Úvod

Tento dokument obsahuje zhrnutie základných termínov a bol vytváraný od zimného semestra akademického roka 2012/2013 (do termínu uvedeného v hlavičke), spočiatku pre predmety programovanie a algoritmy 1, 2 a propedeutika programovania 1, 2, potom pre predmety objektovo orientované programovanie 1, 2. (Plán jeho ďalšieho rozvoja je tiež naznačený v hlavičke tejto strany.)

Tento dokument bol v mnohom od začiatku chápaný ako štartovacia línia. Niektoré vysvetlenia zámerne neboli uvádzané technicky úplne presne (prílišná presnosť je v začiatkoch škodlivá), preto (a nielen preto) je nevyhnutné, aby študujúci vo vlastnom záujme sebarozvoja postupným štúdiom získaval presnejšie predstavy o termínoch a konceptoch v programovaní.

Význam a spôsob budovania pevných základov (v súvislosti s programovaním)

Prvým krokom v začiatkoch programovania je porozumenie terminológii, ktorá sa pri tom používa. Ovládanie základnej terminológie (vysvetlenej aj v tomto dokumente) umožňuje v budúcnosti študentovi prispôsobenie sa novej a hlbšie porozumenie známej terminológie.

Terminológia je dorozumievacím jazykom. Bez ovládania významu jednotlivých termínov nie je možné ďalej napredovať, pretože ich používanie je aj pri vysvetľovaní nevyhnutné. (Sami uvidíte jej význam. Stačí sa pozrieť na to, koľko slov je potrebných na opis a vysvetlenie významu jednotlivých termínov, ktoré sú často samé o sebe jednoslovné…)

Ďalším krokom pri programovaní je syntax. Syntax je pravopis. Podobne ako je terminológia dorozumievacím jazykom medzi odborníkmi (v našom prípade programátormi), je syntax programovacieho jazyka základom na tvorbu jedných z najdôležitejších „písomností“ v oblasti programovania – zdrojových kódov (programov).

Ovládanie syntaxe samotnej nie je priamo dôležité na porozumenie princípom programovania, ale je **nevyhnutné** na začatie práce s programovacím prostredím – na začatie programovania. S pomocou programovacieho prostredia a programovacieho jazyka dostáva začínajúci programátor príležitosť všetkému porozumieť, ale musí vedieť túto príležitosť využiť.

Čiže akceptovanie a porozumenie syntaxe nevedie priamo k porozumeniu, ale je to nevyhnutný krok na to, aby mohol študent (či už skúsený alebo nováčik) v prvom rade *začať pracovať* a neskôr *chápať*, pretože programovaniu je najlepšie možné porozumieť prácou – programovaním.

Ďalšie informácie súvisiace s obsahom tejto podkapitoly sú uvedené v kapitolách Najčastejšie (prípadne najzávažnejšie) omyly vyskytujúce sa počas semestrov 2012 – 2014 (na strane 22, pričom viaceré chyby sa opakovali aj nad rámec spomínaných semestrov) a Často kladené otázky (na strane 21).

Zdrojový kód, kompilácia, prenositeľný kód (bytecode) a strojový kód

Termín **zdrojový kód** označuje text programu zapísaný podľa pravidiel zápisu (syntaxe) programovacieho jazyka tak, aby mu rozumel počítač aj človek. Pre programátora je dôležité vyznať sa v zdrojovom kóde, počítač si vie zdrojový kód preložiť do svojho jazyka. Bez prekladu by počítač nedokázal so zdrojovým kódom zaobchádzať, pretože pre neho je prirodzený **strojový kód**. Strojový kód je binárny kód, je to zároveň kód symbolizujúci inštrukcie počítača. **Inštrukcie** sú elementárne príkazy počítača. Sú to tie najjednoduchšie príkazy, ktoré počítač spracúva (jednoduchšie už nejestvujú). Čiže **strojový kód** je sled jednotiek a núl (t. j. binárny kód), ktoré sú použité vo význame jednoduchých príkazov nazývaných **inštrukcie**. Tým už počítač rozumie a je schopný ich vykonávať.

Strojový kód je vytvorený zo **zdrojového kódu** špeciálnym programom nazývaným **prekladač** alebo **kompilátor**. Proces tvorby strojového kódu sa nazýva **preklad** alebo **kompilácia**. Programovacie jazyky, pre ktoré je charakteristická tvorba strojového kódu, sa nazývajú **kompilované programovacie jazyky**. Jestvujú programovacie jazyky, pre ktoré je charakteristické, že neprodukujú strojový kód. Namiesto toho v reálnom čase (priamo počas vykonávania programu) spracúvajú, *interpretujú*, riadky zdrojového kódu a podľa toho sa nazývajú **interpretované programovacie jazyky**. Java je programovací jazyk, pre ktorý je tiež charakteristické, že neprodukuje strojový kód, napriek tomu nie je interpretovaným programovacím jazykom! Pred spustením programu napísaného v Jave prebieha proces prekladu, avšak výsledkom nie je ani strojový kód (?!), ale prenositeľný kód nazývaný **bytecode** ([:bajtkód:]).

Takáto situácia nastala kvôli (alebo vďaka) tomu, že Java je multiplatformovým programovacím jazykom. To znamená, že preložený program je spustiteľný na viacerých platformách. Na fungovanie preložených Java programov je spravidla potrebný **virtuálny stroj** (čo je vlastne „počítač simulovaný počítačom“), v súčasnosti však jestvujú procesory schopné spracúvať aj Java bytecode (bez nutnosti „simulátora“ – virtuálneho stroja)…

Triedy, inštancie a metódy

Prvými termínmi, s ktorými sa pri objektovo orientovanom programovaní (Java je tiež objektovo orientovaný jazyk) stretneme, sú **trieda** a **inštancia** (alebo **objekt**). **Trieda** je všeobecný predpis. Obsahuje napríklad informácie o tom, aký typ údajov sa bude spracúvať a ako sa s nimi bude zaobchádzať… Z toho čiastočne vyplýva, že trieda je sama o sebe nefunkčná (rôzne výnimky a bližšie detaily zatiaľ úmyselne zamlčiavame). Na to, aby sa to, čo definujeme v triede, stalo „skutočnosťou“, potrebujeme **inštanciu**.

Príkladom **triedy** môže byť všeobecný pojem z bežného života „osoba“. Keď niekto bez určenia bližšieho detailu povie „osoba“, v predstave sa nám zrejme vynorí neforemná postava. Pod týmto slovom rozumieme všetky osoby, vieme si predstaviť podstatu takej osoby, jej atribúty: bežné správanie a podobne. Avšak dokedy nezačneme hovoriť o konkrétnom človeku (konkrétnej osobe), je a vždy to bude len všeobecný pojem…

Keď si spomenieme na konkrétnu osobu, ktorú dobre poznáme, poznáme jej meno, správanie, smiech atď. nadobudne naša predstava úplne iný rozmer. Tie konkrétne osoby nazývame v Jave **inštanciami** (prípadne objektmi – z toho pochádza aj termín „objektovo orientované programovanie“ – avšak „objekt“ je širší pojem, ktorým môžeme označiť aj to, čo nie je inštanciou, preto je najmä v písanom texte lepšie používať termín „inštancia“).

Uveďme nevyhnutný základ syntaxe na definovanie tried a vytváranie inštancií v jazyku Java. Úplne prázdna trieda vyzerá takto:

```
class Osoba
{
}
```

Príkaz na vytvorenie anonymnej inštancie triedy osoba vyzerá takto:

```
new Osoba();
```

Nie je obvyklé vytvárať anonymné inštancie. Bežnejšie je vytváranie pomenovaných inštancií, ktorých definícia vyzerá takto:

```
Osoba peter = new Osoba();
```

(Poznámka 1: Uvedenie prázdnej zátvorky je v Jave nevyhnutné. Vyjadruje to, že ide o základný spôsob konštrukcie bez tzv. „argumentov“, ktoré by vypovedali o ďalších detailoch o osobe.)

(Poznámka 2: Určite ste si všimli, že meno osoby je napísané s malým písmenom. Nie je to omyl. O konvenciách pomenovania tried, inštancií, metód, premenných a pod. si povieme neskôr.)

Žiadny príkaz v Jave nemôžeme napísať osamotene. Každý príkaz musí byť obsiahnutý vo vyhradenom priestore: v **bloku** vo vnútri triedy, najčastejšie v takzvanej **metóde**. Blok v Jave zapisujeme pomocou zložených zátvoriek `{ }` (obvykle každú na samostatnom riadku, pričom medzi nich vkladáme riadky kódu).

Jestvuje jedna špeciálna (rezervovaná) metóda, ktorá je Javou považovaná za vstupný bod programu. Nazýva sa hlavná (`main`), môže byť definovaná v ľubovoľnej triede (ktorú podľa toho zvykneme nazývať tiež hlavnou) a musí byť uvedená v tomto tvare (niektoré detaily je možné mierne pozmeniť, ale to teraz nie je dôležité):

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

Základná definícia plne funkčnej hlavnej triedy by mohla vyzerať napríklad takto (aj s ukážkou príkazu na výpis textu „Ahoj!“):

```
class HlavnáTrieda
{
    public static void main(String[] args)
    {
        System.out.println("Ahoj!");
    }
}
```

Rozdiel medzi *premennými* a *atribútmi*

Táto kapitola **ne***definuje* termíny **premenná** a **atribút**. Iba *vysvetľuje rozdiely* medzi *súvisiacimi termínmi*. Pri prenášaní terminológie Javy (používanej aj v nástroji BlueJ) do slovenského jazyka nastal menší problém. V anglickom jazyku sa na súhrnné označenie *inštančných a statických premenných tried* používa termín *„field“*. Doslovným prekladom (*pole*) by sme spôsobili významnú nejednoznačnosť terminológie, pretože termín *„pole“* je v programátorskej terminológii slovenského jazyka dlhodobo používaný v inom význame (a síce vo význame, v ktorom je zase spätne v anglickom jazyku používaný termín *„array“*). Museli sme zvoliť iný výraz pre anglické *„field“*. Priklonili sme sa k terminológii používanej v českom jazyku a anglické *„field“* prekladáme ako *„atribút“*. Pre úplnosť, termínom **premenná** označujeme všetky druhy *premenných*, nielen *inštančné* a *statické* (t. j. z množiny tých, z ktorými prichádzame do kontaktu, sú to aj rôzne druhy *premenných s obmedzenou platnosťou*, napr. *lokálne premenné* a *parametre*).

Komentáre

Použitie komentárov je dôležitejšie pre programátorov, menej pre počítač. (Najmä ak berieme do úvahy komentáre bližšie vysvetľujúce funkcionalitu kódu.) Dobre komentovaný kód je ľahšie pochopiteľný inými programátormi, ale aj samotným autorom programu, pretože s odstupom času sa v kóde (hoci napísanom samotným autorom) o niečo ťažšie orientuje.

Okrem vysvetľovania zdrojového kódu majú komentáre aj niekoľko ďalších použití. Napríklad: dočasné vypnutie niektorej časti programu (veľmi užitočné pri ladení, t. j. hľadaní chýb v programe), vloženie pomocných (zvýraznených) informácií pre programátora (také sú aj v šablónach BlueJ), vloženie navigačných bodov na rýchlejšiu orientáciu v dlhých zdrojových kódoch, generovanie dokumentácie a podobne.

S použitím nástroja BlueJ rozlišujeme v Jave nasledujúce typy komentárov:

```
// Jednoriadkový komentár

/*
```

```
 * Viacriadkový komentár
 */

/**
 * Komentár dokumentácie (opisy tried, konštruktorov, metód…). Okrem samotného
 * opisu, obsahuje aj informácie o vstupných parametroch metód, návratovej
 * hodnote a iné.
 *
 *   @param  parameter  opis parametra
 *   @return opis návratovej hodnoty
 */

/*# Zvýraznený BlueJ komentár. */
```

Operácia, operátor

Rozdelenie podľa vykonávanej operácie (účelu)

Vyberáme často používané operátory…

Termín	Opis	Príklady
Aritmetický operátor	používaný na (dobre známe „klasické“) aritmetické operácie	`+ - * /`
Logický operátor	používaný na logické (pravdivostné) operácie	`&& || !`
Relačný operátor alebo operátor zhody	používaný na porovnávanie operandov alebo zistenie zhody	`<= >= != ==`
Priraďovací operátor	používaný na priradenie hodnoty alebo priradenie hodnoty so zmenou	`= += *= /=`
Iný operátor („špeciálny“)	umelo vytvorená zjednodušujúca kategória zahŕňajúca tie operátory, ktoré nemožno zahrnúť do predchádzajúcich kategórií	`new (typ)`

Z množstva ďalších, ktoré sme nespomínali, by sme mohli vybrať napríklad bitové operátory alebo operátory bitového posunu… Prehľad (v anglickom jazyku) môžete napríklad nájsť na: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html.

Rozdelenie podľa počtu prijímaných operandov

Toto rozdelenie v skutočnosti hovorí o vlastnosti operátorov, ktorá vyjadruje množstvo vstupných hodnôt (operandov), ktoré daný operátor vyžaduje na to, aby mohol vykonať danú operáciu.

Termín	Vysvetlenie	Príklady
Unárny	pracuje s jedným operandom	`-a` `!x` `--i`
Binárny	pracuje s dvoma operandmi	`a + b` `c && d` `x / y`
Ternárny	pracuje s troma operandmi	*výraz1* `?` *výraz2* `:` *výraz3* (v Jave poznáme len tento jeden ternárny operátor)

	`c = (a < b) ? a : b;`
	funguje ako
	`if (a < b) c = a; else c = b;`	

Operátor `new` je unárny. Operandom tohto špeciálneho operátora je konštruktor. Konštruktor môže prijímať viacero argumentov. Spojením oboch informácií vznikne nasledujúci zápis: `new Farba(0, 0, 0)` – v zátvorke sú uvedené argumenty konštruktora (samotný operátor `new` o nich „nevie“).

Rozdelenie podľa umiestnenia voči operandom

Toto rozdelenie súvisí so spôsobom zápisu operátora voči operandom. V rôznych situáciách (mimo Javy) sa môžeme stretnúť aj s takým spôsobom zápisu, kedy operátor uvádzame pred (napríklad: + 1 2) alebo za (napríklad: 1 2 +) zoznam všetkých operandov. Tieto spôsoby zápisu sa nazývajú *prefixový* a *postfixový*. Najmä v súvislosti s „klasickou" aritmetikou je to pre nás neštandardné. Nemusíme sa však obávať, práve pre tieto operátory používa Java *infixový* spôsob zápisu, na ktorý sme všetci zvyknutí.

Termín	Vysvetlenie	Príklady (v Jave)	Príklad iného použitia (mimo Javy)
Prefixový	je uvedený pred operandmi	`!c` `-a` `--c`	/ a b
Infixový	je uvedený medzi operandmi	`a < b` `a + b`	(v Jave je podobne ako pri „klasickej" aritmetike väčšina dostupných operátorov infixových, skôr tie prefixové a postfixové sú výnimkami)
Postfixový	je uvedený za operandmi	`i++` `x--`	c d *

(Vyššie uvedené rozdelenia by sme pri úplnom opise daného operátora mali vedieť správne skombinovať! Väčšinou sa stretávame s rozdeľovaním operátorov podľa účelu, netreba však zabúdať aj na ostatné vlastnosti operátorov, vďaka ktorým môžeme napríklad povedať, že `+` je v Jave binárnym infixovým aritmetickým operátorom, podobne `!` je unárnym prefixovým logickým operátorom a tak ďalej…)

Údajový typ

Údajové typy sú nutné na stanovenie typu (druhu) údajov, s ktorým chceme pracovať. Vyskytujú sa pri definíciách a deklaráciách (tieto termíny vysvetľujeme neskôr) premenných, konštánt, metód, pri pretypovaní a podobne.

Množina	Príklady
Primitívne údajové typy	`int` `double` `boolean`
Objektové údajové typy	
— ekvivalenty primitívnych typov	`Integer` `Double` `Boolean`
— objektové údajové typy	`String` `GRobot` `Farba`
— „zložené" (parametrické) údajové typy	`Zoznam<Typ>`

Podrobnosti o primitívnych údajových typoch v anglickom jazyku nájdete na: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

Parametrický údajový typ nie je možné v Jave kombinovať s primitívnym údajovým typom, čiže nemôžeme vytvoriť niečo takéto:

```
Zoznam<int>
```

Namiesto toho musíme použiť:

```
Zoznam<Integer>
```

Typické použitia údajových typov sú napríklad:

```
int a = 10;                  // definícia celočíselnej premennej
public int výpočet(int x)    // definícia metódy s jedným argumentom
{
    return (x * x) / 2;
}
```

Identifikátor

Ide v podstate o akýkoľvek názov (premennej, konštanty, metódy, triedy…).

Obmedzenia

Opis	Príklad(y)	Význam príkladov (v Jave)

Smie obsahovať iba povolené znaky (veľké a malé písmená abecied rôznych národností, číslice, znak spodnej čiary _ a znak dolára $ prípadne inej meny)	`početKusov`
	`číslo`
	`Lopta`
	`$α`
	`__konštanta`
	`Δ`
		je šesť platných identifikátorov

Na prvom mieste nesmie byť číslica	`32bitov` `6tyKus`	nie sú platné identifikátory
Nesmie obsahovať medzery	`počet kusov`	nie je platným identifikátorom (mohli by sme ich považovať za dva samostatné identifikátory, ale pri uvedení tesne za sebou by kompilátor Javy ohlásil chybu)

Sú citlivé na veľkosť písmen	`Albert`
	`Albert`
	`ALBERT`
		sú tri rôzne identifikátory

Stojí za povšimnutie…

Aký je rozdiel medzi zápisom identifikátora `veľkosť` a `veľkosť()`?

`veľkosť` – takto zapísaný identifikátor môže označovať premennú, konštantu (prípadne iné objekty)

`veľkosť()` – takto zapísaný identifikátor označuje metódu (prípadne konštruktor)

Čiže podľa toho, či sú za identifikátorom uvedené zátvorky, dokáže Java rozlíšiť dva základné prípady – či ide o identifikátor metódy (čiže treba niečo spustiť) alebo identifikátor premennej, konštanty, parametra (čiže pracuje sa s jej/jeho hodnotou).

Na úrovni triedy je dovolené definovať inštančnú premennú (atribút) a metódu s identickými názvami. V takom prípade je prítomnosť zátvoriek na ich rozlíšenie kľúčová! (Pravdou je, že takéto definície nie sú v praxi odporúčané práve preto, že sú mätúce.)

Príklad

Definujme v rámci jednej triedy atribút a metódu s rovnakým menom:

```
int rovnakéMeno = 11;

public int rovnakéMeno()
{
    return 12;
}
```

Ak za týchto okolností definujeme premenné `a` a `b` takto:

```
int a = rovnakéMeno;
int b = rovnakéMeno();
```

tak v premennej `a` bude hodnota `11` (skopírovaná z celočíselnej inštančnej premennej `rovnakéMeno`) a v premennej `b` bude hodnota `12` (vrátená metódou `rovnakéMeno`).

Konvencie tvorby identifikátorov v Jave

Okrem predpísaných pravidiel nutných na zostavenie korektného identifikátora je daná množina odporúčaných pravidiel, ktoré sú dodržované štandardnými komponentmi Javy a ktoré by mal dodržiavať každý programátor v jazyku Java. Týkajú sa tvaru v akom zapisujeme identifikátory určujúce rozdielne prvky programu.

Identifikátor býva často zložený z viacerých slov. Keďže nemáme možnosť použiť na oddelenie slov medzery, zvyknú sa slová oddeľovať veľkým začiatočným písmenom v každom slove, prípadne znakom spodnej čiary _, avšak tá sa používa len v prípade, že sa identifikátor skladá rýdzo z veľkých písmen. Veľké písmená sa používajú na konštanty (*vo verzii skupiny tried grafického robota, ktorú sme používali počas prvého semestra, bola táto konvencia porušená*), triedy (a tzv. rozhrania) začínajú veľkým písmenom, pokračujú malými (pričom každé ďalšie slovo začína veľkým), všetko ostatné (premenné, metódy…) začínajú malými písmenami (opäť s veľkým písmenom v každom ďalšom slove). Podrobnosti a názorné príklady rôznych druhov identifikátorov zhŕňame v nasledujúcej tabuľke:

	Opis	Príklad
Konštanta	názvy konštánt píšeme veľkými písmenami, slová oddeľujeme znakmi spodnej čiary	`POČET_PRVKOV KONIEC`
Trieda	názvy tried (a rozhraní) zapisujeme s veľkým počiatočným písmenom, zvyšok sa riadi pravidlom oddeľovania slov veľkým písmenom	`HlavnáTrieda Osoba Plátno`
Premenná, metóda…	ostatné identifikátory zapisujeme s malým počiatočným písmenom, zvyšok sa riadi pravidlom oddeľovania slov veľkým písmenom	`main dopredu otočVľavo celéČíslo`

Literál

Hodnoty zapísané priamo v programe označujeme termínom literál (voľne poslovenčené „doslovník“ od slova „doslovný“, z anglického literal = doslovný; ako pomôcka na zapamätanie významu termínu „literál“ môže poslúžiť nasledujúce prirovnanie: identifikátor, napr. názov premennej `číslo`, je „hmla“, v premennej typu `int` sa môže nachádzať ľubovoľné číslo z obrovskej množiny hodnôt, ale keď napíšeme `10`, bude to „doslovne“ znamenať desať a nikdy nič iné; a to je literál). Rozlišujeme niekoľko druhov literálov.

Typ	Opis	Príklady
Číselný	celočíselné, reálne a iné numerické hodnoty	`0 -11 13.4 0x8f`
Znakový	zápisy jednotlivých znakov (vyžadujú uzavretie do '', inak by ich Java považovala za operátor, identifikátor, prípadne by ich nevedela rozlíšiť)	`'z' '+' '*' 'á' '-' '\''`
Reťazcový	rôzne texty (vyžadujú uzavretie do ""…)	`"Dobrý deň!" "Zadajte počet:" "a + b" "Samuel Langhorne Clemens alias \"Mark Twain\"."`
Iný	pravdivostné hodnoty, „prázdna“ inštancia, prípadne iné	`true false null`

Ďalšie informácie o literáloch (v anglickom jazyku) nájdete na vyššie spomenutej stránke o primitívnych údajových typoch: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

Ak chceme vo vnútri znakových a reťazcových literálov použiť znaky ' alebo ", musíme pred nich uviesť znak \, to znamená, že do reťazca vpíšeme sekvenciu \' alebo \". Z toho dôvodu aj ak chceme použiť samotný znak \, musíme ho zapísať ako \\. Sú to takzvané „escape sekvencie“. Jestvuje viacero špeciálnych sekvencií. Napríklad: \n nový riadok, \t tabulátor a podobne. Týmto sekvenciám sa síce môžeme pri programovaní vyhýbať, ale niekedy je ich použitie nevyhnutné.

Príklady:

Cesty v OS Windows obsahujúce znaky „\“ musíme zapisovať v nasledujúcom tvare:

```
String cestaKIkone = "c:\\Windows\\RAR.PIF";
```

Čiže v premennej cestaKIkone sa v skutočnosti nachádza hodnota:

```
c:\Windows\RAR.PIF
```

Nasledujúci príklad:

```
String názovSúboru = "nejestvuje";

svet.vypíšRiadok("Súbor \"" + názovSúboru +
    "\" nebol nájdený.\nZadajte iný názov!");
```

Vypíše:

```
Súbor "nejestvuje" nebol nájdený.
Zadajte iný názov!
```

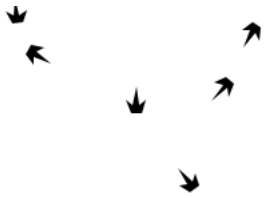
Statické atribúty a metódy

Statický znamená pevný, nemenný, fixný, stabilný, ukotvený… Statické objekty jestvujú spravidla od spustenia programu do jeho ukončenia (to neplatí pre dynamické objekty). Prvky, ktoré sú definované staticky, sú k dispozícii rovnocenne pre všetky inštancie danej triedy, prípadne aj pre celé okolie (ak sú verejné).

Statický je zároveň opakom dynamického, pričom dynamickosť je základom objektovo orientovaného programovania. Každý nový objekt (nová inštancia) je dynamický. Táto dynamickosť je chápaná v súvislosti s časom vytvorenia a trvanlivosťou objektu, čiže v súvislosti s tým, *kedy* bol objekt *vytvorený* (počas činnosti programu – t. j. „dynamicky“) a *kedy* bude *zničený* (hneď ako prestane byť potrebný/používaný – aj keď doteraz sme väčšinou pracovali s objektmi, ktoré boli potrebné až do ukončenia činnosti programu, v budúcnosti uvidíme, že nie vždy to je tak).

V Jave všetko, čo nie je explicitne určené ako statické, je dynamické. Klauzula `static` slúži na statické ukotvenie želaných objektov v pamäti. Tieto objekty (môžu to byť premenné, metódy ale i vnorené triedy…) sú „trvalé“. Má to i jeden praktický dopad. Môžeme tak deklarovať/definovať objekty (premenné), ktoré sú spoločné pre všetky inštancie danej triedy, resp. ak sú verejné, tak pre celý program (všetky objekty všetkých tried). V skupine tried grafického robota je statický napríklad `svet`. Ten, okrem iného, zastupuje aj aplikačné okno a nech vytvoríme ľubovoľné množstvo robotov, všetci budú jestvovať jednom v spoločnom svete:

```
for (int i = 0; i < 5; ++i)
{
    GRobot r = new GRobot();
    r.náhodnáPoloha();
    r.náhodnýSmer();
}
```



Potrebnosť statickosti môžeme demonštrovať aj na príklade s počítadlom inštancií:

Definujme jednoduchú triedu `Postava` (odvodenú od robota) s vnútornou premennou `počítadlo`. Premenná `počítadlo` bude statická. Na začiatku jej „priradíme“ hodnotu `0` (presnejšie premennú inicializujeme hodnotou `0`). V konštruktore triedy `Postava` bude hodnota počítadla zvyšovaná o `1`. Keby premenná `počítadlo` nebola statická, znamenalo by to, že každá inštancia postavy má vlastné počítadlo a príklad by nefungoval tak ako uvedieme. Tu je definícia triedy `Postava` s konštruktorom prijímajúcim meno postavy a statickou metódou `vypíšStavPočítadla` slúžiacou presne na to, na čo poukazuje jej názov – vypísanie aktuálneho stavu počítadla:

```
public class Postava extends GRobot
{
    private static int počítadlo = 0;

    public Postava(String meno)
    {
        svet.vypíšRiadok("Vytvoril som postavu: ", meno);
        ++počítadlo;
    }

    public static void vypíšStavPočítadla()
    {
        svet.vypíšRiadok("Momentálny stav počítadla: ", počítadlo);
    }
}
```

Ak teraz niekde (v hlavnej triede) vykonáme nasledujúcu sériu príkazov (štyrikrát vytvorenie novej inštancie triedy `Postava` a jedno volanie statickej metódy `vypíšStavPočítadla` triedy `Postava` – všimnite si **spôsob volania statickej metódy** na poslednom riadku: meno triedy, bodka a meno metódy):

```
new Postava("Alexander");
new Postava("Barnabáš");
new Postava("Cézar");
new Postava("Dan");
Postava.vypíšStavPočítadla();
```

Získame nasledujúci výstup:

```
Vytvoril som postavu: Alexander
Vytvoril som postavu: Barnabáš
Vytvoril som postavu: Cézar
Vytvoril som postavu: Dan
Momentálny stav počítadla: 4
```

S každou novou inštanciou sa zvyšuje stav počítadla, čo je možné len vďaka tomu, že bolo definované staticky…

Uzavretosť

Uzavretosť hovorí o tom, že pri písaní tried (objektovo orientovaných programov) je dôležité *oddeliť údaje* (atribúty) *od funkčnosti* (metódy). Je jednou z ústredných charakteristík objektovo orientovaného programovania. Filozofiou objektovo orientovaného programovania je (v tomto kontexte), z dôvodu ochrany údajov (v prvom rade z pohľadu zachovania správnosti/korektnosti ich hodnôt) a zachovania integrity objektov, objekty uzavrieť a umožniť prácu s nimi prostredníctvom príslušných metód. V súvislosti s tým používame i termín „skryť“. Hovoríme, že „skrývame“ premenné/metódy pred „okolím“. Okolím sú v tomto prípade ostatné triedy alebo balíčky (packages).

Na dodržanie princípu uzavretosti musíme zamedziť priamy prístup k údajom aj k rýdzo vnútornej funkcionalite objektov (t. j. ku všetkému, čo okolie nemá čo zaujímať, ku všetkému, čo okolie vyslovene nepotrebuje na komunikáciu s objektom). Všetko preto, aby si objekt sám mohol ustriehnuť svoju integritu. Majme napríklad celočíselnú (pravidelne aktualizovanú) premennú `vek`. Aby sme zamedzili vloženiu zápornej hodnoty do tejto premennej, deklarujeme ju ako súkromnú (`private`) a rovnomennú metódu určenú na zápis novej hodnoty ako verejnú (`public`). **Príklad:**

```
class Vek
{
    private int vek;

    public void vek(int novýVek)
    {
        if (novýVek >= 0) vek = novýVek;
    }
}
```

Zvonka triedy `Vek` nemáme právo zmeniť hodnotu vnútornej/súkromnej premennej `vek`. Akákoľvek zmena musí byť sprostredkovaná verejnou metódou `vek`, ktorá hodnotu zapisovanú do premennej kontroluje.

V tomto príklade sme použili dve základné rezervované slová, ktorými určujeme „viditeľnosť“ (ďalej už nebudeme slovo viditeľnosť uzatvárať do úvodzoviek; odteraz ho budeme pokladať za riadny termín). Sú to slová (resp. modifikátory prístupu) `private` (súkromné) a `public` (verejné). Java rozoznáva štyri úrovne viditeľnosti. Modifikátor `private` určuje súkromné prvky triedy. K nim má prístup len a len daná trieda. Presným opakom je modifikátor `public`. Ten určuje prvky, ktoré sú viditeľné „pre všetkých“ (pre celé okolie, to znamená pre všetky balíčky a triedy).

Modifikátor `protected` (chránené) určuje prístup, ktorý je (ľudovo povedané) „niečo medzi tým“ (t. j. medzi `private` a `public`). Ak ľubovoľný prvok triedy (atribút, metóda…) nemá pridelený žiadny z uvedených modifikátorov, ide o štvrtý samostatný druh viditeľnosti. Všetky spôsoby viditeľnosti sú zhrnuté v nasledujúcej tabuľke prístupov:

modifikátor	trieda	balíček	potomok	okolie
`private` (súkromný)	áno	nie	nie	nie
«bez modifikátora»	áno	áno	nie	nie
`protected` (chránený)	áno	áno	áno	nie
`public` (verejný)	áno	áno	áno	áno

Všetky spôsoby viditeľnosti nie sme schopní v rámci našich hodín programovania využiť. Niektoré na svoje prejavenie sa potrebujú rozdelenie projektu do balíčkov. Prakticky to ukazuje nasledujúci príklad vyrobený ako BlueJovský projekt: uzavretost.zip.

Dedičnosť

«*v tejto verzii dokumentu nebude táto téma úplne dokončená*»

Význam tohto termínu v objektovo orientovanom programovaní je v určitom smere podobný významu rovnakého termínu v biológii. Ibaže v tomto prípade vzájomne dedia a rozvíjajú svoje vlastnosti triedy, nie inštancie. V tomto zmysle sa dedičnosť viac podobá na evolúciu druhov než vnútrodruhovú/rodovú dedičnosť.

Java rozlišuje niekoľko druhov tried, ale v súvislosti s dedičnosťou v tomto materiáli vyberieme a opíšeme len dva základné druhy tried – klasické triedy definované rezervovaným slovom `class` a rozhrania (ktoré zároveň súvisia s abstrakciou – pozri aj kapitolu Abstrakcia na strane 16) definované rezervovaným slovom `interface`. Klasické triedy definujú atribúty, metódy a iné elementy, ktoré dedí potomok od rodiča. Rozhrania sú špeciálne (abstraktné) definície primárne určené na stanovenie povinných prvkov – metód, ktoré musí programátor pri ich použití implementovať.

Rozhrania sú niečo ako spoločným predpisom pre viaceré rôzne triedy, ktorý budú musieť alebo chcieť dodržať. Napríklad rozhranie `Comparable` (v preklade porovnateľný), ktoré je súčasťou jazyka Java určuje, že trieda, ktorá sa ho rozhodne použiť musí implementovať metódu `compareTo` (v preklade porovnaj s), ktorá bude schopná kvantitatívne porovnať dve inštancie tejto triedy, vďaka čomu budú inštancie triedy nielen vzájomne porovnateľné, ale aj použiteľné v určitých situáciách, ktoré vyžadujú schopnosť porovnania prvkov.

Klasické triedy smú mať jediného rodiča spomedzi iných klasických tried, v terminológii Javy smú *rozširovať* jedinú triedu – v syntaxi Javy sa na to používa rezervované slovo `extends`:

```
class Dieťa extends Rodič
{
}
```

Rozhrania smú rozširovať ľubovoľný počet iných rozhraní:

```
interface DetskéRozhranie extends RodičovskéRozhranie1, RodičovskéRozhranie2, …
{
}
```

Triedy tiež smú implementovať ľubovoľný počet rozhraní, na čo sa v syntaxi Javy používa rezervované slovo `extends`, a môžu mať tiež súčasne jedného rodiča spomedzi klasických tried (v súlade s informáciami vyššie):

```
class Dieťa extends Rodič implements RodičovskéRozhranie1, RodičovskéRozhranie2, …
{
}
```

Polymorfizmus

Polymorfizmus je termín označujúci *„mnohotvarosť“*… V Jave i iných programovacích jazykov jestvuje viacero druhov polymorfizmu. Niektoré druhy polymorfizmu úzko súvisia s inými objektovými vlastnosťami (napr. s dedičnosťou), iné sú natoľko nezávislé, že ich môžeme nájsť aj v jazykoch, ktoré vôbec nie sú objektovo orientované.

~~Na prednáškach sme tieto viaceré druhy polymorfizmu už spomínali.~~ (Toto bola stará informácia – takéto budú z dokumentu miznúť len pozvoľna, lebo času na revízie je málo.) Medzi tie, ktoré súvisia s inými objektovými vlastnosťami (konkrétne s dedičnosťou), by som zaradil dva: *prepisovanie/prekrývanie* (overriding) a tzv. *„subtypový“ polymorfizmus* (v literatúre sa s ním môžete stretnúť aj ako s *inkluzívnym polymorfizmom*).

Pod *prekrývaním (overriding)*, ktoré môžete nájsť aj pod označením *prepisovanie*, rozumieme nahrádzanie metód v odvodených triedach „novšími“ verziami. Tým chceme dosiahnuť zmeny v správaní odvodených objektov. Mohli by sme napríklad v triede odvodenej od robota prepísať metódu `dopredu` tak, aby robila niečo úplne iné, ale tým by sme zaručene pomýlili prípadných iných programátorov, ktorí by mali záujem takto zmeneného potomka používať, preto konkrétne takéto dačo neodporúčam.

V praxi však budeme prekrývanie často používať. Robot má definovaných niekoľko metód, ktoré sú priamo určené na to, aby sme ich prekryli (predvolene nerobia nič). Napríklad metóda `aktivita`: jej prekrytím dokážeme zmeniť správanie robotovho potomka, ktorého sme uviedli do aktivovaného stavu; alebo metóda `kresli`, ktorej prekrytie reprezentuje najjednoduchší spôsob, akým môžeme zmeniť predvolený tvar robota (resp. robotovho potomka).

Preťažovanie (overloading) – ide o definíciu viacerých verzií tej istej metódy (líšia sa v počte a/alebo type parametrov). Za normálnych okolností, ak sa zameriame len na definície metód, tak jeden identifikátor označuje (pomenúva) jednu konkrétnu metódu. Preťažovaním môžeme *zdanlivo* rozšíriť použiteľnosť niektorej metódy tým, že jej dovolíme akceptovať rôzne druhy (typy) parametrov a/alebo rôzne počty parametrov a na základe toho prispôsobiť svoje správanie. (Pozastavme sa nad tým, prečo sme použili slovo *zdanlivo*. Je to preto, že v skutočnosti nejde o *rozširovanie funkčnosti* tej istej metódy, ale o *definíciu ďalších verzií* metódy, čiže o akési *„recyklovanie“ názvu* metódy…) Rôzne verzie tej istej metódy by však mali dbať na to, aby pracovali rovnako, prípadne čo najviac podobne. Tento druh polymorfizmu nachádza využitie napríklad v takom prípade, keby sme potrebovali, aby niektorá metóda umožňovala:

- prijímať buď číselnú dvojicu súradníc (x, y), alebo taký objekt, z ktorého sa dá takáto dvojica súradníc získať (napr. robota); vtedy by sme mali dve verzie líšiace sa počtom aj typom parametrov;
- prijímať rôzne typy parametrov, s ktorými by dokázala vykonávať konkrétnu operáciu, dajme tomu výpis na obrazovku, t. j. niečo ako vlastná verzia metódy `vypíš` očakávajúca jediný parameter; vtedy by sme mali viacero verzií líšiacich sa typom parametra;
- prijímať buď jedného alebo dvoch robotov a podľa toho vykonať mierne odlišný úkon, napríklad pri zadaní jedného robota vypočítať a vrátiť jeho vzdialenosť od stredu, pri dvoch robotoch vypočítať a vrátiť ich vzájomnú vzdialenosť; vtedy by sme mali odlišný len počet parametrov…

Príkladov sa dá vymyslieť mnoho… Tento typ polymorfizmu môžete v robotovi vidieť napríklad pri metódach: `vzdialenosťOd`, `uholNa`, `elipsa`, ale aj `farba`, `súradnicaX`… Všetky spomenuté metódy majú viacero verzií, t. j. sú preťažené. Inými slovami môžeme povedať, že „jestvuje viacero metód s rovnakým menom (avšak s odlišnosťami v počte a/alebo type parametrov)…“

(Na tomto mieste treba podotknúť, že pri polymorfizme máme na mysli metódy s **úplne rovnakým menom** vrátane diakritiky a veľkosti písmen, pretože trieda `GRobot` obsahuje aj metódy s *podobným* menom a rovnakým účelom, ale to **nie je** polymorfizmus(!), to sú takzvané *aliasy* – prezývky metód.)

Preťažovanie môžeme demonštrovať napríklad na metóde `farba`. Robot má definované viaceré verzie metódy `farba`. Z nich nás budú zaujímať dve: prvá prijímajúca jeden parameter typu `Farba` a druhá prijímajúca tri celočíselné parametre. Vďaka prvej môžeme písať:

```
farba(červená);
farba(zelená);
farba(modrá);
```

Vďaka druhej môžeme definovať farby prostredníctvom farebných zložiek (RGB – červená, zelená, modrá):

```
farba(128, 0, 0);    // 50% červenej, zvyšok 0%
farba(0, 128, 0);    // 50% zelenej, zvyšok 0%
farba(0, 0, 128);    // 50% modrej, zvyšok 0%
```

Nezabúdajte, že všetko o čom tu hovoríme môžete používať aj vy vo svojich triedach. Preťažovanie budeme používať prinajmenšom na definovanie viacerých verzií konštruktorov.

Vďaka *subtypovému (inkluzívnemu) polymorfizmu* môžeme všade tam, kde je požadovaný objekt nadradeného údajového typu, dosadiť aj objekty odvodených tried. To znamená, že všade tam, kde je napríklad požadovaný objekt typu `GRobot`, môžeme dosadiť ľubovoľný objekt triedy odvodenej od robota. Jave to nielenže neprekáža, ale dokáže akceptovať aj prípadné zmeny v správaní objektu (prekryté/prepísané metódy)… Tento typ polymorfizmu je jednou z vlastností jazyka, ktorú priamo nevidíme, ale dokážeme oceniť jej dôsledky. Skôr by sme si všimli problémy, ktoré by vznikali, keby Java túto vlastnosť nemala.

Príklad: Metóda `otočNa` je jednou z metód majúcich viacero verzií (t. j. sú *preťažené*, ale to len na margo rôznych súvislostí v rámci témy polymorfizmus; teraz chceme hovoriť o inom type polymorfizmu…). Jednou z verzií je aj tá, ktorá prijíma parameter typu `GRobot`. Vďaka subtypovému polymorfizmu môžeme túto konkrétnu verziu metódy bez problémov použiť s ľubovoľným potomkom triedy `GRobot`. Čiže napríklad ak definujeme triedy `Raketa`, `Strela` a `Asteroid` odvodené od robota, môžeme bez obáv napísať takýto kód:

```
// Najskôr vytvoríme inštancie odvodených tried:
Raketa raketa = new Raketa();
Strela strela = new Strela();
Asteroid asteroid = new Asteroid();

// Predpokladajme, že sme raketu, strelu aj asteroid náhodne rozmiestnili…

// Teraz otočíme raketu smerom na asteroid, pričom využijeme metódu robota otočNa:
raketa.otočNa(asteroid);

// Podobne môžeme otočiť aj strelu na raketu alebo ľubovoľného robota
// alebo robotovho potomka na iného robota alebo robotovho potomka:
strela.otočNa(raketa);
```

Bez subtypového polymorfizmu by síce (vďaka dedičnosti) bola metóda `otočNa` všetkým objektom k dispozícii, ale *len a výhradne* s parametrom typu `GRobot`, potomkovia (`asteroid`, `raketa`…) by neboli akceptovaní a prekladač by hlásil chybu.

Tento typ polymorfizmu má ešte jeden dôsledok (alebo využitie), ktorý vysvetlíme pri parametrickom polymorfizme.

Pretypovanie (Type Casting) – vďaka tomuto typu polymorfizmu môžeme v nutnom prípade vykonať zmenu údajového typu na príbuzný (napr. `long` na `int`, `double` na `foat` a pod.), resp. čo sa týka rýdzo objektovo orientovanej časti jazyka, tak v prípade, že sme si istí tým, že nejaký objekt je odvodeného údajového typu, môžeme použiť operátor pretypovania, aby sme získali prístup ku všetkým vlastnostiam daného objektu.

Toto je možné najlepšie vysvetliť pomocou príkladu. Predpokladajme, že máme definovanú triedu `Zviera` odvodenú od robota:

```
public class Zviera extends GRobot
{
    private int početNôh;
    private String názov;

    public Zviera(String názov, int početNôh)
    {
        this.názov = názov;
        this.početNôh = početNôh;
    }

    public String názov()
    {
        return názov;
    }

    public int početNôh()
    {
        return početNôh;
    }
}
```

Trieda nerobí nič zvláštne, len umožňuje zapamätanie si názvu zvieraťa a počtu nôh. Vytvorme teraz objekt typu `Zviera`, ale uložme ho premennej typu `GRobot` (to je možné vďaka *subtypovému (inkluzívnemu) polymorfizmu*):

```
GRobot r1 = new Zviera("Mačka", 4);
```

Situácia je síce vytvorená umelo, ale v praxi sa hocikedy môžeme stretnúť s tým, že v premennej typu predchodcu (napr. `GRobot`) sa ocitne objekt typu potomka (napr. `Zviera`). Príklad takejto situácie je na konci bloku vysvetľujúceho *parametrický polymorfizmus*. Keby sme chceli použiť hocijakú vlastnosť zvieraťa:

```
svet.vypíšRiadok(r1.názov(), "má", r1.početNôh(),"nôh.");
```

Java by nám to nedovolila, pretože `r1` je typu `GRobot` a robot „netuší“, čo za metódy sú `názov` a `početNôh`… Musíme zmeniť typ `GRobot` na `Zviera`:

```
Zviera z1 = (Zviera)r1;
```

Túto zmenu môžeme vykonať len vtedy, keď sme si skutočne istí, že v premennej `r1` je objekt typu `Zviera`, inak by vznikla chyba. Po tejto zmene môžeme objekt `z1` bez problémov používať ako plnohodnotný objekt triedy `Zviera`:

```
svet.vypíšRiadok(z1.názov(), "má", z1.početNôh(), "nôh.");
```

Parametrický polymorfizmus – príkladom parametrického polymorfizmu je v rámci skupiny tried grafického robota trieda `Zoznam<Typ>`. Jej premenlivosť (polymorfizmus) spočíva v tom, že nikde nie je vopred

určené, akého typu zoznam bude (presnejšie akého údajového typu budú jeho prvky). Definícia je všeobecná s parametrom `<Typ>`, za ktorý dosádzame konkrétny údajový typ.

Príklady:

Zoznam reťazcov definujeme takto:

```
Zoznam<String> mená = new Zoznam<String>("Alexander", "Barbora", "Cyril");
```

zoznam celých čísiel takto:

```
Zoznam<Integer> čísla = new Zoznam<Integer>(3, 8, 5);
```

a zoznam robotov takto:

```
Zoznam<GRobot> roboti = new Zoznam<GRobot>();
```

Na tomto mieste treba zopakovať, že zoznamy nemôžu byť vytvorené z primitívnych údajových typov, čiže nemôžeme vytvoriť niečo takéto:

```
Zoznam<int>
```

Na záver jeden príklad spájajúci *subtypový (t. j. inkluzívny) polymorfizmus*, *parametrický polymorfizmus* a *pretypovanie (Type Casting)*.

Predpokladajme, že máme definované triedy `Raketa`, `Strela` a `Asteroid` odvodené od robota. Vytvorme inštanciu z každej z týchto tried:

```
Raketa raketa = new Raketa();
Strela strela = new Strela();
Asteroid asteroid = new Asteroid();
```

Definujme zoznam robotov a pridajme do neho všetky tri inštancie (čo vďaka *subtypovému polymorfizmu* môžeme urobiť):

```
Zoznam<GRobot> zoznam = new Zoznam<GRobot>();
zoznam.pridaj(raketa);
zoznam.pridaj(strela);
zoznam.pridaj(asteroid);
```

Použime teraz cyklus `for` na prejdenie zoznamu a náhodné rozmiestnenie všetkých objektov:

```
for (GRobot prvok : zoznam)
{
    prvok.náhodnáPoloha();
}
```

(Pre tri objekty to síce nemá až taký viditeľný význam, ale keby sme mali sto, dvesto, tisíc potomkov robota, významne by sme si ušetrili prácu.)

Spojením *subtypového* a *parametrického polymorfizmu* dokážeme do jedného zoznamu zoskupovať rozdielne objekty, ktoré majú spoločného predchodcu. Následne môžeme napríklad pomocou cyklu `for` vykonávať hromadné akcie pre všetky objekty zoznamu…

Takáto situácia vytvára zároveň prirodzený priestor na využitie *pretypovania*. Napríklad vieme, že prvým prvkom zoznamu je `raketa`. Predpokladajme, že `raketa` má definovanú metódu `vystreľ`. Keby sme nikde inde nemali uloženú inštanciu rakety, jediným spôsobom ako ju získať, by bolo vybrať ju ako prvok zo zoznamu (a následne napríklad zavolať jej metódu). Bez pretypovania by sme sa nezaobišli. Prostý výber prvku zo zoznamu a pokus o volanie (pre neho neznámej) metódy by zlyhal:

```
zoznam.daj(0).vystreľ();   // vznikne chyba: cannot find symbol – method vystreľ()
```

Pretože zoznam je typu `GRobot`. Na správny prvok zoznamu (teda ten, o ktorom sme si istí, že je typu `Raketa`) musíme použiť operátor pretypovania: `(Raketa)`. Najprv postup rozpíšme:

```
GRobot robot = zoznam.daj(0);     // výber nultého prvku zo zoznamu
Raketa raketa = (Raketa)robot;  // pretypovanie prvku na typ Raketa
raketa.vystreľ();               // volanie metódy definovanej v triede Raketa
```

Tieto tri kroky je možné zjednotiť do jedného funkčného riadku kódu. Zápis je trochu „prezátvorkovaný“:

```
((Raketa)zoznam.daj(0)).vystreľ();
```

`(Raketa)` je operátor pretypovania a `zoznam.daj(0)` je výber nultého prvku zo zoznamu (t. j. prvku s indexom nula, čo je z ľudského hľadiska vlastne prvý prvok zoznamu). Keď chceme okamžite použiť volanie metódy `.vystreľ()`, musíme prvú časť príkazu uzavrieť do dodatočnej zátvorky, inak by sa v tom prekladač Javy nevedel vyznať. Preto obsahuje tento prepis toľko zátvoriek… Vy musíte hlavne vedieť tento zápis prečítať, t. j. dešifrovať čo znamená, keby ste sa s ním niekedy stretli.

Takisto si vždy musíte byť istí, že meníte typ správneho prvku. Keby ste v tomto príklade napísali napríklad toto:

```
((Raketa)zoznam.daj(1)).vystreľ();  // rozdiel je nebadateľný – vyberáme prvok
                                    // s indexom 1 (druhý prvok zoznamu)
```

Java by to pokojne preložila, ale počas činnosti programu by vznikla chyba: `Strela cannot be cast to Raketa` (voľne preložené „strelu nemôžete zmeniť na raketu“), pretože v prvku s indexom 1 je objekt typu `Strela`.

Ak si nie ste istí, či môžete pre daný objekt použiť pretypovanie, v Jave jestvuje i operátor na overenie údajového typu. Použime ho v cykle `for`:

```
for (GRobot prvok : zoznam)
{
    if (prvok instanceof Raketa)
    {
        ((Raketa)prvok).vystreľ();
    }
}
```

Cyklus spôsobí, že všetky objekty typu raketa uložené v zozname vystrelia.

Abstrakcia

«*v tejto verzii dokumentu nebude táto téma úplne dokončená*»

rozhranie – rozhrania (angl. interfaces) sú špeciálne druhy tried v Jave, ktoré smú obsahovať iba deklarácie metód a definície statických konštánt.

implementácia rozhrania –

http://docs.oracle.com/javase/tutorial/java/concepts/interface.html

http://docs.oracle.com/javase/tutorial/java/javaOO/index.html

http://docs.oracle.com/javase/tutorial/java/IandI/index.html

http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

http://en.wikipedia.org/wiki/Interface_(Java)

http://mindprod.com/jgloss/interface.html

abstraktné triedy –

http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

http://www.tutorialspoint.com/java/java_abstraction.htm

http://www.javaworld.com/article/2077421/learn-java/abstract-classes-vs-interfaces.html

http://mindprod.com/jgloss/interfacevsabstract.html

Ďalšie termíny

Definícia, deklarácia – hlavný rozdiel medzi nimi je v tom, že definícia má aj „hodnotu“, resp. pri definícii vykonávame zároveň aj inicializáciu (priraďujeme počiatočnú hodnotu, určujeme obsah…)

Deklarácia – v prenesenom význame ide o „vyhlásenie“ (v našom kontexte „o existencii“). Pri programovaní pre zjednodušenie často vnímame deklaráciu ako priradenie (spárovanie) objektu a typu. Určujeme, akého typu má byť nový „objekt“ (napr. premenná) a to následne ovplyvní jeho správanie. Takéto chápanie sa týka najmä premenných, ale (napríklad) deklarácia metódy má zložitejšiu štruktúru…

Definícia – je takmer to isté ako deklarácia, ibaže rozšírená o určenie obsahu, t. j. o inicializáciu. Primitívne premenné obsahujú primitívne hodnoty. Obsahom tried je množina ďalších (jednoduchších, či zložitejších) deklarácií a definícií (atribútov, metód, vnorených tried…). Obsahom metód je ich telo – **príkazy**.

Inicializácia – v preklade znamená „uvedenie do činnosti“, „naštartovanie“. Napríklad inicializovať premennú znamená priradiť je počiatočnú hodnotu. Objektové typy inicializujeme priradením inštancie. Pri inicializácii inštancie (napríklad: `new HlavnáTrieda()`) je automaticky spúšťaný **konštruktor**.

Premenná, konštanta – rozdiel medzi nimi je v tom, že hodnota konštanty je nemenná. Obidve sú v programe reprezentované identifikátorom a majú hodnotu. Pre konštantu v Jave platí, že ak neurčíme jej počiatočnú hodnotu (neinicializujeme ju) hneď, môžeme jej hodnotu priradiť neskôr, ale len raz! Konštanty v programe napísanom v Jave odlišujeme rezervovaným slovom `final`:

```
int a = 10;          // premenná
final int b = 11;    // konštanta
```

Trieda – spojenie údajových (atribúty) a funkčných (metódy) prvkov; príklad: `GRobot`, `HlavnáTrieda`, `Double`, `String` – môžeme si ju predstaviť ako akýsi „popísaný papier“ založený v šuplíku. Sama o sebe, bez inštancie, nefunguje…

Inštancia – je to konkrétny objekt („nejakého“ údajového typu, ktorý určuje triedu). V programe je (podobne ako premenná) spravidla reprezentovaná identifikátorom. Príklad:

```
Farba modrá = new Farba(0, 0, 200);
Boolean pravda = new Boolean(true);
```

Inštancia je fyzický objekt, „oživenie“ toho, čo sme definovali v triede (triedach). Niektoré vyhradené objektové typy v Jave môžeme inicializovať pomocou zjednodušeného zápisu:

```
String meno = "Adam"; // aj tu vzniká inštancia triedy String,
                      // je to meno s hodnotou "Adam"
Boolean pravda = true;
```

Ak výsledok operátora new nepriradíme do žiadnej premennej (žiadnemu identifikátoru), vzniká takzvaná anonymná inštancia. Príklad:

```
new HlavnáTrieda();
```

Takáto inštancia by mala byť schopná postarať sa o svoje fungovanie svojpomocne. Ak nie, nemá jej vytvorenie význam.

Metóda – veľmi zjednodušene povedané „je to miesto, kam píšeme príkazy.“ V skutočnosti ide o dôležitý funkčný prvok triedy. Metódami definujeme, ako sa majú objekty triedy (t. j. vytvorené inštancie) správať. Niektoré problémy riešime definíciou skupiny úzko prepojených metód.

V rámci definície triedy voláme vlastné metódy triedy prostredníctvom identifikátora. Príklad:

```
nakresliŠtvorec();
```

Metódy iných inštancií voláme prostredníctvom bodkovej konvencie. Príklady:

```
inýRobot.dopredu(20);
môjZoznam.ďalší();
```

kde `inýRobot` je typu `GRobot` a `môjZoznam` je typu `Zoznam<Typ>`.

Konštruktor – je špeciálny druh metódy. Nie je možné ho spustiť priamo tak, ako sme zvyknutí pri ostatných metódach. Konštruktor je spúšťaný automaticky pri inicializácii inštancie, čiže počas vytvárania novej inštancie. Príklad inicializácie inštancie čiernej farby: `new Farba(0, 0, 0);`

Výraz – kombinácia operandov (identifikátorov, literálov…) a operátorov spravidla produkujúca nejaký výsledok… Príklad:

```
(a + b) / 4
```

Príkaz – vykonateľný riadok (resp. viacriadkový odsek patrične ukončený bodkočiarkou) zdrojového kódu. Výskyt príkazu môže byť zastúpený blokom.

Výrok – výraz, deklarácia, definícia, riadiaca štruktúra alebo iný príkaz…

Priradenie hodnoty – príklad:

```
a = 7;
```

Zmena hodnoty – príklad:

```
++a;
```

Porovnanie – príklady:

```
x > y, 1 < 2, s >= 10
```

Volanie metódy – termín označujúci spustenie metódy s očakávaním jej návratu s prípadným spracovaním návratovej hodnoty. Metódy voláme prostredníctvom ich názvu (identifikátora) so zadaním vstupných argumentov: `dopredu(20); skočNaMyš();`

Prípadnú **návratovú hodnotu** metódy môžeme priradiť do premennej adekvátneho údajového typu. Napríklad:

```
Farba zálohaAktuálnejFarby = farba();
// ...
farba(zálohaAktuálnejFarby);
```

`void` – v doslovnom preklade „bez hodnoty“ je rezervované slovo, ktoré označuje „prázdny“ údajový typ.

Argument, parameter – tieto dva termíny sú v programovaní často omylom zamieňané. Drobný (ale podstatný) rozdiel je v tom, že argumentom označujeme vstupnú hodnotu zadávanú pri volaní metódy, parametrom označujeme vstupnú časť definície metódy. Čiže ide o uhol pohľadu. Parameter je časťou

hlavičky definície, je reprezentovaný identifikátorom. Používame ho v tele metódy ako premennú. Argument je vstupná hodnota zadaná pri volaní metódy – môže byť vyjadrený výrazom.

Operand je termín označujúci „to, čo vstupuje do operácie“. Operand je súhrnný názov pre všetky druhy operácií. (Napríklad operandy sčítania sa v slovenčine nazývajú aj sčítance, operandy súčinu činitele a podobne…)

Návratová hodnota – hodnota poskytovaná ako výsledok pri volaní metódy.

Riadiaca štruktúra – syntaktická štruktúra ovplyvňujúca (riadiaca) činnosť programu. Môže ísť o vetvenie alebo cyklus.

Blok – skupina príkazov (patrične syntakticky ohraničená). V Jave ide o množinu príkazov zoskupených v zložených zátvorkách `{ }`.

Sekvencia – vykoná sa len raz…

Vetvenie – tiež sa vykoná len raz, ale dáva nám možnosť výberu medzi dvoma (alebo viacerými) blokmi príkazov…

Vetva – jeden z blokov vetvenia.

Cyklus – sa opakuje ak platí podmienka…

Iterácia – jedno vykonanie príkazov tela cyklu (slangovo „jeden beh cyklu“).

Sekvencia, **cyklus** aj **vetvenie** obsahujú príkazy! (Resp. môžu aj bloky príkazov, keďže na pozíciu príkazu môžeme prakticky vždy dosadiť blok…)

Sekvenčné („synchrónne“?) a asynchrónne spracovanie

Úvodom krátke pozastavenie. Ak sa pozrieme do výkladového slovníka, tak pri definícii slova „asynchrónny“ nájdeme aj význam „nesúdobý, časovo nesúhlasný, rozdielny v tempe“… To veľmi dobre opisuje situáciu, o ktorej chceme hovoriť. Pri definícii slova „synchrónny“ môžeme nájsť (okrem iného) význam „prebiehajúci; dejúci sa súčasne, paralelne…“, čo je síce najbližšie k tomu, o čom chceme hovoriť, ale v dokonalom súlade s tým **nie je**. Preto je v súvislosti s opisom toho, čo má byť v našom prípade *opakom asynchrónneho spracovania* priliehavejšie použiť termín „sekvenčný“, čo (podľa slovníka) znamená „nasledujúci za sebou v istom poriadku“.

Sekvenčné a asynchrónne spracovanie – v princípe ide o jeden zásadný rozdiel spočívajúci v tom, *kedy nastane spracovanie*. V prvom prípade (sekvenčné spracovanie) vieme vopred predpovedať, kedy bude ktorý príkaz vykonaný, v druhom opačnom prípade (asynchrónne spracovanie) to vopred jasné nie je, pretože spracovanie je riadené udalosťami, ktorých vznik nie je možné vopred určiť (stláčanie klávesov klávesnice, klikanie tlačidiel myši…)

Špeciálna hodnota `null` – v doslovnom preklade „nulový“, v Jave sa táto hodnota používa vo význame „žiadny“ na vyjadrenie toho, že požadovaný objekt (inštancia) nejestvuje. Deklarácia ľubovoľnej objektovej premennej má predvolene hodnotu `null` – „žiadnu hodnotu“.

Často kladené otázky

Čo je to `void`?

Rezervované slovo `void` sa v Jave používa iba pri definovaní (príp. deklarovaní) metódy, ktorá nemá návratový typ. V doslovnom preklade slovo znamená „bez hodnoty“ (bezcenný, pustý), čo môžeme chápať tak, že metóda nemá návratovú hodnotu.

Čo je to `null`?

Rezervované slovo `null` označuje v Jave nejestvujúci objekt. V doslovnom preklade znamená „nulový“ (neplatný, žiadny), čo môžeme chápať ako „nejestvujúci“.

Koľko má Java tried?

Ak by sme vychádzali z dokumentácie Javy verzie 6 (pozri zoznam všetkých tried), napočítali by sme 3 793 tried. Je to celkom slušná zásoba, ktorá v ďalších verziách porastie. (Rovnakým spôsobom by sme sa pre verziu 7, ktorú zatiaľ v rámci našich predmetov nepoužívame, dopátrali k číslu 4 024.)

Učili sme sa toľko riadiacich štruktúr (if-else, for, while, switch…) Kedy mám čo použiť?

Na túto otázku nejestvuje jednoznačná a jednoduchá odpoveď. Niekedy je odpoveď jasná. Napríklad potrebujem zopakovať niekoľko príkazov a viem presne koľkokrát ich chcem zopakovať – použijem `for`. Alebo mám niekoľko príkazov, ktoré nechcem, aby sa vykonali v danej situácii – použijem `if`. A podobne.

Stále platí, že skúsenosti znamenajú najviac. Čím viac programov napíšete, tým lepšie porozumiete filozofii a princípom, ktoré sú platné pri písaní programov. Niektoré sa naučíte rýchlo, na osvojenie si iných sú potrebné roky. Naučenie sa programovať si vyžaduje čas…

(Ak smiem odporúčať – dobrý spôsob učenia sa je aj „rozbúravanie“ jestvujúcich programov. Zoberte si nejaký krátky funkčný program, urobte si z neho zálohu a skúste ho zmeniť. Nebojte sa, že ho pokazíte, ak sa to aj stane, obnovíte ho zo zálohy. Na chybách sa najlepšie učí. Keď sa prehryziete cez obdobie chýb, skúšajte veci vylepšovať…)

Aký je rozdiel medzi objektom a inštanciou?

V súvislosti s objektovo orientovaným programovaním – nie je objekt ako objekt. Objekt v širšom význame môže znamenať čokoľvek. Objekt v užšom význame sa rovná inštancii. Chápanie slova objekt môžeme priblížiť chápaniu slova vec v slovenskom jazyku. V širšom význame je vecou čokoľvek, v užšom význame je vec kus oblečenia.

Najčastejšie (prípadne najzávažnejšie) omyly vyskytujúce sa počas semestrov 2012 – 2014

To, že nasledujúce chyby boli odhalené v semestroch rokov 2012 až 2014 neznamená, že počas iných semestrov sa nevyskytovali. Naopak, viaceré sa opakovali aj nad rámec spomínaných semestrov.

Neznalosť (prípadne zahmlené chápanie) termínu údajový typ

Údajový typ je dôležitý na určenie **„druhu“ údajov**, s ktorými sa bude pracovať, napríklad: *celé čísla* (`int`, `long`…), *reálne čísla* (`double`, `float`…), *znaky* (`char`…), rôzne *objekty* počnúc reťazcom (`String`) robotom nekončiac a tak ďalej. Treba tiež rozumieť tomu, že „druhom“ údajov sa v Jave rozumie aj typ objektu. To znamená, že každá nová trieda sa zároveň automaticky stáva novým údajovým typom…

Primitívne údajové typy (`char`, `int`, `float`…) najčastejšie používame na určenie typu premennej (prípadne konštanty). Ale vo všeobecnosti údajové typy nenachádzajú využitie len tam. Používame ich napríklad aj na určenie typu návratovej hodnoty metódy (čiže na určenie toho, akého typu bude výsledok poskytovaný metódou)…

Neporozumenie základnému rozdielu medzi definíciou a použitím

Z bežného života je človek zvyknutý na to, že ako náhle niečo vezme do ruky a niečo s tou vecou vykoná, už ju používa. Je to tak prirodzený princíp, že sa zdá zbytočné ho vysvetľovať. Lenže presne toto prirodzené chápanie okolitého sveta často zvádza nováčikov v oblasti programovania inak chápať niektoré súvislosti v programovaní.

Keď definujem premennú, neznamená to, že som ju použil (že s ňou pracujem). Rovnako, keď definujem metódu, ešte to nič neznamená – to nie je jej použitie.

Prirovnaním k reálnemu svetu znamená vo svete programovania definícia (alebo deklarácia) len výrobu určitého „predmetu“. Na čo je človeku skrutkovač, ktorý síce jestvuje, bol vyrobený, ale celý čas leží v šuplíku – nikto ho nepoužíva!?

Definícia/deklarácia premennej značí jej výrobu. Jej používanie súvisí s čítaním, zápisom (alebo zmenou) jej hodnoty. Predovšetkým čítanie hodnoty je chápané ako použitie (alebo „využitie“) premennej v programe. Zápis je síce nevyhnutný, ale premennú, do ktorej len zapisujem, môžem len veľmi ťažko považovať za použitú, resp. *úplne využitú* v programe.

Definícia metódy značí jej výrobu. Použitie metódy znamená jej zavolanie, to v reči programovania znamená jej spustenie. Metódu voláme jej menom s množinou argumentov (ktorá môže byť aj prázdna). Niektoré metódy poskytujú návratovú hodnotu, ktorú je tiež možné použiť na ovplyvňovanie činnosti programu.

Mylná predstava pri chápaní termínu uzavretosť

Viacerí uviedli, že ~~uzavretosť znamená, že príkazy programu sú uzavreté v zložených zátvorkách {}~~. **Nie je to tak…** Uzavretosť z pohľadu programovania v Jave, resp. z pohľadu objektového programovania, je uplatňovanie pravidla o *„oddelení“ údajov od funkčnosti z pohľadu okolia*.

Na úplné spresnenie je nutné dodať, že:

- pod *údajmi* tu máme na mysli najmä také atribúty tried, t. j. také vnútorné premenné, ktoré sú primitívneho údajového typu,
- pod *funkčnosťou* tu máme na mysli predovšetkým metódy
- a slovným spojením *„z pohľadu okolia“* sa snažíme zdôrazniť, že nejde o ich oddelenie navzájom, ale z pohľadu okolia (v skutočnosti sú atribúty a metódy navzájom úzko prepojené; metódy slúžia na ochranu atribútov a na prácu s nimi).

Ide o to, že sa snažíme *skryť* atribúty (resp. vnútorné premenné) pred okolím a tým ich ochrániť. Atribút primitívneho typu sa sám chrániť nevie, musíme sa o to postarať my (spomeňte si na príklad s vekom, ktorý nesmie byť záporný).

Pravidlo o uzavretosti realizujeme úpravou viditeľnosti prvkov pomocou modifikátorov: `private`, `protected`, `public` (prípadne neuvedením žiadneho z modifikátorov, čo je tiež jeden typ viditeľnosti rozlišovaný Javou). Najčastejšie sa stretávame s modifikátormi `public` a `private`.

Nesprávne porozumenie statickosti

Vyskytli sa vyjadrenia typu ~~*„statický je konštantný, preto sú konštanty statické“*~~, ~~*„statický znamená, že sa zopakuje len raz“*~~, ~~*„statický znamená, že má pevné miesto, napr. else nemôže byť inde, než pri if“*~~ a podobne.

Tieto vyjadrenia sú v rôznej miere odchýlené a rôzny spôsobom vzdialené od podstaty… Statický síce znamená „nehybný“, ale nedával by som to do takejto súvislosti s konštantami (vysvetlíme to nižšie, v sekcii „Nesprávne porozumenie termínu konštanta“). Statické prvky sú síce „spoločné pre všetky inštancie triedy“, ale vyjadrenie typu „opakujú sa len raz“ vyvoláva mylný dojem, že by to mohlo mať niečo spoločné s cyklami. Statický prvok síce „má svoje pevné miesto (v pamäti)“, ale vôbec to nesúvisí so štruktúrou programu, pretože statickosť sa týka údajov a štruktúra (napr. if-else) sa týka funkčnosti.

Podstata statickosti sa najlepšie vysvetľuje ako protiklad dynamickosti, ale to podmieňuje dobré porozumenie dynamickosti. Pohovorme preto najskôr v krátkosti o dynamickosti. V predchodcoch objektových jazykov sa mohol programátor začať oboznamovať s dynamickosťou až po dosiahnutí istej úrovne. V objektovom jazyku ju používa od začiatku bez toho, že by si to uvedomoval. V Jave je dynamické všetko (samozrejme okrem toho, čo deklarujeme ako statické). Dynamické objekty, na rozdiel od statických, nemusia v čase štartu aplikácie vôbec jestvovať (a spravidla ani nejestvujú). Práve slovom „dynamický“ sa pri programovaní často označuje objekt, ktorý je vytvorený až počas činnosti programu (operátorom `new` – nový). (Vytváranie objektov až počas činnosti programu nebolo v mnohých skorších programovacích jazykoch také samozrejmé, ako to je v Jave.) Dynamický objekt dáva programátorovi viac možností na manipuláciu s ním (na programátorskej úrovni).

Z tohto pohľadu by sa mohlo zdať, že statickosť je skôr nevýhodou. Opak je pravdou. Niekedy sa bez statickosti nezaobídeme. Ak chceme napríklad vytvoriť počítadlo objektov (trebárs robotov), musíme ho definovať staticky, pretože inak by si každý objekt (robot) vytvoril svoje vlastné počítadlo, do ktorého by vložil 1. Mali by sme mnoho zbytočných počítadiel, z ktorých by ani jedno neobsahovalo skutočný počet vytvorených objektov…

Tak isto príkazy, ktoré majú byť „nezávislé“, „sebestačné“, prípadne „spoločné“, je správne definovať staticky. V prostredí robotov je takým stabilným spoločným prvkom `svet`. Príkazy sveta nie sú viazané na konkrétneho robota (sú medzi nimi príkazy na generovanie náhodných čísiel, nastavovanie farby pozadia, prácu s časovačom, nastavovanie priehľadnosti plátien a podobne.)

Nesprávne porozumenie termínu konštanta

Z pohľadu definícií a deklarácií programovacieho jazyka nie je medzi konštantnou a premennou veľký rozdiel. Hlavný rozdiel je v tom, že hodnotu konštanty nemôžeme meniť. (Z toho potom vyplýva odlišný spôsob ich použitia…)

Pozrite sa na nasledujúce definície:

```
int a = 10;
final int b = 22;
```

Prvá je celočíselná premenná a, druhá je celočíselná **konštanta** b. Všimnime si, že obidve sú celočíselného údajového typu (int), čiže aj konštanty **majú** údajový typ a obidve taktiež majú identifikátor aj hodnotu. Rozdiel medzi nimi určuje slovíčko final, čo znamená konečný, definitívny. Ním je v tomto príklade určené, že hodnota, ktorá bude do b vložená, už nebude zmenená, to znamená že b je konštanta.

Vráťme sa k objasneniu toho, prečo bývajú konštanty často statické (bolo to spomenuté na začiatku časti „Nesprávne porozumenie statickosti“). Pozrime sa na túto triedu:

```
public class TriedaSKonštantou extends GRobot
{
    final int mojaKonštanta = 10;

    public TriedaSKonštantou()
    {
        // atď.
    }

    // atď.
}
```

Takáto definícia triedy (resp. konštanty v nej) by mala za následok vznik samostatnej kópie konštanty mojaKonštanta spolu s každou novou inštanciou triedy TriedaSKonštantou, čo je veľmi neefektívne a absolútne zbytočné. Úplne zbytočne by sme mali toľko konštánt, koľko inštancií a navyše by sme nemohli konštantu používať bez vytvorenia aspoň jednej inštancie triedy (iba statické prvky môžeme používať bez toho, že by sme museli vytvoriť inštanciu triedy).

Preto je správne definovať konštanty staticky, static (a ak majú byť verejne viditeľné, tak aj verejne, public). To je dôvodom, prečo sa v Jave zaužíval nasledujúci spôsob definovania konštánt (na poradí rezervovaných slov public, final a static nezáleží a slovo public môžeme nahradiť private, keď chceme definovať súkromnú konštantu):

```
public final static int mojaKonštanta = 10;
```

Samozrejme, že ide o ukážku. Tak isto môžeme definovať konštantu **ľubovoľného** údajového typu…

Množina jednoduchých príkladov

Jednoduché vetvenie v cykle „for“

```
hrúbkaČiary(15);
skoč(0, -100);

/*# Cyklus „for“ je riadený pomocou riadiacej premennej */

/*#
 * Pre i s počiatočnou hodnotou nula;
 * opakuj v prípade, že je i menšie než desať;
 * po každom opakovaní zvýš hodnotu i o jedna
 */
for (int i = 0; i < 10; ++i)
{
    /*# Vetvenie sa riadi podmienkou */

    if (i % 2 == 0)
    /*# ak (i zvyšok po delení 2 je rovný 0) */
    /*# ak (zvyšok po delení i dvoma je rovný nule) */
    {
        farba(šedá);
    }
    else
    /**# inak */
    {
        farba(čierna);
    }

    dopredu(20);
}
```

Viacnásobné vetvenie

```
/*# Prepínač riadený premennou premenná */
switch (premenná)
{
    /*# V prípade, že hodnota premennej je rovná hodnota1 */
    case hodnota1:
        príkazy1();

        /*# Preruš! */
        break;          // Ak ho zabudneme uviesť, vykoná sa viacero vetiev!

    /*# V prípade, že hodnota premennej je rovná hodnota2 */
    case hodnota2:
        príkazy2();
        break;

    // Prípadné ďalšie vetvy

    /*# Predvolené: */
    default:
        predvolenéPríkazy();
}
```

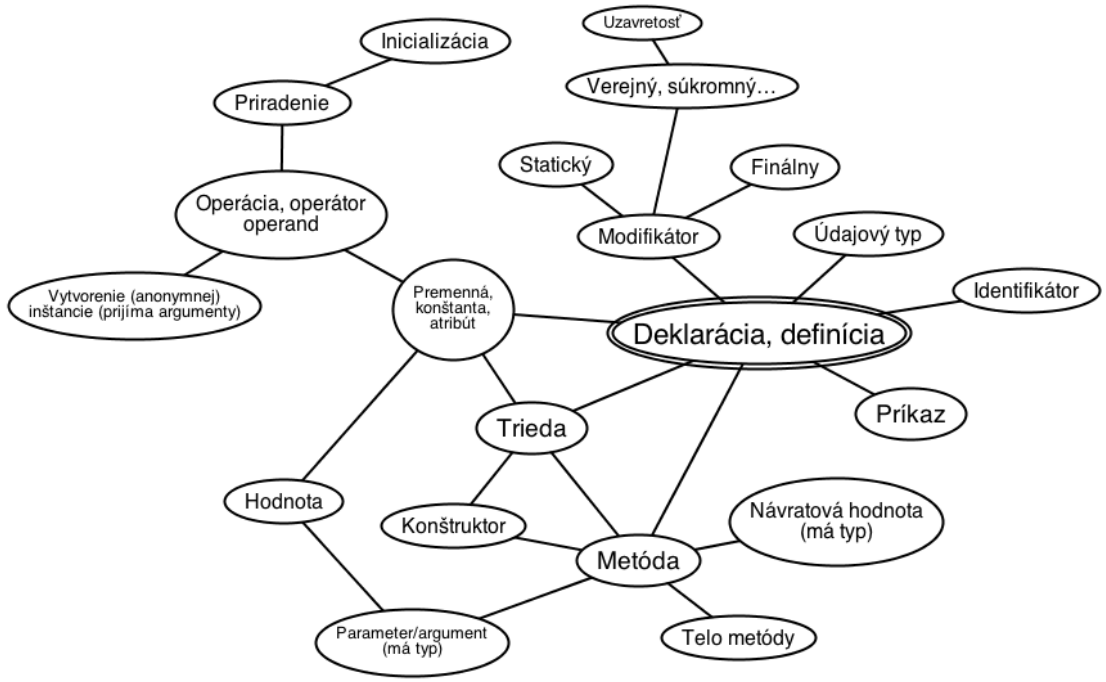
Ukážky metód

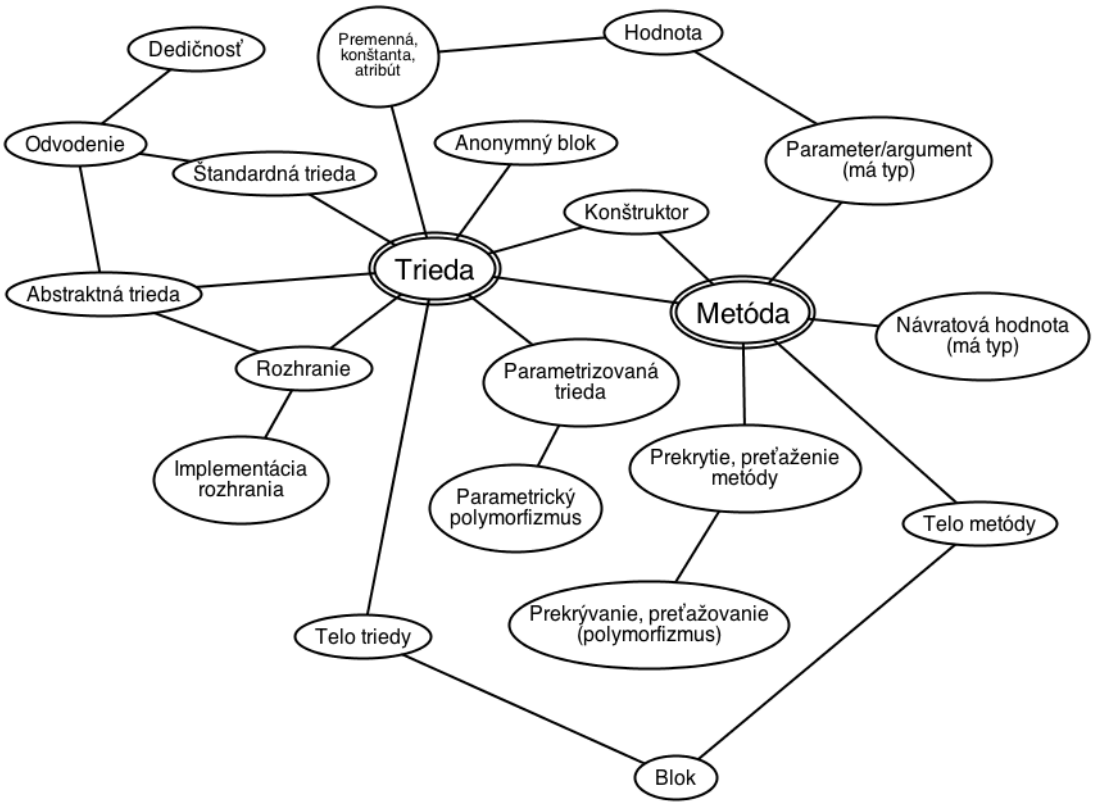
```
/**
 * Konštruktor – v podstate ho tiež považujeme za špeciálny druh metódy, ale
 * v tomto príklade je uvedený najmä preto, že sme do neho umiestnili volania
 * definovaných metód.
 */
private HlavnáTrieda()
{
    /*# Príklady volania metódy trojuholník. */
    trojuholník(30);
    trojuholník(45);
    trojuholník(60);

    /*# Cyklus s volaním metódy nUholník. */
    for (int i = 3; i < 12; ++i)
    {
        nUholník(i, 50);
    }

    /*# Výpis s volaním metódy súčet. */
    svet.vypíš("Súčet 10 a 35 je ", súčet(10, 35));
}

/**
 * Jednoduchá metóda s jedným parametrom/argumentom. Kreslí trojuholník so
 * zadanou dĺžkou strany.
 */
public void trojuholník(int dĺžkaStrany)
{
    /*#
     * Telo metódy obsahuje cyklus obsahujúci dva príkazy na ovládanie robota,
     * čo vo výsledku spôsobí nakreslenie trojuholníka so zadanou dĺžkou strany.
     */
    for (int i = 0; i < 3; ++i)
    {
        dopredu(dĺžkaStrany);
        doprava(120);
    }
}

/**
 * Metóda nUholník prijíma dva argumenty.
 *
 *  @param  n  parameter vyjadruje/určuje počet uhlov (t. j. zároveň strán)
 *             kresleného n-uholníka
 *
 *  @param  dĺžkaStrany  tento parameter určuje dĺžku jednej strany
 *             kresleného n-uholníka
 */
public void nUholník(int n, int dĺžkaStrany)
{
    /*#
     * Telo tejto metódy je podobné ako pre metódu trojuholník. Obsahuje navyše
     * jeden príkaz, ktorým počítame otočenie robota potrebné na správne
     * nakreslenie zadaného n-uholníka.
     */
    double otočenie = 360 / n;

    for (int i = 0; i < n; ++i)
    {
        dopredu(dĺžkaStrany);
```

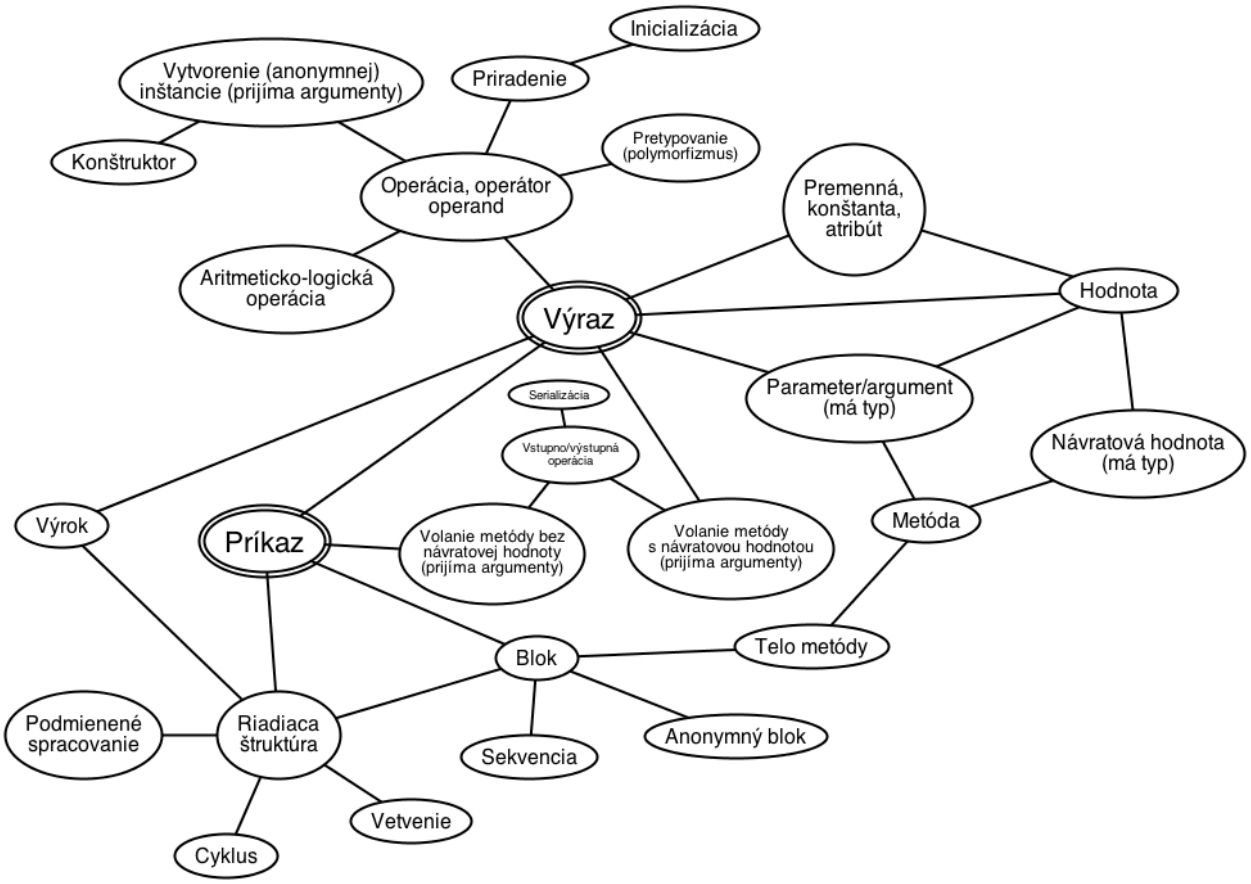
```
            doprava(otočenie);
        }
    }

    /**
     * Táto metóda slúži na predvedenie definície metódy s návratovou hodnotou.
     * Parametre a a b sú sčítance, návratovou hodnotou je výsledok ich súčtu.
     *
     *  @param  a  sčítanec
     *  @param  b  sčítanec
     *  @return  súčet
     */
    public int súčet(int a, int b)
    {
        /*#
         * Jediný príkaz v tele tejto metódy určí návratovú hodnotu metódy ako
         * výsledok súčtu parametrov a a b (ktoré tu zastávajú úlohu sčítancov).
         * Veľmi dôležitý je fakt, že príkaz „return“ zároveň ukončuje činnosť
         * metódy(!), to znamená, že keď ho uvedieme kdekoľvek v rámci riadiacich
         * štruktúr (napr. v podmienenom spracovaní „if“), ďalšie príkazy uvedené
         * v tele metódy sa (po jeho vykonaní) už nevykonávajú.
         *
         * Tým je „return“ predurčený hrať dvojitú úlohu. Po prvé, určuje návratovú
         * hodnotu metódy (ak ide o metódu s návratovou hodnotou, nie „void“) a po
         * druhé, ukončuje činnosť metódy, takže je použiteľný aj v metódach bez
         * návratovej hodnoty („void“) – na ich predčasné ukončenie (ak treba).
         */
        return a + b;
    }
```

Pojmové mapy

Pojmová mapa je mapa termínov (a pojmov) vyjadrujúca vzťahy medzi nimi. Do našich máp nie je možné zaznačiť úplne všetky vzťahy (ani všetky termíny). Označujeme len tie najdôležitejšie vzťahy. Úplná množina vzťahov (a termínov) bude vyjadrená v budúcnosti v interaktívnej podobe vzdelávacieho materiálu.







Odporúčané zdroje

Canadian Mind Products Java & Internet Glossary: http://mindprod.com/jgloss/jgloss.html

(Samozrejme aj všetky zdroje uvedené na stránke vyučujúceho, v dokumentácii Robota a v MAISe)