Bakalářská práce

Java 5.0 Tiger – Novinky a tutoriál

Miloslav Thon

vedoucí práce

RNDr. Jaroslav Icha

Bakalářská práce

Java 5.0 Tiger

Novinky a tutoriál



- ⇨ Generické typy
- ⇨ Statický import
- ⇨ Automatické zapouzdření primitivních typů
- ⇨ Cyklus for-each
- ⇨ Metody s proměnným počtem parametrů
- ⇨ Výčtové typy
- ⇨ Práce s kolekcemi v JDK 5.0
- ⇨ Formátování výstupu
- ⇨ Některé užitečné třídy v JDK 5.0
- ⇨ Anotace

JIHOČESKÁ UNIVERZITA
V ČECKÝCH BUDĚJOVICÍCH
PEDAGOGICKÁ FAKULTA
KATEDRA INFORMATIKY

Miloslav Thon
České Budějovice
2006

Knihovna JU - PF
3115172695

Obsah

Seznam příkladů ix

Seznam tabulek xi

Úvod xiii

Kde se vzalo označení Java 5.0 Tiger xiii
Forma zpracování práce xiii
Uspořádání textu xiv
Příklady k teoretické části xiv
Tutoriál – praktická část xiv
Přiložené CD xv
Typografické a syntaktické konvence xv
Poděkování xv

1 Generické typy 1

1.1 Představení generických typů 1
1.1.1 Shodneme se na pojmech? 3
1.1.2 Hlavní výhody použití generických typů 3
1.1.3 Příklad práce s generickým typem 3
1.1.4 Označení typových parametrů generické třídy 6
1.1.5 Zpětná kompatibilita 6
1.1.6 Šablony v C++ versus generické typy v Javě 6
1.2 Vytvoření vlastního generického typu 7
1.2.1 Deklarace generické třídy 7
1.2.1.1 Vytvoření objektu generické třídy 8
1.2.2 Deklarace rozhraní 9
1.2.3 Deklarace abstraktní generické třídy 9
1.3 Polymorfizmus 10
1.3.1 Zástupný symbol ? 11
1.4 Generické metody 12
1.4.1 Deklarace generické metody 12
1.4.2 Použití generické metody 13
1.5 Ohraničení typových parametrů 14
1.5.1 Ohraničení shora 15
1.5.2 Ohraničení zdola 17
1.5.3 Vícenásobné ohraničení 19
1.5.4 Jak číst některé složitější zápisy generických typů 20
1.5.4.1 Deklarace generické třídy 20
1.5.4.2 Deklarace generické metody 20

1.6 Pole a generické typy ... 21
1.6.1 Typem pole nemůže být generický typ ... 21
1.6.2 Typovým parametrem nemůžeme inicializovat pole ... 22
1.7 Dědičnost a generické typy ... 23
1.7.1 Třída vzniklá děděním od jiné třídy ... 24
1.7.1.1 Generická třída jako potomek běžné třídy ... 24
1.7.1.2 Generická třída jako potomek generické třídy ... 24
1.7.1.3 Běžná třída jako potomek generické třídy ... 25
1.7.2 Implementace rozhraní v generické třídě ... 25
1.7.2.1 Implementace více rozhraní v jedné třídě ... 26
1.7.3 Dědění třídy a současná implementace rozhraní ... 26
1.8 Překlad zdrojového kódu využívajícího generické typy ... 26
1.8.1 Některé nepovolené operace ... 27

2 Statický import ... 29
2.1 Použití statického importu ... 29
2.2 Možný konflikt při použití statického importu ... 31

3 Automatické zapouzdření primitivních typů ... 32
3.1 Použití automatického zapouzdření ... 32
3.2 Kdy nelze použít automatické zapouzdření ... 35

4 Cyklus for-each ... 36
4.1 Použití cyklu for-each pro procházení pole ... 36
4.1.1 Použití pro vícerozměrné pole ... 37
4.2 Použití cyklu for-each pro procházení kolekce ... 37
4.2.1 Rozhraní `Iterable<T>` ... 39
4.2.2 Použití pro průchod mapy ... 39

5 Metody s proměnným počtem parametrů ... 41
5.1 Pravidla pro deklaraci metody s proměnným počtem parametrů ... 41
5.2 Deklarace metody s proměnným počtem parametrů ... 41
5.3 Použití metody s proměnným počtem parametrů ... 42
5.4 Různé způsoby zápisu ... 43
5.4.1 Chybné zápisy ... 43

6 Výčtové typy ... 44
6.1 Představení výčtových typů ... 44
6.1.1 Dokud jsme neznali výčtové typy ... 44
6.2 Definice jednoduchého výčtového typu ... 45
6.3 Použití výčtového typu v programu ... 46
6.3.1 Použití výčtového typu v podmínce ... 47
6.3.2 Procházení konstant výčtového typu ... 48
6.3.3 Použití výčtového typu v příkazu `switch` ... 48
6.4 Složitější definice výčtového typu ... 49
6.4.1 Každá konstanta má jiné chování ... 51

7 Práce s kolekcemi v JDK 5.0 53
7.1 Kolekce jako generický typ 53
7.2 Práce s kolekcí 53
7.2.1 Vytvoření a naplnění kolekce 53
7.2.2 Použití automatického zapouzdření 54
7.2.3 Průchod kolekce iterátorem 54
7.3 Metody pracující s kolekcemi 55
7.4 Některé nové kolekce 56
7.4.1 Datová struktura – fronta 56
7.4.1.1 Rozhraní `Queue` 57
7.4.1.2 Třída `PriorityQueue` 57
7.4.2 Kolekce `EnumMap` a `EnumSet` 57
7.4.2.1 Kolekce `EnumMap` 58
7.4.2.2 Kolekce `EnumSet` 58

8 Formátování výstupu 60
8.1 Metody umožňující formátovat výstup 60
8.2 Formátovací procesor – třída `Formatter` 61
8.2.1 Formátovací procesor bez výstupu 61
8.2.2 Formátovací procesor s určeným výstupem 62
8.3 Syntaxe zápisu formátu 63
8.3.1 Obecný formát entity 63
8.3.2 Typ konverze 63
8.3.2.1 Konverze obecného typu 64
8.3.2.2 Konverze celočíselných typů 64
8.3.2.3 Konverze neceločíselných typů 65
8.3.2.4 Konverze znakových typů 65
8.3.2.5 Konverze data a času 65
8.3.2.6 Speciální typy konverze 67
8.3.3 Značky 67
8.4 Další detaily a chybová hlášení 67
8.4.1 Chybová hlášení při formátování výstupu 68

9 Některé užitečné třídy v JDK 5.0 69
9.1 Třída `Class` a reflexe 69
9.1.1 Třída `Class` je generická třída 69
9.2 Třída `Enum` 70
9.3 Třídy `Math` a `StrictMath` 70
9.4 Třída `StringBuilder` 71
9.5 Třída `Scanner` 71
9.5.1 Vytvoření objektu třídy `Scanner` 71
9.5.2 Jednoduché použití 71
9.5.3 Čtení řetězců odpovídajících zadanému regulárnímu výrazu 72
9.5.4 Čtení řetězců oddělených zadaným regulárním výrazem 73

10 Anotace 75
10.1 Postavení anotací v jazyku Java 75
10.2 Použití anotací v programu 75
10.2.1 Použití anotací pro označení balíčků 76
10.3 Deklarace vlastní anotace 77
10.3.1 Anotace bez parametrů 78
10.3.2 Anotace s parametry 78
10.3.2.1 Anotace s jedním parametrem 79
10.3.2.2 Pole jako typ parametru anotace 79
10.3.2.3 Anotace jako typ parametru jiné anotace 80
10.4 Standardní anotace 80
10.4.1 Metaanotace 81
10.4.1.1 Retention 81
10.4.1.2 Target 81
10.4.1.3 Documented 82
10.4.1.4 Inherited 82
10.4.2 Anotace v balíčku `java.lang` 82
10.4.2.1 Override 82
10.4.2.2 Deprecated 83
10.4.2.3 SuppressWarnings 83
10.5 Zpracování anotací 83
10.5.1 Zpracování za běhu programu 84
10.5.1.1 Získání objektu anotace 84
10.5.1.2 Práce s objektem anotace a metoda `toString()` 85
10.5.1.3 Získání objektu anotace jiné anotace 85
10.5.2 Zpracování zdrojového kódu 85
10.5.2.1 Nástroj `apt` 85
10.5.2.2 Mirror API 86
10.5.3 Možnost zpracování bajtkódu 86
10.6 Další možnosti využití anotací 87

A Java 6 Mustang 88
Některé konkrétní změny ve standardních knihovnách 88
Práce se souborovým systémem 88
Zpracování anotací 89
Java Compiler Tool 89
Možnost skriptování v Java aplikacích 89
Práce s databázemi 90
Zpracování XML dokumentů 90
Některé novinky v knihovnách pro tvorbu GUI 90
System tray icon 90
Desktop API 90
Použité zdroje 91

Literatura 93

Rejstřík 95

Seznam příkladů

Příklad 1.1: `Predstaveni` – Představení generických typů 1 1
Příklad 1.2: `Predstaveni` – Představení generických typů 2 2
Příklad 1.3: `Zasobnik1` – Ukázka programu bez využití generických typů 3
Příklad 1.4: `Zasobnik2` – Ukázka programu s využitím generických typů 4
Příklad 1.5: `SdileniGenerTridy` – Sdílení společné třídy různými generickými typy 6
Příklad 1.6: `Dvojice` – Deklarace generické třídy 7
Příklad 1.7: `Dvojice` – Instance generické třídy 8
Příklad 1.8: `Zasobnik` – Generické rozhraní 9
Příklad 1.9: `AbstractZasobnik` – Abstraktní generická třída 9
Příklad 1.10: `Polymorfizmus` – Polymorfizmus 11
Příklad 1.11: `AbstractZasobnik` – Deklarace generické metody 12
Příklad 1.12: `ZasobnikTest` – Použití generické metody 13
Příklad 1.13: `Ntice` – Ohraničení shora 1 15
Příklad 1.14: `Ntice` – Ohraničení shora 2 17
Příklad 1.15: `Ntice` – Ohraničení zdola 1 17
Příklad 1.16: `Ntice` – Ohraničení zdola 2 18
Příklad 1.17: `Ntice` – Ohraničení zdola 3 18
Příklad 1.18: `Pole` – Pole 1 21
Příklad 1.19: `Pole` – Pole 2 22
Příklad 1.20: `BeznyRodic` – Dědičnost 1 24
Příklad 1.21: `GenerRodic` – Dědičnost 2 24
Příklad 1.22: `Potomek3` – Dědičnost 3 25
Příklad 1.23: `Pretypovani` – Přetypování 26
Příklad 1.24: Sdílení společné třídy různými generickými typy 27
Příklad 2.1: `Puvodni` – Ukázka bez použití statického importu 29
Příklad 2.2: `StatickyImport` – Použití statického importu 30
Příklad 2.3: `Chyba` – Konflikt při použití statického importu 31
Příklad 3.1: `KolekceCisel` – Práce s kolekcí bez automatického zapouzdření 33
Příklad 3.2: `KolekceCisel2` – Práce s kolekcí s využitím automatického zapouzdření 33
Příklad 3.3: `Porovnani` – Operátory porovnání a automatické zapouzdření 34
Příklad 4.1: `Pole` – Průchod polem použitím cyklu *for-each* 36
Příklad 4.2: `Kolekce` – Průchod kolekcí použitím cyklu *for-each* 38
Příklad 5.1: `Zasobnik3` – Deklarace metody s proměnným počtem parametrů 41
Příklad 5.2: `Test` – Použití metody s proměnným počtem parametrů 42
Příklad 6.1: `CastDne1` – Náhrada za výčtový typ 1 44
Příklad 6.2: `CastDne2` – Náhrada za výčtový typ 2 45
Příklad 6.3: `CastDne3` – Jednoduchý výčtový typ 45
Příklad 6.4: `Studium` – Použití výčtového typu v programu 46
Příklad 6.5: `Studenti` – Použití výčtového typu v podmínce 47
Příklad 6.6: `Studenti` – Iterátor napříč výčtovým typem 48
Příklad 6.7: `Studenti` – Výčtový typ a příkaz `switch` 48
Příklad 6.8: `Operatory` – Konstanty výčtového typu s parametrem 49
Příklad 6.9: `Operatory` – Více parametrů u konstanty výčtového typu 50

Příklad 6.10: `Operatory` – Různé chování různých konstant výčtového typu 51
Příklad 6.11: `Operatory2` – Abstraktní metoda výčtového typu 52
Příklad 7.1: `Kolekce` – Vytvoření a naplnění kolekce 53
Příklad 7.2: `Zapouzdreni` – Použití auto-boxingu při práci s kolekcí 54
Příklad 7.3: `Kolekce` – Průchod kolekce iterátorem 55
Příklad 7.4: `Metody` – Metody pracující s kolekcemi 56
Příklad 7.5: `Mnozina` – Použití `EnumSet` 58
Příklad 8.1: `Test1` – Použití formátovacího procesoru 1 61
Příklad 8.2: `Test2` – Použití formátovacího procesoru 2 62
Příklad 8.3: `Test3` – Použití formátovacího procesoru 3 62
Příklad 8.4: `Formatovani` – Formátování objektu obecného typu 64
Příklad 8.5: Formátování celých čísel 64
Příklad 8.6: Formátování reálných čísel 65
Příklad 8.7: Formátování znaků 65
Příklad 8.8: Formátování času 66
Příklad 8.9: Formátování data 1 66
Příklad 8.10: Formátování data 2 67
Příklad 9.1: `Cteni` – Čtení řetězců 1 72
Příklad 9.2: `Cteni` – Čtení řetězců 2 72
Příklad 9.3: `Cteni` – Čtení řetězců 3 73
Příklad 10.1: `Zastarala` – Použití anotací v programu 76
Příklad 10.2: `Databaze` – Použití značkovací anotace 78
Příklad 10.3: `Verze` – Příklad anotace s parametry 78
Příklad 10.4: `Autor` – Příklad anotace s jedním parametrem 79
Příklad 10.5: `Autori` – Pole jako typ parametru anotace 80
Příklad 10.6: `Info` – Anotace jako typ parametru jiné anotace 80

Seznam tabulek

Tabulka 3.1: Primitivní a objektové typy 32
Tabulka 3.2: Použitelnost automatického zapouzdření 35
Tabulka 4.1: Metody umožňující průchod mapy cyklem *for-each* 39
Tabulka 7.1: Metody rozhraní `Queue` 57
Tabulka 7.2: Metody třídy `EnumSet` 58
Tabulka 8.1: Typy konverze obecného typu 64
Tabulka 8.2: Formátování celočíselných typů 64
Tabulka 8.3: Formátování neceločíselných typů 65
Tabulka 8.4: Typ konverze znakových typů 65
Tabulka 8.5: Typy konverze údajů o čase 65
Tabulka 8.6: Typy konverze údajů o datu 66
Tabulka 8.7: Běžně používané formáty data a času 66
Tabulka 8.8: Speciální typy konverze 67
Tabulka 8.9: Značky použitelné pro určité typy konverzí 67
Tabulka 9.1: Metody třídy `Enum` 70
Tabulka 10.1: Typy působnosti anotací 81
Tabulka 10.2: Typy deklarací, u kterých je možno anotaci použít 81
Tabulka 10.3: Potlačení varovných hlášení 83
Tabulka 10.4: Metody rozhraní `AnnotatedElement` 84
Tabulka 10.5: Základní struktura Mirror API 86

Úvod

Tato publikace vznikla jako závěrečná práce mého studia bakalářského oboru Výpočetní technika a informatika na Pedagogické fakultě Jihočeské univerzity v Českých Budějovicích. I zde zajišťuje katedra informatiky výuku objektově orientovaného programování, konkrétně výuku programování v jazyku Java. Snad osud (spíše ale vývojáři společnosti *Sun Microsystems*) tomu chtěl, že v září roku 2004 byla uvolněna zcela nová verze tohoto programovacího jazyka, která podstatně rozšířila jeho možnosti. Tyto změny se dříve nebo později musí zákonitě objevit i ve výuce programování. Z toho se také odvíjí hlavní záměr tvorby této publikace – má za úkol představit nejdůležitější a nejzásadnější změny, kterých programovací jazyk Java 5.0 doznal. Celá tato práce by tedy měla sloužit jako výukový materiál pro studenty prvého ročníku oboru výpočetní technika. Předpokladem pro snazší porozumění textu je alespoň základní znalost programovacího jazyka Java.

Zároveň ale doufám, že bude užitečná i mnohým dalším čtenářům.

Kde se vzalo označení Java 5.0 Tiger

Již podruhé v historii vývoje programovacího jazyka Java došlo k tak výraznému rozšíření, že se autoři rozhodli prezentovat navenek Javu s novým označením čísla verze. Podobně tomu vylo při uvolnění verze 1.2 na přelomu let 1998 a 1999. V této verzi byly knihovny nejen rozšířeny, ale byly zásadním způsobem přeorganizovány. Autoři se tehdy rozhodli tuto verzi prezentovat pod označením **Java 2**.

Poslední změna, v září roku 2004, přinesla nejen rozšíření standardních knihoven, ale i zásadní změny v syntaxi jazyka. Původně byla tato verze uvolněna pod označením Java 2 verze 1.5 – Tiger. Stejně tak standardní knihovny nesly označení JDK 1.5. Postupem času ale společnost *Sun Microsystems*, zřejmě z marketingových důvodů, změnila označení standardních knihoven na **JDK 5.0**. Proto se dnes často v literatuře setkáme s označením **Java 5.0**. Tohoto označení se budeme držet i my.

Nenechte se ale zmást označením čísel verzí např. v dokumentaci k některým třídám standardních knihoven. Autoři Javy pro své vnitřní účely používají stále označení verze 1.5.

Forma zpracování práce

Jelikož výklad jednotlivých témat zahrnuje i poměrně značnou část teorie, rozhodl jsem se po dohodě s vedoucím práce rozdělit celý výukový materiál na dvě relativně samostatné části. Teoretická část, kterou právě čtete, se zaměřuje výhradně na výklad jednotlivých témat na modelových příkladech. Druhá část práce je ryze praktická a zaměřuje se na prezentaci reálného využití poznatků z teoretické části v praxi. Tato praktická část má tedy podobu jakéhosi tutoriálu, který se nezaměřuje na výklad s vysvětlením problematiky, ale pouze tvorbu konkrétních úloh. Některé příklady tutoriálu také kombinují použití několika nových rysů jazyka Java najednou.

Jedním z požadavků na zpracování této práce byla také možnost jejího využití pro studenty kombinované formy studia. Nejvhodnějším způsobem je pochopitelně elektronická podoba. Celá práce je tedy zpracována i ve formátu XML s využitím DTD DocBook v4.4. Různé formáty práce jsou dostupné na internetové adrese *http://milost.wz.cz/bap/*.

Uspořádání textu

Celá publikace je rozdělena do deseti kapitol. Prvních šest kapitol se týká výhradně syntaktických rozšíření jazyka Java. Primárním hlediskem pro rozdělení jednotlivých kapitol je jistě jejich vzájemná návaznost jednotlivých. Ne vždy jde ale toto hledisko dodržet stoprocentně. Mezi jednotlivými kapitolami se proto setkáte s odkazy (viz část **Typografické a syntaktické konvence**) na různé části či konkrétní příklady z jiných kapitol.

Nejvýznamnější (kupodivu i nejdelší) část tvoří první kapitola. Týká se generických typů, které jsou podle mého názoru největší a nejzásadnější změnou v syntaxi jazyka Java. Další čtyři kapitoly popisují některá nepříliš rozsáhlá rozšíření jazyka Java, které ale mohou v mnohých případech velmi usnadnit práci a zjednodušit či zpřehlednit zdrojový kód programu. Další poměrně významnou změnou je zavedení výčtových typů. O těch pojednává šestá kapitola.

Následují tři kapitoly, které již nepopisují změny v syntaxi jazyka Java, se soustřeďují spíše na rozšíření standardních knihoven. Nových tříd v JDK 5.0 je opravdu mnoho, snažil jsem se tedy vybrat ty, se kterými se můžeme setkat častěji, a které také mohou mnohdy výrazně zjednodušit práci.

Poněkud zvláštní postavení má desátá kapitola, týkající se anotací. Anotace jsou dalším poměrně rozsáhlým rozšířením, které se týká jak syntaxe jazyka, tak některých knihovních tříd. Důvod, proč je tato kapitola zařazena až na závěr, je prostý. Jde sice o zásadní rozšíření programovacího jazyka Java, ale pro úplného začátečníka není příliš jednoduché se s ním vypořádat. Také si myslím, že student programování musí v prvé řadě zvládnout základní programovací techniky, a potom je jen na něm, kterou z těch pokročilejších si vybere. Věřte, že kromě anotací jich Java nabízí opravdu mnoho.

Příklady k teoretické části

Příklady z jednotlivých kapitol teoretické části jsou zpracovány jako samostatné projekty `01 – 10` v prostředí **NetBeans 5.0**. Pro přehlednost a pro úplnost dodávám i samostatné zdrojové kódy příkladů z jednotlivých kapitol, se kterými je možné pracovat i v jiných prostředích, např. v prostředí **BlueJ**. Příklady k teoretické části jsou volně dostupné na internetové adrese *http://milost.wz.cz/bap/*.

Tutoriál – praktická část

Součástí bakalářské práce je několik dalších samostatných projektů. Jejich hlavním záměrem by měla být ukázka použití několika nových rysů jazyka Java najednou, na rozdíl od příkladů v teoretické části, které mají být dostatečně jednoduché a jsou tedy spíše modelovými příklady soužícími ke snazšímu pochopení popisované problematiky.

Jednotlivé části těchto samostatných projektů jsou organizovány v balíčcích začínajících vždy `tutorial.`*`název_tutoriálu`*. Úvodní popis každého projektu je zpracován formou referenčních stránek. V příkladech je dále využito dokumentačních komentářů, které popisují konkrétní použitou metodu, ale již nezahrnují výklad problematiky (ten je obsahem teoretické části).

Zdrojové kódy projektů a jejich úvodní popis je rovněž volně k dispozici na internetové adrese *http://milost.wz.cz/bap/*. I tyto příklady jsou k dispozici ve dvou verzích. Jednak jako projekty zpracované v prostředí **NetBeans 5.0**, a také jako samostatné zdrojové soubory, se kterými je možné pracovat i v jiných prostředích.

Přiložený CD-ROM

Součástí práce je i jeden CD-ROM, který obsahuje veškeré součásti práce, jako jsou všechny uvedené příklady, příklady z tutoriálu, elektronická verze práce a nástroje, které byly použity pro zpracování této elektronické verze.

Veškerý software na tomto CD je open-source a je uveden včetně originální dokumentace a licenčních podmínek.

Typografické a syntaktické konvence

V celém textu jsem se určitým způsobem snažil odlišit poznámky, doplnění, příklady atd.

metaanotace	Výraz, který je použit poprvé.
deprecated	Anglický výraz. Často takto označuji i varovná hlášení překladače (ty jsou totiž také anglicky).
kompilátor	Tučně jsou zvýrazněny důležité výrazy.
Poznámka:	Začátek poznámky, upozornění, dobré rady či důležitého doplnění.
	Konec poznámky, upozornění, dobré rady či důležitého doplnění.
Příklad 7.2:	Následuje druhý příklad v sedmé kapitole.
Výstup programu:	Odděluje výsledek programu od jeho zdrojového kódu.
`int počet;`	Neproporcionálním písmem označuji výpis zdrojového kódu programu. Může se vyskytovat i v textu, někdy dokonce i v nadpisu.
TypParametru	Označení obecného zápisu syntaxe.
...	Někdy je uveden pouze fragment zdrojového kódu programu, který je důležitý pro pochopení právě vykládané problematiky. Tři tečky označují místo, kde je část kódu vypuštěna. Kompletní zdrojové kódy všech příkladů naleznete na přiloženém CD.
[9.5/71]	Odkaz na kapitolu 9.5 na straně 71.
[Příklad 8.9]	Odkaz na devátý příklad v osmé kapitole.

Poděkování

Na tomto místě bych velmi rád poděkoval RNDr. Jaroslavu Ichovi, vedoucímu práce, za pomoc při výběru a při návrhu zpracování jednotlivých částí práce, a také za individuální konzultace, které určitě přispěly ke zkvalitnění obsahu práce.

Poděkování také patří mému bratrovi Láďovi za pomoc při korektuře a konečné úpravě textu, které jsou přinejmenším stejně důležité, jako vlastní obsah.

A samozřejmě za psychickou podporu, nejen během zpracovávání této práce, ale i během celého mého studia, děkuji svým rodičům a své přítelkyni Janě.

Miloslav Thon

1 Generické typy

Generické typy jsou bezesporu nejrozsáhlejší a nejužitečnější změnou jazyka Java 5.0, která se týká samotného jazyka, resp. jeho syntaxe. V této kapitole se budeme věnovat základní syntaxi a použití generických typů. Naučíme se deklarovat **vlastní generické typy**. Zmíníme se krátce i o **využití polymorfizmu** ve spojení s generickými typy. V předposlední části kapitoly si ukážeme postavení generických typů v hierarchii **dědičnosti**. A na závěr kapitoly se dozvíme, jak **kompilátor** s našimi generickými typy zachází.

Většina příkladů bude ukazovat spíše obecné použití generických typů. Některé konkrétní a praktické možnosti využití generických typů uvidíme také v kapitole [7/53] o práci s kolekcemi v JDK 5.0 a také v **tutoriálu**. Jak uvidíme dále, konstrukce generických typů je totiž nejvíce uplatněna právě ve třídách pro práci s kolekcemi.

1.1 Představení generických typů

Asi nejširší uplatnění nalezla tato konstrukce v knihovnách pro práci s kolekcemi. Jelikož kolekce využíváme pro uchovávání objektů různých tříd, byly až do verze JDK 1.4 definovány třídy a rozhraní pro práci s kolekcemi obecně pro práci s objekty třídy `Object`. Při použití kolekce tedy není jednoznačně určeno, jaké objekty budeme v kolekci uchovávat. To přináší poněkud neobratné zápisy zdrojového kódu při práci s jednotlivými prvky kolekce.

Příklad 1.1: `Predstaveni` – Představení generických typů 1

```
import java.util.*;

public class Predstaveni {
    public static void kolekceTest1() {
        Map slovnik = new HashMap();

        slovnik.put("pivo", "beer");
        slovnik.put("vino", "vine");
        slovnik.put("mléko", "milk");
        slovnik.put("voda", "water");

        String slovo = (String) slovnik.get("voda");

        System.out.println(slovo);
        System.out.println(slovnik);
    }

    public static void main(String[] args) {
        kolekceTest1();
    }
}
```

Jelikož metoda `get()` vrací typ `Object`, musíme tuto hodnotu explicitně přetypovat na typ `String`. Pokud ovšem vložíme do kolekce jiný objekt, než je objekt typu `String`, dojde při pokusu o přetypování k vyvolání výjimky `ClassCastException`. Této situaci se můžeme vyhnout, použijeme-li konstrukci generických typů.

Je tedy možné říci kompilátoru nejenom to, že proměnná `slovnik` je typu `Map`, ale také to, že půjde o slovník, který bude uchovávat dvojice `String`, `String`? A bude kompilátor kontrolovat, zda vkládáme do kolekce pouze objekty typu `String`? A bude také metoda `get()` vracet přímo objekty typu `String`?

Odpověď na všechny tyto otázky je samozřejmě kladná. Použití generických typů nám umožňuje již při deklaraci proměnné, v našem případě `slovnik`, určit, jaké objekty bude kolekce uchovávat.

Příklad 1.2: `Predstaveni` – Představení generických typů 2

```
import java.util.*;

public class Predstaveni {
    ...

    public static void kolekceTest2() {
        Map<String, String> slovnik = new HashMap<String, String>();

        slovnik.put("pivo", "beer");
        slovnik.put("vino", "vine");
        slovnik.put("mléko", "milk");
        slovnik.put("voda", "water");

        String slovo = slovnik.get("voda");

        System.out.println(slovo);
        System.out.println(slovnik);
    }

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

        kolekceTest2();
    }
}
```

Tímto jsme kompilátoru řekli, že proměnná slovnik bude uchovávat dvojice klíč-hodnota, obě typu `String`. Potom i metoda `get()` bude vracet přímo objekt typu `String`. Použití generických typů je tedy poměrně intuitivní a samozřejmě naprosto typově bezpečné.

Následující kód ukazuje část deklarace generické třídy `HashMap` v JDK 5.0:

```
public class HashMap<K, V>
{
    public V get(Object klic) { . . . }
    public V put(K klic, V hodnota) { . . . }
}
```

V tomto kódu jsou `K` a `V` formálními parametry generické třídy `HashMap` (nebo také typové parametry) reprezentující určité třídy, se kterými bude objekt typu `HashMap` pracovat. Můžeme také říct, že třída `HashMap` je **parametrizovaná**. Takto označené třídy `K` a `V` jsou použity u metod `get()` a `put()` (ale i u dalších) jako jejich návratové hodnoty, popř. jako typy jejich parametrů. To nám zajistí, že tyto metody budou vracet tentýž typ, který jsme určili při deklaraci proměnné `slovnik`.

1.1.1 Shodneme se na pojmech?

V této kapitole je uveden přehled nových pojmů, které budeme dále používat. Není to proto, že by jejich význam nebyl patrný z textu, jde spíše o to, aby význam těchto pojmů byl jednotný a pokud možno dobře vyložený.

- ***generická třída*** – generickou třídou budeme rozumět konkrétní třídu, která je parametrizovaná (tzn. v její deklaraci je uveden seznam formálních parametrů)

- ***generický typ*** – tento pojem budeme používat pro označení typu proměnné nebo typu parametru metody, kterým je generická třída; někdy také obecně pro jakoukoliv generickou třídu, jelikož definicí třídy definujeme vlastně i datový typ; anebo v množném čísle (tj. „generické typy") také jako pojmenování celé této úžasné konstrukce

- ***typový parametr*** – je označení pro formální parametr generické třídy reprezentující nějakou konkrétní třídu; v závislosti na kontextu můžeme typovým parametrem rozumět jak třídu, tak samotný identifikátor tohoto parametru

1.1.2 Hlavní výhody použití generických typů

Generické typy nám umožňují vytvářet jakési **zobecnění tříd**, neboli definovat celou množinu tříd, které mají společné vlastnosti a metody. Při použití generického typu pak uvádíme název jiné třídy jako jeho formální parametr. Jako generický typ můžeme definovat nejen třídu, ale i abstraktní třídu nebo rozhraní.

Jak již bylo zmíněno v úvodu, generické typy jsou značným rozšířením syntaxe jazyka Java, ale ve velké míře také jeho standardně dodávaných knihoven. Asi nejvíce je toto rozšíření patrné v knihovnách pro práci s kolekcemi. Na jednoduchém příkladu práce s kolekcí si ukažme hlavní výhody využití generických typů:

- **Zobecnění definice třídy** – generickým typem definujeme celou množinu tříd, které mají stejné vlastnosti a chování, pouze pracují pokaždé s objekty různých tříd.

- **Typová bezpečnost** – při deklarací generického typu přesně označíme jeho parametry, popř. je vymezíme a ohraničíme pro použití určitých tříd (viz kapitola [1.5/14]).

- **Eliminace přetypování** – při deklaraci proměnné uvádíme jako parametr generického typu konkrétní třídu, odpadá tedy nutnost jakéhokoliv přetypování.

1.1.3 Příklad práce s generickým typem

V této kapitole si na konkrétním příkladě ukážeme praktické použití generických typů. Nebudeme se zde však zatím dotýkat syntaktických pravidel deklarace generických typů, to bude obsahem dalších kapitol. Pouze si ukážeme hlavní výhody použití generických typů, které jsme zmínili v kapitole [1.1.2/3], a srovnáme je s původním přístupem, který jsme používali v předchozích verzích jazyka Java.

Příklad 1.3: `Zasobnik1` – Ukázka programu bez využití generických typů

```
public class Zasobnik1 {
    private Object[] prvky;
    private int pocet;
```

```
    public Zasobnik1(int velikost) {
        prvky = new Object[velikost];
        pocet = 0;
    }
    public boolean vlozit(Object prvek) {
        if (pocet == prvky.length) return false;

        prvky[pocet] = prvek;
        pocet++;
        return true;
    }
    public Object odebrat() {
        if (pocet == 0) return null;

        Object obj = prvky[pocet-1];
        pocet--;
        return obj;
    }
    public int pocetPrvku() {
        return pocet;
    }
    public static void main(String[] args) {
        Zasobnik1 zasobnik = new Zasobnik1(30);

        //zasobnik.vlozit(new Integer(5));
        zasobnik.vlozit("slovo1");
        zasobnik.vlozit("slovo2");
        zasobnik.vlozit("slovo3");

        String prvek;
        while ((prvek = (String) zasobnik.odebrat()) != null)
            System.out.println(prvek);
    }
}
```

Výstup programu:

```
slovo3
slovo2
slovo1
```

Pokud v tomto příkladě zrušíme komentář u příkazu `zasobnik.vlozit(new Integer(5))`, kompilátor nezahlásí při překladu žádnou chybu. Parametr `prvek` metody `vlozit()` je totiž typu `Object` a nic nám tedy nebrání vložit do tohoto zásobníku jednou objekt typu `Integer`, podruhé objekt typu `String` a příště třeba něco úplně jiného. K chybě dojde až za běhu programu, kdy nemáme přehled o typech jednotlivých prvků uložených v zásobníku a všechny se je snažíme přetypovat na typ `String`. V místě, kdy se na typ `String` snažíme přetypovat objekt typu `Integer`, dojde k vyvolání výjimky `ClassCastException`.

V následujícím příkladu uvidíme zásadní rozdíl ve využití generických typů. Jde o stejný případ zásobníku, jako v předchozím příkladě. Srovnejte tedy rozdíl mezi třídami `Zasobnik1` a `Zasobnik2`.

Příklad 1.4: `Zasobnik2` – Ukázka programu s využitím generických typů

```
public class Zasobnik2<T> {
    private Object[] prvky;
    private int pocet;
```

```
    public Zasobnik2(int velikost) {
        prvky = new Object[velikost];
        pocet = 0;
    }
    public boolean vlozit(T prvek) {
        if (pocet == prvky.length) return false;

        prvky[pocet] = prvek;
        pocet++;
        return true;
    }
    public T odebrat() {
        if (pocet == 0) return null;

        Object obj = prvky[pocet-1];
        pocet--;
        return (T) obj;
    }
    public int pocetPrvku() {
        return pocet;
    }
    public static void main(String[] args) {
        Zasobnik2<String> zasobnik = new Zasobnik2<String>(30);

        //zasobnik.vlozit(new Integer(5));
        zasobnik.vlozit("slovo1");
        zasobnik.vlozit("slovo2");
        zasobnik.vlozit("slovo3");

        String prvek;
        while ((prvek = zasobnik.odebrat()) != null)
            System.out.println(prvek);
    }
}
```

Výstup programu:

```
slovo3
slovo2
slovo1
```

Pokud zrušíme komentář u příkazu `zasobnik.vlozit(new Integer(5))` v tomto příkladě, uvidíme, že dojde k chybě už při překladu programu. To je způsobeno tím, že metoda `vlozit()` očekává parametr typu `T`, neboli `String`, jak jsme určili při vytváření instance `zasobnik` v metodě `main()`. Tím je zajištěna typová kontrola již v době překladu.

Poznámka:

Možná vznesete námitku, proč v druhém příkladě deklarujeme třídu `Zasobnik2` jako generickou třídu s parametrem `T`, když prvky zásobníku ukládáme do pole typu `Object` a při jejich odebírání ze zásobníku tyto explicitně přetypováváme na typ `T`. Vaši námitku přijímám, ale mám pro ni dobré zdůvodnění. Není žádný problém deklarovat instanční proměnnou prvky třídy `Zasobnik2` jako proměnnou typu `T[]`. Je ale problém vytvořit instanci třídy `T`. Proč to nejde se dozvíme v kapitole [1.6.2/22]. Berte tedy v tuto chvíli tento příklad jako modelový. Je směrován spíše na použití generické třídy, než na její deklaraci. Tomu se budeme věnovat v dalších částech této kapitoly.

■

Poznámka 2.:
V kapitole [1.3/10] si ukážeme také využití polymorfizmu při práci s generickými typy.

■

1.1.4 Označení typových parametrů generické třídy

Parametry generických tříd budeme vždy označovat jedním velkým písmenem – např. dvojice `A`, `B`, nebo dvojice `K`, `V` (key-value – klíč-hodnota), nebo jen `T`, pokud jde o jeden samostatný typový parametr.

1.1.5 Zpětná kompatibilita

Jakkoli je použití generických typů při práci s kolekcemi elegantní, nic nám nebrání tomu, abychom i v nové verzi jazyka Java použili původní způsob. Pokud tedy při deklaraci proměnné generického typu neuvedeme třídu, se kterou se má pracovat, kompilátor bude vědět, že pracujeme s objekty neurčitých typů, a bude při další práci s nimi vyžadovat explicitní přetypování.

To může být užitečné, pokud chceme např. pokračovat ve vývoji již hotového programu v nové verzi Javy. Nemusíme tedy nutně přetvářet všechny naše knihovny s využitím generických typů, pokud víme, že pracují bezchybně. Kompilátor nás při jejich překladu pouze varuje, že provádíme potenciálně nebezpečné operace.

1.1.6 Šablony v C++ versus generické typy v Javě

Programátorům znalých jazyka C++ mohou generické typy připomínat **šablony**, se kterými se pracuje velmi podobně. Podobnost je ovšem pouze v syntaxi.

Šablony v programovacím jazyku C++ fungují jako makra a lze je navíc použít i pro primitivní typy. Například pro dvě proměnné `List<String> seznamSlov` a `List<int> seznamCisel` vytvoří kompilátor C++ dvě různé třídy, se kterými poté program pracuje.

Objekty typů `LinkedList<String>` a `LinkedList<Integer>` **sdílí v Javě jednu společnou třídu** `LinkedList`. Kompilátor při překladu kontroluje kompatibilitu typů a zajistí správné přetypování. Virtuální stroj tedy pracuje pouze s jednou třídou. Hlavním důvodem, proč je tato koncepce v nové verzi programovacího jazyka Java implementována, je přesunutí co největšího množství typových kontrol do fáze překladu programu. Samotný běh programu tak může být mnohem bezpečnější. Generické typy jsou tedy **záležitostí kompilátoru**, nikoliv virtuálního stroje Javy.

Následující příklad dokazuje, že dva objekty typů `List<String>` a `List<Integer>` jsou v podstatě objekty jedné společné třídy:

Příklad 1.5: `SdileniGenerTridy` – Sdílení společné třídy různými generickými typy

```
import java.util.*;

public class SdileniGenerTridy {
    public static void main(String[] args) {
        List<Integer> i = new LinkedList<Integer>();
        List<String> s = new LinkedList<String>();

        System.out.println(i.getClass());
        System.out.println(s.getClass());
    }
}
```

Výstup programu:

```
class java.util.LinkedList
class java.util.LinkedList
```

1.2 Vytvoření vlastního generického typu

V kapitole [1.1/1] jsme si nastínili intuitivní představu použití generických typů. V této kapitole se pokusíme podrobně popsat práci s generickými typy, jejich deklaraci a použití. Nejprve si ukážeme, jak deklarovat běžnou třídu jako generický typ. Z toho potom odvodíme i deklaraci rozhraní či abstraktní třídy, které také mohou být generickými typy.

Upozornění:
Třídy výjimek a chyb nemohou být deklarovány jako generické.

■

1.2.1 Deklarace generické třídy

Deklarace generické třídy je téměř úplně stejná, jako deklarace jakékoliv jiné třídy. Jediné, co je zde navíc, jsou již zmíněné **formální parametry třídy, neboli typové parametry**. Těmito typovými parametry definujeme ono již zmíněné **„zobecnění třídy"**.

Na následujícím příkladu si ukážeme deklaraci jednoduché generické třídy, která představuje jakousi uspořádanou dvojici zatím neznámých objektů typu `A`, resp. `B`.

Příklad 1.6: `Dvojice` – Deklarace generické třídy

```
public class Dvojice<A, B> {
    private A prvni;
    private B druhy;

    public Dvojice(A prvni, B druhy) {
        this.prvni = prvni;
        this.druhy = druhy;
    }

    public void nastavPrvni(A o) {
        prvni = o;
    }
    public void nastavDruhy(B o) {
        druhy = o;
    }

    public A prvni() {
        return prvni;
    }
    public B druhy() {
        return druhy;
    }
    public String toString() {
        return "[" + prvni.toString() + ", " + druhy.toString()+ "]";
    }

    ...
}
```

Hlavička třídy obsahuje její název (`Dvojice`) a seznam formálních parametrů (`A` a `B`) uvedený ve špičatých závorkách < a > těsně za názvem třídy. Tyto parametry představují určité třídy, které můžeme použít kdekoliv v těle definované generické třídy – jako typ proměnné, jako typ parametru metody nebo jako návratový typ metody.

Parametry `A` a `B` jsou tak svázány s proměnnými či metodami. Pokud tedy při použití naší třídy určíme, že onou dvojicí `A`, `B` bude dvojice typů `Integer`, `String`, kompilátor bude vědět, že první parametr konstruktoru bude `Integer` a druhý `String`. Bude také vědět, že metoda `prvni()` vrací objekt typu `Integer`, a metoda `druhy()` vrací objekt typu `String`.

V těle konstruktoru ani v těle jiné metody **nemůžeme vytvořit instanci typového parametru**, ačkoli jde o třídu. Kdybychom např. v konstruktoru třídy `Dvojice<A, B>` uvedli

```
this.prvni = new A();        // !!! chyba !!!
```

vyvolá kompilátor chybu ‚*unexpected type* – neočekávaný typ'. V době překladu této třídy kompilátor neví, jaká třída bude stát na místě typového parametru `A`, a nemůže tedy vědět, jaký má třída `A` konstruktor.

Upozornění:

Typové parametry nemůžeme použít jako typy **statických členů** nebo jako návratové typy běžných statických metod. Je to z toho důvodu, že k těmto statickým prvkům můžeme přistupovat i bez vytvoření instance dané třídy. A dokud nevytvoříme instanci generické třídy, nevíme, jaké třídy jsou ukryty pod typovými parametry.

■

Důležité:

V pozici typového parametru generické třídy může stát jakákoliv **třída, abstraktní třída, rozhraní nebo výčtový typ.** Typovým parametrem však **nemůže být primitivní typ.**

■

1.2.1.1 Vytvoření objektu generické třídy

Stejně tak, jako jsme uvedli parametry třídy v její deklaraci, musíme je uvést i v deklaraci její instance, a také při volání konstruktoru (v našem příkladu třídy `Dvojice`).

Příklad 1.7: `Dvojice` – Instance generické třídy

```
public class Dvojice<A, B> {
    ...

    public static void main(String[] args) {
        Dvojice<Integer, String> dvojice;
        dvojice = new Dvojice<Integer, String>(new Integer(18),
                                               "nějaký řetězec");
        System.out.println(dvojice);
    }
}
```

Výstup programu:

```
[18, nějaký řetězec]
```

Vidíme, že deklarace proměnné a vytvoření instance generické třídy je stejná, jako u jakékoliv běžné třídy. Navíc je zde pouze seznam typových parametrů uvedený ve špičatých závorkách vždy těsně za názvem generické třídy. Další práce s takovouto proměnnou nemá téměř žádná další specifika. Pouze musíme počítat s tím, že parametry metod volaných

prostřednictvím této proměnné, mohou očekávat, a velice často očekávají, právě ty typy, které jsme uvedli jako typové parametry při deklaraci této proměnné.

1.2.2 Deklarace rozhraní

Obdobným způsobem, jako jsme deklarovali generickou třídu, můžeme deklarovat i jakési *„generické rozhraní"*. Význam je rovněž obdobný, a sice **zobecnění rozhraní** pro práci s předem neurčenými typy.

Připomeňme si [Příklad 1.4] z kapitoly [1.1.3/4]. Zde jsme jako generický typ deklarovali třídu `Zasobnik2`. V tomto příkladě je ale pevně daná implementace všech operací, které můžeme s takovýmto zásobníkem provádět. Uvažme situaci, kdybychom chtěli definovat pouze tyto operace, ale ne konkrétní algoritmy. S výhodou použijeme rozhraní.

Příklad 1.8: `Zasobnik` – Generické rozhraní

```
public interface Zasobnik<T> {
    public boolean vlozit(T prvek);
    public T odebrat();
    public int pocetPrvku();
}
```

Rozhraní `Zasobnik` definuje pouze operace, které budou společné pro jakýkoliv zásobník, jež implementuje toto rozhraní. Už v tomto rozhraní jsme ale definovali, že případný zásobník musí pracovat s prvky zatím neurčeného typu `T`.

1.2.3 Deklarace abstraktní generické třídy

Jiným způsobem, jak můžeme definovat obecný zásobník, je použití **abstraktní třídy**. I abstraktní třída může být deklarována jako generický typ.

Příklad 1.9: `AbstractZasobnik` – Abstraktní generická třída

```
public abstract class AbstractZasobnik<T> implements Zasobnik<T> {
    protected int pocet;

    public abstract boolean vlozit(T prvek);
    public abstract T odebrat();

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

Třída `AbstractZasobnik` pochopitelně implementuje rozhraní `Zasobnik`. Společné operace pro práci s jakýmkoliv zásobníkem jsou tedy stejné. Navíc zde deklarujeme instanční proměnnou `pocet`, která uchovává aktuální počet prvků v zásobníku, a metodu `pocetPrvku()`, která vrací počet prvků v zásobníku. Tyto dva nové prvky budou tedy také společné pro všechny zásobníky.

Vidíme, že použití rozhraní či abstraktní třídy je také velmi intuitivní a příjemné. Zda použijeme rozhraní nebo abstraktní třídu záleží ale pouze na nás. V některých konkrétních případech může být vhodnější použití rozhraní, v některých případech použijeme raději abstraktní třídu.

Poznámka:

V této kapitole jsme si ukázali, jak deklarovat generickou třídu, rozhraní a abstraktní třídu. V kapitole [1.7/23] se budeme podrobněji věnovat **dědičnosti s využitím generických typů.**

■

1.3 Polymorfizmus

S pojmem ***polymorfizmus*** se setkáme v každém objektově orientovaném programovacím jazyce. Obecně si pod tímto pojmem představíme jakousi mnohotvarost, či různotvarost V programovacím jazyce se jedná o možnost volat stejné metody u různých objektů, aniž bychom věděli, jakého přesně jsou typu. Navíc může mít stejná metoda u různých objektů odlišný význam. To je možné díky tomu, že vždy známe společného předka těchto různých objektů. Tím může být třída, abstraktní třída nebo rozhraní.

Polymorfizmus je možno pochopitelně použít i ve spojení s generickými typy. Musíme si dát ovšem pozor na některé zápisy kódu, které mohou mít odlišný význam, než je na první pohled zřejmé.

Vysvětlení problematiky bude dobře srozumitelné na příkladech prezentujících práci s kolekcemi. První příklad ukazuje použití polymorfizmu na úrovni typu proměnné.

```
List<Integer> celaCisla = new LinkedList<Integer>();
```

To je možné, jelikož třída `LinkedList` implementuje rozhraní `List`, neboli je jeho podtřídou, a u obou těchto typů je uvedený stejný typový parametr. Není ovšem možné stejným způsobem použít **polymorfizmus na úrovni typových parametrů** generických tříd.

```
List<Integer> celaCisla = new LinkedList<Integer>();
List<Number> vsechnaCisla = celaCisla;          // !!! chyba !!!
```

I když je třída `Integer` podtřídou třídy `Number`, tento příkaz skončí chybou *‚incompatibile type'*. Je zde totiž jednoznačně určeno, že proměnná `vsechnaCisla` odkazuje na objekt typu `List`, který uchovává objekty typu `Number` – s žádným jiným typem nepracuje. My se ovšem snažíme do této proměnné dostat objekt typu `List`, který uchovává objekty typu `Integer` – s žádným jiným typem nepracuje. Asi už tušíme, že není možné ani následující:

```
List<Object> vsechnaCisla = new LinkedList<Integer>();
```

Pokud budeme chtít ukládat do seznamu různá čísla (`Integer`, `Long`, `Double`, ...) v libovolném pořadí a budeme chtít přistupovat k nim přes jejich společného předka (tj. přes objekt typu `Number`), musíme typ `Number` uvést už při vytváření seznamu.

```
List<Number> vsechnaCisla = new LinkedList<Number>();
```

Teď nám nic nebrání vkládat do seznamu `vsechnaCisla` objekty jakýchkoliv tříd, které jsou podtřídami třídy `Number`.

```
vsechnaCisla.add(new Integer(1));
vsechnaCisla.add(new Double(12.8));
vsechnaCisla.add(Long.MAX_VALUE);
```

Všechny metody např. pro výběr prvku ze seznamu budou vracet objekt typu `Number`.

```
Number cislo = vsechnaCisla.get(2);
System.out.println(cislo);
```

Zkusíme si tyto příkazy napsat do jednoho testovacího scénáře a vyzkoušet.

Příklad 1.10: `Polymorfizmus` – Polymorfizmus

```
import java.util.*;

public class Polymorfizmus {
    public static void test1() {
        List<Integer> celaCisla = new LinkedList<Integer>();
        //List<Number> vsechnaCisla = celaCisla; // !!! chyba !!!
        //List<Object> vsechnaCisla = new LinkedList<Integer>();
                                                  // !!! chyba !!!

        List<Number> vsechnaCisla  = new LinkedList<Number>();

        vsechnaCisla.add(new Integer(1));
        vsechnaCisla.add(new Double(12.8));
        vsechnaCisla.add(Long.MAX_VALUE);

        Number cislo = vsechnaCisla.get(1);
        System.out.println(cislo);

        System.out.println(vsechnaCisla);
    }
    public static void main(String args[]) {
        Polymorfizmus.test1();
    }
}
```

Výstup programu:

```
12.8
[1, 12.8, 9223372036854775807]
```

1.3.1 Zástupný symbol ?

Pokud chceme deklarovat proměnnou typu `LinkedList`, která bude odkazovat na seznam uchovávající předem neurčité prvky, je to možné s použitím ***zástupného symbolu*** `?`.

```
LinkedList<?> cisla = new LinkedList<Integer>();
//   nebo
//List<?> cisla = new LinkedList<Integer>();
```

Není potom ale možné volat instanční metody, které mají některé parametry závislé na určitém parametru generické třídy. To kompilátor hlídá velice obezřetně.

```
cisla.add(new Integer(18));       // !!! chyba !!!
```

My sice víme, že jsme vytvořili seznam typu `LinkedList<Integer>`, ale neví to kompilátor. Drobnou výjimku tvoří **metody bez parametrů**, které mají jako návratový typ uvedený parametr třídy. Následující příkaz proběhne bez chyby.

```
Object o = cisla.getFirst();
```

Metoda `getFirst()` totiž nepotřebuje znát parametr třídy pro samotné její zavolání, nebudeme ale mít tušení, jakého typu bude objekt, který nám tento příkaz vrátí. Bude to `Integer`? Nebo něco jiného?

```
Integer i = cisla.getFirst();      // !!! chyba !!!
Object o = cisla.getFirst();       // bez chyby
```

My sice víme, že by to měl být objekt typu `Integer`*, ale ani v tomto případě to kompilátor neví. Ten stále pracuje s otazníkem (*`?`*).*

Můžeme ale bez problémů provádět akce, které jsou stejné pro všechny seznamy typu `LinkedList`. Např.:

```
cisla.clear();
cisla.isEmpty();
cisla.size();
```

Poznámka:

Teď vás možná napadne, proč vůbec zástupný symbol používat. Vždyť můžeme přeci proměnnou `cisla` deklarovat bez uvedení typového parametru. To sice můžeme, ale potom už nejde o použití generického typu. Všechny metody objektu `cisla` pak budou vracet objekty typu `Object` a parametry těchto metod budou také typu `Object` namísto typu určeného typovým parametrem. A nebudeme také mít zajištěnou typovou kontrolu, kterou nám poskytuje kompilátor v případě použití generických typů.

■

1.4 Generické metody

Z předchozích částí kapitoly víme, jakým způsobem můžeme zobecnit třídu a vytvořit tak tzv. generický typ. **Stejné konstrukce lze použít i u metod.** Jakoukoliv metodu můžeme zobecnit pro práci s jakýmsi obecným typem `T`. ***Generická metoda*** může ale deklarovat zcela odlišný typový parametr, který není závislý na typovém parametru třídy. Navíc je možné deklarovat generickou metodu i ve třídě, která sama není generická.

Tato konstrukce bude velmi užitečná, jak uvidíme později, zvláště u statických metod. Ty totiž velmi často vykonávají operace nezávisle na jakékoliv třídě, pouze jsou v určité třídě zapouzdřeny. Generickou metodou může ale být i jakákoliv instanční metoda.

1.4.1 Deklarace generické metody

Deklaraci generické metody si ukážeme na následujícím příkladě. Půjde o rozšíření třídy `AbstractZasobnik` z [Příklad 1.9] v kapitole [1.2.3/9]. Tuto třídu obohatíme o statickou generickou metodu `prevedDoSeznamu()`, která bude objekty v zásobníku převádět do určeného seznam typu `List`. Tato metoda bude pracovat zcela nezávisle na ostatních prvcích třídy `AbstractZasobnik`.

Příklad 1.11: `AbstractZasobnik` **– Deklarace generické metody**

```
import java.util.*;

public abstract class AbstractZasobnik<T> implements Zasobnik<T> {
    ...

    public static <T> void prevedDoSeznamu(AbstractZasobnik<T> z,
                                           List<T> l) {
        T p = z.odebrat();
        while (p != null) {
            l.add(0, p);
            p = z.odebrat();
        }
```

```
    }
}
```

Význam klíčových slov `public` a `static` je nám velmi dobře známý. Poté následuje seznam typových parametrů se kterými bude metoda pracovat. Další v pořadí jsou uvedeny návratový typ metody, název metody a formální parametry metody. Nakonec pochopitelně nesmí chybět její tělo.

Záhlaví metody můžeme číst: „Metoda je veřejná, statická, generická (používá typový parametr `T`) a vrací odkaz na seznam typu `List<T>`, který uchovává prvky onoho typu `T`."

Poznámka:

Všimněme si rozdílu v zápisu typových parametrů. U generické třídy je uveden seznam typových parametrů **za jménem třídy**. U generických metod je to **před jménem třídy** a před uvedením jejího návratového typu. To má své dobré opodstatnění. Abychom mohli na pozici návratového typu metody použít typový parametr, je nutné uvést seznam typových parametrů dříve.

■

1.4.2 Použití generické metody

Použití metody `prevedDoSeznamu()` ukazuje následující testovací scénář. Pro tento test si vytvoříme třídu `ZasobnikTest`, která bude obsahovat pouze metodu `main()`.

V tomto upraveném příkladě musíme ještě do deklarace třídy `Zasobnik2` připsat, že jde o podtřídu třídy `AbstractZasobnik`. To proto, že parametr metody `prevedDoSeznamu()` očekává objekt třídy `AbstractZasobnik`, nebo objekt jakékoliv její podtřídy, a my chceme tuto metodu použít pro převod zásobníku typu `Zasobnik2`.

```
public class Zasobnik2<T> extends AbstractZasobnik<T> { ... }
```

Příklad 1.12: `ZasobnikTest` – Použití generické metody

```
import java.util.*;

public class ZasobnikTest {
    public static void main(String[] args) {
        Zasobnik2<String> zasobnik = new Zasobnik2<String>(4);

        zasobnik.vlozit("slovo1");
        zasobnik.vlozit("slovo2");
        zasobnik.vlozit("slovo3");
        zasobnik.vlozit("slovo4");

        List<String> l = new LinkedList<String>();
        Zasobnik2.<String>prevedDoSeznamu(zasobnik, l);

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

Výstup programu:

```
[slovo1, slovo2, slovo3, slovo4]
```

Při volání generické metody postupujeme zcela stejně, jako při volání kterékoliv běžné metody. Pouze s tím rozdílem, že těsně před název volané metody uvedeme seznam typových parametrů.

Poznámka:

Všimněte si podobnosti: Při volání generické metody je seznam jejích typových parametrů uveden **před jejím názvem** stejně tak, jako v její deklaraci. V případě generické třídy je při volání jejího konstruktoru seznam typových parametrů **za jejím názvem**, stejně tak jako v deklaraci této třídy.

■

Zajímavé je to, že při volání generické metody nemusíme vždy přímo uvádět seznam typových parametrů, pokud si je ovšem může kompilátor odvodit z typů zadaných parametrů. Tak tomu může být i v našem příkladě. Klidně jsme mohli napsat jen:

```
Zasobnik2.prevedDoSeznamu(zasobnik, l);
```

a vše by proběhlo v pořádku. To je možné díky tomu, že parametry této metody jsou typů `AbstractZasobnik<T>` a `List<T>` a kompilátor ví, že zadaný zásobník je typu `Zasobnik2<String>` a zadaný seznam je typu `List<String>`. Dále ví, že typ `T` musí být u obou parametrů shodný. V našem případě jde u obou parametrů o typ `String`.

Vidíme, že i v tomto případě je z hlediska typové kontroly použití generické metody velmi vhodné. Pokud např. zkusíme provést následující změnu v našem testovacím prográmku, zjistíme, že se kompilátor opravdu ošálit nenechá.

```
List<Integer> i = new LinkedList<Integer>();
Zasobnik2.<String>prevedDoSeznamu(zasobnik, i);
```

Kompilátor zjistí, že parametry `zasobnik` a `i` jsou objekty sice správných tříd, ale s rozdílnými typovými parametry (`String` a `Integer`). A to je chyba, protože u obou parametrů musejí být stejné. Dojde tedy k chybě již v době překladu.

1.5 Ohraničení typových parametrů

Vraťme se ještě jednou k příkladu generické třídy `Dvojice` (viz [Příklad 1.6] kapitola [1.2.1/7])

```
public class Dvojice<A, B> {
    ...

    public GenerickaTrida(A prvni, B druha) { ... }

    ...
}
```

V tomto případě jsou typové parametry `A` a `B` zcela neomezené. Při vytváření objektu takovéto třídy můžeme jako typové parametry použít zcela libovolné třídy. To ovšem nemusí vždy odpovídat našim konkrétním požadavkům. V některých případech můžeme chtít nějakým způsobem **omezit množinu tříd** použitelných jako parametr generické třídy. Jinými slovy: chceme typový parametr `A`, resp. `B`, ***ohraničit***.

Jiným případem může být typ parametru metody, který definujeme jako generický typ s využitím zástupného symbolu. Pokud bychom například napsali metodu, která vypíše všechny prvky libovolné kolekce, vypadalo by to asi takto:

```
public static void vypisKolekci(Collection<?> c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

Jako parametr `c` metody `vypisKolekci()` může být použita libovolná kolekce obsahující objekty libovolných tříd. To jsme určili zástupným symbolem `?`. Někdy ale můžeme požadovat, aby parametrem `c` mohla být sice libovolná kolekce, ale obsahující objekty jen některých typů. Musíme tedy zástupný symbol `?` nějakým způsobem ohraničit.

1.5.1 Ohraničení shora

V následujícím příkladě budeme chtít deklarovat třídu představující uspořádanou n-tici čísel. V takovém případě je omezující podmínkou to, že všechny prvky n-tice musejí být čísla, na druhou stranu ale nechceme třídu omezovat na použití konkrétního číselného typu. Chceme tedy vytvořit jakési zobecnění třídy, u kterého budeme moci použít námi určený číselný typ. Deklarace takové zobecněné třídy bude vypadat takto:

Příklad 1.13: `Ntice` – Ohraničení shora 1

```
public class Ntice<T extends Number> {
    Number[] hodnoty;

    public Ntice(int pocet) {
        hodnoty = new Number[pocet];
    }

    public T prvek(int index) {
        return (T) hodnoty[index];
    }
    public void zmenit(int index, T hodnota) {
        hodnoty[index] = hodnota;
    }
    public int velikost() {
        return hodnoty.length;
    }

    ...
}
```

Tímto jsme omezili typový parametr `T` a můžeme do ní dosadit pouze třídu `Number` či jakoukoliv její podtřídu. Můžeme tedy snadno vytvořit n-tici celočíselných hodnot, racionálních hodnot nebo jakýchkoliv jiných.

```
import java.math.*;

public class Ntice<T extends Number> {
    ...

    public void vytvoreniInstance() {
        Ntice<Integer> i = new Ntice<Integer>(5);
        Ntice<Double> d = new Ntice<Double>(7);
        Ntice<BigDecimal> b = new Ntice<BigDecimal>(4);
    }
```

```
    ...
}
```

V tomto příkladu jsme viděli nové použití klíčového slova `extends`. To zde ovšem nemá nic společného s dědičností tříd. Zápis `<T extends Number>` můžeme číst jako omezující podmínku: `T` musí rozšiřovat `Number`, tzn. třída `T` musí být podtřídou třídy `Number`. Není už ale podmínkou, aby `T` bylo přímým potomkem `Number`. Proto také použitý pojem ***ohraničení shora***.

Poznámka:

Jak vidíme, můžeme použít i typ `BigDecimal`, který není zapouzdřením žádného primitivního typu. Důležité je, že třída `BigDecimal` je podtřídou třídy `Number`. Jen pro úplnost dodejme, že třída `BigDecimal` je součástí balíčku `java.math`. ■

Stejnou syntaxi ohraničení typových parametrů můžeme použít i u deklarace proměnných, popř. u deklarace typů parametrů metod.

```
import java.math.*;

public class Ntice<T extends Number> {
    ...

    public void vytvoreniInstance() {
        ...

        Ntice<? extends Number> n1, n2, n3, n4;
        n1 = new Ntice<Integer>(5);
        n2 = new Ntice<Double>(7);
        n3 = new Ntice<BigDecimal>(4);
        //n4 = new Ntice<Character>(4);     // !!! chyba !!!
    }

    ...
}
```

Víme, že třída `Character` není podtřídou třídy `Number`, potom typ `Ntice<Character>` není podtypem typu `Ntice<? extends Number>`.

Poznámka:

Možná vás zmátlo označení ***třída/podtřída*** a ***typ/podtyp***. Tato dvě různá označení dostala s příchodem generických typů různý význam. Pokud budeme používat označení třída/podtřída, budeme mít na mysli konkrétní třídy a jejich vzájemný vztah co se týče dědičnosti. Označením typ/podtyp budeme myslet pouze generický typ proměnné.

- `LinkedList<Integer>` a `LinkedList<Number>` jsou rozdílné typy, ale stejné třídy.
- `LinkedList<Integer>` není podtypem typu `LinkedList<Number>`.
- `LinkedList<Integer>` je podtypem typu `LinkedList<?>`.
- `LinkedList<Integer>` je podtypem typu `LinkedList<? extends Number>`.

■

Příkladem použití ohraničení typových parametrů u deklarace typu parametru metody může být metoda, která bude na obrazovku vypisovat n-tici složenou z prvků libovolných číselných typů. Tuto metodu deklarujeme jako statickou a přidáme do naší třídy `Ntice`.

Příklad 1.14: `Ntice` – Ohraničení shora 2

```
public class Ntice<T extends Number> {
    ...

    public static void vypisNtici(Ntice<? extends Number> n) {
        System.out.print("[");
        for (int i = 0; i < n.velikost()-1; i++) {
            System.out.print(n.prvek(i) + ", ");
        }
        System.out.print(n.prvek(n.velikost()-1));
        System.out.println("]");
    }

    ...
}
```

V záhlaví metody opět vidíme ohraničení `Ntice<? extends Number>`. Do parametru `n` můžeme tedy dosadit n-tici jakýchkoliv číselných hodnot. Následujícími příkazy tedy můžeme vytvořit např. objekt třídy `Ntice<Integer>` a nechat jej vypsat.

```
public class Ntice<T extends Number> {
    ...

    public static void main(String[] args) {
        Ntice<Integer> n = new Ntice<Integer>(3);

        n.zmenit(0, new Integer(8));
        n.zmenit(1, new Integer(5));
        n.zmenit(2, new Integer(16));

        Ntice.vypisNtici(n);
    }

    ...
}
```

Výstup programu:

```
[8, 5, 16]
```

Stejně tak bychom mohli použít metodu `Ntice.vypisNtici()` pro objekt typu `Ntice<Double>` atp.

1.5.2 Ohraničení zdola

V předchozí kapitole jsme si ukázali, jak ohraničit typový parametr `T` shora. Obdobným způsobem můžeme definovat ohraničení zdola. Místo klíčového slova `extends` použijeme klíčové slovo `super`. ***Ohraničení zdola*** lze ovšem použít pouze ve spojení se zástupným symbolem `<?>`. Nemůžeme tedy přímo parametr generické třídy ohraničit zdola. Příkladem může být rozšíření třídy `Ntice` o metodu `prevedDoKolekce()`. Tato metoda přidá všechny složky n-tice do kolekce určené parametrem metody.

Příklad 1.15: `Ntice` – Ohraničení zdola 1

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

JIHOČESKÁ UNIVERZITA
V ČESKÝCH BUDĚJOVICÍCH
PEDAGOGICKÁ FAKULTA
ÚSTŘEDNÍ KNIHOVNA -8-
Husova 458, 370 05 České Budějovice

```
public class Ntice<T extends Number> {
    ...

    public void prevedDoKolekce(Collection<Number> c) {
        for (int i = 0; i < hodnoty.length; i++) {
            c.add(hodnoty[i]);
        }
    }

    ...
}
```

Pokud bychom ovšem napsali metodu `prevedDoKolekce()` takto, musela by kolekce určená parametrem `c` obsahovat pouze objekty typu `Number`.

```
public void prevedDoKolekce(Collection<?> c) { ... }
```

Použijeme-li tento zápis, bude moci být parametrem `c` určena zcela libovolná kolekce. To by ovšem mohla být např. kolekce typu `ArrayList<String>` a při pokusu o vložení objektu třídy `Number` do kolekce typu `ArrayList<String>` by pochopitelně došlo k chybě. Správné řešení je tedy použití typového parametru ohraničeného zdola.

Příklad 1.16: `Ntice` – Ohraničení zdola 2

```
import java.util.*;

public class Ntice<T extends Number> {
    ...

    public void prevedDoKolekce2(Collection<? super Number> c) {
        for (int i = 0; i < hodnoty.length; i++) {
            c.add(hodnoty[i]);
        }
    }

    ...
}
```

Tímto máme zajištěno, že parametr `c` může určovat libovolnou kolekci, která uchovává objekty libovolné třídy, které je třída `Number` potomkem. Můžeme tedy použít např. kolekci typu `LinkedList<Object>`.

Pro vyzkoušení si můžeme sestavit následující testovací metodu a přidat její zavolání do metody `main()` třídy `Ntice`:

Příklad 1.17: `Ntice` – Ohraničení zdola 3

```
import java.util.*;

public class Ntice<T extends Number> {
    ...

    public static void testPrevodu() {
        LinkedList<Object> l = new LinkedList<Object>();
        Ntice<Integer> n = new Ntice<Integer>(3);

        n.zmenit(0, new Integer(8));
        n.zmenit(1, new Integer(5));
        n.zmenit(2, new Integer(16));
```

```
        n.prevedDoKolekce2(l);
        System.out.println(l);
    }

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

        Ntice.testPrevodu();
    }
}
```

Výstup programu:

```
[8, 5, 16]
```

Trochu odlišný způsob použití ohraničení typového parametru zdola si ukážeme na příkladě generické metody `sort()` ze třídy `Collections`. Jde o statickou metodu, která seřadí zadaný seznam s využitím přirozeného řazení prvků. Deklarace této metody vypadá následovně:

```
public static <T extends Comparable<? super T>> void
    sort(List<T> seznam) { ... }
```

V tomto případě je typový parametr `T` omezen shora. Musí jít o třídu, která je podtřídou rozhraní `Comparable`. Jinými slovy: třída `T` musí implementovat rozhraní `Comparable`. Jenomže i rozhraní `Comparable` je generický typ a musíme u něj také uvést typový parametr. Ten už je ovšem ohraničen zdola. Pokud bychom uvedli pouze `<T extends Comparable<T>>`, musela by přímo třída `T` implementovat rozhraní `Comparable`. Velmi často ale implementuje toto rozhraní nějaká rodičovská třída a my jeho implementaci dědíme. Musíme proto uvést, že rozhraní `Comparable` může implementovat jakákoliv *„nadtřída“* třídy `T`.

1.5.3 Vícenásobné ohraničení

Typový parametr nemusí mít ohraničení určené jen jednou třídou. ***Vícenásobné ohraničení*** je užitečné, pokud např. potřebujeme určit, že třída určená typovým parametrem generické třídy implementuje dvě různá rozhraní.

```
class Kolekce<T extends Comparable<? super T> & Serializable>
```

Tento zápis vyjadřuje, že typovým parametrem `T` může být jakákoliv třída, která implementuje rozhraní `Comparable` **a zároveň** implementuje rozhraní `Serializable`. Symbol `&` má tedy význam **„a zároveň“**.

Upozornění:

Použití vícenásobného ohraničení má některá specifika:

- typový parametr generické třídy či generické metody **může mít více ohraničení shora**
- vícenásobné ohraničení **nelze použít pro ohraničení zdola**
- zástupný symbol `<?>` může mít jen **jedno ohraničení, a to shora nebo zdola**

■

Poznámka:

Jakákoliv proměnná typu `T` může přistupovat k veřejným metodám a položkám jakékoliv třídy či rozhraní, které určují ohraničení tohoto typového parametru shora. To proto, že

ohraničením shora sdělujeme, že parametr `T` představuje třídu, která je podtřídou všech ohraničujících tříd či rozhraní.

■

1.5.4 Jak číst některé složitější zápisy generických typů

V této kapitole jsme se zatím věnovali způsobu ohraničení, neboli vymezení, množiny tříd, které můžeme použít jako hodnotu typového parametru generické třídy nebo generické metody. Podrobně jsme popsali syntaxi a význam ohraničení typových parametrů. V závěru této kapitoly si uvedeme, nebo možná jen zopakujeme, přehled některých **složitějších zápisů** generických typů s použitím ohraničení typových parametrů.

1.5.4.1 Deklarace generické třídy

```
class TridaA<T extends Comparable<T>> {}
```

Typovým parametrem `T` může být jakákoliv třída, která přímo implementuje rozhraní `Comparable`.

```
class TridaB<T extends Comparable<? super T>> {}
```

Typovým parametrem `T` může být jakákoliv třída, která implementuje rozhraní `Comparable`, nebo jeho implementaci dědí od jakékoliv své rodičovské třídy.

```
class TridaC<T extends Cloneable & Serializable> {}
```

Typovým parametrem `T` může být jakákoliv třída, která implementuje obě rozhraní `Cloneable` a `Serializable`.

```
class TridaD<X, Y extends X> {}
```

Typovým parametrem `X` může být jakákoliv třída. Typovým parametrem `Y` může být jakákoliv třída, která je podtřídou třídy `X`.

```
class TridaE<T extends Object> {}
```

Typovým parametrem `T` nemůže být rozhraní.

```
class TridaF<X, Y, Z, V, W> {}
```

Jakákoliv generická třída může mít bezpočet typových parametrů.

1.5.4.2 Deklarace generické metody

```
public <T extends Number> void metoda1(T p) {}
```

Parametrem `p` může být objekt třídy, která je podtřídou třídy `Number`.

```
public <T> void metoda2(Collection<? super T> c, T p) {}
```

Parametrem `c` může být jakákoliv kolekce, která uchovává objekty třídy `T` nebo jakékoliv její rodičovské třídy. Parametrem `p` je pak objekt právě třídy `T`.

```
public <T extends Comparable<? super T>> void metoda3(List<T> l, T p)
```

Parametrem `l` může být libovolný seznam, který uchovává objekty jakékoliv třídy implementující rozhraní `Comparable`. Parametrem `p` je pak objekt takovéto třídy.

```
public <T extends Number> void metoda4(Collection<? super T> c, T p)
```

Parametrem `c` může být jakákoliv kolekce, která uchovává objekty třídy `T`, popř. kterékoliv její rodičovské třídy, která je zároveň podtřídou třídy `Number`. Parametrem `p` je pak objekt právě třídy `T`.

1.6 Pole a generické typy

Práce s poli ve spojení s generickými typy má některá omezení. V této kapitole si na příkladech ukážeme dvě základní omezení, která nemusejí být na první pohled zřejmá. První z nich je problém **vytvoření pole, jehož typem je generický typ**. Druhým problémem je **inicializace pole pomocí typového parametru**.

1.6.1 Typem pole nemůže být generický typ

Ve většině případů můžeme použít generický typ všude tam, kde můžeme použít jakýkoliv jiný typ. Výjimkou je ale deklarace pole. Typem pole může být primitivní typ, třída, abstraktní třída nebo rozhraní. Nemůžeme ale použít generický typ.

Příklad 1.18: `Pole` – Pole 1

```
import java.util.*;

public class Pole {
    public static void main(String[] args) {
        List<Double>[] pole1 =
                    new LinkedList<Double>[18];  // !!! chyba !!!
        List<?>[] pole2 =
                    new LinkedList<Double>[18];  // !!! chyba !!!
    }
}
```

To, že takové pole nemůžeme vytvořit, ale neznamená, že ho nemůžeme deklarovat. Není problém deklarovat proměnnou, která ukazuje na pole, jehož typem je generický typ. Inicializaci pole však musíme provést trochu jinak. Nesmíme také zapomenout na vytvoření jednotlivých seznamů v poli.

```
import java.util.*;

public class Pole {
    public static void main(String[] args) {
        ...

        List<String>[] pole3 = new ArrayList[3];
        for (int i = 0; i < 6; i++) {
            pole3[i] = new LinkedList<String>();
            pole3[i].add("řetězec " + i);
        }
    }
}
```

Tento příkaz sice vyvolá při překladu upozornění, že provádíme neošetřené operace, ale proběhne bez chyby. Typem pole zde není generický typ, ale pouze typ `ArrayList`. Ovšem proměnnou `pole3` můžeme používat pouze v souladu s její deklarací. Tzn. metody, které budeme volat prostřednictvím proměnné `pole3` budou v tomto případě očekávat namísto typového parametru typ `String`. To si můžeme snadno vyzkoušet.

```
import java.util.*;

public class Pole {
    public static void main(String[] args) {
        ...

        pole3[2].add("další řetězec");
        //pole3[1].add(new Integer(18));   // !!! chyba !!!

        System.out.println(pole3[2]);
    }
}
```

Výstup programu:

```
[řetězec 2, další řetězec]
```

Z tohoto příkladu je zřejmé, že práce s poli instancí generických typů není příliš pohodlná a ani typově bezpečná. Pokud ale budeme v nějakém konkrétním případě trvat na tom, že potřebujeme mít několik seznamů v jedné proměnné, je nejsnazším řešením nahradit pole kolekcí typu `ArrayList`.

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

Třída `ArrayList` v podstatě pole zapouzdřuje, ale navíc udržuje naprostou typovou bezpečnost.

1.6.2 Typovým parametrem nemůžeme inicializovat pole

Stejně tak jako nemůžeme instanciovat typový parametr, nemůžeme typovým parametrem inicializovat ani pole.

Příklad 1.19: `Pole` – Pole 2

```
public class Pole<T> {
    private T[] pole;

    public Pole() {
        pole = new T[18];     // !!! chyba !!!
    }

    ...
}
```

To je způsobeno tím, že v době překladu zdrojového kódu nemůže kompilátor vědět, jaká třída bude obsahem typového parametru `T`. Kompilátor by nevěděl, jaký má vygenerovat bajtkód a jak velký prostor má rezervovat v paměti pro objekt `pole`.

Můžeme ale pole inicializovat jako pole objektů typu `Object` a **explicitně jej přetypovat** na pole typu `T`. S takovým polem můžeme bez problémů provádět libovolné operace, jak uvádí následující příklad.

```
public class Pole<T> {
    private T[] pole;

    public Pole(int vel) {
        pole = (T[]) new Object[vel];
    }

    public void nastav(int i, T t) {
        pole[i] = t;
    }

    public String toString() {
        String s = "[";
        for (int i = 0; i < pole.length-1; i++) {
            s = s + pole[i].toString() + ", ";
        }
        s = s + pole[pole.length-1] + "]";
        return s;
    }

    public static void main(String[] args) {
        Pole<Integer> p = new Pole<Integer>(12);

        for (int i = 0; i < 12; i++) {
            p.nastav(i, new Integer(i+100));
        }

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

Výstup programu:

```
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111]
```

Celý program pracuje bezchybně. Pouze při překladu obdržíme hlášení, že provádíme neošetřené operace *‚Pole2.java uses unchecked or unsafe operations'*. Ačkoli je toto řešení bezchybné, nelze je považovat za ideální. Nejjednodušší a nejelegantnější by bylo nahradit pole typu `T` kolekcí typu `ArrayList<T>`.

1.7 Dědičnost a generické typy

Dědičnost je jedním ze základních principů objektově orientovaného programování. Dalo by se i tvrdit, že objektově orientované programovací jazyky jsou založeny na využívání dědičnosti a polymorfizmu. Význam dědičnosti spočívá v tom, že můžeme napsat třídu, která rozšiřuje možnosti a dovednosti jiné třídy, popř. je dotváří nebo specializuje. V této kapitole se ale nebudeme věnovat kompletnímu výkladu principu dědičnosti, ukážeme si na modelových příkladech použití dědičnosti ve spojení s generickými typy.

V této chvíli už veškerá syntaktická pravidla a význam použití generických typů známe. Tato kapitola bude tedy jen jakýmsi přehledem základních typů dědičnosti, jako jsou **dědění od běžné třídy, dědění od generické třídy** či **implementace rozhraní**.

Nebudeme se speciálně věnovat **dědění od abstraktní třídy**. Samotné dědění od abstraktní třídy patří k elementárním znalostem programovacího jazyka Java. A co se týče abstraktních generických tříd, je použití dědičnosti obdobné, jako u běžných tříd.

1.7.1 Třída vzniklá děděním od jiné třídy

1.7.1.1 Generická třída jako potomek běžné třídy

Jde o nejjednodušší případ, a sice když generická třída vzniká děděním od běžné třídy. V tomto případě totiž vůbec nedojde k žádnému propojení typových parametrů. V nově vzniklé třídě můžeme libovolně překrývat metody rodičovské třídy. Uplatnění typových parametrů se týká pouze nových metod či proměnných třídy.

Příklad 1.20: `BeznyRodic` **– Dědičnost 1**

```
public class BeznyRodic {
    protected int id;

    public BeznyRodic(int id) {
        this.id = id;
    }
    public int id() {
        return id;
    }
    public String toString() {
        return "[" + id + "]";
    }
}

import java.util.*;

public class Potomek1<T> extends BeznyRodic {
    private T polozka;

    public Potomek1(int id, T p) {
        super(id);
        polozka = p;
    }
    public T zmenit(T nove) {
        T puvodni = polozka;
        polozka = nove;
        return puvodni;
    }
    public String toString() {
        return polozka.toString() + " " + super.toString();
    }
}
```

Ve třídě `Potomek1` přibyla pouze nová položka `polozka`. Jak vidíme, překrytí metody z rodičovské třídy nepřináší žádné omezení. Typový parametr `T` je použit pouze u nových prvků třídy `Potomek1`.

1.7.1.2 Generická třída jako potomek generické třídy

Tento případ dědičnosti je velmi často používaný. Stejným způsobem, jako uvádí následující příklad, jsou implementovány i třídy ze standardních knihoven JDK 5.0 pro práci s kolekcemi.

Příklad 1.21: `GenerRodic` **– Dědičnost 2**

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

```
public class GenerRodic<T> {
    protected T polozka;

    public GenerRodic(T p) {
        polozka = p;
    }
    public void nastav(T nove) {
        polozka = nove;
    }
    public boolean pridejDoKolekce(Collection<? super T> c) {
        return c.add(polozka);
    }
    public String toString() {
        return polozka.toString();
    }
}

public class Potomek2<T extends Number> extends GenerRodic<T> {

    public Potomek2(T p) {
        super(p);
    }
    public T vrat() {
        return polozka;
    }
}
```

Jak vidíme, ve zděděné třídě jsme přidali ohraničení pro typový parametr `T` a navíc metodu `vrat()` pro získání položky `polozka`. Za povšimnutí zde stojí zápis záhlaví třídy `Potomek2`. Zde deklarujeme dědění od generické třídy `GenerRodic`. Jako typový parametr rodičovské třídy používáme přímo typový parametr dceřiné třídy. To je velice užitečné, jelikož takto ohraničujeme typový parametr i u všech proměnných či metod rodičovské třídy, ke kterým můžeme přistupovat ze zděděné třídy. Pokud vytvoříme jinou třídu zděděnou od třídy `GenerRodic`, může být ohraničení zmíněného typového parametru zase jiné.

1.7.1.3 Běžná třída jako potomek generické třídy

Jde v podstatě o podobný případ, jako v [Příklad 1.21] v kapitole [1.7.1.2/24]. Pokud ovšem dceřiná třída není deklarována jako generická, musíme v její deklaraci uvést jako typový parametr rodičovské třídy nějakou konkrétní třídu.

Příklad 1.22: `Potomek3` – Dědičnost 3

```
public class Potomek3 extends GenerRodic<String> {
    public Potomek3() {
        super("text");
    }
}
```

1.7.2 Implementace rozhraní v generické třídě

Implementace rozhraní nemá ve spojení s generickými typy žádné omezení. S jeho praktickým použitím jsme se již setkali v příkladech [Příklad 1.8] a [Příklad 1.9] v kapitolách [1.2.2/9] a [1.2.3/9]. Syntaktická pravidla jsou obdobná jako v případě použití dědičnosti (viz kapitola [1.7.1/24]). Zásadní rozdíl, který je patrný pouze v záhlaví deklarace třídy, je stejný jako rozdíl mezi dědičností a implementací rozhraní v běžné třídě. Týká se pouze použití klíčových slov `extends` a `implements`.

Je samozřejmě možné v generické třídě implementovat jak běžné rozhraní, tak generické rozhraní. Je ale také možné generické rozhraní implementovat v běžné třídě. Potom musíme, stejně jako v případě dědění běžné třídy od generické třídy, jako typový parametr implementovaného rozhraní uvést nějakou konkrétní třídu.

1.7.2.1 Implementace více rozhraní v jedné třídě

Jediným případem, při kterém může nastat nějaký zásadní problém, je implementace více rozhraní v jedné třídě. Je v tomto případě jedno, zda jde o generickou třídu či nikoliv. Pokud implementujeme v jedné třídě více různých generických rozhraní, je vše v pořádku. Nelze však implementovat **jedno generické rozhraní v jedné třídě dvakrát**, pokaždé s jiným typovým parametrem.

```
// nepřípustná implementace rozhraní
class Trida<T> implements Comparable<T>, Comparable<String> { ... }
```

V kapitole [1.8.1/27] se dozvíme, proč toto není možné.

1.7.3 Dědění třídy a současná implementace rozhraní

I toto je samozřejmě možné. Nemá už ovšem smysl vyjmenovávat všechny možné případy, které mohou nastat. Veškerá pravidla jsou stejná, jako v předchozích částech kapitoly [1.7/23].

Dědění třídy se současnou implementací rozhraní je samozřejmě možné jak pro běžnou třídu, tak pro generickou třídu. Je také možné v jakékoliv děděné třídě implementovat více rozhraní, z nichž některá jsou generická a některá nikoliv. V jakékoliv takto zděděné třídě musíme pouze dbát na správnou implementaci metod deklarovaných ve všech implementovaných rozhraních.

1.8 Překlad zdrojového kódu využívajícího generické typy

Jednou z nejsilnějších stránek konstrukce generických typů je **způsob jejich překladu**, neboli samotná implementace generických typů do jazyka Java. Kompilátor při překladu zdrojového kódu **vypustí většinu informací** o typových parametrech a o jejich ohraničení. Ty už tedy v bajtkódu nejsou. Kompilátor nahradí všechna použití typového parametru vhodným přetypováním. Zrovna tak, jak jsme to museli psát ručně, dokud jsme generické typy nemohli využívat. Neohraničené typové parametry nahradí typem `Object` a přidá přetypování ta onen typový parametr. Pokud je typový parametr ohraničen shora, nahradí ho právě tímto ohraničením. Pokud je použito vícenásobné ohraničení, kompilátor nahradí typový parametr třídou, která je v seznamu ohraničení uvedená jako první.

Jak je ale potom možné udržet jazyk tak robustní a typově bezpečný? To je možné proto, že kompilátor sám je schopen zkontrolovat typovou správnost zdrojového kódu. Tuto kontrolu mu umožňuje právě konstrukce generických typů.

Poznámka:

Následující metoda je sice deklarována jako generická, použití generických typů zde ovšem nemá žádný smysl.

Příklad 1.23: `Pretypovani` **– Přetypování**

```
public class Pretypovani {
    public static <T> T pretypuj(Object o) {
        return (T) o;
```

```
    }
    public static void main(String[] args) {
        Object o = "text";
        String s = Pretypovani.<String>pretypuj(o);
        System.out.println(s);
    }
}
```

Výstup programu:

```
text
```

Poznámka:

Tato metoda provádí pouze explicitní přetypování. Kompilátor ale neudělá nic jiného, než že typový parametr `T` nahradí typem `Object` a opět přidá přetypování. Takovéto využití generických typů v žádném případě nezvýší typovou bezpečnost programu.

■

1.8.1 Některé nepovolené operace

Důsledky implementace generických typů a jejich **překladu** jsou velmi významné, ale někdy také troch matoucí. Na to jsme narazili už když jsme si říkali, že proměnné `List<String> seznamSlov` a `List<Integer> seznamCisel` jsou proměnné různých typů, tyto typy však sdílí společnou třídu.

Příklad 1.24: Sdílení společné třídy různými generickými typy

```
List<String> seznamSlov = new LinkedList<String>();
List<Integer> seznamCisel = new LinkedList<Integer>();

System.out.println(seznamSlov.getClass());
System.out.println(seznamCisel.getClass());
```

Výstup programu:

```
class java.util.LinkedList
class java.util.LinkedList
```

Tento způsob překladu generických typů má několik důsledků. **Nemůžeme například implementovat ve vlastní třídě jedno rozhraní dvakrát**, pokaždé s jiným typovým parametrem.

```
class Trida<T> implements Comparable<T>, Comparable<String> {
    ...
}
// chybná deklarace třídy
```

`Comparable<T>` a `Comparable<String>` jsou sice různé typy, ale stejné třídy. A my již víme, že kompilátor při překladu nahrazuje generické typy vhodným přetypováním. Nic nám sice nebrání deklarovat ve třídě `Trida` metou `compareTo()` dvakrát, pokaždé s využitím jiného typového parametru. Kompilátor ale nahradí typové parametry typem `Object` a u volání těchto metod přidá přetypování. Během překladu by se ve třídě `Trida` tedy objevila stejná metoda `compareTo()` dvakrát. A to není možné.

Jiným příkladem nepovolené operace může být např. **vytvoření objektu nebo pole použitím typového parametru.**

```
T objekt = new T();
T[] pole = new T[18];
```

V době překladu nemůže kompilátor vědět, jaká třída bude použita namísto `T` a jaký má tedy vygenerovat bajtkód. Objekt typu `T` můžeme vytvořit pouze tehdy, když máme odkaz na speciální objekt typu `Class<T>`. Více informací o použití třídy `Class` najdete v kapitole [9.1/69]. Existuje i způsob, jak vytvořit pole typu `T[]`. K tomu slouží statická metoda `newInstance()` třídy `java.lang.reflect.Array`, jejímiž parametry jsou opět objekt typu `Class<T>` a délka pole. Více o použití této metody naleznete v dokumentaci k Java API.

2 Statický import

V této kapitole se seznámíme s jedním drobným rozšířením jazyka Java. ***Statické importy*** nejsou nijak revoluční změnou, zavedeny byly se snahou zkrácení některých zápisů zdrojového kódu. Ne vždy, jak uvidíme dále, nám ale statické importy usnadní práci. Jejich použití je někdy **kritizováno**, protože může činit zdrojový kód spíše nepřehledným než jednodušším. Záleží tedy na programátorovi, zda použije statický import jen tam, kde je to opravdu **výhodné**.

2.1 Použití statického importu

Doposud jsme používali příkaz `import` proto, abychom při každém použití nějaké třídy nemuseli vždy uvádět celý název balíčku, ve kterém se třída nachází. V příkazu `import` jsme uvedli konkrétní třídu (např. `import java.io.File;`) a mohli jsme s touto třídou pracovat vždy jen s uvedením jejího názvu. Popřípadě jsme mohli použít v příkazu import místo názvu konkrétní třídy hvězdičku (např. `import java.io.*;`), což znamenalo importovat všechny třídy daného balíčku.

V nové verzi Javy byl tento příkaz rozšířen i pro **importování statických členů tříd**. Těmi mohou být statické proměnné, konstanty, metody, dokonce i vnitřní statické třídy. S použitím statického importu lze takovéto statické členy používat v programu **bez uvedení třídy**, ve které jsou deklarovány.

Na následujících dvou příkladech můžete pozorovat rozdíl mezi původním přístupem a přístupem využívajícím statický import.

Příklad 2.1: `Puvodni` – Ukázka bez použití statického importu

```
import javax.swing.*;

public class Puvodni {
    public static void main(String[] args) {
        String s;
        double r = 0;

        s = JOptionPane.showInputDialog(null,
                "Zadej poloměr kruhu: ", "Složité počty",
                JOptionPane.QUESTION_MESSAGE);
        if (s == null) {
            JOptionPane.showMessageDialog(null,
                "Když nechceš, tak nech být...",
                "", JOptionPane.ERROR_MESSAGE);
            return;
        }

        try {
            r = Double.valueOf(s);
        }
        catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null,
                "Pro zadanou hodnotu je výpočet příliš složitý. :-)",
                "Upozornění...", JOptionPane.WARNING_MESSAGE);
```

```
            return;
        }

        double obsah = Math.PI * Math.pow(r, 2);
        JOptionPane.showMessageDialog(null,
                "Obsah kruhu o poloměru " + r + " je " + obsah + ".",
                "Výsledek...", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Příklad 2.2: `StatickyImport` – Použití statického importu

```
import javax.swing.*;
import static javax.swing.JOptionPane.*;
import static java.lang.Math.*;
import static java.lang.Double.valueOf;

public class StatickyImport {
    public static void main(String[] args) {
        String s;
        double r = 0;

        s = showInputDialog(null, "Zadej poloměr kruhu: ",
                "Složité počty", QUESTION_MESSAGE);
        if (s == null) {
            showMessageDialog(null, "Když nechceš, tak nech být...",
                "", ERROR_MESSAGE);
            return;
        }

        try {
            r = valueOf(s);
        }
        catch (NumberFormatException e) {
            showMessageDialog(null,
                "Pro zadanou hodnotu je výpočet příliš složitý. :-)",
                "Upozornění...", WARNING_MESSAGE);
            return;
        }

        double obsah = PI * pow(r, 2);
        showMessageDialog(null,
            "Obsah kruhu o poloměru " + r + " je " + obsah + ".",
            "Výsledek...", INFORMATION_MESSAGE);
    }
}
```

Jak vidíme na druhém příkladě, použití statického importu je stejné jako použití běžného importu. V deklaraci importu je navíc za klíčovým slovem `import` klíčové slovo `static`. Statický import lze použít jak pro jednotlivé statické členy (např. `import static java.lang.Double.valueOf;`), tak pro všechny statické členy určité třídy (např. `import static javax.swing.JOptionPane.*;`). Při volání importované statické metody (popř. při použití importované statické proměnné) už nemusíme uvádět název třídy.

Všimněte si, že při importu statické metody neuvádíme žádné závorky, pouze přesný název. Navíc, pokud použijeme statický import s uvedením názvu statického členu třídy, **importujeme zároveň proměnné i metody**, které mají tento společný název. Statický import totiž nerozlišuje, zda jde o proměnnou či metodu, importuje pouze název jakéhokoliv statického členu třídy. Metodu, která je v jedné třídě několikrát přetížena, můžeme tedy importovat jediným příkazem.

Ještě si ale musíme dát pozor na jednu drobnost. Při deklaraci statického importu musíme uvést název třídy, jejíž statické členy importujeme, i **s celým názvem balíčku.** A to i v případě, že importujeme statické členy některé třídy z balíčku `java.lang`.

Dobrá rada:

V žádném případě nepoužívejte statické importy pro všechny statické členy použité ve Vašem programu. Statický import je vhodný pouze pro proměnné či metody, které používáme v programu opravdu často, a navíc víme, co tyto metody dělají. Ne vždy musí být na první pohled patrný význam použití metody bez uvedení příslušné třídy, zvláště v případech, kdy je metoda stejného názvu deklarována v celé řadě různých tříd. V našem příkladě by např. bylo vhodné použít statický import pro metody `showInputDialog()` a `showMessageDialog()`. U těchto metod je zřejmé jejich použití. Zcela nevhodné je použít statický import např. pro metodu `valueOf()` třídy `Double`. V místě volání této metody není na první pohled zřejmé, co vlastně tato metoda provede. ■

2.2 Možný konflikt při použití statického importu

Asi si teď budete myslet, že neexistuje snad žádná programátorská technika, která by fungovala bez jediné výjimky. Ujišťuji Vás, že ve spojení se statickými importy může dojít pouze k jedné možné chybě. Možná, že po pročtení předcházející kapitoly už tušíte, o co půjde.

Jediná kolize, ke které může dojít při použití statických importů, je **nejednoznačnost názvu** použité proměnné či metody.

Představte si, že ve svém programu uvedete import všech statických členů tříd `Integer` a `Double`, a následně někde v programu použijete volání metody `valueOf()`. Jelikož je metoda s názvem `valueOf()` deklarována v obou zmíněných třídách, dojde při překladu takového programu k chybě. Kompilátor totiž **neví, kterou metodu ze které třídy chceme použít.**

Příklad 2.3: `Chyba` **– Konflikt při použití statického importu**

```
import static java.lang.Integer.*;
import static java.lang.Double.*;

public class Chyba {
    public static void main(String[] args)
                                        throws java.io.IOException {

        double d = valueOf(args[0]).doubleValue();  // !!! chyba !!!
        int i = valueOf(args[1]).intValue();        // !!! chyba !!!

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

Pokud ale importujeme zároveň `java.lang.Integer.*` a `java.lang.Double.valueOf`, k žádné kolizi nedojde. Statický import konkrétní proměnné či metody **má přednost.**

3 Automatické zapouzdření primitivních typů

Jak jistě víte, Java není přeci jen čistě objektový jazyk. Z hlediska efektivity používá, stejně jako jiné programovací jazyky, tzv. **primitivní typy**. Java používá čtyři primitivní celočíselné typy `byte`, `short`, `int` a `long`, dva primitivní typy reálných hodnot `float` a `double`, primitivní logický typ `boolean` a primitivní znakový typ `char`. Mnohdy ale potřebujeme hodnoty těchto typů předávat odkazem, nikoliv hodnotou. Proto jsou ve standardní knihovně připraveny objektové (referenční) typy, resp. třídy, které **zapouzdřují hodnoty primitivních typů**. Tyto třídy navíc umožňují provádět s hodnotami primitivních typů nejrůznější operace. Následující tabulka uvádí všechny primitivní typy a jim odpovídající objektové typy.

Tabulka 3.1: Primitivní a objektové typy

Primitivní typ	Třída zapouzdřující primitivní typ
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Novou vlastností jazyka Java 5.0 je možnost **automatického převodu primitivních typů** na třídy, které jejich hodnoty zapouzdřují, a naopak. V praxi to znamená, že na místě, kde je očekáván objektový typ, můžeme použít hodnotu nebo proměnnou primitivního typu. Anebo na místě, kde je očekáván primitivní typ, můžeme použít objekt příslušné třídy, která tento primitivní typ zapouzdřuje.

Podobnou vlastnost můžeme znát např. z programovacího jazyka C#, ve kterém se pro tyto automatické převody používají výrazy ***auto-boxing*** a ***auto-unboxing***. Tato označení převzal i jazyk Java.

3.1 Použití automatického zapouzdření

Asi nejjednodušším příkladem ukazující **automatické zapouzdření** je práce s kolekcemi, které mají uchovávat číselné hodnoty. Pokud jsme dříve chtěli ukládat do nějaké kolekce číselné hodnoty, museli jsme je vždy převést na odpovídající objektové typy a pak až vložit do kolekce. Při jejich odebrání z kolekce jsme naopak museli z těchto objektů nějakým způsobem vydobýt ony číselné hodnoty.

Příklad 3.1: `KolekceCisel` – Práce s kolekcí bez automatického zapouzdření

```
import java.util.*;

public class KolekceCisel {
    public static void main(String[] args) {
        List l = new ArrayList();

        l.add(new Integer(2));
        l.add(new Integer(3));
        l.add(new Integer(4));
        l.add(new Integer(5));

        // ...

        int soucet = 0;

        for (int i = 0; i < l.size(); i++)
            soucet += ((Integer) l.get(i)).intValue();

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

Výstup programu:

```
14
```

Tyto neobratné zápisy můžeme jednoduše opustit. S využitím automatického zapouzdření můžeme převody čísel `2`, `3`, `4` a `5` na typy `Integer` a zpět přenechat na kompilátoru. Musíme mu ale také říct, že metoda `get()` bude vracet typ `Integer`. To je ale logické. Kompilátor přece jinak nemůže vědět, na jaký primitivní typ chceme zrovna převádět typ `Object`. K tomu můžeme velice elegantně použít generické typy.

Příklad 3.2: `KolekceCisel2` – Práce s kolekcí s využitím automatického zapouzdření

```
import java.util.*;

public class KolekceCisel2 {
    public static void main(String[] args) {
        List<Integer> l = new ArrayList<Integer>();

        l.add(2);
        l.add(3);
        l.add(4);
        l.add(5);

        // ...

        int soucet = 0;

        for (int i = 0; i < l.size(); i++)
            soucet += l.get(i);

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

Výstup programu:

```
14
```

V případě vkládání prvků do seznamu kompilátor ví, že metoda `add()` očekává parametr typu `Integer`, je mu tedy jasné, že musí primitivní číselný typ nahradit objektovým typem `Integer`.

V případě odebírání prvků z kolekce kompilátor ví, že metoda `get()` vrací typ `Integer`, také ví, že proměnná `soucet` je typu `int`, je mu tedy jasné, že musí typ `Integer` převést na typ `int`. Jinou možnost nemá.

Toto však není jediný možný případ použití automatického zapouzdření. To lze úplně stejně použít i při běžných **aritmetických operacích**, jako je sčítání, odečítání, násobení, dělení, apod. **Výjimku zde ovšem tvoří operátory porovnání.**

Příklad 3.3: `Porovnani` – Operátory porovnání a automatické zapouzdření

```
public class Porovnani {
    public static void main(String[] args) {
        Integer i1, i2, i3;
        i1 = 86;
        i2 = 326;
        i3 = 326;

        boolean p1 = i1 < i2;
        boolean p2 = i2 == i3;

        System.out.println(p1);
        System.out.println(p2);
    }
}
```

Výstup programu:

```
true
false
```

Jak je vidět, při porovnání proměnných `i1` a `i2` operátorem `<` dojde k automatickému převodu objektových typů na typy primitivní. Porovnání potom proběhne tak, jak předpokládáme. Pokud budeme ale testovat rovnost proměnných `i2` a `i3`, bude situace trochu odlišná. Jelikož operátor rovnosti `==` lze použít i pro objektové typy, kompilátor nemá důvod převádět objekty `i2` a `i3` na primitivní typy. Jenže u objektů slouží tento operátor pro porovnání referencí, nikoliv hodnot obsažených v objektech. A jelikož `i2` a `i3` jsou různé objekty, a jsou tedy na ně i různé reference, výsledkem druhého porovnání je nepravda (`false`).

Poznámka:

Pokud v předchozím příkladě nahradíte hodnoty `326` v proměnných `i2` a `i3` nějakou jinou hodnotou z intervalu `<-128; 127>`, proběhne porovnání správně. To je způsobeno tím, že tyto malé hodnoty nahrazuje kompilátor předdefinovanými konstantami a nevytváří pro dvě různá čísla z tohoto intervalu dva různé objekty.

■

3.2 Kdy nelze použít automatické zapouzdření

Auto-boxing a *auto-unboxing* má jen **několik málo omezení**, kdy ho nelze použít. Můžeme například automaticky převést typ `int` na typ `double`, to ale nelze, použijeme-li objektové typy. Nelze tedy automaticky převést typ `Integer` na typ `Double`, apod.

Následující tabulka ukazuje některé další převody mezi primitivními typy a jejich objektovými protějšky, které nejsou povoleny.

Tabulka 3.2: Použitelnost automatického zapouzdření

Převáděný typ	Typ po převodu	Lze použít?
`Integer`	`double`	ANO
`Integer`	`char`	NE
`int`	`Double`	NE
`int`	`Character`	NE
`Double`	`int`	NE
`Double`	`char`	NE
`double`	`Integer`	NE
`double`	`Character`	NE
`Character`	`int`	ANO
`Character`	`double`	ANO
`char`	`Integer`	NE
`char`	`Double`	NE

4 Cyklus for-each

Nyní se dostáváme k jedné velice užitečné novince jazyka Java 5.0. Jde o vylepšení dosavadního cyklu `for` pro **snazší zápis průchodu prvků pole, případně prvků nějaké kolekce**. Takovýto cyklus určitě znáte i z jiných programovacích jazyků, jako je např. C# či Visual Basic. V těchto jazycích často bývá tento cyklus označován jako cyklus ***for-each***. V syntaxi jazyka Java se sice tento cyklus označuje pouze klíčovým slovem `for`, pro přehlednost a pro rozlišení od klasického cyklu `for` budeme i zde používat obecné označení *for-each*.

Jak jsme již zmínili, tento cyklus je pouze **zjednodušující** zápis, místo něhož bychom mohli použít buď klasický cyklus `for`, popř. iterátor, musíme ale počítat s jedním malým, ale logicky celkem zřejmým, omezením. Tento cyklus, jak uvidíme dále, pracuje pouze s jednou proměnnou, kterou je v každém průchodu cyklu jeden prvek pole či kolekce. Nemůžeme tedy využít **žádnou číselnou proměnnou**, která by udávala pozici právě zpracovávaného prvku. Z toho je jasné, že tento cyklus využijeme pouze tehdy, chceme-li zpracovat všechny prvky pole či kolekce.

Zároveň musíme počítat s tím, že proměnná cyklu *for-each* je pouze lokální, takže tento cyklus nelze použít v případech, kdy chceme prvky pole či kolekce **modifikovat**. Modifikace prvků je možná pouze v případě, že jde o objekty tříd, které deklarují metody určené pro svou modifikaci.

Poznámka:

Možná by Vás napadlo, že i v těle takovéhoto cyklu si můžeme deklarovat lokální proměnnou, která by indexovala prvky při jejich zpracování. Nic nám sice nebrání toto provést, ale proč pak používat cyklus *for-each*, když můžeme použít běžný cyklus `for`?

■

4.1 Použití cyklu for-each pro procházení pole

Nejprve se podíváme na využití tohoto cyklu pro **práci s poli**. Následující příklad ukazuje nalezení největšího čísla v poli čísel a výpočet jejich aritmetického průměru.

Příklad 4.1: `Pole` – Průchod polem použitím cyklu *for-each*

```
public class Pole {
    public static int max(int[] cisla) {
        int max = cisla[0];
        for (int i : cisla) {
            if (i > max) max = i;
        }
        return max;
    }

    public static double prumer(int[] cisla) {
        int suma = 0;
        for (int i : cisla) {
            suma += i;
        }
        return (double) suma / cisla.length;
```

```
    }

    public static void main(String[] args) {
        int[] cisla = new int[] {2, 3, 8, 1, 12, 0, 5, 11};

        System.out.println(Pole.max(cisla));
        System.out.println(Pole.prumer(cisla));
    }
}
```

Výstup programu:

```
12
5.25
```

Zde vidíme velice jednoduchou syntaxi cyklu *for-each*. Pro názornost si ji můžeme zapsat obecně:

```
for (Typ Proměnná : Pole) {
    Tělo cyklu
}
```

Zde *Typ* je typ prvků v poli určeném proměnnou *Pole*. *Proměnná* je proměnná uchovávající v každém průchodu cyklu jeden prvek z pole *Pole*.

Samozřejmě můžeme tento cyklus použít i pro průchod pole libovolných objektů, syntaxe je úplně stejná.

4.1.1 Použití pro vícerozměrné pole

Pokud budeme chtít použít cyklus *for-each* pro zpracování vícerozměrného pole, **musíme postupovat poněkud odlišně.** Pokud budeme např. uvažovat dvourozměrné pole `Object[][] pole2D`, musíme pro jeho průchod použít dva cykly vnořené do sebe.

```
for (Object[] radek : pole2D) {
    for (Object prvek : radek) {
        Tělo cyklu
    }
}
```

Z toho je zřejmé, že cyklus *for-each* umí pracovat pouze s **jednorozměrným polem.** Pro více rozměrů bychom museli vnořit více cyklů *for-each* do sebe. Ale i přes to vidíme značnou výhodu použití cyklu *for-each*.

4.2 Použití cyklu for-each pro procházení kolekce

Jak jsme již zmínili, cyklus *for-each* lze použít stejně jako pro průchod pole i pro průchod kolekcí.

```
for (Typ Proměnná : Kolekce) {
    Tělo cyklu
}
```

Zde *Typ* je typ prvků v kolekci *Kolekce*, *Kolekce* je kolekce typu `Collection<Typ>`. *Proměnná* je proměnná uchovávající v každém průchodu cyklu jeden prvek z kolekce *Kolekce*.

Na následujícím příkladě si ukážeme zpracování všech prvků kolekce s využitím tohoto cyklu. Tento příklad je modifikací příkladu [Příklad 4.1] z kapitoly [4.1/36].

Příklad 4.2: `Kolekce` – Průchod kolekcí použitím cyklu *for-each*

```
import java.util.*;

public class Kolekce {
    public static int max(Collection<Integer> cisla) {
        int max = Integer.MIN_VALUE;
        for (int i : cisla) {
            if (i > max) max = i;
        }
        return max;
    }

    public static double prumer(Collection<Integer> cisla) {
        int suma = 0;
        for (int i : cisla) {
            suma += i;
        }
        return (double) suma / cisla.size();
    }

    public static void main(String[] args) {
        Collection<Integer> cisla = new ArrayList<Integer>();
        cisla.add(2);   cisla.add(3);
        cisla.add(8);   cisla.add(1);
        cisla.add(12);  cisla.add(0);
        cisla.add(5);   cisla.add(11);

        System.out.println(Kolekce.max(cisla));
        System.out.println(Kolekce.prumer(cisla));
    }
}
```

Výstup programu:

```
12
5.25
```

Vidíme, že použití je stejné, jako v případě zpracování pole. Jak je ale možné, že cyklus *for-each* ví, jakým způsobem procházet danou kolekci? Java 5.0 zavádí nové rozhraní `Iterable` (viz kapitola [4.2.1/39]), které obsahuje jedinou metodu `iterator()`. Ta vrací objekt typu `Iterator`, jehož metody stačí cyklu *for-each* na to, aby mohl bezpečně projít všechny prvky kolekce. Cyklus *for-each* v tomto případě tedy nahrazuje použití klasického iterátoru.

Z toho plyne, že cyklus *for-each* není omezen pouze na použití kolekcí definovaných ve standardních knihovnách, ale i na jakoukoliv jinou třídu, která implementuje rozhraní `Iterable`. Takovou třídu si můžeme vytvořit sami a pak ji bez problémů zpracovávat pomocí cyklu *for-each*.

Poznámka:

Za zmínku ještě stojí, že ani v tomto případě nemůžeme použít cyklus *for-each* pro odstraňování či modifikaci prvků kolekce.

■

4.2.1 Rozhraní `Iterable<T>`

Rozhraní `Iterable` je deklarováno v balíčku `java.lang`. Je to proto, že jeho použití je přímo spojeno se syntaxí jazyka. A jistě také proto, aby bylo zřejmé, že jeho použití není omezeno na žádné konkrétní třídy či jiná rozhraní.

Všimněte si, že rozhraní `Iterable` je definováno jako generický typ. Stejně tak je definováno rozhraní `Iterator`.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

U obou těchto rozhraní určuje typový parametr `T` typ prvků v procházené kolekci.

4.2.2 Použití pro průchod mapy

Mapou máme na mysli zvláštní kolekci implementující rozhraní `Map<K, V>`. To je kolekce, která uchovává dvojice klíč-hodnota, kde jedinečný klíč jednoznačně určuje svou hodnotu. Možná jste si již všimli, že toto rozhraní neimplementuje rozhraní `Iterable`. Nelze tedy pro zpracování takovéto kolekce použít přímo cyklus *for-each*.

Máme ovšem několik možností, jak se s průchodem mapy vypořádat. Vše záleží na tom, zda budeme chtít v mapě procházet pouze klíče, pouze hodnoty, nebo dvojice klíč-hodnota. K tomuto účelu jsou v rozhraní `Map` tři metody, které vrací kolekce zpracovatelné cyklem *for-each*.

Tabulka 4.1: Metody umožňující průchod mapy cyklem *for-each*

`Set<K> keySet()`	Metoda `keySet()` vrací množinu všech klíčů v mapě.
`Collection<V> values()`	Metoda `values()` vrací kolekci všech hodnot v mapě.
`Set<Map.Entry<K, V>> entrySet()`	Metoda `entrySet()` vrací množinu objektů typu `Map.Entry`. Rozhraní `Map.Entry<K, V>` je rozhraní vnořené v rozhraní Map<K, V>. Objekt typu `Map.Entry<K, V>` uchovává dvojici klíč-hodnota, kde klíč je objekt typu `K` a hodnota je objekt typu `V`.

Kolekce typu `Set` nebo `Collection` již můžeme procházet s použitím cyklu *for-each*.

Poznámka:

Pokud Vám není úplně jasné, proč pouze metoda `values()` vrací kolekci typu `Collection` a ostatní vrací kolekci typu `Set`, je dobře, že čtete i poznámky. Jelikož hodnoty se mohou v mapě opakovat u různých klíčů, je pro kolekci hodnot použita kolekce typu `Collection`, která může uchovávat opakující se prvky. Na rozdíl od toho klíče musejí být v mapě jedinečné. Proto metoda `keySet()` vrací kolekci typu `Set`. V množině typu `Set` se nesmějí totiž vyskytovat duplicitní prvky. V případě metody `entrySet()` je důvod stejný jako u metody `keySet()`. Není možné mít v jedné mapě dvě stejné dvojice klíč-hodnota.

■

5 Metody s proměnným počtem parametrů

Možnost definovat **metodu s proměnným počtem parametrů** je další, ne příliš rozsáhlá novinka programovacího jazyka Java 5.0. S podobnou konstrukcí se můžeme setkat např. v programovacím jazyku C či C++. V Javě je deklarace a použití metod s proměnným počtem parametrů **mnohem jednodušší.**

Určitě jste se ocitli v situaci, kdy jste potřebovali definovat metodu, která zpracovává nějaké objekty stejného typu, ale předem jste nevěděli, kolik těchto objektů bude. Poměrně elegantní bylo definovat typ parametru takovéto metody jako pole požadovaného typu. To je celkem rozumné, protože takové pole můžeme vytvořit až v době volání metody. Samozřejmě nejsme, co do počtu objektů v poli, nijak omezeni.

Jak uvidíme dále, v těle metody, která má proměnný počet parametrů, k žádné velké změně nedojde, ale při volání takovéto metody bude rozdíl zřejmý na první pohled.

5.1 Pravidla pro deklaraci metody s proměnným počtem parametrů

Dříve, než si ukážeme, jak se metoda s proměnným počtem parametrů deklaruje, uvedeme si **seznam pravidel**, která jsou pro deklaraci takovéto metody důležitá.

- Metoda může deklarovat pouze **jednu skupinu proměnného počtu parametrů**, které jsou všechny stejného typu.
- Pokud má mít metoda i další parametry, musí být tyto uvedené **dříve**, než skupina proměnného počtu parametrů.
- Svým způsobem můžeme i **konstruktor** třídy považovat za metodu, takže i konstruktor může mít proměnný počet parametrů.
- Typem parametrů, jejichž počet je proměnný, může být jakýkoliv **primitivní** i **objektový** typ. Tedy i generický typ nebo výčtový typ (viz [6/44]).

5.2 Deklarace metody s proměnným počtem parametrů

Na následujícím příkladě si ukážeme deklaraci metody s proměnným počtem parametrů. Půjde o rozšíření třídy `Zasobnik2`, se kterou jsme se již setkali v [Příklad 1.4] v kapitole [1.1.3/4]. Abychom neprováděli změny přímo ve třídě `Zasobnik2`, vytvoříme si potomka této třídy a nazveme ho `Zasobnik3`.

Ve třídě `Zasobnik2` jsme definovali metodu `vlozit(T prvek)`, která nám umožnila vložit do zásobníku jeden prvek. Nyní vytvoříme metodu, která nám umožní přidat do zásobníku **předem neurčený počet prvků.**

Příklad 5.1: `Zasobnik3` – Deklarace metody s proměnným počtem parametrů

```
public class Zasobnik3<T> extends Zasobnik2<T> {
    public Zasobnik3(int velikost) {
        super(velikost);
```

```
    }
    public boolean vlozit(T... prvky) {
        for (T prvek : prvky) {
            if (!super.vlozit(prvek))
                return false;
        }
        return true;
    }
}
```

Vidíme, že stačí za typ proměnné zapsat **tři tečky** (...), a parametrů tohoto typů může být několik. Ovšem v těle metody pracujeme s proměnným počtem parametrů typu `T` jako s polem typu `T[]`. V našem příkladě tedy s výhodou použijeme pro zpracování všech těchto parametrů cyklus *for-each*.

Poznámka:

Všimněte si, že nová metoda `vlozit()` se liší od původní metody tím, že má proměnný počet parametrů. I to stačí k tomu, abychom mohli metodu přetížit. ■

5.3 Použití metody s proměnným počtem parametrů

Neopouštějme ještě [Příklad 5.1]. Pro ukázku použití metody s proměnným počtem parametrů si vytvoříme jednoduchý testovací scénář.

Příklad 5.2: `Test` – Použití metody s proměnným počtem parametrů

```
public class Test {
    public static void main(String[] args) {
        Zasobnik3<String> zasobnik1 = new Zasobnik3<String>(10);
        zasobnik1.vlozit("slovo1", "slovo2", "slovo3");

        String prvek;
        while ((prvek = zasobnik1.odebrat()) != null)
            System.out.print(prvek + "  ");

        System.out.println();
        Zasobnik3<Integer> zasobnik2 = new Zasobnik3<Integer>(10);
        zasobnik2.vlozit(3, 2, 8, 5, 10, 0, 12);

        Integer cislo;
        while ((cislo = zasobnik2.odebrat()) != null)
            System.out.print(cislo + "  ");
    }
}
```

Výstup programu:

```
slovo3  slovo2  slovo1
12  0  10  5  8  2  3
```

Vidíme, že při použití metody s proměnným počtem parametrů stačí oddělovat jednotlivé parametry **čárkou**, jako v případě jakýchkoliv jiných parametrů. I když v těle metody pracujeme s proměnným počtem parametrů jako s polem, při volání takové metody nemusíme žádné pole vytvářet. O to se už postará kompilátor.

Samozřejmě bychom mohli metodu `vlozit()` deklarovat následovně:

```
public boolean vlozit2(T[] prvky) { ... }
```

Tělo metody by zůstalo zcela stejné. Ale při použití metody bychom museli explicitně vytvořit pole prvků, které chceme do zásobníku vložit.

```
zasobnik1.vlozit2(new String[] {"slovo1", "slovo2", "slovo3"});
```

Poznámka:
Zajímavé je, že při volání metody s proměnným počtem parametrů můžeme předat takovéto metodě místo jednotlivých parametrů pole. Tzn. metodě, která má deklarované parametry jako `T... prvky` můžeme klidně předat jedním parametrem pole typu `T[]`. Není to ale možné naopak. Pokud je v metodě deklarován pouze jeden parametr typu `T[]`, nemůžeme při volání metody uvést jednoduše seznam parametrů oddělených čárkami.

■

Poznámka 2:
Také je zajímavé to, že na místě proměnného počtu parametrů nemusíme při volání metody uvést parametr ani jeden. V těle metody přeci pracujeme s proměnným počtem parametrů jako s polem a pole může mít klidně nulovou délku.

■

5.4 Různé způsoby zápisu

V závěru kapitoly si pouze uvedeme některé další možné či nemožné zápisy deklarace metod s proměnným počtem parametrů. Jejich použití je zcela stejné, jako v předchozích příkladech.

```
public void metoda(String... slova) { ... }
```

Metoda má proměnný počet parametrů typu `String`.

```
public void metoda(int cislo1, int cislo2, String... slovo) { ... }
```

Metoda má dva parametry typu int a dále proměnný počet parametrů typu `String`.

```
public void metoda(int[] cisla, String... slova) { ... }
```

Prvním parametrem metody je pole celých čísel. Dalšími parametry jsou objekty typu `String`. V případě parametru `cisla` nejde ale o proměnný počet parametrů.

5.4.1 Chybné zápisy

```
public void metoda(String... slova, int cislo) { ... }
```

Samostatný parametr `cislo` musí být uveden před deklarací parametrů typu `String`.

```
public void metoda(String... slovo, int... cisla) { ... }
```

Nelze takto deklarovat dvě skupiny předem neurčeného počtu parametrů.

6 Výčtové typy

Dlouho očekávanou novinkou jazyka Java jsou výčtové typy. Java definuje výčtové typy jako objektové typy a zavádí novou třídu `Enum`, která je jejich společným předkem. I když jsou výčtové typy v podstatě třídy, jejich definice je trochu odlišná než definice klasických tříd.

V úvodu této kapitoly si nejprve nastíníme základní představu o výčtových typech. Pokud již máte s výčtovými typy nějakou, byť malou, zkušenost, klidně kapitolu 6.1 přeskočte. V další části se podíváme na deklaraci výčtových typů a jejich použití v programu. V závěru kapitoly se podíváme na složitější výčtové typy a na jejich možnosti.

Poznámka:

Něco více o třídě `Enum` se také dozvíte v kapitole [9.2/70].

■

6.1 Představení výčtových typů

Výčtové typy můžeme znát i z jiných programovacích jazyků, jako jsou např. Pascal, C++ nebo C#. Výčtovým typem definujeme množinu diskrétních hodnot (např. dny v týdnu, světové strany atp.), kterých může proměnná výčtového typu nabývat. To je v mnoha případech velice užitečné.

Zmíněné programovací jazyky přiřazují každé hodnotě výčtového typu celé číslo podle pořadí jejich definice. Ve výčtovém typu (Po, Ut, St, Ct, Pa, So, Ne) má tedy pondělí hodnotu 1 a neděle 7. Z toho se také odvíjí způsob práce s takovými typy. Na místě, kde kompilátor očekává hodnotu výčtového typu můžeme klidně použít celočíselnou hodnotu a naopak.

Tomuto přístupu se však programovací jazyk Java vyhýbá. V Javě jsou výčtové typy ryze objektové a konstanty výčtových typů jsou instance výčtového typu.

6.1.1 Dokud jsme neznali výčtové typy

Dokud nebyla možnost v Javě využívat výčtové typy, používaly se různé postupy, které se je snažily nahradit. Mohli jsme např. deklarovat třídu, která obsahovala množinu statických konstant, kterým jsme pevně přiřadili např. číselné hodnoty.

Příklad 6.1: `CastDne1` – Náhrada za výčtový typ 1

```
public class CastDne1 {
    public static final int RANO = 0;
    public static final int DOPOLEDNE = 1;
    public static final int ODPOLEDNE = 2;
    public static final int VECER = 3;
    public static final int NOC = 4;
}
```

To je velice jednoduché řešení, které má ovšem jedno úskalí. Pokud bude nějaká metoda očekávat jako parametr „část dne“, musí jít o hodnotu typu `int`. A my nemáme žádnou

kontrolu nad tím, zda jsme do parametru dosadili správnou hodnotu. Jako hodnotu takového parametru můžeme klidně použít číslo 12.

Vynalézavější jedinci si poradili trochu jinak.

Příklad 6.2: `CastDne2` – Náhrada za výčtový typ 2

```
public class CastDne2 {
    public static final CastDne2 RANO = new CastDne2(0);
    public static final CastDne2 DOPOLEDNE = new CastDne2(1);
    public static final CastDne2 ODPOLEDNE = new CastDne2(2);
    public static final CastDne2 VECER = new CastDne2(3);
    public static final CastDne2 NOC = new CastDne2(4);

    private int i;

    private CastDne2(int i) {
        this.i = i;
    }
}
```

Teď už máme konstanty `RANO`, `DOPOLEDNE`, ... objektových typů a nemůžeme je tedy nahradit číselnými hodnotami. Pokud ale budeme potřebovat porovnávat dvě proměnné typu `CastDne2`, musíme ještě překrýt metodu `equals()` třídy `Object`, popř. doplnit další metody pro provádění dalších operací.

```
public class CastDne2 {
    ...

    public boolean equals(Object c) {
        if (c instanceof CastDne2)
            return this.i == ((CastDne2) c).i;
        else return false;
    }
}
```

Vidíme, že se dá bez výčtových typů s trochou úsilí žít. Znamená to ovšem psát spoustu kódu, který je poměrně rozsáhlý a ne příliš přehledný.

6.2 Definice jednoduchého výčtového typu

Jak uvidíme na následujícím příkladu, záhlaví definice výčtového typu vypadá podobně jako záhlaví definice třídy. Klíčové slovo `class` je nahrazeno klíčovým slovem `enum` (zkratka anglického slova *enumeration* – výčet, seznam). V těle typu je pak uveden seznam hodnot oddělených čárkou, kterých může proměnná výčtového typu nabývat.

Příklad 6.3: `CastDne3` – Jednoduchý výčtový typ

```
public enum CastDne3 {
    RANO, DOPOLEDNE, ODPOLEDNE, VECER, NOC
}
```

Ano, skutečně je to takhle jednoduché. Jde přitom o úplně stejný případ částí dne, jako v předchozích dvou příkladech. Z takovéto definice kompilátor vytvoří třídu, která je podtřídou třídy `Enum`. Jednotlivé výčtové konstanty jsou pak jejími instancemi. Každá výčtová konstanta v sobě uchovává svůj název a pořadí. Třída výčtového typu nám umožňuje získat pole všech konstant daného výčtového typu a některé další informace.

6.3 Použití výčtového typu v programu

Na následujících příkladech, které tvoří jakousi databázi studentů, si ukážeme použití výčtových typů v různých řídicích strukturách. Uvidíme, že snadno můžeme použít konstanty výčtového typu v **podmínce**, v **cyklu** nebo v **příkazu switch**.

Pro všechny tyto příklady bude společný následující výčtový typ a dvě třídy.

Příklad 6.4: `Studium` – Použití výčtového typu v programu

```
public enum Studium {
    BC, MGR, ING, RNDR, PHD
}

import java.util.Date;
public class Student implements Comparable<Student>{
    private String jmeno, prijmeni;
    private Studium typ;

    public Student(String jmeno, String prijmeni, Studium typStudia) {
        this.jmeno = jmeno;
        this.prijmeni = prijmeni;
        this.typ = typStudia;
    }

    public String jmeno() { return jmeno + " " + prijmeni; }
    public Studium typStudia() { return typ; }
    public String toString() { return jmeno(); }
    public int compareTo(Student s) {
        return prijmeni.compareTo(s.prijmeni);
    }
}

import java.util.*;
import java.io.PrintStream;
public class Studenti {
    public Set<Student> studenti;

    public Studenti() {
        studenti = new TreeSet<Student>();
    }

    public boolean pridej(Student s) { return studenti.add(s); }

    ...

    public static void main(String[] args) {
        Studenti s = new Studenti();
        s.pridej(new Student("František", "Halounek", Studium.MGR));
        s.pridej(new Student("Karel", "Sobota", Studium.ING));
        s.pridej(new Student("Lucie", "Odřelová", Studium.MGR));
        s.pridej(new Student("Naděžda", "Placatá", Studium.BC));
        s.pridej(new Student("Zdeněk", "Podskalský", Studium.PHD));
        s.pridej(new Student("Jan", "Placatý", Studium.BC));
        s.pridej(new Student("Iveta", "Slámová", Studium.RNDR));
        s.pridej(new Student("Marie", "Zapletalová", Studium.PHD));
        s.pridej(new Student("Petr", "Julínek", Studium.BC));

        ...
    }
}
```

Zde výčtový typ `Studium` představuje možné typy studijních programů. Třída `Student` uchovává informace o jednom studentovi, tj. jméno, příjmení a typ studovaného oboru. Třída `Studenti` reprezentuje databázi studentů. V této třídě máme zatím vytvořenou jedinou instanční metodu `pridej()`, která slouží k přidání nového studenta do databáze. Ve statické metodě `main()` je kód, který vytvoří novou databázi studentů a naplní ji daty. Na místa označená ... (samozřejmě se nejedná o proměnný počet parametrů) budeme postupně přidávat jednoduché testovací scénáře, na kterých si ukážeme práci s výčtovými typy.

6.3.1 Použití výčtového typu v podmínce

I když je v Javě výčtový typ objektový, a tedy i konstanty výčtového typu jsou **objekty**, v podmínce můžeme tyto konstanty porovnávat operátorem `==`. Není nutné používat metodu `equals()`, jak jsme byli zvyklí u jiných objektů.

Příklad 6.5: `Studenti` – Použití výčtového typu v podmínce

```
public class Studenti {
    ...

    public Set<Student> vyber(Studium s) {
        Set<Student> vysledek = new HashSet<Student>();

        for (Student st : studenti) {
            if (st.typStudia() == s) vysledek.add(st);
        }

        return vysledek;
    }

    public void studentiPodleTypuStudia(PrintStream out) {
        out.println("Bakalářské studium: " + vyber(Studium.BC));
        out.println("Magisterské studium: " + vyber(Studium.MGR));
        out.println("Inženýrské studium: " + vyber(Studium.ING));
        Set<Student> dr = vyber(Studium.RNDR);
        dr.addAll(vyber(Studium.PHD));
        out.println("Doktorandské studium: " + dr);
    }

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

        s.studentiPodleTypuStudia(System.out);
    }
}
```

Výstup programu:

```
Bakalářské studium: [Jan Placatý, Naděžda Placatá, Petr Julínek]
Magisterské studium: [Lucie Odřelová, František Halounek]
Inženýrské studium: [Karel Sobota]
Doktorandské studium: [Zdeněk Podskalský, Marie Zapletalová,
                       Iveta Slámová]
```

V metodě `vyber()` procházíme databázi studentů a jednoduchou podmínkou testujeme, zda student studuje v typu studia určeném parametrem `s`. Pokud ano, vložíme jej do pomocné kolekce. Výsledkem metody je tedy kolekce studentů studujících v zadaném typu studia.

Metoda `studentiPodleTypuStudia()` už jen vypíše jména studentů roztříděná podle typu studia.

6.3.2 Procházení konstant výčtového typu

Na následujícím příkladě uvidíme, jak můžeme snadno **procházet konstanty výčtového typu.** K tomu slouží metoda `values()`, která vrací pole všech konstant výčtového typu. Jde o instanční metodu, kterou kompilátor sám **vygeneruje** pro každý výčtový typ. Není tedy deklarována ve třídě `Enum`.

Příklad 6.6: `Studenti` – Iterátor napříč výčtovým typem

```
public class Studenti {
    ...

    public void typyStudia(PrintStream out) {
        for (Studium s : Studium.values()) {
            out.print(s + "  ");
        }
        out.println();
    }

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

        s.typyStudia(System.out);
    }
}
```

Výstup programu:

```
BC  MGR  ING  RNDR  PHD
```

6.3.3 Použití výčtového typu v příkazu `switch`

Příkaz switch je příkaz pro vícenásobné větvení programu. V předchozích verzích jazyka bylo možné použít tento příkaz pouze v případě, kdy výraz, podle kterého se rozhoduje, je typu `char`, `byte`, `short` nebo `int`. Nyní je možné použít tento příkaz i pro **konstanty výčtového typu.** Syntaxe je stejná, jako dříve. U každé větve již ale neuvádíme název výčtového typu, ale pouze samotnou konstantu.

Nesmíme také zapomenout na to, že v příkazu `switch` není nutné napsat větve pro všechny konstanty daného výčtového typu. Můžeme definovat několik větví pro konkrétní hodnoty a větev `default` pro ostatní.

Příklad 6.7: `Studenti` – Výčtový typ a příkaz `switch`

```
public class Studenti {
    ...

    public void poStudiu(PrintStream out) {
        for (Student s : studenti) {
            switch (s.typStudia()) {
                case BC : out.println("Bc. " + s.jmeno()); break;
                case MGR : out.println("Mgr. " + s.jmeno()); break;
                case ING : out.println("Ing. " + s.jmeno()); break;
                case RNDR : out.println("RNDr. " + s.jmeno()); break;
```

```
                case PHD : out.println(s.jmeno() + ", PhD."); break;
            }
        }
    }

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

        s.poStudiu(System.out);
    }
}
```

Výstup programu:

```
Mgr. František Halounek
Bc. Petr Julínek
Mgr. Lucie Odřelová
Bc. Naděžda Placatá
Bc. Jan Placatý
Zdeněk Podskalský, PhD.
RNDr. Iveta Slámová
Ing. Karel Sobota
Marie Zapletalová, PhD.
```

Tato jednoduchá metoda vypíše jména všech studentů z databáze, a připojí k nim tituly, které studenti získají po úspěšném ukončení studia. U každého studenta se tedy program rozhoduje na základě hodnoty `s.typStudia()`. A k tomu právě s výhodou použijeme příkaz `switch`.

6.4 Složitější definice výčtového typu

Jak již víme, výčtový typ je v Javě třída a konstanty výčtového typu jsou instance této třídy. Jako každý objekt v Javě, mají i konstanty výčtových typů **své metody**. Jde například o metody, které nám umožní zjistit název konstanty (`name()`), pořadí konstanty ve výčtovém typu (`ordinal()`), porovnat dvě konstanty (`compareTo()`) atd. Java zavádí pro výčtové typy zcela novou možnost **přiřadit jednotlivým konstantám výčtového typu další vlastnosti**, a také definovat **vlastní metody**. Tím vlastně definujeme chování jednotlivých konstant.

Na následujícím modelovém příkladu si ukážeme, jak můžeme takový výčtový typ definovat.

Příklad 6.8: `Operatory` – Konstanty výčtového typu s parametrem

```
public enum Operatory {
    PLUS('+'),
    MINUS('-'),
    KRAT('*'),
    DELENO('/');

    private char znak;

    private Operatory(char znak) {
        this.znak = znak;
    }

    public char znak() {
        return znak;
    }
```

```
}

public class OperatoryTest {
    public static void main(String[] args) {
        for (Operatory p : Operatory.values()) {
            System.out.print(p.znak() + "  ");
        }
    }
}
```

Výstup programu:

```
+  -  *  /
```

Ke každé konstantě výčtového typu jsme přidali jakýsi **parametr**, kterým specifikujeme vlastnost jednotlivé konstanty. Pokud ale přidáme parametr, musíme i přidat deklaraci **konstruktoru** s jedním parametrem. Parametr konstruktoru může být libovolného primitivního i objektového typu, parametry všech konstant ale musejí být stejného typu (pokud ovšem nedeklarujeme více konstruktorů, viz dále). Pak můžeme definovat libovolné metody výčtového typu, které určují chování. My jsme definovali instanční metodu `znak()`, která vrátí znakovou reprezentaci daného operátoru. Této metodě rozumí každá konstanta výčtového typu.

Počet takovýchto dodatečných vlastností ovšem není nijak omezen, klidně můžeme definovat **více parametrů**.

Příklad 6.9: `Operatory` – Více parametrů u konstanty výčtového typu

```
public enum Operatory {
    PLUS('+', true, 50),
    MINUS('-', false, 50),
    KRAT('*', true,  100),
    DELENO('/', false, 100);

    private char znak;
    private boolean komutativni;
    private int priorita;

    private Operatory(char znak, boolean komutativni, int priorita) {
        this.znak = znak;
        this.komutativni = komutativni;
        this.priorita = priorita;
    }

    public char znak() {
        return znak;
    }
    public boolean komutativni() {
        return komutativni;
    }
    public int priorita() {
        return priorita;
    }
}
```

Můžeme definovat u každé konstanty **jiný počet parametrů** nebo parametry různých typů. Musíme potom také ale definovat **různé konstruktory**, které budou svými parametry odpovídat všem konstantám výčtového typu.

Důležité:

Všimněte si, že jsme konstruktor výčtového typu deklarovali jako `private`. To je podmínka. I když je výčtový typ třída, nemůžeme sami vytvářet její instance. Jedinými instancemi výčtového typu jsou právě jeho konstanty. Proto musí být konstruktor výčtového typu vždy `private`.

■

6.4.1 Každá konstanta má jiné chování

V předchozím příkladě jsme definovali pro výčtový typ `Operatory` metody, které ovšem byly stejné pro všechny konstanty výčtového typu. Tyto konstanty tedy měly stejné chování. Pokud ale budeme chtít vytvořit **různé chování pro různé konstanty**, máme na výběr ze dvou možností.

První z nich si ukážeme na následujícím příkladu. Našemu výčtovému typu `Operatory` přidáme metodu `vypocti(x, y)`, která pro každý operátor provede příslušnou („jeho“) operaci s hodnotami `x` a `y`.

Příklad 6.10: `Operatory` – Různé chování různých konstant výčtového typu

```
public enum Operatory {
    ...

    public double vypocti(double x, double y) {
        double vysledek = 0;
        switch (this) {
            case PLUS : vysledek = x + y; break;
            case MINUS : vysledek = x - y; break;
            case KRAT : vysledek = x * y; break;
            case DELENO : vysledek = x / y; break;
        }
        return vysledek;
    }
}

public class OperatoryTest {
    public static void main(String[] args) {
        for (Operatory p : Operatory.values()) {
            System.out.println("6 " + p.znak() + " 3 = " +
                               p.vypocti(6, 3));
        }
    }
}
```

Výstup programu:

```
6 + 3 = 9.0
6 - 3 = 3.0
6 * 3 = 18.0
6 / 3 = 2.0
```

Druhou možností je definovat metodu `vypocti()` jako **abstraktní** a její implementaci provést pro každou konstantu výčtového typu zvlášť. Následující výčtový typ `Operatory2` je modifikací výčtového typu `Operatory`. Všimněte si u předchozího a následujícího příkladu různé implementace, ale naprosto stejného chování.

Příklad 6.11: `Operatory2` – Abstraktní metoda výčtového typu

```
public enum Operatory2 {
    PLUS('+', true, 50) {
        public double vypocti(double x, double y) {
            return x + y;
        }
    },
    MINUS('-', false, 50) {
        public double vypocti(double x, double y) {
            return x - y;
        }
    },
    KRAT('*', true,  100) {
        public double vypocti(double x, double y) {
            return x * y;
        }
    },
    DELENO('/', false, 100) {
        public double vypocti(double x, double y) {
            return x / y;
        }
    };

    private char znak;
    private boolean komutativni;
    private int priorita;

    private Operatory2(char znak, boolean komutativni, int priorita) {
        this.znak = znak;
        this.komutativni = komutativni;
        this.priorita = priorita;
    }

    public char znak() {
        return znak;
    }
    public boolean komutativni() {
        return komutativni;
    }
    public int priorita() {
        return priorita;
    }

    public abstract double vypocti(double x, double y);
}
```

Pokud teď ve třídě `OperatoryTest` nahradíme typ `Operatory` typem `Operatory2`, uvidíme, že výstup programu je úplně stejný. Záleží tedy na nás, ale i na konkrétních podmínkách řešeného problému, jaký způsob implementace rozdílného chování jednotlivých konstant výčtového typu zvolíme.

7 Práce s kolekcemi v JDK 5.0

Java používá od verze 1.2 velmi dobře propracovaný soubor tříd a rozhraní pro práci s kolekcemi (tzv. *Collection framework*). Všechny tyto třídy a rozhraní najdeme v balíčku `java.util`. Kolekcí přitom můžeme rozumět konkrétní implementaci nějaké datové struktury. Stejný způsob práce s kolekcemi, který jsme používali doposud, můžeme použít i ve verzi JDK 5.0. Jedinou zásadní změnou, která přibyla v nové verzi jazyka Java, je implementace všech tříd a rozhraní pro práci s kolekcemi s využitím **generických typů**.

V úvodu této kapitoly si předvedeme základní práci s kolekcemi. Nebudeme se však zabývat detailním popisem práce s těmito třídami, ta je v zásadě stejná jako v dřívějších verzích Javy. Půjde spíše o modelové příklady prezentující použití kolekcí ve spojení s generickými typy. Ukážeme si také, jakých změn doznala práce s iterátory. Podíváme se také na deklaraci metod, které nějakým způsobem zpracovávají kolekce. V závěru kapitoly si ukážeme některé nové kolekce, které přibyly v JDK 5.0.

V každém případě tuto kapitolu berte pouze jako shrnutí těch nejpoužívanějších dovedností.

7.1 Kolekce jako generický typ

Základní struktura tříd a rozhraní představující různé typy kolekcí je stejná jako v předchozích verzích Javy. Všechny tyto třídy a rozhraní jsou nyní implementovány s využitím generických typů. Všechny třídy a rozhraní odvozené od rozhraní `Collection` mají jeden typový parametr `E`, který představuje typ prvků ukládaných do kolekce. Všechny třídy a rozhraní odvozené od rozhraní `Map` mají dva typové parametry `K` a `V`, z nichž `K` představuje typ objektu klíče a `V` představuje typ objektů stojících v pozici hodnoty.

Rozhraní `Collection` navíc rozšiřuje rozhraní `Iterable`. Tím je zajištěno, že všechny kolekce typu `Collection` můžeme procházet pomocí cyklu *for-each*.

7.2 Práce s kolekcí

7.2.1 Vytvoření a naplnění kolekce

Příklad 7.1: `Kolekce` – Vytvoření a naplnění kolekce

```
import java.util.*;

public class Kolekce {
    public static void main(String[] args) {
        List<String> seznam = new LinkedList<String>();
        seznam.add("a");
        seznam.add("b");
        seznam.add("c");
        seznam.add("d");
        //seznam.add(15);
        ...
```

```
    }
}
```

Vidíme, že vytvoření a použití kolekce je stejné, jako bylo i dříve. Pouze si musíme dát pozor na správné uvedení typového parametru v závorkách < a >.

Při vkládání prvků do kolekce metodou `add()` máme již při překladu zajištěnou typovou kontrolu. Pokud zrušíme komentář na řádku `//seznam.add(15);` zahlásí kompilátor při překladu chybu *‚cannot find symbol: method add(int)'*. V našem případě metoda `add()` očekává parametr typu `String`, jak jsme uvedli v deklaraci proměnné `seznam`.

7.2.2 Použití automatického zapouzdření

Následující příklad ukazuje velmi užitečný případ použití automatického zapouzdření. Pokud budeme mít kolekci čísel, není nutné při jejich vkládání do kolekce vytvářet objekty odpovídajících tříd.

Příklad 7.2: `Zapouzdreni` – Použití auto-boxingu při práci s kolekcí

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

public class Zapouzdreni {
    public static void main(String[] args) {
        ArrayList<Number> cisla = new ArrayList<Number>();
        cisla.add(12);
        cisla.add(15.5F);
        cisla.add(3.8);
        cisla.add(18L);

        out.println(cisla);

        for(Number i : cisla) {
            out.println(i.getClass().getName());
        }
    }
}
```

Výstup programu:

```
[12, 15.5, 3.8, 18]
java.lang.Integer
java.lang.Float
java.lang.Double
java.lang.Long
```

Všimněte si, že kompilátor správně přiřadí datový typ hodnotám `15.5F` (`Float`) a `18L` (`Long`). Pro ověření jsme nechali vypsat názvy tříd, jejichž objekty v kolekci máme.

7.2.3 Průchod kolekce iterátorem

Rozhraní `Iterator` je od JDK 5.0 deklarováno jako generické. Z toho plyne jediný rozdíl při použití iterátorů. Rozhraní `Iterator` má jeden typový parametr `E`, který představuje typ prvků procházené kolekce. Známá metoda `next()` nám tedy vrací přímo objekt typu `E`, nikoliv objekt typu `Object`, jak tomu bylo dříve. Máme tedy již při překladu programu zajištěnou typovou kontrolu a nemusíme se navíc starat o explicitní přetypování.

Příklad 7.3: `Kolekce` – Průchod kolekce iterátorem

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

public class Kolekce {
    public static void main(String[] args) {
        ...

        Iterator<String> i = seznam.iterator();
        String s = "";
        while (i.hasNext()) {
            s += i.next() + " ";
        }
        out.println(s);
    }
}
```

Výstup programu:

```
a b c d
```

7.3 Metody pracující s kolekcemi

Zvláštní část kapitoly si zaslouží metody, které nějakým způsobem zpracovávají kolekce. Typicky může jít o statické metody implementující algoritmy např. pro řazení seznamů. Nemusí jít ale vždy pouze o statické metody. Deklaraci takovýchto metod můžeme obohatit o typové parametry, neboli takovouto metodu můžeme deklarovat jako generickou.

V kapitolách [1.4/12] a [1.5.4.2/20] jsme si ukázali, jak deklarovat jednoduché generické metody a jak číst některé složitější zápisy deklarace generických metod. S trochou nadsázky můžeme nyní říct: V této kapitole si ukážeme, jak postupovat při deklaraci složitější generické metody.

Tuto problematiku si ukážeme na příkladu metody `<T> boolean jeNejvetsi(Collection c, T o)`, která bude zjišťovat, zda je prvek o největším prvkem kolekce `c`.

Mohli bychom definovat metodu velmi prostě.

```
public static boolean jeNejvetsi(Collection c, Object o)
```

My ovšem chceme víc. Základním požadavkem je, že objekty uložené v kolekci budou porovnatelné metodou `compareTo()`, neboli budou implementovat rozhraní `Comparable`. To znamená přidat typový parametr a ohraničit jej shora.

```
public static <T extends Comparable<T>>
    boolean jeNejvetsi(Collection<T> c, T o)
```

Jenomže to by znamenalo, že přímo třída `T` musí implementovat rozhraní `Comparable<T>`. Může ale nastat situace, kdy třída `T` již dědí implementaci rozhraní `Comparable` od nějaké své rodičovské třídy. Musíme tedy navíc typový parametr rozhraní `Comparable` ohraničit zdola.

```
public static <T extends Comparable<? super T>>
    boolean jeNejvetsi(Collection<T> c, T o)
```

Tím jsme vlastně řekli, že rozhraní `Comparable` může implementovat jakákoliv „nadtřída“ třídy `T`. Ještě se ale musíme zamyslet nad typovým parametrem rozhraní `Collection` u typu

parametru metody `c`. Naší metodu bychom mohli použít pouze pro kolekce, které obsahují objekty typu `T`. Aby byla metoda opravdu obecná, je nutné připustit možnost, že kolekce bude obsahovat i objekty tříd, které jsou podtřídami třídy `T`. Musíme tedy typový parametr rozhraní `Collection` ohraničit shora.

```
public static <T extends Comparable<? super T>>
    boolean jeNejvetsi(Collection<? extends T> c, T o)
```

Tuto metodu již můžeme použít pro kolekci libovolných objektů typu `T` nebo jakéhokoliv jejího podtypu. Již při překladu kompilátor ověří, zda třída `T`, popř. nějaká její rodičovská třída, implementuje rozhraní `Comparable`.

Implementace této naší experimentální metody spolu s jednoduchým testem může vypadat následovně.

Příklad 7.4: `Metody` – Metody pracující s kolekcemi

```
import java.util.*;

public class Metody {
    public static <T extends Comparable<? super T>>
                boolean jeNejvetsi(Collection<? extends T> c, T o) {
        if (!c.contains(o)) return false;
        boolean max = true;
        for (T e : c) {
            if (o.compareTo(e) < 0) max = false;
        }
        return max;
    }

    public static void main(String[] args) {
        List<Integer> seznam = Arrays.asList(3, 2, 8, 5, 1, 16, 4,11);

        System.out.println(jeNejvetsi(seznam, 8));
        System.out.println(jeNejvetsi(seznam, 16));
    }
}
```

Výstup programu:

```
false
true
```

7.4 Některé nové kolekce

Java 5.0 přinesla řadu nových tříd pro práci s kolekcemi. V této kapitole si představíme ty, se kterými se můžeme častěji setkat. Nových tříd, představujících některé speciální kolekce, je pochopitelně mnohem více, na ty nám zde ovšem nezbývá prostor. Musí tedy nutně přijít opětovný odkaz na dokumentaci k Java API.

7.4.1 Datová struktura – fronta

V předchozích verzích Javy jsme neměli k dispozici třídu, která by reprezentovala frontu libovolných prvků. Mohli jsme ale použít seznam typu `LinkedList`. To je obousměrně zřetězený seznam. Bylo na nás, zda jsme vkládali prvky na začátek seznamu a z konce seznamu

je odebírali, nebo naopak. Museli jsme pouze dát pozor, abychom vždy vkládali prvky na jeden konec seznamu a odebírali je vždy z druhého konce.

7.4.1.1 Rozhraní `Queue`

Od verze JDK 5.0 máme v balíčku `java.util` k dispozici nové rozhraní `Queue`, které deklaruje základní metody pro práci s libovolnou frontou.

Tabulka 7.1: Metody rozhraní `Queue`

`element()`	Vrátí prvek ze začátku fronty, ale ponechá ho ve frontě. Pokud je fronta prázdná, vyvolá výjimku `NoSuchElementException`.
`offer(E o)`	Vloží prvek `o` do fronty.
`peek()`	Vrátí prvek ze začátku fronty, ale ponechá ho ve frontě. Pokud je fronta prázdná, vrátí `null`.
`pool()`	Vrátí prvek ze začátku fronty a odstraní jej z fronty. Pokud je fronta prázdná, vrátí `null`.
`remove()`	Vrátí prvek ze začátku fronty a odstraní jej z fronty. Pokud je fronta prázdná, vyvolá výjimku `NoSuchElementException`.

Možná Vás už nepřekvapí, že i rozhraní `Queue` je deklarováno jako generické. Jediný typový parametr `E` představuje typ prvků ve frontě.

Toto rozhraní implementuje třída `AbstractQueue`. Ta vytváří konkrétní datovou strukturu a definuje metody pro provádění základních operací s frontou. Stejně jako ostatní typy kolekcí, je i třída `AbstractQueue` implementuje rozhraní `Collection`.

Poznámka:

Rozhraní `Queue` implementuje i třída `LinkedList`. Když jsme ji mohli použít jako frontu dříve, proč bychom nemohli i teď? Navíc můžeme k seznamu typu `LinkedList` přistupovat přes rozhraní `Queue`.

■

7.4.1.2 Třída `PriorityQueue`

Objekt třídy `PriorityQueue<E>` představuje prioritní frontu prvků typu `E`. Tato třída implementuje rozhraní `Queue` a nedefinuje žádné zvláštní metody navíc. Obsahuje několik konstruktorů, které umožňují vytvořit prioritní frontu ze zadané kolekce nebo ze seřazené množiny.

Způsob zařazování prvků do prioritní fronty (resp. způsob určení priority jednotlivých prvků) může být dvojí. Jedním ze způsobů je implementace rozhraní `Comparable<E>` ve třídě `E`, jejíž instance jsou prvky prioritní fronty. V případě, že není tímto způsobem možné prvky porovnat, a určit tak jejich prioritu, vyvolá metoda pro vložení prvku do fronty výjimku `ClassCastException`. Druhým způsobem určení priority prvků je použití komparátoru (tedy objektu implementující rozhraní `Comparator<E>`). Příslušný komparátor předáme prioritní frontě k tomu určeným konstruktorem. Prioritní fronta už neověřuje zda její prvky implementují rozhraní `Comparable`, používá pouze zadaný komparátor.

7.4.2 Kolekce `EnumMap` a `EnumSet`

V balíčku `java.util` najdeme od verze JDK 5.0 dvě nové třídy patřící mezi tzv. kolekce. Jde o kolekce `EnumMap` a `EnumSet`. S těmito kolekcemi se pracuje velmi podobně, jako s ostatními, pouze jsou optimalizovány pro práci s výčtovými typy.

7.4.2.1 Kolekce `EnumMap`

Tato kolekce představuje mapu, jejímiž klíči jsou konstanty nějakého výčtového typu. Hodnotami mohou být libovolné objekty. Pokud se podíváme na deklaraci této třídy, uvidíme, že typový parametr `K` je patřičně ohraničen.

```
public class EnumMap<K extends Enum<K>, V> { ... }
```

Na rozdíl od jiných map, nemusí třída `EnumMap` vytvářet pro uchovávání klíčů složité dynamické datové struktury. Snadno si totiž zjistí, kolik konstant daný výčtový typ `K` obsahuje, a ty může uložit v poli. Navíc třída `EnumMap` nemusí uchovávat jako klíče celé výčtové konstanty, stačí jí čísla jejich pořadí (tj. hodnoty metody `ordinal()`). Tím dosáhne `EnumMap` mnohem vyšší efektivity. K jedné dvojici klíč-hodnota může totiž přistoupit v konstantním čase.

7.4.2.2 Kolekce `EnumSet`

Tato kolekce představuje množinu, jejíž prvky jsou konstanty výčtového typu. Můžeme se opět podívat na její deklaraci:

```
public abstract class EnumSet<E extends Enum<E>> { ... }
```

Zajímavé je, že je tato třída deklarovaná jako abstraktní. Nemůžeme tedy přímo vytvořit její instanci. Pro vytvoření množiny konstant libovolného výčtového typu můžeme použít řadu statických metod této třídy. Mezi častěji používané určitě budou patřit následující metody:

Tabulka 7.2: Metody třídy `EnumSet`

`allOf(Class<E> e)`	Vytvoří množinu obsahující všechny konstanty výčtového typu určeného class-objektem `e`.
`noneOf(Class<E> e)`	Vytvoří prázdnou množinu hodnot výčtového typu určeného class-objektem `e`.
`of(E p, E... o)`	Vytvoří množinu obsahující zadané konstanty výčtového typu `E`.
`range(E od, E do)`	Vytvoří množinu obsahující všechny konstanty výčtového typu `E` ze zadaného rozsahu.

Pokud budeme mít např. výčtový typ

```
public enum DnyVTydnu { PO, UT, ST, CT, PA, SO, NE; }
```

Můžeme vyzkoušet provést následující metody:

Příklad 7.5: `Mnozina` – Použití `EnumSet`

```
import java.util.EnumSet;

public class Mnozina {
    public static void main(String[] args) {
        EnumSet vsechny = EnumSet.allOf(DnyVTydnu.class);
        EnumSet nektere = EnumSet.of(DnyVTydnu.PA, DnyVTydnu.SO,
                                     DnyVTydnu.NE);
        EnumSet rozsah = EnumSet.range(DnyVTydnu.PO, DnyVTydnu.PA);

        System.out.println(vsechny);
        System.out.println(nektere);
        System.out.println(rozsah);
```

```
        }
    }
```

Výstup programu:

```
[PO, UT, ST, CT, PA, SO, NE]
[PA, SO, NE]
[PO, UT, ST, CT, PA]
```

Vidíme, že v proměnné `vsechny` je množina všech prvků výčtového typu, v proměnné `nektere` jsou pouze dny pátek, sobota a neděle (tzv. prodloužený víkend) a v proměnné `rozsah` jsou všechny dny od pondělí do pátku.

8 Formátování výstupu

Jazyk Java zavádí od verze 5.0 zcela novou možnost formátování čísel, řetězců, údajů o datu a čase aj. Základní idea formátování sestává ze dvou hlavních prvků. Prvním z nich je vstupní řetězec, tzv. ***formát***, obsahující veškeré informace o požadovaném tvaru výstupního řetězce. Druhým prvkem je ***seznam vstupních argumentů***, které budou formátovány. Může jít přitom o **primitivní typy**, ale i o **libovolné objekty**. Každý objekt totiž můžeme, použitím metody `toString()`, reprezentovat jako řetězec. Formátování v Javě navíc přímo podporuje i velká čísla (`BigInteger`, `BigDecimal`). Velice snadno také můžeme zformátovat data, která uchovává objekt typu `Calendar`.

Základním předpokladem pro formátování je relativně jednoduchý zápis formátu s přesně danou strukturu. Zavedení této koncepce bylo inspirováno funkcí `printf()`, kterou můžeme znát z programovacího jazyka C. Prvním z parametrů této funkce je právě formát, ostatní jsou vstupní argumenty. Java ovšem, na rozdíl od jazyka C, definuje přesná pravidla pro strukturu formátu a v případě jejich nedodržení vyhazuje výjimky. V jazyku C je neplatný zápis formátu jednoduše ignorován. Tím vytváří Java poměrně robustní a efektivní nástroj pro snadné formátování „téměř libovolných" dat.

V této kapitole si nejprve uvedeme metody, které formátování výstupu umožňují. Dále se podíváme na postavení třídy `Formatter`, která je jádrem pro formátování výstupu. A až se s touto koncepcí blíže seznámíme, budeme se věnovat konkrétním syntaktickým pravidlům zápisu formátu.

8.1 Metody umožňující formátovat výstup

Java definuje pro formátování několik metod, jejímiž parametry jsou právě formát a seznam vstupních argumentů. Mezi nejpoužívanější určitě bude patřit metoda `printf()`, tu nalezneme, kromě jiných, ve třídách `PrintStream` a `PrintWriter`. To jsou běžně používané výstupní proudy. Deklarace těchto metod vypadá následovně:

```
public PrintStream printf(String format, Object... args)
public PrintWriter printf(String format, Object... args)
```

Tyto metody zapíší zformátovaný výstup do příslušného datového proudu a vrátí na něj odkaz. Jelikož i standardní výstup `System.out` je typu `PrintStream`, poskytuje i on možnost formátování pomocí metody `printf()`.

Další hojně používanou metodou pro formátování jistě bude statická metoda `format()` třídy `String`, která vrací řetězec vzniklý zformátováním vstupních argumentů na základě zadaného formátu. Deklarace této metody vypadá podobně, jako deklarace metody `printf()`.

```
public static String format(String format, Object... args)
```

Poznámka:

Možná jste si již všimli, že metody `printf()` a `format()` existují ještě v jedné verzi, ta má navíc jeden parametr typu `Locale`. Tato metoda slouží pro formátování čísel, datumů atp. podle zvyklostí konkrétních zemí světa. To je už ale veliká specialita, která přesahuje

rámec této publikace. Podrobný popis využití této funkce najdete v dokumentaci k Java API. Jeden ukázkový program je také součástí tutoriálu.

■

8.2 Formátovací procesor – třída `Formatter`

Základním stavebním kamenem celé koncepce formátování výstupu v Javě je třída `java.util.Formatter`. Dá se říct, že tato třída definuje jakýsi ***formátovací procesor***, který využívají všechny metody sloužící k formátování nějakých konkrétních vstupů.

Neznamená to ovšem, že bychom nemohli využít přímo tento procesor sami. Třída `Formatter` má dvě skupiny konstruktorů. Ty se liší tím, zda je formátovací procesor vytvořen pro zadaný výstup či nikoliv.

8.2.1 Formátovací procesor bez výstupu

Dva konstruktory třídy `Formatter` umožňují vytvořit formátovací procesor bez určení jeho výstupu. My se budeme zajímat pouze o konstruktor bez parametrů. Druhý z nich má jediný parametr typu `Locale` a jak už jsme zmínili, problematika zvyklostí různých zemí světa přesahuje rámec této publikace.

Nic nám tedy nebrání vytvořit si formátovací procesor bez výstupu.

```
Formatter fp = new Formatter();
```

Formátování potom provedeme zavoláním metody `format()`.

```
fp.format("e = %+10.4f", Math.E);
```

Tím jsme zformátovali konstantu `Math.E` podle předepsaného formátu. Pro získání výstupního řetězce z formátovacího procesoru máme dvě možnosti. Můžeme samozřejmě použít metodu `toString()`. Nebo můžeme použít metodu `out()`, která vrací objekt typu `Appendable`. `Appendable` je rozhraní, které představuje řetězec nebo výstupní datový proud, ke kterému můžeme připojit sekvenci znaků. Toto rozhraní implementují, kromě standardních výstupních datových proudů, i třídy `StringBuilder` a `StringBuffer`, což jsou modifikovatelné řetězce.

Následující příklad ukazuje získání výstupního řetězce z formátovacího procesoru, který nemá předem určený výstup.

Příklad 8.1: `Test1` – Použití formátovacího procesoru 1

```
import java.util.Formatter;

public class Test1 {
    public static void main(String[] args)
                                        throws java.io.IOException {
        Formatter fp = new Formatter();
        fp.format("e = %+10.4f", Math.E);

        String s1 = fp.toString();

        String s2 = fp.out().append(" (Eulerovo číslo)").toString();

        System.out.println(s1);
        System.out.println(s2);
```

```
    }
}
```

Výstup programu:

```
e =    +2,7183
e =    +2,7183 (Eulerovo číslo)
```

Poznámka:
Na syntaktická pravidla zápisu konkrétního formátu se podíváme až v kapitole [8.3/63]. ■

8.2.2 Formátovací procesor s určeným výstupem

Třída `Formatter` definuje celou řadu konstruktorů, které se **liší použitým výstupem**, na který je posílán zformátovaný řetězec. Tímto výstupem může být výstupní datový proud, soubor zadaný názvem, popř. objektem typu `File`, nebo objekt třídy, která implementuje rozhraní `Appendable`.

Použití formátovacího procesoru s určeným výstupem je stejné jako v předchozím příkladě. Jediný rozdíl je ten, že se nemusíme starat o získání výstupního řetězce. Formátovací procesor jej totiž posílá po každém zavolání metody `format()` na určený výstup.

Následující příklad ukazuje použití formátovacího procesoru, jehož výstupem je objekt třídy `StringBuilder`.

Příklad 8.2: `Test2` – Použití formátovacího procesoru 2

```
import java.util.Formatter;

public class Test2 {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Eulerovo číslo: ");

        Formatter fp = new Formatter(sb);
        fp.format("e = %+10.4f", Math.E);

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

Výstup programu:

```
Eulerovo číslo: e =    +2,7183
```

Podobně bychom mohli vytvořit formátovací procesor, jehož výstupem bude standardní výstupní proud `System.out`.

Příklad 8.3: `Test3` – Použití formátovacího procesoru 3

```
import java.util.Formatter;

public class Test3 {
    public static void main(String[] args) {
        Formatter fp = new Formatter(System.out);
        fp.format("Eulerovo číslo: e = %+10.4f", Math.E);
```

```
        }
    }
```

Výstup programu:

```
Eulerovo číslo: e =     +2,7183
```

8.3 Syntaxe zápisu formátu

Jak už víme, formát není nic jiného, než **řetězec**, který splňuje určitá **pravidla**. Spolu s formátem vždy zadáváme i **seznam vstupních argumentů**. Z toho se také odvíjí základní struktura formátu. Formát může obsahovat několik ***entit***, které určují tvar výstupu konkrétních vstupních argumentů. Tyto entity mohou být **kdekoliv uvnitř formátu**. Ta část formátu, která neurčuje žádnou entitu zůstane po zformátování **nezměněná**.

8.3.1 Obecný formát entity

Obecný formát entity můžeme zapsat následovně:

```
%[argument$][značky][šířka][.přesnost]konverze
```

Nepovinný parametr `argument` představuje **index vstupního argumentu** (např. `2$` odkazuje na druhý vstupní argument). Místo indexu argumentu můžeme použít znak `<` (ten uvádíme bez znaku `$`). Taková entita pak odkazuje na stejný vstupní argument, na který odkazovala předešlá entita ve formátu. Pokud vůbec nepoužijeme parametr `argument`, potom každá n-tá entita formátu odpovídá n-tému vstupnímu argumentu.

Nepovinným parametrem `značky` je **množina znaků**, které blíže **specifikují výstupní formát**. Parametr `značky` je závislý na použitém typu konverze, tedy na obsahu parametru `konverze`.

Nepovinným parametrem `šířka` je nezáporné celé číslo určující **minimální délku** výstupního řetězce.

Nepovinným parametrem `přesnost` je nezáporné celé číslo, které obvykle **omezuje shora počet znaků** výstupního řetězce. Pokud je vstupním argumentem řetězec, vyjadřuje tento parametr **maximální délku** výstupního řetězce. Pokud je vstupním argumentem reálné číslo, vyjadřuje tento parametr **počet desetinných míst**, které mají být zobrazeny ve výstupním řetězci. Tento parametr nemůžeme použít, je-li vstupním argumentem, který daná entita formátuje, datum nebo čas (tj. objekt typu `Calendar`).

Jediným povinným parametrem je parametr `konverze`. Jde o znak, popř. dvojici znaků, určující způsob zpracování vstupního argumentu, tedy ***typ konverze***. Množina přípustných znaků je vždy **závislá na datovém typu** zpracovávaného vstupního argumentu.

8.3.2 Typ konverze

Jak už víme, typ konverze je jediný povinný parametr formátovací entity, který určuje **způsob zpracování vstupního argumentu**. A jelikož typy vstupních argumentů mohou být různorodé, rozdělíme množinu znaků, specifikující typ konverze, do několika kategorií podle jejich datového typu. V následujících podkapitolách uvedeme přehled znaků, určujících typ konverze, a jim odpovídající výstupy. U některých typů konverze je možné použít **malé i velké písmeno**. Pokud použijeme velké písmeno, je výstup stejný, jako při použití malého písmena. Pouze je takto získaná výstupní hodnota převedena na velká písmena použitím metody `String.toUpperCase()`.

8.3.2.1 Konverze obecného typu

Tyto typy konverze mohou být použity pro **libovolný typ** vstupního argumentu.

Tabulka 8.1: Typy konverze obecného typu

`b, B`	Pokud je argument `arg` `null`, výstupem je „`false`". Pokud je `arg` typu `boolean` nebo `Boolean`, je výstupem příslušná logická hodnota, tedy „`true`" nebo „`false`". V ostatních případech je výstupem „`true`".
`h, H`	Pokud je argument `arg` `null`, výstupem je „`null`". V ostatních případech je výstupem hodnota výrazu `Integer.toHexString(arg.hashCode())`.
`s, S`	Pokud je argument `arg` `null`, výstupem je „`null`". Pokud `arg` implementuje rozhraní `Formatable`, použije se k výstupu volání metody `arg.formatTo()`. V ostatních případech je výstupem řetězec `arg.toString()`.

Příklad 8.4: `Formatovani` – Formátování objektu obecného typu

```
import java.util.Formatter;

public class Formatovani {
    public static void main(String[] args) {
        Formatter fp = new Formatter(System.out);

        fp.format("Hešovací kód objektu %1$s je %1$h.", "OBJEKT");
    }
}
```

Výstup programu:

```
Hešovací kód objektu OBJEKT je 8a939837.
```

Poznámka:

V dalších příkladech již nebudeme uvádět celou deklaraci třídy `Formatovani`. V každém příkladě pouze uvedeme příkaz `fp.format(...)` a výstup. Všechny příklady z kapitoly [8.3] jsou v této jedné třídě. ■

8.3.2.2 Konverze celočíselných typů

Tento typ konverze je možné použít, je-li vstupním argumentem libovolný **celočíselný primitivní či objektový typ**. Lze jej použít i pro typ `BigInteger`.

Tabulka 8.2: Formátování celočíselných typů

`d`	Výstupem je číslo zapsané v desítkové soustavě.
`o`	Výstupem je číslo zapsané v osmičkové soustavě.
`x, X`	Výstupem je číslo zapsané v šestnáctkové soustavě.

Příklad 8.5: Formátování celých čísel

```
fp.format("Číslo %d se v šestnáctkové soustavě zapisuje %<X.", 123);
```

Výstup programu:

```
Číslo 123 se v šestnáctkové soustavě zapisuje 7B.
```

8.3.2.3 Konverze neceločíselných typů

Tento typ konverze je možné použít, je-li vstupním argumentem **libovolný neceločíselný primitivní či objektový typ.** Lze jej použít i pro typ `BigDecimal`.

Tabulka 8.3: Formátování neceločíselných typů

`e, E`	Výstupem je číslo vyjádřené jako základ a exponent.
`f`	Výstupem je standardní zápis desetinného čísla.
`g, G`	Výstupem je standardní zápis desetinoho čísla. Počet cifer je shora omezen hodnotou parametru `přesnost`.
`a, A`	Výstupem je číslo zapsané v šestnáctkové soustavě vyjádřené jako základ a exponent.

Příklad 8.6: Formátování reálných čísel

```
fp.format("Číslo PI můžeme zapsat následujícími způsoby:
          %.4f, %<e nebo %<.3g.", Math.PI);
```

Výstup programu:

```
Číslo PI můžeme zapsat následujícími způsoby: 3,1416, 3.141593e+00
nebo 3.14.
```

8.3.2.4 Konverze znakových typů

Tento typ konverze je možné použít, je-li vstupním argumentem typ `char`, `byte`, `short`, `Character`, `Byte` nebo `Short`. Lze jej použít i pro typ `int` a `Integer`, pokud toto číslo reprezentuje platný znak ze znakové sady Unicode.

Tabulka 8.4: Typ konverze znakových typů

`c, C`	Výstupem je znak ze znakové sady Unicode.

Příklad 8.7: Formátování znaků

```
fp.format("Znak s kódem %d je %<c.", 112);
```

Výstup programu:

```
Znak s kódem 112 je p.
```

8.3.2.5 Konverze data a času

Tento typ konverze je možné použít, je-li vstupním argumentem objekt typu `Calendar`, `Date`, `long` nebo `Long`.

Ve všech ostatních případech jsme vždy uváděli jako parametr `konverze` jedno malé nebo velké písmeno. Pokud budeme formátovat datum nebo čas, musíme uvést **dva znaky.** Prvním je vždy malé „`t`“ nebo velké „`T`“, podle toho, zda má být výstup vypsán malými nebo velkými písmeny. Druhý znak určuje, jaká **část datumu nebo času** má být výsledkem formátování dané entity. V následujícím přehledu uvedeme pouze znaky, které stojí v určení typu konverze těsně za znakem „`t`“ nebo „`T`“, a výstup, který jim odpovídá.

Tabulka 8.5: Typy konverze údajů o čase

H	Hodina z rozsahu `00 - 23` včetně úvodní nuly.
I	Hodina z rozsahu `01 - 12` včetně úvodní nuly.
k	Hodina z rozsahu `0 - 23` bez úvodní nuly.
l	Hodina z rozsahu `1 - 12` bez úvodní nuly.
M	Minuta z rozsahu `00 - 59` včetně úvodní nuly.
S	Sekundy z rozsahu `00 - 60` včetně úvodní nuly. Pozn.: hodnota `60` je rezervována pro podporu přestupných sekund.
L	Milisekundy z rozsahu `000 - 999` včetně úvodních nul.
N	Nanosekundy z rozsahu `000000000 - 999999999` včetně úvodních nul.
p	Určení dopoledne či odpoledne podle národních zvyklostí. Např. „`am`“, „`pm`“ nebo „`dop.`“ nebo „`odp.`“
z	Číslo časového pásma.
Z	Znakové určení časového pásma.
s	Počet sekund od 1. 1. 1970 00:00:00.
Q	Počet milisekund od 1. 1. 1970 00:00:00.

Příklad 8.8: Formátování času

```
fp.format("Právě je %tk hodin a %<tM minut.",
          java.util.Calendar.getInstance());
```

Výstup programu:

```
Právě je 14 hodin a 40 minut.
```

Tabulka 8.6: Typy konverze údajů o datu

B	Zápis měsíce podle národních zvyklostí. Např. „`prosinec`“ nebo „`December`“.
b	Zkratka zápisu měsíce podle národních zvyklostí. Např. „`XII`“ nebo „`Dec`“.
h	Stejné jako „`b`“.
A	Zápis dne v týdnu podle národních zvyklostí. Např. „`Sunday`“ nebo „`neděle`“.
a	Zkratka zápisu dne v týdnu podle národních zvyklostí. Např. „`Sun`“ nebo „`Ne`“.
C	První dvě cifry ze čtyřciferného zápisu roku.
Y	Čtyřciferný zápis roku.
y	Poslední dvě cifry ze zápisu roku.
j	Číslo dne v roce z rozsahu `000 - 366` včetně úvodních nul.
m	Číslo měsíce v roce z rozsahu `01 - 12` včetně úvodní nuly.
d	Číslo dne v měsíci z rozsahu `01 - 31` včetně úvodní nuly.
e	Číslo dne v měsíci z rozsahu `1 - 31` bez úvodní nuly.

Příklad 8.9: Formátování data 1

```
fp.format("Víte, že dnes je %tj. den v roce?",
          java.util.Calendar.getInstance());
```

Výstup programu:

```
Víte, že dnes je 310. den v roce?
```

Tabulka 8.7: Běžně používané formáty data a času

R	Zápis času ve formátu `%tH:%tM`. Např. `08:59`.
T	Zápis času ve formátu `%tH:%tM:%tS`. Např. `08:59:06`.
r	Zápis času ve formátu `%tI:%tM: %tS %tp`. Např. `08:59:06 DOP`.

D	Zápis data ve formátu `%tm/%td/%ty`. Např. `12/06/05`.
F	Zápis data ve formátu `%tY-%tm-%td`. Např. `2005-12-06`
c	Komplexní zápis data a času ve formátu `%ta %tb %td %tT %tZ %tY`. Např. `Út XII 06 08:59:06 CET 2005`.

Příklad 8.10: Formátování data 2

```
fp.format("Dnešní datum: %tD (%<tF)",
          java.util.Calendar.getInstance());
```

Výstup programu:

```
Dnešní datum: 11/06/05 (2005-11-06)
```

8.3.2.6 Speciální typy konverze

Tabulka 8.8: Speciální typy konverze

%	Výstupem je znak `%`.
n	Výstupem je platformě závislý znak pro zalomení řádku.

8.3.3 Značky

Značky, jak už jistě víme, slouží k **bližší specifikaci formátu výstupního řetězce**. Následující seznam uvádí všechny symboly, které lze použít jako hodnotu parametru `značka`. U každého symbolu je uvedený výčet typů konverze, pro které můžeme danou značku použít, a význam použití této značky.

Tabulka 8.9: Značky použitelné pro určité typy konverzí

-	*všechny*	Zarovná výstup vlevo.
#	*obecné typy*, o, x, X	Pokud je tato značka použita pro obecný typ, závisí výstup na implementaci rozhraní `Formatable`. Pokud je použita pro číselný typ, výstupem je číselná hodnota včetně úvodní nuly.
+	d, o, x, X, d	Výstupem je číslo včetně uvedení znaménka.
mezera	d, o, x, X, d	Výstupem je číselná hodnota včetně úvodní mezery, pokud jde o kladnou hodnotu.
0	d, o, x, X, e, E, f, g, G, a, A	Doplní nevýznamné nuly.
,	o, x, X, e, E, f, g, G	Odděluje řády tisíců, milionů atd. mezerou.
(	d, o, x, X, e, E, f, g, G	Záporná čísla uzavře do závorky a odstraní znaménko.

8.4 Další detaily a chybová hlášení

Možnosti formátování, které jsme doposud uvedli, jsou sice do jisté míry komplexní, v žádném případě však **nepokrývají všechny detaily**, se kterými se můžeme v praxi setkat. Podrobný popis všech detailů by ovšem o mnoho překročil rozsah této publikace. Pokud tedy budeme mít ještě konkrétnější požadavky, než jsme zmínili, odkazuji na dokumentaci třídy `java.util.Formatter`. Věřte, že ji vývojáři zpracovali opravdu precizně.

8.4.1 Chybová hlášení při formátování výstupu

Formátování výstupu v Javě je velice dobře ošetřeno. Pro odlišení různých chybových stavů spojených s formátováním je v JDK 5.0 definováno několik **výjimek**, které mají společného předka `IllegalFormatException`.

Např. pokud formátovací procesor nalezne chybu v uvedení typu konverze, vyvolá výjimku `IllegalFormatConversionException`. Pokud narazí na neznámý symbol na pozici typu konverze, vyvolá výjimku `UnknownFormatConversionException` atd.

Kompletní seznam všech výjimek, které mohou být vyvolány ve spojení s formátováním výstupu opět nalezneme v dokumentaci k Java API. Tam nalezneme i jejich podrobný popis. Bylo by celkem zbytečné na tomto místě uvádět výčet všech výjimek a snažit se o detailní popis. To, jak jistě chápete, není hlavním záměrem této publikace.

9 Některé užitečné třídy v JDK 5.0

Vidíme, že novinek v jazyku Java 5.0 je opravdu mnoho. Tato publikace se samozřejmě snaží ukázat pouze ty nejzásadnější a nejužitečnější, zejména pak pro začínající programátory. V této kapitole se už nebudeme zabývat změnami v syntaxi jazyka Java. Podíváme se na některé změny ve standardně dodávaných knihovnách. Některé třídy byly pozměněny, některé jsou úplně nové. Tuto kapitolu ale berte pouze jako velmi stručný přehled těch tříd, se kterými budeme pracovat nejčastěji.

9.1 Třída `Class` a reflexe

Java umožňuje prostřednictvím získávat různé informace o třídách, jejich proměnných, metodách atd., a také s nimi provádět různé operace, za běhu programu. K tomu slouží sada tříd a rozhraní v balíčku `java.lang.reflect`. Základním stavebním kamenem této konstrukce je třída `java.lang.Class`. Třída `Class` i ostatní třídy a rozhraní byly v Javě 5.0 rozšířeny o následující možnosti:

- získání informací o **generických typech**, jako jsou např. typové parametry, jejich ohraničení atd.; i když jsou tyto informace z třídy při jejím překladu odstraněny, v objektu typu Class zůstávají a lze je využít
- podpora pro **zpracování anotací** za běhu programu; více o anotacích se dozvíte v kapitole [10/75]
- rozpoznávání **výčtových typů** a výčtových konstant od běžných tříd a konstant
- rozpoznání **proměnného počtu parametrů** u metod či konstruktorů
- **rozpoznání typu třídy** (můžeme např. zjistit, zda je třída deklarována jako anonymní či vnitřní)

9.1.1 Třída `Class` je generická třída

I třída `Class` je od verze JDK 5.0 deklarována jako generická. Má jeden typový parametr reprezentující třídu, které přísluší objekt typu `Class`. Deklarace této třídy vypadá následovně.

```
public final class Class<T> { ... }
```

Je asi jasné, že např. objekt `Integer.TYPE` je objekt typu `Class<Integer>`. Samozřejmě i metoda `getClass()` vrátí u nějaké třídy `Trida` objekt typu `Class<Trida>`. K objektu třídy `Class<T>` můžeme také přistupovat prostřednictvím statické konstanty `class`, kterou kompilátor sám přidá při překladu do každé třídy. Hlavní výhodou generičnosti třídy `Class` je opět to, že některé její metody mohou být přesněji definovány. Např. metoda `newInstance()` třídy `Class<T>` nám vrátí objekt typu `T`. Toho můžeme využít i v případě, že neznáme typ `T`, stačí když máme k dispozici objekt typu `Class<T>`.

9.2 Třída Enum

Jak jsme již zmínili v kapitole [6/44], třída `Enum` je společným předkem všech výčtových typů. Je součástí balíčku `java.lang` a má poněkud zvláštní postavení. Nemůžeme totiž sami vytvořit třídu, která je jejím potomkem. Potomkem této třídy může být pouze „třída" definovaná přímo jako výčtový typ. Zajímavé je, že i tato třída je definována jako generická. Její deklarace vypadá následovně.

```
public abstract class Enum<E extends Enum<E>> { ... }
```

Zde typový parametr `E` může představovat pouze výčtový typ, což je patrné z ohraničení typového parametru.

Třída `Enum` obsahuje několik užitečných metod pro práci s výčtovými typy. V následujícím přehledu si uvedeme některé často používané.

Tabulka 9.1: Metody třídy `Enum`

`name()`	Vrátí název výčtové konstanty.
`ordinal()`	Vrátí počadí výčtové konstanty v její deklaraci.
`compareTo(E o)`	Porovná výčtovou konstantu se zadanou konstantou `o`.
`getDeclaringClass()`	Vrátí class-objekt odpovídající výčtovému typu.
`valueOf(Class<T> enumType, String name)`	Vrátí konstantu výčtového typu ze zadaného výčtového typu a názvu konstanty.

Možná jste si již všimli, že třída `Enum` překrývá metodu `clone()` ze třídy `Object`. Tato metoda ovšem vždy vyvolá výjimku `CloneNotSupportedException`. Tím je zajištěno, že se jakákoliv výčtová konstanta může vyskytovat pouze v jedné instanci. Proto je možné použít pro porovnání výčtových konstant operátor `==`.

9.3 Třídy Math a StrictMath

Třída `Math` obsahuje metody pro provádění základních často používaných matematických operací. Mezi ty nejčastěji používané patří metody pro práci s goniometrickými funkcemi (`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`), pro výpočet přirozeného logaritmu (`log()`), pro výpočet obecné mocniny (`pow()`), pro zaokrouhlování (`round()`), pro převody stupňů na radiány a opačně (`toDegrees()`, `toRadians()`) atd.

Ve verzi JDK 5.0 přibyly nově funkce pro výpočet hyperbolických goniometrických funkcí (`sinh()`, `cosh()`, `tanh()`), pro výpočet dekadického logaritmu (`log10()`), pro výpočet třetí odmocniny (`cbrt()`), pro výpočet hodnoty funkce signum (`signum()`) atd. Možná využijete i novou funkci `hypot(x, y)`, která vrací hodnotu výrazu `sqrt(x2 + y2)` bez nebezpečí přetečení.

Třída `Math` obsahuje samozřejmě i další funkce, některé jsou často používané, jiné méně, ale vždy máme při ruce dokumentaci k Java API, kde najdeme úplný popis všech funkcí, které nám tato třída nabízí.

Stejné metody jako třída `Math` poskytuje i třída `StrictMath`. Metody z třídy `StrictMath` zaručují bitově totožný výsledek na všech platformách (zatímco metody třídy `Math` mohou být optimalizovány na rychlost a v důsledku toho poskytovat na různých platformách mírně odlišné výsledky). Zájemce o tuto problematiku odkazuji na dokumentaci jazyka Java – klíčové slovo `strictfp`.

9.4 Třída `StringBuilder`

Možná jste se již setkali s použitím třídy `StringBuffer`. Objekt této třídy představuje řetězec, který na rozdíl od řetězce typu `String` můžeme modifikovat. Od verze JDK 5.0 máme k dispozici novou třídu `StringBuilder`, která se na první pohled velmi podobá třídě `StringBuffer`. Tyto třídy nám umožňují ke stávajícímu řetězci připojovat další řetězce, měnit znaky na určité pozici v řetězci, nahradit části stávajícího řetězce jinými řetězci, nebo ze stávajícího řetězce mazat znaky či podřetězce.

Třída `StringBuilder` byla přidána, aby nahradila použití třídy `StringBuffer` na místech, kde s objektem této třídy pracuje pouze jedno vlákno. Třída `StringBuilder` tedy nezajišťuje synchronizaci při použití jejího objektu ve více vláknech. V situacích, kdy pracujeme s objektem typu `StringBuilder` pouze v jednom vlákně, je použití této třídy rychlejší, než použití třídy `StringBuffer`.

9.5 Třída `Scanner`

Nová třída `Scanner` v JDK 5.0 nám umožňuje získávat hodnoty primitivních typů a řetězce ze zadaného vstupního řetězce či datového proudu použitím regulárních výrazů. Pro práci s regulárními výrazy slouží třídy `Pattern` a `Matcher` z balíčku `java.util.regex`. Tyto třídy jsou součástí Java API od verze 1.4. Třída `Pattern` představuje regulární výraz a třída `Matcher` obsahuje sadu metod pro zpracování řetězce na základě zadaného regulárního výrazu. Konkrétním zápisem regulárních výrazů se zde nebudeme zabývat. Tuto ne příliš složitou záležitost naleznete v dokumentaci k Java API.

9.5.1 Vytvoření objektu třídy `Scanner`

K vytvoření objektu třídy `Scanner` máme několik možných konstruktorů. Typy jejich parametrů se liší použitým vstupem. Můžeme vytvořit `Scanner`, který čte data ze souboru (konstruktor s parametrem typu `File`), ze vstupního datového proudu (konstruktor s parametrem typu `InputStream`) nebo z řetězce (konstruktor s parametrem typu `String`). K dispozici jsou ještě další konstruktory. Některé mají i druhý parametr, který určuje kódování (implicitně je použito kódování Unicode).

Poznámka:

Práce s třídou `Scanner` je obdobná jak pro čtení dat z řetězce, tak pro čtení dat ze souboru či datového proudu. Pro jednoduchost ukázkových příkladů budeme v následujících podkapitolách vždy používat jako vstup řetězec typu `String`.

■

9.5.2 Jednoduché použití

Pokud budeme chtít ze vstupního řetězce získávat celočíselné hodnoty nebo slova a víme, že jsou odděleny pouze neviditelnými znaky (mezera, konec řádku, tabulátor, ...), nemusíme žádný regulární výraz použít.

Ke zjištění, zda se vyskytuje ve vstupu ještě další číslo použijeme metodu `hasNextInt()` a pro jeho načtení použijeme metodu `nextInt()`. Pro čtení slov, resp. řetězců, slouží dvojice metod `hasNext()` a `next()`.

Příklad 9.1: `Cteni` – **Čtení řetězců 1**

```
import java.util.Scanner;
import java.util.regex.*;

public class Cteni {
    public static void cisla() {
        String vstup = "91 308 1 6 1805 94 5056 4 56 254";
        Scanner s = new Scanner(vstup);
        while (s.hasNextInt()) {
            System.out.print(s.nextInt() + ", ");
        }
        System.out.println();
    }
    public static void slova() {
        String vstup = "Jestli ten Chytrý Scanner " +
                       "rozpozná všechna slova.";
        Scanner s = new Scanner(vstup);
        while (s.hasNext()) {
            System.out.print(s.next() + ";  ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        cisla();
        slova();
    }
}
```

Výstup programu:

```
91, 308, 1, 6, 1805, 94, 5056, 4, 56, 254,
Jestli;  ten;  Chytrý;  Scanner;  rozpozná;  všechna;  slova.;
```

Podobně můžeme využít i metody `hasNextLong()`, `hasNextDouble()`, `hasNextBoolean()` a `nextLong()`, `nextDouble()`, `nextBoolean()` atd. Kromě čtení primitivních číselných typů podporuje třída `Scanner` i čtení čísel typu `BigInteger` nebo `BigDecimal`.

Načítání hodnot z jednoho vstupu není ale nijak omezeno na jeden typ. Ve vstupním řetězci bychom např. mohli kombinovat čísla a slova a při načítání bychom mohli kombinovat metody `hasNextInt()` a `hasNext()`.

9.5.3 Čtení řetězců odpovídajících zadanému regulárnímu výrazu

Pro načítání libovolných řetězců oddělených neviditelnými znaky jsme použily metody `hasNext()` a `next()` bez parametrů. Pokud budeme chtít načítat pouze řetězce, které odpovídají nějakému regulárnímu výrazu, použijeme stejné metody s jedním parametrem. Tím je regulární výraz, který můžeme zadat buď jako řetězec, nebo jako objekt typu `Pattern`. Samotné načítání je pak stejné jako v předchozím příkladě.

Následující příklad ukazuje načítání slov, která mají nejméně 1 a nejvíce 7 znaků. Všimněte si, že jakmile narazí `Scanner` na první řetězec, který neodpovídá zadanému regulárnímu výrazu, skončí.

Příklad 9.2: `Cteni` – **Čtení řetězců 2**

```
import java.util.Scanner;
import java.util.regex.*;
```

```
public class Cteni {
    public static void vyrazy() {
        String vstup = "Jestli ten Chytrý Scanner " +
                       "rozpozná všechna slova.";
        Scanner s = new Scanner(vstup);
        Pattern p = Pattern.compile(".{1,7}");
        while (s.hasNext(p)) {
            System.out.print(s.next(p) + ";  ");
        }
        System.out.println();
    }

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

Výstup programu:

```
Jestli;  ten;  Chytrý;  Scanner;
```

9.5.4 Čtení řetězců oddělených zadaným regulárním výrazem

V předchozím příkladě jsme určovali, jakého tvaru musí být řetězce, které načítáme. K jejich oddělení byly ale opět použity neviditelné znaky. Opačným případem je situace, kdy chceme načítat libovolné řetězce, které jsou ve vstupu odděleny námi definovaným oddělovačem (resp. výrazem). Pro určení tohoto oddělovače slouží ve třídě `Scanner` metoda `useDelimiter()`, jejímž parametrem je regulární výraz zadaný buď jako řetězec, nebo jako objekt třídy `Pattern`.

Následující příklad předvádí načítání libovolných řetězců, které jsou ve vstupu odděleny tečkou, čárkou nebo středníkem. Za těmito znaky může být jedna mezera.

Příklad 9.3: `Cteni` – Čtení řetězců 3

```
import java.util.Scanner;
import java.util.regex.*;

public class Cteni {
    public static void oddelovac() {
        String vstup = "3, 6, 1.2, 15; ahoj, nazdar; 6+5; a.b";
        Scanner s = new Scanner(vstup);
        Pattern p = Pattern.compile("[.,;]( )?");
        s.useDelimiter(p);
        while (s.hasNext()) {
            System.out.print(s.next() + ";  ");
        }
        System.out.println();
    }

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

Výstup programu:

```
3; 6; 1; 2; 15; ahoj; nazdar; 6+5; a; b;
```

Tím jsme ale nevyčerpali všechny možnosti, které nám třída `Scanner` nabízí. Zde bylo místo pouze na uvedení těch nejčastěji používaných. Kompletní popis třídy `Scanner` najdete samozřejmě v dokumentaci k Java API.

10 Anotace

Posledním rozšířením jazyka Java, kterému se budeme věnovat, jsou anotace. Java zavádí od verze 5.0 zcela nový způsob zápisu ***anotací***, neboli ***metadat***, přímo do zdrojového kódu. Nejde ale v žádném případě o komentáře, i když by je anotace s trochou úsilí klidně mohly zastoupit. Velmi snadno bychom anotace mohli využít k vytváření programové dokumentace. Využití anotací ovšem nabízí mnohem více, než pouhé komentování zdrojového kódu. Anotace mohou využít pro své účely např. různé vývojové nástroje.

Použití anotací v programu vede k **deklarativnímu stylu programování**. Konkrétní anotací určujeme její význam a význam jí označené deklarace, ale neuvádíme, jakým přesným způsobem má být anotace zpracována. K tomu už jsou určeny jiné nástroje a aplikace. Takto můžeme anotace využít např. pro automatické vytváření pomocných souborů, které jsou závislé na konkrétním zdrojovém kódu. Typicky může jít o různé konfigurační soubory. Ty může vytvořit speciální program (**anotační procesor** – *Annotation Processor*), který na základě anotací ve zdrojovém kódu zjistí, co má být obsahem daného souboru.

Některá další využití anotací si uvedeme v závěru kapitoly. Nejdříve se blíže seznámíme s použitím anotací v programu a s deklarací vlastních anotací. Také se podíváme na to, jakými způsoby můžeme vlastní anotace zpracovávat.

10.1 Postavení anotací v jazyku Java

Anotace mají v Javě dvojí postavení. V době psaní zdrojového kódu s nimi pracujeme jako se **značkami**, které uvádíme před různé deklarace. Některé anotace mohou mít ještě doplňující **parametry**, které blíže specifikují význam značky.

Během zpracování anotací s nimi ale pracujeme jako s **objekty**. Anotace jako takové jsou vlastně **speciální rozhraní** a jejich parametry jsou **metody**, kterými můžeme zjistit právě hodnoty jednotlivých parametrů zapsaných ve zdrojovém kódu. Zápis konkrétní anotace ve zdrojovém kódu s uvedením jejích parametrů představuje vlastně **instanci této anotace**.

Společným předkem všech anotací je rozhraní `java.lang.annotation.Annotation`. Toto rozhraní ale není anotace. Samotná deklarace anotací je ale trochu odlišná (viz kapitola [10.3/77]). Nemůžeme totiž sami vytvořit potomka rozhraní `Annotation`. Pokud bychom explicitně definovali potomka rozhraní `Annotation`, půjde o běžné rozhraní, nikoliv o anotaci. Můžeme dokonce definovat potomka anotace, v takovém případě také ale nepůjde o anotaci, nýbrž o běžné rozhraní.

10.2 Použití anotací v programu

Pokud se nyní budeme zabývat anotacemi pouze na úrovni použití v programu, nikoliv na jejich zpracování, můžeme si pod nimi představit jakési značky, které mohou mít, ale nemusí, parametry. Anotace můžeme použít pro označení jakékoliv deklarace. Může jít o deklaraci **třídy**, **vnitřní třídy**, **rozhraní**, **konstruktoru**, instanční či statické **metody**, instanční, statické či lokální **proměnné**, **konstanty výčtového typu**, **parametru metody**, **anotace** a jejích parametrů nebo **balíčku** (viz [10.2.1/76]).

Zápis jednoduché anotace ve zdrojovém kódu si ukážeme na příkladu použití anotace `@Deprecated`. Jde o standardní anotaci deklarovanou v balíčku `java.lang`, která slouží k označení zastaralých (zavržených) tříd či metod.

Příklad 10.1: `Zastarala` – Použití anotací v programu

```
@Deprecated
public class Zastarala {
    ...

    @Deprecated
    public void nejakaMetoda() { ... }
}
```

Vidíme, že zápis anotace je vždy uvozen znakem `@`. Není ale podmínkou tento znak uvádět těsně před název anotace. Teoreticky mezi nimi může být libovolný počet mezer či jiných netisknutelných znaků. Nebývá to ale zvykem.

Anotace vždy stojí na pozici modifikátoru deklarace. Na pořadí anotací a ostatních modifikátorů však nezáleží. U jedné deklarace můžeme klidně uvést několik různých anotací. Je pouze dobrým zvykem uvádět je jako první a na samostatném řádku. Není to ale podmínkou. Klidně bychom mohli deklaraci třídy `Zastarala` zapsat následovně:

```
public @Deprecated class Zastarala {
    ...

    public @Deprecated void nejakaMetoda() { ... }
}
```

Význam zůstane úplně stejný jako v předchozím případě.

Již v úvodu kapitoly padla zmínka o existenci anotací s parametry. Pro jejich zápis platí stejná pravidla jako pro anotace bez parametrů. Hodnoty jednotlivých parametrů se zapisují do kulatých závorek za název anotace. Představme si, že máme k dispozici anotaci `@Verze`, která má dva parametry `major` a `minor` a oba tyto parametry jsou číselné hodnoty. Zápis takové anotace bude vypadat následovně:

```
@Verze(major=1, minor=1)
public class TestAnotace {
    ...
}
```

Jednotlivé parametry anotace oddělujeme čárkami. U každého parametru uvádíme jeho přesný název a hodnotu oddělujeme znakem `=`.

Mohou existovat i anotace, které mají parametr typu pole. Na tomto místě není ale nutné zabývat se dopodrobna přesnou syntaxí zápisu jednotlivých typů anotací. Tato úvodní část má pouze nastínit představu základního použití anotací v programu. Více o jednotlivých typech anotací a o možnostech jejich deklarace se dozvíme v části [10.3/77].

10.2.1 Použití anotací pro označení balíčků

Označování balíčků anotacemi přináší jednu drobnou odlišnost oproti označování ostatních deklarací. Jak víme, informaci o názvu balíčku uvádíme v každém souboru `.java`. Do kterého z nich ale uvést seznam anotací?

Pokud vybereme náhodně jeden soubor `.java`, do kterého uvedeme před deklaraci balíčku seznam anotací, zahlásí překladač chybu. Pro označení balíčku musíme vytvořit soubor

`package-info.java`. Tento soubor obsahuje pouze deklaraci balíčku. A pouze v tomto souboru je možné uvést i anotace náležející právě tomuto balíčku.

V JDK 5.0 je dokonce možné tento soubor použít pro umístění dokumentačního komentáře, který se týká konkrétního balíčku. V předchozích verzích Javy jsme museli pro tento účel vytvářet soubor `package.html`. Nic nám sice nebrání použít i nyní pro psaní dokumentace k balíčku soubor `package.html`, ale pokud chceme nějaký balíček zároveň okomentovat a také označit anotacemi, je zbytečné pro tyto dva účely vytvářet dva různé soubory.

10.3 Deklarace vlastní anotace

Syntaxe deklarace anotací je poměrně jednoduchá a připomíná deklaraci rozhraní. V deklaraci anotace také uvádíme pouze seznam jejích členů (parametrů), ale neuvádíme žádný výkonný kód. V hlavičce deklarace anotace používáme rovněž klíčové slovo `interface`. Obecně může deklarace anotace vypadat následovně:

```
Modifikátory @interface Název {
    TypParametru NázevParametru1() default VýchozíHodnota;
    TypParametru NázevParametru2();
    ...
}
```

Jako modifikátor anotace můžeme použít klíčové slovo `public`. Pokud jej neuvedeme, můžeme takovou anotaci použít pouze v rámci balíčku. U anotací **nemůžeme** použít modifikátory `protected` a `private`.

Jako modifikátory anotace mohou být použity opět anotace. Dokonce můžeme k označení anotace `@A` použít opět anotaci `@A`. Jelikož sama anotace nedefinuje způsob svého zpracování, nejde v tomto případě o rekurzi. Záleží už jen na programu, který anotace zpracovává, jak s jednotlivými označeními naloží.

Poznámka:

Všimněte si znaku `@` umístěného těsně před klíčovým slovem `interface`. Tímto způsobem se uvozuje deklarace anotace. Mezi znakem `@` a slovem `interface` může být libovolný počet mezer, či jiných netisknutelných znaků. Obvykle je ale zvykem zapisovat je těsně za sebe.

■

Jako typ parametru anotace **nemůžeme** použít jakýkoliv typ. Můžeme použít

- primitivní typy (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`),
- typ `String`,
- typ `Class` (včetně uvedení typových parametrů),
- libovolný výčtový typ,
- jinou anotaci,
- pole uvedených typů.

Klíčové slovo `default` následované výchozí hodnotou daného parametru anotace není povinné. V takovém případě je ale nutné při použití takovéto anotace hodnotu příslušného parametru zadat.

Deklarace anotací má ještě některá specifika:

- Nemůžeme deklarovat anotaci ani její parametry jako generické,

- V deklaraci anotace může být vnořena deklarace rozhraní, třídy, výčtového typu nebo statických či instančních proměnných.
- S metodami, které anotace dědí z rozhraní `Annotation`, nemůžeme v žádném případě pracovat jako s parametry anotace.

10.3.1 Anotace bez parametrů

Anotace bez parametrů použijeme pro označení nějaké skutečnosti, kterou ale není potřeba nijak blíže specifikovat. Proto se někdy takové anotace označují jako **značky** (*markers*) nebo **značkovací anotace** (*marker annotations*). Jejich deklarace se liší pouze tím, že mají prázdné tělo.

Typickým použitím anotace bez parametrů by bylo nahrazení rozhraní `java.io.Serializable` anotací `@Serializable`. Toto rozhraní totiž slouží pouze k informaci, že objekty dané třídy mohou být serializovány.

Příklad 10.2: `Databaze` – Použití značkovací anotace

```
public @interface Serializable {}

@Serializable
public class Databaze {
    ...
}
```

Poznámka:

Při použití anotace bez parametrů můžeme použít i zápis se závorkami – `@Serializable()`. Závorky nejsou ale v tomto případě povinné. ■

10.3.2 Anotace s parametry

Často ale potřebujeme nejen sdělit, co danou anotací označujeme, ale také nějak toto sdělení blíže specifikovat. Potřebujeme tedy anotaci parametrizovat. Deklaraci parametrizované anotace si můžeme ukázat na příkladu anotace označující verzi třídy.

Příklad 10.3: `Verze` – Příklad anotace s parametry

```
public @interface Verze {
    int major();
    int minor() default 0;
}
```

Vidíme, že parametr `minor` má výchozí hodnotu nastavenou na 0, nemusíme jej tedy při použití této anotace uvádět.

```
@Verze(major=1)
public class TestAnotace {
    ...
}
```

V tomto případě má parametr `major` hodnotu 1 a parametr `minor` hodnotu 0. Pokud použijeme tuto anotaci následujícím způsobem, budou mít oba parametry hodnotu 1.

```
@Verze(major=1, minor=1)
public class TestAnotace {
    ...
}
```

10.3.2.1 Anotace s jedním parametrem

Pokud deklarujeme anotaci pouze s jedním parametrem, můžeme použít odlišné deklarace tohoto parametru. To nám usnadní a zpřehlední použití této anotace. Uvažujme například anotaci `@Autor`.

Příklad 10.4: `Autor` – Příklad anotace s jedním parametrem

```
public @interface Autor {
    String jmeno();
}
```

Pokud bychom deklarovali anotaci tímto obvyklým způsobem, musíme při jejím použití uvést spolu s hodnotou parametru i jeho název.

```
@Autor(jmeno="Franta Novak")
public class TestAnotace {
    ...
}
```

Pokud ale místo názvu parametru použijeme slovo `value`, bude použití anotace jednodušší.

```
public @interface Autor {
    String value();
}

@Autor("Franta Novak")
public class TestAnotace {
    ...
}
```

Nic nám nebrání i v tomto případě uvést `@Autor(value="Franta Novak")`, ale je to naprosto zbytečné.

Parametr s názvem `value` můžeme ale použít i u anotací s více parametry. Pokud ale při použití takové anotace zadáváme i jiné parametry než parametr `value`, musíme i u něj použít úplný zápis.

10.3.2.2 Pole jako typ parametru anotace

Někdy také můžeme mít požadavek, aby jeden parametr anotace mohl mít více hodnot. Nabízí se tedy použití pole. Nejde ovšem o běžné pole. Jde o seznam hodnot uvedený ve složených závorkách a oddělených čárkou. Nijak ale takovéto pole neinicializujeme. Nemůžeme ani použít žádné metody, které nám pole vytvoří.

Vše možná bude patrné na příkladu anotace `Autori`, která bude představovat seznam autorů.

Příklad 10.5: `Autori` – Pole jako typ parametru anotace

```
public @interface Autori {
    String[] value();
}

@Autori( {
    "Franta Novak",
    "Pepa Kadlec"
} )
public class TestAnotace {
    ...
}
```

Poznámka:

Pokud chceme uvést pouze jednu hodnotu parametru anotace, jehož typem je pole, nemusíme ji uzavírat do složených závorek. ■

10.3.2.3 Anotace jako typ parametru jiné anotace

Tato konstrukce může být užitečná v situacích, kdy chceme vytvořit nějakou hierarchickou strukturu anotace. Můžeme tedy např. vytvořit jedinou anotaci `Info`, která v sobě bude zahrnovat jak informaci o autorech, tak informaci i o verzi (a klidně by mohla zahrnovat i něco dalšího).

Příklad 10.6: `Info` – Anotace jako typ parametru jiné anotace

```
public @interface Info {
    Autor[] autori();
    Verze verze() default @Verze(major=1);
}
```

V tomto případě jsou oba parametry anotace typu jiné anotace. Parametr autori je navíc typu pole. Podobným způsobem by mohla mít i anotace Autor několik parametrů, jejímiž typy budou další anotace. Tímto způsobem můžeme jednou anotací zapsat určitou, logicky členěnou, strukturu informací.

```
@Info(
    autori={
        @Autor("Franta Novak"),
        @Autor("Pepa Kadlec")
    },
    verze=@Verze(major=1, minor=1)
)
public class TestAnotace {
}
```

10.4 Standardní anotace

I ve standardních knihovnách na nás čeká několik anotací. Zvláště důležité pro zpracování anotací, ať už za běhu programu nebo ze zdrojového kódu, jsou čtyři anotace z balíčku `java.lang.annotation`. Tyto anotace, jak uvidíme dále, můžeme použít pouze k označování jiných anotací. Proto se jim také někdy říká ***metaanotace***. Na další tří anotace narazíme také v balíčku `java.lang`. Dvě z nich zpracovává překladač a jednu program `javadoc`.

10.4.1 Metaanotace

Jak jsme již uvedli, jde o anotace z balíčku `java.lang.annotation`. Všechny tyto anotace slouží k označení jiných anotací a jsou informací pro překladač, jakým způsobem má s deklarovanými anotacemi nakládat.

10.4.1.1 Retention

Určuje, jak dlouho má být uchovávána informace o anotaci, neboli kam až má být informace o použité anotaci promítnuta. Má jeden parametr `value` typu `RetentionPolicy`. `RetentionPolicy` je výčtový typ deklarovaný v balíčku `java.lang.annotation`. Má následující tři konstanty:

Tabulka 10.1: Typy působnosti anotací

`SOURCE`	Anotace označená `@Retention(RetentionPolicy.SOURCE)` bude moci být zpracována pouze ze zdrojového kódu.
`CLASS`	Anotace označená `@Retention(RetentionPolicy.CLASS)` bude moci být zpracována z přeloženého bajtkódu, nikoliv však za běhu programu.
`RUNTIME`	Anotace označená `@Retention(RetentionPolicy.RUNTIME)` bude moci být zpracována za běhu programu, z bajtkódu i ze zdrojového kódu.

Pokud naší anotaci vůbec neoznačíme anotací `@Retention`, bude se chovat, jako by byla označená `@Retention(RetentionPolicy.CLASS)`.

Poznámka:

Všechny anotace v balíčku `java.lang.annotation` jsou označeny `@Retention(RetentionPolicy.RUNTIME)`.

■

10.4.1.2 Target

Informuje překladač o tom, které elementy (deklarace) programu je možné anotací označit. Má jeden parametr `value` typu `ElementType[]`. `ElementType` je výčtový typ deklarovaný v balíčku `java.lang.annotation`. Má následující konstanty:

Tabulka 10.2: Typy deklarací, u kterých je možno anotaci použít

`ANNOTATION_TYPE`	Anotací označenou `@Target(ElementType.ANNOTATION_TYPE)` můžeme označit opět pouze anotaci. Půjde tedy o metaanotaci.
`CONSTRUCTOR`	Anotací označenou `@Target(ElementType.CONSTRUCTOR)` můžeme označit pouze konstruktory.
`FIELD`	Anotací označenou `@Target(ElementType.FIELD)` můžeme označit instanční či statické proměnné, a také konstanty výčtového typu.
`LOCAL_VARIABLE`	Anotací označenou `@Target(ElementType.LOCAL_VARIABLE)` můžeme označit pouze lokální proměnné.
`METHOD`	Anotací označenou `@Target(ElementType.METHOD)` můžeme označit pouze instanční či statické metody.
`PACKAGE`	Anotací označenou `@Target(ElementType.PACKAGE)` můžeme označit pouze deklarace balíčků.

`PARAMETER`	Anotací označenou `@Target(ElementType.PARAMETER)` můžeme označit pouze parametry metod.
`TYPE`	Anotací označenou `@Target(ElementType.TYPE)` můžeme označit deklarace tříd, rozhraní, anotací nebo výčtových typů.

Jelikož parametr anotace `@Target` je typu pole, můžeme použít i zápis `@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})`. Takto označenou anotaci můžeme použít k označování konstruktorů i metod.

Pokud naší anotaci vůbec neoznačíme anotací `@Target`, budeme jí moci použít k označení libovolného elementu.

Poznámka:

Všechny anotace v balíčku `java.lang.annotation` jsou označeny `@Target(ElementType.ANNOTATION_TYPE)`.

■

10.4.1.3 Documented

Jako jsme byli zvyklí nalézt v dokumentaci JDK 5.0 různé informace o třídách, jejich metodách, proměnných atd., můžeme v dokumentaci nalézt u jednotlivých položek i informace o tom, jakými anotacemi jsou označeny. Aby ale tyto anotace mohly být v dokumentaci viditelné, musejí být samy označeny anotací `@Documented`.

Zajímavé je, že i anotace `@Documented` je sama označena anotací `@Documented`. Je to proto, aby i její použití bylo patrné v dokumentaci.

Poznámka:

I všechny ostatní anotace v balíčku `java.lang.annotation` jsou označeny `@Documented`.

■

10.4.1.4 Inherited

Tato anotace má poněkud zvláštní uplatnění. Pokud použijeme k označení třídy nějakou anotaci `@A`, která je označená anotací `@Inherited`, stane se tato anotace součástí všech dceřinných tříd. Dotaz, zda je třída označená anotací `@A`, bude úspěšný i v případě, že je touto anotací označená jakákoliv rodičovská třída.

Jedinou výjimkou jsou v tomto případě rozhraní. Pokud použijeme anotaci `@A` k označení rozhraní, nijak se tato anotace neprojeví u tříd, které toto rozhraní implementují.

10.4.2 Anotace v balíčku `java.lang`

10.4.2.1 Override

Tato anotace bez parametrů slouží k označování metod (je označená anotací `@Target(ElementType.METHOD)`) a říká překladači, že metoda překrývá stejnojmennou metodu z rodičovské třídy. Pokud tomu tak není (tj. i v případě, že metoda jinou metodu pouze přetěžuje), ohlásí překladač chybu *‚method does not override a method from its superclass‘*.

Překladač také ohlásí chybu v případě, že metoda označená anotací `@Override` pouze implementuje metodu nějakého rozhraní, ale nepřekrývá žádnou metodu z rodičovské třídy.

10.4.2.2 Deprecated

Tato anotace, rovněž bez parametrů, slouží k označení libovolného elementu programu, který je považován za zavržený.

Typickým příkladem použití této anotace jsou metody implementující nějaké konkrétní algoritmy. Pokud v nové verzi programu vytvoříme novou metodu implementující lepší algoritmus, označíme původní metodu jako zavrženou (*deprecated*). Pokud takto označenou metodu někde v programu použijeme, nebo ji ve zděděné třídě překryjeme, upozorní nás překladač, že používáme zavrženou metodu.

Varování překladač nevydá pouze v případě, že použijeme zavrženou metodu či třídu uvnitř stejné třídy.

10.4.2.3 SuppressWarnings

Tato anotace má jeden parametr value, jehož typem je pole `String[]`. Hodnotami tohoto parametru mohou být řetězce, které specifikují určité typy varovných hlášení. Překladač tato varovná hlášení při překladu potlačí. Můžeme použít následující řetězce pro potlačení určitých typů chybových hlášení:

Tabulka 10.3: Potlačení varovných hlášení

`deprecation`	Použití třídy nebo metody, která je označená anotací `@Deprecated`.
`unchecked`	Použití typově neošetřených operací.
`fallthrough`	Příkaz `switch` přechází do další části `case` bez přerušení příkazem `break`.
`path`	Parametry překladače `classpath` a `sourcepath` obsahují nesprávnou cestu.
`serial`	Ve třídě označené pro serializaci chybí deklarace statické proměnné `serialVersionUID`.
`finally`	Blok příkazů `finally` nemůže být dokončen bezchybně.
`all`	Všechna chybová hlášení.

Pokud touto anotací označíme třídu, potlačí překladač veškerá varovná hlášení týkající se překladu celé třídy. Můžeme ale touto anotací označit např. pouze některé metody. Překladač potom potlačí varovná hlášení vyplývající z překladu dané metody. Při překladu ostatních metod stejné třídy zůstanou všechna hlášení nepotlačena.

10.5 Zpracování anotací

V této chvíli už máme představu o tom, jak použít anotace v programu, nebo jak si nadefinovat anotace vlastní. V této části se budeme věnovat jejich zpracování. Pokud budeme v programu používat sadu nějakých připravených anotací, budeme mít vždy k dispozici i nástroj pro jejich zpracování. Jak ale víme, Java nabízí možnost definovat vlastní anotace, musí tedy nutně existovat i možnost vytvořit si vlastní nástroj pro zpracování anotací.

Zpracování anotací může probíhat ve třech úrovních:

- zpracování anotací **za běhu programu**
- zpracování **zdrojového kódu**
- zpracování přeloženého **bajtkódu**

V následujících částech si ukážeme jaké třídy a rozhraní nám umožňují anotace zpracovávat, resp. vytvářet vlastní nástroje pro jejich zpracování. Jelikož jsou možnosti využití

anotací značně rozsáhlé, neočekávejte od této kapitoly žádné názorné příklady. Z důvodu jejich rozsahu jsou uvedeny v tutoriálu.

10.5.1 Zpracování za běhu programu

Anotace můžeme za běhu programu zpracovávat pomocí reflexe (**Reflection API**). Sem patří rozhraní a třídy z balíčku `java.lang.reflection` a třídy `java.lang.Class` a `java.lang.Package`. Při práci s anotacemi také využijeme rozhraní `Annotation` z balíčku `java.lang.annotation`. V dalším textu nebudeme již uvádět názvy balíčků u tříd, které tvoří obsah zde zmíněných balíčků.

V kapitole [9.1/69] jsme si ukázali, jak získat tzv. **class-objekt** dané třídy. Za objektem třídy `Class` si můžeme představit jakousi programovou reprezentaci vlastní deklarace třídy. Může jej tedy využít ke zjištění veškerých informacích o třídě, o jejích metodách, konstruktorech a položkách, můžeme zjišťovat informace vyplývající z deklarace generického typu, případně výčtového typu, můžeme zjišťovat informace o rodičovské třídě a o implementovaných rozhraních, o balíčku, ve kterém je třída umístěna, a v neposlední řadě můžeme také zjistit veškeré informace o použitých anotacích.

Pomocí reflexe se nemůžeme dostat pouze k jedinému prvku, a tím je lokální proměnná. Za běhu programu tedy nemůžeme nijak zjistit ani informace o anotacích lokálních proměnných.

Poznámka:

Nezapomínejte na to, že deklarace všech anotací, které chceme zpracovávat za běhu programu, musejí být označeny anotací `@Retention(RetentionPolicy.RUNTIME)`.

■

10.5.1.1 Získání objektu anotace

Pro získání objektu anotace, resp. instance konkrétní anotace, slouží následující metody z rozhraní `AnnotatedElement`:

Tabulka 10.4: Metody rozhraní `AnnotatedElement`

`getAnnotation(Class<T> annotationType)`	Vrátí objekt anotace pro zadaný typ anotace. Pokud se zadaná anotace u dané položky nevyskytuje, vrátí `null`.
`getAnnotations()`	Vrátí pole typu `Annotation[]` obsahující všechny anotace náležející příslušné deklaraci.
`getDeclaredAnnotations()`	Vrátí pole anotací, které jsou přímo umístěny u dané deklarace.
`isAnnotationPresent(Class<? extends Annotation> annotationType)`	Vrátí `true`, pokud se zadaný typ anotace vyskytuje u dané deklarace. V opačném případě vrátí `false`.

Nejčastěji určitě využijeme metodu `getAnnotation(Class<T> annotationType)`. Většinou se totiž pídíme po konkrétní a předem známé anotaci. Málokdy potřebujeme zjišťovat informace o všech anotacích dané deklarace.

Rozhraní `AnnotatedElement` implementují třídy `Class`, `Constructor`, `Field`, `Method` a `Package`. Pokud tedy chceme získat objekt anotace třídy, stačí nám k tomu class-objekt této třídy. Pokud chceme získat objekty anotací u ostatních členů třídy, máme dispozici metody `getConstructor()`, `getConstructors()`, `getField()`, `getFields()`, `getMethod()`, `getMethods()` a `getPackage()`.

10.5.1.2 Práce s objektem anotace a metoda `toString()`

Pokud budeme k nějakému účelu používat pouze anotace bez parametrů, je celkem zbytečné získávat na ně odkaz. V takovém případě nám stačí metoda `isAnnotationPresent(Class<? extends Annotation> annotationType)` (viz část [10.5.1.1/84]), která pouze testuje přítomnost nějaké anotace. Pokud chceme ale zpracovávat anotaci s parametry, je odkaz na objekt anotace nutný. Jak jsme již zmínily v úvodní části kapitoly, parametry anotace jsou vlastně metody. Prostým zavoláním příslušné metody získáme hodnotu, která odpovídá hodnotě parametru anotace uvedené ve zdrojovém kódu.

Zvláštní použití může mít metoda `toString()`. Ta u objektu anotace nevrací pouze její název, ale i seznam všech parametrů s jejich hodnotami stejně tak, jak je uvedena ve zdrojovém kódu.

10.5.1.3 Získání objektu anotace jiné anotace

V předchozích částech jsme si ukázali několik základních metod, které nám umožňují získat odkaz na objekt anotace, kterou je příslušná deklarace označena. Jak ale zjistit informace o anotacích nějaké konkrétní anotace, na kterou již odkaz máme? Rozhraní `Annotation` obsahuje metodu `annotationType()`, která vrací objekt typu `Class<? extends Annotation>`. Tento class-objekt představuje vlastní deklaraci anotace a můžeme je tedy použít pro zjištění veškerých informací o dané anotaci. Metodou `getName()` můžeme např. zjistit samotný název anotace.

Pro zjištění informací o anotacích této anotace můžeme rovněž použít metody `getAnnotation()`, `getAnnotations()` a `isAnnotationPresent()`.

10.5.2 Zpracování zdrojového kódu

Na základě anotací můžeme automaticky vytvářet např. různé pomocné soubory. Někdy tyto pomocné soubory potřebujeme ještě před překladem zdrojového kódu. Může také jít o soubory, které mohou vývojové nástroje využít pro sestavení programového balíků. Vyvstává tedy požadavek na možnost zpracovávat anotace přímo ze zdrojového kódu programu.

V následující části si ve stručnosti představíme nástroj, který můžeme pro zpracování anotací ze zdrojového kódu použít, a který je součástí standardní distribuce Javy.

10.5.2.1 Nástroj `apt`

Pro zpracování vlastních anotací ze zdrojového kódu můžeme využít nástroj `apt`, který je součástí JDK 5.0. Jde o nástroj ovládaný z příkazové řádky, který nám zprostředkuje zpracování zdrojového kódu.

V první fázi provede `apt` syntaktickou analýzu zdrojového kódu a předá potřebné informace námi naprogramovanému ***anotačnímu procesoru***, který může vytvořit dodatečné soubory. Těmi mohou být i zdrojové kódy, které tvoří součást projektu. Poté může tento nástroj zajistit i překlad celého projektu.

Našemu anotačnímu procesoru přitom poskytuje `apt` **objektový model zdrojového kódu** programu se zachováním typové bezpečnosti. Při vytváření anotačního procesoru se tedy nemusíme starat o způsob čtení struktury zdrojových souborů, to zajistí `apt`. My pouze tomuto nástroji dodáme výkonné jádro.

Podrobný popis je pochopitelně součástí dokumentace JDK 5.0. Ukázkový anotační procesor je také součástí **tutoriálu**.

10.5.2.2 Mirror API

Pro vytvoření vlastního anotačního procesoru, který může být využit nástrojem `apt`, je určena sada rozhraní označovaná jako **Mirror API**. Toto aplikační rozhraní je tvořeno následujícími čtyřmi balíčky:

Tabulka 10.5: Základní struktura Mirror API

`com.sum.mirror.apt`	Obsahuje rozhraní vytvářející komunikační prostředek mezi anotačním procesorem a nástrojem `apt`.
`com.sum.mirror.declaration`	Obsahuje rozhraní pro objektové modelování deklarací (deklarace tříd, rozhraní, členů třídy, metod tříd, atd.). Společným předkem všech typů deklarací je rozhraní `Declaration`.
`com.sum.mirror.type`	Obsahuje rozhraní pro objektové modelování použitých datových typů. Společným předkem všech typů je rozhraní `TypeMirror`.
`com.sum.mirror.util`	Jde o pomocné třídy a rozhraní, které je možné využít pro zpracování deklarací a typů.

Klíčovým rozhraním je rozhraní `AnnotationProcessorFactory`. Implementovat toto rozhraní v našem vlastním anotačním procesoru znamená implementovat tři metody. Dvě z nich slouží nástroji `apt` pro zjištění podporovaných typů anotací, které náš procesor zpracovává (metoda `supportedAnnotationTypes()`), a pro zjištění případných vstupních parametrů anotačního procesoru (metoda `supportedOptions()`). Třetí metodou `getProcessorFor(Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)` si nástroj `apt` vytvoří instanci anotačního procesoru a předá mu tak aplikační rozhraní pro zpracování jednotlivých deklarací. Toto aplikační rozhraní představuje rozhraní `AnnotationProcessorEnvironment`.

Mirror API poskytuje značně rozsáhlé a efektivní možnosti pro vytvoření vlastního anotačního procesoru. Na detailní popis všech tříd a rozhraní budeme muset ale opět sáhnout po dokumentaci JDK 5.0.

Důležité:

Při tvorbě vlastního anotačního procesoru pamatujte na to, že Mirror API je součástí pouze JDK 5.0, nikoliv běhového prostředí JRE 5.0. Pro překlad a spuštění anotačního procesoru je potřeba mít správně nastavenou cestu k dostupným balíčkům (classpath). Mirror API je obsahem souboru *javahome*`/lib/tools.jar`. Pro potřeby programů `java.exe`, `javac.exe` a `apt.exe` je možné nastavit cestu pomocí parametru `-cp`.

■

10.5.3 Možnost zpracování bajtkódu

Existuje i možnost zpracovávat anotace z bajtkódu. To mohou využít např. nástroje pro vytváření instalačních programů. Takový nástroj nemá důvod pracovat se zdrojovým kódem. Anotace, které tento nástroj bude zpracovávat, tedy budou obsaženy v přeloženém bajtkódu, ale nebudou moci být zpracovány za běhu programu.

Existují samozřejmě připravené knihovny pro snazší práci s bajtkódem. Jednou z nich je **Byte Code Engeneering Library** (BCEL) – *http://jakarta.apache.org/bcel*. V JDK 5.0 nalezneme knihovnu BCEL v balíčku `com.sun.org.apache.bcel.internal`. Podrobný popis struktury bajtkódu naleznete ve specifikaci virtuálního stroje Javy – *http://java.sun.com/docs/books/vmspec*.

Práce s bajtkódem je ale poměrně složitá a ve své praxi se s ní setká velmi málo programátorů, natož pak programátorů začátečníků. V ukázkových příkladech a programech se tedy zpracování bajtkódu nebudeme věnovat.

10.6 Další možnosti využití anotací

Obsahem této kapitoly jsme v žádném případě nevyčerpali veškeré možnosti praktického využití anotací. Dalo by se říct, že jsme jen pouze nastínily základní představu o postavení anotací v jazyku Java. Nutno je také poznamenat, že v době psaní této práce nebylo ještě mnoho nástrojů pro využití anotací standardizováno. Navíc spousta nástrojů a balíků anotací se vyvíjí spíše pro využití v **Java 2 Enterprise Edition** (J2EE).

Spousta možností využití anotací se nabízí např. při tvorbě **webových aplikací** a při **ukládání objektů do relační databáze** (využití tzv. objektově-relačního mapování – **ORM**). Anotacemi můžeme např. označit jednotlivé položky třídy a určit tak, jaký datový typ pro ně použít v databázi, případně můžeme omezit hodnoty těchto položek. Jiný přístup může spočívat v tom, že označíme metody sloužící pro získání klíčových vlastností objektu a metody pro jejich opětovné nastavení.

Dalším využitím může být technologie **JavaBeans**. Na základě anotací můžeme nechat např. automaticky generovat k dané třídě i rozhraní (`BeanInfo` interface). Obdobně technologie **Enterprise JavaBeans** vyžaduje tzv. *deployment descriptor*. Anotacemi můžeme tedy určit ty metody, které mají být implementacemi onoho rozhraní.

Jistě nalezneme i možnosti uplatnění anotací v J2SE. Často uváděným příkladem pro pochopení významu anotací bývá náhrada rozhraní `Serializable` a klíčového slova `transient`. Toto rozhraní a klíčové slovo slouží pouze pro informaci virtuálnímu stroji, které objekty je možné serializovat a které jejich položky mají být při serializaci vynechány. Implementace rozhraní `Sezializable` a použití klíčového slova `transient` ale nijak neovlivňuje vlastní běh programu. Bylo by tedy stejně snadné u třídy, kterou chceme serializovat uvést anotaci `@Serializable` a u položek této třídy, které chceme vynechat uvést anotaci `@Transient`. Při absolutním odebrání rozhraní `Serializable` a klíčového slova `transient` z jazyka Java by ovšem byla mírně narušena zpětná kompatibilita mezi staršími verzemi Javy. Proto k tomuto nahrazení zatím nedošlo a asi zřejmě ani nedojde.

Mnohem užitečnějším využití anotací v J2SE je ale např. možnost automaticky generovat **testovací a jiné pomocné třídy** pro snazší ladění programů na základě anotací.

Pro práci s celým projektem bychom také mohli anotace využít pro **automatické generování souboru manifestu** (`META-INF/MANIFEST.MF`), popř. pro generování jiných pomocných či konfiguračních souborů.

Příchod anotací také uvítají znalci technologie **Xdoclet**. Tímto způsobem se v podstatě obcházela absence anotací v předchozích verzích Javy.

V každém případě můžeme i anotace využít pro vytváření vlastní programové **dokumentace** či jakýchkoliv jiných **meta-souborů**.

A Java 6 Mustang

Není to zase až tak dlouho, co se objevila zcela nová verze jazyka Java. Byla to verze **5.0**, která zásadním způsobem rozšířila možnosti tohoto programovacího jazyka a posunula jej tak o veliký kus vzhůru. Již během oficiálního uvolnění plné verze **Javy 5.0** byl také uveřejněn nové připravované verze. Tentokrát jde o verzi **Java 6 Mustang**. V současné době se zveřejňují zprávy, že by první plná verze Javy 6 měla být uvolněna během června 2006.

Nemusíme se však bát, že by celá tato práce vznikala zbytečně. Ba naopak. Java 6 se již nezaměřuje na zdokonalení samotné syntaxe jazyka, ale spíše na funkcionalitu celé Java platformy. Znamená to, že všechny části této práce zůstávají platné beze změny.

V následujících částech si pouze formou přílohy představíme některá vylepšení, která na nás budou již brzy čekat ve standardně dodávaných knihovnách.

Nenechte se odradit tím, že jsou v této příloze prezentovány pouze ty nejjednodušší záležitosti. Je to pouze proto, že většina zásadních změn v Javě 6 se týká možnosti využití **webových aplikací** a s tím spojené technologie XML. My se ale zaměříme spíše na nové třídy, které jsou součástí *Java™ Platform, Standard Edition 6*.

Některé konkrétní změny ve standardních knihovnách

Práce se souborovým systémem

Ve třídě `java.io.File` byla vylepšena podpora pro získávání informací o pevném disku či jiném datovém médiu. Přibyly metody pro zjištění celkového a volného místa na disku.

`getTotalSpace()`	Celkové místo na aktuálním diskovém oddílu.
`getFreeSpace()`	Celkové volné místo na aktuálním diskovém oddílu (tj. počet nealokovaných clusterů).
`getUseableSpace()`	Využitelné místo s přihlédnutím k přístupovým právům uživatele.

Dále přibyly ve třídě `File` metody pro nastavení přístupových práv k souboru, resp. adresáři.

```
public boolean setReadable(boolean readable, boolean ownerOnly)
public boolean setReadable(boolean readable)

public boolean setWriteable(boolean writeable, boolean ownerOnly)
public boolean setWriteable (boolean writeable)

public boolean setExecutable(boolean executable, boolean ownerOnly)
public boolean setExecutable (boolean executable)
```

Už z názvů metod je patrné, že vycházejí ze způsobu přidělování přístupových práv k souborům používaného operačními systémy **Unix**. Uvedené tři typy přístupu lze nastavit těmito metodami zcela stejně jako příkazem `chmod`. Nelze ale využít možnosti nastavovat přístupová práva pro skupinu uživatelů. V případě, že použijeme metodu se dvěma parametry

(viz výše), můžeme určit druhým parametrem, zda se mají nastavená práva týkat pouze vlastníka souboru, resp. adresáře (použijeme hodnotu true), nebo všech uživatelů (použijeme hodnotu false).

Pokud používáme operační systém **Windows**, nemůžeme nastavovat práva pro čtení ani práva pro spouštění. Nastavení práva pro zápis odpovídá nastavení atributu *read-only*.

Kromě metod pro nastavení přístupových práv můžeme také využít metody pro zjištění přístupových práv k souboru, resp. adresáři, který je reprezentován objektem typu `File`. Jde o metody `canRead()`, `canWrite()`, `canExecute()`. Význam těchto metod není jistě potřeba detailněji popisovat. :-)

Zpracování anotací

Významné změny doznala v JDK 6 možnost zpracování anotací ze zdrojového souboru. Můžeme ale i nadále využít **Mirror API** a nástroj `apt`. Pro zpracování anotací, resp. pro vytvoření vlastního anotačního procesoru, můžeme využít třídy z balíčku `javax.annotation.processing`. Každý anotační procesor přitom musí implementovat rozhraní `Processor` z tohoto balíčku. Implementací metody `init()` pak získá anotační procesor objekt typu `ProcessingEnvironment`, který představuje aplikační rozhraní pro vlastní zpracování.

Pro objektové modelování jednotlivých typů deklarací je v JDK 6 připravena sada rozhraní v několika balíčcích. Všechny tyto balíčky začínají `javax.lang.model`.

Java Compiler Tool

Zcela novou vlastností JDK 6 je možnost využívat překladač jazyka Java přímo za běhu programu. Toho je možné využít např. ke skriptování v javovském programu. Pro psaní skriptů můžeme využít přímo jazyk Java se všemi jeho vlastnostmi. Za běhu jakéhokoliv programu pak můžeme tento skript přeložit, načíst do paměti a spustit.

Přístup k tomuto nástroji získáme metodou `defaultJavaCompiler()` ze třídy `ToolProvider`. Tato třída je obsažena v balíčku `javax.tools`. Rozhraní a třídy tohoto balíčku vytvářejí obecné rozhraní pro práci s pomocnými nástroji (*tools*) za běhu programu. Každý takovýto nástroj přitom implementuje rozhraní `Tool` ze stejného balíčku. Zmíněná metoda `defaultJavaCompiler()` vrací objekt typu `JavaCompilerTool`. To je rozhraní, které je zděděno od rozhraní `Tool`.

Možnost skriptování v Java aplikacích

Využití Java Compiler Tool pro jednoduché skriptování by bylo přeci jen nepohodlné. Java proto zavádí od verze JDK 6 tzv. **Java Scripting Engines**. Příslušné API najdete v balíčku `javax.script`.

Budeme tedy moci propojit různé skriptovací jazyky, jako je např. **JavaScript**, **PHP**, **Rhino**, **Jython** a další, s Java platformou. To přináší možnost spouštět skripty z javovské aplikace a poskytnout skriptům přistupovat k Java objektům.

Na podpoře zmíněných skriptovacích jazyků se stále pracuje a předpokládá se, že plně podporován bude v JDK 6 skriptovací jazyk **Rhino**, který je defacto rozšířením jazyka **JavaScript**.

Práce s databázemi

JDK 6 přináší novou verzi rozhraní **JDBC**, konkrétně **4.0**. Mělo by dojít k významnému zjednodušení standardních tříd a k vylepšení správy připojení k databázi. Měla by přitom být zachována zpětná kompatibilita s JDBC 3.0.

Zpracování XML dokumentů

Další novinkou v JDK 6 je nové rozhraní pro zpracování XML dokumentů. Kromě dosavadních SAX a DOM budeme moci využít tzv. **Streaming API**. Jde o metodu zpracování označovanou jako *pull-parsing*, to znamená, že uživatel se sám a na svůj příkaz přesouvá na další uzel, resp. element, v dokumentu. Podobnou technologii můžeme znát i z platformy .NET, zde existuje podpora pro **Streaming API** již od verze 1.0.

Některé novinky v knihovnách pro tvorbu GUI

System tray icon

Pravděpodobně velmi zajímavou novinkou bude možnost vytvářet v našich aplikacích *system tray menu*. A to pochopitelně nezávisle na použité platformě. Tzv. systém tray icons podporuje valná většina dnešních operačních systémů, resp. jejich grafických prostředí.

Pro vytvoření a ovládání tray ikony nám budou stačit dvě třídy z balíčku `java.awt`, a sice `SystemTray` a `TrayIcon`. Objekt typu `TrayIcon` představuje přitom samotnou ikonu, ke které je možné nastavit např. *pop-up menu*. Třída `SystemTray` představuje komunikační rozhraní mezi aplikací a příslušným panelem grafického prostředí operačního systému.

Desktop API

Desktop API zastoupené třídou `java.awt.Desktop` nám umožňuje velmi snadno spouštět z javovské aplikace libovolné soubory na základě nastavených asociací v operačním systému. K tomu můžeme využít následujících metod:

`open(File soubor)`	Spustí soubor v aplikaci, která je nastavena pro běžné spuštění tohoto souboru.
`edit(File soubor)`	Spustí soubor v aplikaci, která je nastavena pro jeho úpravu.
`print(File soubor)`	Spustí soubor v aplikaci, která je nastavena pro tisk.
`browse(URI adresa)`	Zobrazí zadanou adresu ve výchozím prohlížeči.
`mail(URI adresa)`	Otevře okno výchozího poštovního klienta a nastaví požadovanou adresu příjemce.
`mail()`	Otevře prázdné okno výchozího poštovního klienta pro psaní e-mailu.

Použité zdroje

[1] **Mustang Project Home**, *http://mustang.dev.java.net*

[2] **Java Platform, Standard Edition (Java SE) 6 Beta**, *http://java.sun.com/javase/6/*

[3] **JDK™ 6 Documentation**, *http://java.sun.com/javase/6/docs/*

[4] **Technical Articles and Tips** – Feature Stories About Java Technology, *http://java.sun.com/features/*

Literatura

[1] *Sun Microsystems*: **JDK™ 5.0 Documentation**, *http://java.sun.com/j2se/1.5.0/docs/*, 2004

[2] *Sun Microsystems*: **Java™ 2 Platform Standard Edition 5.0 API Specification**, *http://java.sun.com/j2se/1.5.0/docs/api/index.html*, 2004

[3] *Gosling James, Joy Bill, Steele Guy, Bracha Gilad*: **The Java™ Language Specification** – Second Edition, *http://java.sun.com/docs/books/jls/*, Addison-Wesley, USA, 2000

[4] *Sun Microsystems*: **The Java™ tutorial – A practical guide for programmers**, *http://java.sun.com/docs/books/tutorial/index.html*, 2005

[5] *Bracha Gilad*: **Generics in the Java Programming Language** (Generics tutorial), *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*, 2004

[6] *Goetz Brian*: **Introduction to generic types in JDK 5.0**, IBM developerWorks, *http://www-128.ibm.com/developerworks/edu/j-dw-java-generics-i.html*, 2004

[7] *Herout Pavel*: **Učebnice jazyka Java**, KOPP, České Budějovice, 2001

[8] *Herout Pavel*: **Java – Grafické uživatelské prostředí a čeština**, KOPP, České Budějovice, 2001

[9] *Herout Pavel*: **Java – Bohatství knihoven**, KOPP, České Budějovice, 2003

[10] *Pecinovský Rudolf*: **Java 5.0 – Novinky jazyka a upgrade aplikací**, CP Books, a. s., Brno, 2005

[11] *Herout Pavel*: **Učebnice jazyka C – 2. díl**, KOPP, České Budějovice, 2000

[12] *Virius Miroslav*: **Od C k C++**, KOPP, České Budějovice, 2002

[13] *Eller Frank*: **C# – Začínáme programovat**, překlad – Grada Publishing, a. s., Praha, 2002

[14] *Kvoch Martin*: **Programování v Turbo Pascalu 7.0**, KOPP, České Budějovice, 1993

Rejstřík

..., 41
?, 11
@, 77

A

abstraktní třída. *viz* třída
AnnotatedElement, 84
Annotation, 75
anotace, 75
~ další využití, 87
~ deklarace, 77
~ označení balíčků, 76
~ pole, 79
~ standardní, 80
~ typy parametrů, 77
~ zpracování, 83
  ~ z bajtkódu, 86
  ~ za běhu, 69
  ~ za běhu programu, 84
  ~ ze zdrojového kódu, 85
anotační procesor. *viz* procesor
apt, 85
argument
~ vstupní, 60
Array, 27
autoboxing. *viz* zapouzdření
automatické zapouzdření. *viz* zapouzdření

B

bajtkód, 26, 81, 86
BCEL, 86
bezpečnost
~ typová, 3

C

Class, 69, 84
class-objekt, 58, 70, 84, 85
cyklus
~ for-each. *viz* for-each

Č

čtení vstupního řetězce, 71

D

dědičnost, 23
Deprecated, 83
Documented, 82

E

ElementType, 81
entita, 63
~ obecný formát, 63
enum, 45, 46
Enum, 70
EnumMap, 58
EnumSet, 58
extends, 15

F

for-each, 36
~ průchod kolekce, 37
~ průchod mapy, 39
~ průchod pole, 36
formát, 60
~ syntaxe zápisu, 63
format(), 60
formátovací procesor. *viz* procesor
formátování, 60
~ chybová hlášení, 68
~ typy konverze, 63
~ značky, 67
Formatter, 60, 61
fronta, 56
~ prioritní, 57

G

generická metoda. *viz* metoda
generické rozhraní. *viz* rozhraní
generický typ. *viz* typ

I

implementace rozhraní, 25
import
~ statický, 29

~ statický, konflikt, 31
Inherited, 82
interface, 77
Iterable, 39
iterátor, 54

J

J2EE, 87
JavaBeans, 87

K

kolekce, 53
~ novinky, 56
kompatibilita, 6
kompilace, 26, 44
konstanty
~ výčtové, 44
konverze, 63

L

Locale, 60

M

manifest, 87
Math, 70
metaanotace, 81
metadata. *viz* anotace
metoda
~ formátování, 60
~ generická, 12
~ generická, deklarace, 12
~ generická, použití, 13
~ pracující s kolekcemi, 55
~ s proměnným počtem parametrů, 41
metody
~ anotace, 75
Mirror API, 86

N

nadtřída, 17, 55

O

objekt
~ generické třídy, 8
~ uložení do databáze, 87
ohraničení, 14
~ vícenásobné, 19
ohraničení shora, 15
ohraničení zdola, 17
omezení. *viz* ohraničení
operátory
~ porovnání, 32
Override, 82
označení balíčků, 76

P

parametr
~ anotace, 75
  ~ typy parametrů, 77
~ proměnný počet, 41
~ typový, 3, 10
~ typový, obecné označení, 6
~ typový, ohraničení, 14
parametrizovaný typ. *viz* typ
podmínka, 47
pole, 21, 36, 79
~ inicializace typovým parametrem, 22, 27
~ vícerozměrné, 37
polymorfizmus, 10
porovnání, 32
printf(), 60
prioritní fronta. *viz* fronta
PriorityQueue, 57
procesor
~ anotační, 75, 85
~ formátovací, 61
  ~ bez výstupu, 61
  ~ s určeným výstupem, 62
průchod kolekce. *viz* for-each
průchod mapy. *viz* for-each
průchod pole. *viz* for-each
překlad. *viz* kompilace
převod
~ automatický. *viz* zapouzdření

Q

Queue, 57

R

reflexe, 69, 84
regulární výraz. *viz* výraz
Retention, 81
RetentionPolicy, 81
rozhraní, 75
~ generické, 9
~ generické, implementace, 25

S

Scanner, 71
sdílení třídy, 6, 27

soubory
~ pomocné, 87
static import, 29
statický import. *viz* import
StrictMath, 70
StringBuffer, 71
StringBuilder, 71
super, 17
SuppressWarnings, 83
switch, 48
symbol
~ zástupný, 11

Š

šablony, 6

T

Target, 81
třída
~ abstraktní, 9
~ generická, 3
~ testovací, 87
třídy
~ pomocné, 87
typ
~ generický, 1, 53
  ~ abstraktní třída, 9
  ~ dědičnost, 23
  ~ deklarace, 7
  ~ deklarace rozhraní, 9
  ~ kompilace, 26
  ~ nepovolené operace, 27
  ~ pojem, 3
  ~ pole, 21
  ~ složitější zápisy, 20
  ~ výhody, 3
  ~ vytvoření, 7
~ parametrizovaný. *viz* typ generický
~ primitivní/objektový, 32
~ výčtový, 44, 57, 70
  ~ konstanty, 44
  ~ náhrada, 44
  ~ podmínka, 47
  ~ průchod konstantami, 48
  ~ složitější definice, **49**
  ~ switch, 48
typová bezpečnost. *viz* bezpečnost
typový parametr. *viz* parametr

V

value, 79
vícenásobné ohraničení. *viz* ohraničení
vstup
~ čtení řetězce, 71
vstupní argument. *viz* argument
výčtový typ. *viz* typ
výraz
~ regulární, 72, 73
výstup
~ formátování. *viz* formátování

X

Xdoclet, 87

Z

zapouzdření
~ anutomatické, 32
~ automatické, 54
zástupný symbol. *viz* symbol
značky, 75
zobecnění
~ metody, 12
~ rozhraní, 9
~ třídy, 3, 7
zparcování anotací. *viz* anotace