. (viz. příloha: Vytváření objektu třídy):

```
private String jmeno;
private int vek;
Reference(String jmeno, int vek){
    this.jmeno = jmeno;
    this.vek = vek;
}
private void vypisJmeno(){
    System.out.println("Jméno: "+jmeno);
    System.out.println("Věk: "+Integer.toString(vek));
    System.out.println("\n");
}
public static void main(String[] args){
    Reference referencniPromenna1 = new Reference("Adam Refer", 25);
    Reference referencniPromenna2= new Reference("Petr Zeman", 48);
    System.out.println("Reference 1:");
    referencniPromenna1.vypisJmeno();
    System.out.println("Reference 2:");
```

```
    referencniPromenna2.vypisJmeno();
    referencniPromenna1 = referencniPromenna2; //přiřazení
    System.out.println("Reference 1 po přiřazení:");
    referencniPromenna1.vypisJmeno();
    System.out.println("Reference 2 po přiřazení:");
    referencniPromenna2.vypisJmeno();
}
```

VÝSTUP PO SPUŠTĚNÍ METODY MAIN:

```
Reference 1:
Jméno: Adam Refer
Věk: 25
Reference 2:
Jméno: Petr Zeman
Věk: 48
Reference 1 po přiřazení:
Jméno: Petr Zeman
Věk: 48
Reference 2 po přiřazení:
Jméno: Petr Zeman
Věk: 48
```

Konstruktory

Konstruktor je speciálním typem metody. Důležité je, že se jmenuje stejně jako třída. Volá se vždy v okamžiku, kdy dojde k vytvoření instance třídy.

Programátor má možnost pomocí konstruktoru popsat počáteční *stav* objektu (inicializovat proměnné instance).

Konstruktorů jednoho objektu může být více, musí se však lišit v parametrech, aby překladač rozeznal, který konstruktor máme v úmyslu zavolat (viz. kap.: Přetěžování metod (Overloading)).

Příklad konstruktoru (viz. příloha: Konstruktory – klíčové slovo „this“):

```
public class Trida(){
 Trida(parametry){
   //tělo konstruktoru
 }
}
```

- **Trida** – pojmenování konstruktoru je shodné s pojmenováním třídy
- **parametry** – přes parametry předáváme počáteční hodnoty proměnným instance

Příklad vytvoření nové instance (viz. příloha: Konstruktory – klíčové slovo „this“):

```
Trida referencniPromenna = new Trida(parametry);
```

Odkaz na nový objekt je vrácen jako hodnota operátoru ***new***. K proměnným a metodám daného objektu posléze můžeme přistupovat přes referenční proměnnou ***referencniPromenna***.

Klíčové slovo this

Klíčové slovo ***this*** odkazuje na současný objekt (současnou instanci třídy). Slovem ***this*** deklarujeme, že chceme pracovat s konstruktorem / proměnnou nebo metodou současného objektu. U statických členů třídy se ***this*** nepoužívá, protože nenáleží žádné instanci a nelze je tedy volat přes odkaz na současný objekt.

Nejčastěji je slovo ***this*** doplňováno překladačem implicitně. My ho však používáme v případě, že chceme v konstruktoru přistoupit k proměnné aktuálního objektu. Například, pokud jsou parametry konstruktoru pojmenovány stejně jako proměnné třídy, rozlišujeme mezi nimi právě pomocí slova ***this***.

Např. (viz. příloha: Konstruktory – klíčové slovo „this“):

```
private String retezec;
priváte boolean pravda;
public Trida(String řetězec, boolean pravda){
   this.text = text;
   this.pravda = pravda;
}
```

Pomocí slova ***this*** také můžeme v konstruktoru požádat o inicializaci proměnných jiný konstruktor stejné třídy (viz. příloha: Konstruktory – klíčové slovo „this“):

```
private String retezec, retezec2;
private int cislo, cislo2;
priváte boolean pravda;
public Trida(int cislo,String retezec, boolean pravda){
   this.cislo = cislo;
   this.text = text;
   this.pravda = pravda;
}
public Trida(int cislo, int cislo2, String retezec, String
            retezec2,boolean pravda){
   this(cislo, retezec, pravda); //volání jiného konstruktoru
   this.cislo2 = cislo2;
   this.retezec2 = retezec2;}
```

Práce s objekty

Při práci s daty a metodami objektu, využíváme odkazů (referencí) na tento objekt. Ať už se jedná o odkaz na aktuální objekt ***this***, nebo odkaz na objekt rodičovské třídy ***super*** (viz. kap.: Klíčové slovo super), nebo o ***referenční proměnnou***, která obsahuje odkaz na konkrétní objekt. Přes tyto odkazy, získáváme přístup k proměnným nebo metodám instance.

Např. volání proměnné / metody přes referenční proměnnou (viz. příloha: Práce s objekty):

```
promenna = referencniPromenna.promennaInstance;
referencniPromenna.metodaInstance();
```

Ukonční (rušení) objetku

Jakmile objekt dokončí práci, na kterou byl vytvořen, nebo na objekt nejsou žádné reference v podobě referenčních proměnných (viz. kap.: Referenční proměnné), jsou mu odebrány zdroje (alokovaná paměť) a objekt je ukončen. Pomyslný úklid po objektu (např. rušení proměnných instance, které už nebudou potřeba), obstarává v Javě ***Garbage Collector*** *(česky popelář)***.** Bez Garbage Collectoru by mohlo hrozit, že běžící program by časem zabral všechny dostupné zdroje (paměť), které má a program by se zhroutil. Java neumožňuje přímé rušení objektů. Z bezpečnostního hlediska se jedná zcela jistě o výhodu. Mohlo by se totiž stát, že bychom zrušili objekt, který ještě neukončil svou činnost a došlo by k pádu programu. (14)

Objekt tedy zruší automaticky Garbage Collector v okamžiku, kdy na objekt neexistuje žádná reference v paměti.

Příklad zrušení reference na objekt (viz. příloha: Ukončení (rušení) objektu):

```
referencniPromenna = null;
```

Hodnota referenční proměnné ***null*** říká, že tato referenční proměnná neukazuje na žádný objekt. Pokud se jednalo o jedinou referenční proměnnou objektu, odebere Garbage Collector objektu přidělené zdroje (paměť z haldy) a tím objekt zruší. (14)

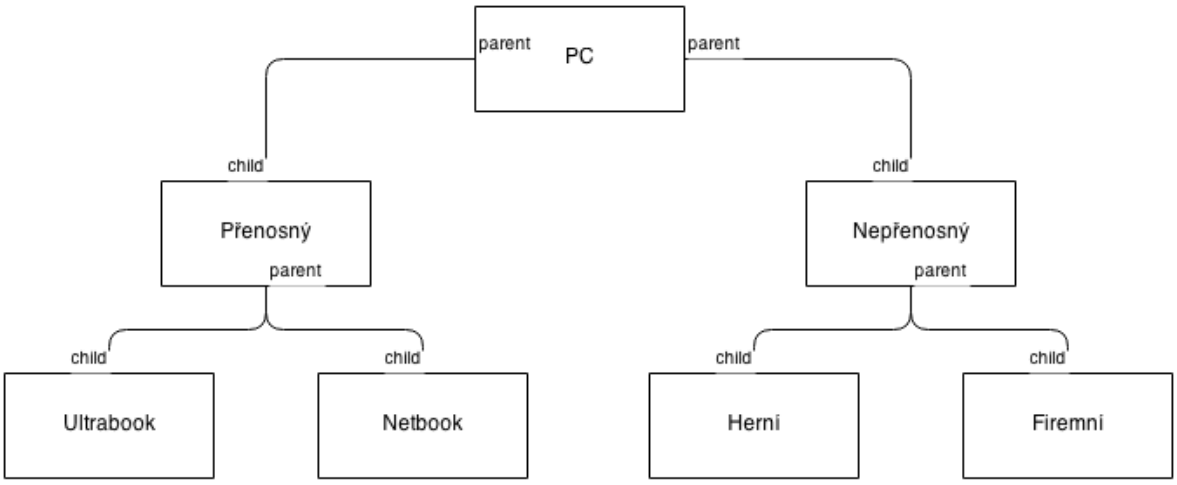
Jedná se však pouze o ukázku, která má přiblížit fungování Garbage Collectoru. V praxi se s podobným postupem téměř nesetkáváme.

2.2 DĚDIČNOST

Dědičnost je jednou z odvozených vlastností objektově orientovaného programování. Představuje možnost vybudování hierarchie tříd. Zavedení tříd v programování napomohlo definici nových objektů. Tato definice měla zabraňovat opakování stejného kódu. Stejně tak má dědičnost za úkol zabránit opakování částí kódu, které jsou pro třídy se stejnými prvky společné.

Jelikož jedna třída obvykle nestačí pro dostatečnou klasifikaci objektů, je běžnou praxí vytvoření třídy (tzv. rodiče), od které jiná třída (tzv. potomek neboli třída odvozená) zdědí vlastnosti. Odvozená třída zpravidla obsahuje kromě zděděných vlastností i další vlastnosti a tím se stává specializovanější, než její rodič.

Dědění nám tak dává možnost opakovaně využívat částí kódu rodiče u množiny instancí se stejnými specifickými vlastnostmi. Tím se stává program čitelnějším, méně rozsáhlým a zároveň se vyhýbáme případným chybám. V neposlední řadě dědičnost významně ulehčuje programátorovi jeho práci.



Obrázek 9: Demonstrace dědičnosti

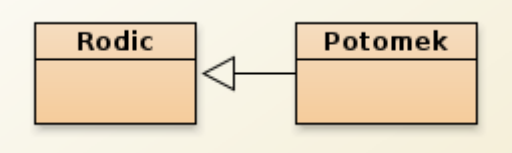
Z diagramu je patrná hierarchie dědění. Nejobecnější uzel diagramu je zároveň jeho kořenem. Tento uzel jako jediný nedědí žádné vlastnosti od jiné třídy. Čím dále je v diagramu uzel od kořene, tím více vlastností (metod) a proměnných, dědí od nadřazených uzlů. A to buď přímo od nejbližšího uzlu nad ním anebo nepřímo od ostatních uzlů, které jsou nad ním směrem ke kořeni diagramu. Dá se tedy říct, že nejníže položené uzly (uzly nejdále od kořene diagramu), jsou uzly nejvíce specializované.

2.2.1 REALIZACE DĚDIČNOSTI

Realizace dědičnosti probíhá pomocí klíčového slova ***extends***. V programovacím jazyce Java existuje takové omezení, že každá třída může dědit pouze od jedné další třídy. Tzn., že každý potomek může mít pouze jednoho přímého rodiče.

Příklad realizace dědičnosti (viz. příloha: Realizace dědičnosti):

```
public class Rodic{}
public class Potomek extends Rodic {}
```



Obrázek 10: Realizace dědičnosti

2.2.2 KLÍČOVÉ SLOVO SUPER

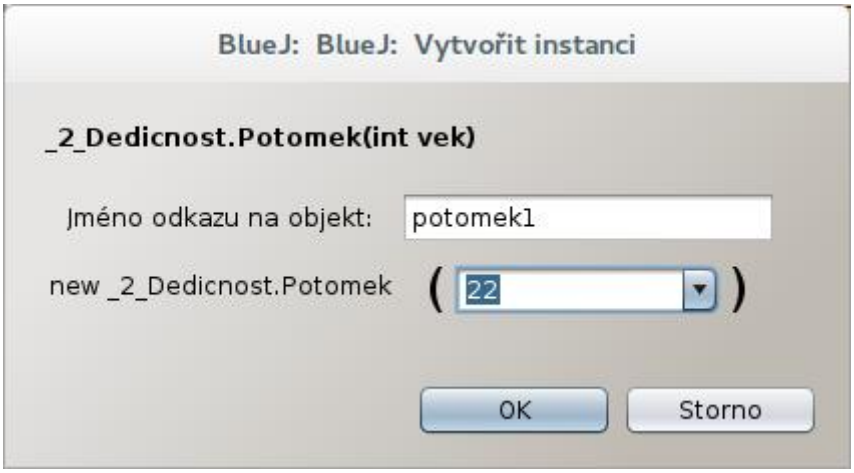
Klíčové slovo ***super*** má podobný princip použití jako slovo ***this***. Na rozdíl od ***this***, které odkazovalo na instanci současné třídy, můžeme slovem ***super*** odkazovat ze třídy potomka na proměnné, metody a konstruktory třídy rodiče. Souvisí tedy úzce s dědičností.

Nejčastěji tak realizujeme překrývání metod a konstruktorů (viz. kap.: Překrývání metod (Overriding)). V konstruktoru potomka ***super*** používáme k volání konstruktoru rodičovské třídy. Jelikož se v Javě konstruktory nedědí, je nutné konstruktory potomka vytvořit. Následně v něm však můžeme zavolat konstruktor rodiče a nemusíme tak znovu deklarovat inicializaci zděděných proměnných rodičovské třídy. Rozlišení rodičovských konstruktorů se provede pomocí zadaných parametrů při volání konstruktorů ve třídě potomka.

Příklad volání rodičovského konstruktoru pomocí slova ***super*** (viz. příloha: Realizace dědičnosti):

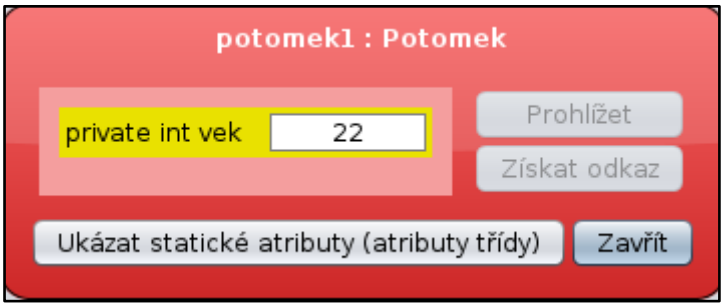
```
public class Rodic{
private int vek;
Rodic(int vek){
   this.vek = vek;
 }
}
public class Potomek exteds Rodic{
Potomek (int vek){
```

```
    super(vek);
  }
}
```



Obrázek 11: Vytvoření instance třídy Potomek. Předání parametru konstruktoru.

Při vytvoření objektu potomka mu předáme přes parametr ***vek*** hodnotu proměnné ***vek*** (např. 22), která je ale implementována v těle rodiče.



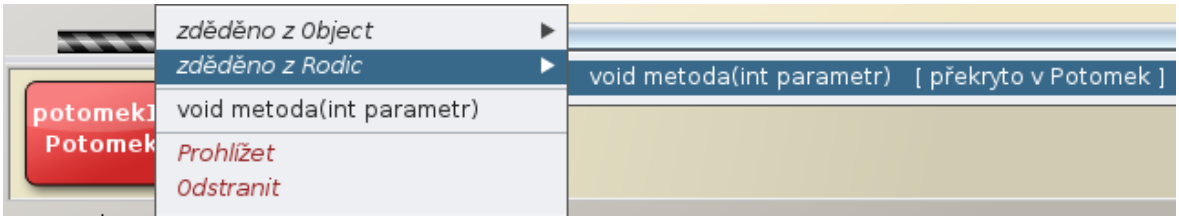
Obrázek 12: Hodnota "vek" uložená v objektu potomek1.

Super také často používáme při volání metod. Pokud má rodičovská třída implementovanou nějakou metodu a my chceme v potomkovi její chování rozšířit, můžeme zavolat zděděnou metodu pomocí klíčového slova ***super***. Jedná se o princip Překrývání metod (Overriding), popsaný v kapitole Polymorfismus.

Příklad volání rodičovské metody pomocí slova ***super*** (viz. příloha: Realizace dědičnosti):

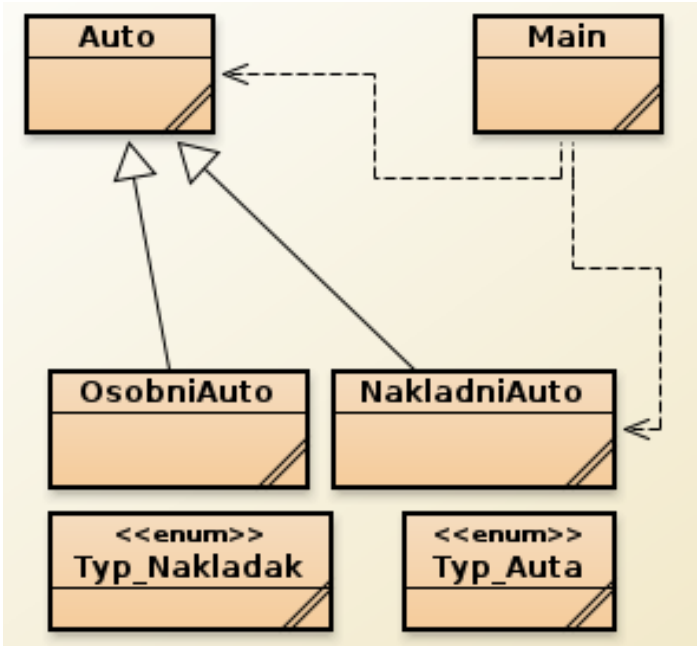
```
public class Rodic{
public void metoda(int parametr){
    //tělo metody
  }
}
public class Potomek extends Rodic{
public void metoda(int parametr){
    super.metoda(parametr);
```

```
    //rozšiřující kód metody
  }
}
```



Obrázek 13: Překrytá metoda, zděděná ze třídy Rodic.

2.2.3 DEMONSTRACE POUŽITÍ DĚDIČNOSTI A KLÍČOVÉHO SLOVA SUPER



Obrázek 14: Demonstra dědičnosti (UML)

- K demonstraci dědičnosti jsem vytvořil 4 třídy a dva výčtové typy. Třída ***Main*** má tři úkoly. Vytvoření instancí ostatních tříd (voláme v ní konstruktury). Zadání počátečních stavů těchto objektů a vypsání stavů objektů do konzole.
- Dalšími objekty v diagramu vývojového prostředí BlueJ jsou dva výčtové typy: ***Typ_Nakladak***, ***Typ_Auta***.
- Třída ***Auto*** je rodičovská třída. To znamená, že od ní třídy ***OsobniAuto*** a ***NakladniAuto*** dědí metody, konstruktory a proměnné.
- Třídy ***OsobniAuto***, ***NakladniAuto*** využívají konstruktoru rodiče. Klíčovým slovem ***„super“*** voláme konstruktor rodiče z konstruktorů potomků. Tímto způsobem využíváme proměnné rodiče, kterým předáváme hodnoty jako parametry konstruktoru.

Konstruktor rodičovské třídy ***Auto*** (viz. příloha: Demonstrace dědičnosti):

```
private String znacka, barva;
private int pocetDveri, pocetKol, pocetKilometru;
public Auto(String znacka, String barva, int pocetDveri, int pocetKol,
            int pocetKilometru){
   this.znacka = znacka;
   this.barva = barva;
   this.pocetDveri = pocetDveri;
   this.pocetKol = pocetKol;
   this.pocetKilometru = pocetKilometru;
}
```

Konstruktor třídy ***OsobniAuto*** (potomek) (viz. příloha: Demonstrace dědičnosti):

```
private int velikostKufru;
private Typ_Auta karoserie;
public OsobniAuto(String znacka, String barva, int pocetDveri, int
            pocetKol, int pocetKilometru, int velikostKufru, Typ_Auta
            karoserie){
   super(znacka, barva, pocetDveri, pocetKol, pocetKilometru);
   this.velikostKufru = velikostKufru;
   this.karoserie = karoserie;
}
```

Konstruktor třídy ***NakladniAuto*** (potomek) (viz. příloha: Demonstrace dědičnosti):

```
private Typ_Nakladak karoserie;
private int nosnost;
public NakladniAuto(String znacka, String barva, int pocetDveri, int
            pocetKol, int pocetKilometru, Typ_Nakladak karoserie, int
            nosnost){
   super(znacka, barva, pocetDveri, pocetKol, pocetKilometru);
   this.karoserie = karoserie;
   this.nosnost = nosnost;
}
```

- Po spuštění třídy ***Main***, se inicializují konstruktory tříd ***Auto***, ***OsobniAuto*** a ***NakladniAuto***, ve kterých předáme objektům počáteční hodnoty. Poté proběhne přes referenční proměnné objektů volání metod instancí (***vypisAuto***), které vypíší zformátovaně stav objektů do konzole.

Metoda ***main*** třídy ***Main*** (viz. příloha: Demonstrace dědičnosti)***:***

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

Auto car1 = new Auto("Škoda", "červená", 5, 4, 200000);
Auto car2 = new Auto("Mercedes", "černá", 4, 4, 120000);
OsobniAuto os = new OsobniAuto("Audi", "zelenáˇ", 5, 4, 250000, 52,
            Typ_Auta.LIFTBACK);
NakladniAuto nak = new NakladniAuto("Avia", "modrá", 2, 8, 10000,
            Typ_Nakladak.CISTERNA, 50000);


        car1.vypisAuto();
```

```
        car2.vypisAuto();
        os.vypisAuto();
        nak.vypisAuto();
    }
```

- Tyto metody u tříd potomků třídy ***Auto*** překrývají zděděnou metodu ***vypisAuto*** z rodičovské třídy ***Auto*** (viz. kap. Překrývání Překrývání metod (Overriding)).

2.2.4 ABSTRAKTNÍ METODY A TŘÍDY

Abstrakce jako taková, slouží ke zjednodušení řešeného problému. Zanedbává se při ní nepodstatných detailů a vlastností. Programovací jazyk Java využívá principu abstrakce při vytváření tříd a metod. Tyto třídy a metody nazýváme abstraktními. Ke zjednodušování dochází při jejich implementaci, kdy tělo třídy i těla metod jsou implementovány velmi obecně, nebo v případě metod nemusí být implementovány vůbec. (15)

Abstraktní třídy

Zvláštním druhem tříd jsou abstraktní třídy. U abstraktních tříd se nepočítá s vytvářením jejich instance (a ani to nelze), proto nemusí (ale může) implementovat chování metod. Abstraktní třídě není možné operátorem ***new*** vytvořit instanci. Metoda v abstraktní třídě mohou mít implementované tělo, které třída potomka pouze rozšiřuje, ale nejčastěji se tato implementace přenechává právě třídám, které od ní dědí (jsou jejími potomky) a tak abstraktní třída definuje pouze hlavičky metod (podobně jako rozhraní).

Na rozdíl od rozhraní ale některé základní metody implementovat může, a pokud potomek abstraktní třídy tyto metody nepředefinuje, pak se pro její instance použije implementace z abstraktní třídy.

Abstraktní třídy se uvozují modifikátorem ***abstract***.

Příklad vytvoření abstraktní třídy (viz. příloha: Vytváření abstraktní třídy):

```
public abstract class Trida{}
```

- **public** – specifikátor přístupu
- **astract** – modifikátor uvozující abstraktní třídu
- **class –** klíčové slovo uvozující třídu
- **Trida** – identifikátor třídy

Abstraktní metody

Abstraktní metody fungují na podobném principu, jako abstraktní třídy. Stejně jako třídy jsou metody uvozeny slovem ***abstract***. Na rozdíl od neabstraktních metod, které jsou implementované v těle abstraktní třídy, ale nesmí mít nadeklarovanou vnitřní strukturu (tělo).

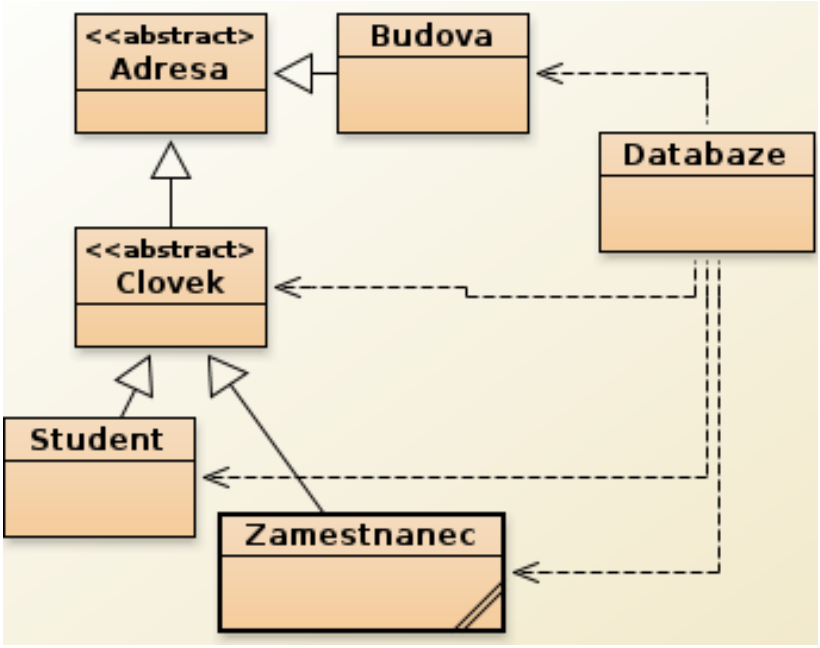
Využívají se v případě, kdy situace vyžaduje, aby odvozené třídy určité metodě vždy sami deklarovali tělo a rodičovskou abstraktní metodu touto metodou překryly (viz. kap.: Překrývání metod (Overriding)).

Příklad zápisu abstraktní metody (viz. příloha: Vytváření abstraktní třídy):

```
public abstract String metoda();
```

Demonstrace použití abstraktních tříd

(viz. příloha: Demonstrace použití abstraktních tříd)

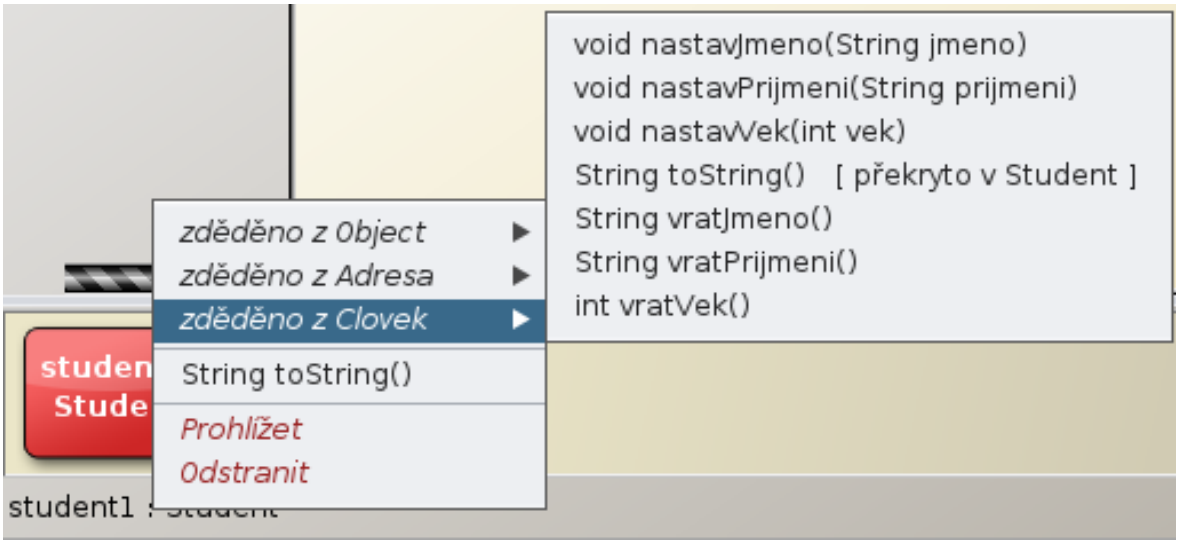


Obrázek 15: Demonstra použití abstraktních tříd (UML).

- Mějme šest tříd. Třídy ***Adresa***, ***Clovek***, ***Student***, ***Zamestnanec***, ***Budova*** a ***Databaze***. Budeme na ně nahlížet z pohledu databáze studentů, zaměstnanců a budov imaginární školy.
- U tříd ***Adresa*** a ***Clovek*** se nepočítá s vytvářením instancí, tudíž jsou nadefinovány jako abstraktní. V těchto třídách máme implementovanou datovou strukturu adresy a datovou strukturu základních informací o člověku (informací, které jsou

pro všechny lidi stejné). Z logického hlediska nemá opodstatnění vytvářet objekt třídy ***Adresa***, pokud není svázána s konkrétním člověkem a stejně tak nemá opodstatnění vytvářet objekt třídy ***Clovek***, která je nedostatečně specializovaná (příliš obecná).

- Třída ***Clovek***, tak představuje zobecněný vzor pro vytvoření objektu studenta a zaměstnance. Toto zobecnění je podstatou abstrakce.
- Z BlueJ UML je jasně patrná struktura závislostí mezi třídami. Třídy ***Clovek*** a ***Budova*** dědí od třídy ***Adresa***. Třídy ***Student*** a ***Zamestnanec*** dědí od třídy **Clovek**. Třída databáze používá třídy ***Clovek***, ***Student***, ***Zamestnanec*** a ***Budova***.
- Třída ***Databaze,*** obsahuje tři dynamická pole. Pro evidenci budov, studentů a zaměstnanců.
- Po vytvoření instance jedné ze tříd ***Student/Zamestnanec***, se dá demonstrovat, že data jako adresa, jmeno, prijmeni atd. jsou uložena v datové struktuře abstraktních tříd a lze je měnit jak přes překryté metody (viz. kap. Překrývání metod (Overriding)) abstraktních tříd, tak přes překrývající metody třídy jíž náleží instance.



Obrázek 16: Zděděné a překryté metody objektu student1.

2.3 ZAPOUZDŘENÍ

Zapouzdření je dalším z paradigmat objektově orientovaného programování. Používá se ke skrývání implementace objektu před jinými objekty. Jeho důsledkem je jistý druh autorizovaného přístupu k datům. Tento autorizovaný přístup zajišťuje, aby kód pracoval se správnými daty a aby nad nimi prováděl operace, které jsou k tomu určené. Zapouzdření také skrývá stav objektu (proměnné instance) a k jejich zpřístupnění dochází

pomocí metod.[7] Čím rozsáhlejší a složitější je kód, tím snadněji se může stát, že nedopatřením budeme chtít ovlivňovat chod části programu, kde to není vyžádáno. Data se snažíme zapouzdřit (chránit) před neautorizovaným přístupem deklarací proměnných, jejichž součástí jsou i přístupová práva k nim vyjádřená pomocí specifikátorů. (16)

Specifikátor	V téže třídě	V jiné třídě	V podtřídě téhož balíku	V podtřídě jiného balíku	V jiné třídě jiného balíku
Private	Ano	X	X	X	X
Neuvedeno	Ano	Ano	Ano	X	X
Protected	Ano	Ano	Ano	Ano	X
Public	Ano	Ano	Ano	Ano	Ano

Tabulka 1: Tabulka specifikátorů přístupu. (16)

2.3.1 PRIVATE

Nejvíce omezujícím specifikátorem je ***private***. Pokud je metoda, nebo proměnná, uvozena specifikátorem ***private***, je *„viditelná“* pouze ve třídě, ve které je nadefinovaná (ve které vznikla). Jakýkoliv jiný přístup z vnějšku třídy není možný. (16)

2.3.2 NEUVEDENO

Další způsob autorizace přístupu je ***neuvést*** žádný specifikátor při definování metody (proměnné). Pokud neuvedeme žádný specifikátor, metoda nebo proměnná jsou *„viditelní“* pro všechny třídy ve stejném balíku. Proto se tento autorizační přístup také nazývá jako (***package friendly***). Metody nebo proměnné, u kterých neuvedeme specifikátor přístupu, zároveň nejsou *„viditelné“* z odvozené třídy. (16)

2.3.3 PROTECTED

Specifikátor ***protected*** zajišťuje, že metoda nebo proměnná jsou *„viditelné“* z balíku a navíc i z potomka třídy (přes instanci potomka). Potomek třídy nemusí být ve stejném

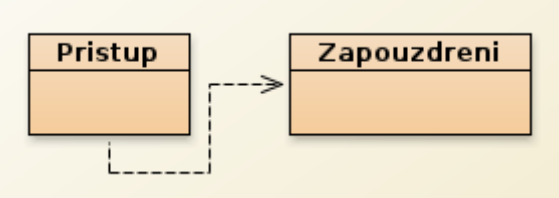
[7] ***Tzv. gettrů a settrů.*** *Zatímco gettry mají za úkol navracet hodnotu proměnných .Settry tuto hodnotu nastavují. Jedná se o speciální programové konstrukce, které mají za úkol chránit proměnné uvozené omezujícími specifikátory.*

balíku jako rodičovská třída. Tzn., že je mnohem méně chránící než např. specifikátor ***private***. (16)

2.3.4 PUBLIC

Nejméně „chránícím“ specifikátorem je specifikátor ***public***. Tento specifikátor říká, že metody a proměnné jím uvozené jsou volně přístupné komukoliv. Běžně se toto přístupové právo používá pro metody. U proměnných bychom si použití tohoto specifikátoru měli dopředu důkladně rozmyslet. (16)

2.3.5 DEMONSTRACE ZAPOUZDŘENÍ



Obrázek 17: Demonstra zapouzdření (UML).

Obrázek 18: Demonstrace zapouzdření 2 (UML).

- Máme vytvořeny tři třídy: ***Zapouzdreni***, ***Pristup***, ***Pristup2***
- Dvě třídy jsou umístěny ve stejném balíčku (balíček _3_Zapouzdreni), třetí třída je umístěna v balíčku nadřazeném
- Ve třídě ***Zapouzdreni***, jsou nadefinované proměnné s různými specifikátory přístupu

Třída Zapouzdření (viz. příloha: Demonstrace zapouzdření):

```
public class Zapouzdreni
{
    private String text_priv = "private";
    protected String text_prot = "protected";
    String text = "neuvedeno";
    public String text_publ = "public";

    public String getText(){
        return „Metoda getText() “+text_priv;
    }
```

```
    protected void vypisPromenne(){
        System.out.println( "Metoda vypisPromenne: "+"\n"
                           +text_priv+"\n"+
                           text_prot+"\n"+
                           text+"\n"+
                           text_publ+"\n");
    }}
```

- Třída ***Pristup*** demonstruje přístup k proměnným s různými specifikátory, které jsou zároveň umístěné ve stejném balíku. Pro přístup k těmto proměnným, musíme vytvořit instanci třídy ***Zapouzdreni***.

Třída Pristup (viz. příloha: Demonstrace zapouzdření):

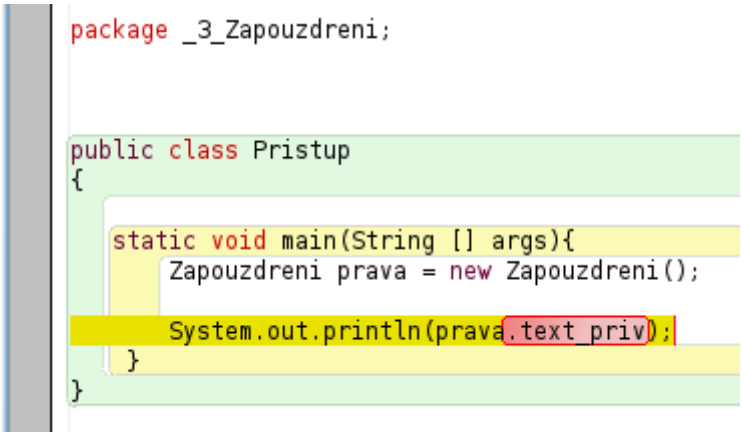
```
public class Pristup
{
   static void main(String [] args){
       Zapouzdreni prava = new Zapouzdreni();
      //instance třídy Zapouzdreni

       System.out.println(prava.getText());
       System.out.println(prava.text_prot);
       System.out.println(prava.text);
       System.out.println(prava.text_publ);
       prava.vypisPromenne();
    }
}
```

```
Metoda getText()private
protected
neuvedeno
public
Metoda vypisPromenne:
private
protected
neuvedeno
public
```

Obrázek 19: Výpis v konzoli, po spuštění metody main třídy Pristup.

- Je patrné, že třída ***Pristup*** má přes vytvořenou instanci třídy ***Zapouzdreni*** (***prava***) přístup k proměnným, které nejsou implementovány s omezujícím specifikátorem ***private***
- Pokud bychom se ve třídě ***Pristup*** pokusili přistoupit k proměnné, která je implementovaná se specifikátorem ***private***, došlo by k chybě při překladu



```
package _3_Zapouzdreni;


public class Pristup
{

    static void main(String [] args){
        Zapouzdreni prava = new Zapouzdreni();

        System.out.println(prava.text_priv);
    }
}
```

Obrázek 20: Demonstrace zapouzdření. Chyba při překladu.

```
text_priv has private access in _3_Zapouzdreni.Zapouzdreni
```

Obrázek 21: Demonstrace zapouzdření. Výpis chyby při překladu.

- K této proměnné však můžeme přistupovat přes metodu ***getText()***. Tímto postupem lze obejít omezující specifikátor ***private***. K proměnné tak máme přístup, ale nemůžeme přímo změnit její hodnotu. Pokud bychom chtěli měnit i její hodnotu, museli bychom nadefinovat metodu ***setText(String text)***, ve které bychom proměnné ***text_priv*** přidělovali hodnotu parametru ***text***.
- Na druhou stranu v třídě ***Pristup2*** máme přímý přístup, pouze k proměnným se specifikátorem ***public***. K proměnným, které nemají uvedený specifikátor, nemáme přístup, ani pokud je třída ***Pristup2*** potomkem třídy ***Zapouzdreni***. K proměnným se specifikátorem ***protected*** máme přístup pouze v případě, že třídy ***Pristup2*** je potomkem třídy ***Zapouzdreni*** a zároveň k nim přistupujeme přes instanci potomka (přes instanci rodiče k těmto proměnným ve třídě ***Pristup2*** přistupovat nelze).

Třída Pristup2 (viz. příloha: Demonstrace zapouzdření):

```
import _3_Zapouzdreni.Zapouzdreni;      //nezbytný import třídy z balíku

public class Pristup2 extends Zapouzdreni   //dědičnost
{
    public static void main (String [] args){
        Zapouzdreni prava = new Zapouzdreni();
        Pristup2 prava2 = new Pristup2();    //instance potomka

        System.out.println(prava.getText());     //public
        System.out.println(prava2.getText());    //public
        //System.out.println(prava.text_prot);   //protected - NELZE
        System.out.println(prava2.text_prot);    //protected
        //System.out.println(prava.text);        //neuvedeno - NELZE
        System.out.println(prava.text_publ);     //public
        //prava.vypisPromenne(); //protected - NELZE
        prava2.vypisPromenne();  //protected
    }
}
```

```
Metoda getText()private
Metoda getText()private
protected
public
Metoda vypisPromenne:
private
protected
neuvedeno
public
```

Obrázek 22: Výpis na konzoli po spuštění metody main třídy Pristup2.

2.4 POLYMORFISMUS

Polymorfismus je v každodenním životě velice běžný a my ho ani nevnímáme. Polymorfismem (česky mnohotvarostí) můžeme rozumět situaci, kdy určitý podmět (zpráva) vyvolá u různých lidí odlišné reakce. Každý na ni reaguje jiným chováním. (17)

Po přenesení do světa objektově orientovaného programování je to situace, kdy stejná metoda má u různých objektů jiný význam. Jedná se o zápis více definicí jedné metody a setkali jsme se s ním již při demonstraci použití slov ***this*** a ***super*** (viz. kap. Klíčové slovo super).

Podstatou jsou metody se stejnou hlavičkou, které mají definované třídy potomků. Tyto metody mohou potomci překrývat z rodičovské třídy (např. z Abstraktní třídy) nebo z implementovaného rozhraní. Každý potomek může mít definované jiné tělo této metody. Instance různých tříd, tedy na zavolání stejné metody reagují různě (podle kódu deklarovaného v těle volaných metod).

Polymorfismus se úzce vztahuje k dědičnosti (Abstraktní třídy) a k implementování rozhraní (viz. kap.: Rozhraní). Zároveň najde uplatnění i při vytváření objektů a to jmenovitě u konstruktorů. Přetěžování konstruktorů je totiž běžnou praxí.

2.4.1 ROZHRANÍ

Java nedovoluje vícenásobné dědění (dědění od více než jedné třídy najednou), což se dá považovat za nevýhodu. Tuto nevýhodu z části kompenzují rozhraní. Rozhraní je struktura obsahující deklarace konstant a hlaviček metod. To znamená, že obsahuje pouze popis

metod, ale neobsahuje implementaci jejich vnitřní struktury. Kromě metod může obsahovat také konstanty, které se chovají stejně jako by se jednalo o konstanty třídy. Rozhraní může také implementovat jiné rozhraní.

Každá třída může implementovat libovolný počet rozhraní, čímž se do jisté míry nahrazuje možnost vícenásobného dědění.

Vytváření rozhraní

Rozhraní se vytváří podobně jako třída. Místo klíčového slova ***class*** však konstrukce rozhraní obsahuje klíčové slovo ***interface***.

Příklad implementace rozhraní (viz. příloha: Implementace rozhraní) [8]:

```
public interface Rozhrani{
public String metodaRozhrani();
}
```

Implementace rozhraní

Použití rozhraní dává smysl v případech, kdy jsou patrné podobnosti mezi třídami. Jedním ze způsobů jak tyto podobnosti sjednotit, je vytvořit třídám rodiče (předka), od kterého zdědí vlastnosti. To však nemusí být vždy možné, nebo při nejmenším to může být obtížné.

V tomto případě je řešením rozhraní, které nám sjednotí hlavičky metod v třídách s podobnými vlastnostmi. Implementace metod rozhraní v jednotlivých třídách již může být odlišná.

K proměnným / konstantám rozhraní můžeme ve třídě, která ho implementuje, přistupovat dvěma způsoby:

- Prvním způsobem je, že k proměnným / konstantám přistupujeme přímo přes identifikátor rozhraní (**a**).
- Druhým způsobem je vytvoření instance rozhraní a následný přístup k proměnným / konstantám rozhraní přes referenční proměnnou rozhraní (**b**).

[8] *Specifikátor přístupu* ***public****, u metody* ***metodaRozhrani()****, není nutné v rozhraní uvádět. Všechny hlavičky metod obsažené v rozhraní jsou implicitně* ***public****.*[8]

U metod je tomu jinak. Metody musíme v těle třídy implementující rozhraní překrýt. Po překrytí těchto metod je poté voláme přes referenční proměnnou objektu vytvořeného z třídy implementující rozhraní (**c**).

Příklad implementování rozhraní (viz. příloha: Implementace rozhraní:

```
public interface Rozhrani
{
public final String KONSTANTA_ROZHRANI = "Ahoj";
public void metodaRozhrani();
}
public class Trida implements Rozhrani
{
public void metodaRozhrani(){
   System.out.println("Implementace metody rozhrani.");
}
public static void main (String[] args){
   System.out.println(Rozhrani.KONSTANTA_ROZHRANI); //a,
   Trida instance = new Trida();
   System.out.println(instance.KONSTANTA_ROZHRANI); //b,
   instance.metodaRozhrani(); //c,
 }
}
```

VÝSTUP PO SPUŠTĚNÍ MAIN:

```
Ahoj
Ahoj
Implementace metody rozhraní.
```

Implementace více rozhraní

Na rozdíl od dědičnosti může třída implementovat více rozhraní najednou. Třída implementující více rozhraní musí zároveň implementovat metody z obou rozhraní. Naopak proměnné a konstanty využívat může, ale nemusí.

Příklad implementování více rozhraní (viz. příloha: Implementace rozhraní):

```
public interface Rozhrani{}
public interface Rozhrani2{}
public class Trida implements Rozhrani, Rozhrani2{
}
```

2.4.2 PŘEKRÝVÁNÍ METOD (OVERRIDING)

Překrývání metod úzce souvisí s dědičností a klíčovým slovem super. V třídě potomka můžeme deklarovat metodu se stejnou hlavičkou jako má zděděná metoda rodičovské třídy. Dojde k překrytí rodičovské metody.

Překrývající metoda potomka zpravidla bývá specializovanější, popřípadě lépe vyhovující potřebám potomka. Nejedná se o předefinování ani přepsání překryté metody. Překrytá metoda může být kdykoliv použita jak v potomkovi, tak ve třídě, kde je implementována. Chceme-li v potomkovi použít překrytou metodu z rodičovské třídy, použijeme při jejím volání klíčové slovo ***super***.

Příklad překrývání metod (viz. příloha: Překrývání metod (overriding)):

```
public class Rodic{
public void metoda(){
   System.out.println(„Metoda rodiče.“);
 }
}
public class Potomek extends Rodic {
public void metoda(){
   System.out.println(„Metoda potomka“);
 }
public void vypisMetody(){
   super.metoda(); //překrytá metoda rodiče
   metoda(); //možno zapsat i takto this.metoda();
 }
public static void main(String[] args){
   Potomek potomek = new Potomek();
   potomek.vypisMetody();
 }
}
```

VÝSTUP PO SPUŠTĚNÍ MAIN:

```
Metoda rodiče.
Metoda potomka.
```

2.4.3 PŘETĚŽOVÁNÍ METOD (OVERLOADING)

Přetížené metody, jsou metody, které mají stejná jména, ale na rozdíl od překrytých metod mají jiné hlavičky (počet parametrů, jejich typy, typ návratové hodnoty). Přetížit můžeme jak metody instance, tak metody třídy. To, která metoda se při volání použije, se rozhoduje podle toho, u které metody souhlasí parametry a návratová hodnota s volanou metodou.

Např. (viz. příloha: Přetěžování metod (overloading)):

```
public static int obdelnik(int x, int y){
   return x s* y;
}
public static double obdelnik(double x, double y){
   return x * y;
}
```

```
public static void main (String[] args){
      System.out.println(Integer.toString(obdelnik(5,2)));
      System.out.println(Double.toString(obdelnik(5.1,2.1)));
}
```

VÝSTUP PO SPUŠTĚNÍ MAIN:

```
10
10.709999999999999
```

Podle toho, jestli metodu zavoláme s argumenty typu Integer / Double kompilátor vybere tu metodu, která vyhovuje argumentům.

ZÁVĚR

Zdrojů pojednávajících o objektově orientovaném programování, je opravdu mnoho. Ve své práci jsem se snažil čerpat ze zdrojů, které o této problematice pojednávají srozumitelně i pro úplné začátečníky.

Jak jsem již deklaroval v úvodu této práce, shrnout problematiku objektově orientovaného programování v jazyce Java komplexně, by množstvím dat několikanásobně přesahovalo zadaný rozsah bakalářské práce. Zaměřil jsem se tedy na podstatu toho, co dělá objektově orientované programování tak odlišným od ostatních programovacích stylů (programování strojového kódu, strukturované programování). Věřím, že se mi to povedlo.

Na druhou stranu paradigmata Javy, které jsem v této práci zmínil jen okrajově (ve zdrojovém kódu) nebo vůbec, považuji rovněž za důležité. Ať už to jsou: balíky (packages), výjimky (exceptions), vlákna (threads), datové struktury (kolekce (fronty, seznamy), pole (jednorozměrná, vícerozměrná), výčtové typy atd.) a tak dále.

Snažil jsem se minimalizovat míru abstrakce, která objektově orientované programování provází, na minimum tím, že jsem při prezentaci využíval grafického rozhraní vývojového prostředí BlueJ. Programy jsem odladil ve vývojovém prostředí Eclipse, protože obsahuje debugger. Utilitu umožňující ladění zdrojového kódu.

Funkci paradigmat jsem prezentoval na jednoduchých programech. Myslím si, že vytváření složitějších aplikací by mohlo být kontraproduktivní, vzhledem k účelu této bakalářské práce, která má sloužit jako výukový materiál. Tomu jsem přizpůsobil i výklad teoretické části, kde jsem kladl důraz na vysvětlení podstaty věci. Zároveň jsem přidal, v místech kde jsem to považoval za vhodné, obrázky vytvořené vektorovou grafikou v programu Inkscape. Struktura bakalářské práce byla vytvořena do logického celku, kdy v textu jsou zmiňovány souvislosti mezi jednotlivými kapitolami. Byly přidány křížové odkazy, ke snadnější navigaci v celé bakalářské práci.

Téma bakalářské práce mě bavilo a dovedu si představit, že bych se problematikou objektově orientovaného programování blíže zabýval i v dalším studiu.

RESUMÉ

This bachelor thesis presents basic paradigms OOP (Object-oriented programming) in Java. In the teoretical part are defined basic paradigms. In the practical part are samples of source code with screenshots from the IDE (Integrated Development environement) BlueJ.

OOP is popular type of programming. In OOP programmers can define data structures that can encapsulate data and functions that can be applied to data structure. OOP have this basic characteristics: Encapsulation, Polymorphism, Inheritance and creating Objects as well. Programmers can create relationships between two objects. They can create a new object that inherits a lot of functions from other object as well. This objects are reuseabe in other programs. It´s one of main advantage over procedural programming.

Java is object-oriented open source programming language, because allows creating objects. Java aplications are compiled to bytecode that can run on any platform of operating system (Windows, Linux, MacOS, Android) trought Java Virtual Machine. Java have got simple syntax (largely derived from C), so it´s suitable language for teaching object-oriented programming.

BlueJ is Java IDE used for teaching object-oriented programming style. BlueJ was created by the University of Kent and Deakin University from popular development enviroment NetBeans. BlueJ is highly interatcive. Graphical user interface in BlueJ is very user friendly and easy to use.

SEZNAM LITERATURY

1. **Novotný, Luděk.** Historie a vývoj jazyka Java. *Fakulta informatiky Masarykovy univerzity.* [Online] Masarykova univerzita, 2003. [Citace: 10.. Březen 2014.] http://www.fi.muni.cz/usr/jkucera/pv109/2003p/xnovtn8.htm.
2. **Herout, Pavel.** *Učebnice jazyka Java.* České Budějovice : KOPP, 2010. ISBN 978-80-7232-398-2.
3. **Pecinovský, Rudolf.** Klíčové vlastnosti Javy. *Myslíme objektově v jazyku Java.* Praha : Grada Publishing a.s., 2009.
4. **Roh, Štěpán.** Java a JIT. *Štěpán Roh - naživu leč ospalý.* [Online] 9.. prosinec 2003. [Citace: 27.. březen 2014.] alivebutsleepy.srnet.cz/java-a-jit.
5. **Oracle.** Java Platform Standard Edition 8 Documentation. *Oracle Java SE Documentation.* [Online] Oracle Corporation, 2014. [Citace: 8.. Duben 2014.] docs.oracle.com/javase/8/docs/index.html.
6. **Pecinovský, Rudolf.** 1.5 Vývojové prostředí BlueJ. [autor knihy] Pecinovský Rudolf. *Myslíme objektově v jazyku Java.* Praha : Grada Publishing a.s., 2009.
7. **Rejnková, Petra.** Příklady použití diagramů UML 2.0. *UML.* [Online] 2009. [Citace: 10.. Březen 2014.] uml.czweb.org.
8. **Pecinovský, Rudolf.** 1.2 Objektově orientované programování. [autor knihy] Pecinovský Rudolf. *Myslíme obejktově v jazyku Java.* Praha : Grapa Publishing a.s., 2009.
9. **Neumann, Antonín.** OOP. *Antonín Neumann.* [Online] 18.. prosinec 2013. [Citace: 11.. březen 2014.] home.zcu.cz/~neumann/data/KIV/OOP/oop-prednasky_2a5.pdf.
10. **Oracle Corporation.** Objects. *The Java Tutorials.* [Online] Oracle Corporation, 2014. [Citace: 12.. Březen 2014.] docs.oracle.com/javase/tutorial/java/javaOO/objects.html.
11. **Builder.cz.** Základní pojmy objektově orientovaného programování. *Builder Informační server o progrmaování.* [Online] Builder, 18.. prosinec 2000. [Citace: 12.. březen 2014.] http://www.builder.cz/rubriky/c/c--/zakladni-pojmy-objektove-orientovaneho-programovani-155665cz.
12. **Čápka, David.** 4. díl - Referenční a hodnotové datové typy. *devbook.cz - programátorská sociální síť.* [Online] 2012. [Citace: 1.. březen 2014.] http://www.devbook.cz/c-sharp-tutorial-referencni-a-hodnotove-tyoy-garbage-collector.
13. **Drbal, Pavel.** OOP 4 - Základní prvky a obraty. *Objekty - Objektová analýza, návrh a programování.* [Online] 13.. září 2005. [Citace: 3.. březen 2014.] http://objekty.vse.cz/Programovani/OopC4.
14. **Builder.cz.** Java - Garbage Collection. *Builder - Informační server o programování.* [Online] bulder.cz, 3.. červen 2002. [Citace: 4.. březen 2014.] www.builder.cz/rubriky/java/garbage-collection-156120cz.
15. **Beneš, Miroslav.** 2.5. Abstraktní třídy a metody. *Katedra inromatiky - Fakulta elektrotechniky a informatiky.* [Online] Vysoká škola báňská - Technidká univerzita Ostrava. [Citace: 18.. březen 2014.] www.cs.vsb.cz/benes/vyuka/upr/texty/java/ch01s05.html.

16. **Háka Software.** Java - zapouzdření a dědičnost. *Výukový portál Háka Software.* [Online] 2.. březen 2011. [Citace: 3.. duben 2014.] http://portal.haka-software.cz/index.php?option=com_remository&Itemid=12&func=startdown&id=6.
17. **Čápka, David.** Java tutorial - 7. díl Dědičnost a Polymorfismus. *devbook.cz - programátorská sociální síť.* [Online] [Citace: 27,. březen 2014.] www.debook.com/java-tutorial-dedicnost-a-polymorfismus.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

PŘÍLOHY

1. Vytváření třídy

Třída Trida.java

```
package _1_Objekty.Tridy;

public class Trida
{
    //tělo třídy
}
```

2. Metoda main

Třída Main.java

```
package _1_Objekty.Metody;

public class Main
{
    private static double a,b,vysledek;
    public static void main(String[] args)
    {
        try{
            a = Double.parseDouble(args[0]);
            b = Double.parseDouble(args[1]);
            vysledek = a / b;
            System.out.println(Double.toString(vysledek));
        }catch (Exception e){
            System.out.println("Vyskytla se chyba při dělení!");
            System.out.println(e.getMessage());
        }
}
}
```

3. Metody třídy / instance

Třída Metoda.java

```
package _1_Objekty.Metody;

public class Metody
{
private int pocet;
//deklarace proměnné, kterou vrací metoda getPocet();

        public static int soucet (int a, int b){    //Metoda třídy
            return a + b;
        }

        public int getPocet(){  //Metoda instance
            return pocet;
        }
}
```

4. Proměnné třídy / instance

Třída Promenne.java

```
package _1_Objekty.Promenne;

public class Promenne
{
	private static String promennaTridy = "Ahoj";

	private String promennaInstance;
}
```

5. Vytváření objektu třídy

Třída Reference.java

```
package _1_Objekty.Objekty;

public class Reference
{
    private String jmeno;
    private int vek;

    Reference(String jmeno, int vek){
        this.jmeno = jmeno;
        this.vek = vek;
    }

    private void vypisJmeno(){
        System.out.println("Jméno: "+jmeno);
        System.out.println("Věk: "+Integer.toString(vek));
        System.out.println("\n");
    }

    public static void main(String[] args){
        Reference referencniPromenna1 = new Reference("Adam Refer", 25);

        Reference referencniPromenna2 = new Reference("Petr Zeman", 48);

        System.out.println("================================"+"\n"+
                           "Reference 1:");
        referencniPromenna1.vypisJmeno();
        System.out.println("Reference 2:");
        referencniPromenna2.vypisJmeno();

        referencniPromenna1 = referencniPromenna2;

        System.out.println("Reference 1 po přiřazení:");
        referencniPromenna1.vypisJmeno();
        System.out.println("Reference 2 po přiřazení:");
        referencniPromenna2.vypisJmeno();

    }
}
```

6. Konstruktory – klíčové slovo „this“

Třída Trida.java

```
package _1_Objekty.Objekty;

public class Trida
{
    private int cislo, cislo2;
    private String retezec, retezec2;
    private boolean pravda;

    Trida(int cislo, String retezec, boolean pravda){
        this.cislo = cislo;
        this.retezec = retezec;
        this.pravda = pravda;
        System.out.println("Konstruktor 1");
    }

    Trida(int cislo, int cislo2, String retezec, String retezec2, boolean
            pravda){
        this(cislo,retezec,pravda);

        this.cislo2 = cislo2;
        this.retezec2 = retezec2;

        System.out.println("Konstruktor 2");
    }
}
```

7. Práce s objekty

Trida PraceSObjektem.java

```
package _1_Objekty.Objekty;

public class PraceSObjektem
{
    private int promennaInstance;  //deklarace proměnné instance

    private void metodaInstance(){} //deklarace metody instance

        public static void main (String[] args){
            PraceSObjektem referencniPromenna = new PraceSObjektem();
            //vytvoření referenční proměnné

            referencniPromenna.metodaInstance();
            //přístup k metodě instance

           int promenna = referencniPromenna.promennaInstance;
            //přístup k proměnné instance
        }
}
```

8. Ukončení (rušení) objektu

Třída RuseniObjektu.java

```
package _1_Objekty.Objekty;

public class RuseniObjektu
{
    public static void main (String [] args){
        RuseniObjektu referencniPromenna  = new RuseniObjektu(); //objekt

        referencniPromenna = null; //prázdná referenční proměnná
    }
}
```

9. Realizace dědičnosti

Třída Rodic.java

```
package _2_Dedicnost;

public class Rodic
{
    private int vek;

    Rodic (int vek){
        this.vek = vek;
    }

    public void metoda(int parametr){
        //tělo metody
    }
}
```

Třída Potomek.java

```
package _2_Dedicnost;

public class Potomek extends Rodic
{
    Potomek(int vek){
        super(vek);
    }

    public void metoda(int parametr){
        super.metoda(parametr);
        //rozšiřující kód metody
    }
}
```

10. Demonstrace dědičnosti

Třída Auto.java

```
package _2_Dedicnost;

public class Auto {
    protected String znacka;
    protected String barva;
    protected int pocetDveri;
    protected int pocetKol;
    protected int pocetKilometru;
    protected static int counter;

    public Auto(String znacka, String barva, int pocetDveri, int
            pocetKol, int pocetKilometru){
        this.znacka = znacka;
        this.barva = barva;
        this.pocetDveri = pocetDveri;
        this.pocetKol = pocetKol;
        this.pocetKilometru = pocetKilometru;
        counter ++;
    }

    public void vypisAuto(){

            System.out.println("=======================================
            ====");
        System.out.println("Automobil:    "+ znacka );
        System.out.println("Barva:        "+ barva);
        System.out.println("Pocet dveri: "+ pocetDveri);
        System.out.println("Pocet kol:    "+ pocetKol);
        System.out.println("Pocet najetych kilometru: "+ pocetKilometru);
        System.out.println("Pocet automobilu celkem: "+counter);
    }

    public String getZnacka(){
        return znacka;
    }

    public void setZnacka(String znacka){
        this.znacka = znacka;
    }

    public String getBarva(){
        return barva;
    }

    public void setBarva(String barva){
        this.barva = barva;
    }

    public int getPocetDveri(){
        return pocetDveri;
    }

    public void setPocetDveri(int pocetDveri){
        this.pocetDveri = pocetDveri;
    }
```

```
    public int getPocetKol(){
        return pocetKol;
    }

    public void setPocetKol(int pocetKol){
        this.pocetKol = pocetKol;
    }

    public int getPocetKilometru(){
        return pocetKilometru;
    }

    public void setPocetKilometru(int pocetKilometru){
        this.pocetKilometru = pocetKilometru;
    }
}
```

Třída OsobniAuto.java

```
package _2_Dedicnost;

public class OsobniAuto extends Auto {
    protected int velikostKufru;
    protected static int counter;
    protected Typ_Auta karoserie;

    public OsobniAuto(String znacka, String barva, int pocetDveri, int
            pocetKol, int pocetKilometru, int velikostKufru, Typ_Auta
            karoserie){
      super(znacka, barva, pocetDveri, pocetKol, pocetKilometru);
      this.velikostKufru = velikostKufru;
      this.karoserie = karoserie;
      counter ++;
    }

    @Override
    public void vypisAuto(){
        super.vypisAuto();
        System.out.println("Velikost kufru:  "+velikostKufru);
        System.out.println("Typ karoserie :  "+karoserie);
        System.out.println("Pocet OSOBNICH automobilu celkem: "+
            counter);

    }
}
```

Třída NakladniAuto.java

```
package _2_Dedicnost;

public class NakladniAuto extends Auto {
    protected Typ_Nakladak karoserie;
    protected int nosnost;
    protected static int counter;
```

```
public NakladniAuto(String znacka, String barva, int pocetDveri, int
            pocetKol, int pocetKilometru, Typ_Nakladak karoserie, int
            nosnost){
    super(znacka, barva, pocetDveri, pocetKol, pocetKilometru);
    this.karoserie = karoserie;
    this.nosnost = nosnost;
    counter++;
    }

    @Override
    public void vypisAuto(){
        super.vypisAuto();
        System.out.println("Karoserie :  "+karoserie);
        System.out.println("Nosnost   :  "+nosnost);
  System.out.println("Pocet NAKLADNICH automobilu celkem: "+counter);

    }
}
```

Výčtový typ Typ_Auta.java

```
package _2_Dedicnost;
public enum Typ_Auta {
COMBI, SEDAN, HATCHBACK, LIFTBACK, PICKUP, SUV, ROADSTER, OFFROAD,
            GRAN_TURISMO, COUPE,LIMUZINE, CABRIOLET, MPV;
}
```

Výčtový typ Typ_Nakladak.java

```
package _2_Dedicnost;

public enum Typ_Nakladak {
BAGR, AUTOJERAB, AUTOBAGR, CISTERNA, VYSOKOZDVIZNE_PLOSINY,
            HASICSKE_AUTO, KAMION;

}
```

Třída Main.java

```
package _2_Dedicnost;

public class Main {


    public static void main(String[] args) {

      Auto car1 = new Auto("Škoda", "červená", 5, 4, 200000);
      Auto car2 = new Auto("Mercedes", "černá", 4, 4, 120000);
      OsobniAuto os = new OsobniAuto("Audi", "zelená", 5, 4, 250000, 52,
            Typ_Auta.LIFTBACK);
      NakladniAuto nak = new NakladniAuto("Avia", "modrá", 2, 8, 10000,
            Typ_Nakladak.CISTERNA, 50000);
```

```
        car1.vypisAuto();
        car2.vypisAuto();
        os.vypisAuto();
        nak.vypisAuto();
    }
}
```

11. Vytváření abstraktní třídy

Abstraktní třída Trida.java

```
package _2_Dedicnost.Abstrakce;

public abstract class Trida
{

    public Trida()  //konstruktor abstraktní třídy
    {
    }

    public abstract String metoda();      //abstraktní metoda
}
```

12. Demonstrace použití abstraktních tříd

Abstraktní třída Adresa.java

```
package _2_Dedicnost.Abstrakce;

public abstract class Adresa
{
    String mesto;
    String ulice;
    int cp;


    public Adresa(String mesto, String ulice, int cp){
        this.mesto = mesto;
        this.ulice = ulice;
        this.cp = cp;
    }

    public String vratMesto(){
        return mesto;
    }

    public String vratUlice(){
        return ulice;
    }

    public int vratCp(){
        return cp;
    }

    public void nastavMesto(String mesto){
        this.mesto = mesto;
    }

    public void nastavUlici(String ulice){
        this.ulice = ulice;
    }

    public void nastavCp (int cp){
        this.cp= cp;
    }

    public String toString(){
        return "Město: "+mesto+"\n"+
               "Ulice: "+ulice+"\n"+
               "Čp: "+Integer.toString(cp)+"\n";
    }

}
```

Abstraktní třída Clovek.java

```
package _2_Dedicnost.Abstrakce;

public abstract class Clovek extends Adresa
{
    String jmeno;
```

```
    String prijmeni;
    int vek;

    Clovek (String jmeno, String prijmeni, String mesto, String ulice,
            int cp, int vek){
        super (mesto, ulice, cp);
        this.jmeno = jmeno;
        this.prijmeni = prijmeni;
        this.vek = vek;
    }

    public void nastavJmeno(String jmeno){
        this.jmeno = jmeno;
    }

    public void nastavPrijmeni(String prijmeni){
        this.prijmeni = prijmeni;
    }

    public void nastavVek (int vek){
        this.vek = vek;
    }

    public String vratJmeno(){
        return jmeno;
    }

    public String vratPrijmeni(){
        return prijmeni;
    }

    public int vratVek (){
        return vek;
    }

    public String toString(){
        return "Jmeno: "+jmeno+"\n" +
               "Prijmeni: "+prijmeni+"\n"+
               "Vek: "+Integer.toString(vek)+"\n"+
               super.toString();
    }

}
```

Třída Budova.java

```
package _2_Dedicnost.Abstrakce;

public class Budova extends Adresa
{
    int rozloha;
    int pocetPater;

    public Budova(String mesto, String ulice, int cp, int rozhloha, int
            pocetPater)
    {
     super(mesto,ulice,cp);
     this.rozloha= rozloha;
     this.pocetPater= pocetPater;
```

```
    }

    public String toString(){
        return super.toString()+
                "Rozloha: "+rozloha+"\n"+
                "PoÄŤet pater:"+pocetPater+"\n";
    }
}
```

Třída Student.java

```
package _2_Dedicnost.Abstrakce;

public class Student extends Clovek
{
    int rocnik;
    int kredity;
    public enum  Pohlavi{
        MUŽ,ŽENA;
    }
    Pohlavi pohlavi;

    public Student(String jmeno, String prijmeni, String mesto, String
            ulice, int cp, int vek, Pohlavi pohlavi, int rocnik, int
            kredity)
    {
       super (jmeno, prijmeni, mesto, ulice, cp, vek);
       this.pohlavi = pohlavi;
       this.rocnik = rocnik;
       this.kredity = kredity;
    }

    public String toString(){
        return super.toString()+
                "Pohlavi: "+pohlavi+"\n"+
                "Rocnik: "+Integer.toString(rocnik)+"\n"+
                "Kredity: "+Integer.toString(kredity)+"\n";
    }

}
```

Třída Zamestnanec.java

```
package _2_Dedicnost.Abstrakce;


public class Zamestnanec extends Clovek
{

    int cisloDveri;
    String specializace;

    public Zamestnanec(String jmeno, String prijmeni, String mesto,
            String ulice, int cp, int vek, int cisloDveri,
    String specializace)
    {
```

```
        super (jmeno, prijmeni, mesto, ulice, cp, vek);
        this.cisloDveri = cisloDveri;
        this.specializace = specializace;
    }

    public String toString(){
        return super.toString()+
                "Číslo dveří: "+Integer.toString(cisloDveri)+"\n"+
                "Specializace: "+specializace+"\n";
    }
}
```

Třída Databaze.java

```
package _2_Dedicnost.Abstrakce;


import java.util.ArrayList;
import java.util.Iterator;

public class Databaze
{
    private ArrayList student;
    private ArrayList budova;
    private ArrayList zamestnanec;

    public Databaze()
    {
        student = new ArrayList();
        budova = new ArrayList();
        zamestnanec = new ArrayList();
    }

    public void pridejStudenta(Student s){
        student.add(s);
    }

    public void pridejBudovu(Budova b){
        budova.add(b);
    }

    public void pridejZamestnance(Zamestnanec z){
        zamestnanec.add(z);
    }

    public void vypisData(){
        System.out.println("STUDENTI DATABAZE \n");
        for (Iterator i = student.iterator(); i.hasNext();){
            System.out.println(i.next());
        }

        System.out.println("STUDENTI ZAMESTNANCI \n");
        for (Iterator i = zamestnanec.iterator(); i.hasNext();){
            System.out.println(i.next());
        }

        System.out.println("BUDOVY DATABAZE \n");
        for (Iterator i = budova.iterator(); i.hasNext();){
```

```
            System.out.println(i.next());
        }
    }
}
```

13. Demonstrace zapouzdření

Třída Zapouzdreni.java

```
package _3_Zapouzdreni;

public class Zapouzdreni
{
    private String text_priv = "private";
    protected String text_prot = "protected";
    String text = "neuvedeno";
    public String text_publ = "public";

    public String getText(){

        return „Metoda getText() “+text_priv;
    }

    public void setText(String text_priv){
        this.text_priv = text_priv;
    }

    protected void vypisPromenne(){
        System.out.println( "Metoda vypisPromenne: "+"\n"
                            +text_priv+"\n"+
                            text_prot+"\n"+
                            text+"\n"+
                            text_publ+"\n");
    }
}
```

Třída Pristup.java

```
package _3_Zapouzdreni;

public class Pristup
{

   static void main(String [] args){
        Zapouzdreni prava = new Zapouzdreni();
        Pristup prava2 = new Pristup();

        System.out.println(prava.getText());
        System.out.println(prava.text_prot);
        System.out.println(prava.text);
        System.out.println(prava.text_publ);
        prava.vypisPromenne();
    }
}
```

Třída Pristup2.java

```
import _3_Zapouzdreni.Zapouzdreni;
```

```
public class Pristup2 extends Zapouzdreni
{
    public static void main (String [] args){
       Zapouzdreni prava = new Zapouzdreni();
       Pristup2 prava2 = new Pristup2();

       System.out.println(prava.getText()); //public
       System.out.println(prava2.getText()); //public
       //System.out.println(prava.text_prot);  //protected
       System.out.println(prava2.text_prot); //protected
       //System.out.println(prava.text);    //neuvedeno
       System.out.println(prava.text_publ); //public
       //prava.vypisPromenne();             //protected
       prava2.vypisPromenne();              //protected
    }
}
```

14. Implementace rozhraní

Rozhraní Rozhrani.java

```
package _4_Polymorfismus.Rozhrani;

public interface Rozhrani
{
    public final String KONSTANTA_ROZHRANI = "Ahoj";

    public void metodaRozhrani();
}
```

Rozhraní Rozhrani2.java

```
package _4_Polymorfismus.Rozhrani;

public interface Rozhrani2
{
}
```

Třída Trida.java

```
package _4_Polymorfismus.Rozhrani;

public class Trida implements Rozhrani, Rozhrani2
{
    public void metodaRozhrani(){
        System.out.println("Implementace metody rozhraní.");
    }

    public static void main (String[] args){
        System.out.println(Rozhrani.KONSTANTA_ROZHRANI);

        Trida instance = new Trida();

        System.out.println(instance.KONSTANTA_ROZHRANI);
        instance.metodaRozhrani();
    }
}
```

15. Přetěžování metod (overloading)

Třída Trida.java

```
package _4_Polymorfismus.Pretezovani_Metod;

public class Trida
{
    public static int obdelnik(int x, int y){
        return x * y;
    }
    public static double obdelnik(double x, double y){
        return x * y;
    }
    public static void main (String[] args){
        System.out.println(Integer.toString(obdelnik(5,2)));

        System.out.println(Double.toString(obdelnik(5.1,2.1)));
    }
}
```

16. Překrývání metod (overriding)

Třída Rodic.java

```
package _4_Polymorfismus.Prekryvani_Metod;

public class Rodic
{
    public void metoda(){
        System.out.println("Metoda Rodiče.");
    }
}
```

Třída Potomek.java

```
package _4_Polymorfismus.Prekryvani_Metod;

public class Potomek extends Rodic
{

    public void metoda(){
        System.out.println("Metoda Potomka.");
    }

    public void vypisMetody(){
        super.metoda();
        metoda();
    }

    public static void main(String[] args){
        Potomek potomek = new Potomek();

        potomek.vypisMetody();
    }
}
```