Objektovo-orientované programovanie

Objekty, Java a aspekty

Valentino Vranić

Objektovo-orientované programovanie

Objekty, Java a aspekty

Slovenská technická univerzita
v Bratislave
2008

PUBLIKÁCIU PODPORILO ZDRUŽENIE

GRATEX IT INŠTITÚT

v rámci fondu GraFIIT www.gratex.com

© Ing. Valentino Vranić, PhD.

Lektori: doc. Ing. Pavol Herout, PhD.
doc. Ing. Ján Kollár, PhD.

Vydala Slovenská technická univerzita v Bratislave vo Vydavateľstve STU, Bratislava, Vazovova 5.

Text neprešiel jazykovou úpravou vydavateľstva.

Schválilo vedenie Fakulty informatiky a informačných technológií STU v Bratislave pre študijný program Informatika a študijný program Počítačové systémy a siete.

ISBN 978-80-227-2830-0

Obsah

Zoznam obrázkov v

Predhovor vii

1 Úvod 1

2 Vhľad do objektovo-orientovaného programovania 3
2.1 Vznik objektovo-orientovaného programovania 3
2.2 Objekty a triedy 4
2.3 Stav a správanie objektu 4
2.4 Typ objektu 5
2.5 Agregácia 5
2.6 Zapuzdrenie 6
2.7 Dedenie 7
2.8 Polymorfizmus 8
2.9 Sumarizácia 10

3 Programovací jazyk Java 11
3.1 Java — jazyk a platforma 11
3.2 Objekty v Jave 13
3.3 Primitívne typy 14
3.4 Triedy 15
3.5 Uvoľňovanie pamäte 19
3.6 Prvý program 19
3.7 Zdrojové súbory a preklad 20
3.8 Balíky 20
3.9 Riadenie prístupu 23
3.10 Komentár a vnorená dokumentácia 24
3.11 Operátory v Jave 25
3.12 Riadenie vykonávania programu 30
3.13 Preťaženie metód 31
3.14 Inicializácia, konštruktory a finalizácia 33
3.15 Polia 38

4 Agregácia a dedenie 43
4.1 Agregácia 43

4.2 Dedenie 43
4.3 Prekonávanie metód 46
4.4 Riadenie prístupu v dedení 48
4.5 Inicializácia pri dedení 48
4.6 Trieda **Object** 49
4.7 Kľúčové slovo **final** 49
4.8 Dedenie a typy 51

5 Polymorfizmus 53
5.1 Polymorfizmus a prekonávanie 53
5.2 Abstraktné triedy a metódy 55
5.3 Polymorfizmus a statické metódy 56
5.4 Rozhrania 58

6 Aplikácia objektovo-orientovaných mechanizmov 63
6.1 Rôznorodosť objektovo-orientovaných prístupov 64
6.2 Abstrakcia 65
6.3 Zapuzdrenie 66
6.4 Modulárnosť 68
6.5 Hierarchia 68
6.6 Typovosť a polymorfizmus 72
6.7 Sumarizácia 73

7 Vhniezdené typy 75
7.1 Preklad vhniezdených typov 75
7.2 Vnútorné triedy 76
7.3 Anonymné triedy 79
7.4 Statické vhniezdené triedy 81

8 Výnimky 83
8.1 Podpora výnimiek v Jave 83
8.2 Kontrola výnimiek 84
8.3 Vlastné výnimky 86
8.4 Výnimky pri prekonaných metódach 88

9 RTTI 89
9.1 Literál **class** 89
9.2 Rozhodovanie na základe typu 89
9.3 Reflektívne triedy 90

10 Zoskupenia objektov a generickosť 93
10.1 Typy zoskupení 93
10.2 Generickosť zoskupení 94
10.3 Náhradný znak pre typ 94
10.4 Kontrola typov v generickosti 96

10.5 Iterátory 97
10.6 Rozšírená slučka **for** 98
10.7 Vymenované typy 98
10.8 Generické metódy 100
10.9 Automatické balenie hodnôt primitívnych typov 102
10.10 Trieda **Class** a generickosť 102

11 Vstupno/výstupný systém Javy 105
11.1 Práca s adresármi 105
11.2 V/V systém Javy založený na prúdoch 107
11.3 Čítanie súboru po riadkoch 108
11.4 Štandardný V/V systém 108
11.5 Formátovaný výstup 109
11.6 Čítanie z pamäte po znakoch 110
11.7 Čítanie z pamäte podľa formátu údajov 110
11.8 Zápis a čítanie súborov 111
11.9 Kanály a vyrovnávacie pamäte 114
11.10 Súbory zobrazené do pamäte 115
11.11 Kompresia 116
11.12 Serializácia objektov 116

12 Viacniťovosť 119
12.1 Vytváranie nití 119
12.2 Riadenie nití 120
12.3 Synchronizácia nití 121

13 Grafické používateľské rozhranie v Jave 125
13.1 Používateľské rozhranie 125
13.2 Swing 126
13.3 Tvorba okien 127
13.4 Pridávanie komponentov do JFrame 129
13.5 Spracovanie udalostí vo Swingu 131
13.6 Niť na odosielanie udalostí vo Swingu 134

14 Objektovo-orientované modelovanie 137
14.1 Modelovanie softvéru 137
14.2 UML 138
14.3 Diagram tried 140
14.4 Diagram prípadov použitia 146
14.5 Diagram sekvencií 150

15 Návrhové vzory 153
15.1 Vzory vo vývoji softvéru 153
15.2 Architektonický vzor Model-View-Controller 155
15.3 GoF vzory 156

15.4 Návrhový vzor Visitor 157
15.5 Idióm double dispatch 161
15.6 Návrhový vzor Observer 162

16 Aspektovo-orientované programovanie 167
16.1 Pretínajúce záležitosti 168
16.2 Príklad: jednoduché monitorovanie 168
16.3 Body spájania 172
16.4 Základné vlastnosti jazyka AspectJ 172
16.5 Bodové prierezy 174
16.6 Videnia 182
16.7 Kontext bodu spájania 185
16.8 Medzitypové deklarácie 186
16.9 Abstraktné aspekty 189
16.10 Reflektívna podpora pre body spájania 190
16.11 Inštanciácia aspektov 191
16.12 Aspektovo-orientovaná implementácia vzoru Observer . . 196

Literatúra 201

Register 203

Zoznam obrázkov

2.1 Stret rytiera a obra. 5
2.2 Agregácia. 6
2.3 Dedenie — hierarchia obrov. 9

5.1 Hierarchia grafických útvarov. 55
5.2 Hierarchia grafických útvarov s abstraktnou triedou `Utvar`. 56
5.3 Hierarchia grafických útvarov s rozhraniami. 60

13.1 Jednoduché okno. 128
13.2 Okno s tlačidlom. 130
13.3 Okno s dvomi tlačidlami. 131

14.1 Všeobecná asociácia medzi triedami. 141
14.2 Detaily triedy `Kruh`. 142
14.3 Agregácia. 143
14.4 Asociačná rola. 143
14.5 Násobnosť vzťahu. 144
14.6 Príklady násobnosti vzťahu. 144
14.7 Generalizácia. 145
14.8 Abstraktné triedy a operácie. 145
14.9 Realizácia rozhrania. 146
14.10 Detailný diagram tried. 147
14.11 Použitie rozhrania a väzba závislosti. 148
14.12 Hierarchia obrov. 148
14.13 Združená šípka pre generalizáciu. 149
14.14 Vhniezdená trieda. 149
14.15 Diagram objektov. 149
14.16 Príklad diagramu prípadov použitia. 150
14.17 Predregistrácia témy. 151
14.18 Potvrdenie témy. 151

15.1 Model-View-Controller (podľa [Hel]). 155
15.2 Štruktúra vzoru Visitor. 157
15.3 Štruktúra vzoru Observer. 162

Predhovor

Táto kniha vznikla na základe mojich prednášok z rovnomenného predmetu na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave. Ako vo svojich prednáškach, tak aj v tejto knihe som sa snažil vystihnúť princípy objektovo-orientovaného programovania, ale aj poskytnúť úplný obraz jedného objektovo-orientovaného programovacieho jazyka (aj keď kniha predsa predpokladá základné vedomosti z programovacieho jazyka C alebo iného podobného procedurálneho jazyka). Nie náhodou je ním Java, ktorej popularita stále stúpa, a ktorej poznanie je v súčasnosti pre programátora veľkou výhodou.

Hovoriť dnes o objektovo-orientovanom programovaní a nespomenúť aspektovo-orientované programovanie by znamenalo zamlčať najperspektívnejší smer jeho rozvoja. Preto som jednu kapitolu venoval programovaciemu jazyku AspectJ, ktorý predstavuje reálne používané aspektovo-orientované rozšírenie Javy.

Chcel by som poďakovať recenzentom za pozorné preštudovanie rukopisu a cenné pripomienky, ktoré sa týkali aj trochu odvážnejšej terminológie (predovšetkým v oblasti objektovo-orientovaného a aspektovo-orientovaného programovania). Mojim cieľom bolo čo najviac sa priblížiť pôvodnej anglickej terminológii. V texte vždy upozorňujem na ďalšie termíny, ktoré sa používajú pre daný pojem.

Ďakujem Gratex IT Inštitútu, ktorý podporil vydanie tejto publikácie v rámci fondu GraFIIT.

Bratislava, 14. január 2008 Valentino Vranić

1 Úvod

Základným cieľom tejto knihy je oboznámiť čitateľa s objektovo-orientovaným programovaním. Splnenie tohto cieľa vyžaduje uvedenie čitateľa do určitého objektovo-orientovaného programovacieho jazyka. V prípade tejto knihy je to programovací jazyk Java.

Skutočné pochopenie určitého spôsobu programovania vyžaduje jeho praktické vyskúšanie v kontexte ostatných čŕt daného programovacieho jazyka. Preto v mnohom tento text zachádza do detailov Javy, ktoré sa bezprostredne nespájajú s objektovo-orientovaným programovaním, ako sú napríklad anonymné triedy, typické a rozsiahlo využívané v Jave.

Táto kniha sa nevenuje konštrukciám na riadenie vykonávania programu a operátorom, ktoré Java prakticky prevzala z jazyka C. Nevysvetľuje ani tvorbu vnorenej dokumentácie, tvorbu appletov a mechanizmus anotácií, ktoré však nie sú kľúčové pre objektovo-orientované programovanie.

Na druhej strane, objektovo-orientované programovanie je potrebné vnímať v širšom kontexte. Preto táto kniha pootvára dvere do troch ďalších oblastí. Objektovo-orientované programovanie je úzko prepojené s objektovo-orientovanou analýzou a návrhom, ktoré sa už takmer výlučne uskutočňujú v jazyku UML. Preto sa tento text dotýka aj jazyka UML a približuje čitateľovi tie jeho vlastnosti, ktoré patria do repertoára základných vedomostí objektovo-orientovaného programátora.

Kniha sa venuje aj objektovo-orientovaným vzorom, ktorých poznanie je predpokladom úspešného vývoja objektovo-orientovaných aplikácií.

Prostredníctvom tejto knihy čitateľ bude mať príležitosť dozvedieť sa aj o aspektovo-orientovanom programovaní. Ide o prístup, ktorý významne rozširuje možnosti objektovo-orientovaného programovania. V knihe je prezentovaný v súvislosti s jazykom AspectJ, ktorý plynulo nadväzuje na Javu, a ktorý sa používa už aj v praxi.

Pre lepšiu názornosť princípov objektovo-orientovaného programovania v texte sa prelína výklad Javy so všeobecnejším pohľadom na objektovo-orientované programovanie v kontexte objektovo-orientovaného vývoja softvéru ako takého:

- Kapitola 2 poskytuje vhľad do objektovo-orientovaného programovania.
- Kapitola 3 uvádza čitateľa do programovacieho jazyka Java.

- Kapitoly 4–6 vysvetľujú principiálne záležitosti objektovo-orientovaného programovania a ich realizáciu v Jave.
- Kapitoly 7–13 sú venované ďalším prvkom Javy a Java API (rozhrania aplikačného programovania), ktoré sa nedajú označiť priamo za objektovo-orientované, ale sú postavené na objektovo-orientovaných základoch jazyka a je nevyhnutné poznať ich pre plnohodnotné využitie tohto jazyka.
- Kapitoly 14–16 ponesú čitateľa za zdanlivé hranice objektovo-orientovaného programovania — k modelovaniu, vzorom a aspektom.

2 Vhľad do objektovo-orientovaného programovania

Spolu s procedurálnym programovaním objektovo-orientované programovanie predstavuje najpoužívanejší prístup k programovaniu. Táto kapitola približuje objektovo-orientované programovanie prostredníctvom príkladu.

2.1 Vznik objektovo-orientovaného programovania

Podľa Thomasa Kuhna k zmene paradigmy — prevládajúceho vedeckého názoru v danej oblasti — dochádza zlomom, teda revolúciou, nie postupne, evolúciou. K zlomu prichádza, keď sa problémy platnej paradigmy stanú neúnosnými. Klasickým príkladom takého zlomu je zmena z newtonovskej mechaniky na teóriu relativity [Kuh70].

Aj keď sa nástup objektovo-orientovaného programovania často vníma ako zlom paradigmy, korene objektovo-orientovaného programovania sú v skutočnosti v procedurálnom programovaní. Prvý objektovo-orientovaný jazyk, Simula 67, ktorý ako súčasť svojho názvu má rok vzniku, za základ mal procedurálny jazyk Algol 60. Autori Simuly 67, Ole-Johan Dahl a Kristen Nygaard, prví použili pojem objekt v zmysle objektovo-orientovaného programovania.[1]

Programovací jazyk Smalltalk[2] predstavuje významný posun vo vývoji objektovo-orientovaného programovania. Tento jazyk je založený na postuláte „Všetko je objekt.“ a dôsledne ho uplatňuje, takže objektmi sú aj samotné riadiace konštrukcie (ako napríklad **if**). Kým v Simule 67 išlo predovšetkým o organizáciu kódu, v Smalltalku je dôraz na spoluprácu objektov ako entít vo vykonávaní programu, ktorých manipulácia je veľmi pružná vďaka tomu, že ide o interpretovaný jazyk.

[1] `http://staff.um.edu.mt/jskl1/talk.html`
[2] `http://www.smalltalk.org`

2.2 Objekty a triedy

Objektovo-orientované programovanie predstavuje programovanie pomocou objektov. Objekt je entita, ktorá má [Boo94]:

- stav
- správanie
- identitu

Stav objektu zahŕňa všetky vlastnosti objektu a ich hodnoty. Správanie objektu predstavuje konanie objektu pri zmenách stavu a aktivácii operácií, ktoré poskytuje. Identita objektu predstavuje jeho jednoznačnú identifikáciu.

Objektovo-orientovaný program sa uskutočňuje ako *interakcia objektov*. Správanie objektu v objektovo-orientovaných jazykoch so statickými typmi (ako je Java) definuje jeho *typ*. Typ objektu sa označuje ako *trieda* (class). V zdrojových textoch programov sa najčastejšie definujú triedy, a objekty pritom predstavujú ich inštancie. Preto sa niekedy zdá, že ide skôr o programovanie pomocou tried. Iný prístup je v dynamických objektovo-orientovaných jazykoch, v ktorých niekedy úplne absentuje pojem triedy ako šablóny pre tvorbu objektov a objekty sa vytvárajú priamo.

2.3 Stav a správanie objektu

Hovorili sme o objektoch, ich spolupráci a triedach ako šablónach pre tvorbu objektov. Pozrime sa na tieto pojmy bližšie prostredníctvom príkladu.

Predpokladajme, že vytvárame hru. Jeden z objektov v hre je `statocnyRytier`. Jeho stav je daný týmito vlastnosťami:

- `statocnyRytier.poloha` — kde sa `statocnyRytier` nachádza
- `statocnyRytier.energia` — akú `statocnyRytier` má energiu

Bodkou vyjadrujeme to, že dané vlastnosti patria spomínanému objektu. Vlastnosti objektu sa označujú ako *atribúty*.

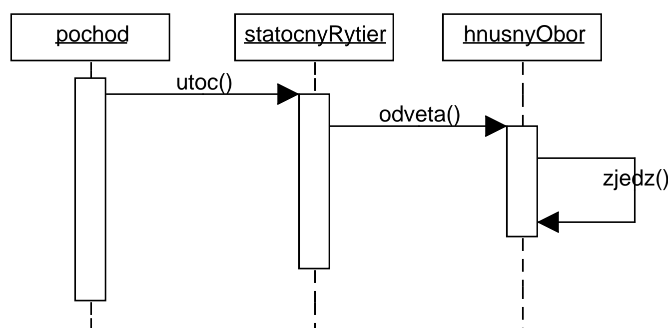
Správanie objektu `statocnyRytier` je dané operáciami, ktoré poskytuje. V našom príklade pôjde o jedinú operáciu: `statocnyRytier.utoc()`. Jej aktivácia spôsobí to, že `statocnyRytier` zaútočí na nepriateľa.

Ďalší z objektov v hre je `hnusnyObor`. Jeho stav je daný týmito vlastnosťami:

- `hnusnyObor.poloha` — kde sa `hnusnyObor` nachádza
- `hnusnyObor.energia` — akú `hnusnyObor` má energiu
- `hnusnyObor.hladny` — či je `hnusnyObor` hladný

Správanie objektu `hnusnyObor` je znovu len jedno: `hnusnyObor.odveta()`. Aktivácia tejto operácie spôsobí odvetu nepriateľovi, ktorý zaútočil na objekt `hnusnyObor`.

Stret rytiera a obra môžeme potom vnímať ako interakciu objektov, ako je to znázornené na obr. 2.1. Interakciu iniciuje objekt `pochod`, o ktorého detaily sa v tomto okamihu nezaujímame. Objekty interagujú prostredníctvom tzv. posielania *správ*, ktoré vlastne predstavuje volanie *operácií*.



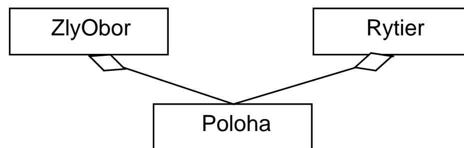
Obrázok 2.1: Stret rytiera a obra.

2.4 Typ objektu

Ako už bolo povedané, správanie objektu definuje jeho typ. Typ objektu sa označuje ako trieda (class). Napríklad `hnusnyObor` predstavuje jedného zo zlých obrov — jeho typ je daný triedou `ZlyObor`. Ďalším príkladom je `statocnyRytier`, ktorý je objektom triedy `Rytier`, alebo `poloha`, ktorá je objektom triedy `Poloha`.

2.5 Agregácia

Triedy `ZlyObor` a `Rytier` obsahujú triedu `Poloha`. Táto skutočnosť je znázornená na obr. 2.2. Takýto spôsob spájania tried do väčších celkov sa označuje ako *agregácia*.



Obrázok 2.2: Agregácia.

Agregácia predstavuje spôsob tvorenia *hierarchie*: agregát, t.j. prvok, do ktorého sa zahŕňajú agregované prvky, je nadradeným prvkom vzhľadom na tieto prvky. Teda, v našom príklade triedy `ZlyObor` a `Rytier` sú nadradené triede `Poloha`.

2.6 Zapuzdrenie

Pokúsme sa bez zavádzania jazykových formalizmov načrtnúť implementáciu triedy `ZlyObor` — jej atribúty a operácie:

```
class ZlyObor {
   Poloha poloha;
   int energia;
   boolean hladny;

   void odveta(Rytier r) {
      if (hladny)
         zjedz(r);
   }

   void zjedz(Rytier r) {
      int e = zistiEnergiu();
      r.uberEnergiu(e);
      zvysEnergiu(e);
   }

   int zistiEnergiu() {
      return energia;
   }

   void zvysEnergiu(int i) {
      energia = energia + i;
   }
```

```
    void znizEnergiu(int i) {
      energia = energia - i;
    }
}
```

Jedným z kľúčových princípov objektovo-orientovaného programovania je *zapuzdrenie* (encapsulation), označované ešte aj ako skrývanie informácií. Tento princíp hovorí, že implementácia objektu má zostať skrytá a prístup k objektu má byť zabezpečený prostredníctvom *rozhrania* (interface), ktoré tvoria vybrané operácie.

Napríklad energia zlých obrov sa nemení priamo siahnutím na atribút `energia`, ale prostredníctvom operácií `zvysEnergiu`() a `znizEnergiu`(). Cudzie objekty by nemali mať možnosť prístupu k tomuto atribútu, na čo — ako uvidíme v nasledujúcej kapitole — jestvujú príslušné jazykové konštrukcie. V uvedenom príklade však pre jednoduchosť tieto konštrukcie nie sú uvedené a princíp zapuzdrenia je porušený aj z iného hľadiska: na zmenu polohy by mali tiež byť poskytnuté zodpovedajúce operácie.

2.7 Dedenie

`ZlyObor` je len jeden možný druh obrov. Niekedy sa potrebujeme vyjadriť o obroch vo všeobecnosti bez ohľadu na špecifické druhy obrov. *Zovšeobecnením* (generalization) obrov by v našom príklade mohla byť trieda `Obor`, ktorá zahŕňa spoločné vlastnosti všetkých obrov:

```
class Obor {
    boolean hladny;
    int energia;

    void odveta(Rytier r) {
      r.znizEnergiu(1);
    }
    int zistiEnergiu() {
      return energia;
    }
    void zvysEnergiu(int i) {
      energia = energia + i;
    }
    void znizEnergiu(int i) {
      energia = energia - i;
    }
}
```

`ZlyObor` potom predstavuje *špeciálizáciu* triedy `Obor`:

```
class ZlyObor extends Obor {

   void odveta(Rytier r) {
      if (hladny)
         zjedz(r);
   }
   void zjedz(Rytier r) {
      int e = zistiEnergiu();
      r.uberEnergiu(e);
      zvysEnergiu(e);
   }
}
```

`ZlyObor` *dedí správanie a štruktúru* triedy `Obor`. Pod štruktúrou väčšinou rozumieme atribúty alebo kód ako taký, kým pod správaním rozumieme funkcionalitu, ktorú implementujú operácie. Inak povedané, trieda `ZlyObor` *rozširuje a konkretizuje* triedu `Obor` — pridáva nové detaily do *abstrakcie* obra. Preto je v Jave kľúčové slovo pre dedenie práve **extends**.

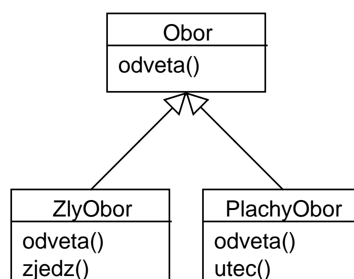
Ďalším príkladom obra môže byť `PlachyObor`, ktorý je, ako vidíme z operácie `odveta`(), naozaj plachý.:

```
class PlachyObor extends Obor {
   void utec() {
      . . .
   }
   void odveta(Rytier r) {
      utec();
   }
}
```

Dôležité je uvedomiť si, že popri agregácii, dedenie predstavuje ďalší spôsob tvorenia hierarchie v objektovo-orientovanom programovaní. Dokonca, keď sa povie len „hierarchia tried", myslí sa na hierarchiu dedenia. Hierarchiu obrov názorne ukazuje obr. 2.3.

2.8 Polymorfizmus

V dedení zďaleka nejde len o zdieľanie spoločného kódu ako sa na prvý pohľad môže zdať. Oveľa dôležitejšie je, že objekt každej triedy, ktorá je v hierarchii dedenia



Obrázok 2.3: Dedenie — hierarchia obrov.

nadradená danej triede, je zároveň aj objektom tejto nadradenej triedy. Presnejšie povedané, typ definovaný podtriedou je podtypom typu definovaným nadtriedou.

V našom príklade aj objekty typu `PlachyObor`, aj objekty typu `ZlyObor` sú typu `Obor` — preto môžeme premenným typu `Obor` priradiť objekty týchto podtypov:

```
Obor o1;
Obor o2;

o1 = new ZlyObor();
o2 = new PlachyObor();
```

Na vytvorenie objektov bolo použité kľúčové slovo **new**. Proces vytvárania objektov je detailne vysvetlený v nasledujúcej kapitole a v tomto okamihu nie je dôležitý.

Nad objektmi `o1` a `o2` môžeme volať všetky operácie definované pre typ `Obor`. Aké však bude správanie v prípade vyvolania operácie, ktorú každý z objektov definuje inak? Príkladom je operácia `odveta`(). Za predpokladu, že premenná `r` predstavuje rytiera, ktorý na obra zaútočil, pri odvete obra `o1`

```
o1.odveta(r);
```

rytier bude zjedený, kým pri odvete obra `o2`

```
o2.odveta(r);
```

sa mu nestane nič, lebo obor `o2` utečie, ako je definované v operáciách `odveta`() príslušných tried, a nie v operácii `odveta`() triedy `Obor`.

Výhoda tohto javu, ktorý sa označuje ako *polymorfizmus* začína byť zrejmá pri vyšších počtoch objektov a dopredu neznámom počte typov. Tak napríklad pre hordu sto

obrov, ktorí nemusia byť len typov `ZlyObor` a `PlachyObor`, pri ich strete s rovnako početnou rytierskou výpravou môžeme napísať jednoducho

```
for (int i = 0; i < 100; i++) {
   obor[i].odveta(r[i]);
}
```

a vieme, že každý obor urobí práve to, čo je pre jeho typ charakteristické. Pre úplnosť treba povedať, že výrazom `obor[]` označujeme pole premenných typu `Obor`, podobne ako v jazyku C, čo bude podrobne vysvetlené v nasledujúcej kapitole. Všimnite si veľmi dôležitú skutočnosť, že nepotrebujeme žiadne podmienené príkazy, a že v prípade zmeny v typoch obrov tento kód vôbec nie je potrebné upravovať.

2.9 Sumarizácia

Skôr ako začneme rozoberať jednotlivé záležitosti objektovo-orientovaného programovania a špecifiká programovacieho jazyka Java, táto kapitola poskytla vhľad do objektovo-orientovaného programovania. Pojmy v tejto kapitole boli definované s istou voľnosťou a príklady kódu boli prezentované bez presnej definície syntaxe a sémantiky, ale ak sa podarilo aspoň trochu demystifikovať problematiku objektovo-orientovaného programovania, kapitola splnila svoj účel.

3 Programovací jazyk Java

Programovanie je presná a formálna záležitosť. Predchádzajúca kapitola pootvorila dvere k objektovo-orientovanému programovaniu, ale nechala príliš veľa nezodpovedaných otázok, aby sme mohli začať s experimentovaním, ktoré je pre zvládnutie programovania kľúčové. Preto sa v tejto kapitole budeme venovať základom programovacieho jazyka Java.

3.1 Java — jazyk a platforma

Java je v zásade prekladaný (kompilovaný) objektovo-orientovaný programovací jazyk. Výsledkom prekladu však nie je strojový kód, ale tzv. bajtkód (bytecode), ktorý sa vykonáva pomocou interpretátora označovaného ako virtuálny stroj Javy (Java Virtual Machnine, JVM).

Prednosťou tohto prístupu je, že sa raz preložený program v Jave dá vykonávať všade, kde je dostupný JVM. Java je teda nezávislá od platformy, lebo Java *je* platforma. Presnejšie povedané platformou je vykonávajúce prostredie Javy (Java Runtime Environment, JRE), ktoré pozostáva z rozhrania aplikačného programovania (Java Application Programming Interface, Java API), JVM a ďalších komponentov potrebných na vykonávanie Javy.

Dostupné sú rôzne druhy JRE vzhľadom na charakteristiky prostredia, v ktorom sa program v Jave má vykonávať:

- Java 2, Micro Edition (J2ME) — pre prostredia s obmedzenými zdrojmi
- Java 2, Standard Edition (J2SE) — pre pracovné stanice
- Java 2, Enterprise Edition (J2EE) - pre rozsiahle distribuované prostredia

Pre vývoj programov v Jave je nevyhnutné mať súpravu pre vývoj softvéru v Jave: Java 2 Software Development Kit. Označuje sa aj skratkou J2SDK alebo aj

JDK a pozostáva z JRE a nástrojov na vývoj, z ktorých je najdôležitejší prekladač (kompilátor).

Vývoj programovacieho jazyka Java začal v Sun Microsystems v roku 1991[1], aj keď samotné meno Java vzniklo až v roku 1995. Jeho pôvod nie je celkom známy, ale najčastejšie sa spája s druhom kávy nazvaným podľa rovnomenného ostrova alebo americkým slangom pre kávu ako takú, odvodeným od názvu tohto druhu kávy. V angličtine sa Java vyslovuje ako *džava*, ale vzhľadom na to, že sa ostrov Java v slovenčine vyslovuje s *j*, pre programovací jazyk Java sú prípustné obidva varianty.

Všetky programovacie jazyky sa postupne menia, ale pre Javu je zmena doslova základnou črtou. V nasledujúcom zozname sú vymenované významné verzie a čo priniesli:

- 1.0 (1996) — uvádzacia verzia
- 1.1 (1997) — rozsiahle zmeny (napr. zavedenie vnútorných tried)
- 1.2 (1998) — vznik tzv. Java 2; v názvoch všetkých ďalších verzií sa vyskytuje číslica 2
- 1.3 (2000) — menšie zmeny a opravy
- 1.4 (2002) — nový vstupno/výstupný systém (NIO)
- J2SE 5.0 (Tiger, 2004) — pôvodne označovaná ako 1.5;[2] významné rozšírenie jazyka
- Java SE 6 (Mustang, 2006) — zmeny v API a vývojových nástrojoch, ale bez rozšírení jazyka
- Java SE 7 (Dolphin) — vo vývoji

Pre Javu jestvuje veľký počet vývojových prostredí. Jedno z najvýznamnejších je prostredie Eclipse[3]. Eclipse je vlastne generické, rozšíriteľné prostredie (pomocou tzv. pluginov), ktoré zďaleka nie je obmedzené na Javu, a dokonca ani na podporu programovania. V tomto prostredí je možné pomocou zodpovedajúcich rozšírení softvér modelovať aj graficky.

Samotné prostredie Eclipse vyvinula firma IBM a uvoľnila ako bezplatný softvér s otvoreným zdrojovým kódom. Jeho vývoj riadi konzorcium firiem prostredníctvom Eclipse Foundation Inc. Na Eclipse sú postavené rôzne komerčné rozšírenia, vrátane komplexného vývojového radu nástrojov Websphere firmy IBM, ktoré zahŕňa aj významný modelovací nástroj IBM Rational Software Modeler.

[1] `http://www.java.com/en/javahistory/`

[2] Sun stále používa aj pôvodný systém označovania verzií na miestach, ktoré sú viditeľné len pre vývojárov.

[3] `http://eclipse.org/`

Ďalšie významné prostredie je NetBeans[4]. Ponúka ho bezplatne priamo Sun Microsystems na svojich stránkach. Z pôvodného integrovaného prostredia pre vývoj v Jave, ktoré sa vyvinulo na základe českého študentského projektu Xelfi, toto prostredie prerástlo do rozšíriteľného prostredia ako Eclipse.

Populárnym vývojovým prostredím je aj JBuilder firmy Borland. JBuilder je platený softvér. Vzhľadom na to, že je Borland členom Eclipse Foundation, Borland pripravuje jeho ďalšiu verziu, ktorá bude založená na Eclipse.

3.2 Objekty v Jave

V zostávajúcej časti kapitoly sa budeme venovať Jave ako programovaciemu jazyku. Syntax Javy je založená na jazykoch C a C++, takže poznanie týchto jazykov značne uľahčuje zvládnutie Javy.

Vzhľadom na to, že Java je objektovo-orientovaný jazyk, objekty v nej zohrávajú kľúčovú úlohu. Pojem objektu sme zaviedli v predchádzajúcej kapitole. V Jave sa k objektom pristupuje vždy cez referenciu. Referencia sa definuje ako premenná typu triedy, ktorej objekt jej má byť priradený.

Jednou z tried v Java API je trieda `String`, ktorá predstavuje typ *reťazec znakov*. Referenciu na objekt typu `String` vytvoríme takto:

```
String s;
```

Objekt sme však týmto nevytvorili. Premenná `s` predstavuje len referenciu. Priradenie prvého objektu referencii sa označuje ako *inicializácia*. Každá referencia musí pred použitím byť inicializovaná, o čom bude reč v časti 3.14.

Spomínanú referenciu `s` môžeme inicializovať priamo pri vytvorení:

```
String s = new String("asdf");
```

Prípadne môžeme špeciálne pre typ `String` použiť skrátený tvar:

```
String s = "asdf";
```

Aj keď v Jave s objektmi pracujeme vždy prostredníctvom referencií, často sa namiesto termínu referencia používa termín objekt, pričom sa týmto myslí na objekt, na ktorý príslušná referencia ukazuje.

[4] `http://www.netbeans.org/`

3.3 Primitívne typy

Programovacie jazyky, ktoré pretendujú na titul „čistého objektovo-orientovaného jazyka," ako napríklad Smalltalk, nepoznajú iný spôsob definície typu než prostredníctvom tried. Tento prístup je transparentný z hľadiska samotného programovania, lebo sa so všetkými premennými narába rovnako, ale môže spôsobovať problémy v zmysle efektivity. Preto napríklad C++ zachováva tzv. primitívne typy pre číselné a znakové údaje. Premenné primitívneho typu sa vytvárajú v zásobníku (stack), a nie na tzv. hromade (heap) ako objekty, čo je efektívnejšie v zmysle časových a pamäťových nárokov.

Java ponúka kompromisné riešenie: poskytuje primitívne typy podobne ako C++, ale pre každý primitívny typ Java API poskytuje aj obaľovaciu triedu (wrapper). Tak je možné vytvoriť celé číslo ako primitívny údaj:

```
int i = 0;
```

ale aj ako objekt:

```
Integer j = new Integer(8);
```

Každý objekt obaľovacej triedy v sebe nesie údaj zodpovedajúceho primitívneho typu. K tomuto údaju v prípade triedy `Integer` možno pristúpiť pomocou operácie `valueInt`():

```
i = j.valueInt() + 5;
```

Údaj uložený v takomto objekte nie je možné meniť. Dôvodom je predovšetkým, aby sa zabránilo problémom s aliasmi, referenciami, ktoré ukazujú na rovnaký objekt. Ak by sme napríklad vytvorili ďalšiu referenciu `k` a nastavili ju na objekt `j`, lebo chceme, aby mala rovnakú hodnotu

```
Integer k = j;
```

vytvorili sme vlastne alias objektu, na ktorý ukazuje referencia `j`. Keby bolo možné meniť stav tohto objektu — teda hodnotu celého čísla, ktorú nesie — zmena by sa prejavila aj prostredníctvom referencie `j`, čo je iné než v prípade primitívneho typu **int**.

Všetky primitívne typy, ktoré Java poskytuje sú:

- **boolean** — boolovská hodnota,
- **char** — 16-bitový znak,
- **byte** — bajt (8 bitov),
- **short** — 16-bitové celé číslo,
- **int** — 32-bitové celé číslo,
- **long** — 64-bitové celé číslo,
- **float** — 32-bitové decimálne číslo,
- **double** — 64-bitové decimálne číslo.

Niekedy sa za primitívny typ pokladá aj **void**, teda prázdny typ. Pomocou tohto typu je možné špecifikovať, že operácia nevracia hodnotu. Nie je možné vytvárať premenné tohto typu.

Názvy obaľovacích typov sú rovnaké ako primitívnych typov, len začínajú veľkým písmenom. Výnimkou sú typy **char**, ktorého obaľovací typ sa volá `Character`, a **int**, ktorého obaľovací typ sa volá `Integer`.

Uvedená veľkosť primitívnych typov je daná špecifikáciou Javy a nezávisí od prostredia, ani od implementácie virtuálneho stroja Javy.

3.4 Triedy

Ako už bolo vysvetlené v predchádzajúcej kapitole, trieda určuje typ objektu, a tým aj jeho správanie. Trieda má názov a telo:

```
class NazovTriedy { /* telo triedy */ }
```

Trieda môže obsahovať atribúty (v Jave *fields*) a metódy. *Atribúty* predstavujú premenné definované v triedach. Implementujú stav objektu príslušnej triedy.

Metódy sú operácie definované v triede. Takéto operácie je možné vykonávať nad objektmi príslušnej triedy alebo nad triedou samotnou. V Jave sú všetky operácie metódy, t.j. každá operácia je definovaná v triede. Jazyk C++ napríklad umožňuje definovať operácie aj mimo tried ako bežné funkcie známe z jazyka C.

Trieda, ktorá obsahuje len atribúty, je podobná štruktúre (**struct**) v jazyku C. Atribúty môžu byť primitívneho typu, ale aj referencie na objekty:

```
class Student {
   String meno;
   boolean zapisany;
}
```

Objekt vytvoríme pomocou operátora **new**:

```
Student s = new Student();
```

K atribútom objektu pristupujeme pomocou operátora . (bodka):

```
s.zapisany = false;
```

Prístup pomocou operátora . funguje aj pre objekty v rámci daného objektu. Majme triedu, ktorá definuje zaradenie študenta:

```
class Zaradenie {
   String fakulta;
   String odbor;
}
```

Rozšírme triedu `Student` o informáciu o zaradení:

```
class Student {
   String meno;
   boolean zapisany;
   Zaradenie zaradenie;
}
```

Zaradenie študenta `s` môžeme nastaviť nasledujúcim spôsobom:

```
s.zaradenie.fakulta = "FIIT";
s.zaradenie.odbor = "Informatika";
```

Inicializácia atribútov na implicitné hodnoty je zabezpečená, ale lokálnych premenných (v metódach) nie je. Prekladač však hlási chybu pri pokuse o prístup k neinicializovanej premennej. Viac o inicializácii budeme hovoriť v časti 3.14.

Metódy sú podobné funkciám v jazyku C. Základná syntax metódy v Jave je nasledujúca:

```
Typ nazovMetody(/* parametre */) {
   /* telo */
}
```

Kľúčové slovo **return** v metóde slúži na vrátenie hodnoty:

```
int vynasob(int i, int j) {
   return i * j;
}
```

Po kľúčovom slove **return** sa metóda skončí. Kľúčové slovo **return** sa dá použiť aj výlučne za účelom ukončenia metódy, ktorá nevracia hodnotu:

```
void nothing() { return; }
```

Atribúty a metódy môžu byť *statické*. Statické atribúty jestvujú ako súčasť triedy a zdieľajú ich všetky objekty danej triedy. Majme napríklad triedu so statickým celočíselným poľom:

```
class A {
   static int i = 1;
}
```

Vytvorme dva objekty tejto triedy:

```
A a1 = new A();
A a2 = new A();
```

Príkazy

```
a1.i++;
a2.i++;
A.i++;
```

inkrementujú ten istý atribút a po ich vykonaní `i` bude mať hodnotu 4.

Podobne prístupu k statickým atribútom, statické metódy sa dajú volať priamo, bez vytvárania objektu. Rozšírme triedu `A` z predchádzajúceho príkladu o statickú metódu na inkrementáciu jej statického poľa:

```
class A {
   static int i = 1;
   static void incr() { i++; }
}
```

Volanie statickej metódy uskutočníme prostredníctvom triedy:

```
A.incr();
```

Statické metódy sa dajú volať aj prostredníctvom objektov — účinok je rovnaký:

```
A a = new A();
a.incr();
```

Statické metódy môžu pristupovať len k statickým atribútom, lebo nepôsobia v kontexte objektu. Prekladač pri pokuse o prístup k nestatickému poľu zo statického kontextu hlási chybu:

```
class C {
   int i;
   static void incr() {
      i++; // chyba!
   }
}
```

V nestatických metódach je dostupná aj referencia na aktuálny objekt prostredníctvom kľúčového slova **this**. Referencia **this** sa používa na vrátenie referencie na aktuálny objekt.

```
class C {
   . . .
   C m() {
      if (...)
         return new C();
      else
         return this;
   }
}
```

Ďalšie významné použitie referencie **this** je pre rozlíšenie medzi formálnym parametrom a poľom objektu ako v nasledujúcom príklade:

```
class Student {
   int rocnik;
   . . .
   void nastavRocnik(int rocnik) {
      this.rocnik = rocnik;
   }
}
```

Tým, že trieda definuje typ sa niekedy namiesto označenia trieda používa označenie typ. Tento termín zahŕňa aj tzv. rozhrania (interfaces), ktoré budú vysvetlené v časti 5.4. Zatiaľ ich môžeme brať ako špeciálne triedy. V tomto význame termín typ nezahŕňa primitívne typy.

3.5 Uvoľňovanie pamäte

Pamäť, ktorú zaberajú premenné primitívnych typov, sa uvoľňuje ako aj v iných programovacích jazykoch, keď tok programu opustí rozsah (scope), v ktorom je daná premenná deklarovaná. Rozsah platnosti je vymedzený skupinovými zátvorkami, napríklad:

```
{
  String s = new String("a string");
} // odtiaľto s už nie je
```

V Jave sa na rozdiel od niektorých iných objektovo-orientovaných jazykov, ako je napríklad C++, objekty nerušia explicitne. Objekty, na ktoré sa neodvoláva, zruší a pamäť, ktorú zaberali, uvoľní tzv. zberač smetí (garbage collector).

3.6 Prvý program

Teraz už môžeme vytvoriť prvý program v Jave. Niekoľko prvkov v ňom predsa bude neznámych, ale podstatné časti by na základe doterajšieho výkladu mali byť jasné. Náš prvý program bude veľmi jednoduchý:

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
```

```
    }
}
```

Aby sme mohli program vyskúšať, musíme ho prepísať v editore textu a uložiť do súboru, ktorého názov je rovnaký ako názov triedy, s príponou `java`, t.j. `HelloWorld.java`.

Za predpokladu, že máme nainštalovaný JDK (Java 5.0 alebo vyššiu verziu), preklad spustíme príkazom

```
javac HelloWorld.java
```

Použili sme `javac`, štandardný prekladač Javy. Program následne spustíme príkazom:

```
java HelloWorld
```

Program pozostáva z jednej triedy, ktorá obsahuje jednu statickú metódu. Ako bude vysvetlené ďalej, táto metóda je špeciálna a umožňuje spustenie programu. Program pomocou metódy `println`() z Java API vypíše pozdrav a skončí.

Metóda `println`() je deklarovaná v triede `PrintStream`, a `out` je statický atribút triedy `System`, ktoré predstavuje referenciu na objekt triedy `PrintStream`.

3.7 Zdrojové súbory a preklad

Na základe skúsenosti s inými prekladanými programovacími jazykmi by sa dalo očakávať, že po preklade programu v Jave vznikne jeden vykonateľný súbor. V Jave každý zdrojový súbor (s príponou `java`) obsahuje jednu alebo viac tried a každý zdrojový súbor sa prekladá zvlášť. Pri preklade pre každú triedu vznikne súbor s bajtkódom uložený v súbore, ktorého názov je rovnaký ako názov triedy s príponou **class**.

Pri preklade je potrebný bajtkód tried, ktoré sa v ňom používajú, pričom ho `javac` vytvorí sám zo zdrojových súborov, ak sú dostupné.

Ak trieda obsahuje metódu `main`(), takýto súbor sa dá spustiť. Každá trieda môže obsahovať metódu `main`(), čo znamená, že program môže mať viac vstupných bodov.

3.8 Balíky

Triedy možno chápať aj ako spôsob organizácie kódu: kód, ktorý súvisí, je v spoločnej jednotke. Niektoré triedy tiež súvisia viac ako iné a tak vzniká potreba zoskupovania

samotných tried. V Jave je možné triedy zoskupovať do *balíkov* (packages).

To, že trieda patrí do nejakého balíka, sa deklaruje príkazom **package**.

```
package nazov_balika;
```

Ak je uvedená, deklarácia balíka musí byť prvým príkazom. Všetky triedy mimo explicitne deklarovaných balíkov patria do jedného, implicitného balíka s čím súvisí prístup k prvkom balíka, o čom budeme hovoriť v nasledujúcej časti.

Balíky sú organizované hierarchicky — každý balík môže obsahovať ďalšie balíky. Umiestnenie balíka v hierarchii sa vyznačuje nasledujúcim spôsobom:

```
package nadnadbalik.nadbalik.balik;
```

Ak chceme používať triedy deklarované v inom balíku, musíme ho zaviesť príkazom **import**. Pritom musíme uviesť celú cestu k balíku:

```
import nadnadbalik.nadbalik.balik.*;
```

Hviezdička na konci znamená, že zavádzame všetky typy, ktoré sú deklarované v balíku. Typy môžeme zavádzať aj jednotlivo:

```
import java.util.ArrayList;
```

Zavádzanie balíkov môže pripomínať pridávanie súborov v jazyku C príkazom `include`, ale medzi týmito dvoma prístupmi je podstatný rozdiel. Kým sa v jazyku C pridané súbory pri preklade fyzicky pripoja na miestach príkazov `include`, v Jave sa zavedením balíka prekladaču len sprístupní priestor názvov, aby sa nemuseli uvádzať plne vymedzené názvy typov. Napríklad, v našom programe by sme bez zavedenia balíka `java.util` metódu `println`() museli vyvolať takto:

```
java.util.ArrayList zoznam;
```

Pri zavádzaní balíkov nedochádza k nárastu preloženého kódu, ani k spomaleniu prekladu. Plne vymedzené názvy sa však musia používať pri kolíziách názvov — ak použijeme typy s rovnakým názvom deklarované v dvoch rôznych balíkoch, prekladač hlási chybu.

Balík `java.lang` a jeho podbalíky nie je potrebné zavádzať, lebo sa to deje automaticky. Tento balík obsahuje triedu `System`, ktorú sme používali v doterajších príkladoch.

Od verzie 5.0 v Jave sa príkazom **import** môže zaviesť statický prvok daného typu. Ako sme už hovorili, `out` je statickým poľom triedy `System`. Ak uvedieme nasledujúcu deklaráciu:

```
import static java.lang.System.out;
```

metódu println() môžeme vyvolať príkazom

```
out.println("Hello, world!");
```

Zavedenie statického prvku je zvlášť výhodné pre matematické funkcie. Ak deklarujeme:

```
import static java.lang.Math.*;
```

môžeme v programe napísať napríklad:

```
out.println(abs(cos(2*PI/3)));
```

kým by pôvodný, neskrátený zápis vyzeral takto:

```
out.println(Math.abs(Math.cos(2*Math.PI/3)));
```

Preložené súbory musia byť zaradené do adresárovej štruktúry, ktorá zodpovedá hierarchii balíka. Napríklad súbory balíka

```
package nadnadbalik.nadbalik.balik;
```

musia byť v adresári

```
nadnadbalik/nadbalik/balik
```

inak sa program nebude dať spustiť.

Začiatok cesty je daný tzv. *cestou k triedam* (classpath), ktorá obsahuje všetky adresáre, v ktorých JVM má hľadať triedy. Cesta k triedam sa definuje ako premenná prostredia (rovnako ako premenná `path`) alebo sa zadáva priamo pri spúšťaní prekladu prekladačom `javac` za prepínačom `−classpath`.

3.9 Riadenie prístupu

V predchádzajúcej kapitole sme hovorili o zapúzdrení ako jednom z kľúčových princípov objektovo-orientovaného programovania. V Jave je tento princíp realizovaný prostredníctvom *modifikátorov prístupu* (access modifiers alebo access specifiers). Uvedením jedného z modifikátorov prístupu **public**, **protected** alebo **private** pred deklaráciou prvku určujeme typ prístupu k nemu. Neuvedenie modifikátora prístupu znamená implicitný prístup v rámci balíka (package access).

Vhodné je rozlišovať medzi deklarovaním prístupu k prvkom balíka, t.j. k typom (triedy a rozhrania), a k prvkom typov, t.j. k atribútom a metódam. Prvkami tried, ako uvidíme v kapitole 7, môžu byť aj typy (tzv. vhniezdené typy).

Prístup k prvkom balíka

Prístup k prvkom balíka môže byť **public**, čo znamená, že je daný prvok prístupný všetkým prvkom vo všetkých balíkoch, alebo v rámci balíka, čo znamená, že je prvok prístupný len prvkom balíka, v ktorom je definovaný.

Typy deklarované v úplne nezávislých zdrojových súboroch, pre ktoré nie je uvedený balík, sú jedny druhým prístupné, lebo sú — ako bolo vysvetlené v časti 3.8 — súčasťou jedného, implicitného balíka.

Najviac jedna **public** trieda sa môže vyskytovať v rámci jedného zdrojového súboru, pričom súbor a trieda musia mať rovnaký názov.

Prístup k prvkom typov

K prvkom typov sú možné štyri typy prístupu: prístup v rámci balíka, **private**, **protected** a **public**. Prístup v rámci balíka je implicitný, t.j. platí, ak sa neuvedie modifikátor prístupu. V takom prípade je prvok dostupný len typom v rámci jeho balíka. Podobne ako pri prvkoch balíka, prvky deklarované v úplne nezávislých zdrojových súboroch, pre ktoré nie je uvedený balík, sú jedny druhým prístupné.

Prvok s prístupom **private** je prístupný len v rámci triedy samotnej. Tento prístup je vhodný, keď je potrebné zabrániť priamemu prístupu k niektorým atribútom alebo volaniu niektorých metód zvonku.

Prvok s prístupom **protected** je prístupný v rámci balíka a v rámci *hierarchie tried*, do ktorej daná trieda patrí. Tento prístup má veľký význam v súvislosti s dedením (inheritance), čo bude vysvetlené v časti 4.2.

Prvok s prístupom **public** je prístupný všade bez obmedzenia. Tento prístup je vhodný pre metódy, ktoré tvoria rozhranie danej triedy voči jej koncovému používateľovi.

Nasledujúci príklad demonštruje modifikátory prístupu **private** a **public**:

```
class Student {
   private String meno;
   . . .
   public void nastavMeno(String m) {
      meno = m;
  }
  . . .
}

class Zapis {
   public static void main(String[] args) {
      Student s = new Student();
      s.nastavMeno("Peter Petrovič");
      s.meno = "Milan Petrovič"; // chyba pri preklade!
}
```

Vidíme, že atribút `meno` je chránený pred prístupom z iných tried. V tomto prípade je to aj vhodné, lebo tak skrývame vnútornú reprezentáciu mena študenta. V budúcnosti by sme sa mohli napríklad rozhodnúť, že pre priezvisko zavedieme samostatný atribút. Následne by sme museli zmeniť všetok kód, ktorý pristupoval priamo k poľu `meno`. Podrobnejšie vysvetlenie tohto problému je v časti 6.3.

3.10 Komentár a vnorená dokumentácia

Veľmi dôležitou súčasťou každého programu je jeho dokumentácia. Dokumentácia môže byť externá alebo vnútorná. Prednosťou vnútornej dokumentácie je možnosť bezprostredného opisu relevantných častí programu priamo v kóde. Väčšina programovacích jazykov podporuje vkladanie komentára a rovnako je to aj v Jave. Java vlastne priamo prebrala syntax pre komentár z jazyka C++:

```
// Jednoriadkovy komentar

/* Bezny komentar
   (ako v jazyku C)
*/
```

Nie vždy však pri študovaní programov potrebujeme vidieť úplný kód. Napríklad v Jave by nám niekedy stačili deklarácie tried, atribútov a metód bez samotných tiel. K tomu by bolo vhodné mať aj príslušný komentár. Presne toto umožňuje tzv. vnorená dokumentácia v Jave.

V rámci štandardného komentára sú vybranými značkami vyčlenené časti, ktoré sa majú objaviť v dokumentácii. Nástroj `javadoc` tieto údaje povyberá a vygeneruje dokumentáciu v HTML. V tomto texte sa nebudeme zaoberať detailami tvorby vnorenej dokumentácie.

3.11 Operátory v Jave

Operátory v Jave sú väčšinou rovnaké ako v jazyku C. Preto v ďalšom texte sa zameriame na rozdiely medzi operátormi v Jave a C.

Operátory v Jave fungujú väčšinou len pre primitívne typy. Operátory =, == a != je možné použiť aj pre objekty. Pre triedu `String` je možné použiť ešte aj operátory + a +=.

Priorita operátorov je podobná ako v jazyku C. Pri hocijakej pochybnosti je najlepšie použiť zátvorky, a preto tu nebudeme ďalej rozvádzať prioritu jednotlivých operátorov.

Priradenie

Operátor priradenia (=) sme už spomínali v časti 3.3. Pre premenné primitívnych typov funguje očakávaným spôsobom: priraďujú sa hodnoty.

Pri objektoch sa priraďujú referencie — nie hodnoty. Keďže s objektmi v Jave pracujeme výlučne prostredníctvom referencií, ide vlastne tiež o priraďovanie hodnôt, ale hodnotou je vlastne referencia. Referencie po priradení ukazujú na ten istý objekt a zmena cez hociktorú z nich vplýva na tento objekt.

Aby sme vyskúšali priraďovanie objektov, vytvoríme nasledujúcu jednoduchú triedu:

```
class Number {
  int i;
}
```

Niekde v programe vytvoríme dve referencie typu `Number`:

```
Number n1 = new Number();
Number n2 = new Number();
```

Týmto objektom nastavíme rozdielne hodnoty poľa `i`:

```
n1.i = 1;
n2.i = 2;
System.out.println(n1.i + " " + n2.i); // 1 2
```

o čom sa presvedčíme aj kontrolným výpisom.

Povedzme, že v nejakom bode programu potrebujeme, aby objekt `n1` nadobudol hodnotu objektu `n2`. Naivne použijeme priradenie:

```
n1 = n2;
System.out.println(n1.i + " " + n2.i); // 2 2
```

a kontrolný výpis nám zatiaľ neprezradil problém, ktorý vznikol.

Ak teraz nastavíme atribút `i` objektu `n1` na inú hodnotu:

```
n1.i = 3;
System.out.println(n1.i + " " + n2.i); // 3 3
```

zistíme, že sa mení aj hodnota poľa `i` objektu `n2`. Priradenie

```
n1 = n2;
```

spôsobilo nastavenie referencie `n1` na `n2`. Referenciu na pôvodný objekt `n1` sme stratili (ak sme ju predtým nepriradili inej premennej). Aj keď samotný objekt nebol zrušený, nemáme už možnosť pristúpiť k nemu a časom ho zruší zberač smetí.

Pri volaniach metód, ktorých parametre sú objekty, prenášajú sa vlastne referencie, nie objekty. Nech je daná nasledujúca metóda:

```
static void f(Number n) {
   n.i = 0;
}
```

Vytvoríme objekt triedy `Number` a nastavíme jeho atribút `i` na 1:

```
Number x = new Number();
x.i = 1;
```

Po volaní metódy `f` s parametrom `x`, hodnota poľa `i` sa zmení na 0.

```
f(x);
System.out.println(x.i); // 0
```

Aritmetické operátory

Aritmetické operátory v Jave sú rovnaké ako v jazyku C. Môžu sa používať len pre premenné číselných typov: **short**, **int**, **long**, **float** a **double**. Základné operácie sú umožnené prostredníctvom nasledujúcich operátorov: + (sčítanie), − (odčítanie), ∗ (násobenie), / (delenie) a % (modulo). Môžeme napríklad napísať

```
a = a + 5;
```

za predpokladu, že je premenná `a` niektorého z číselných typov.

Je možný aj skrátený zápis v tvare

op=

kde *op* je jeden z operátorov základných operácií. Predchádzajúci výraz môžeme zapísať v skrátenom tvare takto:

```
a += 5;
```

Operátory + a − môžu vystupovať aj ako unárne:

```
a = −5;
```

Na zjednodušenie zápisu tzv. inkrementácie a dekrementácie Java poskytuje operátory ++ a −−. Inkrementácia znamená zvýšenie číselnej hodnoty o jeden a dekrementácia jej zníženie o jeden. Tieto operátory sa s výhodou používajú v slučkách.

Relačné operátory

Relačné operátory v Jave sú tiež prevzaté z jazyka C: == (rovnosť), != (nerovnosť), > (väčšie), < (menšie), >= (väčšie alebo rovné) a <= (menšie alebo rovné). Výsledok porovnávania je typu **boolean**.[5]

Dôležité je uvedomiť si, že sa pri objektoch porovnávajú len referencie, nie samotné objekty. Na porovnávanie obsahu objektov jestvuje metóda `equals()` vysvetlená v časti 3.15.

Logické operátory

Logické operátory v Jave sú: & (logické a), | (logické alebo) a ! (logické nie). Môžu sa používať len pre boolovské hodnoty a výsledok logických operácií je tiež boolovská hodnota.

[5] Na rozdiel od jazyka C, v ktorom je výsledok porovnávania typu **int**.

K operátorom & a | jestvujú aj tzv. podmienené verzie && a ||. Logický výraz s týmito operátormi sa vyhodnocuje len kým nie je jasná jeho hodnota. Preto treba byť opatrný, ak sa vyhodnocovanie logických výrazov uskutočňuje pre tzv. vedľajšie účinky (side effects). Predpokladajme, že máme takúto metódu:

```
static boolean test(int val) {
   System.out.println("Porovnávam" + val);
   return val < 2;
}
```

a niektorej z metód tej istej triedy uvedieme nasledujúci podmienený výraz:

```
if (test(3) && test(1))
   ; // Porovnávam 3
```

Správa o porovnávaní sa vypísala len pre prvý výraz, lebo bol nepravdivý, čím bolo hneď jasné, že celá konjunkcia musí byť nepravdivá.

Operátory na bitoch

Operátory na bitoch sú logické a posúvacie. Môžu sa používať len pre celočíselné hodnoty a výsledok logických operácií je tiež celočíselná hodnota. Aj pri týchto operátoroch — okrem unárneho operátora ~ — je možné používať skrátený zápis *op*=.

Logické operátory na bitoch sú rovnaké ako v jazyku C: & (a), | (alebo), ^ (exkluzívne alebo) a ~ (bitový komplement).

Java pozná dva posúvacie operátory na bitoch prevzaté z jazyka C — << (bitový posun doľava) a >> (bitový posun doprava) — a navyše aj operátor >>>, ktorý predstavuje bitový posun doprava bez ohľadu na znamienko (t.j. nuly sa vkladajú zľava začínajúc najvyšším bitom bez ohľadu na znamienko).

Použitie posúvacích operátorov na bitoch na typy **char**, **byte**, **short** a **int** dáva výsledok typu **int**, a ich použitie na typ **long** dáva výsledok typu **long**. Pri použití posúvacích operátorov na bitoch na typy na **byte** a **short** a priradení výsledku do premennej typu menšieho než **int** dochádza k orezaniu hodnoty. Preto treba byť opatrný pri použití skrátených verzií týchto operátorov.

Ternárny `if-else` operátor

Ternárny **if−else** operátor je tiež prevzatý z jazyka C. Zoberme do úvahy napríklad nasledujúci výraz s ternárnym **if−else** operátorom:

```
return i < 10 ? i * 100 : i * 10;
```

Tento výraz je ekvivalentný s nasledujúcim podmieneným príkazom:

```
if (i < 10)
   return i * 100;
else
   return i * 10;
```

Ternárny **if−else** operátor môže znižovať čitateľnosť programu, a preto ho treba používať rozvážne. Niekedy však umožňuje koncíznejšie vyjadrenie. Pomocou ternárneho **if−else** operátora môžeme uskutočniť podmienenú inicializáciu poľa priamo pri jeho deklarácii (inicializácii je venovaná časť 3.14):

```
class A {
   static int c;
   int i = c < 0 ? 0 : c;
}
```

Ďalšie operátory

Operátor , (čiarka) sa v Jave používa len vo **for** slučkách na oddelenie inicializácií a definícií krokov:

```
for (int i = 0, j = 0; i < 5; i++, j = i * 2) {
   . . .
}
```

Operátor spájania reťazcov + sme vlastne už používali v predchádzajúcich príkladoch pri výpisoch ako napríklad:

```
System.out.println("Porovnávam" + val);
```

Pri vykonávaní sa hodnota premennej `val` transformuje na reťazec znakov. Výsledkom operácie spájania je nový objekt typu `String`, ktorý metóda `println()` aj očakáva.

Operátor zmeny typu (cast) umožňuje explicitnú zmenu typu. Tento operátor možno používať pri primitívnych typoch, ale aj pri triedach ako uvidíme v kapitole 5. Väčšinou nepotrebujeme používať operátor zmeny typu, lebo ju prekladač zabezpečí:

```
int i = 1;
long l = (long)i;
long l = i;
```

Posledné dva riadky sú rovnocenné.

Číselná promócia

Pri aritmetických operáciách dochádza k tzv. číselnej promócii. Výsledok aritmetickej operácie alebo operácií na bitoch je aspoň typu **int** alebo najväčšieho zo zúčastnených typov.

Pri priradení do pôvodného typu, treba explicitne zmeniť typ pomocou operátora zmeny typu:

```
byte b1 = 1, b2 = 2;
b1 = b1 + b2; // chyba pri preklade!
b1 = (byte)(b1 + b2); // OK
```

Pri použití skráteného zápisu `op=`, zmena je implicitná

```
b1 += b2; // OK
```

Literály

Literály sú hodnoty zadávané priamo v programe. Prekladač väčšinou dokáže určiť typ týchto hodnôt, ale niekedy ho musíme uviesť explicitne:

```
char c = 0xffff; // max. hodnota pre char hexadecimalne
int i3 = 0177; // oktálna hodnota (začína nulou)
float f4 = 1e-45f; // exponenciálny zápis
```

3.12 Riadenie vykonávania programu

Konštrukcie na riadenie vykonávania programu sú rovnaké ako v jazyku C:

- **if-else**
- **return**

- **while**
- **do-while**
- **for**
- **break** a **continue**
- **switch-case**

Java má tiež návestia, ale nemá príkaz **goto**. V tomto texte sa nebudeme venovať detailom riadenia vykonávania programu.

3.13 Preťaženie metód

Dve metódy tej istej triedy môžu niesť rovnaký názov, ak sa líšia v type parametrov (poradie parametrov je významné). Prekladač na základe zadaných parametrov vyberie správnu metódu. Tento jav sa označuje ako preťaženie (overloading) a takéto metódy ako preťažené. [6]

Nasledujúci príklad demonštruje preťaženie. Majme triedu `Student` s dvoma metódami pre nastavenie údajov.

```
class Student {
   String meno;
   boolean zapisany;
   int rocnik;
   void nastavenie(String m) {
      meno = m;
   }
   void nastavenie(String m, boolean z) {
     meno = m;
     zapisany = z;
   }
}
```

Údaje niektorých študentov zadávame priamo pri zápise, takže môžeme im hneď nastavovať aj príznak, že sú zapísaní:

```
Student a = new Student();
a.nastavenie("Jozef Kohut", true);
```

[6] Presnejšie povedané preťažené sú názvy metód [GJSB05]

Niekedy však len zadávame mená študentov bez toho, aby sme ich aj formálne zapísali:

```
Student b = new Student();
b.nastavenie("Jana Petrová");
```

Návratová hodnota sa nedá použiť na rozlíšenie medzi preťaženými metódami. Dôvod je ten, že sa aj metódy, ktoré vracajú hodnotu, niekedy volajú len pre vedľajšie účinky, a návratová hodnota sa nepriraďuje do žiadnej premennej. Inokedy návratová hodnota býva pretypovaná. Prekladač by v takých prípadoch nemohol určiť očakávaný typ návratovej hodnoty, a tým ani metódu, ktorá sa má vyvolať.

Metóda `println()` predstavuje zaujímavý príklad preťaženia v Java API. V časti 3.11 sme videli, že vďaka automatickej zmene typu na reťazec pri použití operátora spájania reťazcov (+) môžeme jednoducho kombinovať text a hodnoty premenných. Metóda `println()` však vypíše čokoľvek — aj samostatné hodnoty premenných:

```
int a = 5;
println(a);
```

čo je dosiahnuté pomocou preťaženia. Ak zalistujeme v dokumentácii k Java API, ktorá je dostupná vo webovom sídle Sunu, nájdeme množstvo preťažených metód `println()`:

```
println()
println(boolean)
println(char)
. . .
println(java.lang.Object)
println(java.lang.String)
```

Číselná promócia, o ktorej sme hovorili v časti 3.11, platí aj pri volaní metód. Pri preťažených metódach sa vyberie metóda, ktorej veľkosť typu formálneho parametra je najbližšia skutočnému. Ak je parameter len jeden, situácia je jasná. Čo však, keď je parametrov viac? Zodpovedajúca metóda sa nájde len ak je priradenie formálnych k skutočným parametrom[7] jednoznačné. Nasledujúci príklad ozrejmí posledné tvrdenie:

```
class Pretazenie {
   static void m(int i, long l) {
      System.out.println("A");
   }
```

[7] Skutočné paramtre sa označujú aj ako argumenty.

```
    static void m(long l, int i) {
        System.out.println("B");
    }

    public static void main(String[] args) {
        byte b = 8;
        short s = 16;
        int i = 32;
        long l = 64;

        m(b, l); m(s, l); m(i, l); // A
        m(l, b); m(l, s); m(l, i); // B
//      m(b, b); m(b, s); m(b, i); // ambiguous reference
//      m(s, b); m(s, s); m(s, i); // ambiguous reference
//      m(i, b); m(i, s); m(i, i); // ambiguous reference
//      m(l, l); // cannot resolve
    }
}
```

Vidíme, že v posledných desiatich volaniach prekladač nevedel vybrať metódu. V takýchto prípadoch prekladač hlási chybu, že je odkaz na metódu nejednoznačný.

3.14 Inicializácia, konštruktory a finalizácia

Správna inicializácia je v programoch nevyhnutná. Časť inicializácie sa odohráva prostredníctvom tzv. konštruktorov pri vytváraní objektov, ale ako uvidíme, možné sú aj ďalšie spôsoby inicializácie.

Inicializácia

Nechcené vynechanie inicializácie sa ťažko odhaľuje, čo je v jazyku C jeden z častých problémov. Java poskytuje mechanizmy na predchádzanie tomuto problému. Premenné v metódach musia byť explicitne inicializované pred použitím:

```
void f() {
    int i;
    i++; // chyba pri preklade!
}
```

Inicializácia však nemusí byť vykonaná hneď pri deklarácii premennej:

```
void f() {
   int i;
   . . .
   i = 10; // inicializácia
   . . .
   i++;
}
```

Atribúty sa inicializujú priamo pri ich deklarácii:

```
class CInit {
   int i = 1;
   int j = 2;
}
```

Ak sa neinicializujú explicitne, nastavia sa na implicitné hodnoty. Inicializácia prebehne pri načítaní triedy.

Inicializácia nemusí byť vykonaná konštantou:

```
class CInit {
   int i = f();
   int j = g(i);

   void m() {
      int x = f();
      . . .
   }

   int f() { . . . }
   int g(int i) { . . . }
}
```

Poradie inicializácie atribútov je významné:

```
class CInit {
  int j = g(i); // chyba pri preklade!
  int i = f();
  . . .
}
```

Konštruktory

Konštruktor je špeciálna operácia na inicializáciu objektu volaná pri jeho vytváraní. Konštruktor vyzerá ako metóda, ale bez špecifikácie návratovej hodnoty. Konštruktor vždy nesie názov triedy, v ktorej sa nachádza.

Doteraz sme už vytvorili viac objektov, takže sme vlastne aj vyvolávali konštruktory. Volanie konštruktora sa uvádza za kľúčovým slovom `new`:

```
String s = new String("asdf");
```

Ak programátor konštruktor neposkytne, ako v nasledujúcom príklade:

```
class Student {
   private String meno;
   private int rocnik;
}
```

kompliátor poskytne implicitný (prázdny) konštruktor:

```
Student s = new Student();
```

Pri poskytnutí hocijakého konštruktora, ako napríklad:

```
class Student {
   private String meno;
   private int rocnik;
   public Student(int r) {
     rocnik = r;
   }
}
```

implicitný konštruktor sa už nedá použiť:

```
Student s = new Student(); // chyba pri preklade!
```

Ak predsa potrebujeme aj konštruktor bez parametrov, musíme ho uviesť explicitne, hoci aj s prázdnym telom:

```
class Student {
   private String meno;
   private int rocnik;
   public Student() { }
   public Student(int r) {
      rocnik = r;
   }
}
```

Posledný príklad poukázal aj na to, že konštruktory tiež môžu byť preťažené. Preťaženie je vlastne jediný spôsob, ako poskytnúť viac koštruktorov v jednej triede.

Konštruktor sa nedá volať priamo — okrem v iných konštruktoroch tej istej triedy. Takéto volanie sa uskutočňuje pomocou kľúčového slova **this**. Dá sa volať len jeden ďalší konštruktor a len na začiatku konštruktora, ktorý ho volá. Ak sa napríklad pre novoevidovaného študenta predpokladá prvý ročník, môžeme využiť jestvujúci konštruktor, ktorý inicializuje ročník:

```
class Student {
   private String meno;
   private int rocnik;
   public Student() {
      this(1);
   }
   public Student(int r) {
      rocnik = r;
   }
}
```

Aj keď sa uvedené volanie konštruktora v tomto príklade môže javiť ako zbytočné a priame zopakovanie jeho tela ako priamočiarejšie riešenie, volanie konštruktora zvyšuje stabilitu programu vzhľadom na možnosť zmeny vnútornej reprezentácie.

Inicializácia atribútov v konštruktore vlastne ani nie je ozajstná inicializácia, lebo inicializácia (aspoň implicitnými hodnotami) už prebehla pri načítaní triedy. Konštruktory poskytujú väčšiu flexibilitu pri výbere iniciálnej hodnoty pre daný objekt prostredníctvom parametrov, podmienených príkazov (aj keď sa aj pri inicializácii atribútov dá použiť ternárny **if−else** operátor) a slučiek (vhodné pre polia).

Inicializácia blokom príkazov je ďalší spôsob inicializácie, ktorý prebieha až po prvotnej inicializácii atribútov. Blok príkazov v triede umožňuje definovať veci, ktoré sa majú vykonať pri každom vytvorení nového objektu:

```
class C {
   {
```

```
        System.out.println("Nový objekt");
    }
}
```

Každé vytvorenie objektu vypíše oznam:

```
new C(); // Nový objekt
new C(); // Nový objekt
```

Statická inicializácia blokom príkazov umožňuje definovať veci, ktoré sa majú vykonať pred prvým použitím triedy (pri jej načítaní). Bloku príkazov na statickú inicializáciu predchádza kľúčové slovo **static**:

```
class C {
    static {
        System.out.println("Trieda načítaná");
    }
}
```

K načítávaniu triedy dochádza priamo pred jej prvým použitím, takže sa správa vypíše iba raz:

```
new C(); // Trieda načítaná
new C();
```

Finalizácia

Už sme spomínali, že sa objekty v Jave nerušia priamo. Objekty, na ktoré neukazuje žiadna referencia, vyzbiera zberač smetí. Niekedy je potrebné pri rušení objektov uskutočniť určité operácie. Toto sa označuje ako finalizácia a poskytuje sa ako špeciálna metóda

public void `finalize`()

Finalizáciu možno využiť pre uvoľnenie pamäte obsadenej inak než tvorbou inštancií tried: Java umožňuje volať kód napísaný v jazykoch C a C++, prostredníctvom čoho je možné vyvolať funkciu `malloc`().

Vyvolanie zberača smetí a tým aj finalizácia nie sú garantované, čo znamená, že sa môže stať, že program skončí, ale zberač smetí sa nevyvolá (lebo bol dostatok pamäte) a finalizácia sa neuskutoční.

3.15 Polia

Polia (arrays) ako indexované zoznamy hodnôt rovnakého typu sú známe z procedurálnych jazykov.[8] Pozrime sa, ako sú polia implementované v Jave.

Vytváranie polí

V Jave je pole objekt — vytvára sa a inicializuje dynamicky. V Jave neplatia mnohé obmedzenia pre polia, ktoré platia v jazyku C.

Referenciu na pole celých čísel môžeme vytvoriť takto:

```
int[] a;
```

alebo takto:

```
int a[];
```

Prvý spôsob je rozšírenejší, lebo jasnejšie uvádza typ referencie: pole hodnôt typu **int**.

Zatiaľ sme vytvorili len referenciu na pole `a`. Teraz môžeme vytvoriť napríklad pole celých čísel dĺžky 5 a priradiť ho do referencie `a`:

```
a = new int[5];
```

Typ prvkov musí zodpovedať typu poľa.

Pri vytváraní referencie na pole sa neuvádza jeho veľkosť. Veľkosť poľa musí byť daná pri vytváraní samotného poľa, ale nemusí byť známa v čase prekladu. Polia sú objekty, a objekty sa vytvárajú až v čase vykonávania programu. Veľkosť poľa tak môžeme zadať metódou:

```
class C {
   public static int f(...) {
      . . .
   }
. . .
```

[8] *Array*, anglický termín pre *pole*, v skutočnosti znamená niečo usporiadané. *Pole* má hlboko zakorenený význam prázdneho priestoru. Preto by vhodnejší termín pre pole bol *reťazec*. Tento termín sa bežne používa na označenie špeciálneho druhu reťazcov — reťazcov znakov.

```
}
. . .
   a = new int[C.f(...)];
```

Veľkosť poľa sa uchováva v atribúte `length`. Pre uvedené pole `a` je to `a.length`.

K prvkom poľa sa pristupuje prostredníctvom indexu:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Index prvého prvku je 0. Nedá sa siahnuť mimo pamäte rezervovanej pre pole — vznikne chyba pri vykonávaní programu, tzv. výnimka:

```
a[a.length]; // chyba pri vykonávaní programu
```

Inicializácia polí

Inicializácia prvkov poľa je implicitná. V našom príklade všetky prvky budú na začiatku mať hodnotu nula. Polia môžeme pri vytváraní inicializovať explicitne:

```
int[] c1 = new int[] { 1, 2, 3 };
```

Pritom sa neuvádza rozmer, lebo je daný počtom prvkov. Jestvuje aj skrátená forma:

```
int[] c2 = { 1, 2, 3 };
```

Rovnako sa postupuje aj pri inicializácii polí referencií:

```
Student[] r1 = new Student[] {
   new Student(), new Student(), };

Student[] r2 = { new Student(), new Student(), };
```

Čiarka na konci nie je chybou — prekladač ju ignoruje, a ak prvky uvádzame v samostatných riadkoch, značne zjednodušuje ich preusporadúvanie a zadávanie nových prvkov.

Polia ako parametre metód

Ako aj iné objekty, polia sa prenášajú referenciou. Dajú sa použiť na simuláciu metód s premenlivým počtom parametrov známych napríklad z jazyka C (pod názvom vararg/variadic[9] function).

Povedzme, že v našom príklade so zápisom študentov chceme poskytnúť metódu na zápis viacerých študentov naraz (napr. zo zoznamu):

```
class Zapis {
. . .
   static void zapisStudentov(int r, Student[] s) {
      for (int i = 0; i < s.length; i++) {
         s[i].zapisany = true;
         s[i].rocnik = r;
      }
   }
}
```

Metódu môžeme vyvolať aj tak, že pole študentov vytvoríme priamo na mieste parametra:

```
Zapis.zapisStudentov(1,
   new Student[] { s1, s2, s3, s4 });
```

Java 5 podporuje ozajstné metódy s premenlivým počtom parametrov. Podpora je založená na poliach: posledný parameter označený tromi bodkami sa jednoducho správa ako pole. Prepíšeme predchádzajúci príklad do novej syntaxe:

```
class Zapis {
. . .
   static void zapisStudentov(int r, Student... s) {
      for (int i = 0; i < s.length; i++) {
         s[i].zapisany = true;
         s[i].rocnik = r;
      }
   }
}
```

Pri volaní už nemusíme vytvárať pole:

[9] http://en.wikipedia.org/wiki/Variadic_function

```
Zapis.zapisStudentov(2, s1, s2, s3, s4);
```

ale môžeme:[10]

```
Zapis.zapisStudentov(2,
   new Student[] {s1, s2, s3, s4});
```

Takýto zápis možno používať všade kde aj pôvodný. Ide vlastne o tzv. syntaktický cukor (syntactic sugar):[11]

```
public static void main(String... args)
```

Viacrozmerné polia

Doteraz sme pracovali s jednorozmernými poliami. Java, samozrejme, podporuje aj viacrozmerné polia. Najčastejšie sa využívajú dvojrozmerné polia, ktorými môžeme reprezentovať matice. Pri inicializácii sa každý rozmer musí uviesť zvlášť v zátvorkách:

```
char[][] abc = {
   { 'a', 'b', 'c', },
   { 'x', 'y', 'z', },};
```

Výpis po riadkoch realizujeme klasickou **for** dvojitou slučkou:

```
for (int i = 0; i < abc.length; i++) {
   for (int j = 0; j < abc[i].length; j++)
      System.out.print(abc[i][j] + " ");

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

Podpora práce s poliami

Java API poskytuje podporu práce s poliami prostredníctvom statických metód tried `Array` a `Arrays`. Trieda `Array` poskytuje podporu pre dynamické vytváranie a prístup k poliam. Trieda `Arrays` obsahuje rady preťažených metód pre prácu s poliami:

- `equals()` — porovnávanie polí

[10] `http://www.agiledeveloper.com/articles/Java5FeaturesPartII.pdf`
[11] ktorý má „osladiť" programovanie

- `fill`() — naplnenie poľa zadanou hodnotou
- `sort`() — triedenie prvkov poľa
- `binarySearch`() — binárne vyhľadávanie v utriedenom poli

Okrem toho, trieda `System` poskytuje statickú metódu `arraycopy`(), ktorá slúži na kopírovanie polí. Všetky tieto metódy sú podrobne opísané v dokumentácii Java API.

Ako sme už spomínali, reťazce znakov sú v Jave objekty. Tieto objekty môžu byť dvoch typov. Inštancie triedy `String` predstavujú konštantné reťazce znakov, kým inštancie triedy `StringBuffer` premenlivé reťazce znakov. Java API poskytuje rozsiahlu podporu práce s reťazcami znakov: vytváranie, kopírovanie, spájanie, vyhľadávanie podreťazcov, nahrádzanie podreťazcov a pod.

4 Agregácia a dedenie

Už v prvej kapitole sme načrtli dva významné mechanizmy objektovo-orientovaného programovania: agregáciu a dedenie. V tejto kapitole ich rozoberieme podrobnejšie.

4.1 Agregácia

Agregáciu sme pomerne často používali už v predchádzajúcich kapitolách. V Jave objekt môže obsahovať iný objekt len referenciou prostredníctvom atribútov tried. Výnimkou sú inštancie primitívnych typov, ktoré však nie sú objekty v objektovo-orientovanom zmysle.

Vo všeobecnosti objekt môže obsahovať iné objekty hodnotou (containment by value) alebo referenciou (containment by reference). Pre agregáciu sa používa aj termín kompozícia. Z jazykového hľadiska obidva termíny znamenajú to isté (zloženie, zoskupenie), ale (žiaľ) rozšírené použitie je také, že agregácia má význam obsiahnutia referenciou, kým kompozícia má význam obsiahnutia hodnotou. Obsahovanie hodnotou sa pritom nevzťahuje na realizáciu obsahovania v pamäti, ale znamená, že agregát má exkluzívnu kontrolu nad agregovaným objektom, a že pri zániku agregátu zaniká aj agregovaný objekt. Aby nedochádzalo k nedorozumeniam, lepšie je bez ohľadu na označenie agregácia alebo kompozícia explicitne uviesť, či ide o obsahovanie hodnotou alebo referenciou (napríklad „kompozícia hodnotou").

Ako bolo vysvetlené v časti 2.5, agregácia predstavuje významný spôsob tvorenia hierarchie. Agregát možno vnímať ako nadobjekt vzhľadom na objekty, ktoré obsahuje.

4.2 Dedenie

Dedenie (inheritance) umožňuje využiť raz deklarovanú triedu ako základ pre ďalšie triedy odvodené od nej (derived classes). Dedenie vo vývoji softvéru je založené na princípoch zovšeobecňovania (generalizácie) a abstrakcie: odvodená trieda je špeciálnym prípadom (všeobecnejšej) pôvodnej triedy, ale typicky ju aj rozširuje, t.j.

konkretizuje (pozri časť 6.2).

Dá sa dediť štruktúra, t.j. implementácia, a správanie (behavior), t.j. rozhranie (interface). Ťažké je úplne oddeliť štruktúru od správania, ale v Jave je štruktúra zhruba daná atribútmi a implementáciou metód, kým správanie je dané *typom*, tzn. poskytnutými metódami.

V Jave sa hovorí, že trieda, ktorá dedí od inej triedy (podtrieda, subclass), túto triedu (nadtrieda, superclass) rozširuje:

```
class Nadtrieda {
}

class Podtrieda1 extends Nadtrieda {
}
```

Podtrieda získava štruktúru nadtriedy. Môže ju rozšíriť o ďalšie atribúty a metódy, ale — ako uvidíme — môže aj zmeniť jej metódy.

Predpokladajme, že vyvíjame malý grafický systém. Potrebujeme v ňom reprezentovať geometrické útvary. Vytvoríme triedu `Utvar`, ktorá bude zahŕňať spoločné charakteristiky útvarov:

```
class Utvar {
   int farba;
   void nakresli() {}
   void vypln(int f) {
      farba = f;
   }
   void zrus() {}
}
```

Neznámy útvar nevieme nakresliť ani zrušiť. Vieme mu jedine nastaviť farbu.

Ďalšou triedou, ktorú budeme potrebovať na vytvorenie jednotlivých útvarov, je trieda na reprezentáciu bodu:

```
class Bod {
   double x, y;
   public Bod(double x, double y) {
      this.x = x;
      this.y = y;
   }
}
```

Kruh je jedným z útvarov, ktoré potrebujeme. Odvodíme ho od triedy `Utvar`:

```
class Kruh extends Utvar {
   Bod c;
   double r;
   public Kruh(Bod c, double r) {
      this.c = c;
      this.r = r;
   }
   void nakresli() {
      System.out.println("Kruh (" + c.x + ", " + c.y +
         "), r = " + r);
   }
   void vypln(int f) {
      System.out.println("Kruh (" + c.x + ", " + c.y +
         "), r = " + r + " bol vyplnený farbou " + f);
   }
   void zrus() {
      System.out.println("Kruh (" + c.x + ", " + c.y +
         "), r = " + r + " bol zrušený");
   }
}
```

Metódy na prácu s kruhom v skutočnosti len vypisujú správy, ale to stačí pre účely tohto príkladu.

Použitie týchto tried by mohlo vyzerať nasledovne:

```
class GUtvary {
   public static void main(String[] args) {
      Bod b = new Bod(25.0, 50.0);
      Kruh k = new Kruh(b, 10.0);

      k.nakresli(); // Kruh (25.0, 50.0), r = 10.0
   }
}
```

4.3 Prekonávanie metód

Deklarácia nestatickej metódy rovnakej signatúry v podtriede *prekonáva* (overrides) pôvodnú metódu nadtriedy.[1] Pre objekty podtriedy už neplatí prekonaná metóda.

Prekonanú metódu je možné zavolať len zo samotnej prekonávajúcej metódy pomocou kľúčového slova **super**:

```
class Kruh extends Utvar {
   . . .
   void vypln(int f) {
      super.vypln(f);
      . . .
   }
   . . .
}
```

Takto sa dá zavolať len bezprostredne prekonaná metóda (nie napr. metóda nadnadtypu).

Pri dedení platí aj preťaženie metód. Niekedy nemusí byť úplne jasné, o ktorý jav ide. Platia však tieto zásady:

- ak sa metódy nelíšia v parametroch, dôjde k prekonaniu
- ak sa metódy líšia v parametroch, dôjde k preťaženiu

Vhodným príkladom preťaženia pri dedení môže byť metóda na kreslenie. Metóda na kreslenie bez parametra nakreslí objekt jeho farbou danou atribútom `farba`:

```
class Utvar {
   int farba;
   void nakresli() {
      . . .
   }
   . . .
}
```

Povedzme, že pre kruh potrebujeme aj metódu na kreslenie zadanou farbou:

[1] Pre prekonávanie (overriding) metód sa používajú tiež termíny predefinovanie, prekrývanie, prípadne potláčanie. Označenie prekrývanie je úplne nevhodné kvôli skrývaniu, čo predstavuje — ako uvidíme — úplne iný jav.

```
class Kruh extends Utvar {
   . . .
   void nakresli(int f) { // nakresli kruh farbou f
      System.out.println("Kruh (" + c.x + ", " + c.y +
         "), r = " + r + ", farba = " + f);
   }
   . . .
}
```

Pri dedení atribút podtriedy skryje rovnomenný atribút nadtriedy. K pôvodnému atribútu sa dá pristúpiť pomocou referencie na nadobjekt `super`:

```
class A {
   int p = 0;
}

class B extends A {
   int p = 1;
   void f() {
      System.out.println(p + " " + super.p); // 1 0
   }
}
```

Takto sa dá pristúpiť len k bezprostredne skrytému atribútu (ako pri prekonávaní metód).

Podobne ako pri atribútoch, pri dedení dochádza aj k skrývaniu statických metód. K skrytej metóde je možné pristúpiť pomocou referencie **super**. Keďže je táto referencia dostupná len v nestatickom kontexte, musí to byť z nestatickej metódy daného objektu, ako je napríklad metóda `m`() v nasledujúcom príklade:

```
class A {
   static void f() {
      System.out.println("A");
   }
}

class B extends A {
   static void f() {
      System.out.println("B");
   }
   void m() {
      super.f();
   }
}
```

Vyvolaním metódy `m()` vyvoláme metódu `A.f()`:

```
(new B()).m(); // A
```

4.4 Riadenie prístupu v dedení

Čo najviac z nadtriedy treba skryť nastavením prístupu **private**. Ako sme videli v časti 3.9, atribúty a metódy triedy s prístupom **private** sú prístupné len v danej triede, čo znamená, že nie sú prístupné ani v podtriedach.

Často však treba umožniť takýto prístup. V Jave sa to dá pomocou modifikátora prístupu **protected**, ktorého použitím sa prvok stáva dostupným každej podtriede (v celej hierarchii dedenia) a triedam, ktoré sú v tom istom balíku ako daná trieda. Hierarchia dedenia môže zahŕňať aj triedy z iných balíkov. Prvky s prístupom **protected** sú prístupné aj týmto triedam.

Povedzme, že v našom príklade s grafickými útvarmi chceme, aby reprezentácia farby zostala skrytá. Zároveň chceme umožniť nastavenie a zisťovanie farby, ale len v rámci balíka grafických útvarov.

```
class Utvar {
   private int farba; // v triede Kruh už nedostupné
   protected void nastavFarbu(int f) {
      . . .
   }
   protected int ziskajFarbu() {
      return farba;
   }
}
```

4.5 Inicializácia pri dedení

Pri dedení treba dbať na správnu inicializáciu nadtriedy, lebo sa implicitne volajú len konštruktory bez parametrov, ktoré nemusia byť v každej triede. Ostatné konštruktory musia byť vyvolané pomocou kľúčového slova **super**. Volanie konštruktora musí byť prvý príkaz.

Definujme konštruktor triedy `Utvar`, ktorého parametrom je farba útvaru:

```
class Utvar {
   int farba;
   Utvar(int f) {
      . . .
   }
   . . .
}
```

Týmto implicitný konštruktor bez parametrov už nie je prístupný. V odvodenej triede `Kruh` musíme priamo vyvolať konštruktor triedy `Utvar`:

```
class Kruh extends Utvar {
   . . .
   Kruh() {
      super(0);
   }
   . . .
}
```

inak prekladač hlási chybu.

4.6 Trieda Object

Každá trieda, pre ktorú nie je definovaná klauzula `extends`, implicitne dedí od triedy `Object`. Trieda `Object` je súčasťou základného balíka `java.lang`.

Trieda `Object` poskytuje užitočné metódy ako napríklad `toString`(), ktorú sme využívali v doterajších príkladoch bez toho, aby sme si to uvedomovali: implicitne ju volá operátor + spájania reťazcov. Tým, že je definovaná v triede `Object`, metóda `toString`() je definovaná pre každú triedu. Jej prekonaním možno poskytnúť výpis adekvátny danej triede.

4.7 Kľúčové slovo final

Kľúčové slovo **final** zabraňuje zmene prvku, na ktorý sa vzťahuje. Toto kľúčové slovo umožňuje definovať finálne atribúty tried, finálne parametre metód, finálne metódy a finálne triedy.

Finálne atribúty tried môžu reprezentovať konštanty inicializované pri preklade:

```
public final float PI = 3.14f;
```

Takéto konštanty však môžu byť inicializované aj až pri vykonávaní programu:

```
private final int K = f();
```

Ako aj iné atribúty, konštanty môžu byť statické. Musia byť explicitne inicializované pri deklarácii alebo v konštruktore. Pri objektoch si treba uvedomiť, že finálna je len referencia, čo znamená, že sa nedá nastaviť na iný objekt, ale nič nebráni zmene samotného objektu.

Metóda môže používať parametre ako hocijaké iné lokálne premenné:

```
void m(int i, Utvar u) {
   i = 0;
   u = new Utvar();
}
```

Zmena však platí len v metóde. Kľúčovým slovom **final** sa dá zabrániť zmene parametrov. Znovu sa tým nezabráni zmene samotného objektu.

```
void m(final int i, final Utvar u) {
   u.farba = 1;
}
```

Pri metódach kľúčové slovo **final** zabraňuje prekonávaniu. Pokus o prekonanie finálnej metódy spôsobí chybu pri preklade.

Metódy s prístupom **private** sú implicitne finálne. Pokus prekonania metódy s prístupom **private** zdanlivo prejde, ale vlastne nejde o prekonanie, lebo pre podtriedu prvky s prístupom **private** nie sú viditeľné.

Celá trieda môže byť deklarovaná ako finálna. Od takej triedy sa nedá dediť. Všetky jej metódy preto budú implicitne finálne (bez dedenia niet prekonávania). Atribúty finálnej triedy sa správajú rovnako ako pri triede, ktorá nie je finálna.

Použitie kľúčového slova **final** môže viesť k optimalizácii programu, ale netreba ho používať na tieto účely[2]

[2]„Premature optimization is the root of all evil.“ — C. A. R. Hoare

4.8 Dedenie a typy

Ako sme už hovorili, trieda definuje typ. Typ podtriedy je podtypom nadtriedy, čo znamená, že všade kde je použitý nadtyp, možno použiť podtyp. Napríklad metóda `zarad()` deklarovaná pre typ `Utvar`:

```
void zarad(Utvar u) {
    . . .
}
```

bude akceptovať objekt podtriedy

```
Kruh k = new Kruh(5, new Bod(10, 10));
zarad(k);
```

Tento jav sa označuje ako *upcasting*. Ide o implicitnú zmenu typu objektu z typu podtriedy na typ nadtriedy.[3]

Tento jav je súčasťou mechanizmu označovaného polymorfizmus, ktorému je venovaná nasledujúca kapitola.

[3]Preto *up* (hore), lebo je zmena hore, smerom k nadtriede, ktorá je v hierarchii tried postavená vyššie.

5 Polymorfizmus

Ako ste sa dozvedeli v predchádzajúcich kapitolách, trieda v Jave definuje typ. Triedy môžu dediť od iných tried, pričom sa trieda, ktorá dedí, označuje ako podtrieda, a trieda, od ktorej sa dedí, ako nadtrieda. Kľúčové pre mechanizmus *polymorfizmu*, ktorý je esenciálny pre objektovo-orientované programovanie, je, že typ podtriedy je podtypom nadtriedy, vďaka čomu dochádza k tzv. upcastingu, implicitnej zmene typu objektu z typu podtriedy na typ nadtriedy (pozri časť 4.8).

5.1 Polymorfizmus a prekonávanie

Polymorfizmus znamená viactvárnosť: objekt sa môže správať akoby bol objektom hociktorého zo svojich nadtypov. Každý objekt však „pozná" svoj skutočný typ.

Výber tela metódy pri volaniach sa v Jave uskutočňuje až v čase vykonávania programu. Ako sme videli v časti 4.7, jednou výnimkou sú finálne metódy, pri ktorých evidentne nemôže dôjsť k prekonávaniu (a **private** metódy sú implicitne finálne).

Ďalšou výnimkou sú statické metódy, ktorých telo sa vyberá vždy na základe typu referencie, ktorý je známy v čase prekladu, čo je opísané podrobnejšie v časti 5.3.

Hovorí sa, že telo metódy sa *viaže* k volaniu. V jazyku C sa všetky viazania uskutočňujú najneskôr v čase prekladu. V jazyku C++ je možné vybrať, či k viazaniu dôjde až v čase vykonávania, čo sa označuje ako *neskoré viazanie* (late binding). V Jave sú všetky viazania neskoré (viac o viazani v časti 6.6).

Pokračujme v príklade grafických útvarov, ktorý sme skúmali v predchádzajúcej kapitole. Každý z útvarov bude potrebné nejako vykresliť. Preto do základnej triedy hierarchie útvarov pridáme metódu na kreslenie:

```
public class Utvar {
    . . .
    public void nakresli() { }
}
```

Metóda je však prázdna, lebo nevieme nakresliť útvar vo všeobecnosti.

Každý konkrétny útvar bude musieť implementovať svoj spôsob vykreslenia. Napríklad, pre kruh by to vyzeralo nasledovne:[1]

```
public class Kruh extends Utvar {
   . . .
   public void nakresli() {
      System.out.print("Kruh ");
   }
}
```

a pre trojuholník takto:

```
public class Trojuholnik extends Utvar {
   . . .
   public void nakresli() {
      System.out.print("Trojuholník ");
   }
}
```

Predpokladajme, že v našom grafickom systéme útvary, ktoré výkres obsahuje, udržiavame v poli:

```
Utvar[] z = new Utvar[] { new Kruh(a, 3.0),
                          new Trojuholnik(a, b, c),
                          new Kruh(b, 1.0) };
```

Vďaka polymorfizmu, vykreslenie všetkých útvarov môžeme realizovať veľmi jednoducho:

```
for (int i = 0; i < z.length; i++)
   z[i].nakresli(); // Kruh Trojuholník Kruh
```

pričom sa pre každý útvar vyvolá jeho metóda na kreslenie.

Veľmi jednoducho môžeme pridať ďalšie druhy útvarov, vrátane takých, ktoré sú odvodené od jestvujúcich:

[1] Metódy na prácu s kruhom v skutočnosti len vypisujú správy, ale to stačí na účely tohto príkladu.

```
public class PTrojuholnik extends Trojuholnik {
   . . .
   public void nakresli() {
      System.out.print("PTrojuholník ");
   }
}
```

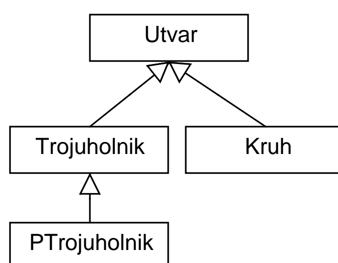
Polymorfizmus bude fungovať aj po pridaní ďalších tried do hierarchie:

```
Utvar[] z = new Utvar[] { new Kruh(a, 3.0),
                          new Trojuholnik(a, b, c),
                          new PTrojuholnik(a, b, c) };
```

Slučka pre vykreslenie útvarov zostáva nezmenená:

```
for (int i = 0; i < z.length; i++)
   z[i].nakresli(); // Kruh Trojuholník PTrojuholník
```

Grafický môžeme túto hierarchiu reprezentovať ako na obr. 5.1. Použitá je notácia UML, o ktorom viac budeme hovoriť v kapitole 14.



Obrázok 5.1: Hierarchia grafických útvarov.

5.2 Abstraktné triedy a metódy

Abstrakcia je proces uberania detailov. „Abstrahovať od niečoho“ znamená „vynechať niečo“. Opačný proces od abstrakcie sa označuje ako konkretizácia.

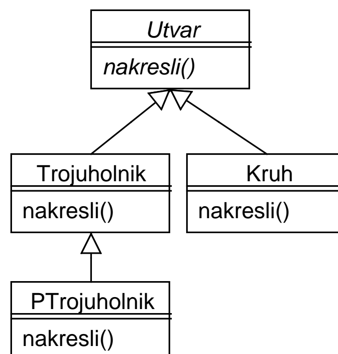
Pre niektoré triedy nemá význam, aby mali inštancie. Takéto triedy sa označujú ako abstraktné a Java túto konštrukciu priamo podporuje.

Abstraktné triedy sú určené len na dedenie. Výlučne v abstraktných triedach sa môžu vyskytovať abstraktné metódy — metódy, ktoré nemajú telo, a ktoré sú určené len na prekonávanie. Logicky, abstraktné metódy nie je možné zavolať.

Vráťme sa znovu nášmu príkladu s grafickými útvarmi. Ak uvažujeme o triede `Utvar`, zistíme, že nikdy nebudeme potrebovať jej inštancie. Aby sa zabezpečilo rozpoznanie volania `nakresli()` v podtriedach, trieda `Utvar` musí obsahovať metódu `nakresli()`, aj keď s prázdnym telom, lebo nevieme nakresliť útvar vo všeobecnosti. Nie je však žiaduce, aby túto metódu niekto zavolal. Toto zabezpečíme deklarovaním metódy ako abstraktnej:

```
abstract class Utvar {
   . . .
   abstract void nakresli();
}
```

Na obr. 5.2 je znázornená hierarchia grafických útvarov s abstraktnou triedou `Utvar`. Abstraktné prvky sa v jazyku UML označujú kurzívou.



Obrázok 5.2: Hierarchia grafických útvarov s abstraktnou triedou `Utvar`.

5.3 Polymorfizmus a statické metódy

Pri statických metódach nedochádza k prekonávaniu — len k skrývaniu. Výber statickej metódy sa uskutočňuje už pri preklade na základe typu referencie. Ak máme napríklad dve triedy vo vzťahu dedenia, ako v tomto príklade:

```
class A {
   static void sf() {
```

```
        System.out.println("A");
    }
}

class B extends A {
    static void sf() {
        System.out.println("B");
    }
}
```

pri volaní metódy sf() nedochádza k prekonávaniu:

```
A a = new B();
a.sf(); // A
```

Môže sa stať, že bol vykonaný upcasting a potrebujeme zavolať metódu podtriedy. V takom prípade musíme urobiť tzv. *downcasting*. Využíva sa pritom RTTI, t.j. identifikácia typov v čase vykonávania, o ktorej budeme viac hovoriť v kapitole 9.

Predpokladajme, že máme triedu `A`:

```
class A {
    public void f() {}
}
```

Trieda `B` rozširuje triedu `A` o ďalšiu metódu `g()` a prekonáva jej jestvujúcu metódu `f()`:

```
class B extends A {
    public void f() {}
    public void g() {}
}
```

Vytvoríme jednu inštanciu triedy `A`:

```
A o1 = new A();
```

a jednu inštanciu triedy `B`, ktorú vďaka upcastingu môžeme uložiť do referencie typu `A`:

```
A o2 = new B(); // upcasting
```

Týmto sme sa však zbavili možnosti priamo vyvolať metódu g():

```
o2.g() // chyba pri preklade
```

Metódu g() vyvoláme pomocou downcastingu:

```
((B)o2).g() // downcasting
```

Na rozdiel od upcastingu, downcasting nie vždy končí úspešne a chyba sa prejaví až v čase vykonávania:

```
((B)o1).g() // chyba pri vykonávaní
```

5.4 Rozhrania

Rozhranie možno chápať ako špeciálny druh triedy, ktorá je abstraktná a ktorej všetky metódy sú tiež abstraktné. Rozhranie umožňuje definovať správanie bez implementácie:

```
interface Kresleny {
   void nakresli();
   void nakresli(int f);
}
```

Rozhrania vlastne predpisujú správanie, a triedy sa môžu zaviazať, že ho *implementujú* prostredníctvom dedenia od rozhrania. Preto sa pre tento druh dedenia v Jave používa kľúčové slovo **implements**:

```
class Kruh implements Kresleny {
   . . .
}
```

Trieda môže dediť súčasne aj od ďalšej triedy, aj od rozhrania:

```
class Kruh extends Utvar implements Kresleny {
   . . .
   public void nakresli() {
```

```
        System.out.println("Kruh (" + c.x + ", " + c.y +
            ") r = " + r);
    }
    public void nakresli(int f) {
        System.out.println("Kruh (" + c.x + ", " + c.y +
            ") r = " + r + "farba " + f);
    }
    . . .
}
```

Tým, že trieda implementuje rozhranie, dané rozhranie je jej nadtypom a môžeme do referencie typu rozhrania priradiť objekt typu triedy, ktorá ho implementuje:

```
Kresleny k = new Kruh();
```

Všetky triedy odvodené od triedy, ktorá implementuje nejaké rozhranie, tiež dedia toto rozhranie. Ak napríklad chceme dosiahnuť, aby každý útvar musel implementovať metódy na kreslenie, stačí to uviesť pri triede `Utvar`:

```
abstract class Utvar implements Kresleny {
    private int farba;
    . . .
}
```

Konkrétne triedy musia implementovať všetky metódy rozhrania, ale abstraktné triedy nemusia. Všetky metódy rozhrania sú abstraktné a majú prístup `public`. Prekonávajúca metóda nesmie obmedzovať prístup, a tak aj všetky implementácie týchto metód v triedach musia mať prístup **public**.

Rozhranie môže obsahovať atribúty. Všetky atribúty rozhrania sú statické a finálne.

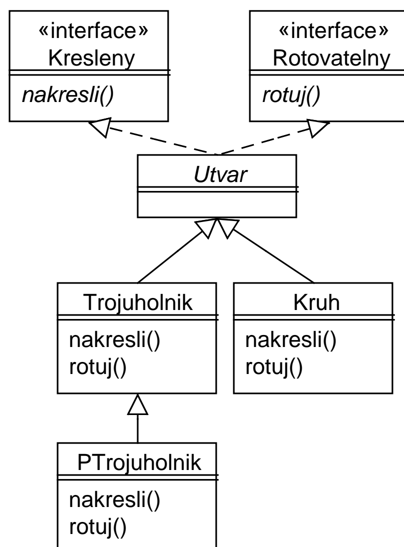
Trieda môže implementovať viac rozhraní. V našom príklade to môžu byť rozhrania na kreslenie a rotovanie.

```
interface Kresleny {
    void nakresli();
    void nakresli(int f);
}

interface Rotovatelny {
    void rotuj(double uhol);
}
```

```
abstract class Utvar implements Kresleny, Rotovatelny {
   private int farba;
   . . .
}
```

Situácia v našom príklade s grafickými útvarmi sa komplikuje — na obr. 5.3 je pre sprehľadnenie uvedený diagram v UML.



Obrázok 5.3: Hierarchia grafických útvarov s rozhraniami.

Ak trieda implementuje viac rozhraní, môžu vznikať kolízie názvov metód v prípade, že sa líšia len typom návratovej hodnoty.

Rozhranie môže dediť od iného rozhrania rovnako ako trieda od triedy — pomocou klauzuly `extends`. Rozhranie však môže dediť od viacerých rozhraní súčasne, ako ukazuje nasledujúci príklad:

```
interface M {
   void m1();
   void m2();
}

interface N {
   void n();
}

interface I extends M, N {
```

```
    void i();
    void n(int i); // preťažená metóda
}
```

Rozhranie `I` rozširuje rozhrania `M` a `N` o ďalšiu metódu `i()` a zároveň preťažuje metódu `n()` rozhrania `N`.

6 Aplikácia objektovo-orientovaných mechanizmov

V predchádzajúcich kapitolách sme sa oboznámili s elementárnymi objektovo-orientovanými mechanizmami. V doterajšom výklade tieto mechanizmy boli prezentované v programovacom jazyku Java. V tejto kapitole sa na ne pozrieme všeobecnejšie.

Hlavné objektovo-orientované mechanizmy sú [Boo94]:

- abstrakcia
- zapuzdrenie (encapsulation)
- modulárnosť
- hierarchia

Vedľajšie objektovo-orientované mechanizmy sú:

- typovosť
- súbežnosť
- perzistencia

Napriek rôznorodosti objektovo-orientovaných prístupov (časť 6.1) tieto mechanizmy majú univerzálny charakter. V tejto kapitole sa bližšie pozrieme na abstrakciu (časť 6.1), zapuzdrenie (časť 6.3), modulárnosť (časť 6.4), hierarchiu (časť 6.5) a typovosť (časť 6.6).

6.1 Rôznorodosť objektovo-orientovaných prístupov

Ako už bolo spomínané v kapitole 1, korene objektovo-orientovaného programovania sú v programovacom jazyku Simula 67, ktorý možno označiť za prvý objektovo-orientovaný jazyk. Autori tohto jazyka, Ole-Johan Dahl a Kristen Nygaard, prví použili pojem objekt v zmysle objektovo-orientovaného programovania.[1]

Základ jazyka Simula 67 bol procedurálny jazyk Algol 60, čím sa procedurálne vlastnosti preniesli aj na Simulu 67. Na procedurálnom základe vznikli ďalšie objektovo-orientované programovacie jazyky, medzi ktoré patria aj Java a C++.

Programovací jazyk Smalltalk[2] predstavuje pokus o odpútanie sa od procedurálnych záležitostí a dôsledné dodržanie princípu, že všetko má byť objekt. V Smalltalku sa všetko odohráva ako spolupráca objektov prostredníctvom tzv. posielania správ, t.j. volania metód.

Ďalšiu vetvu objektovo-orientovaných jazykov predstavujú jazyky postavené na funkcionálnom základe. Prominentným reprezentantom týchto jazykov je CLOS (Common Lisp Object System), ktorý vychádza z programovacieho jazyka Lisp.

Čo je základným prvkom objektovo-orientovaného programovania? Na prvý pohľad sa zdá, že táto otázka je triviálna, ale pozrime sa na programy v Jave a zamyslime sa, z čoho pozostávajú. Všimneme si predovšetkým triedy ako kľúčové prvky [Mey97]. Objekty, ktoré vlastne vzniknú až v čase vykonávania programu, sme modelovali zodpovedajúcimi triedami. Programovanie založené na triedach je príznačné pre mnohé objektovo-orientované programovacie jazyky počnúc Simulou 67, cez Smalltalk a C++, až po Javu.

Objekt je inštanciou triedy. Objekt má [Boo94]:

- stav — zahŕňa všetky vlastnosti objektu a ich hodnoty
- správanie — ako objekt koná a reaguje v zmysle zmeny stavu a volania metód
- identitu — každý objekt je jednoznačne identifikovateľný

Sú však aj také programovacie jazyky, v ktorých je koncept triedy potlačený, aby sa zdôraznila dynamická zložka. Takýto prístup sa označuje ako prototypovo-orientované programovanie. Dedenie sa v prototypovo-orientovaných jazykoch realizuje priamo medzi objektmi: objekt sa odvádza od iného objektu — svojho *prototypu*. Príkladmi takýchto jazykov sú Self a JavaScript.

[1] `http://staff.um.edu.mt/jskl1/talk.html#History%20I`
[2] `http://www.smalltalk.org/`

6.2 Abstrakcia

Abstrakcia predstavuje spôsob vysporiadania sa so zložitosťou. Známym príkladom z každodenného života sú mapy. Mapy predstavujú abstrakcie územia reálneho sveta. Môžu spĺňať rôzne účely a podľa toho sú na nich určité detaily zdôraznené alebo vynechané. Napríklad, pre cestné mapy sú kľúčovým prvkom cesty. Svoj význam má aj ďalšia abstrakcia cestnej mapy, na ktorej sú schematicky zobrazené len cestné prepojenia bez dodržania ich tvaru. Takéto mapy výrazne pomáhajú v orientácii v mestskej doprave.

Treba zdôrazniť, že význam pojmu abstrakcia je uberanie detailov. Samotné slovo pochádza od latinského *abstrahere*, čo znamená *vytiahnuť*.[3] Opakom abstrakcie je konkretizácia.

Abstrakciu si netreba mýliť so zovšeobecňovaním, (generalizáciou) [Náv96]. Pri zovšeobecňovaní nedochádza k uberaniu detailov. Opakom zovšeobecňovania je špecializácia. Všeobecný matematický vzorec, napríklad, obsahuje v sebe všetky alternatívy. Takýto vzorec možno upraviť tak, aby platil len pre určité špeciálne prípady, čím sa uľahčí jeho aplikácia.

Abstrakcia označuje aj proces, aj abstraktné koncepty, ktoré jeho aplikáciou vznikajú. Preto sa v objektovo-orientovanom programovaní triedy často označujú ako abstrakcie. Tieto triedy môžu skutočne priamo predstavovať abstrakcie objektov reálneho sveta, ale väčšinou to tak nie je. Ako uvidíme ďalej v tejto kapitole, objektovo-orientovaný prístup zvádza k takémuto mylnému očakávaniu.

Pre objektovo-orientované programovanie abstrakcia znamená, že sa sústreďujeme na pohľad na objekt zvonka. Súvisí to s potrebou odčlenenia správania objektu od jeho implementácie. Tvorba abstrakcií predstavuje inkrementálny (po častiach) a iteratívny (opakovane vykonávaný) proces. Nie je možné dosiahnuť kvalitné abstrakcie hneď z prvotnej predstavy, lebo veľký vplyv na ich formovanie majú vzťahy medzi nimi.

Pri posudzovaní kvality abstrakcie sledujeme tieto kritériá [Boo94]:

- zviazanosť (*coupling*) — čo najvoľnejšie triedy
- súdržnosť (*cohesion*) — v triede majú byť veci, ktoré vzájomne súvisia
- dostatočnosť — trieda musí poskytovať všetky potrebné operácie
- úplnosť (rozhrania) — rozhranie triedy má poskytovať všetky zmysluplné operácie, a
- primitívnosť (operácií) — poskytnuté operácie majú byť primitívne

[3] Latinsky *trahere* znamená *ťahať* (z čoho je aj slovo *traktor*).

6.3 Zapuzdrenie

Zapuzdrenie (encapsulation) predstavuje skrývanie informácií. Vnútorné detaily objektov nie je vhodné sprístupniť, lebo to vedie k závislosti klientskeho kódu od týchto detailov, čo znemožňuje ich úpravu bez úpravy klientskeho kódu.

Štruktúru triedy, t.j. jej atribúty a jej metódy, ktoré netvoria rozhranie, je vhodné skryť. V objektovo-orientovaných jazykoch sa zapuzdrenie väčšinou dá riadiť. V časti 3.8 sme sa oboznámili s modifikátormi prístupu v Jave.

Ako príklad k čomu môže viesť exponovanie vnútornej reprezentácie triedy uvažujme o jednoduchej evidencii študentov. Údaje o študentovi evidujeme prostredníctvom nasledujúcej triedy:

```
public class Student {
   public String meno;
   public String priezvisko;
   . . .
}
```

Klientsky kód využíva túto triedu tak, že siaha priamo na vnútornú reprezentáciu mena a priezviska:

```
Student s = new Student();
s.meno = "Ján";
s.priezvisko = "Petrov";
```

Čo ak sa rozhodneme zmeniť túto reprezentáciu? Napríklad, mohli by sme meno a priezvisko zlúčiť do jedného reťazca:

```
public class Student {
   public String meno; // meno a priezvisko
}
```

Klientsky kód sa touto zmenou stane nepoužiteľným.

Lepšie riešenie je skryť atribúty, v ktorých sa uchováva meno študenta, a sprístupniť ich prostredníctvom metód:

```
public class Student {
   private String meno;
   private String priezvisko;
```

```
    public void setMeno(String meno) {
        . . .
    }
    public void setPriezvisko(String priezvisko) {
        . . .
    }
    public String getMeno() {
        . . .
    }
    public String getPriezvisko() {
        . . .
    }
}
```

Klientsky kód potom bude nezávislý od vnútornej reprezentácie:

```
Student s = new Student();
s.setMeno("Ján");
s.setPriezvisko("Petrov");
```

Skrývanie atribútov tried je dôsledkom všeobecnejšieho *princípu otvorenosti a uzavretosti* (open-closed principle) [Mar96b]:

> Softvérové entity (triedy, moduly, funkcie atď.) majú byť otvorené pre rozšírenie, ale uzavreté pre zmeny.

Tento princíp nie je obmedzený len na objektovo-orientované programovanie, ale objektovo-orientované programovanie ho prostredníctvom polymorfizmu umožňuje lepšie dodržať. Pozrime sa bližšie na význam tohto princípu prostredníctvom príkladu malého grafického systému známeho z predchádzajúcich kapitol. Povedzme, že je v ňom takáto metóda na vykreslenie zadaných grafických útvarov.

```
void nakresliUtvary(Kresleny... k) {
    for (int i = 0; i < k.length; i++)
        k[i].nakresli();
}
```

V metóde sa využíva fakt, že grafické útvary, ktoré sa dajú vykresliť, implementujú rozhranie `Kresleny`.

Tento kód je otvorený pre rozšírenie — možno pridávať ďalšie útvary. Tieto útvary budú implementovať rozhranie `Kresleny`. Pre pridanie nových útvarov však nie sú potrebné zmeny v kóde na kreslenie, takže kód je uzavretý pre zmeny.

6.4 Modulárnosť

Modul predstavuje ucelenú časť kódu. Aj keď jeho význam je podstatne širší, pojem modulu sa zvyčajne spája so súbormi. V niektorých programovacích jazykoch, ako napríklad C a C++, moduly naozaj priamo korešpondujú so súbormi. V Jave však ako moduly možno vnímať skôr balíky, a tie sú nezávislé od súborov.

Na druhej strane, v objektovo-orientovanom programovaní — a tým aj v Jave — primárnou organizačnou jednotkou sú triedy, takže ich môžeme tiež pokladať za moduly. V tomto zmysle modularizácia úzko súvisí s abstrakciou a zapuzdrením. Závislosť zapuzdrenia od modularizácie je v Jave daná modelom riadenia prístupu, v ktorom významnú rolu hrá zaradenie triedy do balíka (pozri časť 3.8).

Pre moduly je žiaduce, aby boli voľne zviazané a zároveň súdržné. Voľná zviazanosť (loose coupling) znamená minimalizáciu vzájomných väzieb medzi modulmi. Samozrejme, bez väzieb by nebolo programu, ale vhodné vymedzenie nevyhnutného rozhrania triedy podstatne zníži potrebu zmien pri spomínanej zmene vnútornej reprezentácie.

Podobne, všetok kód týkajúci sa jednej záležitosti by mal byť sústredený na jednom mieste, čím sa dosiahne vysoká súdržnosť (high cohesion). V objektovo-orientovanom programovaní nie je vždy možné dosiahnuť súdržnosť na uspokojivej úrovni, čo je zvlášť príznačné pre tzv. pretínajúce záležitosti. O tomto bude reč v kapitole 16.

6.5 Hierarchia

Hierarchia je dôležitým mechanizmom objektovo-orientovaného programovania, ktorý úzko súvisí s abstrakciou. Pre ľudské vnímanie sveta je prirodzená hierarchizácia pojmov. V objektovo-orientovanom programovaní sa tento aspekt prejavuje priamo prostredníctvom dedenia.

Ako už bolo vysvetlené v kapitole 4, dedenie predstavuje predovšetkým vzťah zovšeobecňovania (generalizácie): nadtrieda predstavuje všeobecnejší koncept ako jej podtriedy, ktoré sa označujú za jej špecializácie. Treba si však uvedomiť, že podtriedy môžu rozširovať svoju nadtriedu o ďalšie atribúty a metódy. V takom prípade nejde o špecializáciu, ale o konkretizáciu. To znamená, že nadtrieda nie je len všeobecnejšia ako jej podtriedy, ale zároveň je zvyčajne aj abstraktnejšia.

Java umožňuje len jednoduché dedenie štruktúry. Prostredníctvom rozhraní umožňuje viacnásobné dedenie (multiple inheritance) správania. V niektorých objektovo-orientovaných programovacích jazykoch je možné aj viacnásobné dedenie štruktúry. Príkladom je jazyk C++. Aj keď viacnásobné dedenie štruktúry má svoje využitie pre tzv. *mixin* triedy, ktoré nie sú určené na samostatné použitie, ale na rozšírenie

funkcionality iných tried, prináša so sebou problémy v podobe kolízie názvov a opakovaného dedenia. Preto namiesto neho treba zvážiť pomerne priamočiaru náhradu jednoduchým dedením a agregáciou.

Agregácia predstavuje ďalší spôsob hierarchizácie. Celok prirodzene vnímame ako nadradený jeho častiam: je vyššie postavený v hierarchii. Pozícia v hierarchii určuje úroveň abstrakcie. Agregát je teda na vyššej úrovni abstrakcie než časti, ktoré agreguje. Podobne, aj nadtrieda je na vyššej úrovni abstrakcie než jej podtriedy. Táto podobnosť v tvorení hierarchií často zvádza k mylnému použitiu dedenia.

Liskovej princíp substitúcie

Najdôležitejšie kritérium pre použitie dedenia je existencia vzťahu typ-podtyp [Cop92]. Aby použitie dedenia bolo opodstatnené, nestačí teda štrukturálna podobnosť, ale predovšetkým sledujeme, či objekty uvažovanej nadtriedy bude možné nahradiť objektmi jej podtried.

Túto vlastnosť presnejšie formuluje *Liskovej princíp substitúcie* (Liskov substitution principle) [Lis87]:

> Ak pre každý objekt o_1 typu S jestvuje objekt o_2 typu T taký, že pre všetky programy P definované v zmysle T správanie P je nezmenené, keď sa o_1 nahradí o_2, potom je S podtypom T

Vhodnými prípadmi na použitie dedenia sa môžu zdať napríklad hierarchia typov geometrických útvarov (aj kruh, aj trojuholník sú geometrickými útvarmi) alebo hierarchia zamestnancov (aj šéf, aj sekretárka sú zamestnanci). Veci sa však značne komplikujú pri viacerých úrovniach hierarchie. Bezpodmienečne pri tvorbe hierarchie dedenia treba dbať o pohľad klientskeho kódu a nespoliehať sa na vzťahy všeobecné-špeciálne v modelovanej doméne.

Kedy nepoužiť dedenie

Niekedy nie je ani možné nájsť vzťah všeobecné-špeciálne medzi typmi. Vtedy dedenie rozhodne netreba použiť. Napríklad mohli by sme na uchovávanie prvkov používať zoznamy a množiny. Pri niektorých operáciách, ako sú napríklad výpis všetkých prvkov alebo pridanie prvku, môže naozaj byť irelevantné, či pracujeme so zoznamom alebo množinou. Množina neobmedzuje poradie prvkov a môže sa zdať, že je všeobecnejším konceptom než zoznam, v ktorom poradie prvkov hrá dôležitú úlohu.

Aj keby sme uvažovali o množine s možnosťou viacnásobného výskytu toho istého prvku — tzv. vrece (bag) alebo multimnožina — narazili by sme na problém množinových operácií (prienik, zjednotenie a pod.), ktoré by sa stali súčasťou zoznamu. V tomto prípade ide o vzťah podobnosti (*is-like-a*) a ten sa dá riešiť dedením od spoločnej triedy. Pre zoznam a množinu by sme teda mali spoločnú nadtriedu. Táto

trieda však nebude mať zodpovedajúci koncept v modelovanej doméne.

Niekedy láka nesprávne použitie dedenia na rozšírenie triedy na účely jej zovšeobecnenia. Príkladom môže byť odvodenie elipsy od kruhu s pridaním ďalšieho ohniska a polomeru [Mar96a]. Znamenalo by to, že všade kde sa dá použiť kruh, možno podsunúť elipsu, čo sme pravdepodobne nechceli. Toto neznamená, že odvodené triedy nesmú rozširovať základnú triedu — dôležitý je význam vzhľadom na princíp substitúcie.

Očakávaný vzťah je teda, že kruh bude podtriedou elipsy. Toto zodpovedá matematickému chápaniu týchto útvarov. Ale je to naozaj vhodný prípad pre aplikáciu dedenia? Zaraďme teda elipsu do hierarchie grafických útvarov:

```
public class Elipsa extends Utvar {
   private Bod f1;
   private Bod f2;
   private int a;
   private int b;

   public int getF1() {
      return f1;
   }
   public int getF2() {
      return f2;
   }
   public void setF1(Bod f1) {
      this.f1.setX(f1.getX());
      this.f1.setY(f1.getY());
   }
   public void setF2(Bod f2) {
      this.f2.setX(f2.getX());
      this.f2.setY(f2.getY());
   }
}
```

Táto trieda priamo odzrkadľuje matematické vnímanie elipsy. Elipsa má dve ohniská a dva polomery. Vnútorná reprezentácia je skrytá a pristupujeme k nej prostredníctvom metód. Pre úplnosť uveďme aj triedu `Bod`:

```
public class Bod {
   private int x;
   private int y;

   public int getX() {
      return x;
```

```
      }
      public int getY() {
         return y;
      }
      public void setX(int x) {
         this.x = x;
      }
      public void setY(int y) {
         this.y = y;
      }
   }
```

Pokúsme sa teda od elipsy odvodiť kruh:

```
   public class Kruh extends Elipsa {
      public void setF1(Bod f1) {
         this.f1.setX(f1.getX());
         this.f1.setY(f1.getY());
         this.f2.setX(f1.getX());
         this.f2.setY(f1.getY());
      }
      public void setF2(Bod f2) {
         this.f1.setX(f2.getX());
         this.f1.setY(f2.getY());
         this.f2.setX(f2.getX());
         this.f2.setY(f2.getY());
      }
      public void setA(int a) {
         this.a = this.b = a;
      }
      public void setB(int b) {
         this.b = this.a = b;
      }
   }
```

Niektoré údaje budú pre kruh zbytočné, ale to by bolo skôr otázkou optimalizácie. Z matematického hľadiska to nie je problém, lebo kruh vlastne predstavuje elipsu, ktorej ohniská sú zhodné, a polomery majú rovnakú veľkosť. Vyzerá, že stačí zabezpečiť ich súčasné nastavenie a nevznikne žiaden problém.

Už len zabezpečenie súčasného nastavenia však bude problém, lebo klientsky kód má možnosť oddeleného nastavovania ohnísk a polomerov. To znamená, že sa z kruhu môže stať elipsa, pričom typ objektu (`Kruh`) sa, samozrejme, nezmení.

Ďalší, fundamentálny problém bude jasný, ak zoberieme do úvahy takýto klientsky kód — metódu, ktorá posúva elipsu o (`b.x, b.y`):

```
public class C {
   void move(Elipsa e, Bod b) {
      e.setF1(new Bod(e.f1.getX() + b.getX(),
         e.f1.getY() + b.getY()));
      e.setF2(new Bod(e.f2.getX() + b.getX(),
         e.f2.getY() + b.getY()));
   }
}
```

Vďaka polymorfizmu môžeme všade, kde sa očakáva elipsa, použiť kruh — aj tam, kde sa s tým nepočítalo. Metóda `move`() posunie elipsu správne, ale kruh sa posunie o dvojnásobnú vzdialenosť! Problém je v tom, že sa nemyslelo na správanie odvodeného typu na mieste pôvodného, čo predstavuje porušenie Liskovej princípu substitúcie.

Liskovej princíp substitúcie súvisí s návrhom podľa zmluvy (design by contract). V návrhu podľa zmluvy operácie vystupujú ako zmluvné strany [Mey97]. Každá operácia definuje podmienky, ktoré platia pred operáciou (preconditions), a podmienky, ktoré platia po operácii (postconditions).[4] Operácia garantuje, že ak sú splnené podmienky, ktoré majú platiť pred operáciou, po jej zavolaní budú splnené podmienky, ktoré majú platiť po operácii.

Napríklad operácia `setA`() triedy `Elipsa` by mala garantovať, že sa po jej aplikácii atribút `b` nezmení. Pre elipsu podmienka bude dodržaná, ale pre kruh nie: dôjde k tzv. zoslabeniu podmienky, ktorá má platiť pred operáciou. Aby bol dodržaný Liskovej princíp substitúcie, pri podtypoch: podmienky, ktoré platia pred operáciou, sa môžu len zachovať alebo zoslabiť, kým podmienky, ktoré platia po operácii, sa môžu len zachovať alebo zosilniť.

6.6 Typovosť a polymorfizmus

Na triedy sa možno pozerať ako na implementácie tzv. abstraktných typov údajov. Abstraktné typy údajov predstavujú spôsob algebraickej špecifikácie, ktorá pozostáva zo vstupno-výstupných deklarácií operácií a axióm, ktoré pri ich uskutočňovaní platia.

Trieda teda určuje typ, ale typ nemusí byť určený len triedou. V Jave typ môže byť daný tiež aj rozhraním. Typovosť programovacieho jazyka znamená, že objekty rozdielneho typu je možné zamieňať len presne vymedzeným spôsobom daným vzťahmi

[4] Niekedy prekladané ako vstupné podmienky, resp. výstupné podmienky.

dedenia. V tomto smere je Java tzv. silne typový jazyk. Koncept typu môže byť potlačený až po úplné vynechanie.

Pojem viazania (binding) metód úzko súvisí s typovosťou jazyka. Presnejšie ide o viazanie volania metódy s kódom, ktorý predstavuje jej telo. Viazanie môže byť statické (skoré), uskutočnené počas prekladu, alebo dynamické (neskoré), uskutočnené v čase vykonávania programu. Dynamické viazanie metód v typových jazykoch je podmienkou pre polymorfizmus.

Polymorfizmus však možno chápať podstatne širšie než len ako prekonávanie metód (pozri kapitolu 5). Preťaženie metód je tiež istým druhom polymorfizmu, pri ktorom stači statické viazanie. Parametrický polymorfizmus je ďalším druhom polymorfizmu príznačným pre šablóny (templates) v jazyku C++, ale aj pre generickosť typov v Jave.

Polymorfizmus môže byť aj viacnásobný, ako napríklad pri multimetódach v jazyku CLOS. Vo viacnásobnom polymorfizme výber metódy závisí od typov viacerých objektov. Pre jazyky, ako sú Java alebo C++, vhodnú náhradu viacnásobného polymorfizmu predstavuje idióm *double dispatch*, ktorý je vysvetlený v časti 15.5.

6.7 Sumarizácia

V tejto kapitole bol uvedený prehľad niektorých dôležitých mechanizmov objektovo-orientovaného programovania a s nimi súvisiacich princípov. Niektoré z týchto princípov — ako napríklad princíp otvorenosti a uzavretosti — sú platné aj všeobecnejšie.

V programoch reálnej veľkosti nie je možné úplne dodržať všetky princípy. Miera ich uplatnenia však rozhoduje o flexibilnosti návrhu. Tieto princípy sa často predkladajú ako pravidlá, idiómy alebo (návrhové) vzory, o čom bude reč v kapitole 15.

7 Vhniezdené typy

Termín typ používame ako spoločné označenie pre triedy a rozhrania. Doteraz sme sa stretávali len s typmi na úrovni balíka. Java podporuje aj tzv. vhniezdené typy (nested types):[1] typy deklarované v inej triede alebo rozhraní, vrátane metód a inicializačných blokov príkazov.

Môžeme napríklad mať triedy a rozhrania v triedach:

```
class A {
   interface I { }
   class B { }
}
```

Hĺbka vhniezdenia nie je obmedzená:

```
interface J {
   class X {
      class Y {
      }
   }
}
```

7.1 Preklad vhniezdených typov

Prekladom vhniezdených typov vznikajú súbory s príponou **class** rovnako ako pri typoch na vrchnej úrovni. Názov súboru preloženej triedy obsahuje zreťazené názvy typov, v ktorých je tento typ vhniezdený. Napríklad pri preklade nasledujúceho kódu:

```
class A {
   class B {
```

[1]Vhniezdené typy sa ešte označujú aj ako *vnorené*. Toto označenie je už zaužívané pre tzv. *vnorené systémy* z anglického termínu *embedded systems*.

```
        class C { }
    }
}
```

vzniknú tri súbory s nasledujúcimi názvami:

```
A.class
A$B.class
A$B$C.class
```

Predpokladajme takéto dve vhniezdené rozhrania:

```
interface M {
    interface N {
        void n();
    }
    void m();
}
```

To, že daná trieda implementuje rozhranie `M`, neznamená, že implementuje rozhranie `N`. Ak to chceme dosiahnuť, musíme rozhranie `N` uviesť explicitne vrátane cesty k nemu:

```
class C implements M, M.N {
    public void n() {
        . . .
    }
    public void m() {
        . . .
    }
}
```

7.2 Vnútorné triedy

Vhniezdené triedy, ktoré nie sú implicitne statické alebo deklarované ako také, sa označujú ako vnútorné triedy (inner classes). Vnútorné triedy zahŕňajú nestatické členské triedy (triedy deklarované priamo v triede), lokálne triedy (triedy deklarované v hocijakom bloku vrátane tiel podmienených príkazov a cyklov) a anonymné triedy (nepomenované triedy). Termín vhniezdené triedy sa niekedy používa na označenie statických členských tried.

Nasledujúci príklad poukazuje na rôzne možnosti deklarácie vnútorných tried:

```
class A {
   class B { } // trieda v triede

   void f(int i) {
      class C { } // trieda v metóde
      if (i > 0) {
         class D { // trieda v podmienenom príkaze
            C c = new C();
         }
      }
   }

// D d; // mimo rozsahu!

   void m() {
      B b = new B(); // objekt vnútornej triedy
//    C c; // mimo rozsahu!
   }
}
```

Vnútorné triedy sú prístupné len v rozsahu, v ktorom sú deklarované. Preto napríklad nie je možné vytvoriť referenciu lokálnej triedy `D` mimo rozsahu podmieneného príkazy v metóde `f()`, v ktorom je deklarovaná.

Vnútorné triedy prenášajú objektovo-orientované prostriedky do nižších úrovní programu. Vnútorné triedy predstavujú veľmi flexibilný prostriedok často používaný v Java API, napríklad v grafickom rámci Swing, o ktorom budeme hovoriť v kapitole 13.

Vnútorná trieda je súčasťou triedy, v ktorej je deklarovaná. Všetky časti vonkajšej triedy sú dostupné vnútornej triede.

Inštancie vnútornej triedy sa nedajú vytvárať bez vytvorenia objektu vonkajšej triedy. Na jeden objekt vonkajšej triedy sa môže vzťahovať viac objektov tej istej vnútornej triedy, ale nemusí ani jeden. Vnútorná trieda nemôže obsahovať statické prvky.

Pri nestatických členských triedach môžeme používať modifikátory prístupu:

```
class A {
   private int i = 0;
   class B {
      void m() {
         i = 1;
      }
   }
   private class P { }
}
```

Trieda `P` s prístupom **private** nie je prístupná mimo triedy `A`:

```
class C {
   void f() {
//   A.P p; // private!
     A.B x = (new A()).new B();
     x.m();
   }
}
```

Preklad lokálnych tried

Možnosť deklarácie triedy v podmienenom príkaze môže vyvolávať dojem, že sa takéto triedy prekladajú podmienečne. Podmienené príkazy sa — ako vieme — vyhodnocujú až v čase vykonávania, takže aj trieda deklarovaná v podmienenom príkaze, akou je trieda `D` vo vyššie uvedenom príklade, sa preloží ako aj všetky ostatné triedy.[2]

Napríklad pri preklade nasledujúcej triedy:

```
class A {
  void f(int i) {
    if(i > 0) {
      class D { }
    }
  }
}
```

vzniknú súbory:

```
A.class A$1D.class
```

Objekty vnútorných tried

Objekty vnútornej triedy môžu byť použité aj mimo triedy alebo rozsahu, v ktorom vnútorná trieda bola deklarovaná, ale deklarácia vnútorných tried okrem členských tried s iným prístupom než je **private** je nedostupná. To znamená, že rozhranie objektu vnútornej triedy je zvyčajne neznáme (okrem rozhrania triedy `Object`) a preto vnútorná trieda najčastejšie implementuje rozhranie, ktoré je prístupnejšie. Nasledujúci príklad to demonštruje:

[2] Java nemá preprocesor ako jazyky C a C++ a neumožňuje podmienený preklad.

```
interface I {
  void m();
}

class A {
  I f() { // nemože vraciať typ B!
    class B implements I {
      public void m() {
        . . .
      }
    }
   return new B();
  }
}

class C {
  void f() {
    I b = (new A()).f();
    b.m();
  }
}
```

Metóda `f()` triedy `A` obsahuje deklaráciu ďalšej triedy `B`. Metóda `f()` potrebuje vrátiť objekt triedy `B`, ale nemôže, lebo jej deklarácia je mimo metódy `f()` neznáma. Preto trieda `B` implementuje rozhranie `I`, deklarované na úrovni balíka. Metóda `f()` potom vracia objekt triedy `B` prostredníctvom referencie typu `I`, ktorú je možné využiť aj v iných triedach, akou je napríklad trieda `C`.

7.3 Anonymné triedy

Priamo pri inštanciácii možno deklarovať triedu v tvare:

```
new [Typ]([parametre]) {
  [telo]
};
```

Takáto vnútorná trieda nemá názov, a preto sa označuje ako anonymná trieda (anonymous class). [`Typ`] predstavuje názov triedy alebo rozhrania, od ktorého anonymná trieda dedí (pomocou **extends**, resp. **implements**), [`parametre`] sú parametre konštruktora triedy, od ktorej anonymná trieda dedí, a [`telo`] sú atribúty a metódy anonymnej triedy.

Pri takejto inštanciácii vzniká objekt anonymnej triedy. Keďže trieda nemá meno, už nikdy nebude môcť vzniknúť ďalší.

Nasledujúci kód ukazuje dva príklady anonymných tried:

```
class A {
   . . .
}

class C {
   Object o = new Object() {
      float x;
      void f() {
         . . .
      }
   };

   void f() {
      A o = new A() {
         int a;
         void m() {
            . . .
         }
      };
   }
}
```

Jedna trieda je odvodená od triedy `Object`, a jej objekt figuruje ako atribút triedy `C`. Druhá trieda je lokálna a odvodená je od triedy `A`. Obe triedy rozširujú svoje nadtriedy o ďalšie atribúty a metódu. Tieto prvky však pre používateľa vytvorených objektov zostávajú neznáme, lebo im môže pristupovať len prostredníctvom rozhrania nadtried.

Tento problém sa rieši vytvorením rozhrania, ktoré deklaruje potrebné metódy, a ktoré anonymná trieda implementuje. Toto ilustruje nasledujúci príklad:

```
interface I {
   void m();
}

class A {
   I f() {
      return new I() {
         public void m() {
            . . .
```

```
            }
        };
    }
}
```

To isté sa, samozrejme, dá dosiahnuť pomocou pomenovanej lokálnej triedy:

```
I f() {
    class B implements I {
        public void m() {
            . . .
        }
    }

    return new B();
}
```

Ak je však zámer vytvoriť len jeden objekt danej triedy, verzia s anonymnou triedou je účelnejšia.

Lokálne a anonymné triedy môžu pristupovať len k finálnym parametrom metódy, v ktorej sa nachádzajú:

```
interface I {
    float m();
}

class A {
    void f(final float n) {
        class B implements I {
            public float m() {
                return n * n;
            }
        }
    }
}
```

7.4 Statické vhniezdené triedy

Vhniezdené triedy môžu byť statické s analogickými obmedzeniami ako pri iných statických prvkoch. Lokálne a anonymné triedy nemôžu byť statické. Triedy vhniezdené v rozhraniach sú implicitne statické.

Vhniezdené triedy sú vhodné na ladenie jednotlivých tried. Do každej triedy sa jednoducho pridá statická vhniezdená trieda s metódou `main`().

```
class T {
   int f(int i) {
      . . .
   }

   static class Test {
      public static void main(String[] args) {
         T t = new T();
         System.out.println(t.f(0));
         System.out.println(t.f(-1));
         . . .
      }
   }
}
```

Takéto triedy možno spustiť prostredníctvom ich názvu. V tomto prípade prekladom vnútornej triedy `Test` vznikne súbor s názvom `T$Test.class`. Z finálneho programu možno odstrániť **class** súbory, ktoré vzniknú prekladom týchto tried, čím vo finálnej verzii nebude žiaden kód navyše.

8 Výnimky

Programy by mali byť čo najrobustnejšie, ale pri tvorbe programu nie je možné predpovedať a ošetriť všetky výnimočné situácie. Jestvujú rôzne prístupy k vysporiadaniu sa s výnimočnými situáciami. Vo všeobecnosti ošetrenie tzv. výnimky (exception) môže znamenať prerušenie programu alebo pokus o zotavenie. Priame ošetrovanie výnimočných situácií na mieste ich predpokladaného vzniku robí základný kód neprehľadným. Často je to aj tak možné len vo vyššom kontexte. V tejto kapitole sa pozrieme na mechanizmus výnimiek v Jave.

8.1 Podpora výnimiek v Jave

V Jave pre výnimky jestvuje priama podpora na úrovni jazyka pre

- vyhadzovanie výnimiek (exception throwing)
- zachytávanie výnimiek (exception catching)
- spracovanie výnimiek (exception handling)

Samotná výnimka je v Jave objekt. Rôzne druhy výnimiek sú definované ako triedy. Hierarchia tried pre výnimky začína triedou `Exception`, ktorá je podtriedou triedy `Throwable`.

Metódy, v ktorých môže dôjsť k výnimkám, to deklarujú klauzulou **throws**. Výnimka môže vzniknúť pri chybách vo vykonávaní, ale dá sa vyhodiť aj priamo použitím príkazu **throw**.

Kód, ktorý obsahuje volania metód, v ktorých k výnimkám môže dôjsť, sa uzatvorí do bloku **try**. Výnimky sa zachytávajú v blokoch **catch**, ktoré nasledujú po bloku **try**. Kód, ktorý sa vykoná nakoniec (za každých okolností), sa uvedie v bloku **finally**.

8.2 Kontrola výnimiek

Pri kontrolovaných výnimkách (checked exceptions) prekladač kontroluje, či metóda nevyhadzuje výnimky, ktoré nedeklarovala, a či v metóde jestvuje ošetrenie výnimiek (exception handler), ktoré môžu vzniknúť volaním metód, ktoré ich deklarovali.

Z kontroly sú vynechané výnimky typu `RuntimeException`, medzi ktoré napríklad patria `NullPointerException` alebo `ArrayIndexOutOfBoundsException`). Takéto výnimky sa označujú ako nekontrolované (unchecked exceptions).

Uvažujme o delení a výnimkách, ktoré pri ňom môžu nastať. Vytvorme jednoduchú triedu `Delenie`, ktorá len vydelí dve čísla:

```
public class Delenie {
   public static void main(String[] args) {
      System.out.println(
         Integer.parseInt(args[0]) /
         Integer.parseInt(args[1]));
   }
}
```

Čísla sa zadávajú ako parametre triedy pri spustení. Metóda `parseInt`() konvertuje objekt triedy `String` na celé číslo.

Aj v takomto jednoduchom programe hrozia mnohé výnimočné situácie:

- nezadanie parametrov
- zadanie neceločíselných hodnôt ako parametrov
- delenie nulou

Použijeme mechanizmus výnimiek na vysporiadanie sa s týmito potenciálnymi problémami:

```
public class Delenie {
   public static void main(String[] args) {
      try {
      System.out.println(
         Integer.parseInt(args[0]) /
         Integer.parseInt(args[1]));
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println(
            "Nedostatočný počet parametrov.");
```

```
        } catch (NumberFormatException e) {
            System.out.println(
                "Nesprávny formát parametrov.");
        } catch (ArithmeticException e) {
            System.out.println("Chyba pri delení.");
        }
    }
}
```

Všetky tieto výnimky definuje Java API. Vyhadzujú ich jednotlivé operácie, ktoré sme použili. Pridali sme len ich zachytenie a ošetrenie. Ošetrenie v tomto prípade spočíva len vo výpise hlásenia s informáciou pre koncového používateľa, ktorý by bez ošetrených výnimiek bol vystavený znepokojujúcim správam o abnormálnom ukončení programu zo strany JVM.

Samotná syntax bloku **try−catch−finally** je nasledujúca:

```
try {
    // Blok, ktorý môže vyhadzovať výnimky
    // tried E1, E2 atď. alebo ich podtried.
    // Bloky catch sa vykonajú v zadanom poradí!
} catch(E1 e1) {
    // zachytáva triedu výnimiek E1 a jej podtriedy
} catch(E2 e2) {
    // zachytáva triedu výnimiek E2 a jej podtriedy
} finally {
    // kód, ktorý sa musí nakoniec vykonať vždy
}
```

Trieda `Exception` poskytuje rôzne užitočné metódy, z ktorých sú niektoré uvedené v nasledujúcom zozname:

- String getMessage() — detailná správa
- String toString() — krátky opis
- void printStackTrace() — výpis zásobníka vykonávania
- Throwable fillInStackTrace() — naplnenie zásobníka vykonávania informáciami o súčasnom stave (užitočné pri opätovnom vyhadzovaní výnimky)

8.3 Vlastné výnimky

Niektoré výnimky sú špecifické pre danú aplikáciu. Takéto výnimky sa dajú odvodiť od triedy `Exception` alebo jej podtriedy. Ak sa odvodia od `RuntimeException`, budú nekontrolované.

Uvažujme napríklad o vytváraní trojuholníka v našom grafickom systéme. Trojuholník je v ňom definovaný tromi bodmi v rovine. Pri zadaní troch kolineárnych bodov však trojuholník nevznikne. Toto je jedna zo situácií, ktoré predstavujú výnimku:

```
class NieJeTrojuholnik extends Exception {
}
```

Konštruktor triedy `Trojuholnik` vyhodí túto výnimku v prípade, že dostane tri kolineárne body:

```
class Trojuholnik {
   . . .
   Trojuholnik(Bod a, Bod b, Bod c)
      throws NieJeTrojuholnik {
      if (. . .) // overenie kolinearity
         throw new NieJeTrojuholnik();
   }
   . . .
}
```

K ošetreniu tejto výnimky dochádza vo vyššom kontexte, lebo konštruktor triedy `Trojuholnik` nevie prečo dostal nekorektné parametre. Ale metóda, ktorá ho volala, by to mohla vedieť:

```
class C {
   void m(Bod a, Bod b, Bod c) {
      try {
         Trojuholnik t = new Trojuholnik(a, b, c);
         . . .
      } catch (NieJeTrojuholnik e) {
         . . .
      }
   }
}
```

Ak metóda nevie ošetriť výnimku, môže ju preniesť do ďalšieho vyššieho kontextu jej opätovným vyhodením:

```
class C {
   void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
      try {
         Trojuholnik t = new Trojuholnik(a, b, c);
         . . .
      } catch (NieJeTrojuholnik e) {
         throw(e);
      }
   }
}
```

Ak metóda výnimku neošetruje, predsa musí deklarovať, že ju vyhadzuje.

```
class C {
   void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
       Trojuholnik t = new Trojuholnik(a, b, c);
       . . .
   }
}
```

Takýmto prenášaním výnimka môže skončiť v metóde `main()`, kde je posledná príležitosť na jej ošetrenie:

```
class C {
   void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
      Trojuholnik t = new Trojuholnik(a, b, c);
      . . .
   }

   public static void main(String[] args) {
      Bod[] b = { new Bod(0.0, 0.0),
         new Bod(1.0, 1.0), new Bod(2.0, 2.0) };
      try {
         new C().m(b[0], b[1], b[2]);
      } catch (NieJeTrojuholnik e) {
         . . . // tu by sme už mohli vedieť čo robiť
      }
   }
}
```

Deklarovaním, že metóda `main()` vyhadzuje výnimku, prenášame túto výnimku na výstup:

```
class C {
   void m(Bod a, Bod b, Bod c) throws NieJeTrojuholnik {
      Trojuholnik t = new Trojuholnik(a, b, c);
      . . .
   }

   public static void main(String[] args)
      throws NieJeTrojuholnik {
      Bod[] b = { new Bod(0.0, 0.0),
      new Bod(1.0, 1.0), new Bod(2.0, 2.0) };
      new C().m(b[0], b[1], b[2]);
   }
}
```

Takéto „prehltnutie" výnimky sa dá použiť vždy. Tým, že Java núti ošetrovať kontrolované výnimky, mnohí programátori používajú tento spôsob, aby sa čo najľahšie zbavili hlásení o neošetrených výnimkách. Treba však hľadať kontext, v ktorom možno výnimku skutočne ošetriť.

8.4 Výnimky pri prekonaných metódach

Rozsah výnimiek sa pri prekonávaní metód môže len zachovať alebo zúžiť, t.j.

- prekonávajúca metóda nemusí deklarovať žiadne výnimky
- prekonávajúca metóda môže deklarovať rovnaké triedy výnimiek ako prekonávaná metóda
- prekonávajúca metóda môže deklarovať podtriedy výnimiek tried výnimiek prekonávanej metódy

Pri preťažených metódach sa výnimky neuvažujú.

9 RTTI

Java umožňuje identifikovať typ objektu počas vykonávania programu. Mechanizmus, ktorý sa na to používa, sa označuje ako RTTI — identifikácia typov v čase vykonávania (Runtime Type Identification). Táto kapitola poskytuje základné informácie o mechanizme RTTI.

9.1 Literál **class**

Každý objekt so sebou nesie úplné informácie o svojom type. Každá načítaná trieda je reprezentovaná objektom triedy `Class`. Referencia objektu tejto triedy je dostupná prostredníctvom literálu **class**. Napríklad pre triedu `Trojuholnik` by to bolo:

```
Trojuholnik.class
```

9.2 Rozhodovanie na základe typu

V kapitole 5 sme sa oboznámili s polymorfizmom, ktorý odbremeňuje programátora od sledovania presného typu objektu a umožňuje tvoriť flexibilnejší kód. Niekedy je však nevyhnutné robiť explicitné rozhodnutia na základe typu objektu. Dá sa to pomocou operátora **instanceof**:

```
if (o instanceof Trojuholnik) {
    . . .
}
```

To isté môžeme dosiahnuť aj pomocou metódy `isInstance`():

```
if (Trojuholnik.class.isInstance(o)) {
    . . .
}
```

V obidvoch prípadoch výsledkom je boolovská hodnota, na základe ktorej sa uskutoční alebo nie telo podmieneného príkazu.

9.3 Reflektívne triedy

Literál `class` vlastne predstavuje referenciu na objekt triedy `Class`. JVM pri načítaní triedy vytvorí objekt triedy `Class`, ktorým danú triedu reprezentuje. Samotná trieda je reprezentovaná triedou, čo je prejavom tzv. reflexie. Vo všeobecnosti reflexia znamená uvažovanie o sebe samom (z prvotného významu odraz). Pre jazyk reflexia znamená uvažovanie o jeho štruktúre prostredníctvom samotného jazyka.

S touto triedou súvisia ďalšie triedy, ktorými sú reprezentované prvky tried: `Class`, `Field`, `Method` a `Constructor`. Tieto triedy sa spoločne označujú ako reflektívne triedy alebo reflektívne Java API.

V nasledujúcom zozname sú uvedené najvýznamnejšie metódy triedy `Class`:

- **boolean** `isInstance`(`Object obj`) — zistenie, či je objekt inštanciou danej triedy
- `String getName`() — zistenie názvu triedy
- **static** `Class forName`(`String className`) — zistenie objektu, ktorým je trieda daného názvu reprezentovaná
- `Constructor`[] `getConstructors`() — zistenie zoznamu konštruktorov danej triedy
- `Field`[] `getFields`() — zistenie zoznamu atribútov danej triedy
- `Class getSuperclass`() — zistenie nadtriedy danej triedy
- `Class`[] `getInterfaces`() — zistenie rozhraní danej triedy
- Object newInstance() — vytvorenie inštancie danej triedy

Predstavme si, že je v našom grafickom systéme potrebné zrátať útvary daného typu. Toto predstavuje vhodný problém pre aplikáciu reflexie:

```
public class GUtvary {
   public static int pocet(Utvar[] u, Class typ) {
      int n = 0;

      for (int i = 0; i < u.length; i++)
         if (typ.isInstance(u[i]))
```

```
            n++;

        return n;
    }

    public static void main(String[] args) {
        Utvar[] o = {
            new Trojuholnik(), new Kruh(), new Kruh()};

        System.out.println(pocet(o, Kruh.class));
    }
}
```

10 ZOSKUPENIA OBJEKTOV A GENERICKOSŤ

Zoskupenia objektov v Jave podporuje samotný jazyk v tvare polí, o ktorých sme hovorili v časti 3.15. V užšom zmysle termín zoskupenia (collections) označuje zoskupenia objektov vytvorené pomocou na to určených tried v Java API. Niekedy sú tieto triedy označované ako kontejnery (containers).

V tejto kapitole sa budeme venovať Java API pre podporu zoskupení. Táto časť Java API sa volá Collections Framework.[1]

10.1 Typy zoskupení

Typy zoskupení v Collections Framework sú nasledujúce:

- zoskupenie (collection) — v užšom zmysle:
  - zoznam (list)
  - množina (set)
- tabuľka (map)

Od verzie 5 všetky zoskupenia v Jave uchovávajú prvky ľubovoľného typu. Predtým to bol len všeobecný typ `Object`. Po vybratí objektu zo zoskupenia sa muselo urobiť pretypovanie, teda downcasting, ktorý — ako vieme — nemusí vždy dopadnúť dobre (pozri časť 5.3).

[1] `http://java.sun.com/j2se/1.4.2/docs/guide/collections/`

10.2 Generickosť zoskupení

Generickosť je nová črta pridaná v Jave 5. Zoskupenia v rámci Collections Framework sú generické. Pri vytváraní inštancie zoskupenia treba uviesť ako parameter typ objektov, ktoré sa v ňom budú uchovávať.

Napríklad zoznam objektov typu String vytvoríme takto:

```
List<String> list = new ArrayList<String>();
```

Takýto typ (List<String>) sa označuje ako parametrizovaný.

Všimnime si, že typ referencie list je List. Rozhranie List implementujú všetky zoskupenia tvaru zoznamu. Preto je lepšie deklarovať referenciu typu List než priamo ArrayList. Neskôr možno ľahko zmeniť typ použitého zoskupenia jednoduchým uvedením názvu príslušnej triedy (napríklad na LinkedList) namiesto ArrayList.

Práca so zoznamom je jednoduchá:

```
list.add("jeden");
list.add("dva");
String s = list.get(0); // s = "jeden"
```

Do zoznamu sme pridali dva prvky a zistili hodnotu prvého.

Bezpečné používanie typov stráži prekladač. Typ prvkov zoznamu list je String a nie je možné do neho vložiť napríklad prvok typu Integer:

```
list.add(new Integer(1)); // chyba pri preklade
```

Môže sa zdať, že by sa pre polymorfizmus malo dať písať:

```
List<Object> listObject = list; // chyba pri preklade
```

ale tu o polymorfizmus v skutočnosti nejde, lebo parametrizovaný nie je typ zoznamu, ale prvkov.

10.3 Náhradný znak pre typ

Čo keby sme potrebovali poskytnúť metódu, ktorá akceptuje zoskupenie hocijakých objektov? Použijeme náhradný znak (wildcard) ?. Napríklad pre zoznam by taká metóda mohla vyzerať nasledovne:

```
public class C {
   public void m(List<?> list) {
      for (int i = 0; i < list.size(); i++) {
         Object o = list.get(i));
         . . .
      }
   }
}
```

Náhradný znak pri tvorbe inštancie označuje neznámy typ:

```
List<?> l = new ArrayList<String>();
l.add(new String("s")); // chyba pri preklade
l.add(null); // jedine null sa dá vložiť
```

Do takého zoznamu nie je možné vložiť hocičo, ako by mohlo vyzerať, ale jedine hodnotu **null**. Takýto zoznam sa však dá čítať, čo sa využíva pri definovaní parametrov metód, pre ktoré potrebujeme, aby akceptovali zoskupenia objektov hocijakého typu.

Parameter generického typu je možné obmedziť vzhľadom na dedenie. Povedzme, že chceme poskytnúť metódu, ktorá ako parameter má zoznam hocijakých grafických útvarov odvodených od triedy `Utvar`. Použijeme ohraničený náhradný znak (bounded wildcard):

```
public class C {
   public static void nakresliUtvary(
      List<? extends Utvar> list) {
      . . .
   }
}
```

Takejto metóde potom možno poskytnúť všeobecný zoznam útvarov, ale aj zoznam hocijakých odvodených útvarov:

```
List<Utvar> u = new ArrayList<Utvar>();
List<Circle> k = new ArrayList<Circle>();
List<Trojuholnik> t = new ArrayList<Circle>();
. . .
C.nakresliUtvary(u);
C.nakresliUtvary(k);
C.nakresliUtvary(t);
```

Ak triedy útvarov implementujú rozhranie `Kresleny`, možno urobiť aj toto:

```
public class C {
   public static void nakresliKreslene(
      List<? extends Kresleny> list) {
      . . .
   }
}

List<Kresleny> ko = new ArrayList<Kresleny>();
. . .
C.nakresliKresleny(ko);
```

Výraz ? **extends** `T` zhora ohraničuje náhradný znak. Tento výraz špecifikuje všetky typy, ktoré dedia od typu `T`. Typ `T` môže byť rozhranie alebo trieda. Aj keď je vo výraze použité kľúčové slovo **extends**, výraz sa vzťahuje aj na dedenie typu **extends**, aj **implements**.

Dá sa špecifikovať aj zdola ohraničený náhradný znak. Nasledujúca metóda akceptuje len objekty typu `Trojuholnik` a objekty typov, od ktorých je tento typ odvodený:

```
public static void troj(
   List<? super Trojuholnik> list) {
   . . .
}
```

10.4 Kontrola typov v generickosti

Treba pamätať, že kontrola typov sa vykonáva v čase prekladu. Do zoznamov špecifikovaných pomocou ohraničených náhradných znakov nie je možné pridávať prvky.

```
public class C {
   public void m(List<? extends Kresleny> list) {
      Bod a, b, c;
      . . .
      list.add(new Trojuholnik(a, b, c)); // chyba pri
   }                                      // preklade
}
```

Keby táto situácia nebola ošetrená pri preklade, program by mohol padnúť. Typ parametra by mohol byť napríklad `List<Kruh>`, a my by sme sa do takéhoto zoznamu pokúšali pridať objekt typu `Kruh`.

Pri korektnom generickom kóde sa vždy uvádza parametrizujúci typ. Typ sa však nemusí uviesť a vtedy ide o tzv. čistý (raw) generický typ. Do takého generického typu možno vložiť hocičo:

```
List list = new ArrayList();
list.add(new String("s"));
list.add(new Integer(1));
list.add(new Object());
```

Pri čistých typoch je kontrola typov potlačená. Prekladač upozorní, že kód s čistými typmi používa nepreverené alebo nebezpečné operácie. Čisté typy slúžia na zabezpečenie kompatibility s verziami Javy nižšími ako 5, ktoré ešte nepodporovali generickosť.

10.5 Iterátory

Iterátory predstavujú spôsob ako prechádzať zoskupením bez toho, aby bolo potrebné poznať jeho typ. Každé zoskupenie dokáže poskytnúť iterátor ako objekt zavolaním metódy `iterator()`.

Uvedieme príklad na ozrejmenie práce s iterátormi. Predpokladajme, že máme triedu `Element`:

```
public class Element {
   private int n;
   public Element(int i) {
      n = i;
   }
   public void print() {
      System.out.println("Element " + n);
   }
}
```

Prvky typu `Element` uchovávame v zozname:

```
List<Element> z = new ArrayList<Element>();
```

Na výpis všetkých prvkov zoznamu použijeme iterátor:

```
Iterator i = z.iterator();

while(i.hasNext())
   i.next().print();
```

10.6 Rozšírená slučka **for**

Od verzie 5 už v Jave nemusíme priamo identifikovať iterátor — máme k dispozícii tzv. rozšírenú slučku **for**, označovanú tiež ako slučka *pre každý* (for-each), ktorá implicitne používa iterátor daného zoskupenia:

```
for (Element e : z)
   e.print();
```

Uvedená slučka **for** má nasledujúci význam: „Pre každý element `e` zo zoskupenia `z`.“

Pre názornosť tá istá iterácia v tvare obyčajnej slučky **for** s explicitne použitým iterátorom by vyzerala nasledovne:

```
for (Iterator i = z.iterator(); i.hasNext(); )
   i.next().print();
```

Rozšírenú slučku **for** možno použiť na hocijaké zoskupenie, ktoré poskytuje iterátor (objekty typu `Iterable`). Dá sa však použiť aj na polia. Vykreslenie všetkých objektov v poli objektov, ktoré sa dajú nakresliť (typu `Kreslene`), môžeme realizovať nasledovne:

```
public void nakresliKreslene(Kreslene... k) {
   for (Element e : k)
      e.nakresli();
}
```

10.7 Vymenované typy

Vymenované typy (enumerated types) zavedené v Jave 5 umožňujú definovať typ ako množinu pomenovaných hodnôt. Prvok daného vymenovaného typu potom má vždy

práve jednu z týchto hodnôt. Vymenované typy v Jave sú podobné ako vymenované typy v jazyku C, ale sú bezpečné z hľadiska typu (type-safe), lebo premennej príslušného vymenovaného typu v Jave je možné priradiť len hodnoty tohto typu. V jazyku C je možné pracovať priamo s vnútornou reprezentáciou vymenovaných typov (**int**).

Príkladom vymenovaného typu môže byť typ štúdia, ktoré môže byť bakalárske, inžinierske (magisterské) a doktorandské:

```
public enum TypStudia { BC, ING, PHD }
```

Triedu `Student` z časti 6.3 môžeme rozšíriť o typ štúdia:

```
public class Student {
   private String meno;
   private String priezvisko;
   private TypStudia typStudia;

   public void setMeno(String meno) {
      . . .
   }
   . . .
   public String toString() {
      return meno + " " + priezvisko;
   }
}
```

Rozšírená slučka **for** sa dá tiež použiť aj na vymenované typy. Predpokladajme, že máme zoznam študentov v tvare poľa:

```
Student[] zoznam;
```

Jeden zo spôsobov — aj keď nie veľmi efektívny — ako vypísať všetkých študentov rozdelených podľa typu štúdia je nasledujúci:

```
for (TypStudia t : TypStudia.values())
   for (Student s : zoznam)
      if (s.typStudia == t)
         System.out.println(s);
```

Metóda `values()` vráti zoznam všetkých hodnôt vymenovaného typu.

Tento jednoduchý príklad predstavuje najčastejší spôsob použitia vymenovaných typov. Vymenované typy v Jave môžu podobne ako triedy obsahovať aj atribúty a metódy, čo umožňuje oveľa dômyselnejšie aplikácie.[2]

[2]pozri napr. `http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html`

10.8 Generické metódy

Generické zoskupenia predstavujú príklady generických tried. Metódy v generických triedach môžu byť deklarované v zmysle parametrov typov. Príkladom je metóda `add(E o)`, kde `E` je parameter typu deklarovaný v rozhraní `List`. Metódy môžu tiež deklarovať parametre typov. Také metódy sa označujú ako generické.

Predstavme si, že potrebujeme implementovať metódu pre kopírovanie z poľa do zoskupenia.[3] Toto sa nedá urobiť:

```
static void arrayToCollection(
   Object[] a, Collection<?> c) {
   for (Object o : a)
      c.add(o); // chyba pri preklade
}
```

Prekladač hlási chybu z rovnakých dôvodov ako v príklade v časti 10.4 — typ objektu `o` by mohol byť nekompatibilný s typom zoskupenia `c`.

Riešením je generická metóda:

```
static <T> void arrayToCollection(
   T[] a, Collection<T> c) {
   for (T o : a)
      c.add(o);
}
```

Použitie generických metód bude jasnejšie po nasledujúcom príklade. Povedzme, že potrebujeme implementovať jednosmerne zreťazený zoznam. Výhodné je urobiť ho genericky, t.j. aby sa dal použiť pre údaje hocijakého typu. Generický prvok zoznamu bude vyzerať nasledovne:

```
public class SLLElement<T> {
   private T data;
   private SLLElement<T> next;

   public SLLElement() { }
   public SLLElement(T d) {
      data = d;
   }
   public T getData() {
```

[3] Tento príklad využíva kód prispôsobený z [Bra].

```
            return data;
        }
        public SLLElement<T> getNext() {
            return next;
        }
        public void setData(T d) {
            data = d;
        }
        public void setNext(SLLElement<T> e) {
            next = e;
        }
        public String toString() {
            return data.toString();
        }
    }
```

Všetky operácie nad prvkom sú parametrizované typom `T`. Samotný generický zreťazený zoznam je tiež parametrizovaný typom `T`:

```
    public class SLList<T> {
        private SLLElement<T> head;
        private SLLElement<T> tail;

        public void tailInsert(T e) {
            SLLElement<T> sll_e = new SLLElement<T>(e);
            if (head == null) {
                head = sll_e;
                tail = head;
            }
            else {
                tail.setNext(sll_e);
                tail = sll_e;
            }
            tail.setNext(null);
        }
    . . .
    }
```

Uvedená je len operácia vkladania prvku. Ostatné operácie by sa realizovali podobne.

Typ prvkov zadáme jednoducho pri vytváraní inštancie zoznamu:

```
    SLList<Integer> l = new SLList<Integer>();
```

Z ďalšieho použitia zoznamu už nie je vidieť, že je parametrizovaný:

```
for (int i = 1; i < 10; i++)
   l.tailInsert(i);
```

Všimnime si, že metóde `tailInsert()` poskytujeme parameter typu **int** namiesto typu `Integer`. Nasledujúca časť vysvetľuje mechanizmus, na ktorý sa pri tomto spoliehame.

10.9 Automatické balenie hodnôt primitívnych typov

Od verzie 5 Java zabezpečuje automatické balenie hodnôt primitívnych typov (autoboxing).[4] Túto vlastnosť sme vlastne už použili v príklade generického zoznamu z predchádzajúcej časti. Vytvorená inštancia zoznamu by mala akceptovať typ `Integer`, ale my sme vložili **int**:

```
SLList<Integer> l = new SLList<Integer>();
l.tailInsert(5);
```

V takýchto prípadoch prekladač automaticky zbalil hodnotu primitívneho typu do objektu príslušného typu, ako je uvedené v časti 3.3. Funguje aj automatické rozbaľovanie:

```
Integer o = new Integer(7);
int i = o;
```

10.10 Trieda Class a generickosť

Reflektívna trieda `Class`, o ktorej sme hovorili v časti 9.3 je tiež generická, čo umožňuje lepšiu bezpečnosť typov. V príklade s počítaním útvarov (z tej istej časti) metóde na zistenie počtu útvarov

```
public static int pocet(Utvar[] u, Class typ)
```

[4] `http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html`

na mieste parametra Typ možno zadať hocijaký typ:

```
Utvar[] o = {
   new Trojuholnik(), new Kruh(), new Kruh() };
System.out.println(pocet(o, Integer.class)); // OK (!)
```

Chceme však počítať útvary, nie hocijaké objekty, a to zabezpečíme vhodnou parametrizáciou triedy `Class`:

```
public static int pocet(
   Utvar[] u, Class<? extends Utvar> typ) {
   int n = 0;

   for (int i = 0; i < u.length; i++)
      if (typ.isInstance(u[i]))
         n++;

   return n;
}
```

Takto môžeme naďalej počítať napríklad kruhy:

```
System.out.println(pocet(o, Kruh.class)); // OK
```

ale pri zadaní triedy, ktorá nie je `Utvar` alebo odvodená trieda, prekladač hlási chybu:

```
System.out.println(pocet(o, Integer.class)); // chyba
```

11 Vstupno/výstupný systém Javy

Pôvodný vstupno-výstupný (V/V) systém Javy je zabezpečený prostredníctvom balíka Java API `java.io`. Tento V/V systém Javy je založený na koncepte prúdu údajov (stream). Od verzie 1.4 Java poskytuje aj tzv. nový V/V systém (new I/O) v balíku `java.nio`. Tento V/V systém je založený na kanáloch a vyrovnávacích pamätiach (channels and buffers). Nový V/V systém je rýchlejší, má lepšiu podporu množín znakov (t.j. rôznych jazykov) a umožňuje transparentnú prácu s veľkými súbormi.

Pre určité veci je však pôvodný V/V systém nevyhnutný — napr. pre serializáciu objektov a štandardný vstup a výstup. Pôvodný V/V systém bol reimplementovaný pomocou nového V/V systému pre zvýšenie rýchlosti.

V tejto kapitole sa oboznámime so základnými možnosťami práce s V/V systémom Javy.

11.1 Práca s adresármi

Najprv sa pozrieme, ako sa dá pristupovať k adresárom. Trieda `File` poskytuje abstraktnú reprezentáciu súborov a ciest (pathname). Abstraktné meno cesty pozostáva z voliteľného systémovo závisleho prefixu, ako je napríklad názov disku, a postupnosti názvov (žiaden alebo viac). Každý názov označuje adresár s tým, že posledný názov môže označovať súbor.

Meno cesty môže byť absolútne alebo relatívne. Absolútne meno cesty sa dá interpretovať priamo. Relatívne meno cesty sa interpretuje vzhľadom na iné meno cesty. Môže to byť napríklad aktuálny adresár, ktorý nájdeme v systémovej vlastnosti `user.dir` volaním

```
System.getProperty(user.dir);
```

Nasledujúci program načíta a vypíše obsah aktuálneho adresára:

```
import java.io.*;
import java.util.*;

public class Dir {
   public static void main(String[] args) {
      File path = new File(".");
      String[] list;
      list = path.list();
      for(int i = 0; i < list.length; i++)
         System.out.println(list[i]);
   }
}
```

Často z adresára potrebujeme len súbory, ktoré spĺňajú určité kritériá. Inými slovami potrebujeme filtrovaný obsah adresára. Na tento účel implementujeme vlastný filter podľa rozhrania `FilenameFilter`, ktorý v metóde `accept()` bude obsahovať podmienku akceptovania súborov. Povedzme, že nás zaujímajú len súbory, ktorých názov začína písmenom `x`:

```
class XFilter implements FilenameFilter {
  public boolean accept(File dir, String name) {
    return name.charAt(0) == 'x';
  }
}
```

Tento filter následne poskytneme metóde `list()` triedy `File`:

```
public class FDir {
   public static void main(String[] args) {
      File path = new File(".");
      String[] list;
      list = path.list(new XFilter());
      for(int i = 0; i < list.length; i++)
         System.out.println(list[i]);
   }
}
```

Niektoré ďalšie metódy triedy `File` sú:

- exists() — zistenie existencie súboru alebo adresára
- delete() — vymazanie súboru alebo adresára

- isFile() — zistenie, či prvok adresára je súbor
- isDirectory() — zistenie, či prvok adresára je adresár
- isHidden() — zistenie, či prvok adresára je skrytý
- mkdirs() — vytvorenie nového adresára
- renameTo() — premenovanie súboru alebo adresára

11.2 V/V systém Javy založený na prúdoch

Ako sme už povedali, pôvodný V/V systém Javy je zabezpečený prostredníctvom balíka Java API `java.io` a založený je na koncepte prúdu údajov. Údaje môžu byť v tvare bajtov (8 bitov) alebo znakov (16 bitov).

Prácu s prúdom bajtov (byte stream) podporujú abstraktné triedy `InputStream` a `OutputStream`. S prúdom znakov (character stream) môžeme pracovať prostredníctvom abstraktných tried `Reader` a `Writer`.

Od týchto tried sú odvodené ďalšie triedy pre rôzne typy údajov, ktoré sa čítajú a zapisujú. Súčasťou V/V systému je aj trieda `RandomAccessFile`, ktorá je mimo tejto hierarchie. Od verzie 1.4 ju v podstate nahradili tzv. súbory zobrazené do pamäte (vysvetlené v časti 11.9).

Prúd sa otvára vytvorením príslušného objektu. Otvorme napríklad súbor s názvom `data.txt`:

```
FileReader file = new FileReader("data.txt");
```

Prúd sa zatvára metódou `close()`. Súbor, ktorý sme otvorili v predchádzajúcom príklade, zatvoríme veľmi jednoducho:

```
file.close();
```

Zavretie súboru by sa malo udiať pri finalizácii, ale nie je to zaručené. Preto je lepšie zatvárať prúdy explicitne.

Štandardný vstup, štandardný výstup a štandardný výstup pre chyby (vrátane obaľovacích prúdov) by sa nikdy nemali zatvárať. Tieto prúdy sú dostupné ako statické atribúty triedy `System`: `System.in`, `System.out` a `System.err`. Tieto atribúty sú finálne — nedá sa im priradiť nový prúd.

11.3 Čítanie súboru po riadkoch

Jedným z typických použití V/V systému je čítanie súboru po riadkoch. Majme nasledujúcu triedu uloženú v súbore s názvom `Subor.java`:

```
public class Subor {
   public static void main(String[] args)
      throws IOException {
      BufferedReader in = new BufferedReader(
         new FileReader("Subor.java"));
      String s, s2 = new String();

      while ((s = in.readLine())!= null)
         s2 += s + "\n";

      in.close();
   }
}
```

Metóda `main()` tejto triedy načíta po riadkoch súbor, v ktorom je trieda deklarovaná a uloží ho do premennej `s2`. Za každým riadkom pritom vloží znak nového riadku.

Operácie so súbormi vyhadzujú výnimku `IOException` ak niečo nie je v poriadku. Preto v našom príklade metóda `main()` deklaruje, že vyhadzuje tento typ výnimky. V skutočnosti by sme mali výnimku ošetriť vzhľadom na kontext, v ktorom vznikla (ako bolo vysvetlené v kapitole 8).

Pre rýchlejší a jednoduchší prístup k údajom vstupný prúd je obalený (wrapped) do objektu triedy `BufferedReader`, ktorý zabezpečí prístup k súboru prostredníctvom vyrovnávacej pamäte. Takéto obaľovanie sa vo V/V Javy často využíva.

11.4 Štandardný V/V systém

Pre obvyklé operácie so vstupom a výstupom mnohé operačné systémy poskytujú pojem štandardného V/V systému. Programovacie jazyky umožňujú narábanie s týmto štandardným V/V systémom. Rovnako je to aj v Jave, ktorá poskytuje štandardný V/V systém v tvare prúdov dostupných ako statické atribúty triedy `System`:

- štandardný vstup — `System.in`
- štandardný výstup — `System.out`

- štandardný výstup pre chyby — `System.err`

Štandardný vstup a výstup pre chyby sú obalené v objekte typu `PrintStream`, čo znamená, že pracujú so znakmi. Štandardný vstup je však typu `BufferedInputStream`, čo znamená, že pracuje s bajtmi. Pred použitím ho je vhodné obaliť do objektu triedy `InputStreamReader`. Nasledujúci príklad ukazuje toto obalenie a následné načítanie riadku zo štandardného vstupu:

```
class C {
   public static void main(String[] args)
      throws IOException {
      BufferedReader stdin = new BufferedReader(
         new InputStreamReader(System.in));

      System.out.print("Vstup: ");
      String s = stdin.readLine();
      System.out.println(s);
   }
}
```

Od verzie 5 Java API poskytuje textový skener pre načítavanie s rozborom (parsing)[1] Tento skener sa dá použiť na hocijaký vstupný prúd, reťazec a kanál. Presnejšie zadanie formátu je možné pomocou triedy `java.util.regex.Pattern`. Rozsiahlejšie možnosti na rozbor reťazcov poskytujú regulárne výrazy, čím sa v tomto texte nebudeme zaoberať. Nasledujúci príklad ukazuje jednoduché načítavanie údajov rôzneho typu pomocou skenera:

```
Scanner sc = new Scanner(System.in);
System.out.print("Ulica a cislo: ");
String ulica = sc.next();
int cislo = sc.nextInt();
System.out.println(s + " " + i);
```

11.5 Formátovaný výstup

Od verzie 5 Java API poskytuje aj formátovaný výstup. Prúdy typu `PrintStream` poskytujú metódu `printf()`. Použitie je podobné ako v jazyku C:

[1] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html`

```
double a = 10000.0;
System.out.printf("a = %e%n", a);
```

Všimnime si, že namiesto obvyklého znaku nového riadku `\n` je v príklade použitý znak `%n`, ktorý zaručuje správnu platformovo-špecifickú interpretáciu. Podrobný opis možností formátovaného výstupu je mimo rozsahu tohto textu. Podrobnosti možno nájsť v dokumentácii triedy `java.util.Formatter`.[2]

11.6 Čítanie z pamäte po znakoch

Niekedy je vhodné pristupovať k reťazcu znakov ako k prúdu. Na to je možné použiť triedu `StringReader`. Ukazuje to nasledujúci príklad:

```
class C {
   public static void main(String[] args)
      throws IOException {
      String s = new String("12356789");
      StringReader in = new StringReader(s);

      int c;

      while ((c = in.read()) != -1)
         System.out.print((char)c);

      in.close();
   }
}
```

11.7 Čítanie z pamäte podľa formátu údajov

Na výber údajov z prúdu podľa formátu, možno použiť triedu `DataInputStream`. Potom možno vyberať údaje v tvare primitívnych typov v Jave nezávisle od strojovej reprezentácie. Nasledujúci príklad ukazuje čítanie bajtov z reťazca znakov:

[2] pozri `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html`

```
class C {
   public static void main(String[] args)
      throws IOException {
      String s = new String("123456789");
      DataInputStream in = null;
      try {
         in = new DataInputStream(
            new ByteArrayInputStream(s.getBytes()));

         while (true)
            System.out.print((char)in.readByte());

         System.out.println();
      } catch(EOFException e) {
         System.err.println("\nKoniec prúdu");
      }
      finally {
         in.close();
      }
   }
}
```

11.8 Zápis a čítanie súborov

V tejto časti sa prostredníctvom niekoľkých príkladoch pozrieme na zápis a čítanie zo súborov.

Zápis a čítanie znakov

Najprv sa pozrieme na zápis a čítanie znakov. V príklade zapíšeme reťazec uložený v objekte `s` do súboru s názvom `demo.out` po riadkoch, pričom riadky očíslujeme:

```
class C {
   public static void main(String[] args)
      throws IOException {
      String s = new String("123\nabc");
      BufferedReader in = null;
      PrintWriter out = null;
      try {
         in = new BufferedReader(new StringReader(s));
         out = new PrintWriter(new BufferedWriter(
```

```
                new FileWriter("demo.out")));
            int l = 1;

            while ((s = in.readLine()) != null )
                out.println(l++ + ": " + s);
                // c. riadku: obsah
        } catch(EOFException e) {
            System.err.println("Koniec prúdu");
        } finally {
            in.close();
            out.close();
        }
    }
}
```

Ako vstupné údaje sme použili znaky uložené v objekte typu `String`, ku ktorému pristupujeme ako k prúdu znakov obalením do objektu triedy `StringReader` (pozri 11.6). Pre urýchlenie prístupu prostredníctvom vyrovnávacej pamäte tento objekt je obalený do objektu triedy `BufferedReader`.

Výstupom bude súbor pre zápis otvorený vytvorením objektu triedy `FileWriter`. Podobne ako pri vstupnom prúde, aby prístup k obsahu súboru prostredníctvom vyrovnávacej pamäte bol rychlejší, tento objekt je obalený do objektu triedy `BufferedWriter`. Tento objekt je pre zjednodušenie manipulácie údajmi zase obalený do objektu triedy `PrintWriter`. Takto je možné zapisovať údaje po celých riadkoch.

Pri súboroch treba zvlášť dbať o explicitné zatvorenie prúdu. Neuzatvorenie príslušného prúdu ponecháva súbor otvorený.

Binárne súbory

Nasledujúci príklad ukazuje vytvorenie súboru s binárnymi údajmi. Do súboru zapíšeme text v kódovaní UTF a decimálne číslo v dvojitej presnosti:

```
class C {
    public static void main(String[] args)
        throws IOException {
        DataOutputStream out = null;
        DataInputStream in = null;
        try {
            out = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("data.txt")));

            out.writeUTF("Text");
```

```
            out.writeDouble(1.1);

            in = new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream("data.txt")));

            System.out.println(in.readUTF());
            System.out.println(in.readDouble());
         } catch(EOFException e) {
            throw new RuntimeException();
         } finally {
            in.close();
            out.close();
         }
      }
   }
```

Znovu sme využili vyrovnávaciu pamäť pre rýchlejší prístup k obsahu súboru (triedy `BufferedOutputStream` a `BufferedInputStream`). Obalenie výstupného prúdu do objektu triedy `DataOutputStream` zaručuje, že môžeme použiť metódy pre zápis primitívnych typov údajov, UTF a objektov. Obalenie vstupného prúdu do objektu triedy `DataInputStream` umožňuje ich správne načítanie pomocou inverzných metód.

Súbory s voľným prístupom

Ak je v súbore potrebné sa veľa posúvať, jednoduchšiu prácu so súbormi umožňuje trieda `RandomAccessFile`. Táto trieda, pomerne izolovaná od zvyšku V/V systému, umožňuje tzv. voľný prístup (random access) k súborom.[3] V otvorenom súbore s voľným prístupom sa dá jednoducho pohybovať dopredu a dozadu. Z nastavenej pozície je možné údaje čítať alebo zapisovať. Súbor s voľným prístupom môže byť otvorený pre čítanie a zápis súčasne.

Nasledujúci príklad ukazuje prácu so súborom s voľným prístupom:

```
   class C {
      public static void main(String[] args)
         throws IOException {
         RandomAccessFile f =
            new RandomAccessFile("test.dat", "rw");

         for(int i = 0; i < 10; i++)
```

[3] Anglický výraz *random access* sa najčastejšie prekladá ako *priamy prístup*, lebo k pamäti pristupujeme priamo, ale podstata je vo *voľnosti* prístupu, čo je aj skutočný význam tohto termínu.

```
            f.writeInt(i);

        f.seek(3*4); // nastavenie za tretí int
        f.writeInt(-1);

        f.seek(0); // nastavenie na začiatok

        for(int i = 0; i < 10; i++)
            System.out.println(i + ": " + f.readInt());

        f.close();
    }
}
```

Do súboru zapíšeme čísla 0–9 ako celé čísla. Príkazom

```
f.seek(3*4);
```

sa nastavíme za tretie z nich (celé číslo zaberá v Jave štyri bajty) a zapíšeme −1. Nastavíme sa na začiatok súboru a prečítame jeho obsah. Namiesto čísla 2 v súbore bude číslo −1.

Od verzie 1.4 Java poskytuje tzv. súbory zobrazené do pamäte (pozri časť 11.10), ktoré predstavujú vhodnú alternatívu k súborom s voľným prístupom.

11.9 Kanály a vyrovnávacie pamäte

Ako bolo povedané na začiatku kapitoly, od verzie 1.4 Java poskytuje aj tzv. nový V/V systém (new I/O) v balíku `java.nio`, ktorý je rýchlejší, má lepšiu podporu množín znakov a umožňuje transparentnú prácu s veľkými súbormi.

Nový V/V systém je založený na kanáloch a vyrovnávacích pamätiach. Kanály (channels) predstavujú abstrakciu pre prístup k údajom v súboroch alebo hociktorých iných prostriedkoch, ktoré umožňujú V/V operácie. Ku kanálom sa pristupuje vždy prostredníctvom vyrovnávacích pamätí (buffers).

Kanály pre prácu so súbormi možno získať aj z nasledujúcich tried V/V systému založeného na prúdoch: `FileInputStream`, `FileOutputStream` a `RandomAccessFile`. V nasledujúcom príklade získame kanál pre prístup k súboru otvoreného prostredníctvom triedy `FileInputStream`:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

class C {
   public static void main(String[] args)
      throws Exception {
      FileChannel fc =
         new FileOutputStream("data.txt").getChannel();

      fc.write(
         ByteBuffer.wrap("Text... ".getBytes()));

      fc.close();
   }
}
```

Vyrovnávacie pamäte sú založené na bajtoch. Pri práci s reťazcami znakov preto treba použiť metódu `getBytes()` triedy `String`.

11.10 Súbory zobrazené do pamäte

Súbory zobrazené do pamäte (memory-mapped files) predstavujú lepšiu alternatívu súborov s priamym prístupom. Založené sú priamo na kanáloch a vyrovnávacích pamätiach, a preto je práca s nimi oveľa rýchlejšia.

Súbor zobrazený do pamäte sa programátorovi javí ako keby bol celý v pamäti. V/V systém zabezpečuje, aby sa potrebné časti nachádzali v pamäti.

Nasledujúci príklad predstavuje malú ukážku práce s veľkým (100 MiB) súborom zobrazeným do pamäte:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

class C {
   static int length = 0x6400000; // 100 MiB
   public static void main(String[] args)
      throws Exception {
      MappedByteBuffer out =
         new RandomAccessFile("test.dat", "rw").
```

```
                getChannel().
                    map(FileChannel.MapMode.READ_WRITE,
                        0, length);

        for(int i = 0; i < length; i++)
            out.put((byte)(i % 9));

        for(int i = length - 9; i < length; i++)
            System.out.print(out.get(i));
    }
}
```

Do súboru zapíšeme 100 MiB číslic 0–9, a následne vypíšeme posledných desať:

```
4567890123
```

11.11 Kompresia

V/V systém Javy obsahuje podporu kompresie typu ZIP a GZIP. Táto kompresia je založená na bajtovo orientovaných triedach `InputStream` a `OutputStream`.

Aj samotné tzv. JAR súbory (Java Archive), ktoré slúžia na distribúciu preložených tried, sú vlastne ZIP archívy. JAR súbory sa vytvárajú pomocou programu `jar`, ktorý je súčasťou SDK. Samotné **class** súbory v JAR archívoch sa dajú priamo používať a teda pre spustenie programu nie je potrebné archív rozbaliť. Pritom `classpath` musí zahŕňať príslušný JAR archív (ako adresár).

11.12 Serializácia objektov

Serializácia objektov je spôsob transformácie objektov do postupností bajtov. Serializácia objektov umožňuje ukladanie objektov do súborov a ich neskoršiu obnovu.

Java API umožňuje jednoduchú serializáciu objektov, čo predstavuje podporu perzistencie ako jedného z mechanizmov objektovo-orientovaného programovania. Týmto je umožnený prenos objektov cez sieť a ich obnovu na strane prijímateľa, čo je nevyhnutné pre prenos parametrov pri volaní vzdialených metód (Remote Method Invocation, RMI). Serializácia je nevyhnutná aj pre prácu s tzv. JavaBean objektmi, ktoré v tomto texte nerozoberáme.

Aby sa dal serializovať, objekt musí implementovať rozhranie `Serializable`. Toto

rozhranie neobsahuje metódy — slúži len na značenie. Serializácia sa uskutočňuje pre celú spleť objektov, na ktoré sa daný objekt odkazuje. Pre obnovenie objektov sú potrebné **class** súbory. Samotná serializácia sa vykonáva prostredníctvom tried `ObjectOutputStream` a `ObjectInputStream`.

Predpokladajme, že máme nasledujúcu triedu:

```
class Data implements Serializable {
   int n;
   Data next;
   public Data(int n) {
      this.n = n;
   }
}
```

Trieda `Data` umožňuje reťazenie objektov. V nasledujúcom príklade vytvoríme dva takto spojené objekty (referencia `d1`) a serializujeme ich. Následne ich obnovíme (referencia `d2`) a presvedčíme sa, že sú prepojené a obsahujú pôvodné údaje.

```
class C {
   public static void main(String[] args)
      throws ClassNotFoundException, IOException{
      Data d1 = new Data(1);
      d1.next = new Data(2);

      ObjectOutputStream out =
         new ObjectOutputStream(
            new FileOutputStream("d.out"));

      out.writeObject(d1);
      out.close();

      ObjectInputStream in =
         new ObjectInputStream(
            new FileInputStream("d.out"));

      Data d2 = (Data)in.readObject();

      System.out.println(d2.n + ", next " + d2.next.n);
   } // 1, next 2
}
```

Niekedy je potrebné mať väčšiu kontrolu nad serializáciou. Toto sa dá dosiahnuť použitím rozhrania `Externalizable` namiesto `Serializable`, čím trieda preberá zod-

povednosť za serializáciu a môže ju plne riadiť. Takáto trieda musí implementovať serializáciu v metódach `writeExternal`() a `readExternal`().

Iná možnosť je použitie rozhrania `Serializable` a jeho metód `writeObject`() a `readObject`(). V rámci týchto metód objekt môže zavolať implicitnú serializáciu implementovanú v metódach `defaultWriteExternal`() a `defaultReadExternal`().

Z hľadiska ochrany údajov je dôležité si uvedomiť, že pri serializácii sa ukladajú hodnoty všetkých atribútov objektu bez ohľadu na modifikátory prístupu. Zabrániť uloženiu hodnoty atribútu sa dá použitím kľúčového slova **transient** pri jeho deklarácii.

12 Viacniťovosť

Málokedy na počítači prebieha len jedna úloha. Prostriedky sú však obmedzené, a niektoré sa ani nedajú zdieľať. Predovšetkým sa to týka procesora. Tento problém vzniká aj v prípade viacprocesorových systémov, lebo ťažko možno predpokladať obmedzenie počtu úloh na počet procesorov. Riešením je prepínanie medzi úlohami, ktoré zabezpečí operačný systém.

Takéto úlohy sa označujú ako procesy. Každý proces má vlastný pamäťový priestor. V rámci procesu môže jestvovať viac paralelných tokov riadenia, ktoré zdieľajú jeho pamäťový priestor. Tieto „ľahké" procesy sa označujú ako *nite* (angl. thread).[1]

S pojmami procesu a nite sa pracuje v rozsiahlych oblastiach súbežného (angl. concurrent), distribuovaného a paralelného programovania. V tejto kapitole sa obmedzíme na podporu tzv. viacniťovosti (angl. multithreading) v Jave a pozrieme sa, ako sa nite v Jave vytvárajú (časť 12.1), riadia (časť 12.2) a synchronizujú (časť 12.3).

12.1 Vytváranie nití

V Jave sa niť dá vytvoriť dvomi spôsobmi:

- rozšírením triedy `Thread`
- implementáciou rozhrania `Runnable` (na spustenie sa následne použije trieda `Thread`)

Trieda `Thread` a rozhranie `Runnable` sú súčasťou Java API. V obidvoch spôsoboch sa samotná činnosť nite definuje v metóde `run`(). Niť sa spustí vyvolaním metódy `start`().

Nasledujúci príklad demonštruje vytvorenie a spustenie nite rozšírením triedy `Thread`:

```
class Nit extends Thread {
   int n = 0;
```

[1] *Thread* sa často prekladá ako *vlákno*, ale potom sa stráca alúzia na niť v zmysle niť príbehu alebo myšlienok.

```
    public Nit(int n) { this.n = n; }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("T" + n + ": " + i);
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Nit(i).start();
        }
    }
```

Vidíme, že sa v metóde `main()` vytvorí a spustí päť nití. Úlohou každej z nití je vypísať čísla od 0 do 9. Pred každým číslom niť vypisuje aj svoju identifikáciu (`T` a číslo nite), a tak môžeme sledovať prepínanie nití.

To isté sa dá dosiahnuť aj vytvorením nite implementáciou rozhrania `Runnable`:

```
class Nit implements Runnable {
    int n = 0;
    public Nit(int n) { this.n = n; }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("T" + n + ": " + i);
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Thread(new Nit(i)).start();
    }
}
```

Tento spôsob je výhodný ak potrebujeme triedu, ktorá predstavuje niť, odvodiť od nejakej inej triedy. Dedením od triedy `Thread` strácame túto možnosť.

12.2 Riadenie nití

Bez ohľadu na spôsob jej vytvorenia niť nakoniec bude reprezentovaná objektom triedy `Thread`. Máme možnosť ju riadiť prostredníctvom metód tejto triedy, ako sú napríklad:

- **static void** `yield()` — niť, ktorá práve prebieha, sa vzdá riadenia, čím sa umožní prepnutie na inú niť

- **static void** `sleep`(**long** `millis`) — niť, ktorá práve prebieha, sa zastaví na zadaný čas (v milisekundách)
- **void** `interrupt`() — preruší niť
- **void** `wait`() — zastaví niť kým niekto nezávolá jej metódu `notify`()
- **void** `join`() — niť bude čakať na ukončenie nite, ktorej metódu `join`() zavolala, predtým ako bude pokračovať vo vykonávaní
- **void** `setPriority`(**int** `newPriority`) — nastaví prioritu nite
- **void** `setDeamon`(**boolean** `on`) — nastaví, aby niť bola démonická alebo nie

Niektoré nite predstavujú podporné služby, ktoré prebiehajú v pozadí, kým sa program vykonáva. Po ukončení programu tieto nite už nie sú potrebné a môžu sa automaticky ukončiť. Toto sa dá dosiahnuť nastavením, aby tieto nite boli tzv. démoni alebo démonické nite (daemon threads).

12.3 Synchronizácia nití

K prepnutiu medzi niťami môže dôjsť aj pri prístupe k zdieľaným zdrojom. Tomu sa dá zabrániť tzv. synchronizáciou nití. Podstata je v uzamknutí objektu pre kritickú oblasť (critical section) programu, ktoré sa zabezpečuje kľúčovým slovom **synchronized** pred synchronizovaným blokom (kritickým oblasťou) s uvedením referencie objektu, ktorý má byť uzamknutý:

```
synchronized (referenciaObjektu) {
    . . . // kritická oblasť
}
```

Synchronizovaná môže byť aj celá metóda

```
class Trieda1 {
   synchronized void op1() {
       . . .
   }

   static synchronized void op2() {
       . . .
   }
}
```

Pri metóde inštancie sa uzamyká aktuálny objekt, t.j. **this**:

```
   void op1() {
      synchronized (this) {
         . . .
      }
   }
```

Pri statických metódach sa uzamyká objekt triedy — dostupný prostredníctvom metódy `getClass()` príslušnej triedy:

```
   static void op2() {
      synchronized (getClass()) {
         . . .
      }
   }
```

Pozrime sa bližšie na synchronizáciu nití prostredníctvom jednoduchého príkladu. Majme triedu Test:

```
class Test {
   static int i = 0, j = 0;
   static void increment() {
      i++;
      j++;
   }
   static void print() {
      if (i != j)
          System.out.println("i=" + i + " j=" + j);
   }
   public static void main(String[] args) {
      new Nit1().start();
      new Nit2().start();
   }
}
```

Táto trieda obsahuje dva atribúty, `i` a `j`. Klientom poskytuje dve metódy: metóda `increment()` inkrementuje obidva atribúty, kým metóda `print()` vypíše hodnotu obidvoch atribútov, ak sa líšia.

Metóda triedy Test spúšťa dve nite typu Nit1 a `Nit2`. Tieto nite sú definované nasledujúcim kódom:

```
class Nit1 extends Thread {
   public void run() {
      for (int i = 0; i < Integer.MAX_VALUE; i++) {
         Test.increment();
      }
   }
```

```
}

class Nit2 extends Thread {
   public void run() {
      for (int i = 0; i < Integer.MAX_VALUE; i++) {
         Test.print();
      }
   }
}
```

`Nit1` inkrementuje obidva atribúty triedy `Test` po maximálnu hodnotu typu **int**. `Nit2` volá spomínanú metódu `print`() triedy `Test`. Keďže sa atribúty inkrementujú len pomocou metódy increment(), zdá sa, že k tomu, aby sa hodnoty atribútov `i` a `j` líšili, nemôže dojsť. Nie je však isté, že sa atribúty `i` a `j` budú inkrementovať zároveň, lebo k prepnutiu nite môže dôjsť po inkrementácii atribútu `i`, ale pred inkremenáciou atribútu `j`. Podobne môže dôjsť k prerušeniu vyhodnotenia podmieneného príkazu v metóde `print`(). Občas sa teda uskutoční výpis v metóde `print`().

Aby sme sa vyhli tomuto problému, metódy `increment`() a `print`() musia byť deklarované ako synchronizované:

```
class Test {
   static int i = 0, j = 0;

   static synchronized void increment() {
      i++;
      j++;
   }

   static synchronized void print() {
     if (i != j)
        System.out.println("i=" + i + " j=" + j);
   }

   public static void main(String[] args) {
      new Nit1().start();
      new Nit2().start();
   }
}
```

Synchronizovaná metóda má výlučný prístup k objektu **this**, čím sa zaručí, že sa inkrementácia `i` a `j` vykoná bez prerušenia a neobjaví sa žiaden výpis z metódy print(). Dôležité je podotknúť, že obidve metódy musia byť synchronizované.

13 Grafické používateľské rozhranie v Jave

Časť programu, ktorá umožňuje interakciu s používateľom, sa volá používateľské rozhranie (user interface). JDK poskytuje rozsiahlu podporu pre tvorbu grafického používateľského rozhrania a pre prácu s grafikou vôbec. Oblasť grafiky ako takej je mimo zámeru tohto textu. Oboznámime sa však s princípmi, na ktorých spočíva grafické používateľské rozhranie v Jave, vytvorené pomocou rámca Swing.

13.1 Používateľské rozhranie

Väčšina programov predpokladá určitú interakciu s používateľom. Pri niektorých programoch je to len zadanie vstupných parametrov, na základe ktorých program vytvorí výstup, ktorý dodá používateľovi. Takýmto spôsobom pracujú napríklad príkazy v príkazovom riadku v operačnom systéme DOS alebo shell v operačnom systéme Unix. Interakcia s používateľom pri niektorých programoch predstavuje podstatu ich činnosti. Príkladov takýchto programov je veľa: programy na úpravu textu, kreslenie, grafické modelovanie, informačné systémy atď.

Základné typy používateľských rozhraní sú príkazový riadok (command line interface, CLI) a grafické používateľské rozhranie (graphical user interface, GUI). V dnešných systémoch prevláda grafické používateľské rozhranie založené na oknách, t.j. WIMP štýl (Window, Icon, Menu, Pointing device) používateľského rozhrania vyvinutý v PARCu v roku 1984 a popularizovaný Macintoshom.[1] Jestvujú aj špecializované grafické používateľské rozhrania, ktoré často predpokladajú aj použitie špeciálnych jednotiek na ovládanie aplikácie. Príkladom môžu byť dokonca bežné počítačové hry alebo dômyselné systémy založené na virtuálnej realite.

[1] pozri `http://en.wikipedia.org/wiki/History_of_the_GUI`

13.2 Swing

Swing je aplikačný rámec (framework) pre grafické používateľské rozhranie. Swing je súčasťou tzv. Java Foundation Classes (JFC), ktoré obsahujú aj rámce Abstract Window Toolkit (AWT) a Java2D. Rámec AWT umožňuje tvorbu GUI pomocou prvkov GUI, ktoré poskytuje daný operačný systém. Má značné obmedzenia v zmysle GUI, ale poskytuje vhodný model udalostí, na ktorom je postavený aj Swing. Z alternatívnych rámcov je pomerne populárny Standard Widget Toolkit (SWT).[2].

Tvorba GUI v rámci Swing sa dá realizovať aj pomocou editorov GUI. Toto je možné vďaka tomu, že je Swing založený na *komponentovom prístupe* nazývanom JavaBeans. JavaBean ako komponent je reprezentovaný triedou pre zachovanie transparentnosti z hľadiska objektovo-orientovaného prístupu. Podpora tvorby JavaBeanov je možná priamo v integrovanom vývojovom prostredí (napr. v NetBeans a Eclipse), ale jestvujú aj samostatné generátory GUI.[3]

Jednou z podmienok, aby sa s triedou mohlo pracovať ako s JavaBeanom, je dodržanie konvencie pomenúvania (set/get metódy) a prístupnosti metód a atribútov (označených ako *properties* — vlastnosti). Nebudeme sa tu zaoberať JavaBeanmi vo všeobecnosti. Z pohľadu Swingu, ako uvidíme v ďalšom texte, je dôležité, že JavaBeany sú riadené *udalosťami* (events). Pod udalosťou rozumieme zmenu v aplikácii spôsobenú vonkajším podnetom alebo ukončením určitého procesu v aplikácii. Komponent riadený udalosťami sa registruje prostredníctvom príslušného prijímača udalostí, ktorý v prípade vzniku udalosti vyvolá jej spracovanie týmto komponentom. Viac o riadení udalostí v kontexte Swingu budeme hovoriť v časti 13.5.

Jestvujú dva základné typy komponentov Swingu: kontejnery a atomické komponenty. Kontejnery (containers) sú odvodené od triedy `JContainer`. Ako vyplýva z názvu, kontejnery poskytujú priestor pre iné komponenty. Komponent v kontejneri sa označuje ako jeho potomok (child). Rozloženie komponentov riadi objekt typu `LayoutManager`, o čom budeme hovoriť v časti 13.4.

Atomické komponenty sú odvodené od triedy `JComponent`. Umožňujú interakciu s používateľom. Typické atomické komponenty sú tlačidlá, textové polia, označenia (labels) a pod.

Rozlišujeme kontejnery na vrchnej úrovni a sprostredkovateľské konejnery. Kontejner na vrchnej úrovni (top-level containers) môže byť jedného z týchto typov:

- `JFrame` (a `JWindow`) — okná
- `JDialog` — dialógy
- `JApplet` — applety

[2] `http://www.eclipse.org/swt/`
[3] pozri `http://www.fullspan.com/articles/java-gui-builders.html`

Nebudeme sa zaoberať appletmi, avšak možno poznamenať, že applet („malá“ aplikácia) je program v Jave, ktorý sa spúšťa v rámci webového prehliadača. Applety pre zachovanie bezpečnosti majú obmedzený prístup k lokálnemu disku. Program v Jave možno pripraviť aj tak, aby sa dal spúšťať ako applet, aj ako samostatná aplikácia [Eck02].

`JPanel` patrí medzi sprostredkovateľské kontejnery (intermediate containers). Ďalšie typy sprostredkovateľských kontejnerov sú napríklad `JScrollPane` a `JSplitPane`. `JPanel` sa dá využiť aj na priame kreslenie grafických útvarov.

Okrem tvorby vlastného dialógu pomocou triedy `JDialog` možno využiť predpripravené štandardné dialógy: `JOptionPane`, `JFileChooser`, `JColorChooser`, `JProgressBar` a `ProgressMonitor`. Takéto dialógy sú *modálne*, čo znamená, že sa musia ukončiť, aby mohla pokračovať ďalšia interakcia používateľa so zvyškom GUI.

13.3 Tvorba okien

Bežné okná, ktoré obsahujú základné ovládacie prvky (na zmenu veľkosti a pod.) a názov, sa implementujú pomocou triedy `JFrame`. V najjednoduchšom prípade môžeme len vytvoriť objekt tejto triedy:

```
JFrame w = new JFrame("Moje okno");
```

Lepšie je však odvodiť vlastné okno dedením od triedy `JFrame`. Trieda `JWindow` predstavuje tiež okná, ale „holé“ — bez základných ovládacích prvkov a názvu.

Tu je úplný kód programu, ktorý vytvorí jednoduché okno (pozri obr. 13.1) a nastaví mu názov na „Moje okno“ a veľkosť na 300 x 300 bodov:

```
import javax.swing.*;

class C {
  public static void main(String[] args) {
    JFrame w = new JFrame("Moje okno");
      w.setSize(300, 300);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setVisible(true);
  }
}
```

V tomto okne je metódou `setDefaultCloseOperation()` nastavené, že kliknutie na ikonu zavretia okna („x“ v hornom pravom rohu) naozaj spôsobí jeho zavretie. Bez

tohto nastavenia by sa okno nezavrelo. V prípade potreby okno môžeme len skryť, ak použijeme hodnotu `JFrame.HIDE_ON_CLOSE` (ďalšie možnosti sú opísané v dokumentácii Java API).

Obrázok 13.1: Jednoduché okno.

Vytvorenie okna dedením od `JFrame` umožní jednoduchšiu tvorbu podobných okien:

```
import javax.swing.*;

class MyWindow extends JFrame {
  public MyWindow() {
    setTitle("Moje okno");
    setSize(300, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
class C {
  public static void main(String[] args) {
    JFrame w = new MyWindow();
    w.setVisible(true);
  }
}
```

13.4 Pridávanie komponentov do JFrame

Komponenty sa vytvárajú podobne ako kontejnery — ako objekty príslušnej triedy. Napríklad takto sa vytvorí tlačidlo:

```
JButton b1 = new JButton("Tlačidlo 1");
```

Ak sa vytvára viac podobných, špecializovaných komponentov (napr. podobné tlačidlá) je vhodné — ako pri oknách — použiť dedenie.

Do `JFrame` možno pridať komponenty, ale nepriamo, do jeho `JPanelu`:

```
w.getContentPane().add(b1);
```

Od verzie 5 Javy priame pridanie do `JFrame` má rovnaký účinok:

```
w.add(b1);
```

Nasledujúci program vytvorí okno s tlačidlom s názvom `Tlačidlo` 1 (pozri obr. 13.1):

```
import javax.swing.*;

class C {
  public static void main(String[] args) {
    JFrame w = new JFrame("Moje okno");
    w.setSize(300, 300);
    JButton b1 = new JButton("Tlačidlo 1");
    w.add(b1);
    w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    w.setVisible(true);
  }
}
```

Po spustení predchádzajúceho programu sme videli, že tlačidlo zaberá celé okno. Swing umožňuje nastavovanie rozloženia prvkov pomocou tried odvodených od triedy `LayoutManager`. Najčastejšie používané rozloženia sú:

- `FlowLayout` (prednastavené pre `JPanel`)
- `BorderLayout` (prednastavené pre `contentPane`)

Obrázok 13.2: Okno s tlačidlom.

- GridLayout
- BoxLayout
- GridBagLayout

Je možné aj absolútne umiestňovanie prvkov poskytnutím referencie **null** namiesto referencie na objekt typu `LayoutManager` (null layout), ale väčšinou to nie je vhodné, lebo pri zmene veľkosti okna veľkosti prvkov sa neprispôsobia.

Nasledujúci program predstavuje príklad rozloženia prvkov v okne:

```
import javax.swing.*;
import java.awt.*;

class C {
  public static void main(String[] args) {
    JFrame w = new JFrame("Moje okno");
    w.setSize(300, 300);
    JButton b1 = new JButton("Tlačidlo 1");
    JButton b2 = new JButton("Tlačidlo 2");
//  prednastavené rozloženie:
//  w.setLayout(new BorderLayout());
    w.add(b1, BorderLayout.WEST);
```

```
        w.add(b2, BorderLayout.EAST);
        w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        w.setVisible(true);
        }
}
```

Tento príklad využíva `BorderLayout`, prednastavené rozloženie pre `JFrame`, ktoré rozdeľuje okno na päť oblastí: centrálnu a štyri strany sveta. Dve vytvorené tlačidlá sme umiestnili na západ a východ (pozri obr. 13.3).

Obrázok 13.3: Okno s dvomi tlačidlami.

13.5 Spracovanie udalostí vo Swingu

Komponenty Swingu generujú udalosti (events) vo forme objektov. Aby trieda mohla spracovávať udalosti (event handling), musí implementovať zodpovedajúce rozhranie prijímača (listener), napr.:

```
public class MyListener implements ActionListener {
   . . .
}
```

Trieda potom musí implementovať metódy rozhrania, napríklad:

```
public class MyListener implements ActionListener {
   . . .
   public void actionPerformed(ActionEvent e) {
      . . .
   }
   . . .
}
```

Objekt takejto triedy treba registrovať ako prijímač udalostí pre príslušný komponent. Predpokladajme, že chceme sledovať udalosť kliknutia tlačidla `tlacidlo1` pomocou prijímača typu `MyListener`, ktorý sme práve definovali:

```
tlacidlo1.addActionListener(new MyListener());
```

Rôznym typom udalostí zodpovedajú rôzne typy rozhraní prijímačov. Nasledujúci zoznam poskytuje niektoré typické udalosti a rozhrania, ktoré im zodpovedajú:

- kliknutie tlačidla, výber položky z menu, stlačenie tlačidla Enter pri zadávaní textu — `ActionListener`
- zavretie hlavného okna — `WindowListener`
- kliknutie tlačidla myši nad komponentom — `MouseListener`
- posun myšou nad komponentom — `MouseMotionListener`

Nasledujúci program predstavuje úplný príklad sledovania kliknutia na tlačidlo:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MyWindow extends JFrame {
   private JButton t = new JButton("Tlačidlo 1");
   public MyWindow() {
      setSize(300, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      add(t);
      t.addActionListener(new MyListener());
   }
```

```
    // trieda vhniezdená v MyWindow
    private class MyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (t.getText() != "XXX")
                t.setText("XXX");
            else
                t.setText("YYY");
        }
    }
}

class C {
    public static void main(String[] args) {
        JFrame w = new MyWindow();
        w.setVisible(true);
    }
}
```

Pri kliknutí na tlačidlo sa názov tlačidla bude meniť z `XXX` na `YYY` a opačne.

Prijímač udalostí je špecifická záležitosť daného komponentu. Preto je v predchádzajúcom kóde implementovaný pomocou vnútornej triedy s prístupom **private**. S výhodou ho môžeme implementovať pomocou anonymnej triedy:

```
class MyWindow extends JFrame {
    private JButton t = new JButton("Tlačidlo 1");
    public MyWindow() {
        setSize(300, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(t);
        t.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (t.getText() != "XXX")
                    t.setText("XXX");
                else
                    t.setText("YYY");
            }
        });
    }
}
```

Tento kód lepšie vyjadruje náš zámer — aj keď môže vyzerať zložitejšie — lebo jasne vyjadruje, že prijímač daného druhu je len jeden a ďalšie už nebudeme vytvárať. V

tomto zmysle nezavádza zbytočne názov ďalšej triedy.

Ak by sme potrebovali sledovať pohyb myšou nad tlačidlom, treba použiť triedu `MouseListner`:

```
t.addMouseListener(new MouseListener() {
   public void mouseEntered(MouseEvent e) {
      if (t.getText() != "XXX")
         t.setText("XXX");
      else
         t.setText("YYY");
   }

   public void mouseReleased(MouseEvent e) {}
   public void mousePressed(MouseEvent e) {}
   public void mouseExited(MouseEvent e) {}
   public void mouseClicked(MouseEvent e) {}
});
```

Niektoré rozhrania prijímačov predpisujú implementáciu viacerých metód pre rôzne udalosti. Často však sledujeme len jednu alebo dve udalosti, čo bol prípad aj v poslednom príklade. Dajú sa pritom využiť *adaptéry* (adapters) — triedy, ktoré implementujú všetky metódy daného rozhrania prijímača ako prázdne. Metódu pre príslušnú udalosť jednoducho prekonáme, ostatné necháme tak, ako sú.

Sledovanie pohybu myši nad tlačidlom sa pomocou príslušného adaptéra zjednoduší:

```
t.addMouseListener(new MouseAdapter() {
   public void mouseEntered(MouseEvent e) {
      if (t.getText() != "XXX")
         t.setText("XXX");
      else
         t.setText("YYY");
   }
});
```

13.6 Niť na odosielanie udalostí vo Swingu

Pre prácu so Swingom platí tzv. *pravidlo jednej nite* (Single-Thread Rule):[4]

[4] `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`

> Po realizácii komponentu Swingu všetok kód, ktorý by mohol vplývať na stav tohto komponentu alebo závisieť od neho, sa má vykonať v niti na odosielanie udalostí (Event-Dispatching Thread).

Pri porušení pravidla jednej nite hrozí narábanie s nekonzistentným stavom a uviaznutie. Problém sa nemusí prejaviť hneď a pri každom vykonávaní programu. Realizácia komponentu znamená, že bola zavolaná jeho metóda `paint()`. Pre okno to znamená zavolanie `setVisible(true)`, `show()` alebo `pack()`. Niektoré metódy je bezpečné volať mimo nite na odosielanie udalostí (opísané v dokumentácii Swing API). Práca so zoznamami prijímačov, o ktorej sme hovorili v časti 13.5, je bezpečná z hľadiska nite na odosielanie udalostí.

Ako dodržať pravidlo jednej nite? Kód, ktorý pracuje so stavom komponentov Swingu, treba vykonávať prostredníctvom nite na odosielanie udalostí Swingu. Taký kód treba zabaliť do vykonateľného objektu (`Runnable`) a zaradiť na vykonávanie prostredníctvom volania `invokeLater()` alebo `invokeAndWait()` v závislosti od toho, či chceme, aby sa metóda vrátila hneď po zaradení kódu alebo až keď niť na odosielanie udalostí vykoná kód.

Majme jednoduché okno:

```
class MyWindow extends JFrame {
   private JLabel l;

   public MyWindow() {
      setTitle("Moje okno");
      setSize(300, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      l = new JLabel("Text");
      add(l);
   }
   public void changeText(String s) {
      l.setText(s);
   }
}
```

Takto definované okno využíva nasledujúca trieda, ktorá z nejakých dôvodov potrebuje po zviditeľnení okna zmeniť text označenia v ňom (`JLabel`):

```
class C {
   public static void main(String[] args)
      throws Exception {
      final MyWindow w = new MyWindow();
      w.setVisible(true);
      . . .
```

```
      w.changeText("Novy text");
      . . .
    }
}
```

Priama zmena označenia predstavuje porušenie pravidla jednej nite. Zmena označenia musí byť realizovaná prostredníctvom nite na odosielanie udalostí:

```
class C {
   public static void main(String[] args)
      throws Exception {
      final MyWindow w = new MyWindow();
      w.setVisible(true);
      . . .
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            w.changeText("Novy text");
         }
      });
      . . .
   }
}
```

14 Objektovo-orientované modelovanie

Samotné programovanie je pri tvorbe softvéru kľúčové, ale celý proces tvorby softvéru zahŕňa rad ďalších aktivít, ktorých význam sa stáva zreteľnejším so zložitosťou vyvíjaného softvéru. Jednou z nich, ktorá bezprostredne súvisí s programovaním, je modelovanie softvéru. Modelovanie sa väčšinou chápe ako grafické vyjadrenie softvéru. V tejto kapitole sa budeme zaoberať základmi objektovo-orientovaného modelovania v jazyku UML, ktorý je de facto štandardom v modelovaní softvéru.

14.1 Modelovanie softvéru

Už pri riešení aj pomerne jednoduchých úloh ste si mohli všimnúť, že nie je možné pristúpiť hneď k písaniu programu. Platí to pre všetky paradigmy programovania, a objektovo-orientované programovanie nie je výnimkou. Pred samotným programovaním musíme najprv *analyzovať*, čo presne potrebujeme urobiť. Následne *navrhneme* samotnú realizáciu. Až potom môžeme pristúpiť k samotnému programovaniu, t.j. *implementácii* návrhu do daného programového prostredia daného programovacím jazykom s príslušnými rámcami a knižnicami a operačným systémom.

Tieto etapy vo vývoji softvéru — analýza, návrh a implementácia — nemusia prebehnúť formálne a nemusia byť ani nijako zaznamenané. Zvlášť sa to tak deje pri jednoduchých aplikáciách. V skutočnosti — okrem možno pri veľmi jednoduchých aplikáciách — nikdy neprebiehajú ako jednorazové etapy, ale v malých krokoch — iteráciách — z ktorých každý má všetky tri etapy v sebe. Takýto vývoj je teda iteratívny, a tiež inkrementálny, lebo sa aplikácia s každou iteráciou rozrastie o ďalší prírastok, t.j. inkrement.

Model softvéru sa často chápe ako rámec pre ďalší vývoj. Takýto model neobsahuje všetky detaily, t.j. predstavuje abstrakciu vyvíjaného softvéru. Kľúčová je pritom vizualizácia dôležitých častí softvéru, t.j. ich diagramatická reprezentácia.

Iné chápanie modelu predpokladá jeho úplnosť. Takýmto modelom — aj keď nie grafickým — je práve program.

Abstraktné modely softvéru vyjadrené graficky umožňujú ľahšie pochopenie a uvažovanie o kľúčových veciach v analýze a návrhu. Jestvujú však rôzne notácie, ale — ako uvidíme — aj snaha o ich zjednotenie a štandardizáciu.

V modelovaní softvéru sa sledujú dva rozmery: statický/dynamický a logický/fyzický [Boo94]. Daný pohľad v modeli sa môže orientovať viac na štruktúru, t.j. byť statický, alebo na vykonávanie programu, t.j. byť dynamický. Statický pohľad určuje štruktúru zdrojového textu, kým dynamický pohľad určuje správanie (vykonávanie) programu.

Pohľad v modeli môže vyjadrovať veci prevažne na logickej úrovni nezaťažené implementačnými detailmi alebo na fyzickej úrovni. Fyzická úroveň je blízka samotnej implementácii, kým logická úroveň je orientovaná na aplikačnú (problémovú) oblasť.

14.2 UML

Kľúčové pre skutočnú použiteľnosť hocijakej grafickej notácie je jej zrozumiteľnosť pre široký okruh vývojárov. Notácia musí byť akceptovaná zo strany vývojárov a štandardizovaná. Pre objektovo-orientované modelovanie vzniklo viac notácií, ktoré sa v mnohom prelínali, ale aj dopĺňali. Na ich základe vznikol UML — Unified Modeling Language — zjednotený jazyk modelovania, ktorý — ako vyplýva z jeho názvu — ašpiruje aj na iné oblasti ako je objektovo-orientované programovanie, a dokonca aj mimo hraníc vývoja softvéru ako takého, ako je napríklad biznis modelovanie (business modeling). UML sa stal de facto štandardom v modelovaní softvéru.

UML vyvinula organizácia OMG — Object Management Group — ktorej zakladateľom je Rational Software Corporation. Za touto iniciatívou stáli traja priekopníci objektovo-orientovaného modelovania: Grady Booch, Ivar Jacobson a Jim Rumbaugh. Aj keď je UML navrhnutý tak, aby sa dal rozširovať na rôzne účely, novovznikajúce potreby vo vývoji softvéru vyžadujú zmeny v základoch UML. UML sa teda naďalej vyvíja a jeho vývoj riadi OMG. OMG je neziskové konzorcium s otvoreným členstvom. Rozhodnutia v OMG sa prijímajú na základe hlasovania.

Podpora notácie modelovania nástrojom je esenciálna pre jej reálne použitie. Vo vývoji softvéru takáto podpora sa označuje ako Computer Aided Software Engineering a známa je pod skratkou CASE. Rozšíreným CASE nástrojom je IBM Rational Software Architect (Modeler), ktorý je založený na prostredí Eclipse. Tento nástroj je nasledovníkom populárneho nástroja Rational Rose. Ďalšie CASE nástroje pre UML sú napríklad OMONDO EclipseUML Studio, ktorý je tiež založený na platforme Eclipse, ArgoUML, Poseidon for UML atď.[1]

[1] Pozri `http://www.uml.org/#Links-Tools` pre prehľad CASE nástrojov pre UML.

CASE nástroj nie je len nástroj na kreslenie diagramov. Kým nástroje na kreslenie diagramov umožňujú väčšinou len kreslenie grafov — teda uzlov prepojených hranami — v pevne danej alebo rozšíriteľnej palete, CASE nástroje minimálne zohľadňujú pravidlá danej notácie a udržiavajú informácie o prepojenosti prvkov modelu. Dômyselnejšie nástroje poskytujú aj podporu samotného procesu vývoja podľa určitej metodológie. Často však postačuje nakreslenie niekoľkých diagramov a v tomto editore diagramov majú svoje miesto. Azda najznámejším editorom diagramov je MS Visio, ale jestvujú aj jednoduchšie, ale voľne šíriteľné editory, ako napríklad Dia[2] a UMLet.[3]

Až na triviálne prípady, pre človeka nie je možné naraz pozrieť celý model softvérového systému, presne ako je to aj so samotným programovým kódom. Model je preto organizovaný do diagramov, ktoré predstavujú relevantné pohľady na neho. Podľa typu prvkov, ktoré sa vyskytujú v nich, rozlišujú sa aj typy diagramov. Základné rozdelenie diagramov UML kopíruje dva fundamentálne rozmery vo vývoji softvéru, štruktúru a správanie, ktoré sú zvlášť zreteľné v objektovo-orientovanom programovaní (pozri časť 4.2). Diagramy štruktúry (structure diagrams) v UML znázorňujú balíky, triedy, objekty a fyzické rozloženie systému, kým diagramy správania (behavior diagrams) znázorňujú použitie systému, komunikáciu, interakciu a stavový model.

Pre názornosť, pozrime sa na zoznam diagramov štruktúry a diagramov správania v UML. Diagramy štruktúry sú:

- diagram tried — class diagram
- diagram objektov — object diagram
- diagram kompozitnej štruktúry — composite structure diagram
- diagram rozloženia — deployment diagram
- diagram komponentov — component diagram
- diagram balíkov — package diagram

Diagramy správania sú:

- diagram aktivít — activity diagram
- diagramy interakcie — interaction diagrams:
  - diagram sekvencií — sequence diagram
  - komunikačný diagram — communication diagram (predtým diagram spolupráce — collaboration diagram)

[2]pozri `http://www.gnome.org/projects/dia/`

[3]pozri `http://www.umlet.com/`

  - diagram prehľadu interakcií — interaction overview diagram
  - časový diagram — timing diagram

- diagram prípadov použitia — use case diagram
- stavový diagram — statechart diagram

Pozrieme sa bližšie na tri druhy diagramov, ktorých poznanie patrí už do základnej programátorskej gramotnosti: diagram tried, diagram prípadov použitia a diagram sekvencií. Aj keď sa — ak striktne oddeľujeme implementáciu od analýzy a návrhu — od programátorov neočakáva, že budú vytvárať tieto diagramy, mali by byť schopní ich aspoň čítať, lebo často predstavujú podklad pre kód.

14.3 Diagram tried

Diagram tried znázorňuje (predovšetkým) triedy a vzťahy medzi nimi. Môže obsahovať aj rozhrania, balíky a objekty.

Základné vzťahy medzi triedami sú:

- asociácia — všeobecný vzťah
- agregácia:
  - hodnotou (kompozícia)
  - referenciou
- generalizácia/špecializácia — zodpovedá dedeniu v objektovo-orientovanom programovaní
- závislosť

Než sa pozrieme na príklad diagramu tried, pripomeňme si príklad s grafickými útvarmi z kapitoly 4:

```
class Bod {
   private double x, y;
   public Bod(double x, double y) { . . . }
}

interface Kresleny {
   void nakresli();
   void nakresli(int farba);
```

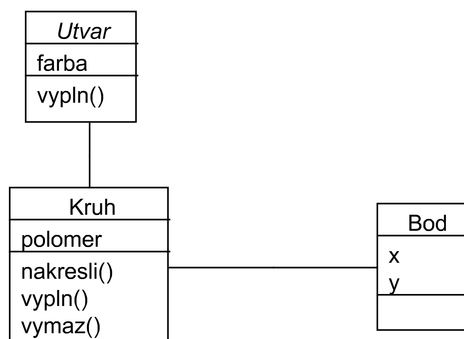
```
}

interface Rotovatelny {
   void rotuj(double uhol);
}

abstract class Utvar implements Kresleny, Rotovatelny {
   private int farba;
   public void vypln(int farba) { . . . }
   protected void setFarba(int farba) { . . . }
   protected int getFarba() { . . . }
}

class Kruh extends Utvar {
   private Bod c;
   private double r;
   Kruh(Bod c, double r) { . . . }
   public void nakresli() { . . . }
   public void nakresli(int farba) { . . . }
   public void vypln(int farba) { . . . }
   public void rotuj(double uhol) { . . . }
}
```

Príklad diagramu tried možno vidieť na obr. 14.1. Trieda je znázornená ako obdĺžnik (v základnej podobe) rozdelený na tri časti (zhora nadol): názov triedy, atribúty a operácie.[4]



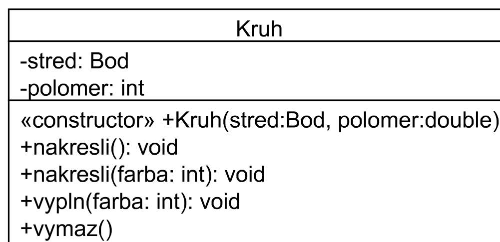
Obrázok 14.1: Všeobecná asociácia medzi triedami.

Všeobecná asociácia je zvlášť užitočná, keď začíname skúmať vzťahy medzi triedami, lebo vtedy ešte nevieme ich presný význam. Zabudnime na chvíľu, že už máme presný kód príkladu. Je možné, že sme v skorých fázach uvažovania o probléme

[4]V UML sa na označenie metódy používa termín operácia.

identifikovali triedy `Utvar`, `Kruh` a `Bod` a to, že medzi nimi by mal byť nejaký vzťah (pozri obr. 14.1).

Na začiatku nevieme veľa povedať ani o vnútorných detailoch tried. Tak atribúty a operácie tried uvedené na obr. 14.1 sú len pomenované. Ako vidíme na obr. 14.2, môžeme presne špecifikovať typ atribútov a signatúry operácií. Typ atribútu, parametra operácie a jej návratovej hodnoty sa uvádza za dvojbodkou za jeho identifikátorom.



Obrázok 14.2: Detaily triedy `Kruh`.

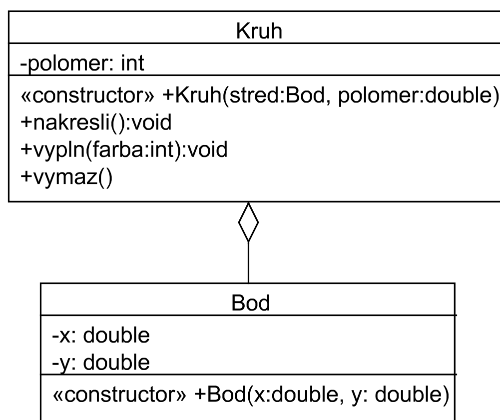
Konštruktor predstavuje špeciálnu operáciu. Označuje sa ako «constructor». Takéto označenia sa nazývajú stereotypy a umožňujú prisúdenie špeciálneho významu prvkom UML. Pre takéto prvky je možné definovať aj grafickú podobu odlišnú od prvkov, od ktorých sú odvodené. Stereotypy predstavujú jeden z mechanizmov rozšírenia UML, ktorých vysvetlenie je mimo rozsahu tohto krátkeho opisu.

Viditeľnosti atribútov a operácií v UML zodpovedá modifikátorom prístupu v Jave. Nasledujúci zoznam uvádza prehľad symbolov a ich význam v Jave:

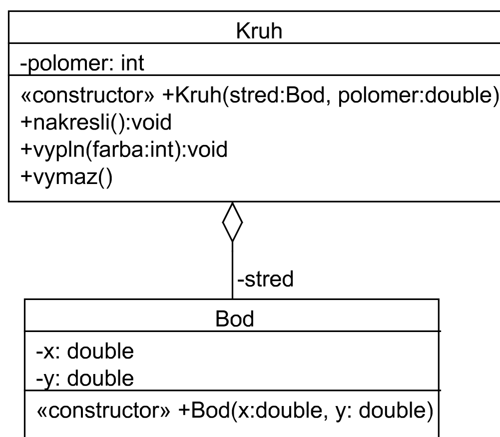
- + — public
- # — protected
- − — private
- ~ — package

Jedným z atribútov triedy `Kruh` je `stred`. Toto môžeme vyjadriť aj graficky použitím vzťahu agregácie ako je to znázornené na obr. 14.3. Pritom môžeme explicitne vyjadriť typ agregácie. Možno povedať, že hrana s plným kosoštvorcom označuje agregáciu hodnotou, ktorá sa v UML označuje ako kompozícia (pozri časť 4.1), kým hrana s prázdnym kosoštvorcom — ako na obr. 14.3 označuje agregáciu referenciou. Presný význam je však taký, že plný kosoštvorec označuje agregáciu, pri ktorej agregujúci objekt nesie zodpovednosť za existenciu a uloženie agregovaných objektov (composite), kým prázdny kosoštvorec znamená, že ide o zdieľanú agregáciu, pri ktorej aj iné objekty môžu agregovať rovnaký objekt (shared).

Inštancia triedy `Bod` v inštancii triedy `Kruh` hrá rolu stredu. Toto sa dá vyjadriť uvedením tzv. asociačnej roly (association role) — roly triedy vo vzťahu k inej triede — ako ukazuje obr. 14.4.



Obrázok 14.3: Agregácia.



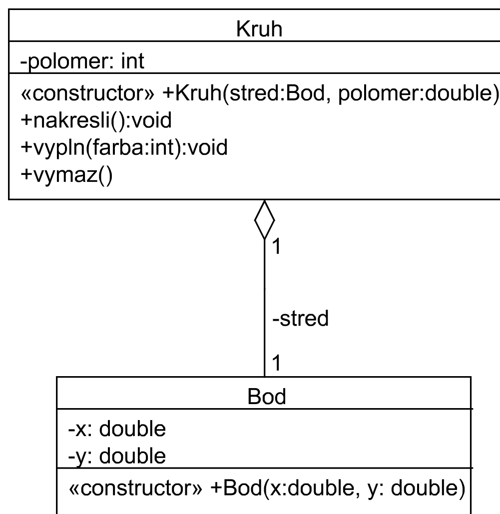
Obrázok 14.4: Asociačná rola.

Koľko inštancií triedy pripadá na výskyt inštancie inej triedy sa vyjadruje násobnosťou (*multiplicity*) vzťahu.[5] Vieme, že kruh má práve jeden stred. Navyše, obvykle — a to môže byť práve náš prípad — ten stred patrí len jednému kruhu, čo je znázornené na obr. 14.5.

Číslo pri danej triede hovorí aký počet inštancií tejto triedy sa vyskytuje na jednu inštanciu inej triedy. Dve bodky (..) vyjadrujú rozsah a hviezdička maximálnu násobnosť. Pozrime niekoľko ďalších príkladov násobnosti z obr. 14.6 na ozrejmenie notácie (zhora nadol):

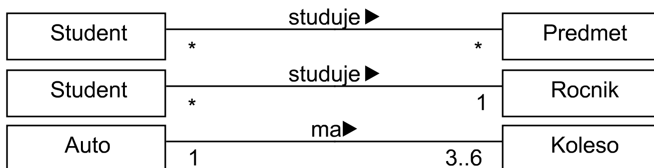
- Študent študuje viac predmetov, a ten istý predmet študujú viacerí študenti.

[5]Násobnosť sa často označuje ako kardinalita.



Obrázok 14.5: Násobnosť vzťahu.

- Študent študuje práve v jednom ročníku, a v tom istom ročníku študujú viacerí študenti.
- Auto má tri až šesť kolies, a koleso patrí len jednému autu.

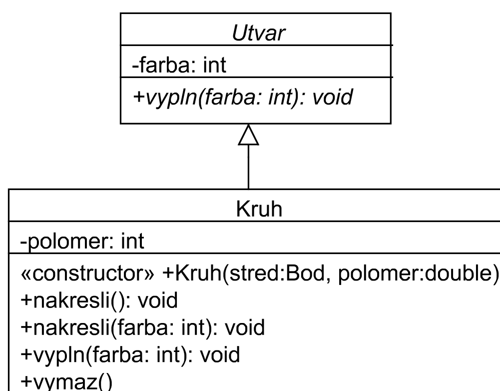


Obrázok 14.6: Príklady násobnosti vzťahu.

Dedenie sa v UML vyjadruje vzťahom generalizácie/špecializácie. Môžeme povedať, že trieda `Kruh` je špecializáciou triedy `Utvar` a graficky tento vzťah vyjadríme ako na obr. 14.7. Vidíme, že sa generalizácia (zovšeobecnenie) znázorňuje hranou ukončenou prázdnou šípkou. Šípka smeruje od špeciálnejšej k všeobecnejšej triede.

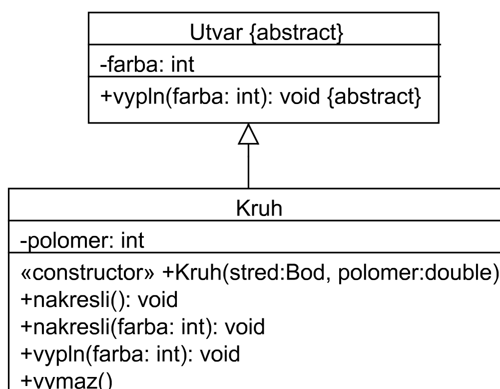
Hovorí sa, že triedy pri dedení tvoria hierarchiu (pozri časť 6.5). Koreňom hierarchie je najvšeobecnejšia trieda a hierarchia predstavuje vetvenie k najšpeciálnejším triedam. Šípka generalizácie teda smeruje od triedy, ktorá je v hierarchii nižšie, k triede, ktorá je v hierarchii vyššie. Agregácia je ďalší spôsob tvorenia hierarchie v objektovo-orientovanom programovaní (pozri opäť časť 6.5). Všimnime si, že šípka agregácie smeruje tiež od nižšie postavenej, agregovanej triedy k vyššie postavenej, agregujúcej triede.

Trieda `Utvar` je abstraktná, preto je jej názov na obr. 14.7 uvedený kurzívou. Rovnako



Obrázok 14.7: Generalizácia.

sa označujú aj abstraktné operácie. Pre zvýšenie čitateľnosti možno použiť označenie `{abstract}`, ako je ukázané na obr. 14.8.

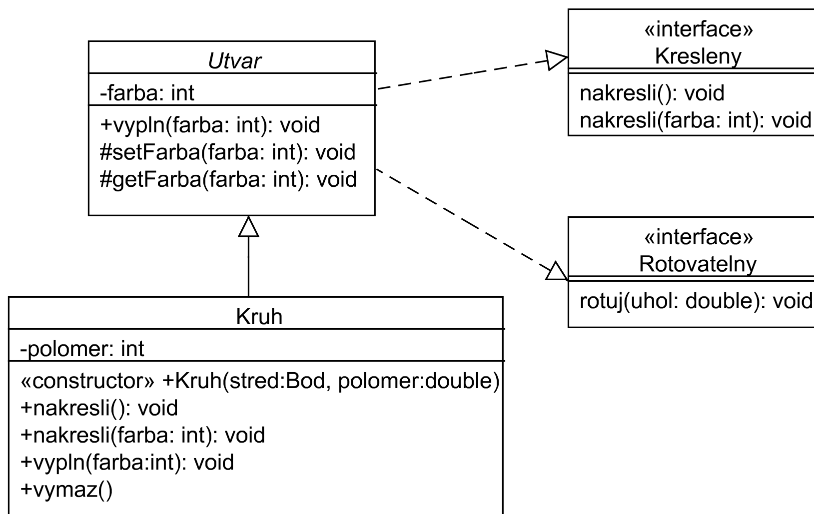


Obrázok 14.8: Abstraktné triedy a operácie.

Rozhrania sa v UML znázorňujú ako triedy so stereotypom «interface». Na obr. 14.9 vidíme triedu `Utvar`, ktorá realizuje (implementuje) rozhranie `Kresleny`. V zmysle objektovo-orientovaného programovania tento vzťah tiež predstavuje formu dedenia — a to dedenie správania. Na rozdiel od Javy, rozhrania v UML nemôžu mať atribúty. Obr. 14.10 zobrazuje prvky, ktoré sme doteraz identifikovali.

Použitie rozhrania predstavuje vzťah závislosti. Táto špeciálna závislosť sa označuje stereotypom «use». Vzťah závislosti vyjadruje hocijakú závislosť. Závislý je prvok, z ktorého smeruje šípka. Tento prvok sa označuje ako klient (client). Zmeny klienta nemajú vplyv na prvok, ku ktorému šípka smeruje, a ktorý sa označuje ako poskytovateľ (supplier).

Diagram tried nemusí obsahovať všetky detaily a v jednom diagrame nemusia byť všetky triedy. Nemusíme znázorniť všetky vzťahy medzi znázornenými triedami.



Obrázok 14.9: Realizácia rozhrania.

Každý diagram má niesť istý základný odkaz, podľa čoho vyberáme aj vzťahy, ktoré v diagrame zobrazíme. Niekedy chceme vyjadriť len hierarchiu dedenia, ako to ukazuje obr. 14.12. Možno pritom použiť aj združenú šípku ako na obr. 14.13.

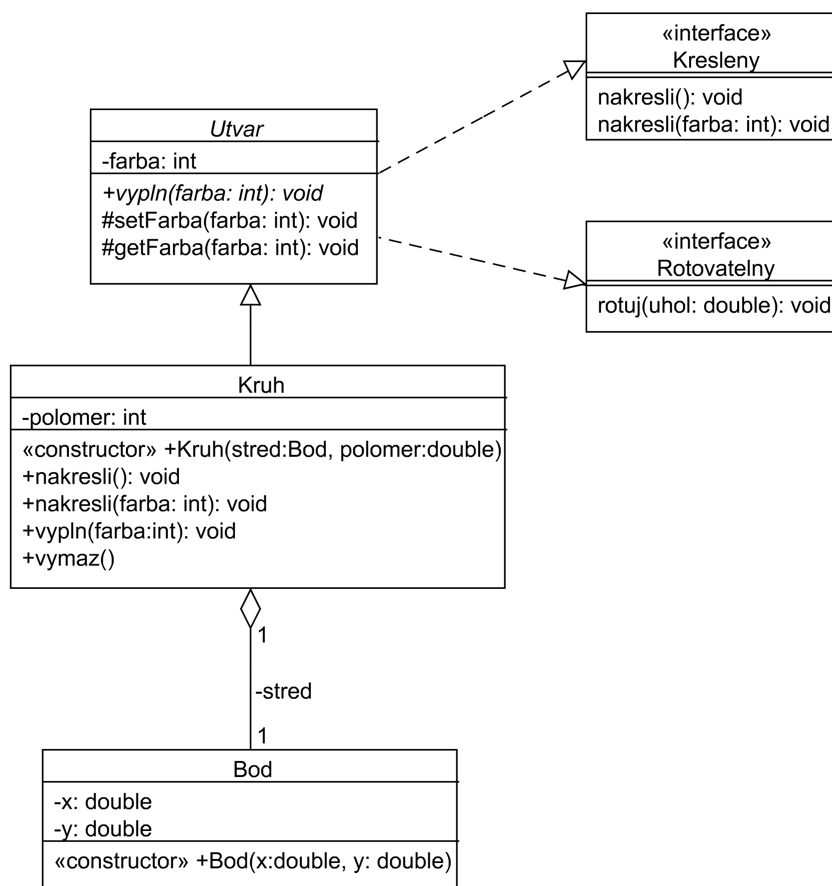
UML umožňuje tiež vyjadriť vhniezdenie tried (alebo iných typov). Príklad je znázornený na obr. 14.14.

Diagram tried znázorňuje potenciálne vzťahy medzi inštanciami tried. Niekedy je potrebné znázorniť realizované vzťahy medzi špecifickými inštanciami tried v určitom okamihu vykonávania programu. Na to slúži diagram objektov, ktorý predstavuje ďalší štrukturálny pohľad. Možno ho chápať ako špeciálny prípad diagramu tried.

Zoberme napríklad už spomínaný príklad vzťahu auto–koleso. Na obr. 14.15 je tento vzťah zopakovaný. Špecifické auto bude mať určitý počet kolies v danom rozpätí 3–6. Jedno z možných áut je znázornené v spodnej časti obrázku.

14.4 Diagram prípadov použitia

Základom pre ďalší vývoj softvérového systému je opis jeho funkcionality. Zistilo sa, že je aj z pohľadu vývojára, aj z pohľadu zákazníka najvhodnejšie funkcionalitu analyzovať v menších celkoch, ktoré predstavujú zmysluplné činnosti. Medzi týmito činnosťami, ktoré sa nazývajú prípady použitia, identifikujú sa určité vzťahy, ale v žiadnom prípade nejde o hierarchickú dekompozíciu funkcionality. Pre každý prípad použitia sa identifikujú jeho účastníci (actors), ktorí môžu predstavovať roly použí-

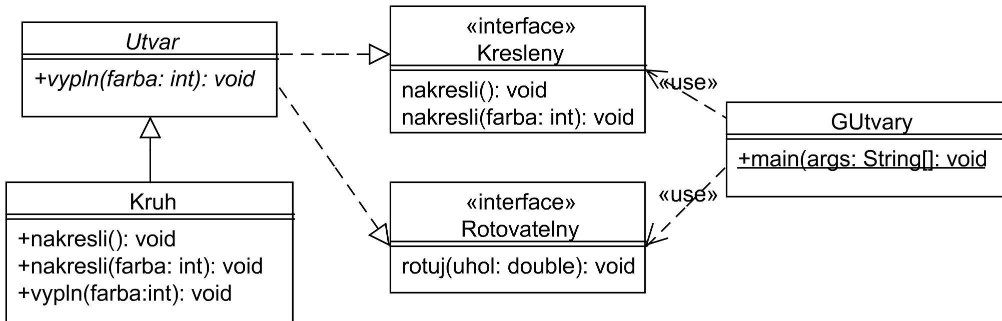


Obrázok 14.10: Detailný diagram tried.

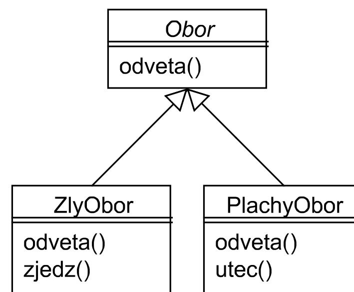
vateľov systému alebo aj iné softvérové systémy a podsystémy vyvíjaného systému.

Prípady použitia predstavujú techniku zachytenia funkcionality systému. Najdôležitejší je pritom ich opis, ktorý môže byť viac alebo menej formálny a viac alebo menej detailný. Často je opis prípadu použitia prezentovaný v tvare postupnosti očíslovaných krokov, ktoré vyjadrujú akcie používateľa a systému. Samotnej postupnosti krokov zvyčajne predchádza krátke vysvetlenie podstaty prípadu použitia. Pre zvýšenie zrozumiteľnosti prípadu použitia možné odchýlky od základnej postupnosti krokov — tzv. základného toku (basic flow) — sa uvádzajú zvlášť ako alternatívne toky (alternate flow).

Opis prípadov použitia najčastejšie sprevádza diagram prípadov použitia. Diagram prípadov použitia znázorňuje súvisiace prípady použitia a príslušných účastníkov. Úroveň podrobnosti prípadov použitia závisí od ich účelu, ktorý sa pohybuje od komunikácie so zákazníkom po detailný návrh funkcionality najčastejšie prostredníctvom diagramov sekvencií, ktorými sa budeme zaoberať v nasledujúcej časti.



Obrázok 14.11: Použitie rozhrania a väzba závislosti.



Obrázok 14.12: Hierarchia obrov.

V diagrame prípadov použitia väzba medzi účastníkom a prípadom použitia znamená, že účastník sa zúčastňuje prípadu použitia. Prípady použitia môžu špecializovať, zahŕňať (stereotyp «include») alebo rozširovať (stereotyp «extend») iné prípady použitia. Účastník môže byť špecializáciou iného účastníka.

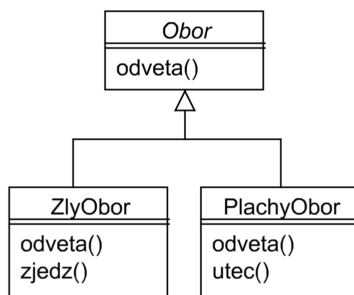
Tieto vzťahy si ozrejmíme na príklade diagramu prípadov použitia z obr. 14.15. Predpokladajme, že navrhujeme informačný systém univerzity, ktorého súčasťou je aj voľba témy bakalárskeho projektu. Študent si predregistruje témy (prípad použitia Predregistrácia témy), o ktoré má záujem, na základe čoho príslušný učiteľ organizuje konzultácie. Predregistrované témy študent môže meniť. Po dohode so študentom, učiteľ registruje výber témy (prípad použitia Registrácia témy). Systém študentovi elektronickou poštou odošle potvrdenie o registrácii témy.

Prípady použitia sme opísali pomerne neformálne. Ak zoberieme ako príklad prípad použitia Predregistrácia témy, príslušná postupnosť krokov by mohla byť nasledujúca:

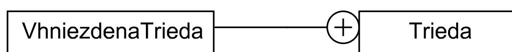
1. Používateľ si zvolí predregistráciu témy.

2. Systém zobrazí evidované témy.

3. Používateľ si vyberie jednu tému.



Obrázok 14.13: Združená šípka pre generalizáciu.



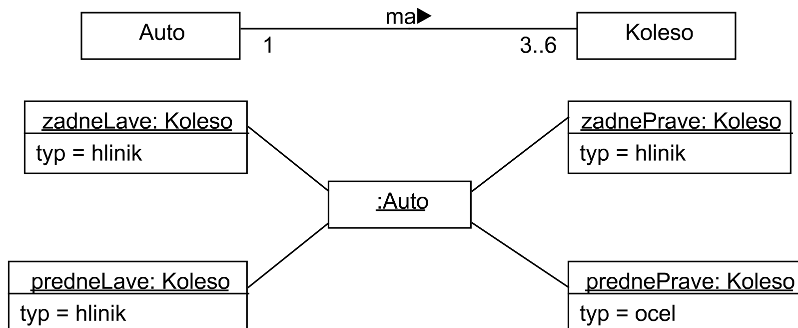
Obrázok 14.14: Vhniezdená trieda.

4. Ak používateľ potvrdí výber témy, systém ju zaradí medzi jeho predregistrované témy.

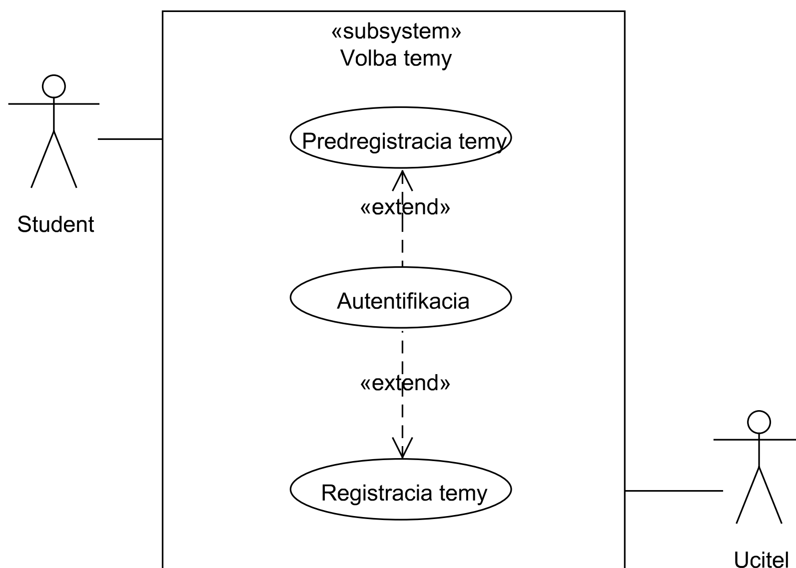
5. Prípad použitia končí.

Všimnime si, že opis prípadu použitia zámerne neobsahuje detaily používateľského rozhrania. Definuje však požiadavky na používateľské rozhranie vzhľadom na funkcionalitu. V uvedenom príklade vidíme, že používateľ musí mať možnosť najskôr si vybrať tému, a až potom potvrdiť jej výber. Používateľské rozhranie musí teda umožniť označenie témy v zozname tém a musí obsahovať nejaký ovládací prvok — zrejme tlačidlo — aktiváciou ktorého používateľ potvrdí výber označenej témy.

Aj predregistrácia témy, aj jej registrácia vyžadujú autentifikáciu používateľa (prípad použitia Autentifikácia), ktorá vlastne rozširuje základný tok. Závislosť «extend» znamená, že prípad použitia, z ktorého hrana smeruje, rozširuje prípad použitia, ku ktorému hrana smeruje, v tzv. bodoch rozšírenia. Body rozšírenia môžu byť



Obrázok 14.15: Diagram objektov.



Obrázok 14.16: Príklad diagramu prípadov použitia.

určené číslami príslušných krokov základného toku alebo definované opisne. Základná funkcionalita rozšíreného prípadu použitia je zachovaná aj bez rozšírenia. V závislosti «extend» rozširujúci prípad použitia má rovnakú sémantiku ako alternatívny tok rozširovaného prípadu použitia.

Pri závislosti «include» prípad použitia zahŕňa iný prípad použitia. Môže to byť funkcionalita, ktorú zdieľajú viaceré prípady použitia. Zahrnutá funkcionalita je nevyhnutná pre prípady použitia, ktoré ju zahŕňajú, čo je rozhodujúci rozdiel oproti rozširujúcej funkcionalite.

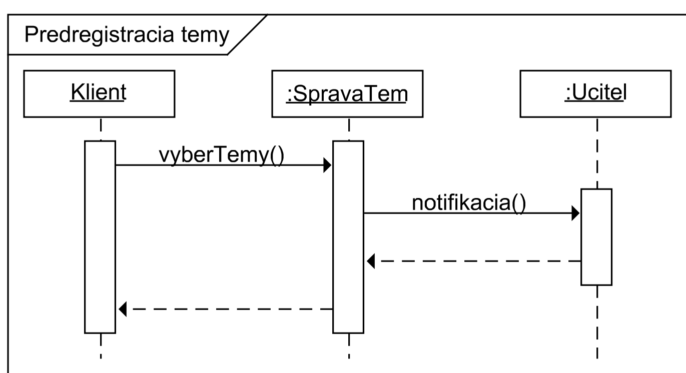
14.5 Diagram sekvencií

Prípady použitia sa opisujú na rôznych úrovniach podrobností. Slovný opis často sprevádza diagram sekvencií alebo diagram aktivít. Diagram sekvencií (sequence diagram) znázorňuje postupnosť správ prenášaných medzi objektmi. Čiarkované šípky vyjadrujú návrat po ukončení príslušnej operácie. Objekty sú usporiadané horizontálne. Čas je vyjadrený pomyselnou vertikálnou osou. Z objektov vychádzajú zvysle prerušované čiary, ktoré sa označujú ako čiary života (lifelines), a označujú časové pásma existencie objektov.

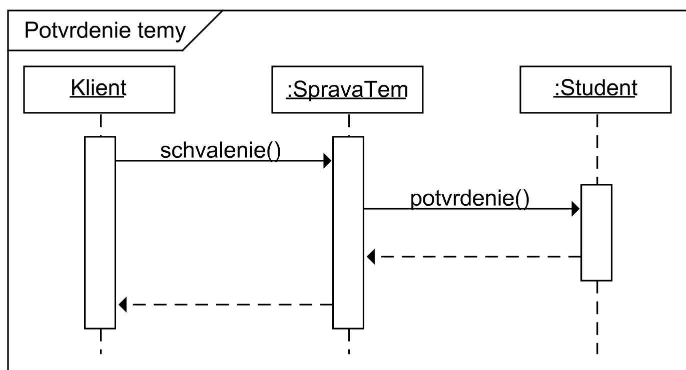
Správu môže poslať len objekt, ktorý v danom čase má riadenie, čo je vyjadrené obdĺžnikom na jeho čiare života. Posielanie správ väčšinou má význam vyvolania

operácie objektu, ku ktorému šípka smeruje. Nad šípkou je vyznačený názov operácie (niekedy aj jej parametre).

Diagramy sekvencií možno využiť na špecifikáciu tokov[6] v prípadoch použitia. Na obr. 14.17 a 14.18 sú bližšie špecifikované predregistrácia témy a potvrdenie témy.



Obrázok 14.17: Predregistrácia témy.



Obrázok 14.18: Potvrdenie témy.

V takýchto diagramoch sekvencií interakciu typicky iniciuje objekt `Klient` (`Client`) uvádzaný bez typu, ktorý predstavuje účastníka v diagrame prípadov použitia (typ účastníka je daný diagramom prípadov použitia). Ďalej v nich vystupujú objekty, ktoré predstavujú časti systému identifikované analýzou prípadov použitia. Tieto objekty sú najčastejšie nepomenované, lebo sa nepotrebujeme na ne v diagrame odvolávať. Diagramy sekvencií môžu obsahovať aj slučky a podmienené príkazy. Podľa diagramov sekvencií už možno vytvárať samotný kód.

[6]označované aj ako scenár

15 Návrhové vzory

Znovupoužitie softvéru bolo dlho chápané ako opakované využitie jestvujúcich častí programov. Už v procedurálnom programovaní sa dosahoval určitý stupeň znovupoužitia prostredníctvom knižníc procedúr. Verilo sa, že objektovo-orientované programovanie prinesie revolučný pokrok v tomto smere a že programy sa pomerne ľahko postavia z už jestvujúcich tried. Ukázalo sa, že trieda obsahuje často príliš špecifický kód a že ako jednotka znovupoužitia je príliš malá, aby mohla tvoriť uzavretú funkcionalitu pripravenú na skladanie. Tento problém riešia rôzne komponentové modely, pričom však stále zápasia s problémom prispôsobenia a konfigurovateľnosti.

V tejto kapitole uvidíme, že znovupoužitie softvéru možno hľadať aj na inej úrovni než na priamom opakovanom využití častí kódu. Zložitosť riešenia tkvie v návrhu. Jeden *vzor* návrhu možno uplatniť v pomerne rozdielnych doménach aplikácie.

15.1 Vzory vo vývoji softvéru

Medzi (stavebnou) architektúrou a vývojom — budovaním — softvéru sa často hľadajú paralely, aby sa do mladej disciplíny vývoja softvéru mohli preniesť aspoň niektoré zo skúseností v oblasti architektúry, ktoré sa hromadili tisíce rokov.

Najvýznamnejší vplyv architektúry na vývoj softvéru malo zistenie, že sa aj v ňom — podobne ako v architektúre — dajú hľadať tzv. vzory. Architektúru založenú na vzoroch formuloval inventívny architekt Christopher Alexander, ktorý poukázal, že krásne a užitočné v architektúre je spojené a vždy predstavuje prejav určitého vzoru. Prejavy jedného vzoru môžu byť veľmi odlišné, lebo závisia od kontextu, v ktorom sa vyskytujú, ale Alexandrovi sa podarilo vyčleniť esenciu mnohých vzorov, pri ktorej aplikácii sa zachovávajú kvality pôvodných výskytov daného vzoru.

Vzor teda predstavuje *riešenie opakujúceho sa problému v kontexte.* Alexander definoval svoje architektonické návrhy v tvare postupnosti uplatnenia jednotlivých vzorov, ktoré tvorili jeden celok: jazyk vzorov.

Hlas o Alexandrových vzoroch sa rozniesol aj mimo architektúry. Táto idea zaujala niektorých prominentov v oblasti vývoja softvéru, ktorí v roku 1993 vytvorili skupinu známu ako Hillside Group: K. Auer, G. Booch, R. Johnson, H. Hilerbrand, K. Beck,

W. Cunnigham a J. Coplien.[1] Alexander sa stal prvým architektom pozvaným, aby prehovoril k informatikom, a zdá sa, že jeho práca mala väčší vplyv na informatiku než na architektúru. Fenomén vzorov vo vývoji softvéru možno vnímať dokonca aj ako špecifickú kultúru — kultúru vzorov [Cop05].

Vzory vo vývoji softvéru možno vnímať na rôznych úrovniach granularity. *Návrhové vzory* (design patterns) sú vzory platné pre viaceré príbuzné jazyky. Mnohé vzory sú platné pre celú spleť objektovo-orientovaných jazykov.

Idiómy sú na nižšej úrovni granularity, t.j. predstavujú „menšie" návrhové vzory. Toto sa netýka toľko rozsahu ako dosahu: podobne ako je to s idiómami v prirodzených jazykoch, daný idióm platí len pre určitý programovací jazyk alebo triedu veľmi príbuzných programovacích jazykov. Na rozdiel od idiómov v prirodzených jazykoch — slovných spojení, ktorých význam sa najčastejšie nedá odvodiť z významu jednotlivých slov, ktoré ich tvoria, význam idiómov v programovacích jazykoch sa vždy dá odvodiť, čo je dané formálnou povahou programovacích jazykov. Tieto konštrukcie však vnímame a používame ako idiómy v tom zmysle, že programátor, ktorý sa s nimi nikdy nestretol, ich ťažko odvodí.

Architektonické vzory sú na vyššej úrovni granularity než návrhové vzory a často ich možno rozložiť na návrhové vzory, ktoré ich tvoria. S týmito vzormi sa operuje na najvyššej úrovni návrhu softvéru.

Jestvujú ešte aj *analytické vzory*, ktoré v istom zmysle stoja nad architektonickými vzormi. Tieto vzory sú nezávislé od domény riešenia (implementácie) a týkajú sa predovšetkým modelov údajov a riešení, ktoré sa v týchto modeloch opakovane vyskytujú.

Bolo objavených a publikovaných veľa vzorov. Vzorom sa venujú mnohé konferencie, z ktorých je najznámejšia PLoP.[2] Vzory bývajú sústredené do katalógov. Prvým katalógom návrhových vzorov boli tzv. GoF[3] vzory [GHJV95], a z hľadiska Javy sú významné napr. vzory J2EE. V nasledujúcich častiach sa pozrieme bližšie na populárny architektonický vzor Model-View-Controller a dva návrhové vzory, ktoré sú pre tento architektonický vzor kľúčové — Visitor a Observer. Obidva tieto vzory patria medzi tzv. GoF vzory. V rámci návrhového vzoru Visitor sa pozrieme na dvojitý polymorfizmus, ktorý sa prezentuje aj samostatne ako idióm double dispatch.

[1] pozri `http://www.hillside.net/`

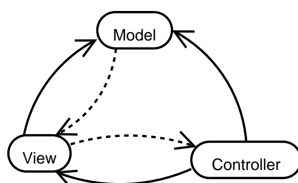
[2] pozri `http://st-www.cs.uiuc.edu/~plop/`

[3] z Gang of Four, prezývky autorov katalógu

15.2 Architektonický vzor Model-View-Controller

Architektonický vzor Model-View-Controller (MVC) prvý identifikoval Trygve M. H. Reenskaug v roku 1979. Tento vzor svoje prvé použitie mal v Smalltalku a dnes predstavuje základ pre grafické používateľské rozhrania (GUI). Napríklad aj Swing je implementovaný podľa vzoru MVC.

Schéma vzoru MVC je znázornená na obr. 15.1. Model predstavuje model spracovávanej oblasti, t.j. aplikačnú logiku.



Obrázok 15.1: Model-View-Controller (podľa [Hel]).

Pohľad (View) je pohľad na model. Pohľad zobrazuje časti modelu relevantné pre prácu s ním. V závislosti od účelu, pohľady sa môžu líšiť. Pohľady často obsahujú aj aktívne prvky, ktorými možno priamo ovplyvniť model. Model je nezávislý od detailov svojich pohľadov. Nanajvýš má ich zoznam a posiela im správy o potrebe občerstvenia ich obsahu v dôsledku zmien v modeli.

Kontroler (Controller) predstavuje riadenie modelu a pohľadu, t.j. zabezpečenie reakcie aplikácie na udalosti. Kontroler posiela správy o udalostiach pohľadu a modelu. V idealizovanej podobe vzoru MVC je to jediný spôsob zmeny modelu a v tom prípade vzor MVC sa javí ako vrstvový. V skutočnosti pohľad často priamo zasahuje do modelu, a mnohé udalosti — napríklad pohyb myšou nad určitým prvkom — bývajú generované pohľadom.

Vzor MVC možno s výhodou použiť v hocijakej aplikácii, ktorá predpokladá nejaký druh používateľského rozhrania. Pritom treba minimalizovať zviazanosť (pozri časť 6.4) medzi modelom, pohľadom a riadením.[4] Nie vždy je vhodné úplné oddelenie týchto častí. Pre menšie aplikácie môže byť vhodné spojenie pohľadu a riadenia do jedného celku. Tento prístup je známy ako vzor Presentation-Model.[5]

[4] pozri `http://www.leepoint.net/notes-java/GUI/structure/40mvc.html`

[5] pozri `http://www.leepoint.net/notes-java/GUI/structure/30presentation-model.html`

15.3 GoF vzory

V roku 1995 Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides, ktorí sa neskôr stali známi ako Gang of Four (GoF), uverejnili prvý katalóg návrhových vzorov[GHJV95].[6] Treba však spomenúť, že už v roku 1992 vyšla priekopnícka práca Jamesa O. Copliena o idiómoch jazyka C++.

Katalóg GoF obsahuje 23 návrhových vzorov rozdelených podľa účelu na tvorebné (creational), štrukturálne a behaviorálne. Iná kategorizácia je podľa rozsahu na vzory, ktoré sa vzťahujú na triedy a vzory, ktoré sa vzťahujú na objekty. Katalóg GoF definuje aj vzťahy medzi vzormi. Nie je to však jazyk vzorov v zmysle ako ho definoval Christopher Alexander, ale schéma niektorých vhodných kombinácií vzorov.

Štruktúra opisu vzoru v katalógu GoF je inšpirovaná štýlom Christophera Alexandra a pozostáva z nasledujúcich častí:

- Názov vzoru a klasifikácia
- Zámer
- Iné názvy
- Motivácia
- Použiteľnosť
- Štruktúra
- Účastníci
- Spolupráce
- Implementácia
- Vzorka kódu
- Známe použitia
- Príbuzné vzory

Opis vzorov Visitor a Observer, ktorý nasleduje, nebude podaný v štruktúre podľa katalógu GoF, ale v kompaktnejšom tvare.

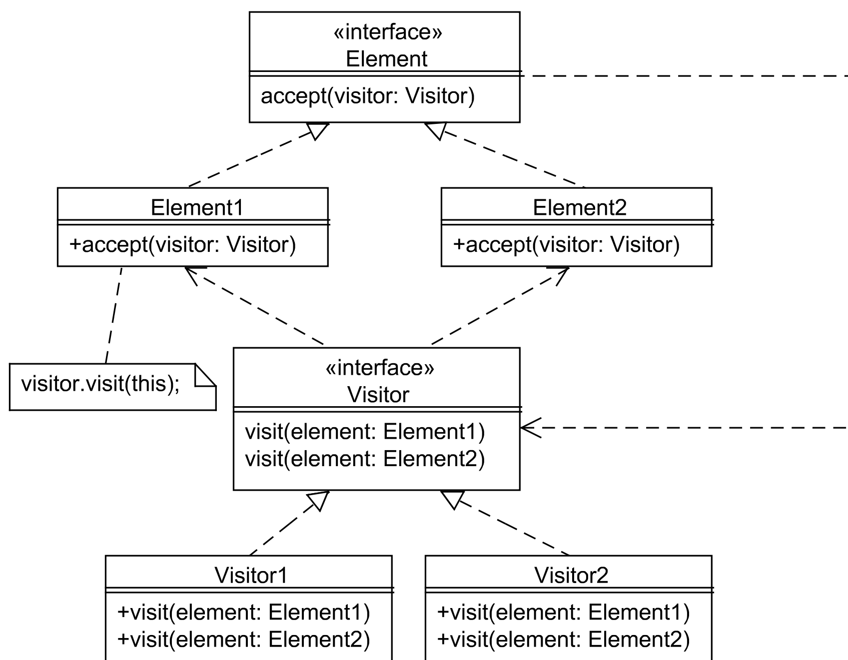
[6] Pekný prehľad GoF vzorov je dostupný napríklad na `http://www.dofactory.com/Patterns/Patterns.aspx`.

15.4 Návrhový vzor Visitor

Návrhový vzor Visitor (návštevník) umožňuje pridávanie operácií nad objektmi rôznych druhov bez toho, aby bolo potrebné meniť ich triedy. Ide teda o behaviorálny vzor, ktorý sa vzťahuje na objekty.

Pomocou vzoru Visitor možno vytvoriť spoločné operácie pre typy, ktoré nie sú vo vzťahu dedenia. Vzor Visitor je vhodný, ak tieto operácie nesúvisia priamo s triedami, nad ktorými operujú. Zabraňuje sa tak „znečisteniu kódu“, ktoré by vzniklo pridaním týchto operácií priamo do zodpovedajúcich tried. Navyše sa dosiahne to, že operácie, ktoré súvisia, budú sústredené v jednej triede. Vzor Visitor je zvlášť vhodný, ak je operácie potrebné často pridávať.

Štruktúra vzoru Visitor je znázornená na obr. 15.2. Prvky, do ktorých je potrebné pridať ďalšie operácie, implementujú rozhranie označené ako `Element`, ktoré predpisuje operáciu `accept`(). Parametrom tejto operácie je príslušný objekt typu `Visitor`, ktorý daný objekt typu Element vyvolaním tejto operácie akceptuje. `Visitor` pritom obsahuje operáciu `visit`() pre každý možný podtyp typu `Element`. Operácií `accept`() a príslušných operácií `visit`() pritom môže byť viac.



Obrázok 15.2: Štruktúra vzoru Visitor.

`Visitor` je len rozhranie, ktoré môže mať viac implementácií. Na obrázku sú naznačené dve: `Visitor1` a `Visitor2`.

Ku grafickému znázorneniu štruktúry vzorov treba poznamenať, že vždy ide len o jeden pohľad na vzor. Ide skôr o príklad štruktúry ako o absolútne platnú štruktúru[7] a podobne je to aj s implementáciou vzoru.

Skúsme sa pozrieť na vzor Visitor na príklade. Predpokladajme, že sa v aplikácii, ktorú vyvíjame, pracuje s položkami, ktoré treba zobrazovať na rôznych zariadeniach. Typy položiek sú rôzne. Zatiaľ sú to slová a zoznamy, ale môžeme očakávať, že pribudnú ďalšie typy. Položky sa zobrazujú na rôznych typoch zariadení ako je napríklad horizontálny zobrazovač a vertikálny zobrazovač, ale tiež možno očakávať ďalšie typy zobrazovacích zariadení.

Položky modelujeme triedami, ktoré obsahujú aplikačnú logiku, ktorá sa ich bezprostredne týka. Je žiaduce, aby sme mohli každej položke povedať jednoducho „zobraz sa“, ale nechceme sa v triedach položiek zaoberať samotným zobrazením. Všetky typy položiek budú teda implementovať rozhranie s operáciou zobrazenia:

```
interface DItem { // rozhranie Element
   void display(DDevice d); // prijatie návštevníka
}
```

Trieda WordItem implementuje slovnú položku:

```
class WordItem implements DItem {
   String word;
   public WordItem(String s) {
      word = s;
   }
   public void display(DDevice d) {
      d.write(this);
   }
}
```

Samotná slovná položka je vnútorne reprezentovaná ako reťazec znakov. Operácia zobrazenia prenecháva skutočné zobrazenie samotnému zariadeniu. Položka teda nepozná detaily zariadenia a tak je od neho nezávislé, avšak zariadenie závisí od položky, lebo — ako uvidíme — musí vedieť ako zobraziť položku.

Podobne je to s položkou v tvare zoznamu, ktorej vnútorná reprezentácia je v tvare objektu typu `List`:

```
class ListItem implements DItem {
   List list;
   public ListItem(List l) {
```

[7] pozri `http://www.industriallogic.com/pulse/20001001.html`

```
        list = l;
    }
    public void display(DDevice d) {
        d.write(this);
    }
}
```

Každé zobrazovacie zariadenie musí implementovať zobrazenie každého typu položky:

```
interface DDevice { // rozhranie Visitor
    void write(WordItem item);
    void write(ListItem item);
}
```

Nasledujúci kód predstavuje implementáciu horizontálneho zobrazovača:

```
class HorDevice implements DDevice {
    // navštívenie WordItem
    public void write(WordItem item) {
        for(int i = 0; i < item.word.length(); i++)
            System.out.print(item.word.charAt(i));
    }
    // navštívenie ListItem
    public void write(ListItem item) {
        for(int i = 0; i < item.list.size(); i++) {
            System.out.print(item.list.get(i));
            System.out.print(", ");
        }
    }
}
```

Horizontálny zobrazovač zobrazuje slovnú položku ako postupnosť jednotlivých písmen, pričom pre názornosť operuje nad jej vnútornou reprezentáciou vyberajúc písmeno za písmenom. Položku v tvare zoznamu zobrazuje podobne, ale jednotlivé písmená oddeľuje čiarkou.

Podobne je implementovaný aj vertikálny zobrazovač s tým, že písmená vypisuje jedno pod druhým:

```
class VerDevice implements DDevice {
    // navštívenie WordItem
    public void write(WordItem item) {
        for(int i = 0; i < item.word.length(); i++)
```

```
            System.out.println(item.word.charAt(i));
        }
        // navštívenie ListItem
        public void write(ListItem item) {
            for(int i = 0; i < item.list.size(); i++) {
                System.out.print(item.list.get(i));
                System.out.println(", ");
            }
        }
    }
```

Pre úplnosť uvedieme aj príklad použitia zobrazovačov a položiek:

```
    class M {
      public static void main(String[] args) {
        List list = new ArrayList();
        list.add("a");
        list.add("b");
        list.add("c");
        DItem[] item = {new WordItem("visit"),
                        new ListItem(list)};
        DDevice[] device = {
           new HorDevice(), new VerDevice()};

        for (int i = 0; i < item.length; i++)
          for (int d = 0; d < device.length; d++) {
            item[i].display(device[d]);
            System.out.println("");
          }
        }
    }
```

Vytvorili sme pole dvoch položiek a dvoch zariadení. Vidíme, že jednoducho v slučke môžeme uskutočniť zobrazenie všetkých položiek na všetkých zariadeniach. Výstup bude nasledujúci:

```
visit
v
i
s
i
t
```

```
a, b, c,
a,
b,
c,
```

Všimnime si, že kód, ktorý sme vytvorili aplikáciou vzoru Visitor vyhovuje princípu otvorenosti a uzavretosti kódu (pozri časť 6.3) vo viacerých smeroch. Pridanie ďalšieho druhu položky ani zobrazovacieho zariadenia neovplyvní slučku pre zobrazenie v triede `M`. Pridanie ďalšieho druhu zobrazovacieho zariadenia nevyžaduje žiadnu zmenu v typoch položiek. Pridanie ďalšieho druhu položky vyžaduje zásah do zobrazovacích zariadení, avšak nie zmenu jestvujúcich metód, ale ich rozšírenie o ďalšiu metódu.

15.5 Idióm double dispatch

Ako sme už hovorili v časti 6.6, polymorfizmus prekonávaním metód predpokladá dynamické viazanie a dedenie. Java podporuje len jednoduchý polymorfizmus, ale sú jazyky — ako napríklad CLOS — ktoré podporujú viacnásobný polymorfizmus.

Pri operácii `accept()` vzor Visitor využíva idióm *double dispatch*, ktorý predstavuje implementáciu viacnásobného polymorfizmu v Jave. V operácii `accept()` sa najprv dynamicky rozhodne o metóde `display()` na základe skutočného typu položky. Následne sa v metóde `display()` dynamicky rozhodne o metóde `write()` na základe skutočného typu zariadenia. Výber metódy na zobrazenie závisí od typu dvoch prijímačov:

```
DItem[] item; // Element
DDevice[] device; // Visitor
. . .
item[i].display(device[d]); // element.accept(visitor)
```

Vo všeobecnosti ide vlastne o metódu `accept()`, čo je aj naznačené v poznámke k triede `Element1` na obr. 15.2:

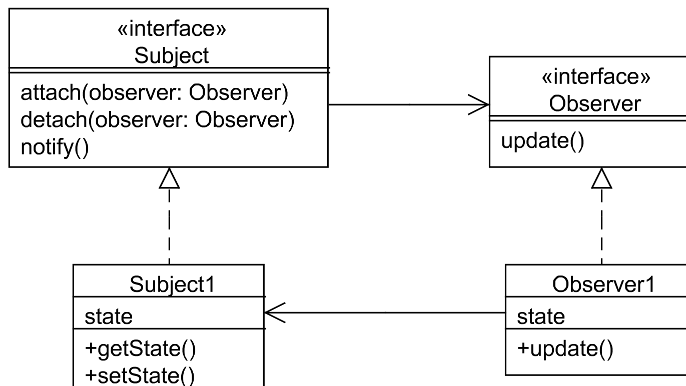
```
public void accept(Visitor v) {
   v.visit(this);
}
```

15.6 Návrhový vzor Observer

Návrhový vzor Observer (pozorovateľ) umožňuje definovanie závislosti stavu viacerých objektov od ďalšieho objektu. Ide znovu o behaviorálny vzor, ktorý sa vzťahuje na objekty.

Observer je vhodný, ak treba oddeliť od seba dva vzájomne závislé aspekty a ak zmena v jednom type objektov vyžaduje zmeny v inom type objektov, ktorých počet a presný typ nie je známy.

Štruktúra vzoru Observer je znázornená na obr. 15.3. Princíp vzoru je založený na vzťahu predmetu pozorovania (subject) a pozorovateľa (observer). Triedy, ktoré majú byť pozorované, udržiavajú zoznam svojich pozorovateľov. Tieto triedy implementujú príslušné rozhranie `Subject`, ktoré predpisuje operácie na pridanie pozorovateľa (`attach`()), odstránenie pozorovateľa (`detach`()) a notifikáciu všetkých pridaných pozorovateľov (`notify`()).



Obrázok 15.3: Štruktúra vzoru Observer.

Triedy, ktoré pozorujú iné triedy, implementujú rozhranie `Observer` s operáciou na aktualizáciu svojho stavu podľa pozorovanej triedy. Aktualizáciu vyvoláva notifikácia zo strany predmetu a preto rozhranie `Subject` pozná rozhranie `Observer`. Na obr. 15.3 je to naznačené orientovanou asociáciou medzi rozhraniami `Subject` a `Observer`. Každý konkrétny pozorovateľ zase pozná predmet pozorovania, čo je naznačené orientovanou asociáciou medzi triedami `Observer1` a `Subject1`. Vo všeobecnosti predmetov pozorovania a pozorovateľov môže, samozrejme, byť viac.

Vzor Observer ozrejmíme príkladom. Predpokladajme, že vyvíjame aplikáciu na zobrazenie teploty vymeranej rôznymi typmi teplotných senzorov. Spôsoby zobrazenia teploty sú rôzne — na digitálnom, analógovom alebo rozsahovom displeji — pričom možno očakávať potrebu zobrazenia aj na ďalších typoch displejov.

Každý displej umožňuje svoje občerstvenie, t.j. aktualizáciu zobrazených údajov:

```
interface TempDisplay { // rozhranie Visitor
   void display();
   void measureTemp();
   void refresh(); // aktualizácia pozorovateľa
}
```

Každý teplotný senzor umožňuje pridanie displeja, jeho odstránenie, notifikáciu všetkých pridaných displejov a vyvolanie fyzického merania teploty.

```
interface TempSensor { // rozhranie Element
   // pripoj pozorovateľa:
   void addDisplay(TempDisplay d);
   // odpoj pozorovateľa:
   void removeDisplay(TempDisplay d);
   // pošli notifikáciu pozorovateľovi:
   void notifyDisplays();
   double readTemp();
   public void measureTemp();
}
```

Jedným zo senzorov je senzor teploty ľudského tela. Samotné zistenie teploty z fyzickej jednotky a implementácia metódy na odstránenie displeja je vynechaná:

```
public class HumanTempSensor implements TempSensor {
   private List displays = new ArrayList();
   private double temp;
   public double refreshRate;

   public void measureTemp() {
      // zistí teplotu z fyzickej jednotky
      notifyDisplays();
   }
   public void setTempDebug(double t) {
      temp = t;
   }
   public void addDisplay(TempDisplay d) {
      displays.add(d);
   }
   public void removeDisplay(TempDisplay d) {
      // . . .
   }
   public void notifyDisplays() {
      for (int i = 0; i < displays.size(); i++) {
```

```
                ((TempDisplay)displays.get(i)).refresh();
            }
        }
    }
```

Tento senzor meria teplotu ľudského tela v pravidelných intervaloch daným hodnotou atribútu `refreshRate`. Samotná realizácia merania je naznačená metódou `measureTemp`(). Na účely ladenia pridaná je metóda `setTempDebug`() na priame nastavenie teploty senzora, ktorá je obsiahnutá v atribúte `temp`. Displej sa do zoznamu displejov v atribúte `displays` pridáva metódou `addDisplay`(), odstraňuje metódou `removeDisplay`(), a obsah všetkých displejov v zozname sa aktualizuje metódou `notifyDisplays`().

Jedným z typov displejov je digitálny displej:

```
public class DigitalTemp implements TempDisplay {
    private HumanTempSensor sensor;
    private float temp;

    public DigitalTemp(HumanTempSensor s) {
        sensor = s;
    }
    public void refresh() {
        temp = (float)sensor.readTemp();
    }
    public void display() { // len dve desatinné miesta
        System.out.println(
            Math.round(temp * 100.0) / 100.0);
    }
    public void measureTemp() {
        sensor.measureTemp();
    }
}
```

Vidíme, že displej obsahuje vlastný stav `temp`, pričom však uchováva teplotu tak, ako ju dokáže zobraziť: zaokrúhlenú na dve desatinné miesta.

Často nás pri meraní teploty ľudského tela vlastne nezaujíma presná teplota, ale informácia, či je teplota normálna, príliš vysoká alebo príliš nízka. Rozsahový displej rozdeľuje teploty do týchto troch pásiem, ktorých názvy sú definované nasledujúcim rozhraním:

```
public enum TempRange { LOW, NORMAL, HIGH }
```

Teplotné pásma sú vymedzené v implementácii samotného rozsahového displeja atribútmi `high` a `low`:

```
public class RelTemp implements TempDisplay {
   private HumanTempSensor sensor;
   TempRange range;
   double high = 37.0;
   double low = 35.0;

   public RelTemp(HumanTempSensor s) {
      sensor = s;
   }
   public void refresh() {
      double temp = sensor.readTemp();

      if (temp <= low)
         range = TempRange.LOW;
      else if (temp >= high)
         range = TempRange.HIGH;
      else
         range = TempRange.NORMAL;
   }
   public void display() {
      switch (range) {
         case TempRange.LOW:
            System.out.println("LOW");
            break;
         case TempRange.HIGH:
            System.out.println("HIGH");
            break;
         default:
            System.out.println("NORMAL");
      }
   }
   public void measureTemp() {
      sensor.measureTemp();
   }
}
```

Aj tento senzor uchováva vlastný stav v tvare, aký dokáže zobraziť (atribút `range`).

Použitie senzora teploty ľudského tela a digitálneho a pásmového displeja demonštruje nasledujúci kód:

```
public class M {
   public static void main(String[] args) {
      HumanTempSensor s = new HumanTempSensor();

      DigitalTemp d1 = new DigitalTemp(s);
      s.addDisplay(d1);
      RelTemp d2 = new RelTemp(s);
      s.addDisplay(d2);

      s.setTempDebug(37.33333333);
      s.notifyDisplays();

      d1.display(); // 37.33
      d2.display(); // HIGH
   }
}
```

Stav jedného senzora `s` sledujú dva displeje: digitálny `d1` a rozsahový `d2`.

Java API obsahuje predprípravu na použitie vzoru Observer.[8] Rozšírením triedy `Observable` možno vytvárať vlastné pozorované objekty. Pozorovatelia musia implementovať rozhranie `Observer`. Nevýhodou je, že dedením od triedy `Observable` blokujeme pozorovanému objektu, ktorý je zvyčajne súčasťou aplikačnej logiky, jedinú možnosť dedenia. Navyše, názvy prvkov vzoru zostanú všeobecné, a nie prispôsobené doméne.

Všimnime si ešte raz vzťah Model-View vo vzore Model-View-Controller (uvedeného v časti 15.2). Tento vzťah je vlastne príkladom použitia vzoru Observer, pričom Model predstavuje predmet pozorovania (Subject), a View pozorovateľa (Observer).

[8] viac na `http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto.html`.

16 Aspektovo-orientované programovanie

Spolu s procedurálnym programovaním objektovo-orientované programovanie predstavuje najpoužívanejší prístup k programovaniu. V porovnaní s procedurálnymi začiatkami objektovo-orientované programovanie a objektovo-orientovaný vývoj softvéru vôbec uľahčujú vysporiadanie sa so zložitosťou softvéru. Mnohé vlastnosti procedurálneho programovania sa zachovali aj v objektovo-orientovanom programovaní. Jednou z nich sú tzv. pretínajúce záležitosti, ktoré spôsobujú prepletenie a roztrúsenie kódu.

Aspektovo-orientované programovanie (aspect-oriented programming, AOP) je prístup, ktorý sa zameriava na modularizáciu pretínajúcich záležitostí. Tento prístup nadväzuje na objektovo-orientovanú paradigmu, t.j. podobne ako pri zmene z procedurálneho na objektovo-orientované programovanie nedá sa hovoriť o zlome paradigmy [Vra02], ako ho definoval Thomas Kuhn (pozri časť 2.1).

Mnohé aspektovo-orientované programovacie jazyky predstavujú rozšírenia objektovo-orientovaných jazykov. Najpopulárnejším aspektovo-orientovaným programovacím jazykom, ktorý sa už používa aj v praxi, je AspectJ. Tento programovací jazyk je plne kompatibilný s Javou: každý regulárny program v Jave je regulárny aj v jazyku AspectJ.

V tejto kapitole sa pozrieme na hlavné vlastnosti aspektovo-orientovaného prístupu a programovanie v jazyku AspectJ. Na príklade aspektovo-orientovanej implementácie návrhového vzoru Observer uvidíme, že výhody modularizácie pretínajúcich záležitostí možno využiť aj pri objektovo-orientovaných návrhových vzoroch. Kapitola nepokrýva úplne a detailne všetky vlastnosti jazyka AspectJ (napríklad generické aspekty) — jej cieľom je prvé priblíženie tohto jazyka čitateľovi v nadväznosti na objektovo-orientované programovanie.

16.1 Pretínajúce záležitosti

Oddelenie záležitostí (separation of concerns) je pojem z článku Edsgera W. Dijkstru, v ktorom pojednával o „inteligentnom uvažovaní" vo všeobecnosti: vždy sa sústreďujeme len na jeden *aspekt* predmetu, ktorý skúmame [Dij82]. Pojem však prenikol do oblasti vývoja softvéru vo význame dekompozície. Inými slovami, pre zvládnutie problému musíme byť schopní ho rozložiť. Následne sa jednotlivými záležitosťami môžeme zaoberať samostatne. Tento proces sa niekedy označuje ako modularizácia.

Modulárnosť je jedným zo základných mechanizmov objektovo-orientovaného programovania (pozri časť 6.4). Nie je však možné modularizovať všetky záležitosti v dôsledku čoho dochádza k zapleteniu kódu (code tangling). Rôzne záležitosti bývajú zmiešané v rámci jedného modulu. Tak napríklad v metódach triedy aplikačnej logiky sa vyskytujú volania metód pre logovanie, bezpečnosť, perzistenciu a pod.

Iným prejavom tohto problému je roztrúsenie kódu (code scattering). Prejavuje sa dvoma spôsobmi. Jedna záležitosť sa môže opakovať v rôznych moduloch, ako napríklad logovanie, ktoré sa vyskytuje v metódach tried, ktoré vôbec navzájom nesúvisia. Druhý spôsob prejavu roztrúsenia kódu je roztrúsenie časti tej istej záležitosti v rôznych moduloch. Napríklad autorizácia zahŕňa autentifikáciu používateľa, manažment práv, kontrolu prístupu atď. Napriek tomu, že tvoria logický celok, každá z týchto častí je potrebná — a implementovaná — v inom module.

K zapleteniu a roztrúseniu kódu dochádza súčasne — ide o dva prejavy toho istého problému. K týmto javom dochádza vo všetkých prístupoch k programovaniu založených na zovšeobecnených procedúrach (generalized procedures) [KLM+97]. Záležitosti, ktoré sa vymykajú modularizácii, sa označujú ako pretínajúce záležitosti (crosscutting concerns). Aspektovo-orientované programovanie umožňuje práve modularizáciu týchto pretínajúcich záležitostí alebo — povedané v Dijkstrovom duchu — oddelenie pretínajúcich záležitostí (separation of crosscutting concerns).

Zapletený a roztrúsený kód sa ťažko udržiava. Zmena v pretínajúcej záležitosti často vyžaduje zásah do všetkých miest, kde je použitá. Podobný problém vzniká pri odstraňovaní záležitosti (napríklad vypnutie logovania) alebo pridávaní ďalších záležitostí.

16.2 Príklad: jednoduché monitorovanie

Pre vytvorenie lepšej predstavy o probléme pretínajúcich záležitostí pozrieme sa na jednoduchý príklad. Príklad bude obsahovať aj kód v jazyku AspectJ. V prvom priblížení sa spoľahneme na intuitívne pochopenie kódu bez rozboru syntaxe jazyka AspectJ, ktorej sa venujú nasledujúce časti.

Predpokladajme, že vyvíjame malý grafický systém. Používame nasledujúcu reprezentáciu bodu:

```
public class Point {
   private int x;
   private int y;

   public void setX(int x) {
      this.x = x;
   }
   public void setY(int y) {
      this.y = y;
   }
   public int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
}
```

Predpokladajme ďalej, že potrebujeme monitorovať operácie nad bodmi. Musíme preto upraviť všetky **set** a **get** metódy:

```
public class Point {
   private int x;
   private int y;

   public void setX(int x) {
      System.out.println("Posunutie bodu.");
      this.x = x;
   }
   public void setY(int y) {
      System.out.println("Posunutie bodu.");
      this.y = y;
   }
   public int getX() {
      System.out.println("Čítanie bodu.");
      return x;
   }
   public int getY() {
      System.out.println("Čítanie bodu.");
      return y; }
}
```

Čo sme dostáli? Došlo k prepleteniu kódu: kód na monitorovanie je prepletený s kódom aplikačnej logiky. Došlo zároveň k roztrúseniu kódu: kód na monitorovanie sa opakuje v rôznych metódach. Čo keby sme chceli vypnúť monitorovanie? Zisťujeme, že sme mali použiť o jednu úroveň abstrakcie viac:

```
public void setX(int x) {
  this.x = x;
  PointMonitor.print("Posunutie bodu.");
}
```

Takto implementované monitorovanie vypneme úpravou metódy `print()`. Zostáva však problém samotnej prítomnosti monitorovania v aplikačnej logike a potreba manuálneho pridávania tohto kódu.

Spomeňme si, čo sme vlastne chceli dosiahnuť: *pripojiť* monitorovacie výpisy k všetkým **set** a **get** metódam. Teraz je na mieste otázka, či vieme presne špecifikovať, o aké metódy ide. Ide o:

1. metódy triedy `Point`, ktorých názov začína na **set**
2. metódy triedy `Point`, ktorých názov začína na **get**

Tento problém by mal priamočiare riešenie v programovacom jazyku, ktorý by priamo umožňoval vyjadriť, že *pred vykonaním* každej metódy, ktorá vyhovuje uvedenej špecifikácii, treba podať príslušný výpis bez potreby zasahovať do metódy samotnej. AspectJ to umožňuje:

```
public aspect AccessMonitoring {
   before(): execution(* Point.set*(..)) {
      System.out.println("Posunutie bodu.");
   }
   before(): execution(* Point.get*(..)) {
      System.out.println("Čítanie bodu.");
   }
}
```

Uvedený kód predstavuje *aspekt* pre sledovanie operácií nad objektmi triedy `Point`. Vykonaním príkazov

```
Point p = new Point();
p.setX(10);
p.setY(p.getX());
```

dostávame nasledujúci výstup:

Posunutie bodu.
Čítanie bodu.
Posunutie bodu.

Predstavme si, že potrebujeme v klasickej Jave monitorovať aj ukončenie operácií. Pri výpise to zdanlivo nie je problém:

```
public void setX(int x) {
   System.out.println("Posunutie bodu.");
   this.x = x;
   System.out.println("Bod bol posunutý.");
}
```

Všimnime si, že výpis je v skutočnosti ešte v rámci vykonávania monitorovanej metódy. Úplne jasné je to pri metódach, ktoré vracajú hodnotu:

```
public int getX() {
   System.out.println("Čítanie bodu.");
   return x;
   System.out.println("Bod bol čítaný."); // nevykoná sa!
}
```

Pre monitorovanie ukončenia operácií by sme v Jave museli pridať výpis za každé volanie v klientskom kóde. V jazyku AspectJ môžeme mať výpis skutočne *po vykonaní* metódy:

```
public aspect AccessMonitoring {
   . . .
   after(): execution(* Point.set*(..)) {
      System.out.println("Bod bol posunutý.");
   }
   after(): execution(* Point.get*(..)) {
      System.out.println("Bod bol čítaný.");
   }
}
```

Môžeme mať výpis aj za každým *volaním* metódy:

```
public aspect AccessMonitoring {
   . . .
   after(): call(* Point.set*(..)) {
      System.out.println("Bod bol posunutý.");
   }
   after(): call(* Point.get*(..)) {
      System.out.println("Bod bol čítaný.");
   }
}
```

16.3 Body spájania

Body spájania (join points) sú kľúčovým pojmom v aspektovo-orientovanom programovaní. Body spájania predstavujú dobre definované miesta vo vykonávaní programu. Na týchto miestach možno ovplyvniť tok programu. Príkladom bodu spájania je vykonávanie metódy, volanie metódy, prístup k atribútu, nastavenie atribútu a pod. Od programovacieho jazyka závisí, ktoré body spájania budú exponované. Napríklad slučky v jazyku AspectJ nepredstavujú exponované body spájania.

Fundamentálnym problémom v aspektovo-orientovanom programovaní je výber bodov spájania, t.j. ako vybrať body spájania, ktoré potrebujeme. Body spájania definujeme pomocou primitívnych *bodových prierezov* definovaných v samotnom programovacom jazyku. Bodové prierezy predstavujú množiny bodov spájania.

Niektoré body spájania možno zachytiť priamo v zdrojovom texte. Tieto body spájania sa označujú ako statické. Dynamické body spájania možno zachytiť len pri vykonávaní programu — ako napríklad spomínané volania metód. Aspekty potom môžu pracovať s kontextom bodu spájania — napríklad ovplyvňovať hodnoty parametrov metódy.

16.4 Základné vlastnosti jazyka AspectJ

Aspektovo-orientované programovanie sa formovalo v štyroch (pôvodne) nezávislých vetvách:

- PARC[1] AOP, kde aspekty predstavujú moduly pre pretínajúce záležitosti.

[1] Palo Alto Research Center

- Subjektovo-orientované programovanie,[2] v ktorom sa aspektovo-orientovaná kompozícia dosahuje skladaním tzv. subjektov ako parciálnych pohľadov na problém.
- Kompozičné filtre (programovací jazyk Sina/st),[3] kde sa aspektovo-orientovaná kompozícia dosahuje skladaním filtrov, ktoré filtrujú správy pre daný objekt.
- Adaptívne programovanie (programovací jazyk DemeterJ),[4] v ktorom sa pracuje s prispôsobivými stratégiami prechádzania grafom tried.

Štyri vymenované aspektovo-orientvané prístupy sú stále základom pre vznikajúce aspektovo-orientované jazyky a prístupy k vývoju softvéru vo všeobecnosti — aj keď problém pretínajúcich záležitostí bol spozorovaný predovšetkým v kóde, prejavuje sa aj v návrhu a analýze; dokonca je prítomný už na úrovni požiadaviek. PARC AOP je najvplyvnejší aspektovo-orientovaný prístup a keď sa hovorí o aspektovo-orientovanom programovaní, väčšinou sa myslí na PARC AOP, ktorého hlavným predstaviteľom je práve jazyk AspectJ. V súčasnosti jestvujú desiatky aspektovo-orientovaných jazykov založených predovšetkým na prístupe PARC AOP a jazyku AspectJ.

Popularita jazyka AspectJ určite pramení aj v jeho otvorenom vývoji od roku 1997.[5] Verzia 1.0 vyšla na konci roku 2001, a v roku 2002 projekt prešiel na eclipse.org.[6] AspectJ sleduje vývoj Javy a jeho súčasná verzia (1.5.4) je založená na Jave 5.0. Jestvuje podpora pre AspectJ v rozšírených vývojových prostrediach: Eclipse, JBuilder a NetBeans.

Základnou aspektovo-orientovanou konštrukciou v jazyku AspectJ je *aspekt* (aspect). Aspekty modifikujú vykonávanie programu v bodoch spájania. Body spájania sú určené *bodovými prierezmi* (pointcuts). Modifikácie sú vyjadrené pomocou *videní*.[7] (advices). Videnie sa vykonáva pred, po alebo namiesto bodu spájania. Aspekty môžu aj dopĺňať triedy novými prvkami pomocou *medzitypových deklarácií* (inter-type declarations). Takto je do triedy možné pridať nové prvky, ale aj závislosti dedenia. Inštancie aspektov sa vytvárajú automaticky.

Vtkanie (weaving) aspektov sa realizuje priamo pri preklade. Dostupný je prekladač `ajc` alebo integrovaný prekladač do pluginu pre prostredie Eclipse na podporu jazyka AspectJ. Výsledkom prekladu je Java bajtkód, t.j. preložený program sa vykonáva na Java VM ako bežný program v Jave. V starších verziách výsledok vtkania bol kód v Jave, ktorý sa následne prekladal prekladačom `javac`. Ako už bolo povedané,

[2] pozri `http://www.research.ibm.com/hyperspace/`

[3] pozri `http://trese.cs.utwente.nl/composition_filters`

[4] pozri `http://www.ccs.neu.edu/research/demeter/`

[5] `http://www.parc.com/research/csl/projects/aspectj/`

[6] `http://eclipse.org/aspectj/`

[7] Priamy preklad *advice* by bol *rada*, ale z etymologického hľadiska anglická rada je vlastne názor, pohľad na niečo alebo *videnie* niečoho, nie riadenie (z *avis*, *advis*; Meriam-Webster Online, `http://m-w.com/dictionary/advice`). Používa sa ešte termín *odporúčanie*.

každý program v Jave je platný AspectJ program. Pri preklade je medzi prekladačmi `javac` a `ajc` jeden významný rozdiel: `ajc` vyžaduje uvedenie všetkých súborov, ktoré sa prekladajú, čo dáva možnosť rozhodnúť o pripojení určitého aspektu pri preklade. Nie je nutné mať zdrojové súbory — `ajc` môže vtkať aspekty aj do bajtkódu.

16.5 Bodové prierezy

Ako už bolo povedané, bod spájania predstavuje dobre definované miesto vo vykonávaní programu, a v ktorom je teoreticky možné ovplyvniť vykonávanie programu. Praktický je to možné len v exponovaných bodoch spájania, t.j. bodoch spájania, ku ktorým je v danom programovacom jazyku možné pristúpiť. Keď sa hovorí o bodoch spájania, myslí sa na exponované body spájania.

Volanie metódy je napríklad v jazyku AspectJ exponovaný bod spájania. Exponovaným bodom spájania však nie je slučka **for**, čo bolo rozhodnuté pri návrhu jazyka.

V jazyku AspectJ možno pracovať s kontextom bodu spájania, ktorý tiež už bol spomenutý. Kontext volania metódy napríklad predstavuje objekt, ktorý ju zavolal, cieľový objekt a parametre metódy.

Exponované body spájania v jazyku AspectJ sú:

- metódy a konštruktory:
  - volanie metódy alebo konštruktora
  - vykonanie metódy alebo konštruktora
- prístup k atribútom:
  - čítanie atribútu
  - zápis atribútu
- spracovanie výnimiek — vykonanie bloku spracovania výnimky
- inicializácia tried a objektov:
  - vykonanie bloku statickej inicializácie
  - vykonanie inicializácie objektu
  - vykonanie predinicializácie objektu (inicializácia pred zavolaním **super**())
- vykonanie videnia — vykonanie všetkých videní v programe

K bodom spájania sa nepristupuje jednotlivo, ale cez bodové prierezy — množiny bodov spájania — ktoré ich obsahujú. Bodové prierezy sa definujú pomocou primitívnych bodových prierezov a logických spojok *a* (&&) a *alebo* (||) a negácie (!).

Bodový prierez, ktorý zachytáva volania metód triedy `Point`, ktorých názov začína na **set** a nevracajú žiadnu hodnotu, by sme v jazyku AspectJ zapísali takto:

```
call(void Point.set*(..))
```

Pomocou logickej spojky *alebo* môžeme do množiny bodov spájania, ktoré zachytávame, pridať aj metódy na čítanie bodov:

```
call(void Point.set*(..)) || call(int Point.get*(..))
```

Signatúry

Metódy, konštruktory, typy a atribúty sa v bodových prierezoch zadávajú pomocou svojich signatúr. Môže sa zadať aj presne jeden prvok, ako napríklad jedna metóda:

```
call(void Point.setX())
```

Najčastejšie sa však používajú signatúry s náhradnými znakmi. Náhradný znak * nahrádza názov typu, metódy alebo atribútu, alebo časť tohto názvu. Signatúra

```
i* P*.get*()
```

špecifikuje všetky metódy s názvom, ktorý začína na **get**, vo všetkých typoch, ktorých názvy začínajú na `P`, a ktoré vracajú hodnotu typu, ktorého názov začína na `i`.

Náhradný znak .. v prípade metódy nahrádza zoznam parametrov alebo jeho časť. Signatúra

```
void Point.set*(..)
```

špecifikuje všetky metódy `void Point.set*()` s hocijakým počtom parametrov.

Náhradný znak .. v prípade balíka nahrádza podbalíky (aj priame, aj nepriame). Signatúra

```
java..*
```

špecifikuje všetky typy v rámci balíka `java`.

Náhradný znak + nahrádza podtypy daného typu. Signatúra

```
Point+
```

špecifikuje triedu `Point` a všetky jej podtriedy. Signatúra

```
* Point+.*(..)
```

špecifikuje všetky metódy triedy `Point` a jej podtried.

Ak v signatúre metódy nie je uvedený modifikátor, signatúra zahŕňa všetky modifikácie metódy. Modifikátory zahŕňajú modifikátory prístupu, **abstract**, **static** a **final**. Dá sa zadať negácia modifikátora a typu vracanej hodnoty, napríklad:

```
final !public !protected !static int MyClass.m*()
```

alebo:

```
!private !public !protected * MyClass.*(..)
```

Negácia sa dá použiť aj na typ návratovej hodnoty:

```
!int MyClass.m*()
```

Metódy možno špecifikovať aj vzhľadom na výnimky, ktoré vyhadzujú. Nasledujúca signatúra špecifikuje všetky metódy, ktorých názov začína na **set**, a ktoré vyhadzujú výnimku `InvalidCoordinatesException`:

```
* *.set*(..) throws InvalidCoordinatesException
```

Ak potrebujeme špecifikovať konštruktor, namiesto názvu metódy použijeme kľúčové slovo **new**. Nasledujúca signatúra špecifikuje všetky konštruktory triedy `Point`:

```
Point.new(..)
```

Primitívne bodové prierezy

Primitívne bodové prierezy zodpovedajú identifikovaným typom bodov spájania. AspectJ zahŕňa viac než dvadsať primitívnych bodových prierezov (bez ohľadu na to, ako ich počítame). Základné skupiny primitívnych bodových prierezov sú:

- volania a vykonávania metód a konštruktorov
- prístup k atribútom
- inicializácia
- spracovanie výnimiek
- bodové prierezy založené na toku riadenia
- bodové prierezy založené na lexikálnej štruktúre
- bodové prierezy vykonávajúcich objektov
- bodové prierezy parametrov
- vykonanie videnia
- podmienený bodový prierez
- bodové prierezy založené na anotáciách[8]

Bodové prierezy na zachytenie *volaní a vykonávaní metód a konštruktorov* sú:

- `call`(`method_signature`) — všetky volania metód, ktoré vyhovujú signatúre
- **call**(`constructor_signature`) — všetky volania konštruktorov, ktoré vyhovujú signatúre
- **execution**(`method_signature`) — všetky vykonávania metód, ktoré vyhovujú signatúre
- **execution**(`constructor_signature`) — všetky vykonávania konštruktorov, ktoré vyhovujú signatúre

Bodové prierezy na zachytenie *prístupu k atribútom* sú:

- **get**(`field_signature`) — všetky čítania atribútov s danou signatúrou
- **set**(`field_signature`) — všetky nastavenia atribútov s danou signatúrou

Bodový prierez

```
get(* Point.x)
```

zachytáva čítanie atribútov objektov triedy `Point`. Bodový prierez

[8] V tomto texte sa nebudeme zaoberať bodovými prierezmi založenými na anotáciách.

```
set(private * *.*)
```

zachytáva nastavenie všetkých atribútov s prístupom typu `private`.

Bodové prierezy na zachytenie *inicializácie* sú:

- **staticinitialization**(`type_signature`),
- **initialization**(`constructor_signature`),
- **preinitialization**(`constructor_signature`).

Bodový prierez

```
staticinitialization(*)
```

zachytáva statickú inicializáciu všetkých tried (napríklad pre sledovanie poradia načítavania tried). Bodový prierez

```
initialization(Point.new())
```

zachytáva inicializáciu objektov triedy `Point` konštruktormi bez parametrov. Bodový prierez

```
preinitialization(Point.new())
```

zachytáva predinicializáciu objektu triedy `Point` konštruktormi bez parametrov.

Spracovanie výnimiek možno zachytiť bodovým prierezom v tvare:

```
handler(type_signature)
```

Napríklad bodový prierez

```
handler(InvalidCoordinatesException)
```

zachytí vykonanie bloku spracovania výnimky `InvalidCoordinatesException` (ak k tejto výnimke príde).

Bodové prierezy *založené na toku riadenia* zachytávajú všetky body spájania vo vykonávaní zadaného bodového prierezu:

- **cflow**(**pointcut**(`arguments`)),
- **cflowbelow**(**pointcut**(`arguments`)).

V prípade **cflowbelow**() vynechávajú sa body spájania samotného zadaného bodového prierezu. Pri aplikácii nasledujúceho videnia nad reprezentáciou bodu, ktorá bola uvedená v časti 16.2:

```
before(): !within(PointMonitoring) &&
   cflow(call(* Point.setX(..))) {
   System.out.println(thisJoinPoint);
}
```

Výstup bude nasledujúci:

```
call(void Point.setX(int))
execution(void Point.setX(int))
set(int Point.x)
```

Keby sme namiesto **cflow**() aplikovali **cflowbelow**(), výstup by neobsahoval prvý riadok.

Bodové prierezy založené na lexikálnej štruktúre sú:

- **within**(`type_signature`),
- **withincode**(`method_signature`),
- **withincode**(`constructor_signature`)

Bodový prierez

```
within(Point+)
```

zachytí všetky body spájania v rámci triedy `Point` jej podtypov.

Ak v danom aspekte `A` potrebujeme zachytiť volania všetkých metód, môžeme použiť nasledujúci bodový prierez:

```
!within(A) && call(* *.*(..))|
```

Tento bodový prierez zachytí volania všetkých metód okrem volaní v rámci aspektu `A`. Bez vylúčenia volaní v rámci aspektu by mohlo dôjsť k nekonečnej rekurzii: bodový prierez by zachytával aj volania v samotnom videní nad týmto bodovým prierezom, čo by znovu aktivovalo to isté videnie.

Bodový prierez

```
withincode(public static void Test.main(String[]))
```

zachytí body spájania v rámci metódy **public static void** `Test.main(String[])`.

Bodové prierezy *vykonávajúcich objektov* sú:

- **this**(`type`)
- **target**(`type`)

Bodový prierez

```
this(Point)
```

zachytí body spájania v rámci vykonávania objektu typu `Point` alebo jeho podtypu, t.j. také, pre ktoré platí:

```
this instanceof Point == true
```

Bodový prierez

```
target(Point)
```

zachytí body spájania, v ktorých je objekt, nad ktorým je zavolaná metóda typu `Point` alebo jeho podtypu. Typ môže byť reprezentovaný aj identifikátorom objektu (pre potreby práce s kontextom).

Na zachytenie *parametrov* slúži bodový prierez:

```
args(argument_list)
```

kde `argument_list` je zoznam parametrov. Parameter je reprezentovaný alebo svojím typom definovaným signatúrou typu, alebo identifikátorom objektu (pre potreby práce s kontextom).

Bodový prierez

```
args(int)
```

zachytí všetky body spájania vo všetkých metódach, ktorých jeden parameter je typu **int**.

Samotné vykonania videnia možno tiež zachytiť bodovým prierezom:

```
advice()
```

Bodový prierez `advice()` zachytí vykonanie všetkých videní.

Pomocou lexikálnych bodových prierezov môžeme upresniť videnie, ktoré chceme zachytiť. Bodový prierez

```
within(PointMonitoring) && advice())
```

zachytí vykonanie všetkých videní v rámci aspektu `PointMonitoring`.

Pomocou podmieneného bodového prierezu možno ovplyvniť vyhodnotenie zloženého bodového prierezu:

```
if(boolean_expression)
```

Tento bodový prierez zachytáva všetky body spájania, v ktorých platí podmienka `boolean_expression`.

Bodový prierez

```
if(p.x < 100)
```

zachytáva všetky body spájania, v ktorých platí podmienka, že atribút `x` objektu `p` má hodnotu menšiu než 100. Objekt `p` je súčasťou kontextu a deklaruje sa vo videní. Mechanizmus prenášania kontextu bude vysvetlený v časti 16.6.

Pomenované bodové prierezy

Bodové prierezy je možné pomenovať. Nasledujúci aspekt definuje bodový prierez `getAndSet()`:

```
aspect PointMonitoring {
   . . .
   pointcut getAndSet():
      call(* Point.set*(..)) || call(* Point.get*());
   . . .
}
```

Pomenované bodové prierezy sa ďalej dajú použiť ako bodové prierezy pre vykonávanie videní alebo pre definovanie iných zložených bodových prierezov:

```
aspect PointMonitoring {
   . . .
   pointcut getMethods(): call(* Point.get*());
   pointcut getAndSet():
      call(* Point.set*(..)) || getMethods();

   before(): getAndSet() {
      . . .
   }
   . . .
}
```

Pomenované bodové prierezy sa dedia, o čom bude reč v časti 16.9.

16.6 Videnia

Videnia definujú aktivity, ktoré sa majú vykonať v spojitosti so zachytenými bodmi spájania. Rámcová syntax videní je nasledujúca:

```
advice_type(argument_list): pointcut(arguments) {
   advice_body
}
```

Telo videnia (`advice_body`) je podobné telu metódy. Zoznam parametrov videnia (`argument_list`) sa používa na prenos kontextu. Bodový prierez videnia (**pointcut**()) môže byť pomenovaný alebo nepomenovaný. Jestvujú tri typy videní (`advice_type`):

- **before**()
- **after**()
- **around**()

Videnie before()

Videnie **before**() má nasledujúcu syntax:

```
before(argument_list): pointcut(arguments) {
   . . .
}
```

Videnie **before()** sa vykonáva pred zachytenými bodmi spájania. S týmto typom videnia sme sa už stretli v príklade ohľadom monitorovania operácií nad bodom (pozri časť 16.2):

```
before(): call(void Point.get*(..)) {
   System.out.println("Čítanie bodu.");
}
```

Videnie **after()**

Videnie **after()** má tri tvary:

```
after(argument_list): pointcut(arguments) {
   . . .
}

after(argument_list): returning(return_value) pointcut(arguments) {
   . . .
}

after(argument_list): throwing(exception) pointcut(arguments) {
   . . .
}
```

Všetky tri typy videnia **after()** sa vykonávajú po zachytených bodoch spájania. Aj s týmto videním sme sa už stretli v príklade ohľadom monitorovania operácií nad bodom:

```
after(): call(int Point.get*()) {
   System.out.println("Point read.");
}
```

Videnie **after()** je možné obmedziť na úspešne vykonané body spájania (pri ktorých nevznikla výnimka) klauzulou `returning`(). Predpokladajme napríklad, že metódy `Point.set*(..)` vyhadzujú `InvalidCoordinatesException` v prípade, že sa zadajú nepovolené súradnice. Nás však zaujíma len prípad skutočného premiestnenia bodu — nie prípad, keď vznikne výnimka:

```
after() returning: call(void Point.set*(..)) {
   System.out.println("Bod bol posunutý.");
}
```

Videnie `after()` **returning**() môže pracovať aj s návratovou hodnotou. Návratovú hodnotu treba deklarovať za kľúčovým slovom **returning**. V našom príklade by sme mohli napr. vypísať prečítanú súradnicu bodu:

```
after() returning(int c): call(int Point.get*()) {
   System.out.println("Bod bol čítaný " + c);
}
```

Videnie **after()** je možné obmedziť práve na body spájania, pri ktorých došlo k výnimke klauzulou **after**() **returning**(). V našom príklade by sme mohli vypísať dodatočné upozornenie, ak vznikne výnimka:

```
after() throwing: call(void Point.set*(..)) {
   System.out.println("Chyba pri posúvaní bodu");
}
```

Videnie **after**() **throwing**() môže pracovať so samotnou výnimkou, ktorá je — ako si pamätáme z kapitoly 8 — reprezentovaná objektom. Výnimku treba deklarovať za kľúčovým slovom **throwing**. V príklade monitorovania operácie posunu bodu výpis môžeme rozšíriť o výpis informácií o výnimke samotnej:

```
after() throwing(InvalidCoordinatesException e):
   call(void Point.set*(..)) {
   System.out.println("Chyba pri posúvaní bodu: " + e);
}
```

Videnie **around()**

Videnie **around()** má nasledujúcu syntax:

```
return_value around(argument\_list): pointcut(arguments) {
   . . .
}
```

Videnie **around()** sa vykonáva namiesto každého zo zachytených bodov spájania. Vykonanie samotného bodu spájania sa dá vyvolať pomocou kľúčového slova **proceed**(). Videnie musí deklarovať typ návratovej hodnoty (`return_value`) bodu spájania. Typ návratovej hodnoty môže byť **void**. Ak sa uvedie typ `Object`, AspectJ zabezpečí pretypovanie na zodpovedajúci typ, čo platí aj pre primitívne typy vrátane typu **void**. Typ `Object` sa musí uviesť, ak bodový prierez zachytáva body spájania s rôznymi návratovými hodnotami.

Videnie **around()** nemá veľký zmysel bez využitia kontextu bodu spájania. Prenos kontextu bodu spájania a jeho využitie vo videní vysvetľuje nasledujúca časť.

16.7 Kontext bodu spájania

Kontext bodu spájania predstavujú aktuálne hodnoty prvkov, ktoré sú s nim spojené. Napríklad kontext volania metódy predstavuje objekt, ktorý ju zavolal, cieľový objekt a jej parametre. Kontext bodu spájania závisí aj od videnia — pri videniach typu **after()** prichádzajú do úvahy aj návratová hodnota a zachytená výnimka.

Kontext sa dá zachytiť prostredníctvom reflektívneho API jazyka AspectJ a pomocou bodových prierezov **this**(), **target**() a **args**() a videní **after**() **returning**() a **after**() **throwing**() Tak ako v Jave uprednostňujeme polymorfizmus pred reflektívnym API, treba uprednostňovať zachytenie kontextu bodovými prierezmi, kde sa to dá.

Kontext bodu spájania sa prenáša podobne ako parametre do metódy. Videnie (akoby metóda) deklaruje typ premenných kontextu (akoby parametrov tejto metódy). Pri metóde sa hodnoty parametrov zadávajú pri volaní. Podobne je to aj s videním. Videnie však býva vyvolané bodovým prierezom a hodnoty premenných kontextu teda dostáva z bodového prierezu. Tieto hodnoty sa následne dajú využiť v tele videnia — rovnako ako parametre v tele metódy.

Vráťme sa znovu k príkladu monitorovania posunu bodu. Predpokladajme, že potrebujeme zakázať také posuny, ktorými by sa bod dostal mimo obrazovky. Toto môžeme dosiahnuť pomocou nasledujúceho videnia **around()**:

```
void around(int i):
   call(void Point.setX(int)) && args(i) {
   if (i > 0 && i < 320)
      proceed(i);
}
```

Videnie deklaruje jeden parameter `i` typu **int**. Parameter `i` získa svoju hodnotu v bodovom priereze **args**(). Hodnota je následne dostupná v tele videnia.

Pri definícii pomenovaného bodového prierezu sa tiež definujú parametre, na mieste ktorých sa pri využití pomenovaného bodového prierezu uvádzajú názvy príslušných parametrov deklarovaných vo videní. Upravený kód riadenia posunu bodu v rámci stanoveného rozsahu demonštruje použitie pomenovaného bodového prierezu:

```
pointcut xSetter(int i):
   call(void Point.setX(int)) && args(i);

void around(int i): xSetter(i) {
   if (i > 0 && i < 320)
      proceed(i);
}
```

Hodnotu, na ktorú sa chystáme nastaviť atribút objektu alebo triedy, zachytíme tiež pomocou bodového prierezu **args**(). Namiesto sledovania volaní metódy na nastavenie súradnice x bodu môžeme sledovať priamo zmenu tejto súradnice a zachytávať hodnotu, na ktorú sa má nastaviť bodovým prierezom **args**():

```
void around(int i): set(int Point.x) && args(i) {
   if (i > 0 && i < 320)
      proceed(i);
}
```

Súčasťou kontextu bodu spájania je aj jeho návratová hodnota. Na jej zachytenie, ako už bolo povedané, môžeme použiť videnie typu **after**() **returning**(). Zachytená hodnota je následne dostupná v tele videnia. V nasledujúcom príklade zachytený bod spájania je volanie metódy na prečítanie hodnoty súradnice x bodu, ktorá je následne vypísaná:

```
pointcut xGetter(): call(int Point.getX();

void after() returning(int i): xGetter() {
   System.out.println("Vratena hodnota: " + i);
}
```

Aj výnimka, ktorá vznikne pri vykonávaní bodu spájania, je súčasťou jeho kontextu. Výnimku môžeme zachytiť videním typu **after**() **throwing**(). Zachytená výnimka je následne dostupná v tele videnia. Predpokladajme, že sú niektoré hodnoty súradníc neprípustné. Pri pokuse o nastavovanie súradnice na takú hodnotu vznikne výnimka. Túto výnimku môžeme zachytiť videním typu **after**() **throwing**() a následne vypísať:

```
pointcut xSetter(int i): call(void Point.set*(i));

void after() throwing(InvalidCoordinatesException e):
   call(void Point.set*(int)) {
   System.out.println("Suradnica mimo rozsahu: " + e);
}
```

16.8 Medzitypové deklarácie

Aspekty môžu zavádzať nové prvky do typov a vzťahy medzi typmi pomocou tzv. medzitypových deklarácií (inter-type declarations).[9] AspectJ podporuje nasledujúce

[9] predtým označované ako zavedenia (introductions)

druhy medzitypových deklarácií:

- zavedenie členského prvku (atribútu alebo metódy)
- zavedenie vzťahu dedenia
- zavedenie chyby pri preklade
- zmäkčenie výnimky (exception softening)

V tomto texte sa obmedzíme na prvé dva typy medzitypových deklarácií.

Zavedenie členského prvku

Najpoužívanejším druhom medzitypovej deklarácie je zavedenie členského prvku. Pozrieme sa na zavedenie členského prvku na príklade obmedzenia súradníc bodu. Súradnice bodu majú základné obmedzenie, že nemôžu byť záporné. V prípade pokusu o nastavenie súradnice na zápornú hodnotu, vznikne výnimka:

```
public class Point {
   private int x;
   private int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public void setX(int x)
      throws CoordinateOutOfBoundsException {
      if (x < 0) {
         throw new CoordinateOutOfBoundsException(
            "Coordinates out of bounds.");
      }
      else {
         this.x = x;
      }
   }
   public void setY(int y)
      throws CoordinateOutOfBoundsException {
      if (y < 0) {
         throw new CoordinateOutOfBoundsException(
            "Coordinates out of bounds.");
      }
      else {
         this.y = y;
```

```
        }
    }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

Od triedy `Point` odvodíme bod, ktorý má ďalšie obmedzenie hodnoty súradníc:

```
public class BoundedPoint extends Point {
    public BoundedPoint(int x, int y) {
        super(x, y);
    }
}
```

Obmedzenie hodnoty súradníc pre bod typu `BoundedPoint` zabezpečuje aspekt:

```
public aspect {
    private int Point._maxX;
    private int Point._maxY;

    public bool Point.checkCoors() {
        if (getX() <= _maxX && getY() <= _maxY)
            return true;
        return false;
    }

    after(Point p):
        execution(BoundedPoint.new(..)) && this(p) {
        p._maxX = 319;
        p._maxY = 199;
    }

    before(Point p, int c)
        throws CoordinateOutOfBoundsException:
        execution(* p.set*(..)) && this(p) && args(c) {
        if (p.checkCoors()) {
            throw new CoordinateOutOfBoundsException(
                "Coordinates out of bounds.");
        }
    }
}
```

Zaviedli sme atribúty `_maxX` a `_maxY` a metódu `checkCoors()` na sledovanie hodnoty súradníc. Modifikátory prístupu týchto prvkov sú vzhľadom k aspektu, ktorý ich zaviedol. Dôvodom, pre ktorý sme tieto prvky neuviedli priamo v triede `BoundedPoint` je, že sledovanie prekročenia povoleného rozsahu hodnôt súradníc pokladáme za zvláštnu záležitosť a nechceme ju miešať so základnou funkcionalitou bodu.

Zavedenie vzťahu dedenia

Medzitypovou deklaráciou sa dá zaviesť aj vzťah **implements**, aj vzťah **extends**. Pritom sa musia dodržiavať pravidlá dedenia platné v Jave. Jedným z použití zavedenia vzťahu dedenia je zavedenie značkovacieho rozhrania za účelom sledovania prvkov:

```
aspect PointTrackingAspect {
   declare parents:
      graphic..entities.* implements Identifiable
}
```

Zavádzanie vzťahov **extends** je zaujímavé predovšetkým z hľadiska konfigurovateľnosti. Môžeme mať viac tried rovnakého rozhrania, ale inej implementácie. Aspektom určíme, ktorá z týchto tried sa využije.

16.9 Abstraktné aspekty

Aspekty môžu byť abstraktné. Deklarujú sa rovnako ako triedy pomocou kľúčového slova **abstract**. Podobne ako abstraktné triedy ani abstraktné aspekty nemôžu mať inštancie. Od abstraktných aspektov môžu dediť ďalšie abstraktné a konkrétne aspekty. Od konkrétnych aspektov sa nedá dediť. Aspekty môžu dediť od tried bez ohľadu na ich abstraktnosť a abstraktnosť tried.

Abstraktné aspekty môžu obsahovať abstraktné bodové prierezy. Tieto bodové prierezy nemajú telo. Abstraktný aspekt nad abstraktnými bodovými prierezmi definuje plnohodnotné videnia. Abstraktné bodové prierezy sa dajú prekonať (konkretizovať) v odvodených aspektoch. Abstraktné aspekty sú výhodné ak vieme špecifikovať, čo sa má urobiť pre množinu bodov spájania, ktorú nevieme špecifikovať vo všeobecnosti.

Pri monitorovaní operácií nad bodmi nemusíme napríklad vedieť aké presne operácie chceme monitorovať, ale vieme čo sa pri ich zachytení má udiať. Vytvoríme preto abstraktný monitor:

```
public abstract aspect MyMonitor {
   public abstract pointcut monitoredOperations();

   before(): monitoredOperations() {
      System.out.println("Monitor:" + thisJoinPoint);
   }
}
```

Konkrétny monitor môže napríklad monitorovať zmenu a čítanie súradníc bodu. Stačí v ňom upresniť bodový prierez na zachytenie operácií:

```
public aspect MyPointMonitor extends MyMonitor {
   public pointcut monitoredOperations():
      call(void Point.set*(..)) && call(* Point.get*());
}
```

16.10 Reflektívna podpora pre body spájania

Analogicky k Java Reflective API, AspectJ poskytuje reflektívnu podporu pre body spájania. Za týmto účelom AspectJ poskytuje tri špeciálne referencie podobné referencii **this**:

- **thisJoinPoint**
- **thisJoinPointStaticPart**
- **thisEnclosingJoinPointStaticPart**

Tieto referencie sa dajú použiť pre získanie jednoduchej textovej informácie (príslušné objekty majú prekonanú metódu `toString`()):

```
after() returning(Object x): call(* C.m()) {
   System.out.println(thisJoinPoint);
}
```

Referencie na bod spájania sa dajú využiť aj pre zistenie kontextu bodu spájania. Prostredníctvom referencie **thisJoinPoint** môžeme zistiť informácie o aktuálnom bode spájania prostredníctvom nasledujúcich metód:

- getArgs() — parametre bodu spájania
- getTarget() — cieľový objekt
- getThis() — zdrojový objekt
- getStaticPart() — statická časť informácií, t.j. `thisJoinPointStaticPart`

Prostredníctvom referencie `thisJoinPointStaticPart` môžeme zistiť statickú časť informácií o aktuálnom bode spájania prostredníctvom nasledujúcich metód:

- getKind() — typ bodu spájania
- getSignature() — signatúra bodu spájania
- getSourceLocation() — informácia o umiestnení v zdrojovom texte

Prostredníctvom referencie `thisEnclosingJoinPointStaticPart` môžeme zistiť statické informácie o zahŕňajúcom bode spájania, t.j. bode spájania, ktorý obsahuje daný bod spájania. Napríklad pre volanie metódy je to vykonávanie metódy, v rámci ktorej bola zavolaná.

16.11 Inštanciácia aspektov

V predchádzajúcich častiach to nebolo príliš zdôrazňované, ale inštancie aspektov sa nedajú priamo vytvárať. Inštancie aspektov predsa vznikajú a aspekty môžu mať stav daný ich atribútmi. Ako už bolo povedané, tak ako triedy, aj aspekty môžu obsahovať atribúty a metódy. Aspekty sa aj inicializujú podobne ako triedy:

- konštruktormi
- blokom príkazov
- statickým blokom príkazov

Vlastnosti inštanciácie apektov budú vysvetlené prostredníctvom príkladov. Aspekty v príkladoch budú operovať nad triedou `C`:

```
public class C {
   int i;

   int m() {
      return i * i;
```

```
        }

        public static void main(String[] args) {
            int a = new C().m();
            int b = new C().m();
            int c = new C().m();
        }
    }
```

Konkrétne aspekty môžu mať len konštruktor bez parametrov:

```
    public aspect B {
        public B() {
            System.out.println("Aspekt B");
        }

        Object around(C t): execution(* C.m()) && this(t) {
            System.out.println("B: " + this);
            return proceed(t);
        }
    }
```

Videnie v aspekte `B` sa vyvolá trikrát. Zakaždým sa vypíše rovnaká referencia aspektu **this**.

Abstraktné aspekty môžu mať aj konštruktory s parametrami:

```
    public abstract aspect A {
        protected int n;

        public A(int i) {
            n = i;
        }
        public A(int i, int j) {
            n = i * j;
        }
    }
```

Konkrétne aspekty potom môžu vyvolať konštruktor, ktorý potrebujú:

```
    public aspect B extends A {
        public B() {
            super.A(25);
```

```
    }
}
```

Inštancie aspektov sa nedajú priamo vytvárať, ale dá sa regulovať ako sa vytvárajú, t.j. spôsob inštanciácie.[10] Presnejšie ide o to v spojitosti s čím sa inštancie aspektov vytvárajú — preto sa hovorí aj o druhoch asociácie aspektov. Spôsoby inštanciácie aspektov sú nasledujúce:

- pre celý program, t.j. virtuálny stroj (implicitne) — **issingleton**()
- pre objekt — **perthis**() a **pertarget**()
- pre tok riadenia — **percflow**() a **percflowbelow**()
- pre typ — `pertypewithin`()

Spôsob inštanciácie sa špecifikuje za názvom aspektu. Aspekt dedí spôsob inštanciácie od nadaspektu a nemôže ho zmeniť. Syntax pre špecifikáciu inštanciácie — s výnimkou aspektu pre typ (pozri časť 16.11) — je vo všeobecnosti nasledujúca (hranaté zátvorky označujú voliteľnosť):

```
aspect Aspect1 [inst_specifier([pointcut()])] {
    . . .
}
```

Jednočlenný aspekt

Implicitný spôsob inštanciácie je jednočlenný aspekt (singleton). Možno ho uviesť aj explicitne:

```
aspect Aspect1 issingleton() {
    . . .
}
```

Pri jednočlennom aspekte sa vytvorí presne jeden aspekt daného typu. Inštanciácia sa odohrá, keď sa dosiahne prvý bod spájania zachytený niektorým z bodových prierezov príslušného aspektu.

[10] Inštancie tried sa volajú objekty. Inštancie aspektov sa volajú — aspekty! Ak z kontextu nie je jasné o čo ide, lepšie je inštancie aspektov explicitne takto označiť.

Aspekt pre objekt

Keď potrebujeme v inštancii aspektu udržiavať stav pre každý dotknutý objekt, máme k dispozícii dva spôsoby inštanciácie:

- `perthis`() — pre vykonávaný objekt
- **pertarget**() — pre cieľový objekt

V zátvorkách sa uvedie bodový prierez. Inštancia aspektu sa vytvorí pre každý zachytený vykonávaný alebo cieľový objekt pri príslušnom bode spájania bodového prierezu uvedeného v zátvorkách.

Na demonštráciu inštanciácie typu **perthis**() môže poslúžiť tento aspekt:

```
public aspect X perthis(this(C)) {
   Object around(C t): execution(* C.m()) && this(t) {
      System.out.println("B: " + this);
      return proceed(t);
   }
}
```

Pri pripojení tohto aspektu k triede `C` vypíšu sa tri rôzne referencie na aspekt. Namiesto referencie **this** sme pri výpise mohli použiť aj volanie

```
aspectOf(t)
```

Ide o statickú metódu aspektu, ktorá vráti referenciu aspektu svojho parametra. Pre jednočlenný aspekt je táto metóda bez parametrov.

Bodový prierez pre inštanciáciu pri abstraktnom aspekte nemusíme určiť hneď. Môžeme ho definovať ako abstraktný bodový prierez:

```
public abstract aspect Abc perthis(mypoints()) {
   abstract pointcut mypoints();
   . . .
}
```

Konkrétne aspekty tak predsa môžu ovplyvniť inštanciáciu, aj keď predsa nemôžu zmeniť jej typ:

```
public aspect AbcSpec {
   pointcut mypoints(): call(* C.m(..));
   . . .
}
```

Aspekt pre tok riadenia

Okrem inštanciácie aspektov pre objekty, inštanciácia môže byť viazaná aj na tok riadenia. Jestvujú dva druhy takejto inštanciácie — analogicky k bodovým prierezom `cflow`() a **cflowbelow**():

- **percflow**(),
- **percflowbelow**().

V zátvorkách sa tiež uvedie bodový prierez ako pri aspektoch pre objekty. Inštancia aspektu sa vytvorí pre každý zachytený tok riadenia pri (**cflow**()) alebo pod (**cflowbelow**()) zachyteným bodom spájania.

O toku riadenia možno uvažovať ako o konceptuálnom objekte. Inštancie aspektov pre toky riadenia možno potom využiť pri manažmente transakcií. Abstraktný aspekt pre sledovanie transakcií by mohol vyzerať takto:

```
public abstract aspect TransactionManagement
   percflow(transacted()) {
   abstract pointcut transacted();
   . . .
}
```

Pre operácie nad bodmi upresníme bodový prierez `transacted`() tak, aby vznikla inštancia aspektu pre každú sledovanú operáciu nad bodom:

```
public aspect PointTransactionManagement
   extends TransactionManagement {
   pointcut transacted(): execution(* Point+.set*(..)) ||
       execution(void Point+.get*());
   . . .
}
```

Aspekt pre typ

Niekedy nie je potrebné mať aspekt pre každú inštanciu nejakého typu, ale stačí jeden aspekt pre typ ako taký. Napríklad môžeme potrebovať aspekt pre každý typ v určitom balíku:

```
aspect AspectT pertypewithin(myPackage..*) {
   . . .
}
```

V zátvorkách sa na rozdiel od iných typov inštanciácie neuvádza bodový prierez, ale signatúra typu.

16.12 Aspektovo-orientovaná implementácia vzoru Observer

V kapitole 15 sme sa dotkli problematiky objektovo-orientovaných návrhových vzorov. Návrhové vzory vnášajú do aplikačnej logiky vlastné prvky, ktoré predstavujú inú záležitosť — logiku vzoru. V kóde vzoru potom možno sledovať zapletenie a roztrúsenie kódu. Hannemann a Kiczales implementovali vzory GOF aspektovo-orientovaným spôsobom [HK02].[11] V tejto časti rozoberieme aspektovo-orientovanú implementáciu vzoru Observer trochu inak ako u Hannemanna a Kiczalesa. Implementácia bude demonštrovaná na príklade aplikácie na zobrazenie teploty vymeranej rôznymi typmi teplotných senzorov. Spôsoby zobrazenia teploty sú rôzne — na digitálnom, analógovom alebo rozsahovom displeji — pričom možno očakávať potrebu zobrazenia aj na ďalších typoch displejov. Objektovo-orientovaná implementácia vzoru Observer na rovnakom príklade bola uvedená v časti 15.6.

Použijeme aspektovo-orientované programovanie na oddelenie logiky vzoru od aplikačnej logiky. To znamená, že v pôvodnej implementácii treba vyčleniť nasledujúce časti:

- časti rozhraní `TempDisplay` a `TempSensor` a ich implementácia
- logika spojenia displeja s príslušným senzorom
- obnovenie displejov pri zmene hodnoty v senzore

Aplikačnú logiku naopak oslobodíme od prvkov vzoru. Senzor má byť len senzorom a nemá sa starať o displeje, ktoré sú k nemu pripojené:

```
public interface TempSensor {
   double readTemp();
   void measureTemp();
}

public class HumanTempSensor implements TempSensor {
   private double temp;
   double refreshRate;
   public double readTemp() {
      return temp;
   }
   public void measureTemp() {
      // zistí teplotu z fyzickej jednotky
   }
```

[11] pozri `http://www.cs.ubc.ca/~jan/AODPs/`

```
    void setTempDebug(double t) {
        temp = t;
    }
}
```

Implementácia displejov zostáva nezmenená. Tu je zopakovaná pre úplnosť. Jedným z typov displejov je digitálny displej:

```
public class DigitalTemp implements TempDisplay {
    private HumanTempSensor sensor;
    private float temp;
    public DigitalTemp(HumanTempSensor s) {
        sensor = s;
    }
    public void refresh() {
        temp = (float)sensor.readTemp();
    }
    public void display() {
        System.out.println(
            Math.round(temp * 100.0) / 100.0);
    }
    public void measureTemp() {
        sensor.measureTemp();
    }
}
```

Rozsahový displej rozdeľuje teploty do týchto troch pásiem, ktorých názvy sú definované nasledujúcim rozhraním:

```
public enum TempRange { LOW, NORMAL, HIGH }
```

Implementácia samotného rozsahového displeja je nasledujúca:

```
public class RelTemp implements TempDisplay {
    private HumanTempSensor sensor;
    TempRange range;
    double high = 37.0;
    double low = 35.0;
    public RelTemp(HumanTempSensor s) {
        sensor = s;
    }
    public void refresh() {
```

```
        double temp = sensor.readTemp();

        if (temp <= low)
            range = TempRange.LOW;
        else if (temp >= high)
            range = TempRange.HIGH;
        else
            range = TempRange.NORMAL;
    }
    public void display() {
        switch (range) {
            case TempRange.LOW:
                System.out.println("LOW");
                break;
            case TempRange.HIGH:
                System.out.println("HIGH");
                break;
            default:
                System.out.println("NORMAL");
        }
    }
    public void measureTemp() {
        sensor.measureTemp();
    }
}
```

Logiku vzoru Observer zabezpečí aspekt:

```
public aspect TempObserver {
    private List TempSensor.displays = new ArrayList();
    public void TempSensor.addDisplay(TempDisplay d) {
        displays.add(d);
    }
    public void TempSensor.removeDisplay(TempDisplay d) {
        // . . .
    }
    public void TempSensor.notifyDisplays() {
        for (int i = 0; i < displays.size(); i++) {
            ((TempDisplay)displays.get(i)).refreshTemp();
        }
    }
    public void TempDisplay.refreshTemp() {
        refresh();
    }
```

```
    after(TempSensor sensor): this(sensor) &&
       (execution(* TempSensor+.measureTemp(..)) ||
        execution(* TempSensor+.setTempDebug(..))) {
       sensor.notifyDisplays();
    }
}
```

Aspekt `TemObserver` zavedie zoznam displejov, metódy na jeho správu do teplotného senzora, ako aj metódu na notifikáciu displejov o zmene stavu senzora. Podobne zavedie aj metódu na aktualizáciu stavu displeja. Jadrom aspektovo-orientovanej implementácie vzoru je videnie, ktoré zabezpečí notifikáciu displejov pri každom meraní teploty. Pre účely ladenia to zahŕňa aj metódu na priame nastavenie teploty.

Takto je logika vzoru Observer sústredená na jednom mieste. Pomerne jednoducho by sa všeobecne použiteľné časti aspektu `TempObserver` dali vyčleniť do abstraktného aspektu, od ktorého by sa odvádzali konkrétne aspekty prispôsobené danej doméne.

Literatúra

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, second edition, 1994.

[Bra] Gilad Bracha. Generics in the Java programming language. Tutorial, July 2005. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[Cop05] James O. Coplien. Culture of patterns. *Computer Science and Information Systems Journal (ComSIS)*, 1(2):1–26, 2005.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982. EWD 447, `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`.

[Eck02] Bruce Eckel. *Thinking in Java.* Prentice Hall, third edition, 2002. Electronic edition, revision 4.0 (final print version).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Prentice Hall, third edition, 2005. `http://java.sun.com/docs/books/jls/`.

[Hel] Dean Helman. Model-View-Controller. `http://ootips.org/mvc-pattern.html`. comp.object news transcript, May 14, 1998.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*, pages 161–173, Seattle, USA, November 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer.

[Kuh70] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, 1970.[12]

[Lis87] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, 1987.

[Mar96a] Robert C. Martin. The liskov substitution principle. *C++ Report*, 8(3):14, 16–17, 20–23, 1996. `http://www.objectmentor.com/resources/articles/lsp.pdf`.

[Mar96b] Robert C. Martin. The open-closed principle. *C++ Report*, 8(1):37–43, 1996. `http://www.objectmentor.com/resources/articles/ocp.pdf`.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Náv96] Pavol Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.

[Vra02] Valentino Vranić. Towards multi-paradigm software development. *Journal of Computing and Information Technology (CIT)*, 10(2):133–147, 2002.

[12] V češtine vyšlo ako *Struktura vědeckých revolucí*, OIKYMENH, 1997.

Register

!, 27, 170
!=, 25, 27
&, 27
&&, 28
*, 27, 171
+, 25, 27, 29, 49, 142
++, *viď* inkrementácia
+=, 25
-, 27, 142
-, *viď* dekrementácia
.., 171
/, 27
<, 27
<=, 27
=, 25
==, 25, 27
>, 27
>=, 27
?, *viď* náhradný znak pre typ
#, 142
&&, 170
%, 27
~, 142
||, 28, 170
|, 27

abstract, 56, 185
Abstract Window Toolkit, 126
abstrakcia, 8, 55, 65
  kvalita, 65
abstraktná metóda, 56
abstraktná trieda, 55
  v UML, 143
abstraktný typ údajov, 72
accept(), 106, 153
access modifiers/specifiers, *viď* modifikátory prístupu
ActionListener, 130
actor, *viď* účastník
adaptér, 133
adaptívne programovanie, 169
add(), 100
adresár, 105
  filtrovaný obsah, 106
advice, *viď* videnie
advice(), 176
after(), 179
  kontext, 181
agregácia, 5, 43
  ako hierarchizácie, 69
  hodnotou, 43
  referenciou, 43
  v diagrame tried, 140, 142
ajc, 169
Alexander, Christopher, 149
Algol 60, 3
alternatívny tok, 145
alternate flow, *viď* alternatívny tok
analýza, 137
analytický vzor, 150
anonymná trieda, 76, 79
  prijímač udalostí, 132
anonymous class, *viď* anonymná trieda
applet, 126
architektúra, 149
architektonický vzor, 150
ArgoUML, 138
args(), 176
argument, *viď* parameter
around(), 180
array, *viď* pole
ArrayList, 94
asociácia, 140
asociačná rola, 142
AspectJ, 163, 169

`aspectOf()`, 190
aspekt, 169
 abstraktný, 185
 inštancia, 169, 187
 inicializácia, 187
 jednočlenný, 189
 pre objekt, 190
 pre tok riadenia, 191
 pre typ, 191
 singleton, 189
 spôsob inštanciácie, 189
aspektovo-orientované programovanie, 163
atomický komponent, 126
atribút, 4, 15
 v UML, 141
`attach()`, 158
autoboxing, *viď* automatické balenie hodnôt
automatické balenie hodnôt, 102
AWT, *viď* Abstract Window Toolkit

bajtkód, 11
balík, 21
 implicitný, 21
 zavedenie, 21
basic flow, *viď* základný tok
before`()`, 178
behavior diagrams, *viď* diagramy správania
binding, *viď* viazanie
bod spájania
 exponovaný, 168, 170
 kontext, 168, 170, 181
bodový prierez, 168–170
 abstraktný, 185
 cieľový objekt, 176
 lexikálna štruktúra, 175
 parametre, 176
 podmienený, 177
 pomenovaný, 177
 prístup k atribútom, 173
 primitívny, 168
 tok riadenia, 174
 videnie, 176
 vykonávaný objekt, 176
body rozšírenia, 146
body spájania, 168
 reflektívna podpora, 186

Booch, Grady, 138
`boolean`, 15
buffer, *viď* vyrovnávacia pamäť
`BufferedInputStream`, 109, 113
`BufferedOutputStream`, 113
`BufferedReader`, 108, 112
`BufferedWriter`, 112
`byte`, 15
bytecode, *viď* bajtkód

C++, 64
 šablóny, 73
 modul, 68
 viacnásobné dedenie, 68
call`()`, 171, 173
CASE, 138
catch, 83
cesta, 105
cesta k triedam, 22
cflow`()`, 175
cflowbelow`()`, 175
`char`, 15
Class,
 idClass90
`Class`
 parametrizovaná, 102
class, *viď* trieda
class (literál),
 kwclass (literál)90
classpath, *viď* cesta k triedam
CLOS, 64, 73
`close()`, 107
code scattering, *viď* roztrúsenie kódu
code tangling, *viď* zapletenie kódu
cohesion, *viď* súdržnosť
collection, *viď* zoskupenie
Collections Framework, 93
command line interface, *viď* príkazový riadok
`Constructor`, 90
container, *viď* kontejner, *viď* kontejner
Coplien, James O., 150, 152
coupling, *viď* zviazanosť
critical section, *viď* kritickú oblasť
crosscutting concerns, *viď* pretínajúce záležitosti

číselná promócia, 30

démon, 121
daemon, *viď* démon
Dahl, Ole-Johan, 3, 64
`DataInputStream`, 110, 113
`DataOutputStream`, 113
dedenie, 8, 43
  ako konkretizácia, 68
  kritérium použitia, 69
  pri aspektoch, 185
  v UML, 143
dekrementácia, 27
`delete()`, 106
design by contract, *viď* návrh podľa zmluvy
design pattern, *viď* návrhový vzor
`detach()`, 158
Dia, 139
diagram, 139
  objektov, 144
  prípadov použitia, 140
  sekvencií, 140, 146
  tried, 140
diagram prípadov použitia, 145
diagram správania, 139
diagramy štruktúry, 139
dialóg, 126
  modálny, 127
Dijkstra, Edsger W., 164
`double`, 15
double dispatch, 73, 157
downcasting, 57, 93
  neúspešný, 58

Eclipse, 12
  a AspectJ, 169
encapsulation, *viď* zapuzdrenie, 66
event, *viď* udalosť
event handling, *viď* spracovanie udalostí
Event-Dispatching Thread, *viď* niť na odosielanie udalostí
`Exception`, 83, 85
exception, *viď* výnimka
exception handler, *viď* ošetrenie výnimiek
execution`()`, 173
`exists()`, 106
`«extend»`, 145, 146
extends, 8
  ohraničenie náhradného znaku, 96
  zavedenie, 185
`Externalizable`, 117

`Field`, 90
field, *viď* atribút
`File`, 105
`FileInputStream`, 114
`FilenameFilter`, 106
`FileOutputStream`, 114
`FileWriter`, 112
final
  lokálne a anonymné triedy, 81
final, 49
finalizácia, 37
finally, 83
`float`, 15
for
  rozšírená slučka (for-each), 98, 99
formátovaný výstup, 109

Gamma, Erich, 152
garbage collector, *viď* zberač smetí
generalizácia, *viď* zovšeobecnenie
  v diagrame tried, 140
generické zoskupenie, 94
generickosť, 94
  kontrola typov, 96
get`()`, 173
`getArgs()`, 187
`getBytes()`, 115
`getKind()`, 187
`getSignature()`, 187
`getSourceLocation()`, 187
`getStaticPart()`, 187
`getTarget()`, 187
`getThis()`, 187
GoF, 150, 152
grafické používateľské rozhranie, *viď* GUI
GUI, 125
  a Model-View-Controller, 151
  generátory, 126

handler`()`, 174
Helm, Richard, 152
hierarchia, 5, 8, 68

agregácia, 43, 143
v UML, 143
Hillside Group, 149
Hoare, C. A. R., 50

idióm, 150
if(), 177
if-else, 28
implementácia, 137
implements, 58
zavedenie, 185
implicitný balík, 23
import, 21
inštancia, 4
aspektu, 187
«include», 145, 146
inheritance, *viď* dedenie
inicializácia, 33
blokom príkazov, 36
explicitná, 33
finálnych atribútov, 49
implicitná, 34
konštánt, 50
nadtriedy, 48
initialization(), 174
inkrementácia, 27
inkrementálny, 65
inkrementálny vývoj, 137
inner classes, *viď* vnútorné triedy
InputStream, 107, 116
InputStreamReader, 109
instanceof, 89
int, 15
Integer, 14
inter-type declarations, *viď* medzitypové deklarácie
medzitypová deklarácia, 169
interface, 58
interrupt(), 121
introductions, *viď* zavedenia
invokeAndWait(), 134
invokeLater(), 134
IOException, 108
is-like-a, 69
isDirectory(), 107
isFile(), 107
isHidden(), 107
isInstance(), 89
issingleton(), 189
iterátor, 97
Iterable, 98
iteratívny, 65
iteratívny vývoj, 137
iterator(), 97

J2SDK, 11
Jacobson, Ivar, 138
JApplet, 126
JAR, 116
Java, 11
API, 11
pôvod slova, 12
platformy, 11
prostredia, 12
výslovnosť, 12
verzie, 12
Java Foundation Classes, 126
Java Runtime Environment, 11
Java Virtual Machnine, 11
java.io, 105, 107
java.nio, 105, 114
Java2D, 126
JavaBeans, 126
javac, 20
JavaScript, 64
JBuilder, 13
JComponent, 126
JContainer, 126
JDialog, 126
JDK, 12
JFrame, 126, 127
EXIT_ON_CLOSE, 127
HIDE_ON_CLOSE, 128
pridanie komponentu, 128
JLabel, 134
Johnson, Ralph, 152
join points, *viď* body spájania
join(), 121
JRE, *viď* Java Runtime Environment
JVM, *viď* Java Virtual Machine
JWindow, 126

kanál, 105, 114
násobnosť, 142

komentár, 24
komponent, 126
  realizácia vo Swingu, 133
  Swingu, 126
  vytvorenie vo Swingu, 128
kompozícia, *viď* agregácia
  obsahovanie hodnotou, 43
  v diagrame tried, 140, 142
kompozičné filtre, 169
kompresia, 116
konštanta
  statická, 50
konštruktor, 33, 35
  aspektu, 188
  v dedení, 48
  v UML, 142
konkretizácia, 55, 65
kontejner, 93, 126
  sprostredkovateľský, 127
kritickú oblasť, 121
Kuhn, Thomas, 3, 163

label, *viď* označenie
ladenie tried, 82
`LayoutManager`, 126, 129
Liskovej princíp substitúcie, 69
Lisp, 64
`List`, 94, 100
list, *viď* zoznam
listener, *viď* prijímač
literál, 30
lokálna trieda, 76
  preklad, 78
`long`, 15

Macintosh, 125
main(), 122
`main()`, 20
map, *viď* tabuľka
medzitypová deklarácia, 169
medzitypové deklarácie, 182
metóda, 15
  abstraktná, 59
  generická, 100
  s premenlivým počtom parametrov, 40
  synchronizovaná, 121
  výber tela, 53
  viazanie, 53, 72
`Method`, 90
mixin triedy, 68
`mkdirs()`, 107
množina, 93
Model-View-Controller, 151
  a Observer, 162
modelovanie, 137
modifikátory prístupu, 23
  v UML, 142
modul, 68
modulárnosť, 68
modularizácia, 164
`MouseListner`, 132
MS Visio, 139
multimetody, 73
multiple inheritance, *viď* viacnásobné dedenie
multiplicity, *viď* násobnosť
multithreading, *viď* viacniťovosť
MVC, *viď* Model-View-Controller

náhradný znak
  v bodových prierezoch, 171
náhradný znak pre typ, 94
  ohraničený, 95
násobnosť, 142
návrh, 137
návrh podľa zmluvy, 72
návrhový vzor, 150
  GoF, 150
nadtrieda, 44
nadtyp, 51
  rozhranie, 59
nestatická členská trieda, 76
nested type, *viď* vhniezdený typ
NetBeans, 13
`new`
  v bodových prierezoch, 172
new, 9, 16
niť, 119
  reprezentácia, 120
  riadenie, 120
  vytvorenie, 119
`synchronizácia`, 121
niť na odosielanie udalostí, 133
`notify()`, 121, 158

Nygaard, Kristen, 3, 64

ošetrenie výnimiek, 84
obaľovacia trieda, 14
`Object`, 49
 v zoznamoch, 93
`ObjectInputStream`, 117
`ObjectOutputStream`, 117
objekt, 4
 interakcia objektov, 5
 rušenie, 19
 vykonateľný, 134
objektovo-orientované mechanizmy, 63
`Observable`, 162
Observer, 158
 a Model-View-Controller, 162
 aspektom, 192
`Observer`, 162
oddelenie záležitostí, 164
 pretínajúcich, 164
okno, 126
 priame vytvorenie, 127
 rozloženie prvkov, 129
 skrytie, 127
 vytvorenie dedením, 128
 zavretie, 127
OMG, 138
OMONDO EclipseUML Studio, 138
open-closed principle, *viď* princíp otvorenosti a uzavretosti
operácia, 141
 abstraktná, 143
operátor, 25
 aritmetický, 27
 logický, 27
 na bitoch, 28
 priradenia, 25
 relačný, 27
 ternárny, 28
 unárny, 27
 zmeny typu, 29
`out`, 20
`OutputStream`, 107, 116
overloading, *viď* preťaženie
overriding, *viď* prekonávanie
označenie, 126
`pack()`, 134
package, *viď* balík
package, 21
package access, *viď* prístup v rámci balíka
 v UML, 142
`paint()`, 133
paradigma, 3, 163
parameter, 16, 32
PARC, 125
 AOP, 168
`parseInt()`, 84
pathname, *viď* cesta
percflow`()`, 191
percflowbelow`()`, 191
pertarget`()`, 190
perthis`()`, 190
perzistencia
 podpora v Java API, 116
PLoP, 150
podtrieda, 44
podtyp, 9
pointcut, *viď* bodový prierez
pole, 10, 38
 ako parameter metódy, 40
 operácie, 41
 viacrozmerné, 41
polimorfizmus
 viacnásobný v Jave, 157
polymorfizmus, 9, 53
 parametrický, 73
 preťaženie, 73
 viacnásobný, 73
 zdanlivý pri zoznamoch, 94
Poseidon for UML, 138
postconditions, 72
používateľské rozhranie, 125
 a prípady použitia, 146
prúd, 105, 107
 bajtov, 107
 neuzatvorenie pri súboroch, 112
 obalený, 108
 otvorenie, 107
 zatvorenie, 107
 znakov, 107
príkazový riadok, 125
prípad použitia, 144
prístup v rámci balíka, 23, ŬML142

pravidlo jednej nite, 133
porušenie, 134
preťaženie, 31, 73
konštruktorov, 36
pri dedení, 46
preconditions, 72
preinitialization(), 174
preklad, 20
prekonávanie, 46
private metódy, 50
zabránenie, 50
premature optimization, 50
Presentation-Model, 151
pretínajúce záležitosti, 68, 164
pretypovanie, 93
prijímač, 130
rozhranie, 133
prijímač udalostí, 126
primitívny typ, 14
vytvorenie premennej, 14
princíp otvorenosti a uzavretosti kódu
a Visitor, 157
princípu otvorenosti a uzavretosti, 67
printf(), 109
println(), 20
PrintStream, 20, 109
PrintWriter, 112
priradenie, 25
private
v UML, 142
private, 23, 48
proceed(), 180
protected
v UML, 142
protected, 23, 48
public
v UML, 142
public, 23

random access file, *viď* súbor
RandomAccessFile, 107, 113, 114
Rational Rose, 138
Rational Software Architect/Modeler, 138
reťazec, 13, 29
ako prúd, 110
spájanie, 49
Reader, 107
Reenskaug, Trygve M. H., 151
referencia, 13
inicializácia, 13
reflexia, 90
pri bodoch spájania, 186
Remote Method Invocation, *viď* volanie vzdialených metód
renameTo(), 107
return, 17
RMI, *viď* volanie vzdialených metód
rozhranie, 58
atribúty, 59
dedenie, 60
implementácia abstraktných metód, 59
implementácia viac rozhraní, 59
metódy, 59
prijímača, 130, 131
v UML, 143
rozsah, 19
roztrúsenie kódu, 164
RTTI, 57, *viď* Runtime Type Identification
Rumbaugh, Jim, 138
run(), 119
Runnable, 119, 134
Runtime Type Identification, 89
RuntimeException, 84

súbor, 105
binárny, 112
s voľným prístupom, 113
zápis a čítanie, 111
zobrazený do pamäte, 115
súdržnosť, 65, 68
scope, *viď* rozsah
Self, 64
separation of concerns, *viď* oddelenie záležitostí
serializácia, 116
serializácia objektov, 105, 116
Serializable, 116
set, *viď* množina
set(), 173
setDeamon(), 121
setDefaultCloseOperation(), 127
setPriority(), 121
short, 15
show(), 134

signatúra, 171
silne typový jazyk, 72
Simula 67, 64
Simula 67, 3
Single-Thread Rule, *viď* pravidlo jednej nite
skener, 109
skrývanie, 47, 56
    statických metód, 47
skrývanie informácií, *viď* zapuzdrenie
`sleep()`, 121
Smalltalk, 3, 64, 151
správanie, 44
spracovanie udalostí, 130
`start()`, 119
staticinitialization`()`, 174
statická členská trieda, 76, 81
statická metóda, 56
stereotyp, 142
stream, *viď* prúd
`String`, 13
`StringReader`, 110, 112
structure diagrams, *viď* diagramy štruktúry
subclass, *viď* podtrieda
subjektovo-orientované programovanie, 169
super, 46–48
    ohraničenie náhradného znaku, 96
superclass, *viď* nadtrieda
Swing, 125, 126
synchronizácia nití, 121
`synchronized`, 121
`System`, 20, 107, 108

špecializácia, 8, 65, 140
štandardný V/V systém, 108
štandardný výstup, 107, 108
štandardný výstup pre chyby, 107, 109
štandardný vstup, 107, 108
štruktúra, 44

tabuľka, 93
target`()`, 176
textové pole, 126
this, 36
this`()`, 176
thisEnclosingJoinPointStaticPart, 186
thisJoinPoint, 186
thisJoinPointStaticPart, 186
`Thread`, 119
thread, *viď* niť
throw, 83
`Throwable`, 83
throws, 83
tlačidlo, 126
    kliknutie, 131
    pohyb myšou nad, 132
    vytvorenie, 128
tok
    v prípadoch použitia, 145, 147
`toString()`, 49
`transient`, 118
trieda, 4
    ako abstrakcia, 65
    generická, 100
    odvodená, 43
    v UML, 141
try, 83
try-catch-finally, 85
typ, 4, 5, 19, 23, 44, 51, 72, 75
    čistý (raw), 97
    generický, 95
    identifikácia, 89
    identifikácia v čase vykonávania, 57
    kolízia názvov, 21
    parametrizovaný, 94
    plne vymedzený názov, 21
    primitívny, 14
    vymenovaný (enumerated), 98
typovosť, 72

účastník, 144
udalosť, 126, 130
    kliknutie tlačidla, 131
    pohyb myšou, 132
    vo vzore Model-View-Controller, 151
UML, 138
UMLet, 139
Unified Modeling Language, *viď* UML
upcasting, 51, 53, 57
user interface, *viď* používateľské rozhranie
`user.dir`, 105
UTF, 112

V/V systém Javy, 105
výnimka, 83

kontrolovaná, 84
nekontrolovaná, 84
opätovné vyhodenie, 87
„prehltnutie", 88
prenos na výstup, 88
`valueInt()`, 14
`values()`, 99
vhniezdený typ, 75
v UML, 144
viacnásobné dedenie, 68
viacniťovosť, 119
viazanie, 53, 72
videnie, 169, 178
viditeľnosť v UML, 142
`visit()`, 153
Visitor, 153
vlákno, *viď* niť
Vlissides, John, 152
vnútorná trieda, 76
inštancie, 77
objekt, 78
použitie, 132
vnorená dokumentácia, 24
`void`, 15
volanie vzdialených metód, 116
vtkanie, 169
vyšší kontext, 83, 86
vyrovnávacia pamäť, 105, 114
vzor, 149

`wait()`, 121
`vtkanie`, 169
WIMP, 125
within(), 175
withincode(), 175
wrapper, *viď* obaľovacia trieda
`Writer`, 107

`yield()`, 120

základný tok, 145
závislosť, 140
«use», 144
zapletenie kódu, 164
zapuzdrenie, 7, 66
zavedenia, 182
zavedenie členského prvku, 183
zavedenie statického prvku, 21
zavedenie vzťahu dedenia, 185
zberač smetí, 19
ZIP, 116
zoskupenie, 93
zovšeobecnenie, 7, 65
zoznam, 93
zviazanosť, 65
vo vzore Model-View-Controller, 151

Autor:	Ing. Valentino Vranić, PhD.
Názov:	Objektovo-orientované programovanie: Objekty, Java a aspekty
Vydanie:	prvé
Náklad:	600 výtlačkov
Rozsah:	221 strán, 30 obrázkov
Edičné číslo:	
Tlač:	Vydavateľstvo STU v Bratislave
Vytlačené:	2008

ISBN 978-80-227-2830-0