RUDOLF PECINOVSKÝ

Java 5.0

Novinky jazyka a upgrade aplikací

Největší inovace v historii Javy a co znamená pro vaše aplikace

Import statických objektů ▪ Zdokonalený příkaz for ▪ Automatické převody typů ▪ Metody s proměnným počtem parametrů ▪ Výčtové typy ▪ Anotace ▪ Rozšíření znakové sady ▪ Převod aplikací na verzi 5.0

www.cpbooks.cz

computer press

CP Books

Svolení s vystavením PDF

Nakladatelství **Computer Press, a.s**. svoluje s vystavením PDF knihy K0971 *Java 5.0 Novinky jazyka a upgrade aplikací* na serveru Java.cz. Doufáme, že se stane užitečným nástrojem pro každého čtenáře v jeho každodenní praxi.

O autorovi:

Rudolf Pecinovský patří k našim špičkovým odborníkům na výuku programování. Publikoval již 35 učebnic, jež byly přeloženy do pěti jazyků. Učí programování na VŠE a současně pracuje jako Senior EDU Expert ve firmě ICZ a.s.

Rudolf Pecinovský

Java 5.0

Novinky jazyka a upgrade aplikací

CP Books, a.s.
Brno
2005

Java 5.0
Novinky jazyka a upgrade aplikací

Rudolf Pecinovský

Copyright © CP Books, a.s. 2005. Vydání první. Všechna práva vyhrazena.
Vydalo nakladatelství CP Books, a.s. jako svou 1835. publikaci.

Vydavatelství a nakladatelství CP Books, a.s.,
nám. 28. dubna 48, 635 00 Brno, http://www.cpbooks.cz

ISBN 80-251-0615-2

Prodejní kód: K0971

Odborná korektura: Bogdan Kiszka
Jazyková korektura: Ivo Magera
Vnitřní úprava: CP Books
Sazba: Bogdan Kiszka
Rejstřík: Rudolf Pecinovský
Obálka: Martin Sodomka

Komentář na zadní straně obálky: Ivo Magera
Technická spolupráce: Jiří Matoušek, Tomáš Zeiner
Odpovědný redaktor: Ivo Magera
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Žádná část této publikace nesmí být publikována a šířena žádným způsobem a v žádné podobě bez výslovného svolení vydavatele.

CP Books, a.s., nám. 28. dubna 48, 635 00 Brno
tel.: 546 122 111, fax: 546 122 112
Objednávejte na: **www.cpbooks.cz**
distribuce@cpress.cz

Bezplatná telefonní linka: **800 555 513**

Dotazy k vydavatelské činnosti směřujte na: **knihy@cpress.cz**

Máte-li zájem o pravidelné zasílání informací o knižních novinkách do Vaší e-mailové schránky, zašlete nám zprávu, obsahující váš souhlas se zasíláním knižních novinek, na adresu **novinky@cpress.cz.**

Novinky k dispozici ve dni vydání, slevy, recenze, zajímavé programy pro firmy i koncové zákazníky.

http://www.vltava.cz

Nejširší nabídka literatury, hudby, MP3, multimediálního softwaru a videa za bezkonkurenční ceny.

Obsah

Úvod 7

Java a její verze 7
Komu je kniha určena 8
Doprovodné programy 8
Uspořádání knihy 9
Varovná hlášení překladače 9
Použité konvence 10

KAPITOLA 1
Import statických členů 11

Základní použití 11
Sloučení importu atributů a metod 13
Kolize importů 15

KAPITOLA 2
Rozšíření příkazu for 17

Předehra pro méně zkušené 17
Syntaxe 18
Použití při práci s poli 18
Jednorozměrné pole 19
Vícerozměrná pole 19
Kdy cyklus for(:) použít nelze 20
Použití při práci s kontejnery 20
Skrytý iterátor 22
Definice vlastní iterovatelné třídy 23

KAPITOLA 3
Automatické převody mezi primitivními a obalovými typy 27

Předehra pro méně zkušené 27
Automatické převody 28

Nebezpečí při porovnávání 29
Omezení převodů 30
Rady do života 31

KAPITOLA 4
Metody s proměnným počtem parametrů 33
Jediný parametr 33
Parametry primitivních typů 35
Více parametrů 35
Více skupin parametrů s proměnným počtem členů 36

KAPITOLA 5
Parametrizované datové typy a metody, typové parametry 39
Nejprve trocha terminologie 40
Předehra pro méně zkušené 40
PTM versus šablony jazyka C++ 41
Použití PT v programu 42
Fronta textových řetězců v dřívějších verzích 42
Fronta textových řetězců ve verzi 5.0 44
PT s několika typovými parametry 46
Definice vlastního PT 46
Definice třídy 46
Definice rozhraní 49
Potomci parametrizovaných typů 50
Definice metod s typovými parametry 53
Volání parametrizovaných metod 55
Omezení použitelných hodnot typových parametrů 57
Definice PT s několika parametry 58
Vzájemné závislosti typových parametrů 60
Parametrizace a očišťování 62
Ještě jednou terminologie 62
Očištění parametrizovaného kódu 63
Pořadí uvedení omezujících tříd a rozhraní 64
Očištění výrazů 64
Ztráta informace při běhu 65
Přemosťovací metody 65
Spolupráce s programy vytvořenými v předchozích verzích 68
Vypínání typové kontroly 69
Zakázané operace 70
Za typové parametry nelze dosazovat primitivní typy 70
Typové parametry třídy není možno použít u statických členů 71
Nelze vytvořit instanci typového parametru 71
Nelze vytvořit pole instancí typového parametru ani parametrizovaného typu 72

Výjimky 73
Nejednoznačnosti a kolize 75
Falešně přetížená metoda 75
Nová metoda koliduje se zděděnou 76
Kolize požadovaných rozhraní 77
Kolize implementovaných rozhraní 78
Špatné pochopení dědičnosti 78
Žolíky 80
Omezení hodnot žolíků 81
Žolík jako potomek zadaného typu 81
Žolík jako předek zadaného typu 83
Převod klasických tříd na parametrizované 84
Parametrizované typy a reflexe 86

KAPITOLA 6
Výčtové typy 87
Předehra pro méně zkušené 87
Historické pozadí 88
Nejjednodušší definice 88
Překladačem přidané atributy a metody 90
Třída Enum 91
Nově definované metody 92
Překryté verze metod zděděných z Object 93
Serializace 93
Použití výčtových typů v programu 93
Přepínač 93
Cyklus 96
Složitější definice výčtových typů 96
Konstanty anonymních typů 100
Nepoužitelnost lokálních atributů 104
Kontejnery EnumSet a EnumMap 107
EnumMap 107
EnumSet 107

KAPITOLA 7
Anotace (metadata) 109
Označování deklarací anotacemi 110
Anotování balíčků 113
Anotace ve standardní knihovně 113
Standardní anotace 113
Metaanotace 115
Syntaxe definice anotací 118
Jednoduchá značkovací anotace 119
Anotace s parametry 121
Zjednodušení zápisu jediného parametru 122
Anotace jako parametry jiných anotací 123

Další vlastnosti anotací 123
Získávání informací o anotacích za běhu programu 124
Získání anotací třídy a jejích členů 124
Získání hodnot anotačních metod 128
Práce s anotacemi mimo běh programu 132
Nástroj apt pro zpracování anotací ve zdrojovém kódu 132
Zpracování bajtkódu 132

KAPITOLA 8
Rozšíření znakové sady 133
Znaková sada Unicode 133
Práce se znaky 136
Znaky jako parametry a návratové hodnoty 136
Práce s řetězci 137
Shrnutí 138

KAPITOLA 9
Přechod na verzi 5.0 139
Běh programu 139
AWT 139
Java 2D Graphics 140
Bezpečnost 141
Serializace 141
Swing 142
Další změny 142
Instalace (deployment) 142
Aplety 142
Knihovny 143
Instalační programy 143
Překlad 144
Změny v API 144
Parametrizované typy 144
Nová klíčová slova 144
Změny ovlivňující používané nástroje 145
Class-soubory, vnitřní třídy, instrumented code 145
Inicializace tříd po vyhodnocení literálu X.class 145
Parametry metod zavaděče tříd 146
API pro ladění a profilaci 146

PŘÍLOHA
Varovná hlášení překladače 147
Rejstřík 151

Úvod

Nová verze Javy přišla s největší inovací v historii tohoto jazyka. Nejenom výrazně rozšířila knihovny, ale zavedla především několik nových syntaktických konstrukcí, které v mnohých situacích výrazně mění možnosti tvorby efektivního, robustního a snadno spravovatelného kódu.

Tato kniha se vás pokusí seznámit s novinkami, s nimiž přišla Java ve verzi 5.0, označované dříve jako verze 1.5. Pro mnohé „nejavisty" je číslování verzí Javy trochu záhadou. Zkusme si je proto nyní připomenout.

Java a její verze

Java se narodila v roce 1995 jako „maličký" jazyk, jehož standardní knihovna obsahovala pouhých 211 veřejných tříd a celá Java se vešla na jedinou disketu. Hned po svém uvedení sklidila velký úspěch a motivovala své tvůrce k rychlému vylepšování.

Verze 1.1, která se objevila za dva roky, zavedla vnitřní a vnořené třídy, některé knihovny výrazně upravila a počet veřejných tříd více než zdvojnásobila – rozšířila jej na 477.

Zlomem v historii jazyka byla verze 1.2, která přišla na přelomu let 1998 a 1999 se zásadním přepracováním celé knihovny a zavedením řady dalších knihoven, takže celkový počet veřejných tříd stoupl na 1524. Změny koncepce práce s jazykem považovali její tvůrci za natolik zásadní, že novou verzi jazyka i platformy začali označovat jako Java 2.

Verze 1.3 (přesněji Java 2 verze 1.3), která se objevila v roce 2000, pouze pokračovala v evoluci a přinesla další rozšíření knihovny (ta nyní obsahovala 1840 veřejných tříd).

Verze 1.4 přinesla v roce 2002 další zásadní rozšíření knihovny – počet veřejných tříd v ní dosáhl 2723. Kromě toho přišla po Javě 1.1 s druhým (i když daleko drobnějším) rozšířením syntaxe: zavedla nové klíčové slovo `assert`, které usnadnilo ověřování bezchybnosti programu a vyhledávání chyb v instalovaných programech.

Na podzim roku 2004 byla uvedena další verze, která byla zpočátku označována jako verze 1.5, ale kterou v srpnu 2004 přejmenovalo marketingové oddělení firmy *Sun* na Java 2 verze 5.0. Všechny současné materiály o ní proto hovoří jako o verzi 5.0.

Tato verze přišla vedle očekávatelného rozšíření knihovny o dalších téměř 500 veřejných tříd (knihovna má nyní 3270 veřejných tříd, počet všech tříd včetně interních a pomocných ale přesahuje 15 000) především se zásadními rozšířeními syntaxe jazyka.

Právě tato syntaktická rozšíření vedla marketingové oddělení k změně zavedeného číslování. Syntaktická rozšíření a jejich podrobný rozbor pak budou hlavní náplní této publikace.

Komu je kniha určena

Kniha je určena především těm, kteří s Javou pracují a nemají čas bloudit internetem či literaturou, zjišťovat speciální vlastnosti nových rozšíření a bádat nad tím, co všechno jim tyto novinky umožňují a kde si musí při jejich aplikaci dávat pozor.

Pokusil jsem se ji však napsat tak, aby ji mohli číst i ti z vás, kteří se nepovažují za experty jazyka, ale spíše za trochu pokročilejší začátečníky, jejichž znalosti se pohybují na úrovni běžných začátečnických učebnic jazyka (nejlépe té mojí ;-)). Na začátek řady kapitol je proto zařazena speciální podkapitola nazvaná *Předehra pro méně zkušené*, ve které se snažím připomenout pojmy, o nichž bude kapitola pojednávat.

Doprovodné programy

Suchá teorie je na nic. Málo programátorů si přečte specifikaci používaného jazyka a ještě méně jich jejímu suchému a úzkostlivě přesnému jazyku doopravdy porozumí, protože nemají čas bádat nad všemi souvislostmi.

Programátoři dávají přednost příkladům, na nichž jsou probírané rysy názorně demonstrovány. Proto jsem se i já snažil postavit celý výklad především na příkladech. Všechny příklady si můžete stáhnout na mých webových stránkách, konkrétně na adrese `http://knihy.pecinovsky.cz/java5novinky`. Na začátku každé kapitoly (a občas i samostatné pasáže) se pak dozvíte, ve kterém balíčku či třídě najdete kód, o němž se v následujícím textu hovoří.

Příklady najdete na webu ve dvou provedeních. První z nich je uloženo v souboru `Priklady_cs.zip` a obsahuje české identifikátory i názvy tříd tak, jak jsou použity ve výkladu v knize. Tuto verzi však mohou používat pouze programátoři pracující v prostředí Windows. Druhá bude uložena v souboru `Priklady_cs_bhc.zip` a budou v ní uloženy všechny soubory „odháčkované“ (bhc = bez háčků a čárek).

Původně jsem měl v úmyslu připravit ještě soubor `Priklady_cs.jar`, který by obsahoval samorozbalitelný archiv s českými soubory stáhnutelnými pod jakoukoliv platformu. Trápí mne však „tygří problém“[1], takže vám to nemohu slíbit.

Doprovodné programy jsou uspořádány tak, aby mohly být použity i jako projekty v prostředí *BlueJ*. Každá složka proto obsahuje i konfigurační soubor `bluej.pkg`,

[1] Ptali se tygra, proč žere zelí. Tygr odpověděl: „Nejsou lidi.“

v němž je uloženo rozmístění jednotlivých objektů (tříd, rozhraní, balíčků) v diagramu tříd. Používáte-li jiná vývojová prostředí, můžete tyto soubory ignorovat nebo smazat.

Uspořádání knihy

Kniha je rozdělena do osmi kapitol. Každá z prvních sedmi probírá jedno syntaktické rozšíření jazyka, poslední je pak věnována specifikám převodu starších programů na novou verzi překladače a virtuálního stroje.

Bývá dobrým zvykem v úvodu jednotlivé kapitoly vyjmenovat a prozradit čtenáři, co jej v nich čeká. Domnívám se, že tato kniha je z rodu těch, u nichž vše potřebné prozradí již obsah. Nebudu zde proto opakovat něco, co bezpečně najdete jinde (kromě toho jsem přesvědčen, že za to, že se vám tu o obsahu kapitol rozpovídám, mi nakladatel na honoráři stejně nepřidá).

Varovná hlášení překladače

Prostor, který ušetřím vynecháním popisu kapitol, věnuji zmínce o varovných hlášeních překladače (kterou v textu ještě několikrát zopakuji pro ty, kteří úvody nečtou). Nová verze překladače se totiž snaží zvýšit informovanost programátora, takže nejenom vypisuje zprávy o chybách, ale přidává k nim i varovná hlášení, v nichž jej upozorňuje na místa, která by se mohla stát příčinou chyb.

Abyste se zbavili varovných hlášení, museli byste používat parametrizované typy, ty však poznáme až v páté kapitole (dřív to nejde, protože při jejich výkladu budu používat dříve vyložené konstrukce). Do té doby, než se naučíte upravit program tak, aby překladač necítil potřebu vás varovat, se budete muset smířit s tím, že vás překladač před něčím varuje, ale vy nevíte přesně před čím.

Překladač vydává dva různé druhy varovných hlášení: souhrnné a podrobné. Implicitně jsou nastavena souhrnná varování, při nichž vás překladač pouze upozorní na to, že by vás chtěl před něčím varovat. Současně přidá doporučení, jaký parametr si máte v příkazovém řádku nastavit, aby vás mohl varovat podrobněji. Po jeho nastavení vás pak upozorňuje na jednotlivá podezřelá místa obdobně, jako vás upozorňuje na syntaktické chyby.

Doporučuji vám nastavit podrobnou verzi těchto varovných hlášení. Pro ty z vás, kteří pracují ve vývojových prostředích *BlueJ*, *Eclipse* či *NetBeans*, jsem v příloze připravil návod, jak podrobná varovná hlášení nastavit.

Použité konvence

K tomu, abyste se v textu lépe vyznali a také abyste si vykládanou látku lépe zapamatovali, používám několik prostředků pro odlišení a zvýraznění textu.

Důležité Texty, které chci z nějakého důvodu zvýraznit, vysazuji **tučně**. Většinou jsou takto zvýrazněny první výskyty nových termínů

Názvy Názvy firem a jejich produktů vysazuji *kurzivou.*

Citace Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazuji **tučným bezpatkovým písmem**.

`Adresy` Názvy souborů a internetové adresy spolu s texty programů a jejich částí vysazuji `neproporcionálním písmem`.

Kromě částí, které považuji za důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v rámečku připomínajícím stránku v kroužkovém bloku.

Poznámka:
Obyčejná poznámka, ve které jsou informace z hlavního proudu výkladu doplněny o nějakou zajímavost, nebo poznámka týkající se používané terminologie.

Programy:
Poznámka oznamující umístění doprovodných programů.

Pozor!
Upozornění na věci, které byste měli určitě vědět a dávat si na ně pozor, nebo úskalí programovacího jazyka či programů, s nimiž budeme pracovat (bude vám radit, jak se těmto nástrahám vyhnout či jak to zařídit, aby vám alespoň pokud možno nevadil).

Tip:
Různé tipy, kterými můžete vylepšit svůj program nebo zefektivnit svoji práci.

Kapitola 1

Import statických členů

Nejprve se „pro zahřátí" seznámíme s nejmírnějším syntaktickým rozšířením Javy, kterým je bezesporu statický import. Ukážeme si, jak jej používat a kde na nás při jeho používání může čekat nějaké nebezpečí.

Programy:
Všechny třídy, o nichž se v této kapitole hovoří, jsou umístěny v balíčku `rup.česky.java15.simport`.

Statický import je nejenom nejmírnějším, ale pravděpodobně také nejrozpačitěji přijímaným rozšířením. Umožňuje deklarovat, které statické atributy a metody jiných tříd bude možné v těle třídy používat bez nutnosti jejich kvalifikace.

Základní použití

Ve verzích do 1.4 včetně bylo nutno každé volání statického atributu nebo metody z jiné třídy kvalifikovat – např.:

```
1 import javax.swing.JOptionPane;
2
3 public class StatickýImport_Dříve
4 {
5   private static final double PŘESNOST = 100;
6
7   public static void dříve() {
```

```
 8     double podíl = Double.parseDouble(
 9       JOptionPane.showInputDialog(null,
10       "Zadejte požadovaný podíl" ) );
11     podíl = zaokrouhli( Math.max(.1, Math.abs(podíl)) );
12     double úhel = zaokrouhli( Math.PI / podíl );
13     double sinus = zaokrouhli( Math.sin( úhel ) );
14     String odpoved = null;
15     switch( JOptionPane.showConfirmDialog( null,
16       "Úhel = " + úhel + " radiánů" +
17       "\n\nsin( PI/" + podíl + " ) = " + sinus +
18       "\n\nBude to vyhovovat?" ) )
19     {
20       case JOptionPane.YES_OPTION:
21         odpoved = "To mne těší";
22         break;
23       case JOptionPane.NO_OPTION:
24         odpoved = "To mne mrzí";
25         break;
26       case JOptionPane.CANCEL_OPTION:
27       case JOptionPane.CLOSED_OPTION:
28         odpoved = "Ukončil jste program";
29         break;
30     }
31     JOptionPane.showMessageDialog( null, odpoved );
32   }
33
34
35   public static double zaokrouhli( double číslo ) {
36     return Math.round( PŘESNOST * číslo ) / PŘESNOST;
37   }
38 }
```

Ve verzi 5.0 stačí deklarovat import použitých statických členů, a to jak atributů, tak metod. Přitom lze použít stejné hvězdičkové konvence jako u klasického příkazu `import`.

Import statických členů se od běžného importu liší pouze tím, že za klíčové slovo `import` je třeba ještě zapsat klíčové slovo `static`. Při deklaraci importovaného členu pak postupujeme stejně jako u klasického importu, pouze za název třídy ještě připíšeme tečku a název importovaného statického členu, anebo hvězdičku oznamující import všech statických členů dané třídy.

```
1 //Všimněte si, že imporuji pouze název metody bez závorek
2 import static java.lang.Double.parseDouble;
3 import static java.lang.Math.*;
4 import static javax.swing.JOptionPane.*;
5
6
```

```
 7 public class StatickýImport_Nyní
 8 {
 9   private static final double PŘESNOST = 100;
10
11   public static void dříve() {
12     double podíl = parseDouble(
13        showInputDialog(null, "Zadejte požadovaný podíl" ) );
14     podíl = zaokrouhli( max(.1, abs(podíl)) );
15     double úhel  = zaokrouhli( PI / podíl );
16     double sinus = zaokrouhli( sin( úhel ) );
17     String odpoved = null;
18     switch( showConfirmDialog( null,
19       "Úhel = " +  úhel + " radiánů" +
20       "\n\nsin( PI/" + podíl + " ) = " + sinus +
21       "\n\nBude to vyhovovat?" ) )
22     {
23       case YES_OPTION:  odpoved = "To mne těší";      break;
24       case NO_OPTION:   odpoved = "To mne mrzí";      break;
25       case CANCEL_OPTION:
26       case CLOSED_OPTION: odpoved = "Ukončil jste program";   break;
27     }
28     showMessageDialog( null, odpoved );
29   }
30
31
32   public static double zaokrouhli( double číslo ) {
33     return round( PŘESNOST * číslo ) / PŘESNOST;
34   }
35 }
```

Hlavní výhradou oponentů proti statickému importu je ztráta informace o mateřské třídě použitého statického členu, která může ve čtenáři vyvolat dojem, že použité členy jsou statickými atributy či metodami dané třídy.

Pozor!

Ve svých programech myslete na to, že není vhodné používat statický import pro všechny použité statické členy jiných tříd. Doporučuje se omezit pouze na deklaraci importu takových členů, u nichž zároveň nehrozí nebezpečí, že by je mohl programátor, který bude váš kód číst, považovat za členy třídy, v níž jsou použity. Navíc je vhodné deklarovat statický import pouze pro ty členy, které jsou v daném kódu použity poměrně často, či pro ty, u nichž použití statického importu program výrazně zpřehlední.

Sloučení importu atributů a metod

Při deklaraci importu je třeba si uvědomit, že příkaz `import` nerozlišuje atributy a metody. Importuje pouze název. Používá-li proto třída stejně nazvaný atribut i metodu,

dovezete jedním statickým importem oba dva. Navíc jste jistě odhadli, že při importu názvu metody importujete názvy všech jejích přetížených verzí.

Všechny popisované skutečnosti demonstruje následující program. Způsob použití statických členů bez využití statického importu demonstruje metoda `postaru()`, způsob jejich použití s využitím statického importu pak demonstruje konstruktor `SloučeníImportů`.

```
 1 import static rup.česky.java15.simport.Zdroj.K;
 2 import static rup.česky.java15.simport.Zdroj.Vnořená;
 3 import static rup.česky.java15.simport.Zdroj.Vnořená2.KVV;
 4
 5 public class SloučeníImportů
 6 {
 7   private String s1, s2, s3, s4, s5;
 8
 9   public void postaru() {
10     s1 = Zdroj.K;
11     s2 = Zdroj.K();
12     s3 = Zdroj.K( s2 );
13     s4 = Zdroj.Vnořená.KV;
14     s5 = Zdroj.Vnořená2.KVV;
15   }
16
17   SloučeníImportů() {
18     s1 = K;
19     s2 = K();
20     s3 = K( s2 );
21     s4 = Vnořená.KV;
22     s5 = KVV;
23   }
24
25   public static void test() {
26     SloučeníImportů si = new SloučeníImportů();
27     System.out.println("Instance: s1=" + si.s1 + ", s2=" + si.s2 +
28       ", s3=" + si.s3 + ", s4=" + si.s4 + ", s5=" + si.s5);
29   }
30 }
31
32 class Zdroj {
33   public static final String K = "Konstanta";
34
35   public static String K() {
36     return "Metoda";
37   }
38
39   public static String K( String s ) {
40     return "Parametr: " + s;
41   }
```

```
42
43   static class Vnořená  {
44     static final String KV = "Vnořená";
45   }
46
47   static class Vnořená2  {
49     static final String KVV = "Druhá vnořená";
50   }
51 }
```

Po spuštění metody `test()` ve třídě `SloučeníImportů` se na standardní výstup vypíše:

```
Instance: s1=Konstanta, s2=Metoda, s3=Parametr: Metoda, s4=Vnořená, s5=Druhá vnořená
```

Poznámka:
Na předchozím programu si všimněte, že chcete-li používat statické členy jiné třídy bez kvalifikace, musíte je importovat i tehdy, jsou-li obě třídy uloženy ve stejném souboru.

Možnost importu statických členů neplatí jenom pro třídy „prvního sledu“, ale i pro jejich vnořené třídy (vnořená třída patří také mezi statické členy své vnější třídy) a pro statické členy těchto vnořených tříd. Pro možnost importu stačí, když je z daného místa příslušný statický člen viditelný a dosažitelný.

Současně je třeba mít na paměti, že deklarované statické importy platí pro všechny třídy definované v daném souboru. Neexistuje možnost, jak pro různé třídy v jednom souboru deklarovat různé sady importovaných názvů.

Kolize importů

Při statickém importu musíme myslet na to, že neimportujeme konkrétní členy, ale pouze názvy. To nám může občas přinést některé problémy. Tentokrát nejprve uvedu příklad kolizního programu, a teprve pak začnu jednotlivé problémy rozebírat.

```
 1 import static rup.česky.java15.simport.První.*;
 2 import static rup.česky.java15.simport.Druhá.*;
 3 //import static rup.česky.java15.simport.Druhá.společná;
 4
 5 public class KolizeImportů
 6 {
 7   public static void testKolize()  {
 8     první();
 9     druhá();
10 //    společná();
11     společná(1);
12     společná(1,2);
13     První.společná();
14     Druhá.společná();
```

```
15    }
16
17  }
18
19  class První  {
20    static void první()  {}
21    static void společná() {}
22    static void společná( int i )  {}
23  }
24
25
26  class Druhá  {
27    static void druhá()  {}
28    static void společná() {}
29    static void společná( int i, int j )  {}
30  }
```

Hlavním problémem předchozího příkladu je metoda `společná`, která je definována v obou „číslovaných" třídách.

Přeložíte-li program v té podobě, v jaké je v předchozí ukázce vytištěn, proběhne překlad bez problémů, avšak pouze díky tomu, že v metodě `testKolize()` je na řádku 10 zakomentován nekvalifikovaný příkaz volání bezparametrické verze metody `společná()`. Od tříd `První` a `Druhá` jsou totiž importovány všechny názvy jejich statických členů, takže název `společná` je importován dvakrát. V současné verzi programu jsou však volány pouze ty verze metody `společná`, které v těchto třídách nekolidují, takže se v nich překladač dokáže zorientovat.

Problém by nastal v okamžiku, kdy bychom zrušili komentář u volání bezparametrické verze. Tu totiž definují obě třídy a překladač si pak nemůže vybrat – ohlásí proto syntaktickou chybu.

Rozhodnete-li se proto deklarovat statický import této metody explicitně a odkomentujete-li příkaz na řádku 3, problém na řádku 11 sice vyřešíte, ale současně vytvoříte jiný.

Problém na řádku 11 vyřešíte proto, že explicitní import názvu přehlasuje implicitní hvězdičkový import, takže překladač okamžitě pozná, kterou ze stejnojmenných a stejně parametrizovaných metod má volat.

Explicitní import ale také zařídí, že překladač se již nepídí po daném názvu v jiných třídách, takže následující volání jednoparametrické verze dané metody na řádku 12 nevyřeší, protože třída `Druhá`, z níž byl daný název importován, žádnou jednoparametrickou verzi `společná(int)` nedefinuje. Překladač proto ohlásí syntaktickou chybu.

Jak vidíte, ať tak či tak, vždy vytvoříte syntakticky chybný program. Nezbude vám tedy, než se rozhodnout, která z uvedených dvou cest je jednodušší, a které volání budete muset kvalifikovat názvem třídy.

Kapitola 2

Rozšíření příkazu for

Častou stížností programátorů, kteří přecházeli na Javu z jiných jazyků, byla neexistence příkazu cyklu, který by umožnil jednoduše zapsat algoritmy, při nichž je třeba provést nějakou operaci se všemi položkami uloženými v nějakém kontejneru. Příkazu, který býval často označován jako *cyklus for each*.

V této kapitole si ukážeme, jak se s tímto požadavkem vypořádali autoři nové verze jazyka. Seznámíme se s novou podobou příkazu `for` a se základními syntaktickými pravidly jeho použití. Současně si také vysvětlíme, v jakých situacích jej není možno použít a musíme v nich vystačit s klasickými „ukecanými" konstrukcemi.

V závěru kapitoly si předvedeme, jaké vlastnosti musí mít třída (nemusí to být nutně kontejner), pro jejíž instance lze tento příkaz použít, a jednu takovou třídu definujeme.

Programy:

Všechny třídy, o nichž se v této kapitole hovoří, jsou umístěny v balíčku `rup.česky.java15.foreach`*. Samostatně uvedené metody najdete ve třídě* `ForEach`*.*

Předehra pro méně zkušené

Začínající programátoři mají někdy problémy s chápáním termínu *kontejner*. Kontejner bychom mohli stručně charakterizovat jako objekt, který je určen pro ukládání jiných objektů.

Začínající programátoři často znají z kontejnerů pouze pole, které lze chápat jako statický kontejner: jakmile je jednou definujete, má pevně danou velikost, kterou nemůžete měnit.

Vedle polí však existuje celá plejáda dalších kontejnerů, o kterých se ale popis syntaxe nezmiňuje, protože jsou to obyčejné třídy ze standardní knihovny. To však ještě neznamená, že by byly méně důležité. Naopak, moderní programy jim dávají stále častěji přednost před poli.

Poznámka:

Některé programovací jazyky (např. Visual Basic) sice zavádějí tzv. dynamické pole*, ale ve skutečnosti se jedná o statické pole, pro nějž lze uprostřed programu definovat novou velikost. Skutečný dynamický kontejner totiž svoji velikost mění operativně sám bez nutnosti explicitního zásahu programátora.*

Protože ve standardní knihovně implementuje převážná většina definovaných kontejnerových tříd rozhraní `Collection`, hovoří někteří autoři o kontejnerech jako o *kolekcích*.

Syntaxe

Java 5.0 definuje vedle stávajícího příkazu `for` ještě jeho další podobu, která je obdobou příkazu typu `for each`, známého z jiných jazyků. Syntaxe tohoto příkazu je velice jednoduchá:

`for(` ⟨*Typ*⟩ ⟨*Parametr*⟩ `:` ⟨*Kontejner*⟩ `)`

Typ označuje typ parametru cyklu, *Parametr* je identifikátor parametru cyklu a *Kontejner* je identifikátor kontejneru (případně volání metody vracející kontejner), jehož prvky budou postupně přiřazovány parametru cyklu. V dalším textu budu tento typ cyklu označovat jako cyklus `for(:)`.

Kontejnerem musí být pole nebo objekt implementující rozhraní `Iterable`. Typ prvků uložených v kontejneru musí být automaticky převeditelný na deklarovaný typ parametru cyklu.

Poznámka:

Pokud jste to nepostřehli, tak ještě jednou připomenu, že to, že pro danou instanci můžeme použít cyklus `for(:)`, není dáno tím, že se jedná o kontejner, ale tím, že její třída implementuje rozhraní `Iterable`. Tento cyklus proto můžeme využít nejenom při práci se skutečnými kontejnery, ale i s různými generátory a dalšími iterovatelnými objekty.

Použití při práci s poli

Podívejme se nyní na několik řešení, jak bychom mohli definovat metodu, která vrací průměr hodnot v kontejneru.

Jednorozměrné pole

Začneme prací s vektory, tj. jednorozměrnými poli:

```
 1 public static double průměrPoleStarý( double[] dd ) {
 2   double součet = 0;
 3   for( int i=0;   i < dd.length;   i++ )
 4     součet += dd[i];
 5   return součet / dd.length;
 6 }
 7
 8
 9 public static double průměrPoleNový( double[] dd ) {
10   double součet = 0;
11   for( double d : dd )
12     součet += d;
13   return součet / dd.length;
14 }
15
16
17 public static void testPrůměruPole() {
18   double[] d = { 1, 3, 5, 7, 9 };
19   System.out.println("Průměr postaru: " + průměrPoleStarý( d ));
20   System.out.println("Průměr ponovu:  " + průměrPoleNový ( d ));
21 }
```

Vícerozměrná pole

Použití cyklu `for(:)` není omezeno pouze na jednorozměrná pole. Můžete jej použít pro pole libovolných rozměrů. Následující program např. definuje metodu pro tisk dvourozměrného pole obsahujícího pouze kladné celočíselné hodnoty menší než 100.

```
 1 public static void tisk2DPole( String název, int[][] matice ) {
 2   System.out.println( název );
 3   for( int[] řádek : matice ) {
 4     for(  int i : řádek ) {
 5       //Očekává pole celých nezáporných čísel menších než 100
 6       String výstup = ((i<10) ? "  " : " ")  +  i;
 7       System.out.print( výstup );
 8     }
 9     System.out.println();
10   }
11 }
12
13
14 public static void testTisku2DPole() {
15   final int[][] ii = { {0, 1, 2, 3}, {10, 11, 12}, {20, 21}, {30} };
16   tisk2DPole( "Trojúhelník", ii );
17 }
```

Po spuštění metody `testTisku2DPole()` se na standardní výstup vypíše:

```
1 Trojúhelník
2   0  1  2  3
3  10 11 12
4  20 21
5  30
```

Poznámka:

V předchozím programu si všimněte, že cyklus `for(:)` vytiskl pole správně proto, že prvním indexem byl index řádku a druhým index sloupce. Kdybyste se rozhodli interpretovat indexy jako souřadnice `x` a `y`, byla by vytištěné pole transponované a pro správné zobrazení byste museli použít klasický cyklus s parametrem.

Kdy cyklus for(:) použít nelze

Pole jsme mohli takto elegantně vytisknout proto, že definice cyklu `for(:)` zaručuje, že při práci s poli budou parametru cyklu předávány jednotlivé prvky pole ve stejném pořadí, v jakém jsou umístěny v poli, tj. od nultého do posledního.

Při práci s cyklem `for(:)` musíme mít stále na paměti, že cyklus:

```
for( <typ> <proměnná> : <pole> )  {  /* tělo cyklu */ }
```

se přeloží do podoby:

```
for( int <index>=0;   <index> < <pole>.length;   <index>++ ) {
  <typ> <proměnná> = <pole>[ <index> ];
  /* tělo cyklu */
}
```

kde proměnná *<index>* je pro nás nedosažitelná.

Z toho logicky vyplývá, že cyklus `for(:)` využijeme pouze tehdy, nepotřebujeme-li v těle cyklu pracovat s indexem zpracovávaného prvku. Budeme-li potřebovat pracovat s indexem zpracovávaného prvku, musíme zůstat u klasické podoby cyklu s parametrem, protože cyklus `for(:)` nám hodnotu indexu právě zpracovávaného prvku neprozradí.

Cyklus `for(:)` nemůžeme použít ani tehdy, potřebujeme-li měnit prvky procházeného pole, protože jejich obsah se na počátku těla cyklu uloží do pomocné proměnné *<proměnná>* a tím, že bychom do této proměnné přiřadili jakoukoliv hodnotu, obsah příslušného prvku pole nijak neovlivníme.

Použití při práci s kontejnery

Jak jsme si řekli, cyklus `for(:)` můžeme použít nejenom k procházení polí, ale i dynamických kontejnerů a obecně instancí libovolné třídy implementující rozhraní `Iterable`.

Pokud bychom potřebovali definovat metodu, která by nějak přehledně tiskla páry *klíč-hodnota* uložené v mapě, definovali bychom ji asi následovně:

```
1 public static void tiskniMapuDříve( String titulek, Map mapa ) {
2   System.out.println( titulek );
3   for( Iterator it = mapa.entrySet().iterator();   it.hasNext();  ) {
4     Entry entry = (Entry)it.next();
5     System.out.println( "  " + entry.getKey() +
6                       " -- " + entry.getValue() );
7   }
8 }
```

Doposud jsme museli vždy nejprve požádat kontejner o iterátor a s jeho pomocí pak získávat jednotlivé prvky uložené v kontejneru. S využitím příkazu `for(:)` se o iterátor nemusíme starat a můžeme použít jednodušší definici:

```
1 public static void tiskniMapuNyní( String titulek, Map mapa ) {
2   System.out.println( titulek );
3   for( Object entry : mapa.entrySet() )
4     System.out.println( "  " + ((Entry)entry).getKey() +
5                       " -- " + ((Entry)entry).getValue() );
6 }
```

Zde jsme ještě museli při používání obdržené dvojice přetypovávat na `Entry`, ale zanedlouho si ukážeme, jak se můžeme od této nutnosti osvobodit.

Budete-li si chtít obě definice vyzkoušet, můžete použít např. následující testovací metodu:

```
1 public static void testTiskuMapy() {
2   Map m = new HashMap();
3   m.put( "česky",    "Dobrý den" );
4   m.put( "anglicky", "Hello" );
5   m.put( "německy",  "Guten Tag" );
6
7   tiskniMapuDříve( "Dříve - pozdravy:", m );
8   tiskniMapuNyní ( "\nNyní - pozdravy:", m );
9 }
```

Po jejím spuštění bude na standardní výstup odeslán text:

```
1 Dříve - pozdravy:
2   německy -- Guten Tag
3   anglicky -- Hello
4   česky -- Dobrý den
5
6 Nyní - pozdravy:
7   německy -- Guten Tag
8   anglicky -- Hello
9   česky -- Dobrý den
```

Skrytý iterátor

Při využívání cyklu `for(:)` jako náhražky práce s iterátory platí, že cyklus `for(:)` je možno používat pouze v případě, kdy nepotřebujeme využívat iterátor řídící běh cyklu. Pokud bychom chtěli např. definovat cyklus, který z kolekce objektů odstraní ty, jež budou vykazovat nějakou vlastnost, bez iterátoru se neobjedeme.

Podívejme se na následující trojici metod. Obě dvě odstraňovací metody, tj. `odstraňPrázdnéStarý(Collection)` i `odstraňPrázdnéŠpatný(Collection)` odstraňují z kolekce textových řetězců takové, které jsou nezadané nebo prázdné. První z nich to dělá klasicky pomocí iterátoru, druhá se pokouší využít cyklus `for(:)`.

```
 1 public static void odstraňPrázdnéStarý( Collection c ) {
 2   //Předpokládám, že se jedná o kolekci textových řetězců
 3   for( Iterator it = c.iterator();   it.hasNext();   ) {
 4     String s = (String)it.next();
 5     if( (s == null)  ||  s.equals("") )
 6       it.remove();
 7   }
 8 }
 9
10
11 public static void odstraňPrázdnéŠpatný( Collection c ) {
12   for( Object o : c ) {
13     //Předpokládám, že se jedná o kolekci textových řetězců
14     if( (o == null)  ||  ((String)o).equals("") )
15       c.remove( o );
16   }
17 }
18
19
20 public static void testRemove() {
21   Collection col = Arrays.asList(
22                     new String[] { "A", "", "C", null, "D" } );
23   Collection col1;
24
25   col1 = new LinkedList( col );
26   System.out.println("Postaru - původní:  " + col1 );
27   odstraňPrázdnéStarý( col1 );
28   System.out.println("Postaru - probraný: " + col1 );
29
30   col1 = new LinkedList( col );
31   System.out.println("Ponovu - původní:  " + col1 );
32   odstraňPrázdnéŠpatný( col1 );
33   System.out.println("Ponovu - probraný: " + col1 );
34 }
```

Spustíme-li testovací metodu `testRemove()`, zjistíme, že první metoda v pořádku projde, kdežto druhá zhavaruje a systém vyhodí výjimku:

```
1 Postaru - původní:  [A, , C, null, D]
2 Postaru - probraný: [A, C, D]
3 Ponovu - původní:  [A, , C, null, D]
4 Exception in thread "main" java.util.ConcurrentModificationException
5   at java.util.LinkedList$ListItr.checkForComodification(Unknown Source)
6   at java.util.LinkedList$ListItr.next(Unknown Source)
7   at java15.foreach.ForEach.odstraňPrázdnéŠpatný(ForEach.java:108)
8   at java15.foreach.ForEach.testRemove(ForEach.java:129)
9   at java15.foreach.ForEach.main(ForEach.java:159)
```

Proč k tomu došlo? I když to na první pohled není vidět, cyklus `for(:)` používá iterátor stejně, jako jsme jej používali v klasicky koncipovaném cyklu. Jediný rozdíl je v tom, že se o jeho použití nemusí starat programátor, ale postará se o něj překladač.

Jak ale víme, v cyklu používajícím iterátor nesmíme měnit strukturu iterované kolekce. Chceme-li nějaký prvek z kolekce vyjmout, musíme to udělat prostřednictvím iterátoru a jeho metody `remove()`. Chceme-li naopak do kolekce něco přidat, musíme přidávat prvky buď mimo cyklus, anebo použít nějaké specializované kolekce se specializovaným iterátorem (např. `ListIterator`), který umí do kolekce přidávat nové prvky. Nikdy však nesmíme vzít změnu struktury do vlastních rukou.

Jenomže to právě metoda `odstraňPrázdnéŠpatný(Collection)` dělá. Protože programátor nemá v cyklu `for(:)` k dispozici iterátor, mohl by si myslet, že může odstranit nebo přidat prvek přímo. Ale to jej nesmí ani napadnout!

Poznámka:

Budete-li potřebovat měnit uvnitř cyklu strukturu procházené kolekce, musíte použít klasickou podobu cyklu for a příslušný iterátor.

Definice vlastní iterovatelné třídy

Jak jsem se již zmínil, používání cyklu `for(:)` není omezeno pouze na pole a kolekce ze standardní knihovny. Tento cyklus můžete používat i při zpracovávání instancí vaší vlastní třídy, která navíc vůbec nemusí být kontejnerem. Jediné, co musíte zabezpečit, je implementace rozhraní `java.util.Iterable`.

Rozhraní `Iterable` požaduje po třídách, jež je implementují, pouze to, aby definovaly metodu `iterator()`, která vrátí odkaz na iterátor, jenž zprostředkuje přístup k prvkům daného kontejneru.

Ukažme si vše na příkladu. Třída `Fronta` v následujícím programu uchovává zadaná čísla, která čekají ve frontě na to, až budou obsloužena. Statická metoda `test()` pak demonstruje, jak lze na instancích třídy `Fronta` použít cyklus `for(:)`.

```
1 public class Fronta implements Iterable
2 {
3   List prvky = new LinkedList();
4
```

```
 5   public void zařaď( Object o ) {
 6     prvky.add( o );
 7   }
 8
 9   public Object další() {
10     Object ret = prvky.get(0);
11     prvky.remove(0);
12     return ret;
13   }
14
15   public Iterator iterator() {
16     return new Iterator() {
17       int pořadí = 0;
18
19       public boolean hasNext() {
20         return (pořadí < prvky.size());
21       }
22       public Object next() {
23         return prvky.get( pořadí++ );
24       }
25       public void remove() {
26         throw new UnsupportedOperationException();
27       }
28     };
29   }
30
31   public static void test() {
32     Random rnd = new Random();
33     Fronta fronta = new Fronta();
34
35     System.out.println("Přidáváme:");
36     for( int i=0;   i < 5;   i++ ) {
37       int číslo = new Integer(rnd.nextInt(100));
38       fronta.zařaď( číslo );
39       System.out.println("Přidáno: " + číslo);
40       System.out.print  ("   Stav:");
41       //Použití cyklu for(:) na instance třídy Fronta
42       for( Object fo : fronta )
43         System.out.print( " " + fo );
44       System.out.println("");
45     }
46     System.out.println("\nOdstraňujeme:");
47     for( int i=0;   i < 5;   i++ ) {
48       Object objekt = fronta.další();
49       System.out.println("Obslouženo: " + objekt);
50       System.out.print  ("    Stav:");
51       //Použití cyklu for(:) na instance třídy Fronta
52       for( Object fo : fronta )
```

```
53          System.out.print( " " + fo );
54        System.out.println("");
55      }
56    }
57  }
```

Typický výstup po spuštění metody `test()` by mohl vypadat např. následovně:

```
 1 Přidáváme:
 2 Přidáno: 73
 3    Stav: 73
 4 Přidáno: 19
 5    Stav: 73 19
 6 Přidáno: 62
 7    Stav: 73 19 62
 8 Přidáno: 78
 9    Stav: 73 19 62 78
10 Přidáno: 76
11    Stav: 73 19 62 78 76
12
13 Odstraňujeme:
14 Obslouženo: 73
15       Stav: 19 62 78 76
16 Obslouženo: 19
17       Stav: 62 78 76
18 Obslouženo: 62
19       Stav: 78 76
20 Obslouženo: 78
21       Stav: 76
22 Obslouženo: 76
23       Stav:
```

Kapitola 3

Automatické převody mezi primitivními a obalovými typy

Automatické převody mezi primitivními a obalovými typy, kterým se bude věnovat tato kapitola, umožňují, abychom mohli hodnoty primitivních typů používat velice jednoduše i tam, kde syntaxe vyžaduje některý z typů objektových. Zpočátku si opět ukážeme jednoduché použití dané konstrukce a v dalších částech vás pak upozorním na některé nepříjemné vedlejší efekty, s nimiž se můžete při jejím bezmyšlenkovitém použití setkat.

Programy:
Všechny metody, o nichž se v této kapitole hovoří, jsou definovány ve třídě `rup.česky.java15.autoboxing.Autoboxing`.

Předehra pro méně zkušené

Začneme opět předehrou pro ty méně zkušené a prozradíme si základní informace o datových typech Javy. Jak jistě všichni víte, Java není čistý objektově orientovaný jazyk. Její autoři rozdělili v zájmu zvýšení efektivity používané datové typy na primitivní a objektové.

Primitivních datových typů je devět: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char` a `void`. Práce s jejich hodnotami je součástí instrukční sady téměř každého procesoru, takže je maximálně rychlá a použití hodnot primitivních typů v programu je velice efektivní.

Všechny ostatní datové typy řadíme mezi objektové. Program nikdy nepracuje s jejich hodnotami, ale vždy pouze s odkazy do speciální paměti nazývané *halda* (heap), kde se o jejich vznik, působení a zánik stará *správce paměti* (garbage collector).

Práce s hodnotami (instancemi) objektových datových typů je sice pomalejší, ale na druhou stranu nám tyto typy umožňují neporovnatelně elegantnější a průzračnější vyjadřování, než jaké měli k dispozici programátoři v „předobjektové" době. Tím obrovsky zvyšují produktivitu naší práce.

V zájmu zachování maximální variability a transparentnosti programu je řada užitečných objektů ochotna pracovat pouze s odkazy na instance objektových typů. Abychom těmto objektům mohli „předhodit" ke zpracování i hodnoty primitivních typů, byly do Javy zavedeny tzv. *obalové typy*, které jsou schopny zabalit hodnotu primitivního typu do objektové slupky a odkaz na takto vytvořený objekt pak předat dalším objektům ke zpracování.

Každý primitivní typ má svůj obalový typ – existuje proto devět obalových typů: `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Doule`, `Character` a `Void`. V dosavadních verzích bylo v případě potřeby nutno vždy „ručně" vytvořit pro danou hodnotu primitivního typu odpovídající instanci jejího obalového typu a zase naopak od instancí obalových typů „ručně" vyloudit příslušné obalené hodnoty. Tohle nová verze Javy mění.

Automatické převody

Automatické převádění hodnot primitivních typů na instance příslušných obalových typů a zpět bylo pravděpodobně inspirováno obdobnou vlastností jazyka C#. Z C# byly nakonec převzaty i termíny **autoboxing** a **auto-unboxing**, které používá i oficiální dokumentace. Někteří autoři se jim ale brání a prosazují vhodnější termíny **autowrapping** a **autounwrapping**.

Základní ideou vedoucí k zavedení této konstrukce bylo usnadnit používání hodnot primitivních typů při práci s metodami, které vyžadují parametry objektových typů, především pak s kontejnery.

Část programu, kterou bychom museli ve starších verzích jazyka napsat následovně:

```
1 public static void dříve()
2 {
3   int     i2    = 2;
4   Integer čtyři = new Integer( 2*i2 );
5   int     i8    = 2 * čtyři.intValue();
6   System.out.println("Dříve - hodnoty i2=" + i2 + ", čtyři=" + čtyři +
7                      ", i8=" + i8 );
8 }
```

je nyní možno napsat jednodušeji:

```
1  public static void nyní()
2  {
3    int     i2    = 2;
4    Integer čtyři = 2 * i2;
5    int     i8    = 2 * čtyři;
6    System.out.println("Nyní  - hodnoty i2=" + i2 + ", čtyři=" + čtyři +
7                       ", i8=" + i8 );
8  }
```

Použití automatických převodů ve výrazech ale není příliš doporučováno, protože je spojeno s velkou režií. S takovýmto použitím se proto většinou setkáte v různých ukázkových programech jako byl ten předchozí, v nichž se autor snaží ukázat, co lze díky automatickým převodům provádět. Do reálných programů však nepatří.

Hlavní použití automatických převodů můžete očekávat při předávání hodnot parametrů metodám vyžadujícím parametry objektových typů a mezi nimi pak především metodám pracujícími s dynamickými kontejnery.

Nebezpečí při porovnávání

Zůstaňme ale nyní ještě chvíli u ukázkových programů. Ukážeme si na nich, jaké nebezpečí hrozí při neřízeném použití automatických převodů.

Instance obalových typů mohou vystupovat nejenom v aritmetických výrazech, ale můžeme je i porovnávat pomocí klasických operátorů porovnání. Podívejte se na následující metodu:

```
 1  public static void porovnání()
 2  {
 3    int     i2    = 2;
 4    Integer čtyři = 2 * i2;
 5    Integer osm   = 2 * čtyři;
 6    boolean i2_čtyři  = i2 < čtyři;
 7    boolean čtyři_osm = čtyři < osm;
 8    System.out.printf("%d < %s: %b\n%2$s < %s: %b",
 9      i2,  čtyři,  i2_čtyři, osm,  čtyři_osm );
10  }
```

Po spuštění této metody program vypíše na standardní výstup:

```
 i2   < čtyři : true
čtyři < osm   : true
```

Nicméně i když jsou hodnoty instancí porovnatelné pomocí operátoru nerovnosti, nemusejí ještě být stejně dobře porovnatelné pomocí operátoru rovnosti. Zkuste si spustit následující metodu:

```
 1 public static void rovnost()
 2 {
 3   Integer i1 = 0;
 4   Integer i2 = 0;
 5
 6   for( ;   i1 == i2;   i1++, i2++ );
 7   System.out.println("Rovnost přestala platitpro i1=" + i1 +
 8                      ", i2=" + i1 );
 9   i1 = new Integer( 0 );
10   i2 = new Integer( 0 );
11   System.out.println("new Integer(0) == new Integer(0): " + (i1==i2) );
12 }
```

Kdybychom v předchozí metodě použili místo typu `Integer` typ `int`, byl by naprogramovaný cyklus nekonečný. Pro hodnoty typu `Integer` ale tento cyklus nekonečný není. Jeho konečnost vyplývá z toho, že pro hodnoty, kterých mohou nabývat proměnné typu `byte`, tj. pro hodnoty z intervalu <-128;127>, jsou předem definovány konstanty typu `Integer`. Nepožádáte-li proto o vytvoření nové instance zavoláním konstruktoru, bude pro malé absolutní hodnoty převáděných čísel použita některá z předdefinovaných konstant.

Na počátku jsou proměnným `i1` a `i2` přiřazeny hodnoty prostřednictvím automatického převodu literálu – je jim proto oběma přiřazen odkaz na tutéž předdefinovanou konstantu. Proto aplikace operátoru == vrátí `true`.

Automatických převodů je využito i pro přiřazení nové hodnoty oběma proměnným v modifikační části hlavičky cyklu. Tím jim byla přiřazena příslušná předdefinovaná konstanta – vzhledem k provedené operaci oběma stejná.

Jakmile ale hodnota obou proměnných překročila magickou hodnotu 127, byl před vytvořením příslušného odkazu zavolán konstruktor, a do každé proměnné byl proto uložen odkaz na nově vytvořenou instanci. Přestože obě takto vytvořené instance měly stejnou hodnotu, pro virtuální stroj to byly dvě různé instance, a proto cyklus předčasně ukončil.

Jak ukazují poslední tři příkazy, kdybychom místo automatického převodu použili konstruktor hned při prvním přiřazení, neplatila by rovnost od samého počátku.

Tento příklad je pouze další ukázkou toho, že při práci s proměnnými obalových typů můžeme dostat jiné výsledky než při práci se stejnými hodnotami primitivních typů.

Omezení převodů

Z práce s primitivními datovými typy jste zvyklí na automatické převody z „menších“ typů na větší, tj. např. z `char` na `int`, z `int` na `double` atd. Při převodech z primitivních datových typů na obalové však tyto převody nefungují. V obráceném směru, tj. při převodech z obalových typů na primitivní, ale není problém.

Vše demonstruje následující ukázka, v níž jsou příkazy obsahující syntaktickou chybu zakomentovány.

```
 1 public static void převody()
 2 {
 3   Character cC;
 4   Integer   iI;
 5   Double  dD;
 6   char   cc = 'a';
 7   int  ii = 1;
 8   double dd = 1d;
 9   cC = 'a';       //char -> Character - ANO
10   ii = 'a';       //char -> int - ANO
11   ii = cc;        //char -> int - ANO
12   ii = cC;        //Character -> int - ANO
13 //  iI = 'a';      //char -> Integer - NELZE
14   iI = (int)'a';  //int -> Integer - ANO
15   dd = 1;         //int -> double - ANO
16   dd = iI;        //Integer -> double - ANO
17 //  dD = 1;        //int -> Double - NELZE
18   dD = 1d;        //double -> Double - ANO
19 }
```

Rady do života

Puristé mají výhrady i proti automatickým převodům, protože se díky nim ztrácí explicitní informace o tom, co je hodnotou primitivního typu a co je instancí objektového typu. Praktici zase namítají, že nyní ztrácejí přehled nad vytvářením nových instancí, což může vést k neočekávanému snížení efektivity programu a občas i nežádoucímu chování (příklady jsme si ukazovali).

Opět mohu jenom poradit: dávejte si proto pozor na to, kde automatické převody používáte, a nezneužívejte jejich možností. Využívejte je pouze v situacích, pro něž byly určeny, čímž je především předávání hodnot parametrů metodám, které vyžadují odkazy na instance objektových typů. Jejich nadužívání zbytečně snižuje efektivitu programu a v některých situacích si „koleduje“ o chybu, která způsobí podivné chování programu.

Největší nebezpečí automatických převodů ale nespočívá v jejich nezřízeném nadužívání – tomu se rozumný programátor instinktivně brání. Jejich největší nebezpečí spočívá v tom, že si někde neuvědomíte, že překladač automatický převod použil. Hlídejte si takové situace a naučte se jim předcházet.

Kapitola 4

Metody s proměnným počtem parametrů

Po metodách s proměnným počtem parametrů dlouho volali programátoři, kteří na Javu přešli z jazyků C či C++, v nichž na ně byli zvyklí. Na rozdíl od těchto jazyků je však jejich použití v Javě mnohem jednodušší.

V této kapitole si nejprve ukážeme, jak jednoduše lze metody s proměnným počtem parametrů definovat a jak je lze ještě snadněji použít. Poté si prozradíme, na která omezení musíme při definicích těchto metod myslet a kdy nás může chování programu překvapit.

Metody s proměnným počtem parametrů jsou pravděpodobně nejméně problémovým přídavkem nové verze. Metodu, jejíž chování odpovídalo chování metody s proměnným počtem parametrů, jste sice mohli definovat již dříve, ale to bylo přece jenom výrazně pracnější. Podívejme se nyní na to, jak bychom mohli procedury s proměnným počtem parametrů definovat.

Programy:
Všechny třídy, o nichž se v této kapitole hovoří, jsou definovány v balíčku `rup.česky.java15.varpar`.

Jediný parametr

V podkapitole *Definice vlastní iterovatelné třídy* na straně 23 jsem použil třídu `Fronta`, která definovala superjednoduchou frontu. Podívejme se nyní na to, jak bychom mohli rozšířit definici této třídy o konstruktor, který by převzal jako parametr počáteční

sadu hodnot zařazených do fronty. V dřívějších verzích Javy bychom takovýto konstruktor definovali následovně:

```
1 public Fronta( Object[] o ) {
2   for( int i=0;   i < o.length;   i++ )
3     prvky.add( o[i] );
4 }
```

Definice konstruktoru je jednoduchá, ale jeho použití by již tak jednoduché nebylo. Před vlastním vyvoláním konstruktoru bychom museli nejprve připravit a naplnit vektor, který mu předáme jako parametr. I kdybychom využili možností automatického převodu celých čísel na jejich obalové typy, vypadalo by jeho vyvolání v testovací metodě přibližně takto:

```
fronta = new Fronta( new Object[] { 1, 3, 5, 7, 9 } );
```

Java 5.0 tento zápis zjednodušuje zavedením metod s proměnným počtem parametrů. Předchozí definici bychom podle ní mohli upravit do tvaru:

```
1 public Fronta( Object... o ) {
2   for( Object oo : o )
3     prvky.add( oo );
4 }
```

Vynechám-li použití příkazu `for(:)`, tak se tu toho moc nezměnilo – hranaté závorky byly nahrazeny třemi tečkami a to je vše. Základní změna totiž není v definici metody, ale v možnostech jejího použití. Takto definovanou metodu můžeme použít nejenom postaru (tato možnost stále zůstává), ale také následovně:

```
fronta = new Fronta( 1, 3, 5, 7, 9 );
```

Odpadá tedy nutnost explicitně, vlastníma rukama definovat pole předávané jako parametr – tuto definici opět necháme na překladači.

Pole zastupující parametry, jejichž počet není v době definice metody znám, může být i prázdné, tj. můžeme volat:

```
fronta = new Fronta();
```

Poznámka:

Předchozí příklad v sobě opět nese problém, který nastane v okamžiku, kdy máme paralelně definovanou druhou metodu, na níž takovéto volání také sedí – v našem případě máme-li definován bezparametrický konstruktor. Java takovéto potenciální kolize nehlídá, protože pro ni se jedná o metody s odlišnou charakteristikou. Budete-li proto mít současně definován bezparametrický konstruktor, zavoláte výše uvedeným příkazem jej.

Parametry primitivních typů

Doposud jsme v deklarovaných metodách používali pouze parametry objektových typů. V řadě úloh je však použití takových parametrů neefektivní, protože je s ním spojena zbytečná režie. Java naštěstí umožňuje definovat proměnný počet parametrů i pro primitivní datové typy. Kdybychom např. potřebovali metodu počítající průměr zadaných parametrů, mohli bychom ji definovat následovně:

```
1  public static double průměr( double ... dd ) {
2    double součet = 0;
3    for( double d : dd )
4      součet += d;
5    return součet / dd.length;
6  }
7
8  public static void testPrůměr()  {
9    System.out.println("Celá čísla:   " + průměr( 10, 20, 30, 40, 50 ) );
10   System.out.println("Reálná čísla: " + průměr( 1d, 2d, 3d, 4d, 5d ) );
11 }
```

Jak dokazuje přiložený test, díky tomu, že se jedná o primitivní typy, budou v tomto případě fungovat i automatické převody z „menších“ datových typů na „větší“, které při převodech na obalové typy nefungují.

Více parametrů

Metody s proměnným počtem parametrů mohou mít vedle vektoru zastupujícího ony potenciální parametry, jejichž počet neznáme, ještě libovolný počet dalších parametrů. „Proměnný vektor“ však musí být vždy uveden jako poslední, a to nezávisle na tom, jestli byste chtěli definovat následující parametry tak, aby je bylo možno bezpečně odlišit od těch, jejichž počet není předem znám.

Není proto možno definovat metodu:

```
public void syntaktickáChyba( String s1, int... ii, String s2 )
```

Potřebujete-li definovat metodu se dvěma dalšími řetězcovými parametry, musí mít její hlavička tvar:

```
public void takhleJeToSprávně( String s1, String s2, int... ii )
```

Opět ale platí, že budete-li mít současně definovánu přetíženou metodu

```
public void takhleJeToSprávně( String s1, String s2 )
```

překladač ji akceptuje a v případě, že budete volat metodu `takhleJeToSprávně` pouze se dvěma řetězcovými parametry, zavolá verzi bez závěrečného celočíselného vektoru. Kdybyste ji definovanou neměli, zavolal by původní verzi s prázdným vektorem `ii`.

Více skupin parametrů s proměnným počtem členů

Tato podkapitola je věnována spíše začínajícím programátorům, kteří občas tápou při řešení některých úloh.

Programy:

Všechny metody, o nichž se v této podkapitole hovoří, jsou spolu s testem definovány ve třídě `rup.česky.java15.varpar.Sklad`.

Potřebujete-li definovat metodu, která má více skupin parametrů, jejichž počet předem neznáte, můžete použít definici proměnného typu parametrů pouze pro jednu z nich. Možných postupů je několik:

- Použijete u ostatních skupin klasický způsob zápisu a budete na poslední chvíli vytvářet potřebné vektory.
- Definujete pro zbylé skupiny pomocné metody, které vám umožní zadat i ostatní skupiny co nejstručněji.
- Je-li ve všech skupinách stejný počet prvků (tvoří vlastně n-tice), definujete si přepravku a metoda může přijímat proměnný počet instancí této přepravky.
- Je-li ve všech skupinách stejný počet prvků, můžete definovat metodu tak, že bude přijímat pouze jednu n-tici, a aby se práce s více n-ticemi zjednodušila, může metoda vracet odkaz na svoji instanci a umožnit tak zřetězení volání.

Všechny způsoby řešení si postupně ukážeme na příkladech. Představte si, že definujeme třídu `Sklad`, pro kterou potřebujeme definovat metody pro příjem a výdej zboží, přičemž každé zboží je definované názvem, s nímž je současně uváděn počet přijímaných, resp. vydávaných položek. Názvy zboží spolu s počtem položek na skladě budou ukládány do mapy `sklad`.

Dejme tomu, že metodu pro příjem zboží definujeme tak, že názvy zboží bude očekávat jako vektor, kdežto pro zadání počtů využijeme proměnný počet parametrů. Její definice by proto mohla vypadat následovně:

```
 1 public void příjem( String[] názvy, int... počty ) {
 2   if( názvy.length != počty.length )
 3     throw new IllegalArgumentException(
 4       "Počet názvů zboží a přijatých množství si neodpovídá");
 5   for( int i=0;   i < názvy.length;   i++ ) {
 6     String název = názvy[i];
 7     if( sklad.containsKey( název ) )
 8       sklad.put( název,  ((Integer)sklad.get(název))+počty[i] );
 9     else
10       sklad.put( název, počty[i] );
11   }
12 }
```

Bez využití různých „udělátek“ bychom tuto metodu mohli volat např. následovně:

```
Sklad s = new Sklad();
s.příjem( new String[]{ "káva",  "čaj", "kola"}, 3, 3, 2 );
```

Abychom při své lenosti nemuseli vypisovat celou klauzuli potřebnou pro vytvoření vektoru řetězců, definujeme si pomocnou metodu `vs` (= vektor stringů):

```
1 public static String[] vs( String ... ss )  {
2   return ss;
3 }
```

Výše uvedené volání by se pak zjednodušilo do tvaru:

```
Sklad s = new Sklad();
s.příjem( vs( "káva",  "čaj", "kola"), 3, 3, 2 );
```

Jak vidíte, moc jsme si nepomohli a hlavně je volání neustále nepřehledné, protože není možno jednoduše zkontrolovat, zda jsou dodaná množství uváděna ve stejném pořadí jako názvy příslušného zboží.

Jednou z cest, jak můžeme tento problém vyřešit, je definovat přepravku, což je třída, jejíž instance slouží pouze k tomu, abychom do nich někde „naložili“ skupinu hodnot, „dopravili“ ji na jiné místo programu a tam tyto hodnoty zase „vyložili“.

Přepravku můžeme definovat jako vnořenou třídu a lenoši, kteří neradi píší `new` doplní definici této třídy o definici tovární metody, která vrátí odkaz na nově vytvořenou přepravku:

```
 1 private static class Pár {
 2   String název;
 3   int počet;
 4 }
 5
 6 //Tato metoda je statickou metodou vnější třídy
 7 //Jejím jediným účelem je nepatrně zjednodušit zápis
 8 private static Pár pár( String název, int počet )  {
 9   Pár ret = new Pár();
10   ret.název = název;
11   ret.počet = počet;
12   return ret;
13 }
```

Když bychom pak definovali přetíženou verzi metody pro ukládání zboží do skladu následovně:

```
1 public void příjem( Pár... páry ) {
2   for( Pár pár : páry )  {
3     String název = pár.název;
4     if( sklad.containsKey( název) )
5       sklad.put( název,  ((Integer)sklad.get(název))+pár.počet );
```

```
6     else
7       sklad.put( název,  pár.počet );
8   }
9 }
```

mohli bychom předchozí příkazy přepsat do tvaru:

```
Sklad s = new Sklad();
s.příjem( pár("káva", 3), pár("čaj", 3), pár("kola", 2) );
```

Toto řešení sice vytváří řadu instancí, které se po chvíli zase zruší, ale nové verze virtuálních strojů jsou na práci s takovýmito dočasnými objekty připraveny a umějí s nimi pracovat tak, aby minimalizovaly potřebnou režii.

Poslední uváděná možnost počítá s tím, že se metoda s proměnným počtem parametrů vůbec využívat nebude, protože bude pokaždé pracovat pouze s jednou n-ticí. Metodu bychom pak definovali následovně:

```
1 public Sklad příjem( String název, int počet )  {
2   if( sklad.containsKey( název) )
3     sklad.put( název,  ((Integer)sklad.get(název))+počet );
4   else
5     sklad.put( název, počet );
6   return this;
7 }
```

Naši neustále omílanou dvojici příkazů bychom pak mohli přepsat následovně:

```
Sklad s = new Sklad();
s.příjem("káva", 3).příjem("čaj", 3).příjem("kola", 2);
```

Kapitola 5

Parametrizované datové typy a metody, typové parametry

Tato kapitola je nejdelší a pravděpodobně také nejobtížnější kapitolou celé knihy. Rozebereme v ní nejvýznamnější rozšíření nové verze jazyka, kterým jsou bezesporu parametrizované datové typy.

Nejprve se vysvětlíme jejich význam a ukážeme si jejich typické použití v programech. Pak se naučíme definovat vlastní parametrizované datové typy a řekneme si něco o tom, jak je to s parametrizovanými typy v hierarchii dědičnosti. Vzápětí si ukážeme, jak je možno využít typových parametrů i v definicích metod a jak je možno nastavovat pro skutečné hodnoty typových parametrů různé omezující podmínky.

V dalších podkapitolách si prozradíme něco o tom, jak s parametrizovanými typy a metodami zachází překladač a jak vytvářet své programy, aby se naše představy nedostaly do konfliktu s představami překladače. Seznámíme se s různými nejednoznačnými nebo konfliktními konstrukcemi a předvedeme si, jak se těmto problémům vyhýbat a jak je řešit.

V závěrečné části si ukážeme, jak lze při práci s typovými parametry používat žolíky, v jakých situacích nám pomohou správně specifikovat naše požadavky, kdy je pro ně třeba definovat nějaká omezení a jak tato omezení správně nastavit. V samotném závěru kapitoly si pak prozradíme něco o tom, jaké prostředky k analýze parametrizovaných typů a metod nám nabízí třída `Class` a třídy s ní spolupracující.

Programy:

Vzhledem k tomu, že je tato kapitola delší a obsahuje také více programů, rozmístil jsem je do několika balíčků. Všechny jsou podbalíčky svého společného rodiče `rup.česky.java15.ptm`. *Protože jsem jednotlivé programy řadil do balíčků podle jejich obsahu, a ne podle toho, ve které podkapitole jsou uvedeny, nemohu vám dát hromadný návod na to, kde jednotlivé programy najdete. Budu proto ve zdrojovém kódu tříd uvádět jejich příkaz* `package` *a u samostatně uvedených metod přidám komentář s odkazem na třídu, v níž je metoda definována.*

Nejprve trocha terminologie

Zavedení tzv. **generických** neboli **parametrizovaných typů a metod** je nejvýznamnějším rozšířením nové verze jazyka. Protože tyto datové typy nic negenerují a k lepšímu instruování překladače o použitých typech využívají **typové parametry**, budu v dalším textu většinou dávat přednost termínu *parametrizované typy a metody.* Pro stručnost budu pak pro parametrizované typy používat zkratku **PT**, pro parametrizované metody zkratku **PM** a pro společné označení parametrizovaných typů a metod zkratku **PTM**.

Poznámka:

Nepleťte si termíny parametrizovaná metoda *a* metoda s parametry. *Tyto dva termíny jsou navzájem nezávislé. Parametrizovaná metoda může, ale také nemusí mít parametry a naopak metoda s parametry může, ale také nemusí být parametrizovaná.*

Parametrizovanou metodou budu nazývat metodu, která má definovány typové parametry, jež prozrazují překladači pomocné informace o typech skutečných parametrů a návratových hodnot dané metody.

Předehra pro méně zkušené

Při definici kontejnerových tříd (ale nejenom těch) a jejich následném používání narážíme na problém, zda dát přednost univerzálnosti algoritmu nebo typové kontrole. Tyto dva požadavky šly v předchozích verzích Javy často proti sobě: pokud jste chtěli definovat třídy univerzální, museli jste použít také univerzální datový typ, s nímž pracují – většinou typ `Object`, čímž jste ale přišli o typovou kontrolu. Když jste chtěli naopak zachovat typovou kontrolu, museli jste přesně deklarovat použité typy dat, jenomže pak již ztratila třída na své univerzálnosti, protože pro jiný typ ji nebylo možno použít.

Kontejnerové třídy dávaly v předchozích verzích knihoven přednost univerzálnosti, a jsou proto definovány tak, že přijímají „do úschovy" instance třídy `Object`. Metody, pomocí nichž získáváme odkazy na uložené hodnoty, vrací odkazy na instance třídy `Object`, takže je programátor před následným použitím musí většinou nejdříve přetypovat na jejich skutečný typ.

Historie snahy o řešení tohoto problému je mnohem starší než programovací jazyk Java. S jedním z řešení přišli autoři jazyka C++, z jehož syntaxe Java vychází. Definovali tzv. *šablony* (templates), které umožňují, aby třídy byly definovány dostatečně univerzálně, avšak na druhou stranu zachovávaly možnost typové kontroly.

Autoři Javy chtěli svůj jazyk vybavit podobným mechanizmem, a již při uvedení jazyka slibovali, že v jeho dalších verzích plánují i rozšíření o generické typy a metody. Dlouhou dobu však neměli jasno v tom, jak přesně by se měly tyto typy a metody chovat a jaké by měly mít základní vlastnosti. Řešení, které nabízel jazyk C++, jim nepřipadalo jako to pravé ořechové, protože v některých směrech poněkud odporovalo duchu Javy. Ke svému „správnému" řešení se ale propracovávali pomalu a těžce. Na realizaci jejich slibu proto musela javová komunita čekat bezmála deset let.

PTM versus šablony jazyka C++

Funkce PTM může někomu připomínat funkci šablon jazyka C++, avšak to je jenom zdání. Šablony C++ a PT Javy představují dvě zcela různé koncepce. Jejich možnosti se sice částečně překrývají, avšak každá z nich nabízí něco, co ta druhá neposkytuje.

Šablony jazyka C++ si zaslouží název *generické*, protože jejich zpracování má opravdu za následek vygenerování nového datového typu nebo metody. Naproti tomu PT slouží pouze jako nástroj, jenž nám umožní předat překladači informace umožňující zlepšení typové kontroly.

Hlavním účelem PT je podle specifikace „zlepšení vyjadřovacích schopností a bezpečnosti pramenící z přesnější definice typů a možnosti přetypování". PT umožňují typovou kontrolu i v situacích, v nichž to doposud nebylo možné. Programátoři v těchto případech většinou využívali implicitního přetypování na rodičovskou třídu či implementované rozhraní doprovázené pozdějším explicitním přetypováním zpět na vlastní třídu dané instance.

PT tak umožňují přesunout co nejvíce typových kontrol z doby běhu do doby překladu. PT jsou proto záležitostí překladače. Virtuální stroj se o jejich použití vůbec nedozví. Z přeloženého kódu totiž není možno poznat, jestli původní program PT používal, protože soubory generované překladačem již obsahují pouze původní, neupravované datové typy očištěné od případných parametrů.

PT bychom mohli označit jako typy s parametry, jimiž jsou třídy a rozhraní, které budou použity při práci s konkrétní instancí daného parametrizovaného typu. PT nevytvářejí rodiny typů, jako je tomu u šablon C++. Jedná se pokaždé o týž základní typ, jehož parametry slouží pouze jako další informace pro překladač, jenž na jejich základě provádí některé dodatečné kontroly a/nebo automaticky vkládá potřebná přetypování.

Specifikace PT má však na paměti požadavek plné zpětné kompatibility, takže programy napsané s využitím PT mohou bez problému využívat třídy definované ve starších verzích Javy a nevyužívající proto PT.

Jakékoliv použití parametrizovaných typů a metod bez uvedení příslušných typových parametrů překladač komentuje varovnými hlášeními. Tvůrci programu tím dosáhli toho, že pokud napíšete program, přeřaďovači jehož překladu překladač neohlásí žádnou chybu ani nevydá varovné hlášení, víte, že program je typově bezpečný (type safe). (To samozřejmě neznamená, že v něm nemáte nějaké chyby ;-).)

Tip:
Až do této kapitoly jsme varovná hlášení překladače ignorovali. Nyní bych vám naopak doporučil, abyste si vypisování varovných zpráv zapnuli, protože vám tato hlášení poskytnou podrobné informace o potenciálních chybách v programu. V této kapitole si budeme navíc ukazovat, jak tyto zprávy eliminovat a potenciální zdroje chyb odstraňovat.
Postup, jak v nejpoužívanějších vývojových nástrojích zobrazování varovných hlášení zapnout, najdete v příloze Varovná hlášení překladače *na straně 147.*

Použití PT v programu

Nejčastější použití PT lze očekávat při práci s dynamickými kontejnery, které doposud neumožňovaly typovou kontrolu při vkládání dat a nutily uživatele k explicitnímu přetypovávání při jejich vybírání.

V podkapitole *Definice vlastní iterovatelné třídy* na straně 23 jsme definovali třídu implementující frontu objektových dat. Na této třídě si ukážeme rozdíl mezi možnostmi dřívějších verzí a verze 5.0.

Fronta textových řetězců v dřívějších verzích

Představme si, že bychom potřebovali definovat frontu, do níž bychom ukládali pouze textové řetězce, tj. instance třídy `String`. Pokud bychom nepotřebovali, aby třída byla součástí knihovny kolekcí, a chtěli zabezpečit typovou kontrolu, mohli jsme takovouto třídu v minulých verzích Javy definovat přibližně následovně:

```
 1 package rup.česky.java15.ptm.fronta;
 2
 3 import java.util.LinkedList;
 4 import java.util.List;
 5
 6 public class FrontaTextůStará
 7 {
 8   List názvy = new LinkedList();
 9
10   public void zařaď( String název ) {
11     názvy.add( název );
12   }
13
14   public void zařaď_1( Object objekt ) {
15     //Umožní zařadit do seznamu jakýkoliv objekt
```

```
16      názvy.add( objekt );
17    }
18
19    public String další() {
20      //Při vyjímání prvku z kontejneru je třeba obdržený odkaz přetypovat
21      String ret = (String)názvy.get(0);
22      názvy.remove(0);
23      return ret;
24    }
25
26    public boolean isDalší() {
27      return (názvy.size() > 0);
28    }
29
30  //===== TESTY ==========================================================
31
32    public static void test() {
33      FrontaTextůStará fts = new FrontaTextůStará();
34      for( String s : new String[] { "raz", "dva", "tři" } )
35        fts.zařaď( s );
36      fts.zařaď_1( fts );
37
38      System.out.println("Texty ve frontě: ");
39      while( fts.isDalší() ) {
40        System.out.println("   " + fts.další() );
41      }
42    }
43  }
```

Překladači se ale nebude tato naše definice na řádcích 11 a 16 líbit, a pošle nám proto dvakrát varovnou zprávu:

```
[unchecked] unchecked call to add(E) as a member of the raw type
java.util.List
```

Tím se nás snaží upozornit, že v příkazech na těchto řádcích nevyužíváme možností, které nám nová verze nabízí, a trváme na ne zcela bezpečných „starých“ postupech.

V minulých verzích Javy nebylo možno deklarovat typ údajů ukládaných do kolekcí. Překladač proto nemohl zkontrolovat, zda do seznamu `názvy` ukládáme data povolených typů. Nic tedy nebránilo definovat metodu `zařaď_1(Object)`, která do fronty zařadila jakýkoliv objekt. To, že je ve frontě něco nepatřičného, jsme se pak mohli dozvědět až při příslušném volání metody `další()`, která vrací odkaz na prvek, jenž je zrovna na řadě.

Spustíte-li proto metodu `test()`, objeví se na standardním výstupu následující hlášení:

```
1 Texty ve frontě:
2    raz
3    dva
```

```
4    tři
5  Exception in thread "main" java.lang.ClassCastException:
6  java15.ptm.FrontaTextůStará
7    at java15.ptm.FrontaTextůStará.další(FrontaTextůStará.java:21)
8    at java15.ptm.FrontaTextůStará.test(FrontaTextůStará.java:38)
9    at java15.ptm.FrontaTextůStará.main(FrontaTextůStará.java:48)
```

Jako čtvrtý prvek jsme totiž zařadili do fronty samu frontu a při vybírání jejího obsahu jsme očekávali jenom texty. Problém je ale v tom, že jsme se z výpisu zásobníku sice dozvěděli, že máme ve frontě objekt, který tam nemá co dělat, a kde jsme na to přišli, ale nedozvěděli jsme se nic o tom, jak se tam tento objekt dostal.

Fronta textových řetězců ve verzi 5.0

Java 5.0 umožňuje díky PT kontrolu korektnosti typů vkládaných instancí. S vyžitím PT bychom mohli definovat frontu textových řetězců následovně:

```
 1 package rup.česky.java15.ptm.fronta;
 2
 3 import java.util.LinkedList;
 4 import java.util.List;
 5
 6 public class FrontaTextůNová
 7 {
 8   //Seznam odkazů na instance String - tuto skutečnost je třeba uvést
 9   //jak v typu seznamu, tak ve volání konstruktoru.
10   List<String> názvy = new LinkedList<String>();
11
12   public void zařaď( String název ) {
13     //Při vkládání odkazu na instanci správného typu
14     //se zdánlivě nic nemění - kontrola správnosti totiž není vidět
15     názvy.add( název );
16   }
17
18   public void zařaď_1( Object objekt ) {
19     //Po odkomentování způsobí následující příkaz syntaktickou chybu
20     //Podle deklarace lze do seznamu názvy ukládat pouze instance String
21 //    názvy.add( objekt );
22   }
23
24   public void zařaď_2( Object objekt ) {
25     //Chceme-li do seznamu názvy uložit odkaz deklarovaný jako Object,
26     //musíme jej nejprve přetypovat na String,jinak jej překladač nepřijme
27     názvy.add( (String)objekt );
28   }
29
30   public boolean isDalší() {
31     return (názvy.size() > 0);
```

```
32    }
33
34    public String další() {
35      //Při získávání odkazů z kontejneru již nemusíme vracené odkazy
36      //přetypovávat, to za nás udělá překladač.
37      String ret = názvy.get(0);
38      názvy.remove(0);
39      return ret;
40    }
41  }
```

Projděme si nyní novou definici řádek za řádkem a podívejme se, co se oproti předchozí definici změnilo.

První, čeho si jistě všimnete, je nová podoba deklarace seznamu `názvy` na řádku 10. Tentokrát je v ní využito PT `List<String>` – seznam je deklarován jako seznam instancí třídy `String`. Typ, jehož instance má daný seznam uchovávat, se zapisuje do špičatých závorek za název typu tohoto seznamu.

Jak je poznamenáno ve zdrojovém kódu, tuto skutečnost je třeba uvést nejenom při uvedení typu daného seznamu, ale také při volání příslušného konstruktoru. V naší deklaraci sice není možné vytvořit seznam jiného typu, ale v některých situacích to může být možné a občas i výhodné (časem si tuto možnost předvedeme).

Pokračujme ale v analýze programu. Metodu `zařaď(String)` definovanou na řádcích 12 až 16 překladač přeloží bez námitek, protože v ní zařazujeme do seznamu odkazy na instance správného typu, tj. typu `String`.

Naproti tomu metodu `zařaď_1(Object)` definovanou na řádcích 18 až 22 překladač přeložit odmítne, protože tato metoda nezaručuje vložení povoleného typu dat – proto je také zdrojový kód této metody uveden v komentáři.

Potřebujeme-li proto z nějakých důvodů opravdu metodu s parametrem typu `Object`, musíme ji upravit – např. tak, jak ukazuje metoda `zařaď_2(Object)` definovaná na řádcích 24 až 28. Případná chyba se tentokrát sice objeví stále až v době běhu, ale překladač nás alespoň přinutil upravit program tak, aby se projevila již při vkládání dat do kontejneru, a ne až při jejich vyjímání nebo dokonce až při jejich následném použití. Kdyby se proto vyskytla, bude se nám daleko lépe hledat její příčina.

```
1 Exception in thread "main" java.lang.ClassCastException:
2 java15.ptm.FrontaTextůNová
3   at java15.ptm.FrontaTextůNová.zařaď_2(FrontaTextůNová.java:27)
4   at java15.ptm.FrontaTextůNová.test(FrontaTextůNová.java:46)
5   at java15.ptm.FrontaTextůNová.main(FrontaTextůNová.java:60)
```

Metoda `další()` se pak drobně zjednodušila, protože už nemusíme vkládat do programu přetypování získaného odkazu. Z deklarace seznamu `názvy` totiž překladač ví, že metoda `get(int)` bude vracet odkazy na instance typu `String`, a potřebné přetypování vyzvedávaného objektu na typ `String` proto zabezpečí sám.

PT s několika typovými parametry

PT nemusí být charakterizované pouze jediným typovým parametrem, typových parametrů může být teoreticky libovolný počet. Všechny parametry se pak píší do společných špičatých závorek a oddělují se navzájem čárkami.

Typickými případy tříd, které ke své specifikaci potřebují více parametrů, jsou mapy – těm musíme zadat typ klíče a typ ukládané hodnoty, přičemž první se uvádí typ klíče a druhý typ hodnoty. Definujme např. metodu, která vygeneruje mapu naplněnou předem zadaným počtem dvojic (číslo)-(vyjádření čísla slovy).

```
 1 package rup.česky.java15.ptm.generátor;
 2
 3 import java.util.*;
 4
 5 import static rup.česky.společně.Slovy.slovy;
 6
 7 public class GenerátorMap
 8 {
 9   public static Map<Integer,String> generujMapu( int prvků ) {
10     Random rnd = new Random();
11     Map<Integer,String> mis = new HashMap<Integer,String>();
12     for( int i=0;   i < prvků;   i++ ) {
13       int číslo = rnd.nextInt(999);
14       mis.put( číslo, slovy( číslo ));
15     }
16     return mis;
17   }
18
19 //===== TESTY =======================================================
20
21   public static void testGenerátoruMap()
22   {
23     Map<Integer,String> mis = generujMapu( 5 );
24     System.out.println("Vygenerováno:" + mis );
25   }
26 }
```

Definice vlastního PT

Ukažme si nyní, jak bychom mohli definovat vlastní parametrizované třídy a rozhraní.

Definice třídy

Kdybychom chtěli předchozí definici zobecnit a umožnit zadání typu ukládaných dat až při vytváření příslušné fronty, museli bychom použít typové parametry, jejichž prostřednictvím bychom zadali, jaký typ prvků má právě vytvářené fronta ukládat a vracet. Používali bychom ji pak podobně, jako jsme v minulé definici používali datové typy `List` a `LinkedList`.

Protože definici třídy doplníme o typové parametry, nazveme ji `FrontaP`. Tato definice by mohla vypadat následovně:

```
 1 package rup.česky.java15.ptm.fronta;
 2
 3 import java.util.LinkedList;
 4 import java.util.List;
 5
 6 /** Instance takto definované třídy může pojmout instance různých typů,
 7  *  přičemž typ uchovávaných instancí se zadá až při vytváření
 8  *  příslušné instance fronty. */
 9 public class FrontaP<E>
10 {
11   //Seznam odkazů na instance String - tuto skutečnost je třeba uvést
12   //jak v typu seznamu, tak ve volání konstruktoru.
13   List<E> prvky = new LinkedList<E>();
14
15   public FrontaP()  {}
16
17   public FrontaP( E... ee)  {
18     for( E e : ee )
19       zařaď( e );
20   }
21
22   public void zařaď( E název ) {
23     //Při vkládání odkazu na instanci správného typu
24     //se zdánlivě nic nemění - kontrola správnosti totiž není vidět
25     prvky.add( název );
26   }
27
28   public boolean isDalší() {
29     return (prvky.size() > 0);
30   }
31
32   public E další() {
33     //Při získávání odkazů z kontejneru již nemusíme vracené odkazy
34     //přetypovávat, to za nás udělá překladač.
35     E ret = prvky.get(0);
36     prvky.remove(0);
37     return ret;
38   }
39
40 //===== TESTY =======================================================
41
42   public static void test() {
43     FrontaP<String> fString = new FrontaP<String>();
44     for( String s : new String[] { "raz", "dva", "tři" } )
45       fString.zařaď( s );
```

```
46     System.out.println( "\nFronta String po naplnění: " + fString.prvky );
47     while( fString.isDalší() ) {
48       System.out.print("   " + fString.další() );
49     }
50     System.out.println( "\nFronta po vyprázdnění: " + fString.prvky );
51
52
53     FrontaP<Double> fDouble = new FrontaP<Double>( 1d, 2d,  3d );
54     System.out.println( "\nFronta Double po naplnění: " + fDouble.prvky );
55     while( fDouble.isDalší() ) {
56       System.out.print("   " + fDouble.další() );
57     }
58     System.out.println( "\nFronta po vyprázdnění: " + fDouble.prvky );
59
60     // Následující příkaz by způsobil syntakticku chybu, protože
61     // konstruktor očekává parametry Double nebo alespoň double.
62 //    fDouble = new FrontaP<Double> ( 1, 2, 3 );
63 //    fDouble = new FrontaP<Integer>( 1, 2, 3 );
64     FrontaP<Integer> fInteger = new FrontaP<Integer>( 1, 2, 3 );
65     // Následující příkaz by způsobil syntakticku chybu, protože
66     //Typovým parametrem nesmí být primitiví typ
67 //      FrontaP<int> fInt = new FrontaP<int>( 1, 2, 3 );
68   }
69 }
```

Projděme si nyní tuto definici krok za krokem a podívejme se, jaké konstrukce jsme v ní použili.

Prvním zajímavým příkazem je samotná hlavička třídy, v níž jsme deklarovali použití formálního typového parametru `E`. Jak vidíte, formální typový parametr se deklaruje za názvem třídy ve špičatých závorkách.

Poznámka:

Identifikátory typových parametrů bývají většinou jednopísmenné. Tvůrci standardní knihovny zavedli konvenci, podle které se pro prvky (elementy) kontejnerů používá písmeno `E`, pro klíče (key) a hodnoty (value) v mapě písmena `K` a `V` a pro obecné typy písmeno `T`, případně jeho sousedi `S` a `U`.

Typový parametr jsme použili hned v definici seznamu `prvky`, kde jsme definovali atribut `prvky` jako zřetězený seznam. Ten bude obsahovat pouze instance třídy, jejíž název bude při konstrukci příslušné instance fronty předán jako skutečný typový parametr:

```
List<E> prvky = new LinkedList<E>();
```

Bezparametrický konstruktor je nezajímavý, takže se podíváme na konstruktor, jehož parametry jsou první hodnoty zařazené do vytvářené fronty. Tyto hodnoty jsou opět deklarovány jako instance třídy `E`.

Typové parametry mohou být použity nejenom k definici konkrétního použitého parametrizovaného typu, ale také k definici typu parametrů a typu návratové hodnoty definovaných metod. Jako příklad může sloužit metoda `zařaď(E)`, v níž je pomocí typového parametru specifikován typ parametru metody, a metoda `další()`, v jejíž definici je prostřednictvím typového parametru specifikován typ její návratové hodnoty.

V přiloženém testu jsou vytvořeny dvě fronty: fronta `fString`, do níž jsou zařazovány textové řetězce, a fronta `fDouble`, do niž jsou zařazovány instance třídy `Double`. Můžete si ověřit, že takto definované fronty nedovolí zařadit do fronty instance jiného než deklarovaného typu.

Na konci testu je několik zakomentovaných příkazů, které obsahují syntaktickou chybu vyvolanou pokusem o přiřazení hodnoty nesprávného typu.

První příkaz inicializuje frontu s prvky typu `Double` hodnotami typu `int`. V podkapitole *Omezení převodů* na straně 30 jsme si ale vysvětlili, že při používání automatického převodu z primitivních typů na příslušné obalové typy je třeba zadat převáděnou hodnotu správného typu. Požaduje-li proto metoda hodnoty typu `Double`, můžeme místo nich zadat hodnoty typu `double`, ale nemůžeme zadávat hodnoty typu `int`.

Druhý příkaz se pokouší tento problém napravit tím, že definuje frontu typu `Fronta<Integer>`. To by sice bylo přijatelné řešení, ale nesměl by se pokoušet uložit odkaz na tuto frontu do proměnné `fDouble`, která je typu `Fronta<Double>`. I když si za chvíli vysvětlíme, že se při překladu všechny typy očistí, takže pro virtuální stroj budou obě instance stejného typu, překladač podobná přiřazení neakceptuje.

Třetí příkaz ukazuje správné řešení: neexistuje-li proměnná potřebného typu, je třeba ji vytvořit a odkaz do ní uložit.

Poslední příkaz upozorňuje na to, že typovými parametry nesmí být primitivní datové typy, ale musí jimi být vždy třídy či rozhraní, aby jejich instance bylo možno převést na společného rodiče.

Definice rozhraní

Stejně jako třídu můžeme samozřejmě definovat i rozhraní. Představme si, že bychom např. chtěli definovat obecný generátor náhodných objektů, který bychom použili v jiných parametrizovaných třídách. Protože nebudeme předem znát typ objektů, které budeme chtít generovat, musíme generátor definovat parametrizovaný, abychom mohli skutečný typ jeho objektů určit až ve chvíli, kdy jej budeme znát.

Protože však generátor musí generovat reálné objekty, není vhodné obecný generátor definovat jako třídu, ale mnohem výhodnější je použít pro jeho zavedení rozhraní, které nebude omezováno skutečným typem generovaných objektů.

```
1 package rup.česky.java15.ptm.generátor;
2
3 public interface IGenerátor<E>
4 {
5   public E další();
```

```
6    public void start( long semeno );
7  }
```

Druhou možností je definovat abstraktní rodičovskou třídu společnou všem generátorům. I ta umožňuje, aby skutečný typ generovaných objektů definovali až její potomci. Nicméně přiznejme si, že současné programování preferuje před dědičností implementaci rozhraní:

```
 1  package rup.česky.java15.ptm.generátor;
 2
 3  public abstract class AGenerátor<E> implements IGenerátor<E>
 4  {
 5    static final java.util.Random rnd = new java.util.Random();
 6
 7    public abstract E další();
 8
 9    public void  start(long semeno) {
10      rnd.setSeed( semeno );
11    }
12  }
```

Potomci parametrizovaných typů

Parametrizované datové typy mohou vystupovat i v dědické hierarchii. Je asi zřejmé, že parametrizovaný datový typ může být potomkem neparametrizovaného typu – toho jsme doposud využívali, protože předkem naší třídy je neparametrizovaný typ `Object`.

Možné je ale i dědění od parametrizovaného typu. Představte si, že bychom chtěli naši frontu dovybavit schopností vracet na požádání iterátor a umožnit tak procházení hodnotami ve frontě prostřednictvím cyklu `for(:)`.

Jednou z možností, jak toho dosáhnout, je vytvoření potomka dosavadní fronty, který bude implementovat rozhraní `Iterable`. Protože jsme ale již znalci PT, nebudeme implementovat neozdobené rozhraní `Iterable`, jehož iterátory vrací odkazy na instance typu `Object`, které je třeba vždy nejdříve přetypovat, ale budeme implementovat rozhraní `Iterable<E>`, jehož iterátory vrací rovnou odkazy na instance typu `E`.

```
 1  package rup.česky.java15.ptm.fronta;
 2
 3  import java.util.Iterator;
 4  import java.util.Random;
 5
 6  public class FrontaPI<E> extends FrontaP<E> implements Iterable<E>
 7  {
 8    public FrontaPI( E... e ) {
 9      super( e );
10    }
11
```

```
12   public Iterator<E> iterator() {
13     return new Iterator<E>() {
14       int pořadí = 0;
15
16       public boolean hasNext() {
17         return (pořadí < prvky.size());
18       }
19       public E next() {
20         return prvky.get( pořadí++ );
21       }
22       public void remove() {
23         prvky.remove( 0 );
24       }
25     };
26   }
27
28 //===== TESTY =====================================================
29
30   public static void test() {
31     Random rnd = new Random();
32     FrontaPI<String> fronta = new FrontaPI<String>( "raz", "dva", "tři" );
33     fronta.tiskni();
34
35     System.out.print("Přidáváme:");
36     for( int i=0;   i < 3;   i++ ) {
37       //Generovaná náhodná čísla musím před vložením do fronty
38       //převést na String
39       String číslo = ""+rnd.nextInt(100);
40       fronta.zařaď( číslo );
41       System.out.println("\n  Přidáno: " + číslo);
42       System.out.print  (  "   Stav:");
43       for( String s : fronta )
44         System.out.print(" \"" + s + "\" " );
45     }
46     System.out.println("\nObsluhujeme:");
47     while( fronta.isDalší() ) {
48       System.out.print  ("     Stav:");
49       for( String s : fronta )
50         System.out.print(" \"" + s + "\" " );
51       String objekt = fronta.další();
52       System.out.println("\n Obslouženo: " + objekt);
53     }
54   }
55 }
```

Při dědění od PT musíme vždy zadat jeho typový parametr. V předchozím příkladu třída `FrontaPI` použila svůj vlastní typový parametr a předala jej jak rodiči, tak implementovanému rozhraní. Tento případ je relativně častý, ale není jediný možný.

Potomci PT nemusí být nutně parametrizovanými typy. Představte si, že bychom chtěli často používat naši frontu s textovými řetězci, a protože bychom byli líní neustále zadávat typový parametr, nadefinovali bychom si bezparametrického potomka, který je nevyžaduje. Jeho definice by byla velice jednoduchá:

```
 1 package rup.česky.java15.ptm.fronta;
 2
 3 import java.util.Random;
 4
 5 public class FrontaPIS extends FrontaPI<String>
 6 {
 7   public FrontaPIS( String... ss )
 8   {
 9     super( ss );
10   }
11
12 //===== TESTY ======================================================
13
14   public static void test() {
15     Random rnd = new Random();
16     FrontaPIS fronta = new FrontaPIS( "raz", "dva", "tři" );
17
18     System.out.print("Přidáváme:");
19     for( int i=0;   i < 3;   i++ ) {
20       //Generovaná náhodná čísla musím před vložením do fronty
21       //převést na String
22       String číslo = ""+rnd.nextInt(100);
23       fronta.zařaď( číslo );
24       System.out.println("\n  Přidáno: " + číslo);
25       System.out.print  (  "      Stav:");
26       for( String s : fronta )
27         System.out.print(" \"" + s + "\" " );
28     }
29     System.out.println("\nObsluhujeme:");
30     while( fronta.isDalší() ) {
31       System.out.print  ("     Stav:");
32       for( String s : fronta )
33         System.out.print(" \"" + s + "\" " );
34       String objekt = fronta.další();
35       System.out.println("\n Obslouženo: " + objekt);
36     }
37   }
38 }
```

Jako neparametrizované potomky bychom definovali např. i generátory jednotlivých datových typů – např.:

```
 1 package rup.česky.java15.ptm.generátor;
 2 
 3 public class GenString extends AGenerátor<String>
 4 {
 5   int rozsah;
 6 
 7   public GenString( int rozsah ) {
 8     this.rozsah = rozsah;
 9   }
10 
11   public String další() {
12     return rup.česky.společně.Slovy.slovy( rnd.nextInt( rozsah ) );
13   }
14 }
```

Definice metod s typovými parametry

Typové parametry je možno využít nejenom v definici třídy, ale i v definici jednotlivých metod, a to i v třídách, které samy nejsou definovány jako PT. Jinými slovy: i neparametrizované typy mohou obsahovat parametrizované metody.

V předchozích příkladech jsme např. několikrát použili tisk obsahu fronty. Možná vás už napadlo, zda definici fronty nedoplnit o metodu:

```
1 public  void tiskni() {
2   System.out.print( "[" );
3   for( E e : this )
4     System.out.print( " <" + e + "> " );
5   System.out.println( "]" );
6 }
```

To je samozřejmě možné. Ale zkuste se vžít do situace, kdy je definice fronty součástí nějaké knihovny, kterou obdržíte jako soubor jar, a vy byste chtěli její možnosti rozšířit právě o tisk.

Definovat potomka, kterého rozšíříte o metodu pro tisk, nebývá v takovém případě vždy nejvhodnější, protože řada jiných tříd z knihovny již často původní třídu používá, takže by se jich vaše rozšíření nedotýkalo.

Můžete ale definovat knihovní třídu, v níž potřebnou metodu definujete. Jedna z možností, jak takovou třídu definovat, je následující:

```
1 package rup.česky.java15.ptm.fronta;
2 
3 public final class FrontaUtil
4 {
5   //Knihovní třída zakazuje vytvářet své instance
6   private FrontaUtil() {}
7 
```

```
 8   public static <E> void tiskni( String nadpis, FrontaPI<E> fpi ) {
 9     System.out.print( nadpis + ": [" );
10     for( E e : fpi )
11       System.out.print(" <" + e + "> " );
12     System.out.println("]");
13   }
14
15 //===== TESTY ==========================================================
16
17   public static void testTiskni() {
18     FrontaPI<String> fString = new FrontaPI<String>("raz", "dva", "tři");
19     tiskni( "FrontaPI<String>", fString);
20
21     FrontaPIS fpis = new FrontaPIS( "Adam", "Bedřich",  "Cyril" );
22     tiskni( "FrontaPIS", fpis );
23
24     FrontaPI<Integer> fInt = new FrontaPI<Integer>( 10, 20, 30 );
25     tiskni( "FrontaPI<Integer>", fInt );
26   }
27 }
```

Všimněte si, že jsme definovali parametrizovanou metodu ve třídě, která sama nebyla parametrizovaná. Takové definice nejsou žádnou výjimkou.

Dále si všimněte, že typový parametr se v definici metody zadává před typem návratové hodnoty, aby jej bylo možno použít už v zadání návratové hodnoty (za chvíli si to ukážeme).

Teď vám ale prozradím, že jsme při definici metody pro tisk typový parametr používat vůbec nemuseli. Přestože jsme třídu `FrontaIP` definovali jako PT, můžeme ji v některých situacích používat i v surovém stavu neozdobenou typovými parametry (o zdobení a očišťování typů se zmíníme podrobněji v kapitole *Parametrizace a očišťování* na straně 62). Protože typ prvků uložených ve frontě vlastně vůbec nepotřebujeme, můžeme metodu definovat také následovně:

```
1 //Metoda je definována ve třídě rup.česky.java15.ptm.fronta.FrontaUtil
2 public static void tiskniFrontu( String nadpis, FrontaPI fpi ) {
3   System.out.print( nadpis + ": [" );
4   for( Object o : fpi )
5     System.out.print(" <" + o + "> " );
6   System.out.println("]");
7 }
```

Tuto definici překladač bez problému přeloží a obdržené výsledky budou totožné s definicí předchozí.

Jsou ale situace, kdy je použití typových parametrů nanejvýš vhodné. Představte si, že by třída s frontou byla součástí nějaké knihovny, a my bychom chtěli definovat pomocnou knihovní třídu s rozšiřujícími metodami – např. s metodou, která vrátí prvek, jenž je zrovna na řadě, ale nebude jej z fronty odebírat.

Pokud bychom v této definici nepoužili typový parametr, museli bychom vracet instanci typu `Object` a tu pak v programu přetypovávat. Výhodnější proto je definovat metodu jako parametrizovanou (stále přidáváme metody do třídy `FrontaUtil`).

```
 1 //Metody jsou definovány ve třídě rup.česky.java15.ptm.fronta.FrontaUtil
 2
 3 public static <E> E naŘadě( FrontaPI<E> fpi ) {
 4   Iterator<E> it = fpi.iterator();
 5   if( it.hasNext() )
 6     return it.next();
 7   else
 8     return null;
 9 }
10
11 //===== TESTY ================================================
12
13 public static void testNaŘadě() {
14   FrontaPI<String> fString = new FrontaPI<String>( "raz", "dva", "tři" );
15   tiskni( "FrontaPI<String>", fString);
16   Iterator<String> its = fString.iterator();
17   while( its.hasNext() ) {
18     String s = naŘadě( fString );
19     System.out.print("Na řadě je: " + s);
20     s = fString.další();
21     System.out.println("  --  Obsloužen: " + s);
22   }
23
24   FrontaPI<Integer> fInt = new FrontaPI<Integer>( 10, 20, 30 );
25   tiskni( "\nFrontaPI<Integer>", fInt );
26   Iterator<Integer> iti = fInt.iterator();
27   while( iti.hasNext() ) {
28     int i = naŘadě( fInt );
29     System.out.print("Na řadě je: " + i);
30     i = fInt.další();
31     System.out.println("  --  Obsloužen: " + i);
32   }
33 }
```

Volání parametrizovaných metod

V doprovodném testu si všimněte, že na rozdíl od deklarací tříd a volání konstruktorů jsme při vyvolávání metody nezadávali žádné hodnoty typových parametrů. Z typu parametru metody si totiž překladač většinou odvodí dostatek informací pro to, aby správný typový parametr přiřadil sám.

Jsou ale situace, kdy to překladač odhadnout nedokáže nebo kdy máme nějaké speciální požadavky a chceme, aby metoda pracovala s jinou hodnotou typového parametru. Pak musíme typový parametr zadat.

Hodnoty typových parametrů se při volání metod zadávají do špičatých závorek před identifikátor volané metody, avšak za tečku následující za její kvalifikací. Metodám volaným bez kvalifikace proto není možno typové parametry zadat. Vše ukazuje následující příklad:

```
 1 package rup.česky.java15.ptm.demo;
 2 
 3 public class Metody
 4 {
 5   public static <T> List<T> dvojice( T prvek ) {
 6     List<T> ret = new ArrayList<T>();
 7     ret.add( prvek );
 8     ret.add( prvek );
 9     return ret;
10   }
11 
12   public void volání() {
13     Metody m = new Metody();
14     List<String> ls = dvojice( "Text" );
15     List<Object> lo;
16 //    lo = dvojice( "Text");  //Nesouhlasí typy - nelze
17 //    lo = <Object>dvojice("objekt");   //Nekvalifikované volání - nelze
18     lo = Metody.<Object>dvojice("Kvalifikace třídou");
19     lo = m.<Object>dvojice( "Kvalifikace instancí");
20     lo = dvojice( (Object)"Řešení přetypováním" );
21   }
22 }
```

V předchozích případech byl parametrem metody `dvojice` vždy textový řetězec. Pokud jsme proto hodnotu typového parametru nezadali explicitně, překladač dosadil typovému parametru hodnotu `String` a vracel návratovou hodnotu typu `List<String>`.

Jak si vysvětlíme za chvíli, parametrizované typy s různými hodnotami svých typových parametrů jsou neslučitelné. Do proměnné typu `List<Object>` proto není možné přiřadit odkaz na instanci typu `List<String>`. Proto je příkaz na řádku 16, který se o to pokouší, zakomentován, protože jinak jej překladač označí za syntakticky chybný.

Musíme proto zařídit, aby metoda `dvojice` vracela potřebný typ – toho dosáhneme explicitním zadáním typových parametrů. Jak jsme si ale řekli, můžeme je zadat pouze při kvalifikovaném volání. Proto je příkaz s nekvalifikovaným voláním na řádku 17 také zakomentován, protože i on je syntakticky chybný.

V pořádku jsou až následující dva příkazy na řádcích 18 a 19. První kvalifikuje volání metody `dvojice` třídou, druhý instancí této třídy (jak víme, to je u metod třídy možné, i když to není z hlediska čistoty programu doporučované).

Poslední příkaz ukazuje alternativní možnost řešení daného problému: vhodné přetypování parametru, z jehož typu překladač odvozuje hodnoty typového parametru.

Já bych vám ale doporučoval spíše explicitní uvádění typových parametrů, protože takto sdělíte své požadavky mnohem průzračněji.

Omezení použitelných hodnot typových parametrů

V některých situacích bychom potřebovali, aby hodnoty typových parametrů splňovaly nějaké podmínky. Nejčastěji požadujeme, aby měly definované předky nebo potomky či aby implementovaly definovaná rozhraní.

Pokud bychom např. chtěli definovat třídu `IntervalU`, která by reprezentovala uzavřený interval hodnot, potřebovali bychom zajistit, aby instance datového typu, jehož interval budeme definovat, byly vůbec porovnatelné, tj. aby daný datový typ byl potomkem typu `Comparable`. Potřebná definice by mohla vypadat následovně:

```
 1 package rup.česky.java15.ptm.interval;
 2 
 3 public class IntervalU<T extends Comparable<T>>
 4 {
 5   T dolní, horní;
 6 
 7   public IntervalU( T dolní, T horní ) {
 8     if( dolní.compareTo( horní ) > 0 )
 9       throw new IllegalArgumentException(
10         "Dolní mez nemůže být větší než horní" );
11     this.dolní = dolní;
12     this.horní = horní;
13   }
14 
15   public T getDolní() {
16     return dolní;
17   }
18 
19   public T getHorní() {
20     return horní;
21   }
22 
23   public void setDolní( T dolní ) {
24     if( dolní.compareTo(horní) > 0 )
25       throw new IllegalArgumentException(
26         "Příliš vysoká dolní mez: dolní=" + dolní + ", horní=" + horní);
27     this.dolní = dolní;
28   }
29 
30   public void setHorní( T horní ) {
31     if( dolní.compareTo(horní) > 0 )
32       throw new IllegalArgumentException(
```

```
33         "Příliš nízká horní mez: dolní=" + dolní + ", horní=" + horní);
34     this.horní = horní;
35   }
36
37   public boolean uvnitř( T t ) {
38     return (dolní.compareTo( t ) <= 0)  &&
39          (t.compareTo( horní ) <= 0);
40   }
41
42   public String toString() {
43     return "<" + dolní + ";" + horní + ">";
44   }
45
46 //===== TESTY ==============================================
47
48   public static <T extends Comparable<T>>
49        void porovnej( T a, T b, T ... tt ) {
50     IntervalU<T> in = new IntervalU<T>( a, b );
51     System.out.println("\nInterval: " + in);
52     for( T t : tt )
53       System.out.println( t + " leží uvnitř: " + in.uvnitř(t));
54   }
55
56   public static void test() {
57     porovnej("Aleš", "Cyril", "Běta", "Teta", "Cyril", "Bohouš", "Adam");
58     porovnej( 10, 20, 30, 15, 10, 5  );
59   }
60 }
```

V předchozím programu vás možná zarazilo, že u typového parametru `T` je uvedeno `extends Comparable<T>` místo očekávaného `implements Comparable<T>`. Definice jazyka totiž zavádí společné klíčové slovo `extends` pro vyjádření dědičnosti tříd i implementace rozhraní.

U typového parametru je možno definovat i požadavek na současnou implementaci několika rozhraní. V takovém případě jsou jednotlivá rozhraní (a případně i rodičovská třída) oddělována znakem `&` – např.:

```
public chytráTřída< T extends Comparable & Serializable >
```

Je-li přitom definovaná třída potomkem jiné třídy, musí být tato rodičovská třída uvedena jako první.

Definice PT s několika parametry

Před chvílí jsem vysvětloval, že pro oddělení názvu rodičovské třídy a jednotlivých implementovaných rozhraní se používá znak `&`. Možná některé z vás napadlo, proč se jednotlivá implementovaná rozhraní neoddělují čárkou, jako je tomu v hlavičce třídy. Je to proto, že čárka má již přiřazen jiný význam: slouží k oddělení jednotlivých typo-

vých parametrů v případě, kdy jich zadáváme více (viz pasáž *PT s několika typovými parametry* na straně 46).

Pokud bychom se např. rozhodli definovat univerzální přepravku určenou pro uchování a předávání tří instancí předem neznámých objektových typů, mohli bychom ji definovat např. následovně:

```
 1 package rup.česky.java15.ptm.přepravka;
 2 
 3 import java.util.*;
 4 
 5 /** Univerzální přepravka trojice objektů. */
 6 public class Přepravka123<T1, T2, T3>
 7 {
 8   T1 t1;
 9   T2 t2;
10   T3 t3;
11 
12   public Přepravka123( T1 p1, T2 p2, T3 p3 ) {
13     t1 = p1;   t2 = p2;   t3 = p3;
14   }
15 
16   public T1 get_1()  { return t1; }
17   public T2 get_2()  { return t2; }
18   public T3 get_3()  { return t3; }
19 
20   public void set_1( T1 p1 )  { t1 = p1; }
21   public void set_2( T2 p2 )  { t2 = p2; }
22   public void set_3( T3 p3 )  { t3 = p3; }
23 
24 //===== TESTY ========================================================
25 
26   /** Metoda vracející instanci přepravky. */
27   public static <T extends Comparable<T>>
28        Přepravka123<T, T[], T> meze( T[] pole )
29   {
30     T min, max;
31     min = max = pole[0];
32     for( T prvek : pole ) {
33       if( min.compareTo( prvek ) > 0 )
34         min = prvek;
35       if( max.compareTo( prvek ) < 0 )
36         max = prvek;
37     }
38     return new Přepravka123<T, T[], T>( min, pole, max );
39   }
40 
41   /** Metoda očekávající přepravku jako parametr. */
42   public static <T> void tiskni( Přepravka123<T,T[],T> p ) {
```

```
43     List<T> seznam = Arrays.asList( p.get_2() );
44     System.out.println("Pole " + seznam +
45       "\nmá minimum " + p.get_1() +
46       "\n a maximum " + p.get_3() );
47   }
48
49   public static void test() {
50     tiskni( meze( new String[] { "raz", "dva", "tři", "čtyři", "pět"} ) );
51     tiskni( meze( new Integer[] { 9, 5, 1, 3, 7, 2, 4 } ) );
52   }
53 }
```

Vzájemné závislosti typových parametrů

V předchozím příkladu byly typové parametry navzájem nezávislé. Občas však potřebujeme definovat třídu, jejíž typové parametry na sobě navzájem závisí. Pokud bychom např. chtěli demonstrovat rozdílnou výkonnost různých implementací rozhraní `List`, mohli bychom příslušnou testovací třídu definovat následovně (můžete si vyzkoušet, že na pořadí typových parametrů v hlavičce třídy nezáleží):

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 import java.util.*;
 4
 5 public class TestSeznamů < E, L extends List<E>>
 6 {
 7   private String        název;
 8   private List<E>       seznam;
 9   private IGenerátor<E> gen;
10   private int           počet;
11
12   public TestSeznamů(String název,L seznam,IGenerátor<E> gen,int počet ) {
13     this.název  = název;
14     this.seznam = seznam;
15     this.gen    = gen;
16     this.počet  = počet;
17   }
18
19   /** Metoda spouští nad seznamem zadanou posloupnost testů. */
20   public void testuj() {
21     long start, stop;
22     //Nemožnost vytvořit pole instancí typového parametru je třeba obejít
23     //vytvořením pole objektů
24     E[] prvky = (E[])(new Object[ počet ]);
25     gen.start( 0 );
26     for( int i=0;   i < prvky.length;   prvky[i++] = gen.další() );
27     System.out.println("\n\nTest: " + název);
28
```

```
29      start = System.nanoTime();
30      for( E e : prvky )
31        seznam.add( e );
32      stop = System.nanoTime();
33      System.out.println( "\nPřidání " + počet + " prvků typu " +
34        prvky[0].getClass() + " trvalo " + (stop-start) + " nanosekund." );
35
36      start = System.nanoTime();
37      int střed;
38      while( (střed = seznam.size()) > 0 )
39        seznam.remove( střed>>1 );
40      stop = System.nanoTime();
41      System.out.println( "Odebrání ze středu " + počet + " prvků typu " +
42        prvky[0].getClass() + " trvalo " + (stop-start) + " nanosekund." );
43    }
44
45  //===== TESTY ==========================================================
46
47    /** Spuštění metody testuj() nad různými druhy seznamů. */
48    public static void test() {
49      TestSeznamů ts;
50      int počet = 1000;
51      ts = new TestSeznamů<String, ArrayList<String>>
52          ( "ArrayList obsahující String", new ArrayList<String>(),
53            new GenString(1000), počet );
54      ts.testuj();
55
56      ts = new TestSeznamů<Integer, LinkedList<Integer>>
57          ( "LinkedList obsahující Integer", new LinkedList<Integer>(),
58            new GenInteger(1000), počet );
59      ts.testuj();
60    }
61  }
```

Vzájemná závislost typových parametrů může být ještě těsnější – jeden typový parametr může být např. deklarován jako potomek druhého.

```
 1  package rup.česky.java15.ptm.demo;
 2
 3  class Závislost<Předek, Potomek extends Předek>
 4  {
 5    Předek  předek;
 6    Potomek potomek;
 7
 8    Závislost( Předek p, Potomek pp ) {
 9      předek  = p;
10      potomek = pp;
11    }
12  }
```

Při takto těsné závislosti ale již začne záležet na pořadí deklarace typových parametrů a překladač vám nedovolí uvést v seznamu parametrů potomka před předkem.

Parametrizace a očišťování

Většinou není nutné, aby programátor znal detaily toho, jak překladač převádí zdrojový kód do bajtkódu, ale v případě PT je základní porozumění tomuto mechanizmu nutné. Proto mu bude věnována tato podkapitola.

Jak jsem již řekl v úvodu, jedním z hlavních požadavků, které bylo nutno při specifikaci rozšíření nové verze Javy dodržet, bylo zachování zpětné kompatibility. To umožní vytvářet programy bezproblémově spolupracující s programy napsanými v dřívějších verzích.

Zpočátku šla snaha o maximální kompatibilitu ještě dál. Autoři se snažili zavést nové konstrukce tak, aby bylo možno stále generovat kód, který by využíval nově zavedená rozšíření syntaxe a zároveň byl spustitelný i na starších verzích virtuálních strojů. Ještě první beta-verze nabízela možnost nastavit nezávisle verzi zdrojového kódu a verzi přeloženého kódu. Postupně se však od této možnosti ustoupilo a současná verze Javy nic podobného nenabízí. Budete-li chtít využívat programových konstrukcí, které nová verze nabízí, budete muset výsledné programy spouštět také na nové verzi virtuálního stroje.

Nová verze sice nenabízí možnost generace kódu využívajícího nové prvky jazyka pro dřívější verze virtuálních strojů, avšak na druhou stranu byla všechna rozšíření koncipována tak, aby byla zachována možnost bezproblémové komunikace nově vytvořených programů s programy vytvořenými ve starších verzích jazyka.

Poznámka:
Chcete-li využívat výhod nových konstrukcí a zachovat přitom kompatibilitu přeložených souborů s některou ze starších verzí virtuálních strojů, navštivte `http://sourceforge.net/projects/retroweaver` *a seznamte se s programem* Retroweaver*, který upraví vaše class-soubory tak, aby mohly běžet na starších typech virtuálních strojů.*

Zachování této kompatibility bylo dosaženo především díky koncepci očišťování parametrizovaných datových typů od typových parametrů a jejich převodu do neparametrizovaného tvaru, s nímž si budou vědět rady i dříve napsané programy.

Ještě jednou terminologie

Než se o implementaci parametrizovaných datových typů rozhovořím podrobněji, zavedeme si nejprve několik pojmů.

Termínem **ozdobený datový typ** budu označovat datové typy doplněné o typové parametry. V příkladu s frontou byly ozdobenými typy rozhraní `List<String>` a třída `LinkedList<String>`.

Překladač zbavuje v průběhu překladu datové typy jejich případného zdobení. Tento proces budu označovat jako **čištění** nebo **očištění** (anglicky *erasure*) datových typů.

Očištěné (nebo prostě jen neozdobené) datové typy jsou ve specifikaci označovány jako *raw types*. Protože slovo *raw* lze překládat mnoha způsoby a v různých kontextech jsou různé z nich vhodné, budu datové typy bez přídavných typových parametrů označovat střídavě jako **surové**, **neozdobené** nebo **očištěné** (je to obráceně než u cukru – surový cukr je nečištěný). V příkladu s frontou bychom mohli za surové datové typy označit (vedle řady dalších) např. typy `List` a `LinkedList`.

Očištění parametrizovaného kódu

Kdykoliv použijete v programu nějaký ozdobený datový typ, překladač jej v průběhu překladu od jeho ozdob očistí a do class-souboru jej uloží jako očištěný, surový typ. Místo neomezovaných typových parametrů pak dosadí typ `Object` a místo omezovaných dosadí první z uvedených omezujících typů.

Protože předchozí popis nemusí být každému zcela jasný, tak jej pro přiblížení vysvětlím na příkladu. Představme si třídu `Čištění_Před` definovanou následovně:

```
1  package rup.česky.java15.ptm.demo;
2
3  import java.io.Serializable;
4
5  class Čištění_Před <CS extends Comparable<CS> & Serializable,
6                      SC extends Serializable & Comparable<SC>>  {
7    CS cs;
8    SC sc;
9
10   void upravCS( CS p ) { if(cs.compareTo(p) < 0)  cs = p; }
11   void upravSC( SC p ) { if(sc.compareTo(p) < 0)  sc = p; }
12 }
```

Přeložíte-li ji a pošlete na přeložený kód nějaký zpětný překladač (dekompilátor), dostanete následující podobu třídy (přejmenoval jsem ji na `Čištění_Po`):

```
1  package rup.česky.java15.ptm.demo;
2
3  import java.io.Serializable;
4
5  class Čištění_Po
6  {
7    Comparable   cs;
8    Serializable sc;
9
10   void upravCS(Comparable p) {
11     if(cs.compareTo((Object)p) < 0)  cs = p;
12   }
13
```

```
14   void upravSC(Serializable p) {
15     if(((Comparable)sc).compareTo((Object)p) < 0)  sc = p;
16   }
17 }
```

Nyní se vrátíme k jednotlivým konstrukcím a podíváme se na to, jak při jejich čištění překladač postupuje.

Pořadí uvedení omezujících tříd a rozhraní

Jak jsem již řekl, má-li některý z typových parametrů více omezení (v naší ukázce typové parametry `CS` a `SC`), nahradí se tento typový parametr při čištění prvním z nich. (Proto také musí být případná rodičovská třída uvedena jako první.)

V naší ukázkové třídě jsem toto pravidlo demonstroval na atributech. První z nich byl proto v očištěném kódu deklarován podle tohoto pravidla jako instance rozhraní, `Comparable` druhý jako instance rozhraní `Serializable`.

V místech, kde je třeba typový parametr interpretovat jako instanci některého z dalších implementovaných rozhraní, použije překladač přetypování.

Porovnáte-li v ukázce práci s atributy `cs` a `sc` nebo volání metod `upravCS()` a `upravSC()`, bude vám hned jasné, proč je při implementaci několika rozhraní vhodné uvést jako první to, které je ve zdrojovém textu nejčastěji používané, a muselo by se proto na něj nejčastěji přetypovávat.

Naopak rozhraní, která nevyžadují implementaci žádných metod a jsou pouze informační (takovým je např. rozhraní `Serializable`), je při výskytu více současně implementovaných rozhraní vhodné uvádět až na konci sady deklarovaných implementovaných rozhraní.

Očištění výrazů

Jsou-li ve výrazech použity instance některého z typových parametrů, pak řešení závisí na tom, odpovídá-li deklarovaný typ aktuálnímu požadovanému typu. Není-li tomu tak, překladač příslušnou instanci v okamžiku potřeby přetypuje.

Automatické přetypování přichází samozřejmě v úvahu pouze v případě, že odpovídá deklaraci daného typového parametru. Tak byl např. v ukázkové třídě přetypován v případě potřeby atribut `sc` na `Comparable`.

V některých situacích ale zjistíte, že vaše představy o realizovatelnosti některých operací se od představ překladače výrazně liší. V takových situacích musíte občas vymýšlet nejrůznější kličky, abyste překladač přesvědčili, že má váš kód akceptovat. Podívejte se na následující metodu:

```
1 //Metoda je definována ve třídě rup.česky.java15.ptm.demo.InstanceOf
2 public static <T extends Comparable<T>> void whoAmI( T a ) {
3   System.out.println("Object: " + (a instanceof Object) );
4   System.out.println("Comparable: " + (a instanceof Comparable) );
5 //  System.out.println("Double: " + (a instanceof Double) );
```

```
6    System.out.println("Double 0: " + ((Object)a instanceof Double) );
7    System.out.println("Double 1: " + (a.getClass() == Double.class) );
8    System.out.println("Double 2: " + (Double.class.isInstance(a)) );
7  }
```

Odkomentujete-li v ní řádek testující, zda se nejedná o instanci třídy `Double` (případně instanci jakékoliv jiné třídy implementující `Comparable` a tím pádem dosaditelné za `T`), překladač ohlásí syntaktickou chybu:

```
found   : T
required: java.lang.Double
    System.out.println("Double: " + (a instanceof Double) );
```

Překladač např. nepovoluje použití operátoru `instanceof` s pravým operandem, který by mohl být některým z možných typů levého operandu, jehož typ je parametrizovaný. V takovémto případě bude asi nejjednodušším řešením přetypovat levý operand na `Object` a operátor `instanceof` pak aplikovat na přetypovaný operand. Jak ale vyčtete ze zdrojového kódu, možné jsou i jiné postupy.

Ztráta informace při běhu

Jak jsme již řekli, veškeré informace o typových parametrech, které do zdrojového kódu zadáte, jsou určeny pro překladač. Při běhu programu je již není možné použít. Různé testy, které se vyhodnocují až za běhu programu, proto nemohou brát tyto informace v úvahu. Např. podmínka

```
seznam instaceof List<String>
```

může otestovat pouze to, je-li proměnná `seznam` instancí rozhraní `List`. O tom, co má daný seznam obsahovat, se při běhu již nic nedozvíte.

Obdobně i operátor přetypování může přetypovat svůj operand pouze na základní typ, avšak nemůže již zabezpečit použití dodatečné informace. Příkaz

```
List<String> seznam1 = (List<String>) seznam2;
```

bude v přeloženém kódu pracovat naprosto stejně, jako kdybyste místo parametrizovaných typů použili pouze základní typ `List`. Jediný rozdíl bude v tom, že při přetypování na surový, neparametrizovaný typ pošle překladač varovnou zprávu.

Přemosťovací metody

Odvodíte-li od PT nějakého potomka, může se stát, že pro něj budete potřebovat překrýt některou ze zděděných metod. Ne vždy ale budete mít k dispozici typový parametr rodičovské nebo dokonce prarodičovské třídy.

Přestavte si, že součástí knihovny, kterou zakoupíte a používáte, bude např. třída `FrontaPIS`, kterou jsme definovali na str. 52, a že budete potřebovat upravit její schopnosti tak, aby do fronty ukládala svěřené texty převedené na velká písmena. Vytvoříte proto jejího potomka, který by mohl být definován např. následovně:

```
1 package rup.česky.java15.ptm.fronta;
2
3 public class FrontaPISU extends FrontaPIS
4 {
5   public void zařaď( String s ) {
6     super.zařaď( s.toUpperCase() );
7   }
8 }
```

Vše je v pořádku až do chvíle, kdy si uvědomíte, že třída `FrontaPIS` je vlastně potomkem třídy `FrontaP`, v níž je ale po očištění definována příslušná metoda jako `zařaď(Object)`, kdežto v naší třídě jsme ji definovali jako `zařaď(String)`. Metoda proto původní metodu nepřekrývá, avšak překrývat by ji měla!

Řešením není definovat metodu jako `zařaď(Object)`, protože pak bychom přišli o typovou kontrolu, kvůli které ale byly typové parametry a parametrizované typy do Javy zavedeny.

Překladač proto postupuje obráceně. Definuje **přemosťovací metodu** (*bridge method*), která překryje původní metodu `zařaď(Object)` a jejíž tělo bude tvořeno jediným příkazem: voláním nově definované metody `zařaď(String)`.

Obdobný problém nastane, budeme-li definovat metodu, která místo původního očištěného typu bude vracet aktuální typ platný pro danou třídu. Tady je ale situace složitější. Bude-li totiž takováto metoda bez parametrů (a to je u přístupových metod typu `getXxx()` běžné), definujete tím metodu, která se od původní metody neliší v počtu a/nebo typu deklarovaných parametrů, ale pouze v typu návratové hodnoty. Jenže něco takového bylo až do minulé verze zakázané!

Pokud se autoři nové verze chtěli vyhnout problémům, museli podobné konstrukce alespoň v nějakém rozsahu povolit. Postup je shodný jako v předchozím případě: překladač definuje skrytou přemosťovací metodu, která zavolá metodu nově definovanou a vrácenou hodnotou příslušným způsobem přetypuje.

Poznámka:

Možnost překrýt zděděnou metodu metodou vracející podtyp typu vraceného překrytou metodou není omezen pouze na parametrizované typy. Můžete jej použít i u běžných potomků běžných tříd.

Ukažme si vše na potomkovi naší demonstrační třídy, kterého nazveme `DemoPřed_2`.

```
1 package rup.česky.java15.ptm.demo;
2
3 class Čištění_Před_2 extends Čištění_Před<String, String>  {
4
5   void upravCS( String p ) {
6     if(cs.compareToIgnoreCase(p) < 0)
7       cs = p;
8   }
```

```
 9
10   void upravSC( String p ) {
11     if(sc.compareToIgnoreCase(p) < 0)
12       sc = p;
13   }
14 }
```

Předáte-li opět její přeložený class-soubor nějakému zpětnému překladači, obdržíte následující výsledek:

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 import java.io.Serializable;
 4
 5 class Čištění_Po_2 extends Čištění_Po {
 6
 7   Čištění_Po_2() {}
 8
 9   void upravCS(String p) {
10     if(((String)cs).compareToIgnoreCase(p) < 0)
11       cs = ((Comparable) (p));
12   }
13
14   void upravSC(String p) {
15     if(((String)sc).compareToIgnoreCase(p) < 0)
16       sc = ((Serializable) (p));
17   }
18
19 //===== Překladačem přidané přemosťovací metody =====================
20
21 //  volatile
22   void upravSC(Serializable x0) {
23     upravSC((String)x0);
24   }
25
26 //  volatile
27   void upravCS(Comparable x0) {
28     upravCS((String)x0);
29   }
30 }
```

Poznámka:

Modifikátor `volatile` *na řádcích 21 a 26 je zakomentován proto, že je sice při zpětném překladu u překladačem přidaných metod uveden, ale vy jej použít nesmíte (proto jsem jej zakomentoval). Předpokládám, že jím si překladač označuje metody, které do kódu přidal sám.*

Spolupráce s programy vytvořenými v předchozích verzích

V předchozích verzích Javy byly vytvořeny miliony řádků kódu a nikomu se nejspíš nebude chtít tyto vyzkoušené programy upravovat tak, aby odpovídali nové syntaxi Javy 5.0.

Při překladu kódu používajícího neparametrizovaných (surových) verzí některých datových typů, překladač velice často znejistíme, protože nás nemůže zkontrolovat, jestli používáme správné datové typy. V minulých verzích Javy mu to sice nevadilo, ale tehdy jsme také neměli prostředky k tomu, abychom se s tímto problémem vyrovnali. Nyní tyto prostředky máme, takže se nás překladač snaží upozornit na každý příkaz, kde je nevyužijeme.

To, jestli nám pouze prozradí, že podle něj kód není dokonalý, anebo nám blíže specifikuje, čeho jsme se dopustili, je možno nastavit pomocí parametrů příkazového řádku. Podrobnosti o nastavování těchto parametrů najdete v příloze *Varovná hlášení překladače* na stránce 147.

Nastavte varovná hlášení a zkuste přeložit následující program – za každý příkaz, který vede k vydání varovného hlášení, jsem zapsal do řádkového komentáře příslušné hlášení.

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 import java.util.*;
 4
 5 public class Spolupráce<T>
 6 {
 7   List<T> seznam;
 8
 9   Spolupráce( List<T> lst ) { seznam = lst; }
10
11   public static void spolupráceSeStaršímKódem() {
12     //Vytvoření seznamu celých čísel
13     List<Integer> iList = new LinkedList<Integer>();
14     for( int i=1;   i <= 3;   i++ )
15       iList.add( i );
16     System.out.println("Seznam: " + iList);
17
18     //Vytvoření surového, neparametrizovaného seznamu
19     List starý = new LinkedList();
20     for( int i=1;   i <= 3;   i++ )
21       starý.add( 10*i );
22       //warning: [unchecked] unchecked call to add(E)
23       //as a member of the raw type java.util.List
24
25     //Přiřazení parametrizovaného seznamu neparametrizovanému
26     iList = starý;
```

```
27      //warning: [unchecked] unchecked conversion
28
29      System.out.println("Starý: " + iList);
30
31      List<String> sList = new LinkedList();
32      //warning: [unchecked] unchecked conversion
33
34  //    iList = sList;
35  //    //incompatible types
36  //    //found   : rup.česky.java15.ptm.Pokus<java.lang.String>
37  //    //required: rup.česky.java15.ptm.Pokus<java.lang.Integer>
38
39      iList = (List)sList;
40
41      //Chování není omezeno pouze na třídy ze standardní knihovny,
42      //ale vztahuje se na všechny třídy definované jako PT
43      FrontaPI fpi = new FrontaPI( 100, 200, 300 );
44      //warning: [unchecked] unchecked call to FrontaPI(E...) as a member
45      //of the raw type rup.česky.java15.ptm.FrontaPI
46      System.out.println("FrontaPI: " + fpi);
47    }
48  }
```

Všimněte si, že varovným hlášením jsou označeny prakticky všechny operace, které pracují se surovými, neozdobenými typy dat. Je přitom jedno, jedná-li se o operace s daty ze standardní knihovny nebo o operace s právě definovanými PT.

Varovné hlášení neříká, že je v daném místě programu chyba (i když by mohla být). Spíš se vás snaží upozornit, abyste se na daný příkaz ještě jednou podívali a přesvědčili se, že jste jej chtěli naprogramovat právě takto.

Můžete to chápat tak, že varovná hlášení označují místa, ve kterých nevyužíváte některé z nových možností jazyka, a snižujete tak bezpečnost na úroveň programů napsaných v jeho minulých verzích.

Vypínání typové kontroly

Důvodů, proč potřebujeme používat neozdobené typy, je několik. Kromě používání tříd naprogramovaných v předchozích verzích Javy mezi ně může patřit i potřeba dočasně vypnout přísnou typovou kontrolu. Musíme si však být vědomi nebezpečí, kterému se tím vystavujeme. Podívejte se např. na příkaz

```
iList = (List)sList;
```

ze závěru předchozího programu. V něm je do proměnné vyhrazené pro uložení odkazu na seznam celých čísel ukládán odkaz na seznam textových řetězců.

Jak je ukázáno v zakomentované části programu o pár řádků výše, bez přetypováním vypnuté typové kontroly označí překladač tento příkaz za syntakticky chybný. Jakmile však pravou stranu přetypujeme na surový typ, nechá si překladač tento převod líbit.

Ještě jednou vás proto upozorňuji, abyste varování překladače brali vážně a pokud možno vždy upravili program tak, aby jej překladač přeložil bez vydávání varovných zpráv.

Zakázané operace

Při používání PT a jejich typových parametrů musíme respektovat některá omezení, která vyplývají většinou z toho, že virtuálnímu stroji chybějí při používání PT informace, které byly při očišťování od typových parametrů odstraněny spolu s příslušnými typovými parametry.

Za typové parametry nelze dosazovat primitivní typy

Toto omezení je logickým důsledkem celé koncepce parametrizovaných typů a metod, která vychází z toho, že skutečné typy parametrů mohou být spolehlivě převedeny na některý z omezujících typů, a není-li deklarován žádný omezující typ, tak na typ `Object`. Tuto možnost však primitivní typy neposkytují, a proto se v případě potřeby práce s hodnotami některého z primitivních typů musíme u typových parametrů spokojit s použitím příslušného obalového typu.

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 public class Primitivní<T1 extends Number & Comparable<T1>,
 4                         T2 extends Number & Comparable<T2>>
 5 {
 6   public Primitivní() {
 7   }
 8
 9   public static <T1 extends Number & Comparable<T1>,
10                  T2 extends Number & Comparable<T2>>
11     double průměr( T1 a, T2 b )
12   {
13     System.out.println("Skutečné typy parametrů: \na: " +
14       a.getClass() + "\nb: " + b.getClass() );
15     return (a.doubleValue() + b.doubleValue()) / 2;
16   }
17
18 //===== TESTY =======================================================
19
20   public static void test() {
21     System.out.println("průměr(1,   2  ): " + průměr( 1,   2   ));
22     System.out.println("průměr(1.5, 2.5): " + průměr( 1.5, 2.5 ));
23     System.out.println("průměr(1,   3.0): " + průměr( 1,   3.0 ));
24   }
25 }
```

Jak jsme si již ale řekli v kapitole *Automatické převody mezi primitivními a obalovými typy*, při používání automatických převodů musíme mít na paměti zvýšenou režii virtuálního stroje spojenou s nutností vytvářet a zase rušit příslušné instance.

Typové parametry třídy není možno použít u statických členů

Typové parametry deklarované v hlavičce třídy není možno použít v souvislosti se statickými členy. Deklarace typu statického atributu, typu návratové hodnoty či parametru statické funkce vyvolá syntaktickou chybu. Zkuste definovat následující třídu:

```
 1 package rup.česky.java15.ptm.chybné;
 2
 3 public class Statické<T>
 4 {
 5   static T t;
 6
 7   static void metoda2( T parametr ) {
 8     t = parametr;
 9   }
10
11   static T metoda1() {
12     return t;
13   }
14 }
```

Přesvědčte se, že každé použití typového parametru `T` u statického členu vyvolá syntaktickou chybu:

```
non-static class T cannot be referenced from a static context
```

Toto omezení vyplývá z toho, že typ statických členů nesmí záviset na typu některé z instancí třídy, v níž je daný člen definován.

Nelze vytvořit instanci typového parametru

Není možno vytvářet instance typových parametrů, není tedy možné, aby se v naší třídě vyskytovaly následující deklarace a příkazy:

```
public class Zakázaná<T> {
  T t = new T();
}
```

V přeloženém programu totiž nejsou dostatečné informace o typu, jehož instance se má vytvořit. Protože není předem známý typ instance, nemůžeme dokonce ani předem zaručit, že třída bude nabízet bezparametrický konstruktor použitý v předchozí ukázce, případně jakýkoliv jiný, který bude v daném okamžiku zrovna potřeba.

Instance můžeme získat dvěma způsoby:

- Metody, které s nimi mají pracovat, získají potřebné instance prostřednictvím svých parametrů. Jako ukázka tohoto přístupu může sloužit konstruktor třídy `TestSeznamů`, o níž jsme hovořili v pasáži *Vzájemné závislosti typových parametrů* na straně 60, a její statická metoda `test()`, která pro konstruktor příslušnou instanci vytvoří a při jeho volání mu ji předá.
- Potřebujeme-li získat danou instanci opravdu až v průběhu výpočtu, musíme použít reflexi.

Nelze vytvořit pole instancí typového parametru ani parametrizovaného typu

Není možné vytvářet pole s prvky typového parametru ani parametrizovaného typu. Pokus o jeho vytvoření vyvolá syntaktickou chybu. Není tedy možno zadat:

```
FrontaP<String>[] fronta = new FrontaP<String>[4];
```

Problém je v tom, že pole si pamatuje typ svých prvků, a to i poté, co pole přetypujeme. Pokusíme-li se do něj vložit hodnotu nekompatibilního typu, vyvolá výjimku `ArrayStoreException`. Toto chování však není možné po očištění od typových parametrů zaručit. Tento problém názorně demonstruje následující program:

```
 1 package rup.česky.java15.ptm.chybné;
 2
 3 public class Pole
 4 {
 5   public static void pročNelzePovolitPolePT() {
 6     FrontaP<String>[] fsp =
 7 //        new FrontaP<String>[10]; //-- toto nelze
 8         new FrontaP[10];   //Náhradní řešení, ale s varovnou zprávou
 9
10     //Kdybychom mohli vytvořit pole parametrického typu,
11     //musely by projít i následující operace
12     Object   o = fsp;
13     Object[] op = (Object[]) o;
14
15     //Jenže to bychom si zadělali na problémy - viz dále
16     FrontaP<Integer> fi = new FrontaP<Integer>();
17     fi.zařaď( 5 );
18     op[0] = fi;                  //Tohle by správně nemělo projít
19     String s = fsp[0].další(); //Tady se na to přijde - ale až za běhu
20   }
21 }
```

Jak ale program ukazuje, to, že pole instancí typového parametru či parametrizovaného typu nejde vytvořit, ještě neznamená, že je nemůžeme deklarovat. Musíme je však inicializovat jinak.

První možností je deklarovat proměnnou, která je polem parametrizovaného typu, a inicializovat ji odkazem na instanci neparametrizovaného typu. Předchozí příklad lze tedy upravit:

```
FrontaP<String>[] fronta = new FrontaP[4];
```

Tento příkaz sice vyvolá při překladu varování, nicméně se v pořádku přeloží a proměnnou `fronta` je pak možno používat v souladu s její deklarací. (Tento obrat je použit např. v metodě `testuj()` instancí třídy `TestSeznamů`, o níž jsem hovořil v pasáži *Vzájemné závislosti typových parametrů* na straně 60.)

Druhou možností inicializace pole instancí PT je získat toto pole jako parametr. Tím však pouze odsuneme problém s vytvořením pole na někoho jiného. Nepomůže nám ani použití metody s proměnným počtem parametrů. V následujícím programu vyvolá první příkaz v metodě `test()` varovné hlášení a druhý je rovnou označen za syntaktickou chybu.

```
 1 //Metody jsou definovány ve třídě rup.česky.java15.chybné.Pole
 2
 3 public static void polePTJakoParametr( FrontaP<Integer> ... fip ) {
 4   System.out.println("Pole délky " + fip.length);
 5 }
 6
 7 //===== TESTY ========================================================
 8
 9 public static void test() {
10   polePTJakoParametr( new FrontaP<Integer>(), new FrontaP<Integer>() );
11   polePTJakoParametr( new FrontaP<Double>(), new FrontaP<Double>() );
12 }
```

Práce s poli instancí typových parametrů či parametrizovaných typů je potenciálně nebezpečná, protože ji překladač nedokáže ohlídat. Protože však autoři nechtěli generovat varovná hlášení při každém nešikovném použití pole, posílá překladač varovnou zprávu jenom při jeho inicializaci.

Při práci s poli je zabezpečení korektní práce na nás. Pokud bychom chtěli využít možnosti, že správnou práci s typy za nás ohlídá překladač, museli bychom místo klasického pole dát přednost použití některé z kolekcí implementujících rozhraní `RandomAcces`, nejlépe pak zřejmě třídu `ArrayList` nebo (chceme-li využít její synchronizovanosti) třídu `Vector`.

Výjimky

Parametrizované typy nemohou být potomky typu `Throwable`. Deklarace typu

```
public class MojeVýjimka<T> extends Exception
```

je považována za syntaktickou chybu. Program proto nemůže deklarovat vyhození výjimky parametrizovaného typu. Při vyhození výjimky totiž není známo, kdo výjimku zachytí, a není mu proto možno jakýmkoliv způsobem předat informace, které jsou

při očišťování od typových parametrů odstraněny. Parametrizace výjimky by proto neměla žádný smysl.

Potomek třídy `Throwable` sice nemůže být definován jako PT, ale může být deklarován jako typový parametr. Dokonce umožňuje vyhodit výjimku příslušného typu obdobně, jako je tomu v následující ukázce:

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 public class Výjimka<T extends Throwable>
 4 {
 5   Class<T> výjimka;
 6
 7   public Výjimka( Class<T> výjimka ) {
 8     this.výjimka = výjimka;
 9     System.err.println("Vytvářím objekt s výjimkou typu " + výjimka);
10   }
11
12   public void zkus( int param )  throws T {
13     T vyhodit;
14     System.err.println(" - Vyvoláno s parametrem " + param);
15     if( (param & 1) == 0 ) {
16       try {
17         vyhodit = výjimka.getConstructor().newInstance();
18           //"Výjimka typu " + výjimka );
19         System.err.println(" - Vyhazuji výjimku " + vyhodit );
20       }catch( Exception e ) {
21         throw new RuntimeException(
22           "Nepovedlo se mi vyhodit výjimku", e );
23       }
24       throw vyhodit;
25     }
26     else
27       System.err.println(" - BEZ VÝJIMKY");
28   }
29
30   public static <U extends Throwable> void prověř( Výjimka<U> v ) {
31     for( int i=0;   i < 2;   i++ ) {
32       try {
33         System.err.println("Volám s parametrem " + i );
34         v.zkus( i );
35         System.err.println("Operace zdárně prošla");
36       }catch( Throwable t) {
37         System.err.println("Operace vyhodila výjimku " + t);
38         System.err.println("  Zpráva: " + t.getMessage());
39       }
40       System.err.println("==============");
41     }
42   }
```

```
43
44 //===== TESTY ========================================================
45
46   public static void test() {
47     System.err.println("==============");
48     prověř( new Výjimka<Throwable>( Throwable.class ) );
49     prověř( new Výjimka<Exception>( Exception.class ) );
50     prověř( new Výjimka<RuntimeException>( RuntimeException.class ) );
51   }
52 }
```

Typový parametr je sice možno využít k vyhození výjimky, avšak není jej možno použít ke specifikaci zachytávané výjimky v klauzuli `catch`. Opět totiž budou při běhu chybět informace, které by odlišily jednotlivé typy zachytávaných výjimek. To znamená, že následující metoda způsobí syntaktickou chybu:

```
1 //Metoda je definována ve třídě rup.česky.java15.chybné.Zakázané
2 public static <T extends Throwable> void zkus() {
3   try {
4     //Zkus provést požadovanou činnost
5   }catch( T výjimka ) {
6     Systém.out.println( "Zachytil jsem výjimku typu " + výjimka );
7   }
8 }
```

Nejednoznačnosti a kolize

Očistěním kódu se ztratí spousta informací, takže se pak může stát, že zdrojový kód, který se zdá být na první pohled jednoznačný, se ve skutečnosti ukáže být plným nejrůznějších skrytých nejednoznačností a kolizí.

Falešně přetížená metoda

Začnu trochu ze široka. Občas potřebujeme vytvořit metodu, která by vracela několik hodnot současně. Standardním postupem bývá vytvoření pomocné třídy, jejíž instance slouží jako přepravky, do nichž volaná metoda vracené hodnoty uloží a volající metoda si je odtud zase vyzvedne.

Poté, co jsme se seznámili s PT, by nás mohlo napadnout, že bychom nemuseli vytvářet pro každý speciální případ jednoúčelovou přepravku. Místo toho bychom mohli vytvořit univerzální přepravku a tu přizpůsobit zadáním vhodných typových parametrů okamžité potřebě.

Předpokládejme, že z nadšení nad touto úsporou dále podlehneme záchvatu šetřivosti a definujeme třídu následovně:

```
 1 package rup.česky.java15.ptm.přepravka;
 2 
 3 public class Přepravka12<T1, T2>
 4 {
 5   T1 první;
 6   T2 druhý;
 7 
 8   public T1 get1() { return první; }
 9   public T1 get2() { return první; }
10 
11   public void set( T1 o )  { první = o; }
12   public void set( T2 o )  { druhý = o; }
13 }
```

Pak ke svému překvapení zjistíme, že překladač takovouto třídu odmítne přeložit. (Než budete číst dál, zkuste sami přijít na to, proč.) Zaskočíme jej totiž nastavovacími metodami, při jejichž překladu ohlásí kolizi jmen:

```
name clash: set(T1) and set(T2) have the same erasure
```

Ve své definici jsme totiž zapomněli na to, že typy `T1` a `T2` jsou odlišné jenom zdánlivě a že v přeloženém programu bude oba dva zastupovat typ `Object`. Budeme-li chtít proto vzniklý problém obejít, musíme vytvořit dvě nastavovací metody s odlišnými identifikátory obdobně, jako jsme udělali u metod čtecích.

Nová metoda koliduje se zděděnou

Dobrá, rozhodneme se, že vytvoříme specializovanější přepravku, jejíž oba dva atributy budou stejného typu. Při té příležitosti by se nám mohlo zdát šikovné definovat metodu `equals(T)`, kde `T` je typ obou atributů, a to tak, že budeme považovat přepravku za rovnu dané hodnotě, pokud jí budou rovny hodnoty obou atributů. Definovali bychom ji tedy následovně:

```
 1 package rup.česky.java15.ptm.přepravka;
 2 
 3 public class Přepravka11<T>
 4 {
 5   public T první;
 6   public T druhý;
 7 
 8   public boolean equals( T t ) {
 9     return (první.equals(t) && druhý.equals(t));
10   }
11 }
```

Bohužel, ani tentokrát nepochodíme. Překladač nás opět osočí z kolize názvů – tentokrát se mu nebude líbit, že metoda `equals(T)` koliduje po očistění se stejnojmennou metodou zděděnou od třídy `Object`, a přitom ji nepřekrývá.

Zděděná verze metody totiž akceptuje jako parametr instance všech objektových typů, kdežto právě definovaná akceptuje jako parametr pouze instance typu `T`. S tím se setkáváme běžně, to není na škodu. Problém je ale v tom, že při očišťování třídy bude typ `T` nahrazen typem `Object` a obě metody pak budou mít naprosto stejnou charakteristiku.

Tady posluchači občas namítají, že jsme podobný případ již řešili, a že tehdy překladač vše vyřešil definicí přemosťovací metody. Když se ale nad problémem zamyslíte, zjistíte, že jsou to dvě rozdílné situace.

Tentokrát jsme se pokoušeli definovat novou metodu, která nebude překrývat její zděděnou jmenovkyni, kdežto tehdy jsme se naopak snažili zděděnou metodu překrýt.

Jinými slovy: nyní se snažíme definovat dvě různé metody, avšak ukázalo se, že obě budou mít po očištění stejné charakteristiky. Tehdy jsme však definovali metodu, u níž jsme naopak potřebovali, aby měla stejnou charakteristiku jako její překrývaná jmenovkyně, protože jinak by ji nemohl překrýt. Proto nám překladač vypomohl přemosťovací metodou, kterou tento nedostatek vyřešil.

Kolize požadovaných rozhraní

Implementuje-li rodičovská třída parametrizované rozhraní, musíme počítat s tím, že všichni její potomci budou toto rozhraní implementovat se shodnými typovými parametry. Bude-li tedy rodičovská metoda deklarována

```
class Rodič implements Comparable<Rodič>
```

bude její potomek implementovat také rozhraní `Comparable<Rodič>`. V řadě případů to nevadí, ale může nastat situace, kdy tato skutečnost vyvolá kolizi. Podívejte se na následující definici:

```
 1 package rup.česky.java15.ptm.chybné;
 2
 3 public class Kolize_NesedíRozhraní
 4 {
 5   class Rodič implements Comparable<Rodič>
 6   {
 7     public int compareTo( Rodič r )  { return 0; }
 8   }
 9
10   class Potomek extends Rodič {}
11     //implements Comparable<Potomek> {}
12
13   IntervalU<Rodič>   iur =
14     new IntervalU<Rodič>( new Rodič(), new Potomek() );
15
16   //Implementuje špatné rozhraní - zdědil implementaci Comparable<Rodič>,
17   //avšak konstruktor požaduje implementaci Comparable<Potomek>
18   IntervalU<Potomek> iup =
```

```
19      new IntervalU<Potomek>( new Potomek(), new Potomek() );
20    //Chyba: type parameter
21    // rup.česky.java15.ptm.chybné.Kolize_NesedíRozhraní.Potomek
22    //is not within its bound
23 }
```

Jak vidíte, při definici proměnné `iur` na řádcích 13 a 14 žádné problémy nenastaly, protože instanci třídy `Potomek` předávanou jako parametr bylo možno považovat za instanci třídy reprezentované typovým parametrem `<T extends Comparable<T>>`, tj. po dosazení za instanci třídy `<Rodič extends Comparable<Rodič>>`.

Při definici proměnné `iup` na řádcích 18 a 19 se však překladač vzbouřil, protože po dosazení zadaného typu za typový parametr začal požadovat, aby parametry konstruktoru byly typu `<Potomek extends Comparable Potomek>`.

Naštěstí je tento problém řešitelný. Jak na to si ukážeme v pasáži *Žolík jako předek zadaného typu* na straně 83.

Kolize implementovaných rozhraní

Pokusíte-li se vyřešit předchozí problém tím, že doplníte hlavičku třídy `Potomek` o specifikaci příslušného rozhraní, tj. že definujete

```
class Potomek extends Rodič implements Comparable<Potomek>
```

vyvoláte další chybu – překladač vám totiž oznámí, že

```
java.lang.Comparable cannot be inherited with different arguments:
<rup.česky.java15.ptm.chybné.Kolize_NesedíRozhraní.Potomek> and
<rup.česky.java15.ptm.chybné.Kolize_NesedíRozhraní.Rodič>
```

Specifikace totiž říká: Třída ani typový parametr nesmí vícenásobně implementovat stejné rozhraní, které by mělo v různých deklaracích různé typové parametry.

Špatné pochopení dědičnosti

Při práci s PT musíme dát pozor na to, abychom na ně nepřenesli naše zkušenosti z práce s poli a dynamickými kontejnery. Mezi instancemi tříd s typovými parametry totiž platí zcela jiná pravidla přípustnosti a nepřípustnosti vzájemného zastupování.

Vezměme si např. následující příklad, v němž vystupují dva seznamy, jejichž prvky váže vztah dědičnosti:

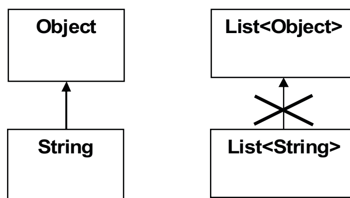
```
/* 1 */  List<String> str = new ArrayList<String>();
/* 2 */  List<Object> obj = str;
/* 3 */  obj.add(new Object());
/* 4 */  String s = str.get(0);
```

První řádek je zcela v pořádku. Na druhém řádku však překladač ohlásí syntaktickou chybu. Kdyby tak neučinil, tak bychom po zdánlivě korektním přiřazení ve třetím řádku ukládali ve čtvrtém řádku do řetězcové proměnné odkaz na obecný objekt.

Obecně platí: *To, že je jeden typ potomkem nebo implementací druhého nijak neimplikuje obdobnou vazbu u parametrizovaných typů, v nichž dané typy vystupují jako typové parametry.*

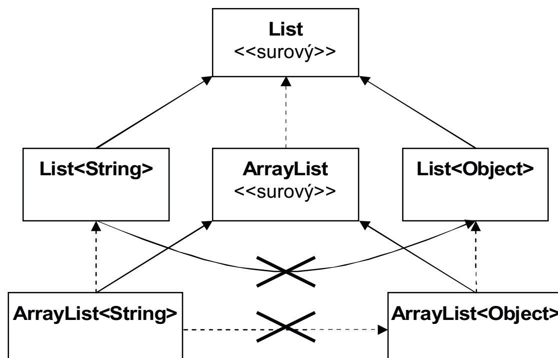
Pokusil jsem se vše znázornit na obrázku, který vám má připomenout, že z toho, že třída String je potomkem třídy Object nijak nevyplývá, že by třída List<String> měla být považovatelná za potomka třídy List<Object> nebo že by instance třídy List<String> měly být ve výrazech dosaditelné do proměnných typu List<Object>.

Dědičnost mezi parametrizovanými typy respektuje dědičnost očištěných typů. Současně se můžeme chovat k libovolnému parametrizovanému typu jako k potomkovi příslušného očištěného typu, tj. proměnným očištěného typu můžeme přiřazovat hodnoty kteréhokoliv z jeho parametrizovaných typů.



Obrázek 5.1: *Dědičnost typů nemá žádný vliv na vztah parametrizovaných typů, v nichž jsou dané typy použity jako typové parametry*

Předchozí obrázek bychom tedy mohli rozvést do následujícího podoby, která již ukazuje i klasickou dědičnost.



Obrázek 5.2: *Parametrizované typy se chovají jako potomci příslušných očištěných (surových) typů*

Žolíky

Jak jsme si právě řekli, dva PT, které se liší pouze hodnotami svých typových parametrů, jsou považovány za vzájemně nezávislé bez ohledu na to, jsou-li jejich typové parametry nějak příbuzensky spřízněny. Tato skutečnost však při používání PT velmi svazuje ruce. Ve starších verzích Javy bychom definovali metodu pro tisk jednotlivých členů kolekce na samostatné řádky následovně:

```
1 //Metoda je definována ve třídě rup.česky.java15.ptm.demo.TiskKolekcí
2 public static void tiskniKolekci_1( Collection c ) {
3   for( Iterator it = c.iterator();   it.hasNext();   )
4     System.out.println( it.next() );
5 }
```

Takto definovaná metoda nám umožní tisk libovolné kolekce. Definujeme-li však v „nové Javě“ zdánlivě ekvivalentní metodu

```
1 //Metoda je definována ve třídě rup.česky.java15.ptm.demo.TiskKolekcí
2 public static void tiskniKolekci_2( Collection<Object> c ) {
3   for( Iterator it = c.iterator();   it.hasNext();   )
4     System.out.println( it.next() );
5 }
```

můžeme s ní tisknout pouze kolekce obsahující prvky typu `Object`. Jak jsme si vysvětlili před chvílí, kolekci obsahující instance typu `Object` nemůžeme považovat za předka žádné jiné kolekce, takže je tato metoda pro jakoukoliv jinou kolekci nepoužitelná.

Aby nám PT nesvazovaly ruce svojí neschopností zavést vztahy ekvivalentní dědičnosti, zavádí Java 5.0 **zástupný typ** (wildcard type) ? (otazník), který označuje libovolný typ vyhovující případným omezujícím podmínkám. Tento typ pak funguje mezi typovými parametry obdobně jako **žolík** (wildcard), který hodnotu typového parametru v daném místě nijak neomezuje.

K parametrizovanému typu, jemuž jsme jako typový parametr dosadili žolík, se pak můžeme chovat jako k rodiči všech typů s konkrétními hodnotami typových parametrů, jenž za zástupný typ dosadil nějaký konkrétní typ. Naopak k PT s konkrétním typovým parametrem se můžeme chovat jako k potomkovi odpovídajícího PT používajícím místo daného typu žolík.

Kdybychom použili v našem předchozím příkladu žolík, získala by definice tvar:

```
1 //Metoda je definována ve třídě rup.česky.java15.ptm.demo.TiskKolekcí
2 public static void tiskniKolekci( Collection<?> c ) {
3   for( Iterator it = c.iterator();   it.hasNext();   )
4     System.out.println( it.next() );
5 }
```

Takto definovanou metodu již můžeme použít k tisku libovolné kolekce bez ohledu na typ jejích prvků.

Omezení hodnot žolíků

I na žolíky můžeme klást dodatečná omezení, v nichž můžeme požadovat, aby byl typ zastupovaný žolíkem potomkem nebo předkem definovaného typu. Ukážeme si nyní obě možnosti a vysvětlíme si, v jakých situacích je použijeme.

Žolík jako potomek zadaného typu

Představte si např. sadu tříd představujících grafické objekty, které jsou schopny se nakreslit na virtuální plátno. Všechny třídy těchto objektů budou implementovat rozhraní `IKreslený` vyžadující implementaci metody `nakresli(Kreslítko)`.

Kdybychom chtěli definovat metodu, která je schopna nechat nakreslit všechny prvky v zadané kolekci, nemůžeme použít metodu deklarovanou následovně:

```
public void nakresli( Collection<IKreslený> c )
```

Jejím parametrem totiž může být pouze kolekce, která za své prvky deklarovala instance rozhraní `IKreslený`. Nemůže jím ale být kolekce instancí kterékoliv z tříd, jež toto rozhraní implementují, ani kolekce případných potomků tohoto rozhraní.

Pokud bychom chtěli deklarovat metodu, která bude jako svůj parametr akceptovat kolekci čehokoliv, co je schopno se nakreslit (přesněji čehokoliv, co implementuje rozhraní `IKreslený`), musíme ji deklarovat:

```
public void nakresli( Collection<? extends IKreslený> c )
```

Zástupné typy můžeme použít nejenom při deklaraci parametrů metod, ale i při deklaraci typu proměnných a atributů. Metoda testující právě deklarovanou metodu by pak mohla vypadat následovně:

```
1 public static void test()
2 {
3   List<? extends IKreslený> lik;
4   lik = new ArrayList<Tvar>();   naplň( lik );   nakresli( lik );
5   lik = new LinkedList<Kruh>();  naplň( lik );   nakresli( lik );
6 }
```

Konkrétnější ukázkou použití žolíků představujících instance nějakého typu či jeho potomků je následující program, v němž je definována třída `Generátory`, jež nabízí dvě metody vracející seznam instancí zadaného typu.

```
1 package rup.česky.java15.ptm.generátor;
2
3 public final class Generátory
4 {
5   private Generátory() {}
6
7   public static <T> List<T> připravSeznam(
8     int počet, IGenerátor<? extends T> gen )
```

```
 9   {
10     List<T> seznam = new ArrayList<T>( počet );
11     return doplňSeznam( počet,  gen,  seznam );
12   }
13
14   public static <T> List<T> doplňSeznam(
15     int počet, IGenerátor<? extends T> gen, List<T> seznam )
16   {
17     for( int i=0;   i < počet;   i++ )
18       seznam.add( gen.další() );
19     return seznam;
20   }
21
22 //===== TESTY =======================================================
23
24   public static void test() {
25     List<Object> lo = připravSeznam( 10, new GenMix() );
26     System.out.println("Mix: " + lo );
27
28     List<String> ls = připravSeznam( 5, new GenString(100) );
29     System.out.println("String: " + ls);
30
31     List<Number> ln;
32     //Následující příkaz považuje překladač za syntakticky chybný,
33     //protože z typu parametrů usoudil, že návratovou hodnotou bude
34     //instance typu List<Integer>
35 //    ln = připravSeznam( 5, new GenInteger(100) );
36
37     //Musíme hodnotu typového parametru zadat explicitně
38
39     //Hodnoty typových parametrů nelze zadat u nekvalifikovaých volání
40 //    ln = <Number>připravSeznam( 5, new GenInteger(100) );
41
42     //Je třeba ji zadat mezi kvalifikaci a název volané metody
43     ln = Generátory.<Number>připravSeznam( 5, new GenInteger(100) );
44
45     //Druhou možností je vhodné přetypování parametru
46     ln=připravSeznam(5,(IGenerátor<? extends Number>)new GenInteger(100));
47
48     //Zadáním seznamu ln mezi parametry dáváme překladači
49     //dostatek informací pro správné určení typu návratové hodnoty
50     ln = doplňSeznam( 5, new GenDouble(100), ln );
51     System.out.println("Number: " + ln);
52   }
53 }
```

Všimněte si hlavně problémů, které vyvolal špatný odhad překladače o typu návratové hodnoty příkazu na řádku 35:

```
připravSeznam( 5, new GenInteger(100) )
```

Překladač totiž nejprve odhadne typ návratové hodnoty metody, a teprve pak zjišťuje, zda je možno hodnotu tohoto typu přiřadit. Proto si také „nespočítal“, že typ `Number` patří mezi rodiče typu `Integer` a že by tedy mohl ve volání metody `připravSeznam` přiřadit typovému parametru `T` hodnotu `Number` a výsledek pak přiřadit proměnné `ln`.

Problém jsme v následujícím příkazu vyřešili tak, že jsme při volání metody zadali hodnotu jejího typového parametru explicitně. Tady bych jen znovu připomenul, že explicitní zadání typových parametrů je možné pouze v kvalifikovaném volání metody. Při nekvalifikovaném volání (viz řádek 40) typové parametry zadat nelze.

Druhou možností je vhodné přetypování některého parametru, který může nastavení hodnoty typového parametru ovlivnit. V našem příkladu je tímto parametrem předávaný generátor. Jak se mnou ale budete jistě souhlasit, explicitní zadání hodnot typových parametrů je průzračnější a lépe specifikuje, co vlastně chcete zadat.

Žolík jako předek zadaného typu

Žolík může zastupovat nejenom potomky zadaného typu, ale (na rozdíl od typových parametrů) také jeho předky. V ilustračním programu v pasáži *Kolize požadovaných rozhraní* na straně 77 jsem ukazoval, jak překladač odmítl vytvořit instanci třídy `IntervalU` s instancemi potomka. Nyní si ukážeme, jak tuto jeho neochotu zlomit.

Musíme nejprve upravit definici intervalu. Nebudeme požadovat, aby typový parametr implementoval rozhraní `Comparable<x>`, v němž bude sám vystupovat jako typový parametr, ale spokojí se s implementací tohoto rozhraní, jehož typovým parametrem bude kterýkoliv z jeho předků. Stačí, využijeme-li v definici intervalové třídy žolíka a definujeme např.:

```
public class IntervalUž<T extends Comparable<? super T>>
```

Programy:

Kompletní definici třídy `IntervalUž` *neukazuji – můžete si ji prohlédnout v balíčku* `rup.česky.java15.ptm.interval`.

Konstruktor pak bude akceptovat i potomka, který zdědí implementaci parametrizovaného rozhraní od rodiče. Překlad následující třídy proto proběhne bez problémů:

```
 1 package rup.česky.java15.ptm.demo;
 2
 3 public class SrovnáníRozhraníŽolíkem
 4 {
 5   class Rodič implements Comparable<Rodič>
 6   {
 7     public int compareTo( Rodič r )  { return 0; }
 8   }
 9
10   class Potomek extends Rodič {}
```

```
11     //implements Comparable<Potomek> {}
12   IntervalUž<Rodič>   iur = new IntervalUž<Rodič>( new Rodič(), new Potomek() );
13   IntervalUž<Potomek> iup = new IntervalUž<Potomek>( new Potomek(), new Potomek() );
14 }
```

Převod klasických tříd na parametrizované[1]

Při parametrizaci stávajících tříd musíte neustále dbát na to, abyste nenarušili stávající rozhraní a aby proto všem dříve napsaným programům stačil pouze nový překlad.

Podívejme se třeba na příklad kolekcí. V předchozí verzi jazyka bylo rozhraní `Collection` definováno následovně:

```
 1 public interface Collection {
 2   int size();
 3   boolean isEmpty();
 4   boolean contains(Object o);
 5   Iterator iterator();
 6   Object[] toArray();
 7   Object[] toArray(Object a[]);
 8   boolean add(Object o);
 9   boolean remove(Object o);
10   boolean containsAll(Collection c);
11   boolean addAll(Collection c);
12   boolean removeAll(Collection c);
13   boolean retainAll(Collection c);
14   void clear();
15 }
```

Nová verze knihoven již definovala všechny kontejnerové třídy a rozhraní s využitím parametrizace. Nyní tedy máme rozhraní `Collection<E>`. Pokud bychom při úpravách příliš nepřemýšleli, mohli bychom je definovat např. následovně:

```
1 //Naivní verze transformace
2 public interface Collection<E> extends Iterable<E> {
3   // ...
4   boolean containsAll(Collection<E> c);
5   boolean addAll(Collection<E> c);
6   boolean removeAll(Collection<E> c);
7   boolean retainAll(Collection<E> c);
8 }
```

Takto definované kolekce by jistě byly typově zabezpečené, jenže neodpovídají původnímu kontraktu. Podle něj totiž např. metoda `containsAll` musí být schopna zpra-

[1] Tato pasáž je silně inspirována obdobnou pasáží z publikace *Gilad Bracha: Generics in the Java programming language*, kterou si můžete zdarma stáhnout ze stránek firmy *Sun* na adrese `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`. Lepší příklad, než je zde uveden, se mi vymyslet nepodařilo.

covat libovolnou kolekci libovolných objektů, avšak nyní je ochotna pracovat s kolekcí instancí typu daného jejím typovým parametrem. To ale nemusí platit a dost často toho ani nejde dosáhnout (např. proto, že volající metoda přesnou hodnotu typového parametru vůbec nezná).

Kromě toho by zrovna v příkladu metody `containsAll` jistě mělo smysl umožnit metodě porovnávat svoji kolekci s libovolnou jinou, protože v případě rozdílnosti kolekcí prostě vrátí `false` a může jí být úplně jedno, co byla druhá kolekce zač.

V případě zbylých tří uvedených metod zase není důvod nepřipustit zařadit do kolekce instance, které budou podtypem typového parametru. I ty totiž vyhovují požadavkům.

Skutečná definice nové verze rozhraní proto využívá žolíků a množinu přípustných typů poněkud rozšiřuje. Je definována následovně:

```
 1 public interface Collection<E> extends Iterable<E> {
 2   int size();
 3   boolean isEmpty();
 4   boolean contains(Object o);
 5   Iterator<E> iterator();
 6   Object[] toArray();
 7   <T> T[] toArray(T[] a);
 8   boolean add(E o);
 9   boolean remove(Object o);
10   boolean containsAll(Collection<?> c);
11   boolean addAll(Collection<? extends E> c);
12   boolean removeAll(Collection<?> c);
13   boolean retainAll(Collection<?> c);
14   void clear();
15 }
```

Podobně nesmíme jednat zbrkle ani při definicích jednotlivých metod. Opět si vezmeme příklad ze standardní knihovny. Ve třídě `Collections` je definována metoda `max`, která očekává jako parametr kolekci porovnatelných prvků a vrací odkaz na největší z nich. V předchozích verzích byla hlavička této metody:

```
public static Object max( Collection col )
```

Předpokládám, že byste si vzpomněli, že jsme si před chvílí říkali o problémech se špatně definovanou implementací rozhraní `Comparable`, a napadlo by vás deklarovat metodu s následující hlavičkou:

```
public static <T extends Comparable<? super T>> T max( Collection<T> col )
```

Problémem této definice je ale to, že po očištění se změní hlavička metody do tvaru:

```
public static Comparable max( Collection col )
```

Ten ale neodpovídá doposud zaužívanému kontraktu a je třeba jej proto změnit. Aby hlavička metody odpovídala po očištění tvaru, ve kterém byla definována v předchozích verzích, využijeme toho, že typový parametr je nahrazen prvním z uvedených rodičů či implementovaných rozhraní a upravíme deklaraci do tvaru:

```
public static <T extends Object & Comparable<? super T>> T max( Collection<T> col )
```

To už je lepší. Stále to ale ještě není ono, protože nás tato deklarace nutí používat kolekci obsahující stejný typ prvků, jako je typ návratové hodnoty. My ale víme, že by pro nás byla tato metoda užitečná i v případě, kdy by typ prvků v kolekci byl některým z potomků typu návratové hodnoty. Upravíme proto hlavičku do tvaru:

```
public static <T extends Object & Comparable<? super T>>
T max( Collection<? extends T> col )
```

V tomto tvaru ji také najdete ve standardní knihovně.

Takovéto složité úvahy asi většinu řadových programátorů nečekají, nicméně tvůrci knihoven musí umět domýšlet všechny důsledky svého návrhu, aby nebylo nutno knihovny následně upravovat.

Parametrizované typy a reflexe

Třída `Class` doznala v nové verzi výrazných rozšíření. Přibyla v ní řada metod – oproti 36 metodám ve verzi 1.4 je jich nyní 57. Většina přidaných metod přitom souvisí s generickými datovými typy.

V nové verzi Javy je nyní parametrizovaná i samotná třída `Class`. Např. objekt `String.class` je nyní instancí parametrizovaného typu `Class<String>`. Hlavní výhodou typového parametru je to, že typy parametrů a návratových hodnot metod instancí třídy `Class<T>` mohou nyní být mnohem přesněji specifikovány. Třída nyní definuje metody

```
T newInstance()
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class...)
```

které již vracejí přímo hodnoty požadovaného typu, a nemusíme je už proto přetypovávat. Kromě toho přibyla nová metoda

```
T cast( Object obj )
```

která vrací odkaz na svůj argument přetypovaný na typ příslušného class-objektu.

Novinky jdou ale ještě dál. Nová verze přinesla celou plejádu nových typů, které slouží k detekci toho, že daný typ byl deklarován jako parametrizovaný. Při čištění se sice tyto informace z programu ztratí, ale v class-objektech tříd a rozhraní zůstanou zachovány a lze je odtud vyloudit.

Tyto informace sice nepomohou zjistit, s jakými hodnotami typových parametrů byla vytvořena ta která instance, ale mohou prozradit, že typ (třída či rozhraní) byl definován jako parametrizovaný, a mohou pomoci určit i podobu této definice.

Kapitola 6

Výčtové typy

Java 5.0 zavádí výčtové typy jako nový druh třídy, která je oproti uživatelově definici doplněna překladačem o další atributy a metody. V této kapitole si po úvodu, v němž si vysvětlíme nevhodnost dosavadních způsobů obcházení absence výčtových typů v Javě, nejprve předvedeme, jak vypadá nejjednodušší definice výčtového typu a ukážeme si, co vše překladač do výsledného class-souboru přidá. Poté se podrobně seznámíme s třídou `Enum`, která je společným rodičem všech výčtových typů, a ukážeme si, jak lze definovat složitější výčtové typy a co nám takovéto typy mohou nabídnout.

Závěrečné podkapitoly jsou věnovány speciálním druhům výčtových typů, jejichž instance jsou ve skutečnosti instancemi anonymních tříd. Vysvětlíme si, jak se takové výčtové typy definují a ukážeme si některá rizika, s nimiž se můžete při jejich definici setkat.

Programy:
Všechny třídy, o nichž se v této kapitole hovoří, jsou definovány v balíčku `rup.česky.java15.výčty`.

Předehra pro méně zkušené

Výčtové typy zavedl na počátku sedmdesátých let programovací jazyk Pascal, aby umožnil typovou kontrolu proměnných a konstant, které mohly nabývat pouze omezeného počtu hodnot (dny v týdnu, měsíce v roce, světové strany, pohlaví atd.). Svůj název získaly z toho, že všechny jejich možné hodnoty lze jednoduše vyjádřit výčtem.

Poznámka:
Někteří možná namítnou, že výčtem možných hodnot lze vyjádřit i celá či dokonce reálná čísla. Přiznejme si však, že takovýto výčet bychom asi těžko považovali za jednoduchý. Charakteristickou vlastností výčtových typů je, že počet jejich možných hodnot je opravdu malý, typicky menší než 10.

Java na počátku výčtové typy nezavedla, protože se jejím autorům nelíbilo, že hodnoty výčtových typů tak, jak byly tehdy implementovány, bylo možno relativně jednoduše zaměnit za celá čísla a ztrácela se tak možnost typové kontroly. Výčtové typy zařadili do jazyka až poté, když konečně přišli na to, jak je udělat bezpečné a opravdu objektově orientované.

Historické pozadí

Absence výčtových typů byla jednou z věcí, kterou kritici minulým verzím Javy vyčítali. Tento nedostatek byl obcházen nejrůznějšími způsoby, přičemž většina z nich používala sadu celočíselných konstant znemožňujících řádnou typovou kontrolu.

Řada programátorů obcházela absenci výčtových typů tak, že místo požadovaného výčtového typu definovala rozhraní a v něm sadu (většinou celočíselných) konstant zastupujících hodnoty definovaného výčtového typu. Chtěl-li někdo tyto konstanty používat, stačilo implementovat příslušné rozhraní. Od té chvíle mohla třída dané konstanty používat jako by byly její vlastní.

Tento přístup měl několik nevýhod:

- Třída se hlásila k implementaci rozhraní, aniž něco doopravdy implementovala.
- Do rozhraní třídy se dostaly informace o typech a hodnotách, které třída a její instance používaly pouze interně.
- Tyto konstanty byly většinou definovány jako celočíselné, čímž byla znemožněna jejich typová kontrola. Pokud byly např. mezi parametry metody konstanty různých výčtových typů, musel si programátor pamatovat, která z nich se zadává jako první a která až jako druhá.

Výše uvedené nevýhody popsal Joshua Bloch v knize *Effective Java*[1] a současně zde popsal, jak by podle něj bylo třeba výčtové typy v tehdejších verzích Javy definovat. Zcela zavrhl celočíselnou reprezentaci a výčtové typy definoval jako standardní objektové typy.

Blochovy výčtové typy byly typově zabezpečené (tj. umožňovaly typovou kontrolu) a jejich instance mohly navíc poskytovat řadu služeb. Koncepce výčtových typů popsaná ve výše zmíněné knize se stala základem definice výčtových typů ve verzi 5.0.

Nejjednodušší definice

V definici výčtových typů je klíčové slovo `class` nahrazeno klíčovým slovem `enum`. Překladač vytvoří třídu, která je potomkem třídy `java.lang.Enum`, a definuje v ní skrytý kód, který později v některých konstrukcích využívá.

---

[1] Bloch, Joshua: Effective Java, Addison-Wesley Professional, 2001, ISBN 0-201-31005-8. Český překlad: Java efektivně, Grada, 2002, ISBN 80-247-0416-1.

Seznam jednotlivých konstant daného typu (dále je budu označovat jako **výčtové konstanty**) je možno definovat buď jenom jejich prostým výčtem – např.:

```
1 package rup.česky.java15.výčty;
2
3 public enum Období { JARO, LÉTO, PODZIM, ZIMA }
```

nebo výčtem se zadáním parametrů konstruktorů příslušných instancí – to v případě, kdy je konstrukce jednotlivých hodnot složitější a vyžaduje explicitní předání parametru použitému konstruktoru (takovouto definici si ukážeme za chvíli).

Pro začátek zůstaňme u této nejjednodušší definice. Použijete-li pouhý seznam konstant, nemusíte jej dokonce ani ukončovat středníkem. Doporučuji však tuto možnost ignorovat, protože jakmile do těla třídy cokoliv přidáte, budete muset přidat i středník ukončující seznam konstant.

Definice vypadá velice jednoduše, ale podíváte-li se na výsledný class-soubor zpětným překladačem (decompilerem), zjistíte, že přeložený program už tak jednoduchý není:

```
 1 package rup.česky.java15.výčty;
 2
 3 public final class Období extends Enum {
 4
 5   public static final Období JARO;
 6   public static final Období LETO;
 7   public static final Období PODZIM;
 8   public static final Období ZIMA;
 9
10   private static final Období[] $VALUES;
11
12   static {
13     JARO   = new Období("JARO",   0);
14     LETO   = new Období("LETO",   1);
15     PODZIM = new Období("PODZIM", 2);
16     ZIMA   = new Období("ZIMA",   3);
17     $VALUES = new Období[] {JARO, LÉTO, PODZIM, ZIMA};
18   }
19
20   public static final Období[] values() {
21     return (Období[])($VALUES.clone());
22   }
23
24   public static Období valueOf(String name) {
25     Období[] arr$ = $VALUES;
26     int len$ = arr$.length;
27     for(int i$ = 0; i$ < len$; i$++) {
28       Období období = arr$[i$];
29       if( období.name().equals(name) )
30         return období;
```

```
31        }
32      throw new IllegalArgumentException(name);
33    }
34
35    private Období(String s, int i) {
36      super(s, i);
37    }
38  }
```

Na tomto zdrojovém kódu si můžete všimnout několika věcí:

- Překladač definoval třídu jako potomka třídy `Enum` (přesněji `java.lang.Enum`).
- Z definice jazyka víme, že nedefinuje-li programátor žádný konstruktor, překladač za něj definuje implicitní. U klasických tříd je tento implicitní konstruktor veřejný a bezparametrický. U výčtových typů je však soukromý a dvouparametrický: jeho prvním parametrem je název definované konstanty a druhým její pořadí. Tělo tohoto konstruktoru obsahuje pouze vyvolání rodičovského konstruktoru se stejnými parametry.
- Překladač definoval statický atribut `$VALUES`, jenž je vektorem (jednorozměrným polem) obsahujícím odkazy na definované výčtové konstanty.
- Překladač definoval metodu `values()`, která vrací kopii vektoru `$VALUES`.
- Překladač definoval metodu `valueOf(String)`, která vrací odkaz na instanci, jejíž název převzala jako parametr.

Poznámka:

Uvedený zdrojový kód platil v době, kdy jsem začal tuto knihu psát. Při závěrečné kontrole těsně před vydáním jsem zjistil, že JDK má ve verzi 1.5.0_03 vylepšené tělo metody `valueOf(String)`, *která je nyní definována následovně:*

```
public static Období valueOf(String name) {
    return (Období)Enum.valueOf(Období.class, name);
}
```

Rodičovská metoda `valueOf(Class,String)` *si při prvním volání vytvoří mapu dvojic [jméno, instance] a při dalších voláních již pouze sahá do této mapy a relativně rychle vrátí požadovaný odkaz.*

Překladačem přidané atributy a metody

Atribut $VALUES

Jak jsem již řekl, překladač definuje vlastní atribut, který v prvním přiblížení nazve `$VALUES`. Tento atribut však definuje pouze pro sebe. Je totiž tak soukromý, že jej (na rozdíl od ostatních překladačem dodaných entit) není možno použít ani ve třídě, v níž byl deklarován (debugger vám jej však mezi statickými atributy ukáže).

Potřebujete-li využívat vektor odkazů na jednotlivé instance výčtového typu, nezbude vám, než si definovat vlastní a naplnit jej zavoláním metody `values()`, která vektor naklonuje.

> **Poznámka:**
> *Použití atributu je sice blokované, avšak jeho název blokován není – deklaruje-li vlastní atribut s tímto názvem, vymyslí překladač pro toto „supersoukromé“ pole nějaký jiný.*

public static final Třída[] values()

Metoda vrací vektor (jednorozměrné pole) deklarovaných konstant daného výčtového typu. Využijete ji např. tehdy, budete-li chtít projít všechny deklarované konstanty a s každou něco udělat – např. textový řetězec s názvy všech konstant třídy `VýčtovýTyp` bychom mohli získat následovně:

```
StringBuilder sb = new StringBuilder();
for( VýčtovýTyp vt : VýčtovýTyp.values() ) {
  sb.append( vt ).append("  ");
String seznam = sb.toString();
```

public static Třída valueOf(String name)

Metoda očekává název výčtové konstanty daného typu a vrátí odkaz na tuto konstantu. V běžných programech ji asi moc nepoužijete, ale při zpracovávání textových vstupů se vám může hodit, např.:

```
Období oo = Období.valueOf(P.zadej("Tvá oblíbená roční doba", "JARO").
               toUpperCase());
```

Třída Enum

Když už jsme si prozradili, co vše překladač do definice třídy přidal, měli bychom si také povědět, co třída převezme ze své mateřské třídy.

Než se ale rozhovořím o tom, co námi definovaný typ zdědí, chtěl bych se nejprve zmínit o samotné třídě `Enum`. Tato třída se totiž zařadila mezi *zvláštní* třídy, kam patří např. třídy `Object` nebo `String`. Její zvláštnost spočívá v tom, že programátor nemůže explicitně definovat potomka této třídy. Pokusíte-li se definovat např. třídu

```
public class Výčet extends Enum {}
```

vyvoláte chybu překladu:

```
classes cannot directly extend java.lang.Enum
```

Potomky třídy `Enum` mohou být pouze třídy definované explicitně jako výčtový typ, tj. třídy, v jejichž definici je místo `class` použito `enum`.

Rodičovský podobjekt třídy `Enum` uchovává u každé instance její název a pořadí, v němž byla definována. Obě tyto hodnoty lze kdykoliv získat zavoláním metod

`name()`, resp. `ordinal()`. Vlastní atributy jsou však deklarovány jako soukromé, takže jsou pro potomky nepřístupné.

Podíváte-li se do zdrojového kódu na definici třídy `Enum`, zjistíte, že její hlavička má tvar:

```
public abstract class Enum<E extends Enum<E>>
implements Comparable<E>, Serializable
```

Jak vidíte, třída `Enum` deklaruje třídu svého potomka již ve své hlavičce, aby pak mohla ve svém těle definovat metody, které s tímto potomkem pracují.

S výjimkou metody `toString()` jsou všechny metody deklarovány jako konečné, takže je není možno v potomcích třídy `Enum` překrýt. Podívejme se nyní na jednotlivé metody podrobněji.

Nově definované metody

public final String name()

Vrátí název příslušné výčtové konstanty.

public final int ordinal()

Vrátí pořadí definice konstanty. Konstanta uvedená v definici jako první má pořadí `0`. Je-li třída `Tř` výčtovým typem, pak pro každou z jejích výčtových konstant `KON` platí:

```
KON == Tř.values()[ KON.ordinal() ]
```

public final int compareTo(E o)

Metoda `compareTo(E)` deklarovaná v rozhraní `Comparable<E>` předpokládá, že jejím parametrem bude instance potomka třídy `Enum`. Vzhledem k typu svého parametru je doprovázená skrytou přemosťovací metodou `compareTo(Object)` (ve zdrojovém kódu ji nenajdete – o její vytvoření se postará až překladač viz pasáž *Přemosťovací metody* na straně 65), která pouze přetypuje svůj parametr a zavolá svoji jmenovkyni.

Metoda `compareTo(E)` porovnává instance daného typu podle jejich ordinálních čísel. Instance, která byla v definici třídy uvedena dříve, je pokládána za menší a naopak.

public final Class<E> getDeclaringClass()

Další zajímavou metodou je metoda `getDeclaringClass()`, která vrátí class-objekt třídy, v níž je daná výčtová konstanta deklarována (tento objekt je typu `Class<E>`).

Na první pohled by se mohlo zdát, že tato metoda není potřeba, protože příslušný class-objekt přece vrátí metoda `getClass()`, ale není tomu tak. Jak uvidíme později, každá z instancí výčtového typu může být instancí nějakého podtypu své rodičovské třídy.

public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)

Pro úplnost bychom ještě měli zmínit také statickou metodu `valueOf` deklarovanou:

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
```

Tato metoda vrátí odkaz na zadanou konstantu zadaného výčtového typu, přičemž tuto konstantu zadáváte jejím jménem a výčtový typ jeho class-objektem.

Překryté verze metod zděděných z Object

protected final Object clone() throws CloneNotSupportedException

Klonování instancí výčtových typů je zakázáno. Proto tato metoda při svém zavolání pouze vyhodí výjimku `CloneNotSupportedException`. Tato definice metody `clone()` spolu s ošetřením deserializace zabezpečuje, že se nikdy neobjeví další výčtové konstanty daného typu.

equals(Object)

Metoda považuje hodnoty instancí za sobě rovné právě tehdy, jedná-li se o jednu a tutéž instanci. V programu je proto naprosto jedno, použijete-li při porovnávání dvou hodnot daného výčtového typu volání metody `equals(Object)` nebo operátor ==.

Toto chování kopíruje chování překryté jmenovkyně z třídy `Object`. Metoda je zde totiž definována proto, aby mohla být označena jako konečná a zabezpečila tak, že žádný výčtový typ nebude mít metodu `equals(Object)` definovánu jinak.

hashCode()

Metoda vrací týž heškód, který by vrátila metoda zděděná z třídy `Object`. I tato metoda je zde definována pouze proto, aby mohla být označena jako konečná a zabezpečila tak, že žádný výčtový typ nebude mít heškód definován jinak.

public String toString()

Jediná z definovaných metod, kterou je možno překrýt. V implicitní verzi vrací název příslušné konstanty.

Serializace

Výčtové typy jsou serializovatelné. Na systémové úrovni je však zabezpečeno, aby při čtení hodnot daného výčtového typu nikdy nemohly vzniknout další instance. Toto ošetření deserializace spolu s definicí metody `clone()` zabezpečuje, že od daného typu budou vždy existovat pouze ty instance, které jsou uvedeny v definici příslušné třídy.

Použití výčtových typů v programu

Konstanty výčtových typů je možné používat nejenom k uchovávání hodnot a k jejich případnému porovnávání v příkazech `if` nebo `while`, ale můžete je použít také v přepínači `switch` a v cyklu `for`.

Přepínač

Nová verze Javy umožňuje používat konstanty výčtových typů i v přepínači. Nesmíme však zapomenout na to, že překladač nekontroluje, zda jsme vyčerpali všechny mož-

nosti, a může se chybně domnívat, že existuje ještě nějaká cesta, kterou jsme v přepínači nepokryli.

Pokud bychom např. v následující definici neuvedli závěrečný příkaz `return`, upozornil by nás překladač na to, že daná metoda musí vracet hodnotu, což podle něj naše metoda nezaručuje.

```
1  //Metoda je definována ve třídě rup.česky.java15.výčty.Výčty
2  public static String činnost_1( Období období ) {
3    switch( období ) {
4      case JARO:   return "kvete";
5      case LÉTO:   return "zraje";
6      case PODZIM: return "plodí";
7      case ZIMA:   return "spí";
8    }
9    //Překladač nekontroluje, zda jsem všechna období vyčerpal
10   return null;
11 }
```

Přebytečnému příkazu `return` se můžete vyhnout tak, že nahradíte návěští poslední větve přepínače návěštím `default`. Takto definovaný program je pak sice kratší, ale na druhou stranu pro leckoho méně srozumitelný.

Optimální by bylo upravit definici tak, aby v případě, kdy je použita konstanta, s níž přepínač nepočítá, vyhodila výjimku:

```
1  //Metoda je definována ve třídě rup.česky.java15.výčty.Výčty
2  public static String činnost( Období období ) {
3    switch( období ) {
4      case JARO:   return "kvete";
5      case LÉTO:   return "zraje";
6      case PODZIM: return "plodí";
7      case ZIMA:   return "spí";
8      default:     throw new IllegalArgumentException(
9                     "Neočekávaná hodnota parametru období=" + období );
10   }
11 }
```

Takováto definice je připravena i na případ, kdy bychom se v budoucnu náhodou rozhodli počet konstant daného výčtového typu změnit a zapomněli příslušným způsobem upravit některé části programu.

Přepínače reagující na konstanty výčtových typů implementuje překladač tak, že definuje pomocnou vnořenou třídu, jejímž statickým atributem je vektor mapující ordinální čísla použitých konstant daného výčtového typu na návěští `case`. Část kódu obsahující předchozí metodu analyzoval dekompilátor následovně:

```
1  package rup.česky.java15.výčty;
2
3  public class Výčty
```

```
4  {
5    public static String cinnost(Období období)
6    {
7      switch( _1.$SwitchMap$java15$Období[období.ordinal()] )
8      {
9        case 1: return "kvete";
10
11       case 2: return "zraje";
12
13       case 3: return "plodi";
14
15       case 4: return "spi";
16     }
17     throw new IllegalArgumentException(
18       "Neočekávaná hodnota parametru období: " + období );
19   }
20
21   //Název následující vnořené třídy je pouze pomocný -
22   //třídě je přidělen stejný název, jako by byla anonymní
23   static class _1
24   {
25     static final int $SwitchMap$java15$Období[];
26     static
27     {
28       $SwitchMap$java15$Období = new int[Období.values().length];
29       try
30       {
31         $SwitchMap$java15$Období[Období.JARO.ordinal()] = 1;
32       }
33       catch (NoSuchFieldError ex) { }
34       try
35       {
36         $SwitchMap$java15$Období[Období.LETO.ordinal()] = 2;
37       }
38       catch (NoSuchFieldError ex) { }
39       try
40       {
41         $SwitchMap$java15$Období[Období.PODZIM.ordinal()] = 3;
42       }
43       catch (NoSuchFieldError ex) { }
44       try
45       {
46         $SwitchMap$java15$Období[Období.ZIMA.ordinal()] = 4;
47       }
48       catch (NoSuchFieldError ex) { }
49     }
50   }
51 }
```

Cyklus

Použití v klasických cyklech je zřejmé. Ne každého ovšem napadne, že výčtové typy je možno použít i v nově zavedené verzi cyklu for. Pouze nesmíme zapomenout, že o vektor, přes který iterujeme, musíme nejprve příslušný výčtový typ požádat – např.:

```
1 //Metoda je definována ve třídě rup.česky.java15.výčty.Výčty
2 public static String vyjmenuj() {
3   String s = "";
4   for( Období období : Období.values() )
5     s += období + " ";
6   return s;
7 }
```

Složitější definice výčtových typů

Na počátku článku jsem uváděl nejjednodušší možnou definici výčtového typu. Výčtové typy však mohou obsahovat i vlastní metody a využívat konstruktory s parametry.

Ukažme si tyto možnosti na příkladu třídy Směr8. Všimněte si, jak jsou předávány parametry konstruktoru – za definovanou konstantu se pouze vloží závorky a za ně se vypíší hodnoty předávaných parametrů.

> **Poznámka:**
> *Před chvílí jsme si říkali, že místo bezparametrického konstruktoru je pro třídu definován ve skutečnosti konstruktor dvouparametrický. Jak vám ale prozradí dekompilátor, tyto dva parametry jsou přidány na počátek seznamu parametrů každého konstruktoru, který definujete.*

Při definici konstruktoru je třeba ignorovat skutečnost, že rodičovská třída má pouze dvouparametrický konstruktor, a rodičovský konstruktor nevolat. Volání rodičovského konstruktoru v konstruktorech výčtových typů totiž patří mezi zakázané operace a je výhradním právem a povinností překladače.

Další věcí, která stojí za zmínku, je statický blok inicializující statické atributy. Těmto atributům totiž není možné přiřazovat hodnoty v konstruktoru (např. přidat do seznamu vytvářenou instanci jako další prvek), protože v době, kdy jsou instance konstruovány, dané atributy ještě vůbec neexistují.

Vše vyplývá z toho, že deklarace výčtových **konstant** musí být první deklarací v těle výčtového typu. Proto vám nezbude, než umístit deklarace statických atributů až za ně. Statické atributy budete proto moci používat až poté, co budou definovány výčtové konstanty. Odkázat se v konstruktoru na jakýkoliv statický atribut je zakázáno a jakmile překladač něco takového objeví, ohlásí syntaktickou chybu.

U výčtových typů proto neplatí, že statické atributy a metody je možné používat ještě před tím, než bude vytvořena první instance. Vzhledem k syntaktickým pravidlům se

instance vždy vytvoří před tím, než je možné použít jakýkoliv jiný statický atribut či metodu.

Potřebujete-li proto naplnit nějaké statické kontejnery hodnotami odpovídajícími jednotlivým výčtovým konstantám, musíte zvolit nějaké náhradní řešení. V uváděném příkladu je definována pomocná vnitřní třída `Přepravka`, do jejíchž instancí se v konstruktoru uloží potřebné hodnoty. Po definici výčtových konstant se tyto hodnoty vloží do příslušných kontejnerů a přepravky, které od této chvíle nejsou potřeba, se předají správci paměti (garbage collector) k odstranění.

Ošetříte-li všechny tyto nebezpečné situace, je definice dalších metod již rutinní záležitostí a v následující ukázce jsou proto uvedeny jen částečně (v doprovodném programu jsou uvedeny všechny).

```
 1 package rup.česky.java15.výčty;
 2 
 3 /*******************************************************************
 4  * Třída sloužící jako výčtový typ pro 8 hlavních světových stran
 5  * a zvláštní hodnotu reprezentující nezadaný směr. Třída je
 6  * zjednodušenou verzí stejnojmenné třídy z balíčku rup.česky.společně.
 7  *
 8  * @author   Rudolf Pecinovský
 9  * @version  2.01, duben 2004
10  */
11 public enum Směr8
12 {
13 //== VÝČTOVÉ KONSTANTY ===========================================
14 
15   //Následující definice přidávají další tři parametry konstruktoru:
16   // - změny vodorovné a svislé souřadnice při pohybu v daném směru
17   // - zkratka používaná pro daný směr
18   VÝCHOD      (  1,  0, "S"  ),
19   SEVEROVÝCHOD(  1, -1, "SV" ),
20   SEVER       (  0, -1, "S"  ),
21   SEVEROZÁPAD ( -1, -1, "SZ" ),
22   ZÁPAD       ( -1,  0, "Z"  ),
23   JIHOZÁPAD   ( -1,  1, "JZ" ),
24   JIH         (  0,  1, "J"  ),
25   JIHOVÝCHOD  (  1,  1, "JV" ),
26   ŽÁDNÝ       (  0,  0, "@"  ),
27   ;
28 
29 
30 //== KONSTANTNÍ ATRIBUTY TŘÍDY ========================================
31 
32   /** Celkový počet definovaných směrů. */
33   public  static final int SMĚRŮ = 9;
34 
35   /** Maska pro dělení modulo. */
```

```
36   private static final int MASKA = 7;
37
38   /** Všechny použitelné názvy směrů. */
39   private static final Map<String,Směr8> názvy =
40                    new HashMap<String,Směr8>( SMĚRŮ*3 );
41
42   /** Velikost změny příslušné složky dvojrozměrných souřadnic
43    *  po přesunu na sousední políčko v daném směru. */
44   private static final int[][] posun = new int[SMĚRŮ][2];
45
46   /** Vektor jednotlivých směrů. */
47   private static final Směr8[] SMĚRY = values();
48
49
50   //Inicializace statických atributů je nerealizovatelná před definicí
51   //jednotlivých konstant ==> je ji proto potřeba realizovat dodatečně
52   static
53   {
54     for( Směr8 s : SMĚRY )
55     {
56       posun[s.ordinal()][0] = s.přepravka.dx;
57       posun[s.ordinal()][1] = s.přepravka.dy;
58       názvy.put( s.přepravka.zkratka, s );
59       názvy.put( s.name(),            s );
60       s.přepravka = null;             //Přepravka už nebude potřeba
61     }
62   }
63
64
65 //== PROMĚNNÉ ATRIBUTY INSTANCÍ ======================================
66
67   /*******************************************************************
68    * Pomocná vnořená třída sloužící jako dočasné úložiště hodnot,
69    * které je třeba přiřadit do kontejnerů, jež však v době volání
70    * konstruktoru ještě neexistují.
71    */
72   private static class Přepravka
73   {
74     int    dx, dy;
75     String zkratka;
76   }
77   /** Pomocná proměnná uchovávající požadované hodnoty do zavolání
78    * statického konstruktoru. Pak je vynulována. */
79   Přepravka přepravka;
80
81
82 //##########################################################################
83 //== KONSTRUKTORY A TOVÁRNÍ METODY ===================================
```

```
 84
 85   /****************************************************************
 86    * Vytvoří nový směr a zapamatuje si zkratku jeho názvu
 87    * spolu se změnami souřadnic při pohybu v daném směru.
 88    *
 89    * @param dx  Změna vodorovné souřadnice
 90    *        při přesunu na sousední políčko v daném směru.
 91    * @param dy  Změna svislé souřadnice
 92    *        při přesunu na sousední políčko v daném směru.
 93    * @param zkratka  Jedno- či dvojpísmenná zkratka označující daný směr.
 94    */
 95    private Směr8( int dx, int dy, String zkratka )
 96    {
 97      přepravka = new Přepravka();
 98      přepravka.dx = dx;
 99      přepravka.dy = dy;
100      přepravka.zkratka = zkratka;
101    }
102
103
104
105 //== VLASTNÍ METODY INSTANCÍ ==========================================
106
107   /*******************************************************************
108    * Vrátí směr otočený o 90° vlevo.
109    * Oproti metodě vlevoVbok nepotřebuje přetypovávat výsledek na Směr8.
110    *
111    * @return Směr objektu po vyplnění příkazu vlevo v bok
112    */
113   public Směr8 vlevoVbok()
114   {
115     ověřPlatný();
116     return SMĚRY[MASKA & (2+ordinal())];
117   }
118
119   //Obdobně lze definovat i vpravoVbok, čelemVzad,
120   //nalevoVpříč a napravoVpříč
121
122   /******************************************************************
123    * Vrátí znaménko změny x-ové souřadnice při pohybu v daném směru.
124    *
125    * @return znaménko změny x-ové souřadnice při pohybu v daném směru
126    */
127   public int dx()
128   {
129     ověřPlatný();
130     return posun[ordinal()][0];
131   }
```

```
132
133
134   /***************************************************************
135    * Vrátí x-ovou souřadnici políčka v daném směru
136    * vzdáleného vodorovně o zadanou vzdálenost.
137    *
138    * @param x            x-ová souřadnice stávajícího políčka
139    * @param vzdálenost   Vzdálenost políčka ve vodorovném směru
140    *
141    * @return x-ová souřadnice požadovaného políčka
142    */
143   public int dalšíX( int x, int vzdálenost )
144   {
145     ověřPlatný();
146     return x + posun[ordinal()][0]*vzdálenost;
147   }
148
149   //Obdobné metody lze definovat i pro svislý směr
150
151
152
153 //== SOUKROMÉ A POMOCNÉ METODY INSTANCÍ ================================
154
155   /****************************************************************
156    * Ověří použitelnost daného směru, tj. že instance opravdu definuje
157    * smysluplný směr.
158    */
159   private void ověřPlatný()
160   {
161     if( this == ŽÁDNÝ )
162       throw new IllegalStateException(
163           "Operaci není možno provádět nad směrem ŽÁDNÝ" );
164   }
165 }
```

Konstanty anonymních typů

Při vyjmenovávání metod třídy `Enum` jsem se zmínil o tom, že každá výčtová konstanta může být jiného typu. Nyní bych vám ukázal, jak se takový typ definuje a k čemu může být dobrý.

Takovýto výčtový typ využijete tehdy, budete-li potřebovat, aby jeho instance nepředstavovaly datový, ale funkční objekt, tj. aby se jednotlivé instance lišily chováním svých metod.

Výčtový typ, jehož instance se liší chováním svých metod, definujeme tak, že jeho konstanty definujeme jako instance anonymních tříd.

```
 1 package rup.česky.java15.výčty;
 2
 3 /*****************************************************************************
 4  * Třída sloužící k demonstraci možností "funkcionálních"
 5  * výčtových typů.
 6  *
 7  * @author   Rudolf Pecinovský
 8  * @version  2.01, duben 2004
 9  */
10 public enum Operátor
11 {
12 //== VÝČTOVÉ KONSTANTY ==============================================
13
14   SOUČET( '+' )
15   {
16     double proveď(double x, double y) { return x + y; }
17     void nic() {}
18   },
19
20   ROZDÍL( '-' )
21   {
22     double proveď(double x, double y) { return x - y; }
23   },
24
25   SOUČIN( '×' )
26   {
27     double proveď(double x, double y) { return x * y; }
28   },
29
30   PODÍL( ':' )
31   {
32     double proveď(double x, double y) { return x / y; }
33   },
34
35   MOCNĚNÍ( '^' )
36   {
37     double proveď(double x, double y) { return Math.pow(x,y); }
38   },
39
40   ODMOCNĚNÍ( 'V' )  //Přesněji '\u221A'
41   {
42     double proveď(double x, double y) { return Math.pow(y,1/x); }
43   };
44
45
46 //== KONSTANTNÍ ATRIBUTY INSTANCÍ ========================================
47
48   /** Znak příslušného operátoru */
```

```
49   private final char znak;
50
51
52 //######################################################################
53 //== KONSTRUKTORY A TOVÁRNÍ METODY ====================================
54
55   /***********************************************************************
56    * Vytvoří binární operátor zobrazovaný znakem zadaným jako parametr.
57    *
58    * @param znak  Znak označující daný operátor.
59    */
60   private Operátor( char znak )
61   {
62     this.znak = znak;
63   }
64
65
66 //== ABSTRAKTNÍ METODY ================================================
67
68   /***********************************************************************
69    * Metoda, kterou musejí implementovat všechny mateřské třídy
70    * definovaných operátorů.
71    *
72    * @param x   Levý operand
73    * @param y   Pravý operand
74    */
75   abstract double proveď(double x, double y);
76
77
78 //== TESTY A METODA MAIN ==============================================
79
80   /***********************************************************************
81    * Metoda testující funkce všech definovaných operátorů
82    * a zobrazující příslušnost jednotlivcýh konstant k mateřským třídám.
83    */
84   public static void test()
85   {
86     double x = 3;   //Levý operand
87     double y = 8;   //Pravý operand
88
89     String formát = "\n%-10s %3.1f %c %3.1f = %8.3f";
90
91     //Postupně provedeme nad zadanými operandy jednotlivé operátory
92     //a po každé operaci zobrazíme výsledek
93     for (Operátor o : values() )
94     {
95       System.out.printf( formát, o+":", x, o.znak, y, o.proveď(x, y) );
96     }
```

```
 97      //Samostatné vyvolání některých operátorů
 98      x = 2;
 99      y = 3;
100      System.out.printf( formát, "Ještě jednou mocnění:   ",
101        x, MOCNĚNÍ.znak, y, MOCNĚNÍ.proveď(x, y));
102      System.out.printf( formát, "Ještě jednou odmocnění: ",
103        x, ODMOCNĚNÍ.znak, y, ODMOCNĚNÍ.proveď(x, y) );
104
105      System.out.println('\n');
106
107      //Zobrazení skutečného a výčtového typu jednotlivých operátorů
108      for (Operátor o : values() )
109      {
110        System.out.printf(
111          "%-11skonstanta třídy %s,instance třídy %s\n",
112          o + ":",
113          o.getDeclaringClass().getName(),
114          o.getClass()        .getName() );
115      }
116    }
117  }
```

Při spuštění metody `test()` se na standardní výstup vypíše následující text:

```
 1 SOUČET:    3,0 + 8,0 =   11,000
 2 ROZDÍL:    3,0 - 8,0 =   -5,000
 3 SOUČIN:    3,0 × 8,0 =   24,000
 4 PODÍL:     3,0 : 8,0 =    0,375
 5 MOCNĚNÍ:   3,0 ^ 8,0 = 6561,000
 6 ODMOCNĚNÍ: 3,0 V 8,0 =    2,000
 7 Ještě jednou mocnění:    2,0 ^ 3,0 =  8,000
 8 Ještě jednou odmocnění:  2,0 V 3,0 =  1,732
 9
10 SOUČET:  konstanta třídy java15.Operátor,instance třídy java15.Operátor$1
11 ROZDÍL:  konstanta třídy java15.Operátor,instance třídy java15.Operátor$2
12 SOUČIN:  konstanta třídy java15.Operátor,instance třídy java15.Operátor$3
13 PODÍL:   konstanta třídy java15.Operátor,instance třídy java15.Operátor$4
14 MOCNĚNÍ: konstanta třídy java15.Operátor,instance třídy java15.Operátor$5
15 ODMOCNĚNÍ:konstanta třídy java15.Operátor,instance třídy java15.Operátor$6
```

Uvedená třída využívá jednoparametrický konstruktor. Kdyby vystačila s bezparametrickým konstruktorem, nebylo by třeba za názvy výčtových konstant uvádět ani prázdné závorky a mohli bychom za ním rovnou začít psát otevírací složenou závorku s tělem příslušné anonymní třídy.

Výčtové konstanty mohou ve svých třídách definovat libovolné metody, ale zvenku budou dostupné pouze ty, které definuje současně jejich mateřská třída (tj. třída, jejímiž jsou konstantami) a které jejich anonymní třídy překryjí.

Metody, které budou definovány ve třídách všech konstant, mohou být v mateřské třídě deklarovány jako abstraktní (ostatně jak jinak, když budou vždy překryty).

Přestože se tak mateřská třída stane abstraktní třídou, nesmíte tuto skutečnost uvést v její hlavičce. To, že je třída abstraktní, pochopí překladač z přítomnosti abstraktních metod. Přítomnost klíčového slova `abstract` v hlavičce výčtového typu však považuje za syntaktickou chybu.

V závěru testovací metody jsem vám také ukázal cyklus dokazující nepoužitelnost metody `getClass()` pro zjištění mateřského výčtového typu konstant a z toho plynoucí nutnost existence speciální metody pro určení typu výčtové konstanty.

Nepoužitelnost lokálních atributů

Vzhledem k tomu, že konstanty výčtových typů jsou definovány jako statické atributy své třídy, nelze atributům jejich tříd jednoduše přiřadit počáteční hodnoty prostřednictvím konstruktoru výčtového typu.

Ukážu vám to na příkladu ze života. Nedávno jsme s dětmi v kroužku programování vytvářeli hru, ve které se na šachovnicově uspořádané hrací desce vyskytuje řada diamantů, které se všelijak přesouvají. Definovali jsme výčtový typ představující různé podoby hypotetického diamantu, kterého hrací deska žádala voláním metody `nakresli(int,int,int,Kreslítko)` o vykreslení diamantu se zadaným indexem na zadaném políčku aktivního plátna. Potřebnou třídu jsme definovali následovně:

```
 1 package rup.česky.java15.výčty;
 2
 3 import rup.česky.tvary.Barva;
 4 import rup.česky.tvary.IKreslený;
 5 import rup.česky.tvary.Kreslítko;
 6
 7 import static rup.česky.tvary.AktivníPlátno.AP;
 8
 9
10 public enum Diamant
11 {
12 //== VÝČTOVÉ KONSTANTY ==========================================
13
14   ČTVEREC( Barva.MODRÁ )
15   {
16     public void nakresli( int x, int y, Kreslítko k )
17     {
18       k.vyplňRám( x, y, MODUL, MODUL, barva );
19     }
20   },
21
22   HVĚZDA( Barva.ČERVENÁ )
23   {
24     public void nakresli( int x, int y, Kreslítko k )
```

```
25      {
26        k.vyplňPolygon(
27          new int[] { x+MODUL/2, x+MODUL, x, x+MODUL, x, x+MODUL/2 },
28          new int[] { y, y+MODUL, y+MODUL/3, y+MODUL/3, y+MODUL, y },
29          barva );
30      }
31    },
32
33    KOLO( Barva.ČERNÁ )
34    {
35      public void nakresli( int x, int y, Kreslítko k )
36      {
37        k.vyplňOvál( x, y, MODUL, MODUL, barva );
38      }
39    },
40
41    //... Další definice možných tvarů diamantů
42    ;
43
44  //== KONSTANTNÍ ATRIBUTY TŘÍDY ==========================================
45
46    private static final Diamant[] DRUH  = Diamant.values();
47    public  static final int       DRUHŮ = DRUH.length;
48
49    private static final int MODUL = AP.getKrok();
50
51
52  //== KONSTANTNÍ ATRIBUTY INSTANCÍ =======================================
53
54    private final Barva barva;
55
56
57  //== OSTATNÍ METODY TŘÍDY ===============================================
58
59    public static void nakresli( int index, int xp, int yp, Kreslítko k )
60    {
61      Diamant d = DRUH[ index ];
62      int x = MODUL * xp;
63      int y = MODUL * yp;
64      d.nakresli( x, y, k );
65    }
66
67
68  //== KONSTRUKTORY A TOVÁRNÍ METODY ======================================
69
70    private Diamant( Barva barva )
71    {
72      this.barva = barva;
```

```
 73   }
 74
 75
 76 //== ABSTRAKTNÍ METODY ==================================================
 77
 78   abstract void nakresli( int x, int y, Kreslítko k );
 79
 80
 81 //== TESTY A METODA MAIN ================================================
 82
 83   public static void main( String... args )
 84   {
 85     class Deska implements IKreslený
 86     {
 87       public void nakresli( Kreslítko k )
 88       {
 89         for( Diamant d : DRUH )
 90         {
 91           int i = d.ordinal();
 92           Diamant.nakresli( i, i, i, k );
 93         }
 94       }
 95     }
 96
 97     Deska d = new Deska();
 98     AP.přidej( d );
 99   }
100 }
```

Překladač ale takto definovanou třídu odmítne přeložit. Tvrdí, že atribut `barva` použitý v metodách anonymních tříd je volán nepřípustně:

```
non-static variable barva cannot be referenced from a static context
```

Problém je v tom, že anonymní třídy výčtových konstant jsou definovány jako statické. Proto musejí k atributům svých výčtových konstant přistupovat zvenku. Nejsou pro ně proto viditelné soukromé atributy instancí.

Problém lze vyřešit tak, že u atributu `barva` zrušíme modifikátor `private`. Pak se tento atribut stane viditelný všemi potomky bez ohledu na to, jsou-li definováni uvnitř třídy nebo jen uvnitř stejného balíčku. Trochu se tím sice naboură zapouzdření, protože atribut bude viditelný i z ostatních tříd v balíčku, ale považuji tuto daň za přijatelnou.

Kdybychom totiž nechtěli porušovat zapouzdření, museli bychom nejspíš definovat tento atribut v každé z anonymních tříd a v rámci jejich konstruktoru jej inicializovat např. ze statické proměnné, kterou by jim naplnil rodičovský konstruktor. Toto řešení je zbytečně těžkopádné a krkolomné.

Poznámka:
V uvedeném příkladu je inkriminovaný atribut konstantou, jejíž hodnota je zadána jako parametr konstruktoru. V takovém případě je samozřejmě nejjednodušší „ochudit" konstruktor o tento parametr a zadat příslušnou hodnotu atributu přímo v těle anonymní třídy. V jiné situaci ale tak jednoduché řešení mít k dispozici nemusíme.

Kontejnery EnumSet a EnumMap

Ve třídě `java.util` se objevily dvě kontejnerové třídy optimalizované pro práci s konstantami výčtových typů: `EnumSet` a `EnumMap`. Obě třídy využívají vlastností výčtových typů k tomu, aby zefektivnily svoji práci.

EnumMap

Instance třídy `EnumMap` deklarované `EnumMap<K extends Enum<K>,V>` jsou mapy, jejichž klíče jsou instancemi některého výčtového typu. Oproti ostatním mapám nepřinášejí kromě zvýšení efektivity nic převratného.

Vyšší efektivity těchto map je dosaženo tím, že uchovávané hodnoty ukládají do vektoru o délce dané počtem konstant daného výčtového typu a hodnoty klíčů neukládají vůbec, protože místo celých instancí pracují pouze s jejich ordinálními čísly, která označují pozici příslušné hodnoty ve vektoru.

Zajímavou vlastností těchto map je, že kontrakt zaručuje, že jejich iterátory nikdy nevyhodí výjimku `ConcurrentModificationException`. Kontrakt navíc zaručuje, že prvky mapy budou zpracovávány v pořadí jejich klíčů (tj. v pořadí ordinálních hodnot klíčů), a že toto pořadí respektují i vracené iterátory. Současně deklaruje, že iterátor může (ale také nemusí) respektovat změny ve struktuře kontejneru, které se udají během iterace.

Současná implementace pracuje tak, že přidáte-li do mapy (případně odeberete-li z mapy) dvojici, jejíž klíč leží až za naposledy zpracovaným klíčem, iterátor bude tuto změnu respektovat. Naopak, přidáte-li či odeberete-li dvojici, jejíž klíč leží před naposledy zpracovaným klíčem, iterátor bude tuto změnu ignorovat. Toto chování však již kontrakt nezaručuje, takže by bylo vhodné na těchto vlastnostech aktuální implementace nestavět.

EnumSet

Třída `EnumSet` deklarovaná `abstract EnumSet<E extends Enum<E>>`, jejíž instance jsou množinami konstant výčtových typů, je zajímavější, protože přináší několik zajímavých (alespoň pro někoho) programátorských obratů.

První, co vás na ní možná zarazí, je, že třída je deklarována jako abstraktní, avšak mezi veřejnými (a tím i mezi dostupnými) třídami nenajdete jejího konkrétního potomka, jehož instance byste mohli používat. Tento potomek ale není potřeba, protože k vy-

tváření příslušných instancí nabízí třída sadu továrních metod, které požadované množiny vytvářejí a vracejí. Z těchto metod bych uvedl především následující:

```
<E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)
```

vrátí prázdnou množinu hodnot zadaného výčtového typu.

```
<E extends Enum<E>> EnumSet<E> of(E first, E... rest)
```

vytvoří množinu zadaných konstant.

```
<E extends Enum<E>> EnumSet<E> range(E from, E to)
```

vytvoří množinu obsahující konstanty ze zadaného rozsahu včetně zadaných mezí.

Budeme-li tedy mít definován výčtový typ

```
public enum Dny { PO, ÚT, ST, ČT, PÁ, SO, NE }
```

můžeme (mimo jiné) definovat následující množiny:

```
EnumSet es = EnumSet.noneOf( Den.class );   //Prázdná množina
EnumSet psp = EnumSet.of( PO, ST, PÁ };     //Tříprvková množina {PO,ST,PÁ}
EnumSet všední = EnumSet.range( PO, PÁ );   //Množina prvků od PO do PÁ,
                                            //tj. množina {PO,ÚT,ST,ČT,PÁ}
```

Iterátory vracené instancemi třídy `EnumSet` se chovají obdobně jako iterátory vracené instancemi množiny `EnumMap`:

- vracejí hodnoty z množiny v pořadí jejich ordinálních čísel,
- nikdy nevyhodí výjimku `ConcurrentModificationException`,
- mohou (ale nemusí) respektovat změny v obsahu množiny, které se udály v průběhu iterace.

Kapitola 7

Anotace (metadata)

Anotace jsou další z poměrně zásadních rozšíření syntaxe jazyka. Protože je to však konstrukce poměrně mladá, nejsou s ní prozatím žádné rozsáhlejší zkušenosti, a to ani při práci s jinými jazyky. Tato kapitola proto nebude tak hýřit příklady „ze života", jako její předchůdkyně, které probíraly konstrukce, jejichž užitečnost a typické příklady použití jsou již dávno známé.

V této kapitole si nejprve vysvětlíme, co to vlastně anotace jsou, k čemu bychom je mohli v našich programech využít a jak se vlastně používají. Pak se seznámíme s anotacemi, které čekají ve standardní knihovně, a nakonec si ukážeme, jak bychom mohli definovat anotace vlastní.

Programy:
Všechny třídy, o nichž se v této kapitole hovoří, jsou definovány v balíčku `rup.česky.java15.anotace`.

Anotace (často se používá také termín *metadata*) jsou značky vkládané do zdrojového kódu a určené většinou pro různé nástroje, které s daným programem pracují. Tyto značky mohou být zpracovány v následujících etapách (někdy jen v jedné, jindy klidně ve všech třech):

- mohou být určeny pouze pro nástroje pracující se zdrojovým kódem (např. `javadoc`),
- mohou být začleněny do přeloženého class-souboru a určeny pro nástroje pracující s těmito soubory (např. pro nástroje připravující instalaci (deployment) na aplikační server) před vlastním spuštěním programu,
- mohou být začleněny do class-souboru a určeny pro zpracování za běhu programu.

Označování deklarací anotacemi

Metadata jsou data o datech přidávaná do zdrojového kódu. Umožňují přidat do programu další informace, které však nijak neovlivní chování programu. Přidáním anotací (metadat) se sémantika programu nijak nezmění.

Jak jsem již řekl, jimi dodané informace jsou určeny především pro vývojové nástroje. Pomohou jim kontrolovat splnění některých požadavků, které doposud nebylo možno ověřit (např. že definovaná metoda nepřetěžuje, ale doopravdy překrývá svoji jmenovkyni v rodičovské třídě), a umožní jim, aby na základě dodaných informací vytvořily další soubory, které by jinak vývojář musel otrocky psát ručně (např. různé testovací třídy a metody nebo povinná doprovodná rozhraní při tvorbě EJB).

Anotace patří mezi modifikátory a stejně jako ostatní modifikátory se vkládají před anotované položky (deklarace a definice) bez oddělujícího středníku. Podle konvencí se anotace umísťují před všechny ostatní modifikátory, ale není to povinné.

Od jiných syntaktických prvků odlišuje anotace znak `@` (šnek, zavináč), jenž jim předchází. Tento znak používají jako předponu i značky v dokumentačních komentářích. Na rozdíl od těchto značek se však anotace nevyskytují uvnitř komentářových závorek, ale jsou součástí kódu.

I když se všechny texty o anotacích, které mi prošly rukama, tvářily, že znak `@` je součástí identifikátoru anotace, není tomu tak. Znak `@` vystupuje v kódu sám za sebe (jak říká specifikace: je to samostatný syntaktický element) a pouze oznamuje, že identifikátor, který za ním následuje, je identifikátorem anotace.

Poznámka:

Teoreticky vám tedy nic nebrání vložit mezi znak @ a následující identifikátor libovolný počet bílých znaků a/nebo komentářů. Nebývá to ale zvykem a program se tím jenom znepřehlední (v této knize proto tuto možnost nebudu využívat). Je ale důležité o samostatnosti znaku @ vědět pro případ, že by vám tento znak někam upadl do textu, a vy byste se pak divili, proč překladač považuje následující identifikátor za anotaci.

Konec teorie, začněme s příklady. Představte si, že vaše IDE je schopno reagovat na anotaci `@Testovat`, kterou můžete označit metody, pro něž má IDE automaticky vygenerovat prázdné testy. Příkladem použití takovéto anotace může být např. následující kód:

```
public MojeGeniálníTřída
{
  //...
  @Testovat
  public void chytráMetoda() {
    //...
  }
}
```

Poznámka:
Všimněte si, že bezparametrická anotace nemusí být následována dvojicí prázdných závorek, jak tomu musí být u metod. Kdo však chce, ten je tam samozřejmě uvést může.

Jak jsem naznačil v předchozí poznámce, anotace mohou mít parametry. Pokud bychom např. rozdělili prováděné testy podle časové náročnosti na operativní, noční a víkendové, definovali pro ně výčtový typ `Druh` a zavedli tuto informaci jako parametr anotace, mohli bychom předchozí anotaci v závislosti na její definici zapsat např.:

```
@Testovat( Druh.OPERATIVNÍ )
```

nebo

```
@Testovat( druh = Druh.OPERATIVNÍ );
```

Poznámka:
O tom, jak musí být anotace definována, aby mohla být takto volána, se zmíníme o kus dále v kapitole Syntaxe definice anotací *na straně 118.*

Jak se za chvíli dozvíte, parametrem anotací nemusí být pouze jednoduchá hodnota, ale může jím být i celý vektor (= jednorozměrné pole) hodnot. Pokud bychom např. rozšířili předchozí anotaci o parametr `frekvence`, ve kterém bychom uváděli hodnoty výčtového typu `Den` označující jednotlivé dny v týdnu, mohli bychom některou metodu označit třeba následovně:

```
@Testovat( druh=Druh.NOČNÍ, frekvence={Den.PO, Den.ST, Den.PÁ} )
```

Jak jste si jistě všimli, vektorový parametr jsme nezadávali způsobem, který bychom použili v programu, tj.

```
frekvence = new Den[] {Den.PO, Den.ST, Den.PÁ}
```

ale uvedli jsme seznam požadovaných hodnot rovnou do složených závorek. Právě uvedený klasický způsob předání vektoru je u parametrů anotací syntakticky chybný. Ve skutečnosti se totiž žádný vektor nevytváří a výčet ve složených závorkách je jediným povoleným způsobem, jak v daném parametru předat anotaci skupinu hodnot.

Tip:
Potřebujete-li zadat pouze jediný prvek, nemusíte jej vkládat do složených závorek, překladač si je domyslí.

Deklaraci můžeme označit více anotacemi. V takovém případě se anotace prostě skládají za sebou (nebo pod sebe – co je komu sympatičtější) – např.:

```
@Anotace_1  @Anotace2( parametr="dobře" )
@Anotace3() public void anotovanáMetoda() {}
```

Použité anotace ale musí být navzájem různé. Není možné označit deklaraci stejnou anotací dvakrát – následující deklarace způsobí syntaktickou chybu.

```
@Anotace2  @Anotace2( parametr="špatně" ) private int;
```

Představme si, že máme definovánu:

- jednoparametrickou anotaci `@A1`, které můžeme předávat parametr bez pojmenování,
- dvouparametrickou anotaci `@A2` s řetězcovým parametrem `text` a celočíselným parametrem `číslo`,
- jednoparametrickou anotaci `@AA1`, jejímž parametrem je vektor anotací `@A1`.

S využitím těchto anotací bychom mohli definovat následující skrz naskrz anotovanou třídu:

```
 1 @A1( "Anotace třídy" )
 2 public class Anotovaná
 3 {
 4   @A1( "Anotace atributu" )  String atribut = "Atribut";
 5
 6   @A1( "Anotace metody" )
 7   @AA1( {@A1("String lp"), @A1("int číslo")} )
 8   public void metoda( @A2(text="parametr int i", číslo=1) int i,
 9                       @A2(text="parametr String s", číslo=2) String s )
10   {
11     @A2(text="lokální String lp",číslo=1) String lp = "Lokální proměnná";
12     @A2(text="lokální int číslo",číslo=2) int číslo = -99;
13   }
14
15   @A1( "Anotace vnořeného rozhraní" )
16   interface Rozhraní
17   {
18     @A1( "Metoda vnořeného rozhraní")
19     void metoda();
20   }
21
22   @A1( "Anotace vnořeného výčtového typu" )
23   enum Kvalita
24   {
25     @A1( "Konstanta výčtového typu") DOBRÁ,
26     @A1( "Konstanta výčtového typu") ŠPATNÁ;
27   }
28 }
```

V této definici jsem se pokusil použít anotace všech možných elementů s výjimkou anotací definic anotací (ty probereme, až se budeme učit vytvářet vlastní anotace) a anotací balíčků – ty probereme nyní.

Anotování balíčků

Z předchozího výkladu by vám již mělo být jasné, jak lze anotovat třídy, rozhraní, atributy, metody, parametry metod i jejich lokální proměnné. Anotace je modifikátor a prostě ji přidáte mezi modifikátory příslušného objektu (podle konvence by měla být uvedena jako první).

Jak jsem již naznačil, anotovat je možné i celý balíček. Ten sice žádné modifikátory za běžných okolností nemá, ale anotaci před klíčové slovo `package` přidat můžete. S anotováním balíčku je však spojeno několik dalších podmínek, které si nyní probereme.

Prvním omezením je, že anotace smí být balíčku přiřazeny pouze na jediném místě. Aby nedocházelo k násobnému přiřazení anotací v několika souborech, doporučuje specifikace jazyka (a překladač v SDK firmy *Sun* to i vyžaduje), aby byl pro přiřazení anotací balíčku definován soubor `package-info.java` (v názvu je opravdu pomlčka), který bude obsahovat pouze anotovaný příkaz `package`. Budete-li anotovat příkaz `package` v kterémkoliv jiném souboru, bude to překladač považovat za syntaktickou chybu.

Využijeme-li dosud známé anotace, mohli bychom soubor `package-info.java` balíčku `rup.česky.java15.anotace` definovat např. následovně:

```
1 @A0
2 @A1( "Anotace balíčku" )
3 @AA1( { @A1("rup"), @A1("česky"), @A1("java15"), @A1("Anotace")} )
4 package rup.česky.java15.anotace;
```

Se souborem `package-info.java` počítá dokonce i nová verze programu `javadoc`, takže nalezne-li ve složce daného balíčku soubor `package-info.java`, nebude se dále pídit po soubor `package.html`, do nějž byla umísťována dokumentace balíčku v dřívějších verzích, a bude číst dokumentační soubory odtud.

Jinými slovy: rozhodnete-li se vytvořit dokumentaci i pro balíček, umístěte ji místo dosavadního souboru `package.html` do souboru `package-info.java` a budete-li náhodou chtít balíček také anotovat, učiňte to v témže souboru.

Anotace ve standardní knihovně

Anotace jsou definovány obdobně jako rozhraní. Standardní knihovna obsahuje sedm anotací, které bychom podle jejich určení (a také podle umístění) mohli rozdělit do dvou skupin.

Standardní anotace

Standardní anotace jsou definovány v balíčku `java.lang`, a proto je není třeba importovat. Jsou tři:

java.lang.Deprecated

Anotace `@Deprecated` nemá žádné parametry a lze ji použít u libovolné deklarace. Oznamuje, že jí označená deklarace je považována za zavrženou (deprecated) a příslušný deklarovaný objekt (většinou metoda, občas i typ, teoreticky je ale možné označit cokoliv) by se neměl používat.

Překladač je povinen vydat varovné hlášení u každého použití prvku označeného anotací `@Deprecated` . Výjimkou jsou pouze případy, kdy:

- Zavržený prvek je použit uvnitř jednotky (metoda, třída, rozhraní), která je sama označena jako `@Deprecated`.
- Zavržený prvek je použit uvnitř stejné vnější třídy, ve které je deklarován.
- Použití zavrženého prvku se nachází uvnitř jednotky, v níž je vydávání varovných zpráv potlačeno anotací `@SuppressWarnings("deprecation")`.

> **Pozor!**
> *Poslední tvrzení je sice doslovný překlad ze specifikace, nicméně překladač v JDK 1.5.0_03, který jsem používal v době závěrečných úprav knihy, stále ještě na tuto anotaci neslyšel a vesele vydával varovná hlášení dál.*

java.lang.Override

Anotace `@Override` je bezparametrická a je určena k označení metod, které mají překrýt stejnojmenné metody v rodičovské třídě. Umožňuje tak překladači, aby zkontroloval, že metoda svoji jmenovkyni doopravdy překrývá a že ji pouze nepřetěžuje. Zjistí-li opak, ohlásí syntaktickou chybu.

V minulých verzích se dalo překrytí rodičovské metody poměrně snadno zkontrolovat (i tak ale vznikaly chyby z přehlédnutí). Při používání parametrizovaných typů se však lehce může stát, že je programátor přesvědčen o tom, že metoda rodičovskou metodu překrývá, a neuvědomí si, že ji pouze přetěžuje.

> **Pozor!**
> *Za chybu je považováno i to, když `@Override` metoda překrývá pouze metodu implementovaného rozhraní, avšak nepřekrývá žádnou z metod rodičovské třídy. Překrýt metody implementovaného rozhraní totiž beztak musí – kvůli tomu by nebylo třeba zavádět anotaci `@Override`. Když už si však programátor dá práci s označením metody touto anotací, trvá překladač na tom, aby byla překryta některá z metod rodičovské třídy.*

java.lang.SuppressWarnings

Anotace `@SuppressWarnings` má jako parametr definován vektor textových řetězců specifikujících varování, která má překladač při překladu anotované entity (a případně všech jí obsahovaných elementů) potlačit.

Vydávání varovných hlášení je možné pouze potlačit a není je možné znovu nastavit. Potlačíte-li některá varování pro třídu, budou tím automaticky potlačena u všech jejích

členů (atributů, metod, vnořených tříd), a vy u těchto členů už budete moci seznam potlačených varování pouze rozšířit.

Při potlačování varovných hlášení je proto vhodné toto potlačení nastavit co nejhlouběji: chcete-li ve vybrané metodě potlačit některé z varování, nastavte toto potlačení pouze pro danou metodu.

Parametry anotace se zadávají jako seznam čárkami oddělených textových řetězců specifikujících potlačovaná varování. Překladači nebude vadit, uvedete-li některé z varování vícekrát. Obdobně mu nebude vadit, uvedete-li varování, které nezná – prostě tento požadavek ignoruje. Může však vydat varování, že ono potlačované varování nezná.

Varování, jejichž povolení překladač nekontroluje (a která tedy není možno potlačit), obsahují na výstupu povinně slovo `uhchecked`.

Tvůrci překladačů mají v dokumentaci svého překladače uvést seznam varování, která je možno touto anotací potlačit. Překladač firmy *Sun* z JDK 1.5.0_03 však ani v době závěrečných úprav tohoto textu žádné varování potlačit neuměl.

Metaanotace

Metaanotace jsou anotace určené k označení definic jiných anotací (metaanotace bychom tak mohli nazvat metametadaty). Pomocí metaanotací dodáváme překladači, pomocným vývojovým nástrojům a případně i virtuálnímu stroji další informace o charakteru, chování a vlastnostech právě definované anotace.

Ve standardní knihovně jsou všechny předdefinované metaanotace umístěny v balíčku `java.lang.annotation`.

Poznámka:

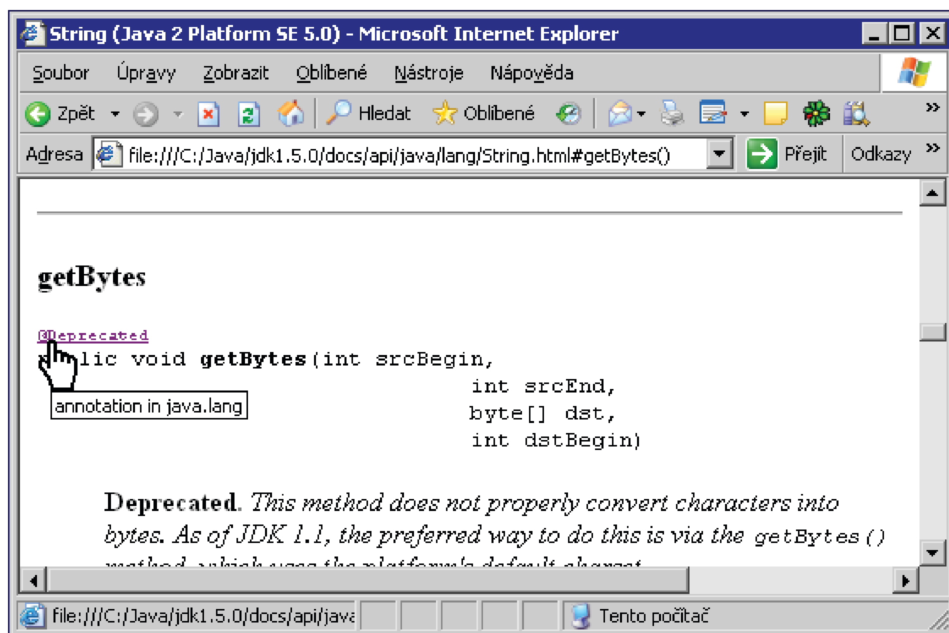
V následujícím přehledu nebudu uvádět moc příkladů, protože ještě neumíte definovat vlastní anotace. To jsem vás ale nemohl doposud naučit, protože při definování vlastních anotací je potřeba znát metaanotace, které vlastnosti námi definovaných anotací doladí. Až se naučíte definovat anotace vlastní, ukážeme si pár definic a chování metaanotací si na nich vyzkoušíme.

java.lang.annotation.Documented

Metaanotace `@Documented` nemá žádné parametry. Sděluje, že jí označená anotace musí být uvedena v dokumentaci touto anotací označených entit (typů, atributů, metod, anotací). Program generující dokumentaci (např. `javadoc`) ji pak bude uvádět jako součást popisu deklarace dané entity.

Příkladem anotace, která je označená metaanotací `@Documented`, je anotace `@Deprecated`. Touto anotací jsou v nové verzi knihovny označeny všechny zavržené metody. Podíváte-li se do dokumentace kterékoliv z nich, najdete v její hlavičce uvedenou i anotaci `@Deprecated` (viz obrázek 7.1, na němž je část dokumentace zavržené verze metody `getBytes` ze třídy `String` – na uvedení anotace ukazuje kurzor).

Poznámka:
V dalším textu budu místo „anotace je označená metaanotací `@Documented`" říkat jenom „anotace je `@Documented`", „anotace je `@Deprecated`" apod.



Obrázek 7.1: *Dokumentace zavržené verze metody getBytes ze třídy String*

java.lang.annotation.Inherited

Metaanotace `@Inherited` indikuje, že anotace, jejíž definici s ní označíme, se stane součástí dědictví dceřiných tříd touto anotací označené třídy. Zeptáme-li se proto některé třídy na to, je-li označena anotací, o níž víme, že je `@Inherited`, pak v případě záporné odpovědi bude automaticky dotázána její rodičovská třída, a pokud neuspějeme ani zde, tak rodičovská třída této rodičovské třídy. V rekurzivním dotazování dalších a dalších rodičů v hierarchii budeme pokračovat tak dlouho, dokud nebude nalezena třída označená danou anotací nebo dokud nebude dosaženo třídy `Object`.

Z výše popsaného mechanizmu vyplývá, že metaanotací `@Inherited` má smysl označovat pouze anotace, na které je možné se ptát za běhu (viz dále popis metaanotace `@Retention`).

Anotace, která je `@Inherited`, se uplatní pouze tehdy, budou-li jí označeny třídy. Označíme-li s ní rozhraní, nijak se to u jeho potomků neprojeví.

java.lang.annotation.Retention

Metaanotace `@Retention` definuje, jak dlouho bude uchovávána informace předaná anotací, jejíž definice je danou instancí `@Retention` označena. Anotace je jednoparame-

trická s parametrem typu `java.lang.annotation.RetentionPolicy` (výčtový typ), jenž může nabývat následujících hodnot:

`SOURCE`	Takto označená anotace je určena pouze pro programy pracující se zdrojovým kódem (např. překladač). Do přeloženého class-souboru se již daná informace „neprobojuje“, protože ji tam překladač nevloží.
`CLASS`	Takto označená anotace se „probojuje“ až do přeloženého class-souboru. V něm ji mohou najít programy, které s tímto souborem pracují – např. programy pro instalaci (deployment) aplikací a knihoven. Zavaděč tříd (class loader) ji však již nezpracuje a do class-objektu dané třídy ji nezačlení. Anotací předanou informaci proto nebude možno využít za běhu programu, protože o ní virtuální stroj nebude vědět.
`RUNTIME`	Takto označenou anotací předávaná informace se „probojuje“ až do class-objektu třídy, jejíž členy (či celou třídu) anotace označuje. Prostřednictvím mechanizmu reflexe proto bude možno za běhu příslušnou informaci zjistit a podle ní se zařídit.

Nepředáme-li metaanotaci žádný parametr, zvolí zlatou střední cestu a označí definovanou anotaci stejně, jako kdybychom zadali parametr `CLASS`.

java.lang.annotation.Target

Metaanotace `@Target` specifikuje, které druhy entit je možno jí anotovanou anotací označit. Je jednoparametrická, přičemž její parametr je vektor hodnot výčtového typu `java.lang.annotation.ElementType`. Jednotlivé hodnoty povolují označit anotovanou anotací následující definice a deklarace:

`ANNOTATION_TYPE`	Definice jiné anotace.
`CONSTRUCTOR`	Definice konstruktoru.
`FIELD`	Deklarace atributu.
`LOCAL_VARIABLE`	Deklarace lokální proměnná.
`METHOD`	Deklarace metody.
`PACKAGE`	Deklarace balíčku.
`PARAMETER`	Deklarace parametru.
`TYPE`	Deklarace třídy, rozhraní (včetně anotací) nebo výčtového typu.

Neoznačíme-li definici anotace anotací `@Target`, bude to stejné, jako kdybychom ji označili a v parametru vyjmenovali všechny typy deklarací a definic. Anotací, která nebude označena metaanotací `@Target`, bude proto možno označit kteroukoliv definici či deklaraci.

Poznámka:
Anotace lokálních proměnných se nemají šanci „probojovat" do přeloženého class-souboru, a je proto možné je využívat pouze při práci se zdrojovým kódem.

Syntaxe definice anotací

Už jsme si o anotacích a jejich možném použití povídali dost dlouho. Nejvyšší čas ukázat si, jak bychom mohli definovat anotaci vlastní. Syntaxi definice anotací bychom mohli popsat následovně:

```
Modifikátory @interface NázevAnotace
{
  DeklaracePrvku
  DeklaracePrvku
  ...
}
```

kde případná *DeklaracePrvku* (nemusí tam být také žádná) má tvar buď

```
Typ NázevPrvku();
```

nebo

```
Typ NázevPrvku() default ImplicitníHodnota ;
```

Jak vidíte, anotaci definujeme obdobně jako rozhraní, dokonce použijeme i klíčové slovo `interface`. Oproti definici rozhraní tu jsou pouze dvě viditelné odchylky:

- před klíčové slovo `interface` se vkládá znak `@`,
- za deklaraci metod je možné vložit definici implicitní hodnoty.

Poznámka:
Stejně jako u anotací, ani na klíčového slovo `interface` *nemusí být znak* `@` *nalepen. Bývá ale zvykem jej přilepovat.*

Další odchylky již nejsou viditelné, ale je třeba o nich vědět:

- Typem návratové hodnoty metod deklarovaných v anotaci nemůže být cokoliv – množina povolených typů je poměrně omezená. Jako typ návratové hodnoty metody anotace můžete použít:
  - primitivní typ (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, `boolean`),
  - `String`,
  - `Class` (může mít zadány typové parametry – např. `Class<? extends Něco>`),
  - výčtový typ (`enum`),

- jinou anotaci, která nemá mezi svými metodami takovou, jež vrací odkaz na právě deklarovanou anotaci nebo na anotaci, která takovou metodu ve svém repertoáru má (stručně: nesmí vzniknout cyklický odkaz),
- vektor výše uvedených typů.

Poznámka:
Když jsem v posledním bodu uvedl, že návratovým typem anotačních metod může být vektor výše uvedených typů, myslel jsem tím doopravdy pouze jednorozměrně pole těchto typů. Pole polí, tj. dvou a vícerozměrná pole, nejsou jako návratové typy anotací podporována.

- Anotace nesmí deklarovat žádného předka. Všechny anotace jsou vnímány jako potomci rozhraní `java.lang.annotation.Annotation`, ale nesmí to explicitně deklarovat. (Je to obdobné jako u výčtových typů, které také nesmějí deklarovat, že jsou potomky třídy `Enum`.)

Jednoduchá značkovací anotace

Poznámka:
Anotace, které nedefinují žádné parametry, nazývají někteří autoři značky *(markers) případně* značkovací anotace *(marker annotations).*

Budeme-li chtít definovat jednoduchou, bezparametrickou anotaci `@Značka`, kterou bychom posléze označovali různé deklarace, mohla by její definice vypadat následovně:

```
1 @java.lang.annotation.Retention(
     java.lang.annotation.RetentionPolicy.RUNTIME)
2 public @interface Značka  {}
```

Jak vidíte, označil jsem ji jako anotaci, která své informace protlačí až do doby běhu programu. Tam se je také za chvíli naučíme zjišťovat. Pokud se navíc pamatujete na metaanotaci `@Target`, tak si jistě vybavujete, že díky tomu, že jsem naši anotaci touto metaanotací neoznačil, můžeme s ní označovat cokoliv.

Nyní definujme demonstrační třídu `Značkovaná`, jejíž některé atributy a metody právě definovanou anotací označíme. Za chvíli si pak ukážeme, jak je možné označkovanost příslušných atributů a metod zjistit.

```
 1 @SuppressWarnings( "nesmysl" )
 2 @Deprecated
 3 @Značka
 4 public class Značkovaná
 5 {
 6   @Značka
 7   public static final String   ZNAČKOVANÝ = "Atr. třídy - značkovaný";
 8   public static final String NEZNAČKOVANÝ = "Atr. třídy - neznačkovaný";
 9
10   @Značka
```

```
11   private String   značkovaný = "Atr. instance - značkovaný";
12   private String neznačkovaný = "Atr. instance - neznačkovaný";
13
14             public        Značkovaná(){}
15   @Značka   public        Značkovaná( int i ) {}
16   @Značka() public void   značkovaná( int i ) {}
17             public void neznačkovaná(){}
18 }
```

Předáte-li class-objekt této třídy jako parametr metodě `Analýza.anotace(Class)` (za chvíli se ji naučíte naprogramovat sami), vytiskne vám na standardní výstup následující text:

```
 1 TEST TŘÍDY class rup.česky.java15.anotace.Značkovaná
 2    Anotace třídy:
 3      @java.lang.Deprecated()
 4      @rup.česky.java15.anotace.Značka()
 5    Analýza konstruktorů
 6      Neanotované:
 7        public rup.česky.java15.anotace.Značkovaná()
 8      Anotované:
 9        public rup.česky.java15.anotace.Značkovaná(int)
10          @rup.česky.java15.anotace.Značka()
11    Analýza metod
12      Neanotované:
13        public void rup.česky.java15.anotace.Značkovaná.neznačkovaná()
14      Anotované:
15        public void rup.česky.java15.anotace.Značkovaná.značkovaná(int)
16          @rup.česky.java15.anotace.Značka()
17    Analýza atributů
18      Neanotované:
19        public static final java.lang.String
   rup.česky.java15.anotace.Značkovaná.NEZNAČKOVANÝ
20        private java.lang.String rup.česky.java15.anotace.Značkovaná.neznačkovaný
21      Anotované:
22        public static final java.lang.String
   rup.česky.java15.anotace.Značkovaná.ZNAČKOVANÝ
23            @rup.česky.java15.anotace.Značka()
24        private java.lang.String rup.česky.java15.anotace.Značkovaná.značkovaný
25          @rup.česky.java15.anotace.Značka()
```

Poznámka:

Řádky 19 a 22 předchozího výpisu byly bohužel příliš dlouhé, takže jsou zobrazeny zlomené.

Metoda vypsala nejprve anotace, jimiž byla označena třída. Všimněte si, že o anotaci `@SuppressWarning` není v textu ani zmínka. Tato anotace je totiž určena pouze pro

překladač (obecně bychom mohli říci pro programy pracující se zdrojovým kódem), a proto se o ní při běhu programu již nic nedozvíme.

Po třídě si pak vzala na paškál postupně konstruktory, metody a atributy dané třídy a jejích instancí. Pro každou z těchto tří skupin nejprve vypsala, které členy dané skupiny nejsou anotované, a pak vypsala ty anotované spolu se seznamem jejich anotací.

Anotace s parametry

Bezparametrické anotace jsme zvládli a můžeme tedy přejít k anotacím s parametry. Parametry anotací jsou trochu jiného druhu než parametry metod, s nimž jste pracovali doposud. Členy anotací jsou metody vracející hodnotu. Prostřednictvím parametrů anotace zadáváte hodnotu, kterou budou tyto metody při svém vyvolání vracet.

Poznámka:

V dalším textu se budou vyskytovat dva zvukomalebně podobné termíny:

- *termín **anotační metoda** budu používat pro metody, které jsou součástí deklarace anotace a jejichž návratové hodnoty zadáváme prostřednictvím parametrů anotace,*
- *termín **anotovaná metoda** budu používat pro metody označené nějakou anotací.*

Nezaměňte je.

Možná si vzpomínáte, jak jsme na počátku této kapitoly používali anotaci `@Testovat`, která měla na závěr dva parametry: `druh` a `frekvence`. Takovouto anotaci bychom mohli deklarovat např. následovně:

```
1 public @interface Testovat
2 {
3   Druh  druh();
4   Den[] frekvence();
5 }
```

Na předchozím prográmku si všimněte dvou věcí:

- každému z možných parametrů anotace je přiřazena jeho anotační metoda,
- anotační metody nemají parametry (ani je mít nesmějí).

Tato definice má pro nás ale jednu nevýhodu: trvá na tom, abychom jejím parametrům přiřadili hodnoty. Pokud bychom chtěli mít možnost žádnou hodnotu parametrům nepřiřazovat s tím, že se v takovém případě použije hodnota implicitní, musíme definici doplnit o zadání této implicitní hodnoty. Upravená definice by mohla mít např. tvar:

```
1 import static rup.česky.java15.anotace.Den.*;
2
3 public @interface Testovat
4 {
```

```
5   Druh  druh()      default Druh.OPERATIVNÍ;
6   Den[] frekvence() default { PO, ÚT, ST, ČT, PÁ, SO, NE };
7 }
```

Možná vás napadlo, proč jsem na řádku 6 vyjmenovával všechny možné hodnoty a nenechal jsem vytvořit příslušný vektor zavoláním metody `Den.values()`. Problém je v tom, že zadávané hodnoty musí být vyhodnotitelné v době překladu, a to pro objekt vracený metodou `Den.values()` v žádném případě neplatí – ten vznikne až za běhu.

Přiznejme si, že to, co jsem deklaroval jako implicitní hodnotu pro anotační metody vracející vektor hodnot, není pravověrný vektor, tj. objekt, jenž je jednorozměrným polem. Je to jenom skupina hodnot, pro něž bude požadovaný vektor vytvořen až za běhu.

Možná namítnete, že hodnoty výčtových typů a typu `String` jsou také objekty, které budou vytvářeny až za běhu, jenomže to není totéž. Na tyto hodnoty je překladač „duševně připraven“ a má pro ně připravenou dočasnou interní reprezentaci.

Když už jsme u těch zákazů, přidám ještě jeden: při zadávání implicitních hodnot anotačních metod nesmíte použít konstantu `null`! Potřebujete-li dosadit za implicitní hodnotu něco prázdného, musíte zvolit náhradní řešení – např.:

`String` Prázdný řetězec `""`,

Anotace Anotaci `Null` (tu byste sice museli nejprve definovat, ale to už pro vás není problém),

Výčet Rozšířit množinu hodnot daného výčtového typu o nějakou příhodnou hodnotu – např. výčtový typ `Směr8` má mezi svými konstantami vedle osmi hlavních a vedlejších světových stran i hodnotu `ŽÁDNÝ`.

Zjednodušení zápisu jediného parametru

Mezi možnými parametry anotací je jeden, který má výjimečné postavení. Tento parametr má předdefinované jméno `value` a jeho zvláštností je to, že pokud budete zadávat jenom jej, nemusíte používat „ukecanou“ syntaxi *název=hodnota*, ale můžete uvést pouze zadávanou hodnotu.

Upravme proto ještě jednou definici naší anotace a přidejme do ní deklaraci metody `value()`:

```
1 import static rup.česky.java15.anotace.Den.*;
2
3 public @interface Testovat
4 {
5   Druh   druh()      default Druh.OPERATIVNÍ;
6   Den[]  frekvence() default { PO, ÚT, ST, ČT, PÁ, SO, NE };
7   String value()     default "Nezadáno";
8 }
```

Nyní můžeme anotovat testované metody kterýmkoliv z následujících tvarů (následující text berte pouze jako seznam možných tvarů – syntakticky by takovéto označení neprošlo, protože jednou anotací nesmíme nic označit vícekrát):

```
1 @Testovat
2 @Testovat()
3 @Testovat( "Není kompletní" )
4 @Testovat( value = "Není kompletní" )
5 @Testovat( frekvence= {Den.ST, Den.So} )
6 @Testovat( frekvence=Den.PO,  druh = Druh.NOČNÍ )
7 @Testovat( value="Vylepšená", frekvence={Den.SO, Den.ÚT} )
```

Tvary na 1. a 2. řádku jsme již probrali. Řádek 3 a 4 hovoří o tom, že zadáváme-li samotný parametr `value`, je na našem rozhodnutí, budeme-li uvádět i jeho jméno s rovnítkem. Průzračné jsou jistě i zadání anotace na řádcích 5 a 6.

Chtěl bych se ale zastavit u anotace na posledním řádku. Zde nemůžeme název parametru `value` vynechat, protože není mezi závorkami sám. Jakmile potřebujeme spolu s parametrem `value` zadat ještě nějaký jiný, musíme uvádět jména u všech parametrů včetně parametru `value`.

Anotace jako parametry jiných anotací

Zajímavou vlastností anotací je, že mezi jejich členy můžete zařadit i anotace. Jak jsme si ale řekli, musí se jednat o jinou anotaci a její metody nesmí vracet odkazy na instance původní anotace, a to ani zprostředkovaně – tj. nesmí tvořit cykly.

Přímým důsledkem toho, že návratovými hodnotami anotačních metod mohou být jiné anotace, je i možnost zadat anotaci jako parametr jiné anotace.

Další vlastnosti anotací

- Anotace smí označit svoji vlastní definici.
- Je rozumné uvádět parametry ve stejném pořadí, v jakém byly deklarovány, i když to vzhledem k jejich pojmenování není nutné (považuje se to však za slušné).
- Implicitní hodnoty se anotačním metodám přiřazují až za běhu. Pokud tedy nějaká třída používá anotace, tak se v ní nesmíte spoléhat na nastavení implicitních hodnot. Bude-li používaná anotace později přeložena znovu s jinou sadou implicitních hodnot, anotované entity převezmou tuto novou sadu.
- Anotace jsou potomky rozhraní `java.lang.annotation.Annotation`, které je však jen obyčejným rozhraním a není anotací. Tuto skutečnost však nesmějí mít ve své hlavičce uvedenu obdobně, jako nesmějí mít výčtové typy v hlavičce uvedeno, že jsou potomky třídy `Enum`.
- Anotace mohou mít potomky (rozhraní), avšak ti už nebudou anotacemi.
- Anotace nesmí být definovány jako parametrické typy, tj. nesmí mít typové parametry.

- Anotační metody nesmí mít typové parametry. Typové parametry mohou být uvedeny pouze k bližší specifikaci návratové hodnoty typu `Class<?>`.
- Anotační metody nesmí mít uvedenu klauzuli `throws`.
- Těla anotačních tříd mohou obsahovat stejné další členy jako běžná rozhraní (statické konstanty, vnořené třídy). Uvnitř anotačního typu může být např. definován výčtový typ použitý jako typ návratových hodnot jeho anotačních metod.
- Každá deklarace metody představuje element anotačního typu. Anotační typ nemůže mít jiné elementy než ty, které jsou explicitně deklarovány. Jinými slovy: přestože je anotace potomkem rozhraní `java.lang.annotation.Annotation`, metody zděděné od tohoto rozhraní ani metody zděděné od třídy objekt nejsou elementy anotace, tj. nepatří mezi anotační metody.

Získávání informací o anotacích za běhu programu

Už umíme anotace definovat, takže ještě zbývá se naučit využívat informace, které tyto anotace do kódu dodají. V této publikaci se omezím pouze na zpracování informací, které můžete získat za běhu programu. O tom, kde získat informace pro práci s anotacemi v předchozích fázích, se dozvíte v následující podkapitole.

Nejprve jedno drobné omezení: za běhu můžete získat anotační informace pouze od tříd, jejich členů a parametrů metod. Lokální proměnné jsou ze hry o anotace vyřazeny, protože se jejich anotace (podle definice) vůbec nedostanou do class-souboru. Anotace lokálních proměnných proto mohou využívat pouze nástroje pracující se zdrojovým kódem.

Na druhou stranu si ale přiznejme, že lokální proměnné jsou něco tak soukromého a dočasného, že by ani nebylo vhodné se snažit po jejich vlastnostech zvenku pídit.

Získání anotací třídy a jejích členů

V minulé podkapitole jsme definovali třídu `Značkovaná`, jejíž některé atributy a metody jsme označili bezparametrickou anotací `Značka`. Nyní si ukážeme, jak je možno za běhu tyto informace získat.

Základem práce s anotacemi je reflexe. Kdykoliv budete potřebovat zjistit nějakou informaci uloženou anotací, musíte začít od příslušného class-objektu dané třídy, který poskytuje metody, jež nám umožní se dozvědět více.

Podívejte se na definici metody `vypišAnotace(Class)`, která převezme ve svém parametru class-objekt třídy a má za úkol vypsat anotace přiřazené této třídě, jejím metodám a atributům:

```
 1 public static <T> void vypišAnotace( Class<T> třída )
 2 {
 3   System.out.println("\nTEST TŘÍDY " + třída );
 4
 5   List<Annotation> at;
 6
 7   List<Constructor> ac = new ArrayList<Constructor>();
 8   List<Constructor> nc = new ArrayList<Constructor>();
 9   List<Method>      am = new ArrayList<Method>();
10   List<Method>      nm = new ArrayList<Method>();
11   List<Field>       af = new ArrayList<Field>();
12   List<Field>       nf = new ArrayList<Field>();
13   try {
14     at = Arrays.asList( třída.getAnnotations() );
15     for( Constructor c : třída.getDeclaredConstructors() )
16       if( c.getAnnotations().length > 0 )
17         ac.add( c );
18       else
19         nc.add( c );
20     for( Method m : třída.getDeclaredMethods() )
21       if( m.getAnnotations().length > 0 )
22         am.add( m );
23       else
24         nm.add( m );
25     for( Field f : třída.getDeclaredFields() )
26       if( f.getAnnotations().length > 0 )
27         af.add( f );
28       else
29         nf.add( f );
30   }
31   catch( Exception e ) {
32     throw new RuntimeException( e );
33   }
34   System.out.println( "Anotace třídy:             " + at );
35   System.out.println( "Anotované konstruktory:   " + ac );
36   System.out.println( "Neanotované konstruktory: " + nc );
37   System.out.println( "Anotované metody:         " + am );
38   System.out.println( "Neanotované metody:       " + nm );
39   System.out.println( "Anotované atributy:       " + af );
40   System.out.println( "Neanotované atributy:     " + nf );
41 }
```

Veškerá logika metody je soustředěna ve třech cyklech, v nichž program požádá třídu o vektor jejích konstruktorů, metod a atributů a pak se každého z nich zeptá, obsahuje-li nějaké runtimové, tj. za běhu dostupné anotace. Podle odpovědi pak zařadí příslušný element mezi anotované nebo neanotované a na závěr vypíše pro každou skupinu seznam anotovaných a neanotovaných členů.

Poznámka:

Metoda se nesnaží pídit po dalších anotovatelných entitách, tj. vnořených třídách a parametrech metod. Předpokládám však, že takovéto rozšíření byste ale se současnými znalostmi měli umět naprogramovat sami.

Kdybyste zavolali tuto metodu s parametrem `Značkovaná.class`, vypsala by na standardní výstup následující text (poslední dva řádky byly příliš dlouhé, tak jsem je ručně zlomil, aby byly čitelnější):

```
1 TEST TŘÍDY class rup.česky.java15.anotace.Značkovaná
2 Anotace třídy:             [@java.lang.Deprecated(), @rup.česky.java15.anotace.Značka()]
3 Anotované konstruktory:    [public rup.česky.java15.anotace.Značkovaná(int)]
4 Neanotované konstruktory: [public rup.česky.java15.anotace.Značkovaná()]
5 Anotované metody:        [public void rup.česky.java15.anotace.Značkovaná.značkovaná(int)]
6 Neanotované metody:      [public void rup.česky.java15.anotace.Značkovaná.neznačkovaná()]
7 Anotované atributy:      [
     public static final java.lang.String rup.česky.java15.anotace.Značkovaná.ZNAČKOVANÝ,
     private java.lang.String rup.česky.java15.anotace.Značkovaná.značkovaný]
8 Neanotované atributy:    [
     public static final java.lang.String rup.česky.java15.anotace.Značkovaná.NEZNAČKOVANÝ,
     private java.lang.String rup.česky.java15.anotace.Značkovaná.neznačkovaný]
```

Kdybyste ji zavolali s parametrem `Anotovaná.class`, vypsala by:

```
1 TEST TŘÍDY class rup.česky.java15.anotace.Anotovaná
2 Anotace třídy:             [@rup.česky.java15.anotace.A1(value=Anotace třídy)]
3 Anotované konstruktory:    []
4 Neanotované konstruktory: [public rup.česky.java15.anotace.Anotovaná()]
5 Anotované metody:          [
     public void rup.česky.java15.anotace.Anotovaná.metoda(int,java.lang.String)]
6 Neanotované metody:        []
7 Anotované atributy:        [java.lang.String rup.česky.java15.anotace.Anotovaná.atribut]
8 Neanotované atributy:      []
```

Poznámka:

Na předchozím výpisu výstupu programu si na řádku 2 všimněte, že má-li anotace nějaké parametry, vypisuje při tisku i jejich hodnoty.

Předchozí metoda to se získáváním informací nepřeháněla – většinou jí stačilo zjistit pouze to, je-li daná entita anotovaná či nikoliv. Definujme nyní metodu, která nám u každého anotovaného prvku vypíše jmenovitě všechny jeho anotace.

Kdybychom metodu definovali jako pouhé rozšíření předchozí metody, zbytečně by se v ní opakoval poměrně dlouhý kód. Využijeme proto toho, že všechny třídy, jejichž instance představují entity programu (tj. třídy `Package`, `Class`, `Field`, `Constructor` a `Method`), implementují rozhraní `AnnotatedElement`, a definujeme jednoúčelovou metodu `analyzuj(String,AnnotatedElement[])`, která v prvním parametru převezme označení analyzované skupiny a ve druhém skupinu analyzovaných entit.

Metoda rozdělí tuto skupinu entit na anotované a neanotované a vypíše jejich seznam. U anotovaných prvků navíc vypíše i jejich anotace.

```
 1 private static final String M1 = "  ";
 2 private static final String M2 = M1 + M1;
 3 private static final String M3 = M2 + M1;
 4 private static final String M4 = M3 + M1;
 5
 6
 7 public static <T> void anotace( Class<T> třída )
 8 {
 9   System.out.println("\nTEST TŘÍDY " + třída );
10
11   List<Annotation> at;
12
13   try {
14     System.out.println( M1 + "Anotace třídy:" );
15     for( Annotation a : třída.getAnnotations() )
16       System.out.println("    " + a );
17     analyzuj( "konstruktorů", třída.getDeclaredConstructors() );
18     analyzuj( "metod",        třída.getDeclaredMethods() );
19     analyzuj( "atributů",     třída.getDeclaredFields() );
20   }
21   catch( Exception e ) {
22     throw new RuntimeException( e );
23   }
24 }
25
26
27 private static void analyzuj( String název, AnnotatedElement[] aae )
28 {
29   System.out.println( M1 + "Analýza " + název );
30   List<AnnotatedElement> lae = new ArrayList<AnnotatedElement>();
31   List<Annotation[]>     laa = new ArrayList<Annotation[]>();
32   List<AnnotatedElement> lne = new ArrayList<AnnotatedElement>();
33
34   for( AnnotatedElement ae : aae )
35   {
36     Annotation[] aa;
37     if( (aa = ae.getAnnotations()).length > 0 )
38     {
39       lae.add( ae );
40       laa.add( aa );
41     }
42     else
43       lne.add( ae );
44   }
45   System.out.println( M2 + "Neanotované:");
```

```
46   for( AnnotatedElement ae : lne )
47     System.out.println( M3 + ae );
48
49   System.out.println( M2 + "Anotované:");
50   for( int i = 0;   i < lae.size();   i++ )
51   {
52     System.out.println( M3 + lae.get( i ));
53     for( Annotation a : laa.get( i ) )
54       System.out.println( M4 + a );
55   }
56 }
```

Pokud bychom takto definované metodě `anotace(Class)` předali jako parametr objekt `Anotovaná.class`, vypsala by na standardní výstup následující text (řádek 13 byl příliš dlouhý, tak jsem jej ručně zalomil):

```
 1 TEST TŘÍDY class rup.česky.java15.anotace.Anotovaná
 2    Anotace třídy:
 3     @rup.česky.java15.anotace.A1(value=Anotace třídy)
 4    Analýza konstruktorů
 5     Neanotované:
 6      public rup.česky.java15.anotace.Anotovaná()
 7      Anotované:
 8    Analýza metod
 9      Neanotované:
10      Anotované:
11        public void
12          rup.česky.java15.anotace.Anotovaná.metoda(int,java.lang.String)
13         @rup.česky.java15.anotace.A1(value=Anotace metody)
           @rup.česky.java15.anotace.AA1(value=
             [@rup.česky.java15.anotace.A1(value=String lp),
14            @rup.česky.java15.anotace.A1(value=int číslo)])
15    Analýza atributů
16      Neanotované:
17      Anotované:
18        java.lang.String rup.česky.java15.anotace.Anotovaná.atribut
           @rup.česky.java15.anotace.A1(value=Anotace atributu)
```

Získání hodnot anotačních metod

Zatím jsme pouze vypisovali anotace. Občas bychom ale potřebovali v programu použít hodnoty, které byly v anotacích zadány jako parametry. Představte si, že bychom např. definovali anotaci `@Autor`, pomocí níž každý označí třídu, jejíž tvorbu a údržbu má na starosti, a jako parametr jí předá své jméno nebo přezdívku.

```
1 @java.lang.annotation.Inherited
2 @java.lang.annotation.Retention(
      java.lang.annotation.RetentionPolicy.RUNTIME )
3 public @interface Autor
```

```
4 {
5   String value();   //Nemá implicitní hodnotu => Zadání hodnoty je povinné
6 }
```

Když pak při testech něco zhavaruje, program projde výpis zásobníku vygenerované výjimky a pošle zprávu všem autorům tříd, které v seznamu vystupují. Když bychom pro testování nahradili poslání zprávy opět tiskem na standardní výstup, mohli bychom potřebnou metodu definovat např. následovně:

```
 1 import java.util.List;
 2 import java.util.ArrayList;
 3 import java.lang.annotation.Annotation;
 4
 5 public class AutořiTříd
 6 {
 7 //== KONSTANTNÍ ATRIBUTY INSTANCÍ ========================================
 8
 9   public final String zpráva;
10   public final String autor;
11
12
13 //== PROMĚNNÉ ATRIBUTY INSTANCÍ ==========================================
14 //== PŘÍSTUPOVÉ METODY ATRIBUTŮ TŘÍDY ====================================
15 //== OSTATNÍ METODY TŘÍDY ================================================
16
17   public static List<AutořiTříd> najdi( Throwable t )
18   {
19     List<AutořiTříd> ret = new ArrayList<AutořiTříd>();
20     StackTraceElement[] aste = t.getStackTrace();
21     for( StackTraceElement ste : aste )
22     {
23       String zpráva = ste.toString();
24       String autor;
25       try {
26         String názevTřídy = ste.getClassName();
27         Class<?> třída = Class.forName( názevTřídy );
28         Autor anotace = třída.getAnnotation( Autor.class );
29         if( anotace != null )
30           autor = anotace.value();
31         else
32           autor = "???";
33       }
34       catch( Exception e ) {
35         throw new RuntimeException( e );
36       }
37       ret.add( new AutořiTříd( autor, zpráva ) );
38     }
39     return ret;
```

```
40   }
41
42
43 //== KONSTRUKTORY A TOVÁRNÍ METODY ======================================
44   private AutořiTříd( String autor, String zpráva )
45   {
46     this.autor = autor;
47     this.zpráva = zpráva;
48   }
49 }
```

Všimněte si, že hodnoty parametrů anotace zjišťujeme tak, že zavoláme příslušnou metodu instance dané anotace (řádky 29 a 31).

Pro otestování si připravíme metodu náhodně generující volání metod několika „podepsaných" tříd:

```
 1 import java.util.List;
 2 import java.util.Random;
 3
 4 public class TestAutorůTříd
 5 {
 6   public static void test()
 7   {
 8     try {
 9       P.p();
10     }catch( Exception e ) {
11       System.out.println(
12         "Výpis volání metod a jejich autorů pro chybu:\n" + e );
13       List<AutořiTříd> chyby = AutořiTříd.najdi( e );
14       for( AutořiTříd at : chyby )
15         System.out.println("   Autor="  + at.autor +
16                            ",  Zpráva=" + at.zpráva );
17       System.out.println("\nPůvodní chybový výpis:");
18     }
19   }
20
21   public static void main( String... args )
22   {
23     test();
24   }
25 }
26
27 @Autor("Vojta")
28 class P
29 {
30   private static final Random rnd = new Random();
31   private static int ještě = 5;
32
```

```
33   @SuppressWarnings( "fallthrough" )
34   static void p() {
35     int r= rnd.nextInt(4);
36     if( --ještě < 0 )
37       throw new RuntimeException( "Zásoba vyčerpána");
38     switch( r ) {
39       case 0: V0.proveď();  break;
40       case 1: V1.proveď();  break;
41       case 2: V2.proveď();  break;
42       case 3: V3.proveď();  break;
43     }
44   }
45 }
46
47 @Autor("Nultý")
48 class V0 { public static void proveď() { P.p(); } }
49
50 @Autor("První")
51 class V1 { public static void proveď() { P.p(); } }
52
53 @Autor("Druhý")
54 class V2 { public static void proveď() { P.p(); } }
55
56 @Autor("Třetí")
57 class V3 { public static void proveď() { P.p(); } }
```

Spuštěním tohoto testu bychom mohli obdržet např. následující výpis:

```
 1 Výpis volání metod a jejich autorů kvůli chybě:
 2 java.lang.RuntimeException: Zásoba vyčerpána
 3   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:40)
 4   Autor=Nultý,  Zpráva=rup.česky.java15.anotace.V0.proveď(TestAutorůTříd.java:51)
 5   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:42)
 6   Autor=Druhý,  Zpráva=rup.česky.java15.anotace.V2.proveď(TestAutorůTříd.java:57)
 7   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:44)
 8   Autor=Třetí,  Zpráva=rup.česky.java15.anotace.V3.proveď(TestAutorůTříd.java:60)
 9   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:45)
10   Autor=První,  Zpráva=rup.česky.java15.anotace.V1.proveď(TestAutorůTříd.java:54)
11   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:43)
12   Autor=Třetí,  Zpráva=rup.česky.java15.anotace.V3.proveď(TestAutorůTříd.java:60)
13   Autor=Vojta,  Zpráva=rup.česky.java15.anotace.P.p(TestAutorůTříd.java:45)
14   Autor=???,  práva=rup.česky.java15.anotace.TestAutorůTříd.test(TestAutorůTříd.java:11)
15   Autor=???,  práva=rup.česky.java15.anotace.TestAutorůTříd.main(TestAutorůTříd.java:26)
```

Práce s anotacemi mimo běh programu

Sebekriticky přiznám, že název této kapitoly je jenom „namlsávací", protože se jedná o příliš speciální problematiku, abychom mohli očekávat, že ji bude při své práci po-

třebovat rozumné procento čtenářů. Většinou se totiž bude jednat o tvůrce různých vývojových nástrojů a přiznejme si, ti jsou mezi programátory v menšině. Omezím se proto pouze na velice stručný přehled.

Nástroj apt pro zpracování anotací ve zdrojovém kódu

Pro zpracování zdrojového kódu tříd s anotacemi je v JDK připraven program `apt`, který je klasickým programem ovládaným z příkazového řádku. Po spuštění vyhledá zadaný anotační procesor, jemuž poskytne API, jehož prostřednictvím tento procesor zpracuje zadané zdrojové soubory a provede potřebné akce – např. vytvoří dodatečné potřebné soubory, připraví konfigurační soubory pro instalaci (deployment descriptor) apod.

Součástí JDK je podrobná dokumentace k tomuto nástroji i úvodní tutoriál, s jehož pomocí se naučíte nástroj používat.

Zpracování bajtkódu

Vedle anotací, s nimiž je možno pracovat za běhu programu a anotací, které jsou dostupné pouze ve zdrojovém kódu, existuje třetí skupina anotací: anotace, které překladač umístí do vytvořeného class-souboru, avšak nejsou použitelné za běhu. Takovéto anotace mohou najít své uplatnění např. při instalaci knihoven.

Nemáte-li k dispozici zdrojový kód, musíte pracovat přímo s výslednými class-soubory. Jejich formát je sice podrobně popsán a lze jej nalézt např. na `http://java.sun.com/docs/books/vmspec`, ale je poměrně složitý a není jednoduché s ním pracovat bez pomoci speciálních knihoven.

Takové knihovny naštěstí existují. Jednou z nich je BCPL (Bytecode Engineering Library), kterou lze získat na `http://jakarta.apache.org/bcel`. Zde můžete stáhnout její kód i dokumentaci.

Kapitola 8

Rozšíření znakové sady

V této kapitole se seznámíme s posledním syntaktickým rozšířením „páté Javy", kterým je podpora nové verze znakové sady *Unicode*. Ta zavedla nové rozšíření znakové sady, které již vyžaduje kódování znaků třemi bajty. Nejprve se s novou znakovou sadou trochu seznámíme, a pak si ukážeme, jak je třeba kódovat rozšířenou sadu znaků v řetězcích i při zadávání samostatných znaků.

Programy:

Veškerý kód, který je v této kapitole uveden, je definován ve třídě `rup.česky.java15.char24.Znaky`.

Java se od samého začátku odlišovala od většiny ostatních programovacích jazyků tím, že neomezovala programátory pouze na základní sedmibitovou sadu znaků ASCII, ale definovala jako základní znakovou sadu pro zpracovávané řetězce i pro zdrojové programy sadu *Unicode*. Ta definuje kódování pro všechny znaky potřebné k tomu, aby bylo možno psát texty v kterémkoliv jazyku včetně čínštiny, japonštiny a korejštiny. Navíc definuje i kódování řady dalších všeobecně používaných znaků, mezi něž patří matematické symboly, notové, kartografické, architektonické a jiné značky, různé obrázkové znaky (např. „usměváčci") a řada dalších často používaných znaků.

Znaková sada Unicode

Než se začneme podrobněji zabývat implementací znakové sady *Unicode* v jazyku Java, povězme si nejprve něco z její historie a seznamme se s její koncepcí.

V sedmdesátých letech minulého století se začalo ukazovat, že rostoucí nasazení počítačů v nejrůznějších oblastech si během krátké doby vynutí odstranění omezení znakových sad a používání plnohodnotných národních abeced.

V první etapě byly zavedeny lokalizované znakové sady pro jednotlivé jazykové oblasti. To však způsobovalo problémy při tvorbě textů, v nichž bylo použito několik jazyků současně.

Tento problém měla odstranit znaková sada *Unicode*, která měla místo doposud používaného 8bitového kódování zavést 16bitové kódování znaků, které mělo se svými 65 536 kombinacemi umožnit zahrnout kódy pro znaky všech národních abeced současně a ponechat dokonce i místo pro případná další rozšíření.

Práce na definici sady začaly v roce 1980 a v roce 1991 byla zveřejněna definice verze 1.0, která vyhradila pro různé znaky zhruba polovinu dostupných pozic. Tato sada byla v jazyku Java použita jako výchozí znaková sada pro všechny textové řetězce včetně textových řetězců použitých při tvorbě zdrojových textů.

V průběhu dalších let se ale stalo něco, co při původní definici znakové sady nikdo neočekával: začaly se objevovat další a další skupiny znaků, které bylo třeba do kódovací tabulky zanést, a zanedlouho jejich množství přesáhlo oněch 65 536 možných kombinací.

Verze *Unicode 4.0* proto rozšířila kódovací prostor a místo dosavadního dvoubajtového (16bitového) kódování zavedla kódování tříbajtové, přičemž se z možných 24 bitů používá spodních 21, takže kódy znaků mohou nabývat hodnot 0 až 0x10FFFF („česky" 1 114 111). Současně definovala i kódovací algoritmus UTF-16, který popisuje, jak lze všechny znaky zakódovat pomocí dvoubajtového, tj. 16bitového kódování.

Než si začneme povídat o možnosti kódovaní všech znaků nové sady *Unicode 4.0* pomocí 16 bitů, musíme si nejprve zavést některé pojmy:

Kód znaku (code point)

Kódem znaku označujeme kódovou hodnotu přiřazenou danému znaku. Jak jsem již řekl, kód znaku může nabývat hodnot od 0 do 0x10FFFF, tj. do 1 114 111.

Kódovací stránka (code plane)

Kódy znaků jsou rozděleny do 17 **kódových stránek** po 65 536 potenciálních znacích (stránky nejsou zcela zaplněné). První stránka označovaná jako **základní multijazyková stránka** (*basic multilingual plane* – **BMP**) obsahuje kódy znaků podle původní definice *Unicode*, tj. znaky s kódy 0 až 0xFFFF.

Šestnáct zbylých stránek obsahuje tzv. **doplňkové znaky** (*supplementary characters*), což jsou znaky s kódy od 0x10000 do 0x10FFFF.

Kódování UTF-16 a kódovací jednotka (code unit)

Kódování **UTF-16** popisuje algoritmus, jak zakódovat všechny znaky pomocí 16bitových čísel – **kódovacích jednotek**. Znaky z BMP, tj. znaky s kódy 0 až 0xFFFF,

se kódují přímo prostřednictvím svého kódu. Pro zakódování každého z těchto znaků proto stačí jedna kódovací jednotka, která obsahuje přímo převáděný kód.

Pro kódování doplňkových znaků se využívá toho, že v BMP není zaplněna 2048 bajtů velká oblast s kódy od 0xD800 (binárně 1101 1000 0000 0000) do 0xDFFF (binárně 1101 1111 1111 1111), kterou označujeme jako **rozšiřující oblast** (*surrogates area*). Kódy z této oblasti jsou proto použity k zakódování doplňkových znaků pomocí dvojice kódových jednotek.

Doplňkové znaky jsou zakódovány tak, že se nejprve uloží kódovací jednotka s kódem vytvořeným jako součet čísla 0xD800 a horních 10 bitů převáděného kódu (*high-surrogate*) a za ní pak druhá kódovací jednotka s kódem vytvořeným jako součet čísla 0xDC00 a spodních 11 bitů převáděného kódu (*low-surrogate*).

Výhodou tohoto algoritmu je to, že z obsahu kódovací jednotky lze okamžitě zjistit, zda se jedná přímo o kód znaku z BMP, nebo zda se jedná o první, resp. druhou z dvojice kódovacích jednotek, obsahujících kód rozšiřujícího znaku.

Kdybyste si chtěli kódovaní UTF-16 podrobněji vyzkoušet a osahat, zkuste spustit následující program:

```
 1 public class Znaky
 2 {
 3   public static final int TST_MIN = 0x010000;
 4   public static final int TST_MAX = 0x10FFFF;
 5
 6   public static void převod_UTF16()
 7   {
 8     System.out.print('\f');
 9     for( int cp=TST_MIN, ii=1;   cp <= TST_MAX;   cp += ii, ii <<= 1 )
10     {
11       System.out.print( Integer.toHexString( cp ) );
12       char[] cu = Character.toChars( cp );
13       for( int i=0;   i < cu.length;   i++ )
14         System.out.print( " - " +  Integer.toHexString( cu[i] ) );
15       System.out.println();
16     }
17     System.out.println("Znak \uD835\uDD6B označuje množinu celých čísel");
18   }
19 }
```

Tento program po svém spuštění vypíše na standardní výstup následující sekvenci:

```
1 10000 - d800 - dc00
2 10001 - d800 - dc01
3 10003 - d800 - dc03
4 10007 - d800 - dc07
5 1000f - d800 - dc0f
6 1001f - d800 - dc1f
7 1003f - d800 - dc3f
```

```
 8 1007f - d800 - dc7f
 9 100ff - d800 - dcff
10 101ff - d800 - ddff
11 103ff - d800 - dfff
12 107ff - d801 - dfff
13 10fff - d803 - dfff
14 11fff - d807 - dfff
15 13fff - d80f - dfff
16 17fff - d81f - dfff
17 1ffff - d83f - dfff
18 2ffff - d87f - dfff
19 4ffff - d8ff - dfff
20 8ffff - d9ff - dfff
21 10ffff - dbff - dfff
22 Znak ? představuje množinu celých čísel
```

Jak vidíte na řádku 22, nestačí pouze zadat správný kód znaku, ale musíte mít také instalovanou znakovou sadu, která umí příslušný znak vytisknout. Jinak se vám místo příslušného znaku vytiskne otazník, obdélníček či jiný znak zastupující znaky, jež dané písmo neobsahuje.

Práce se znaky

Jak jsem již řekl, nová verze Javy podporuje práci s úplnou množinou znaků *Unicode 4.0*. Znakový typ `char` však zůstává z důvodu kompatibility i nadále 16bitový, takže do proměnných typu `char` není možno doplňkové znaky uložit.

Chceme-li uložit kód jednoho konkrétního doplňkového znaku, musíme použít proměnnou typu `int`, anebo pole obsahující dvojici proměnných typu `char`, v nichž jsou uloženy jednotlivé kódovací jednotky vytvořené podle algoritmu UTF-16.

Při práci s řetězci ale podobné problémy nenastanou, protože do nich se doplňkové znaky ukládají automaticky prostřednictvím UTF-16. Chceme-li však řetězec, který by mohl obsahovat doplňkové znaky, procházet znak za znakem, musíme použít některé nové metody, které nám to umožní. Této problematice je věnována následující podkapitola.

Znaky jako parametry a návratové hodnoty

Metody, jejichž parametr má typ `char`, proto mohou i nadále zpracovávat pouze znaky z BMP. S kompletní sadou znaků mohou pracovat pouze metody, které sice podle kontraktu očekávají v parametru kód znaku, avšak používají parametr typu `int`.

Obdobné omezení platí pro metody vracející hodnotu typu `char` – mohou vracet pouze znaky z BMP. Bez omezení, tj. s úplnou sadou znaků, mohou pracovat pouze metody, které sice také vracejí kód znaku, ale vracejí jej jako hodnotu typu `int`.

Pro překonání problémů s metodami pracujícími s parametry, resp. návratovými hodnotami typu `char`, byly některé třídy, především pak třída `Character`, v nové verzi do-

plněny o rozšiřující sadu metod. Ty jsou ekvivalentní stávajícím metodám, avšak místo parametru typu `char` používají parametr typu `int`, resp. místo hodnoty typu `char` vracejí hodnotu typu `int`.

Třída `Character` byla rozšířena o množství různých metod. Z nich bych si pro tuto chvíli dovolil vypíchnout následující dvě:

public static int codePointAt(char[] a, int index)

Vrací kód znaku na zadané pozici v zadaném vektoru znaků. Najde-li na dané pozici první kódovací jednotku kódující horní bity doplňkového znaku následovanou kódovací jednotkou kódující jeho spodní bity, sestaví a vrátí příslušný kód. V opačném případě vrátí kód znaku na dané pozici.

public static int codePointCount(char[] a, int offset, int count)

Vrátí počet *Unicode* znaků obsažených v zadané části zadaného vektoru znaků. Bude-li zadaná část obsahovat některou z kódovacích jednotek kódujících doplňkové znaky bez jejího protějšku, bude tato jednotka počítána za jeden znak.

Práce s řetězci

Jak jsme si již řekli, znaky, které nespadají do základní znakové sady, se v řetězcích zadávají prostřednictvím dvojice odpovídajících kódovacích jednotek, přičemž kód první jednotky musí být v rozsahu 0xD800 až 0xDBFF a kód druhé musí být v rozsahu 0xDC00 až 0xDFFF, např.:

```
String u = "Znak \uD835\uDD6B představuje množinu celých čísel";
```

Zeptáte-li se však řetězce na jeho délku, neoznámí vám počet znaků, ale počet kódovacích jednotek. Budete-li očekávat, že by řetězec mohl obsahovat doplňkové znaky, budete se muset na počet znaků v řetězci zeptat prostřednictvím metody

```
public int codePointCount(int beginIndex, int endIndex)
```

které v jejích parametrech předáte index první a „poposlední" kódovací jednotky. Počet znaků v celém řetězci `sss` tak zjistíte zadáním příkazu:

```
int znaků = sss.codePointCount( 0, sss.length() );
```

Obdobně si musíte dát pozor při zjišťování znaku na dané pozici. Klasická metoda `charAt()` totiž vrací pouze kódovací jednotky, takže chcete-li z řetězců obsahujících doplňkové znaky získat kód nějakého znaku, musíte použít metodu:

```
public int codePointAt( int index )
```

Protože však zadávaný index je indexem první kódovací jednotky daného znaku, musíte nejprve zjistit, jaký má první kódovací jednotka daného znaku index. K tomu slouží metoda

```
public int offsetByCodePoints(int index, int codePointOffset)
```

která vrací index první kódovací jednotky znaku, který je `codePointOffset`-tým znakem za zadaným indexem.

Kdybyste tedy chtěli definovat metodu, která vrátí celočíselný vektor obsahující kódy jednotlivých znaků v řetězci, museli byste ji definovat např. následovně:

```
 1 public static int[] getKódyZnaků( String s )
 2 {
 3   int znaků = s.codePointCount( 0, s.length() );
 4   int[] kód = new int[znaků];
 5   for( int i=0, znak=0;   i < s.length();   )
 6   {
 7     kód[znak] = s.codePointAt( i );
 8     i += Character.isSupplementaryCodePoint(kód[znak])
 9        ?  2
10        :  1;
11     znak++;
12   }
13   return kód;
14 }
```

Rozdílnost zpracování klasických a doplňkových znaků se obdobným způsobem projevuje i v ostatních operacích s textovými řetězci. Ty zde však již nebudu podrobněji rozebírat. Pokud byste chtěli práci s úplnou sadou implementovat, jistě si požadované postupy odvodíte sami.

Shrnutí

Budete-li nadále používat pouze znaky z BMP (tj. vystačíte-li s oněmi více než 60 000 znaky), nemusíte své programy nijak měnit. Rozhodnete-li se však vybavit své třídy schopností pracovat i s doplňkovými znaky, budete muset metody operující se znaky upravit tak, aby na doplňkové znaky dokázaly správně reagovat.

Jak bylo ukázáno, komfort práce s řetězci i efektivita výsledných programů při používání doplňkových znaků rychle klesá. Doporučuji vám proto implementovat zpracování rozšířené sady *Unicode* pouze v případech, kdy ocenění této možnosti uživatelem bude úměrné vynaložené námaze.

Kapitola 9

Přechod na verzi 5.0

Tato kapitola obsahuje výtah z dokumentu *Java™ Platform Migration Guide Version 1.3 to 5.0*, jehož plnou verzi si můžete stáhnout na adrese `http://java.sun.com/j2se/JM_White_Paper_R6A.pdf`. Dozvíte se v ní, jaké kroky musíte podniknout, aby vaše programy napsané pro verzi 1.4 optimálně spolupracovaly s novými programy a běžely na nových virtuálních strojích. Uvedený dokument se zmiňuje i o změnách vůči verzi 1.3 a zmiňuje proto o novinky zavedené verzí 1.4. Ty já však v této kapitole pomíjím a omezuji se na novinky, s nimiž přišla verze 5.0. Kdo proto převádí program původně napsaný a provozovaný na platformě 1.3, měl by si zmíněný dokument přečíst celý.

Běh programu

Vlastnosti jazyka a knihovny zmíněné v této podkapitole ovlivňují běh programů vytvořených v předchozích verzích Javy při spuštění na nových verzích virtuálního stroje.

AWT

Knihovna AWT byla upravena tak, aby se zdokonalila její platformní nezávislost a aby se zlepšila její spolupráce s lehkými komponentami GUI.

- Byl znemožněn požadavek předání fokusu „nefokusovatelné“ komponentě. Dřívější verze tuto operaci povolovaly.
- Není-li okno viditelné, metody `Window.toFront()` a `Windos.toBack()` nic neudělají. V předchozích verzích bylo chování těchto metod platformně závislé.

- Kontejnery mohou definovat přechodovou politiku fokusů (focus traversal policy). O tom, jestli ji opravdu definují, informuje ve třídě `Container` vlastnost `FocusTravesalPolicyProvider`. Dříve mohly svoji politiku nabízet pouze kořenové kontejnery cyklů (focus cycle roots). Verze 5.0 tuto zásadu změnila.
- V systému *Windows* přepíná metoda `Windows.toBack()` zaměřené (focused) okno na vrchní okno.
- V systému *Windows* nyní metoda `request Focus()` umožňuje přepínání mezi okny každému. Dříve to umožňovala pouze těžkým (heavy weight) komponentám.
- Na platformách *Linux* a *Solaris* bylo AWT reimplementováno. Nová implementace nabízí následující výhody:
  - Odstraňuje závislost na knihovnách *Motif* a *Xt*.
  - Lépe spolupracuje s ostatními „GUI Toolkits".
  - Nabízí vyšší výkon a kvalitu.

  Nový *XToolkit* je na systému *Linux* nastaven jako implicitní. Na systému *Solaris* je sice prozatím stále implicitním *MToolkit*, ale *Sun* slibuje, že v budoucnosti bude nahrazen sadou *XToolkit*.

  Tento toolkit můžete explicitně nastavit pro aplety a aplikace pomocí systémových proměnných. Podrobnosti (konkrétní nastavení pro konkrétní systémy) najdete ve výše zmíněném dokumentu.
- Na X11 (*Linux*, *Solaris*) byl pro operaci táhni-a-pusť (drag and drop) podporován pouze protokol *Motif*. Nyní byla tato podpora přeprogramována tak, aby nezávisela na knihovně *Motif* a navíc je podporován i protokol XDND.

Java 2D Graphics

- Změnily se informace o metrice písem. Pro programy, které používají znaky z tabulky ANSI, je změna malá, ale kumuluje se s rostoucím počtem komponent. U asijských písem se odchylky projeví mnohem rychleji.
- V předchozích verzích způsobilo předání prázdného odkazu na obrázek (`Image`) metodě `Graphic.drawImage()` vyhození výjimky `NullPointerException`. Nyní je již metoda schopna převzetí hodnoty `null` úspěšně přežít.
- Java 5.0 již podporuje akcelerované zobrazování kopií obrázků bez ohledu na to, jak byly obrázky vytvořeny. Aplikace nyní automaticky využívají výhod akcelerace bez ohledu na to, zda byly obrázky vytvořeny prostřednictvím `VolatileImage`, `Component.createImage()` nebo zda byly vytvořeny „manuálně" prostřednictvím `new BufferedImage()`. Použití akcelerace nezávisí na tom, jak je akcelerace dosaženo.
- Od verze 5.0 může být Java2D spouštěna pomocí *OpenGL*, takže aplikace mohou díky využití hardwarové akcelerace dosahovat při řadě speciálních operací

vysokých výkonů. *OpenGL* je podporováno na všech platformách zabezpečovaných firmou *Sun* (*Windows*, *Linux*, *Solaris*).

Vzhledem k nekonzistenci ovladačů není možno zabezpečit tuto podporu pro všechny platformy. Budete-li chtít mít na používané platformě zapnutu podporu *OpenGL*, zadejte na příkazovém řádku příznak:

```
-Dsun.java2d.opengl=true
```

Počítejte však s tím, že výkon a robustnost velmi závisí na použitém hardwaru. *OpenGL* byste proto měli povolovat pouze po pečlivém otestování platformy, abyste pak nebyli udiveni dosaženými výsledky.

Bezpečnost

- Byla přidána třída `Java.security.KeyRep`, jejíž instance reprezentují serializovatelné šifrovací klíče. Současné serializovatelné klíče jsou serializovatelné a deserializovatelné pouze v rámci virtuálního stroje jednoho dodavatele, avšak při přechodu mezi virtuálními stroji různých dodavatelů není jejich funkčnost zaručena. To by se nyní mělo změnit.

  Nové implementace klíčů, které používají pro svou serializovanou reprezentaci instancí `KeyRep`, bude nyní možno serializovat na virtuálním stroji jednoho dodavatele a deserializovat na jiném. Instance, které tuto reprezentaci využijí, však nemohou být deserializovány na žádném stroji, který nepodporuje verzi 5.0 či mladší.

- Třída `javax.security.auth.kerberos.KerberosKey` nyní definuje své vlastní soukromé `serialVersionUID`. Toto nové pole nyní zakrývá pole zděděné z rozhraní `java.security.Key`, jež `KerberosKey` implementuje.

  Změna nepřináší žádné problémy při běhu stávajících aplikací. Dříve přeložené třídy, jež se odkazují na pole `KerberosKey.serialVersionUID`, pracují korektně. Tato změna způsobuje pouze nekompatibilitu ve zdrojových kódech, protože třídy, které se na dané pole odkazují, nebude nyní možno přeložit, dokud nebude jejich kód upraven.

Serializace

- Mezi verzemi 1.3 a 5.0 se změnila spočtená hodnota UID serializované verze. Ve verzi 5.0 vkládá odkaz na literál třídy (např. `String.class`) do bajtkódu instrukci, kdežto dříve vkládal odkaz na statickou metodu. Důsledkem této změny je, že instance jakékoliv serializovatelné třídy, jež se odkazuje na literál třídy, nebude pod novým virtuálním strojem pracovat.

  Chcete-li odstínit svůj kód od změn, kterým podléhá překladač při přechodu mezi verzemi (případně mezi virtuálními stroji různých dodavatelů), definujte UID svých verzí explicitně (můžete využít `serialvertool`).

Swing

- Byla opravena nekonzistence mezi chováním komponenty `JTable` a komponent `JList` a `JTree`.

Další změny

- Nová verze virtuálního stroje umožňuje sdílení tříd. Proto tuto informaci přidala do vlastnosti `Java.vm.info`, která je zobrazována po zadání příkazu

  `java -version`.
- Součástí JDK se stala nová verze knihovny z JAXP (přešlo se z 1.1 na 1.3), která přináší výrazné zvýšení funkčnosti.
- V JDBC změnila své chování metoda `BigDecimal`. Tento problém lze vyřešit instalací ovladače, který podporuje verzi 5.0.
- Od verze 5.0 přestalo být možné porovnávat instance `java.sql.TimeStamp` a `java.util.date`. Nyní takovýto pokus vede na `ClassCastException`. Předpokládá se, že problém bude vyřešen v příštích verzích.
- Dříve bylo možno zadat konstruktoru `java.util.logging.Level(String name, int value, String resourceBundleName)` prázdný odkaz jako hodnoty parametru `name`. Nyní to již nedovoluje, a pokud se o to pokusíte, vyhodí výjimku `NullPointerException`.

Instalace (deployment)

Aplety

- Byl zaveden nový řídící panel, jenž sloučil *Java Plug-in Control Panel* a *Java Web Start Application Manager* a poskytl tak jednotné instalační prostředí.
- Od verze 5.0 mají aplety nezávislou rychlou zápisníkovou paměť (cache), kterou sdílejí nezávisle na použitém prohlížeči. Minulé verze zřizovaly vlastní zápisníkovou paměť pro každý prohlížeč, takže nebylo vyloučeno, že každý prohlížeč měl staženu svoji verzi apletu.

  Díky této změně přebírá řídící panel od prohlížeče řadu úkolů. Sám se nyní stará o:

  - umístění paměti,
  - omezení její velikosti,
  - její komprimaci,
  - prostředky pro její správu,
  - politiku odstraňování starších a méně používaných záznamů.
- Certifikáty certifikační autority (CA) používané pro ověřování podpisů přicházejí:
  - z „cacerts souborů“ běhového prostředí (JRE) – vždy povoleno,

  - z úložiště prohlížeče – implicitně povoleno, ale je možno zakázat v řídícím panelu.

  Předchozí verze se omezovaly na certifikáty získané z prohlížeče.

- Instalovaný archiv již nemusí být každoročně znovu podepisován. Místo toho může program `jarsigner` vygenerovat podpis obsahující časové razítko. Systémové a instalační programy si pak mohou kdykoliv zkontrolovat, zda časový údaj v razítku pochází z doby, kdy byla příslušná certifikační autorita platná.

Knihovny

- Integrální součástí standardní knihovny jsou nyní následující specializované knihovny (tj. nejsou již distribuovány jako volitelné části):
  - JSSE – Java™ Secure Socket Extension
  - JAAS – Java™ Authentication and Authorization Service
  - JCE – Java™ Cryptography Extension
  - JWS – Java™ Web Start
  - JMX™ – Java™ Management Extensions

  Pro detekci případných konfliktů mezi cílovým jar-souborem a instalovanými rozšiřujícími jar-soubory můžete použít program `extcheck`, jenž je nyní součástí JDK.

Instalační programy

- Byla zavedena možnost on-line instalace. Její instalační program je poměrně krátký (cca 200 KB), takže se rychle natáhne. Po svém spuštění si zjistí konfiguraci počítače, na němž běží, a stahuje již jen tu část kompletního instalačního souboru, kterou doopravdy potřebuje.

  Uživatelům s pomalejším připojením asi bude vadit, že neindikuje, jak dlouho bude ještě instalace trvat a ani příliš podrobně nehlásí, jak je s instalací daleko. Proto tito uživatelé občas dávají přednost úplné, off-line instalaci, i když se při ní musí stáhnout větší množství dat.

- Nová verze změnila řadu názvů složek, balíčků, registrů a další objektů, takže je třeba aktualizovat skripty, které s těmito názvy pracují. Nové jmenné konvence jsou následující:

Starý název	Nový název
`j2se`	`java`
`j2re`	`jre`
`j2sdk`	`jdk`

  Použití velkých písmen odpovídá zvykům každé platformy.

Překlad

Změny v API

- Byla přidána třída `java.net.Proxy`, takže nyní standardní knihovna obsahuje dvě třídy `Proxy`:
  - `java.lang.reflect.Proxy`
  - `java.net.Proxy`

  V nové verzi proto nebudete moci přeložit soubor, pro který bude platit:
  - obsahuje deklarace importů

    ```
    import java.lang.reflect.*;
    import java.net.*
    ```

  - neobsahuje deklaraci importu, která by importovala právě jednu z proxy tříd,
  - kód se obrací na třídu `Proxy` jejím prostým názvem bez uvedení balíčku.

  Při splnění předchozích podmínek je totiž nejasné, kterou z tříd má programátor na mysli, takže překladač ohlásí syntaktickou chybu. Tento problém můžete odstranit např. tak, že mezi importy přidáte deklaraci:

  ```
  import java.lang.reflect.Proxy
  ```

  Pak se bude zdrojový kód chovat naprosto stejně, jako se choval doposud.

Parametrizované typy

- Kontejnerové třídy, třída `Class` a další třídy ze standardní knihovny jsou nyní definovány s použitím typových parametrů. Překlad většiny dosavadního kódu tato změna neovlivní. S překladem některých tříd však můžete mít problémy. Nejjednodušším řešením je přidat do příkazového řádku překladače parametr

  ```
  -source 1.4
  ```

- Mnohé ze současných IDE nabízejí mezi svými refaktorovacími funkcemi i možnost automaticky převést stávající zdrojový kód do tvaru využívajícího typové parametry a parametrizované typy a metody.

Nová klíčová slova

- Nová verze přidala jediné nové klíčové slovo: `enum`. To by mohlo způsobit problémy pouze v případě, že program toto slovo využívá jako identifikátor. Nepotřebujete-li v daném programu využívat nových možností jazyka a chcete-li zachovat použití tohoto identifikátoru (to vám ale nedoporučuji), můžete vše vyřešit opět přidáním parametru překladače

  ```
  -source 1.4
  ```

Změny ovlivňující používané nástroje

Class-soubory, vnitřní třídy

- Změnil se formát class-souborů. Programy, které (např. v zájmu zvýšení efektivity) class-soubory upravují, proto mohou vytvářet neplatné výsledné soubory.
- Změnily se také vytvářené názvy vnitřních tříd, takže programy, které vycházejí ze starších schémat, nemusejí nově generované vnořené a vnitřní třídy rozpoznat.

Inicializace tříd po vyhodnocení literálu X.class

- Vyhodnocení literálu třídy (např. `String.class`) nyní nemusí nutně vést k inicializaci třídy, což dříve vedlo.

  Nové chování je důsledkem skutečnosti, že virtuální stroj nyní uchovává literály tříd v prostoru konstant (constant pool). Třídy, které byly přeloženy staršími překladači nebo které byly přeloženy s parametrem

  ```
  -source 1.4
  ```

  se však chovají stejně jako dříve, a to i tehdy, běží-li na virtuálním stroji Javy 5.0.

  Kód, jenž závisí na dříve definovaném chování, musí být upraven následovně:

```
 1 //... Něco.class ...                //Starý kód
 2 //... inicializuj(Něco.class) ...   //Nový kód
 3
 4 /**
 5  * Zajistí inicializaci třídy nalezením zadaného <tt>Class</tt>
 6  * objektu. Je-li třída již inicializována, neudělá metoda nic.
 7  *
 8  * @param  třídaTřída, jejíž inicializace je požadována
 9  * @return <tt>třída</tt>
10 */
11 public static <T>Class<T> inicializuj(Class<T> třída) {
12   try {
13     Class.forName( třída.getName(), true,
14                    třída.getClassLoader());
15   } catch (ClassNotFoundException e) {
16     throw new AssertionError(e);   //Nemůže nastat
17   }
18   return třída;
19 }
```

Bližší informace lze nalézt ve specifikaci jazyka v podkapitole *12.4 Initialization of Classes and Interfaces.*

Poznámka:

Specifikace jazyka se nezměnila. Ve specifikaci nikdy nebyl uveden literál třídy jako spouštěč její inicializace. Toto chování byla čistá iniciativa implementátorů virtuálního stroje.

Parametry metod zavaděče tříd

- Dříve bylo možno metodám zavaděče tříd, jež vyžadovaly zadání názvu třídy jako textového řetězce, zadat nekorektní název třídy (v originálu non-binary name – termín *binary name* viz *Java Language Specification,* podkapitola *13.1 The Form of a Binary*). Nyní tyto metody korektnost názvů tříd kontrolují.

API pro ladění a profilaci

- Nová verze zařadila JVMDI (Java Virtual Machine Debug Interface) mezi zavržené (deprecated). **Do příští verze JDK již nebude zařazen!** Všechny nové vývojové nástroje nyní musí používat JVMTI (Java Virtual Machine Tool Interface).
- Nová verze zařadila JVMPI (Java Virtual Machine Profiling Interface) mezi zavržené. **Do příští verze JDK již nebude zařazen!** Všechny nové vývojové nástroje nyní musí používat JVMTI (Java Virtual Machine Tool Interface).

Příloha A

Varovná hlášení překladače

Jak jsem řekl již v úvodu, v prvních několika kapitolách jsou použity některé konstrukce, které již podle nové verze nejsou úplně bezpečné. Překladač proto při jejich výskytu vydává varovná hlášení. Dokud se však neseznámíte s parametrizovanými datovými typy, tak stejně nebudete mít dostatek informací k tomu, abyste upravili program do podoby, s níž bude překladač spokojen.

Implicitně je překladač nastaven tak, že vás pouze upozorňuje, že kdybyste mu to dovolili, tak by vám měl co říci. Na konci překladu tříd, v nichž našel některé podle něj nedokonalé konstrukce, vypíše zprávu typu:

```
Note: Q:\Projekty\src\rup\Program.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Této zprávy se neděste. Překladači se totiž v nové verzi přestala líbit řada konstrukcí, které byly dosud běžně používané. V kapitole *Parametrizované datové typy a metody, typové parametry* na straně 39 si vysvětlujeme, jak tyto konstrukce upravit, aby se začaly překladači opět líbit. V pasáži *Spolupráce s programy vytvořenými v předchozích verzích* na straně 68 pak ukazuji, jaké konstrukce se překladači nelíbí a jaká hlášení při jejich odhalení vydává.

Budete-li chtít, aby vám překladač podrobně referoval o každém svém podezření, musíte do příkazového řádku vložit parametr:

```
-Xlint
```

Postačí-li vám, bude-li vás překladač varovat pouze tehdy, nepoužijete-li typové parametry v místech, kde byste měli, zadejte:

```
-Xlint:unchecked
```

Pokud vás však nejrůznější varovná hlášení neustále ruší a rádi byste je omezili jenom na souhrnnou zprávu, že překladač nepovažuje kód za úplně dokonalý, zadejte parametr v podobě:

```
-Xlint:none
```

Nastavení v prostředí *NetBeans*

Novou verzi Javy podporuje vývojové prostředí *NetBeans* od verze 4.0. (Programy sice bylo možno bez problémů překládat i ve starších verzích, nicméně jejich editor občas označoval jako chybné i příkazy, které podle nové verze Javy chybné nebyly a překladač je bez problému přeložil.)

V prostředí *NetBeans* se zadávají parametry příkazového řádku překladače následovně:

1. Příkazem **Windows → Files** otevřete panel (nebo okno – záleží na používaném režimu) s projekty a v něm kartu se stromovou strukturou použitých složek.
2. Otevřete složku daného projektu a v ní podsložku `nbproject`.
3. Otevřete soubor `project.properties`.
4. Najdete v něm řádek s parametrem `javac.compilerargs` a zadáte do něj požadované parametry – např.:

   ```
   javac.compilerargs=-Xlint
   ```

5. Soubor uložíte a zavřete.

Nastavení v prostředí *BlueJ*

Javu 5.0 podporuje vývojové prostředí *BlueJ* od verze 2.0. V tomto prostředí přidáte požadovaný parametr do příkazového řádku překladače následovně:

1. Máte-li spuštěné prostředí *BlueJ*, zavřete je.
2. Otevřete složku, do které jste *BlueJ* instalovali, a v ní podsložku `lib`.
3. Ve složce najdete a otevřete soubor `bluej.defs`.
4. Najdete v něm řádek s parametrem `bluej.compiler.options` a k případným parametrům přidáte požadovaný. Pokud tam takový řádek nebude, můžete jej přidat. Budete-li nastavovat zobrazování všech varovných hlášení, bude mít řádek tvar:

   ```
   bluej.compiler.options=-Xlint
   ```

5. Uložíte a zavřete soubor a spustíte znovu *BlueJ*. Pošle-li vám překladač nějakou varovnou zprávu, prostředí po úspěšně skončeném překladu zobrazí všechny případné varovné zprávy ve společném dialogovém okně.

Nastavení v prostředí *Eclipse*

Vývojové prostředí *Eclipse* podporuje Javu 5.0 až od verze 3.1. Používá vlastní překladač, jehož parametry se nastavují poněkud odlišně než u prostředí využívajících překladač z JDK. Chybová a varovná hlášení tohoto překladače se nastavují následovně:

1. Chcete-li nastavit úroveň hlášení společnou pro všechny projekty, zadejte **Window → Preferences**.
2. Ve stromu preferencí vlevo postupně rozbalte **Java → Compiler** a zde klepněte na **Errors/Warnings**.
3. V pravém panelu se objeví seznam chybových oblastí vysazených tučně. Klepnutím na kteroukoliv z nich rozbalíte podseznam konkrétních problémů, u nichž můžete zadat, zda je má prostředí označovat jako chyby, varování nebo zda je má prostě ignorovat.
4. Problémy, které souvisí s novou verzí, najdete v posledním seznamu nazvaném **JDK 5.0 options**. Projděte si jej a u každé možnosti nastavte v rozbalovacím seznamu vpravo, jak má *Eclipse* na tento druh situace reagovat.

Rejstřík

@

@Deprecated, 114
@Documented, 115
@Inherited, 116
@Override, 114
@Retention, 116
@SuppressWarnings, 114
@Target, 117

A

adresa
  doprovodné programy, 8
  formát class-souborů, 132
  Java Platform Migration Guide, 139
  knihovna BCPL, 132
anotace, 109
  @Deprecated, 114
  @Documented, 115
  @Inherited, 116
  @Override, 114
  @Retention, 116
  @SuppressWarnings, 114
  @Target, 117
  balíčků, 113
  jako parametr, 123
  nástroje pro práci, 132
  označení deklarací, 110
  předdefinované, 113
  s parametry, 121
  syntaxe definice, 118
  získání hodnot parametrů, 128
  získání informací za běhu, 124
  značkovací, 119
autoboxing, 28
automatický převod
  omezení, 30
auto-unboxing, 28
autounwrapping, 28
autowrapping, 28

B

basic multilingual plane, 134
BlueJ
  nastavení překladače, 148
BMP, 134

C

C++
  šablona, 41
code plane, 134
code point, 134
code unit, 134
cyklus
  for each, 17
    kdy nelze, 20

Č

čištění typu, 63

D

doplňkový znak, 134
doprovodné programy, 8

E

Eclipse
  nastavení překladače, 149
Enum, 91
EnumMap, 107
EnumSet, 107

F

for each, 17
  kdy nelze, 20

G

garbage collector, 28
generická metoda, 53, viz parametrizovaná metoda
generické rozhraní, 49
generický typ. viz parametrizovaný typ, viz parametrizovaný typ

H

halda, 28
heap, 28
hlášení
  varovné, 9

I

import
  statický, 11
    kolize, 15
    sloučení importů, 13
iterátor
  skrytý, 22
iterovatelná třída, 23

J

Java
  verze, 7
javadoc, 109
jednorozměrné pole, 19
jednotka
  kódovací, 134

K

kód znaku, 134
kódovací jednotka, 134
kódovací stránka, 134
kolekce, 18
kolize importů, 15
konstanty
  výčtové, 89
    anonymních typů, 100
    jako klíče v mapě, 107
    množiny konstant, 107
kontejner, 17, 40
  statický, 18
kontejnerová třída, 40
kvalifikace, 11

M

metaanotace, 115
metadata. viz anotace
metoda
  generická, 53, viz parametrizovaná metoda
  parametrizovaná, 40, 53

přemosťovací, 65
s proměnným počtem parametrů, 33
metody
generické
volání, 55
parametrizované
volání, 55

N

nastavení překladače
BlueJ, 148
Eclipse, 149
NetBeans, 148
neozdobený typ, 63
NetBeans
nastavení překladače, 148

O

obalový typ, 28
objektový typ, 27
oblast
rozšiřující, 135
očištění typu, 63
očištěný typ, 63
očišťování, 62
ozdobený typ, 62

P

parametr
proměnný počet, 33
typový, 40
omezení, 57
žolík, 79
parametrizace, 62
parametrizovaná metoda, 40, 53
parametrizované rozhraní, 49
parametrizovaný typ, 39, 40, 50
definice, 46
parametry
typové
vzájemná závislost, 60
PM, 40
pole, 18
jednorozměrné, 19
vícerozměrné, 19
primitivní typ, 27
program
doprovodný, 8
proměnný počet parametrů, 33
překladač
nastavení
BlueJ, 148
Eclipse, 149
NetBeans, 148
varovné hlášení, 9
přemosťovací metoda, 65
přepravka, 37
převod
automatický
omezení, 30
PT, 40
PTM, 40

R

rozhraní
generické, 49
parametrizované, 49
rozšíření znakové sady, 133
rozšiřující oblast, 135

S

skrytý iterátor, 22
sloučení, 13
správce paměti, 28
statický import, 11
kolize, 15
sloučení importů, 13
statický kontejner, 18
stránka
kódovací, 134
surový typ, 63

T

třída
iterovatelná, 23
kontejnerové, 40
třída.Enum, 91
typ
čištění, 63
generický. viz parametrizovaný typ
neozdobený, 63
obalový, 28
objektový, 27
očištěný, 63
ozdobený, 62
parametrizovaný, 39, 40
dědičnost, 50
definice, 46
primitivní, 27
surový, 63
výčtový, 87
přepínač, 93, 96
přidané atributy a metody, 90
zástupný, 79
typ generický. viz typ parametrizovaný
typové parametry
vzájemná závislost, 60
typový
parametr
žolík, 79
typový parametr, 40
omezení, 57

U

Unicode, 134

V

varovné hlášení překladače, 9
vektor. viz jednorozměrné pole
vícerozměrné pole, 19
volání generických metod, 55
volání parametrizovaných metod, 55
výčtové konstanty, 89
anonymních typů, 100
jako klíče v mapě, 107
množiny konstant, 107
výčtový typ, 87
přepínač, 93, 96
přidané atributy a metody, 90

Z

zástupný typ, 79
značka, 119
znak
doplňkový, 134
kód znaku, 134
znaková sada
rozšíření, 133

Ž

žolík, 79
a dědičnost, 83