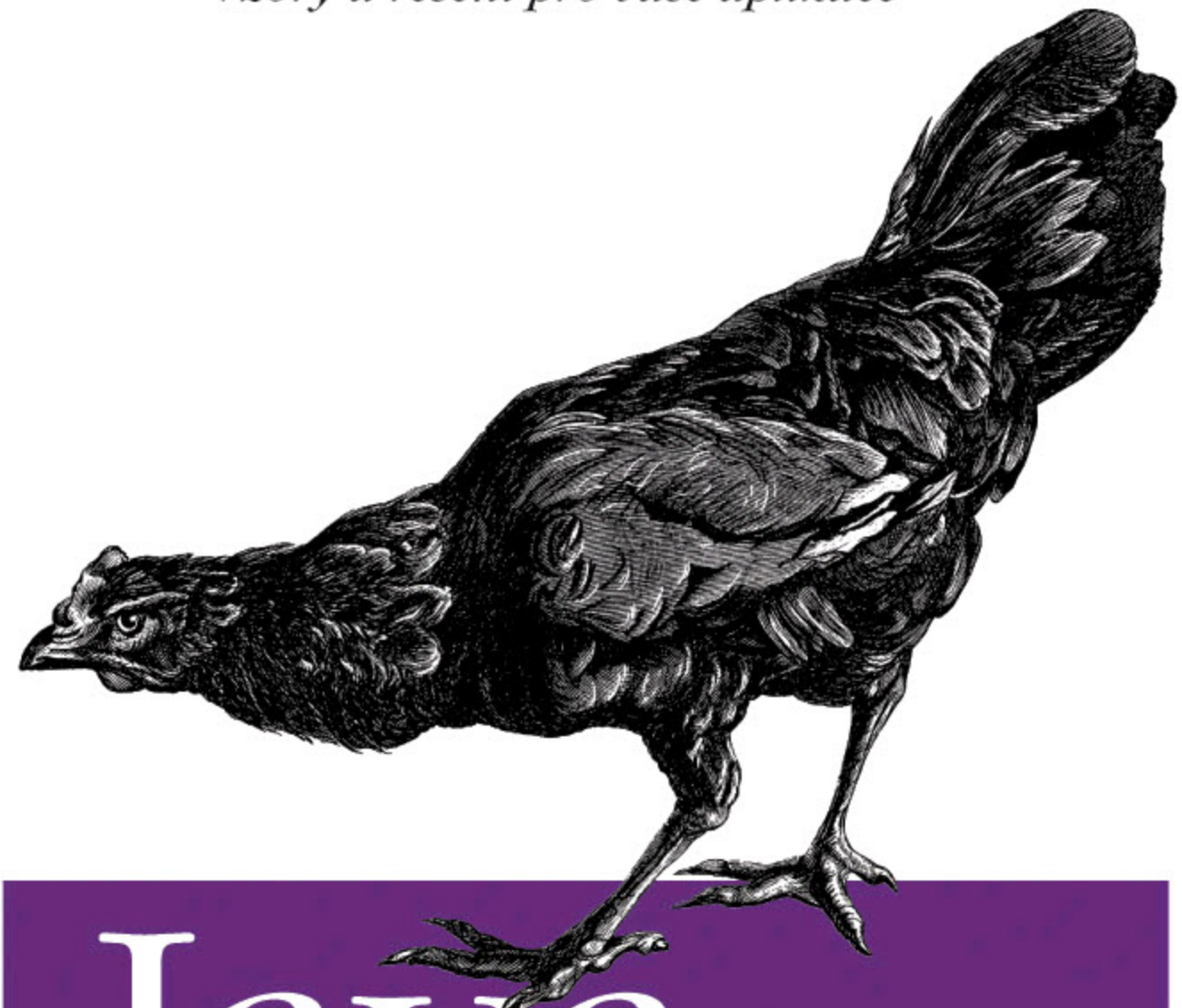
Vzory a řešení pro vaše aplikace



Java™ kuchařka programátora

Ian F. Darwin

O'REILLY®

computer press

Ian F. Darwin

Java
Kuchařka programátora

Computer Press, a.s.
Brno
2006

Java Kuchařka programátora

Ian F. Darwin

Vydalo nakladatelství Computer Press, a.s. jako svou 2151. publikaci.

Vydavatelství a nakladatelství Computer Press, a.s.,
nám. 28. dubna 48, 635 00 Brno, knihy.cpress.cz

ISBN 80-251-0944-5

Prodejní kód: K1223

Překlad: Jan Gregor
Odborná korektura: Rudolf Pecinovský
Jazyková korektura: Marie Schreinerová
Vnitřní úprava: O´Reilly, Jiří Matoušek
Sazba: Jiří Matoušek, Petr Klíma
Rejstřík: Daniel Štreit
Obálka: O´Reilly

Komentář na zadní straně obálky:
Martin Domes, Ivo Magera
Technická spolupráce: Petr Klíma
Odpovědný redaktor: Martin Domes
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a.s., nám. 28. dubna 48, 635 00 Brno
tel.: 546 122 111, fax: 546 122 112
Objednávejte na: **knihy.cpress.cz**
distribuce@cpress.cz

Bezplatná telefonní linka: **800 555 513**

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

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

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

Obsah

Předmluva 13

Předmluva k druhému vydání 13
Předmluva k prvnímu vydání 14
Pro koho je tato kniha určena 15
Co v této knize najdete? 16
Poznámky k platformě 18
Jiné knihy 19
Konvence používané v knize 21
Komentáře a otázky 22
Stažení zdrojových kódů 23
Poděkování 24

1. Začínáme: překlad, spuštění a ladění 27

1.1 Překlad a spuštění programů v Javě: JDK 27
1.2 Upravování a překlad pomocí editoru s podporou barevného zvýrazňování syntaxe 29
1.3 Překlad, spouštění a testování pomocí IDE 30
1.4 Efektivní používání CLASSPATH 36
1.5 Použití tříd API cz.darwinsys z této knihy 38
1.6 Překlad zdrojového kódu příkladů z této knihy 39
1.7 Automatický překlad pomocí nástroje Ant 40
1.8 Spouštění apletů 43
1.9 Zpracování varování o zavržení 44
1.10 Podmíněné ladění bez #ifdef 46
1.11 Ladění výstupů 47
1.12 Udržování korektnosti programu pomocí asercí 49
1.13 Ladění pomocí JDB 50
1.14 Jednotkové testy: vyhněte se potřebě ladících nástrojů 53
1.15 Získejte čitelné zpětné sledování 55
1.16 Hledání dalších zdrojových kódů v Javě 56
1.17 Program: Debug 58

2. Komunikace s prostředím 59

2.1 Získání proměnných prostředí 59
2.2 Systémové vlastnosti 61
2.3 Psaní kódu závislého na verzi JDK 63
2.4 Psaní kódu závislého na operačním systému 64
2.5 Použití rozšíření nebo dalších zapouzdřených API 66
2.6 Syntaktická analýza argumentů příkazového řádku 67

3. Řetězce 75

3.1 Rozkládání řetězců do podřetězců 77
3.2 Rozkládání řetězců pomocí StringTokenizer 78
3.3 Spojování řetězců pomocí operátoru +, StringBuilder (JDK 1.5) a StringBuffer 81
3.4 Zpracování řetězce po jednom znaku 83
3.5 Zarovnání řetězců 84
3.6 Převody mezi znaky Unicode a řetězci 86
3.7 Převracení řetězce po slovech nebo znacích 88
3.8 Rozšiřování a komprese tabulátorů 89
3.9 Ovládání velikosti písmen 94
3.10 Odsazení textových dokumentů 95
3.11 Vkládání netisknutelných znaků 96
3.12 Odstraňování mezer z konce řetězce 97
3.13 Syntaktická analýza dat oddělených čárkou 99
3.14 Program: Jednoduchý formátovač textu 104
3.15 Program: Fonetický porovnávač názvů (soundex) 106

4. Porovnávání vzorů pomocí regulárních výrazů 109

4.1 Syntaxe regulárních výrazů 111
4.2 Použití regulární výrazů v Javě: Test na vzor 117
4.3 Hledání odpovídajícího textu 121
4.4 Nahrazení odpovídajícího textu 123
4.5 Tisk všech výskytů vzoru 124
4.6 Vypisování řádků obsahující vzory 126
4.7 Ovládání velikosti písmen v regulárních výrazech 127
4.8 Porovnávání „akcentovaných“ nebo složených znaků 129
4.9 Porovnávání nových řádků v textu 130
4.10 Program: Syntaktická analýza protokolového souboru serveru Apache 132
4.11 Program: Dolování dat 133
4.12 Program: Úplný Grep 135

5. Čísla 141

5.1 Kontrola, jestli je řetězec platné číslo 143
5.2 Ukládání větších čísel do menších čísel 144
5.3 Převádění čísel na objekty a naopak 145
5.4 Vytvoření zlomku z celého čísla bez použití čísla s plovoucí desetinnou čárkou 146
5.5 Zajištění přenosnosti čísel s plovoucí desetinnou čárkou 147
5.6 Porovnávání čísel s plovoucí desetinnou čárkou 149
5.7 Zaokrouhlování čísel s plovoucí desetinnou čárkou 151
5.8 Formátování čísel 151
5.9 Převod mezi dvojkovými, osmičkovými, desítkovými a šestnáctkovými čísly 154
5.10 Zpracování skupiny celých čísel 155
5.11 Práce s římskými číslicemi 156
5.12 Správné formátování množného čísla 159
5.13 Generování náhodných čísel 161
5.14 Generování lepších náhodných čísel 162
5.15 Počítání trigonometrických funkcí 164
5.16 Logaritmování 164
5.17 Násobení matic 165
5.18 Používání komplexních čísel 166
5.19 Zpracování velmi velkých čísel 169
5.20 Program: Převodník teplot 171
5.21 Program: Číselné palindromy 174

6. Data a časy 177

6.1 Nalezení dnešního datumu 178
6.2 Vypsání datumu/času v daném formátu 179
6.3 Znázornění dat v jiných epochách 182
6.4 Převádění YMDHMS na kalendář nebo sekundy letopočtu 182
6.5 Převod řetězců na data 184
6.6 Převádění sekund epochy na DMYHMS 185
6.7 Sčítání nebo odčítání množství od datumu nebo kalendáře 186
6.8 Rozdíly mezi dvěma daty 187
6.9 Porovnávání dat 188
6.10 Den týdne/měsíce/roku nebo číslo týdne 189
6.11 Vytváření stránky kalendáře 190
6.12 Měření uplynulého času 192
6.13 Dopřejte si chvilku odpočinku 194
6.14 Program: Upomínací služba 195

7. Strukturování dat pomocí Javy 199

7.1 Použití polí pro strukturování dat 200
7.2 Změna velikosti pole 201
7.3 Dynamičtější pole 202
7.4 Použití iterátorů pro datově nezávislý přístup 204
7.5 Strukturování dat v spojovém seznamu prvků 205
7.6 Mapování pomocí Hashtable a HashMap 207
7.7 Ukládání řetězců do vlastností a preferencí 208
7.8 Seřazení kolekce 212
7.9 Vyhýbejte se nutnosti uspořádávat data 216
7.10 Vyvarujte se duplicitních hodnot 217
7.11 Hledání objektu v kolekci 218
7.12 Převádění kolekce na pole 220
7.13 Definice vlastního iterátoru 220
7.14 Zásobník 222
7.15 Vícerozměrné struktury 223
7.16 Konečně kolekce 225
7.17 Program: Porovnání časové náročnosti 227

8. Novinky ve verzi 5.0 229

8.1 Použití parametrizovaných kolekcí 230
8.2 Použití rozšíření cyklů for 231
8.3 Vyhnutí se přetypování pomocí parametrizace typů 232
8.4 Dovolte Javě automatické převody mezi primitivními a jejich obalovými typy 234
8.5 Výčtové typy 235
8.6 Program: MediaInvoicer 239

9. Objektově orientované techniky 243

9.1 Tisk objektů: Formátování pomocí toString() 245
9.2 Překrytí metody equals 246
9.3 Překrytí metody hashCode 249
9.4 Metoda clone 250
9.5 Metoda finalize 252
9.6 Použití interních, tj. vnořených, vnitřních a lokálních tříd* 254
9.7 Poskytování zpětných volání přes rozhraní 256
9.8 Polymorfizmus/abstraktní metody 259
9.9 Předávání hodnot 260
9.10 Aplikace návrhového vzoru Jedináček (Signleton) 263
9.11 Vytváření vlastních výjimek 265
9.12 Program: Plotter 265

10. Vstup a výstup 269

10.1 Načítání ze standardního vstupu 270
10.2 Zapisování do standardního výstupu 274
10.3 Vypisování pomocí třídy Formatter z JDK 1.5 275
10.4 Prohledávání souboru pomocí StreamTokenizeru 279
10.5 Prohledávání vstupu pomocí třídy JDK 1.5 Scanner 283
10.6 Otevírání souboru podle názvu 287
10.7 Kopírování souboru 288
10.8 Načítání souboru do řetězce 291
10.9 Přesměrování standardních proudů 291
10.10 Duplikování proudu tak, jak se zapíše 292
10.11 Čtení/zapisování různých znakových sad 294
10.12 Otravné znaky konce řádku 295
10.13 Dejte si pozor na kód závislý na platformě 295
10.14 Čtení „pokračujících" řádků 296
10.15 Binární data 301
10.16 Hledání 302
10.17 Zapisování datových proudů z C 303
10.18 Ukládání a obnovování objektů Javy 305
10.19 Předcházení výjimkám ClassCastExceptions pomocí SerialVersionUID 308
10.20 Čtení a zapisování archívů JAR nebo Zip 309
10.21 Čtení a zapisování komprimovaných souborů 312
10.22 Program: Převod textu na PostScript 313

11. Operace nad adresáři a systémem souborů 317

11.1 Získání informací o souborech 317
11.2 Vytvoření souboru 320
11.3 Přejmenování souboru 321
11.4 Mazání souboru 322
11.5 Vytvoření dočasného souboru 323
11.6 Změna atributů souboru 325
11.7 Procházení adresářové struktury 326
11.8 Získání kořenových adresářů 328
11.9 Vytváření nových adresářů 329
11.10 Program: Find 330

12. Programování externích zařízení: sériové a paralelní porty 333

12.1 Výběr portu 335
12.2 Otevření sériového portu 338
12.3 Otevírání paralelního portu 342
12.4 Řešení konfliktů mezi porty 345

12.5 Čtení a zápis: lock- step 348
12.6 Čtení a zápis: řízené událostmi 350
12.7 Čtení a zápis: vlákna 354
12.8 Program: plotter Penman 356

13. Grafika a zvuk 361

13.1 Zobrazení grafiky pomocí objektu třídy Graphics 361
13.2 Testování grafických komponent 363
13.3 Zobrazení textu 364
13.4 Zobrazení textu zarovnaného na střed 364
13.5 Zobrazení stínu 365
13.6 Zobrazení textu pomocí tříd z balíčku 2DGraphics 368
13.7 Zobrazení textu s integrovaným typem písma 370
13.8 Zobrazení obrázku 373
13.9 Přehrávání zvukových souborů 377
13.10 Přehrávání videoklipů 378
13.11 Tisk v Javě 381
13.12 Program: PlotterAWT 384
13.13 Program: Grapher 387

14. Grafické uživatelské rozhraní 391

14.1 Zobrazení komponent GUI 392
14.2 Návrh rozvržení okna 394
14.3 Zobrazení panelu s kartami 397
14.4 Správa událostí: tlačítka 398
14.5 Ošetřování událostí pomocí anonymních vnitřních tříd 400
14.6 Ukončení programu „zavřením okna“ 402
14.7 Dialogy: když „později“ je příliš pozdě 406
14.8 Zachycení a formátování výjimek grafického uživatelského rozhraní 408
14.9 Zobrazení programového výstupu v okně 410
14.10 Výběr hodnoty pomocí komponenty JSpinner 413
14.11 Výběr souboru pomocí komponenty JFileChooser 414
14.12 Výběr barvy 417
14.13 Formátování komponent JComponents pomocí HTML 419
14.14 Vystředění hlavního okna 420
14.15 Změna vzhledu a chování programu používajícího pro vytvoření GUI třídy knihovny Swing 421
14.16 Rozšíření grafického uživatelského rozhraní pro Mac OS X 425
14.17 Program: upravený dialog pro výběr písma 427
14.18 Program: upravený správce rozvržení 431

15. Podpora cizích jazyků a lokalizace 439

15.1 Tvorba tlačítka pomocí prostředků I18N 439
15.2 Výpis dostupných národních prostředí 441
15.3 Vytváření nabídky pomocí prostředků I18N 442
15.4 Vytváření konvenčních rutin pro internacionalizaci 443
15.5 Vytváření dialogového okna pomocí prostředků I18N 445
15.6 Vytváření svazku prostředků (resource bundle) 446
15.7 Vytahování řetězců ze zdrojového kódu 447
15.8 Nastavení odlišného národního prostředí 448
15.9 Nastavení výchozího národního prostředí 449
15.10 Formátování textu zprávy 450
15.11 Program: MenuIntl 452
15.12 Program: BusCard 454

16. Síťoví klienti 459

16.1 Navázání spojení se serverem 461
16.2 Hledání a oznamování síťových adres 462
16.3 Ošetřování síťových chyb 463
16.4 Čtení a zápis textových dat 464
16.5 Čtení a zápis binárních dat 466
16.6 Čtení a zápis serializovaných dat 468
16.7 Datagramy protokolu UDP 470
16.8 Program: Klient TFTP implementovaný pomocí protokolu UDP 472
16.9 Program: klient programu Telnet 476
16.10 Program: Klient pro chat 479

17. Serverová Java: sokety 485

17.1 Uvedení serveru do provozu 485
17.2 Odeslání odpovědi (textové i binární) 488
17.3 Odeslání informací pomocí objektů 491
17.4 Zpracování více klientů 492
17.5 Obsluha protokolu HTTP 497
17.6 Zabezpečení webového serveru pomocí SSL a JSSE 499
17.7 Protokolování síťového provozu 501
17.8 Protokolování síťě pomocí log4j 506
17.9 Protokolování v síti pomocí JDK 1.4 508
17.10 Hledání síťových rozhraní 510
17.11 Program: Chatový server v Javě 511

18. Síťoví klienti II: Aplety a weboví klienti 517

18.1 Vložení Javy do webové stránky 517
18.2 Techniky vytváření apletů 519
18.3 Kontaktování serveru na hostitelském počítači apletu 521
18.4 Zobrazení dokumentu pomocí apletu 524
18.5 Spuštění JavaScriptu pomocí apletu 526
18.6 Spuštění skriptu CGI pomocí apletu 527
18.7 Čtení obsahu z dané adresy URL 528
18.8 URI, URL nebo URN? 529
18.9 Vytahování značek HTML z URL 530
18.10 Vytahování adres URL ze souboru 533
18.11 Převod názvu souboru na adresu URL 534
18.12 Program: MkIndex 535
18.13 Program: LinkChecker 540

19. Java a elektronická pošta 547

19.1 Zasílání e-mailu: Verze pro prohlížeč 547
19.2 Opravdové zasílání e-mailů 552
19.3 Serverový program s podporou pro e–maily 554
19.4 Zasílání e-mailů v kódování MIME 558
19.5 Poskytování nastavení e-mailu 560
19.6 Zasílání e-mailů bez použití JavaMail 561
19.7 Čtení e-mailů 566
19.8 Program: MailReaderBean 570
19.9 Program: MailClient 573

20. Databázový přístup 585

20.1 Jednoduchý databázový přístup pomocí JDO 586
20.2 Textové databáze 589
20.3 Databáze DBM 593
20.4. Nastavení JDBC a připojení 596
20.5 Připojení k databázi JDBC 598
20.6 Zasílání dotazu JDBC a získávání výsledků 601
20.7 Použití připravených příkazů JDBC 604
20.8 Použití uložených procedur pomocí JDBC 608
20.9 Modifikace dat pomocí ResultSet 608
20.10 Uchovávání výsledků v RowSet 609
20.11 Modifikace dat pomocí SQL 611
20.12 Hledání metadat JDBC 613
20.13 Program: SQLRunner 617

21. XML 629

21.1 Generování XML z objektů 631
21.2 Transformování XML pomocí XSLT 632
21.3 Syntaktická analýza XML pomocí SAX 635
21.4 Syntaktická analýza XML pomocí DOM 637
21.5 Ověřování struktury pomocí DTD 641
21.6 Generování vlastního XML pomocí DOM 642
21.7 Program: xml2mif 644

22. Distribuovaná Java: RMI 647

22.1 Definování podmínek RMI 648
22.2 Vytváření klienta RMI 650
22.3 Vytváření serveru RMI 651
22.4 Zavádění RMI přes síť 653
22.5 Program: Zpětné volání RMI 654
22.6 Program: NetWatch 657

23. Balíčky a balení 665

23.1 Vytváření balíčku 666
23.2 Dokumentace tříd pomocí Javadoc 667
23.3 Za hranice JavaDoc: Anotace/Metadata (JDK 1.5) a XDoclet 671
23.4 Archivace do souborů jar 673
23.5 Spouštění apletu z archívu JAR 674
23.6 Spuštění apletu pomocí moderního JRE 675
23.7 Spouštění aplikace z archívu JAR 678
23.8 Příprava třídy jako JavaBean 680
23.9 Vkládání komponenty JavaBean do archívu JAR 684
23.10 Balení serveletu do souboru WAR 685
23.11 „Jedenkrát vytvořte a instalujte kdekoli“ 685
23.12 „Jednou vytvořte a instalujte na systémy Mac OS X“ 686
23.13 Java Web Start 688
23.14 Podepisování vašeho souboru JAR 693

24. Práce s vlákny 695

24.1 Spouštění kódu v odlišném vláknu 696
24.2 Zobrazení pohyblivého obrázku pomocí animace 700
24.3 Zastavení vlákna 704
24.4 Rendezvous vláken a časové limity 706
24.5 Synchronizace vláken pomocí klíčového slova synchronized 707
24.6 Zjednodušování synchronizace pomocí zámků JDK 1.5 712

24.7 Synchronizace vláken pomocí wait(?) a notifyAll(?) 716
24.8 Zjednodušení modelu producent-konzument pomocí rozhraní Queue sady JDK 1.5 721
24.9 Automatické ukládání v editoru 724
24.10 Program: Síťový server s vlákny 725
24.11 Zjednodušení serverů pomocí Concurrency Utilities (JDK 1.5) 732

25. Reflexe 735

25.1 Deskriptor tříd 736
25.2 Hledání a používání metod a proměnných 737
25.3 Dynamické zavádění tříd a vytváření instancí 740
25.4 Vytváření třídy úplně od začátku 742
25.5 Měření výkonu 743
25.6 Vypisování informací o třídě 746
25.7 Program: CrossRef 748
25.8 Program: AppletViewer 753

26. Použití Javy s jinými programovacími jazyky 761

26.0 Úvod 761
26.1 Spuštění programu 761
26.2 Spuštění programu a zachytávání jeho výstupu 765
26.3 Propojení Javy se skripty pomocí rozhraní BSF 768
26.4 Kombinace jazyků Java a Perl 772
26.5 Použití nativního kódu (jazyky C/C++) 776
26.6 Volání Javy z nativního kódu 781
26.7 Program: DBM 782

27. Doslov 785

Rejstřík 787

Předmluva

Předmluva k druhému vydání

Vzrušující změnu na území Javy přináší vývojářská sad a JDK 1.5 s krycím názvem Tiger. Tato verze uvádí několik nových hlavních schopností, například generické typy pro lepší strukturování dat, metadata pro opatřování tříd Java™ komentáři flexibilním, ale přesně stanoveným způsobem, nové mechanismy pro čtení dat založené na vzorech a nový mechanismus pro formátovaný tisk. JDK 1.5, která je nezbytností pro vývojáře Javy, tvoří kromě těchto hlavních změn i mnohem větší množství menších avšak důležitých změn. Nějaký čas ještě uplyne, než budou tyto mechanismy plně pochopeny a dostanou se do širšího oběhu, nicméně poté o nich budete chtít okamžitě vědět.

V doslovu prvního vydání jsem uvedl, že „psaní této knihy bylo ponižující zkušeností". Měl bych tedy doplnit, že také její údržba byla ponižující. I když celá řada recenzentů a autorů nešetřila chválou – jeden velmi laskavý recenzent ji dokonce označil za „prokazatelně nejlepší knihu, která kdy byla o programovacím jazyce Java napsána" – byl jsem zahanben počtem chyb a opomenutí v prvním vydání. Při přípravě druhého vydání jsem se snažil všechny nedostatky napravit.

Současně jsem doplnil celou řadu nových návodů a menší počet starých návodů jsem odstranil. Největší samostatný doplněk představuje kapitola 8, v níž se zabýváme generickými typy a výčty, funkcionalitami, které poskytují zvýšenou flexibilitu pro kontejnery, jako například Java Collections. Nyní, když Java zahrnuje aplikační programové rozhraní pro regulární výrazy, se kapitola 4 změnila z API pro regulární výrazy Apache na Regulární výrazy JDK 1.4.

Poněkud váhavě jsem odstranil kapitolu Síťový web, která obsahovala program JabaDot Web Portal Site. Jednalo se o nejdelší samostatný příklad programu v knize, který vyžadoval značné předělání (ve skutečnosti bylo třeba tento příklad kompletně přepsat). Kdybychom psali takové webové stránky nyní, mnohem více bychom využili značky JSP a téměř určitě bychom pracovali s nějakým webovým frameworkem, jako je například Struts (*http://jakarta.apache.org/struts*), SOFIA *(http://www.salmonllc.com/)* nebo Spring Framework *(http://www.springframework.org/)*, abychom se vyhnuli značnému množství zdlouhavého programování. Případně bychom mohli využít nějaký stávající balík – například JLCP od Java Lobby. Informace o Servletech a JavaServer Pages lze najít v knize vydavatelství O'Reilly *Java Servlet & JSP Cookbook* od Bruce W. Perryho. Pojednání o samotném Struts obsahuje kniha Chucka Cavanesse *Programming Jakarta Struts* (O'Reily). Webové služby založené na SOAP objasňuje kniha *Java Web Services* od Dave Chappella a Tylera Jewella (O'Reily), takže tímto tématem se zde nebudeme zabývat.

I když jsme příklady testovali na různých systémech a pro znovusestavení všech kódů poskytujeme skripty Ant, převážnou část nového vývoje a textů k tomuto vydání jsem dělal na systému Mac OS X, který je skutečně „Unixem pro širokou veřejnost“ a který poskytuje jednu z nejlépe podporovaných platforem pro práci s Javou. Mac OS X Java však trochu trpí syndromem „zpožďování nových verzí“, protože v době, kdy šlo druhé vydání knihy do tisku, verze 1.5 nebyla pro Macintosh dostupná. Materiál JDK 1.5 byl vyvíjen a testován na Linuxu a Windows.

Rád bych vyjádřil své upřímné poděkování všem, kteří po vytištění prvního anglického vydání * zaslali komentáře nebo kritiku knihy. Zejména musím uvést jednoho z německých překladatelů knihy, Gisberta Selkeho, který během svého překladu pročetl celé první vydání knihy a zjednodušil mou angličtinu. Totéž udělal znovu i pro druhé vydání a rovněž přetvořil celou řadu programových kódů, čímž udělal tuto knihu mnohem lepší, než by jinak byla. Gisbert dokonce zašel za hranice svých povinností a přispěl do knihy jedním návodem (návod 26.4). Dále přepracoval některé jiné návody v téže kapitole. Děkujeme, Gisberte! Druhé vydání také obohatily komentáře Jima Burgesse, který přečetl značnou část knihy. K jednotlivým kapitolám jsme obdrželi komentáře od Jonathana Fuertha, Kima Fowlera, Marca Loya a Mika McCloskeye. Některé kapitoly byly také zkorigovány mou ženou Betty a našimi dospívajícími dětmi.

Odhalením závažných chyb nebo navržením zlepšení oproti prvnímu vydání přispěli do knihy tito lidé: Rex Bosma, Rod Buchanan, John Chamberlain, Keith Goldman, Gilles-Philippe Gregoire, B. S. Hughes, Jeff Johnston, Rob Konigsberg, Tom Murtagh, Jonathan O’Connor, Mark Petrovic, Steve Reisman, Bruce X. Smith a Patrick Wohlwend. Všem patří moje díky a omlouvám se, pokud jsem na někoho zapomněl.

Mé díky rovněž patří lidem v diskusním fóru nakladatelství O’Reilly „Otázky týkající se knih“ za poskytování nenacvičených odpovědí na tolik otázek. Za redakční a produkční práci na druhém vydání děkuji Mikovi Loukidesovi, Deb Cameron a Marlowe Shaeffer.

Předmluva k prvnímu vydání

Znáte-li trochu Javu, je to skvělé. Znáte-li Javu trochu více, je to ještě lepší! Tato kniha je ideální pro čtenáře, kteří mají základní znalosti o jazyce Java a chtějí se něco nového naučit. Nevíte-li zatím o Javě nic, měli byste si nejprve přečíst některou z knih, které se více zaměřují na základy tohoto programovacího jazyka. Nakladatelství O’Reilly například vydalo *Head First Java* or *Learning Java*, pokud se poprvé setkáváte s touto rodinou jazyků nebo *Java in a Nutshell*, pokud patříte ke zkušeným programátorům v jazyce C.

V C jsem začal programovat v roce 1980, když jsem pracoval na Torontské univerzitě, a v průběhu osmdesátých a devadesátých let dvacátého století mi jazyk C docela dobře sloužil. V roce 1995, když se rodící jazyk Oak přejmenoval na Javu, se na mě usmálo štěstí a můj kolega, J. Greg Davidson, mi o novém jazyce pověděl. Poslal jsem e-mail na adresu, kterou mi Greg doporučil, a v březnu roku 1995 jsem obdržel od Jamese Goslinga, tvůrce Javy, následující odpověď:

```
> Zdravím. Přítel mi řekl o Vašem rozšiřitelném síťovém prohlížeči
> WebRunner(?). WebRunner a Oak(?), jeho jazykové rozšíření, nezní to skvěle? Můžete
```

* První vydání je v současnosti k dispozici v angličtině, němčině, francouzštině, polštině, ruštině, korejštině, tradiční čínštině a zjednodušené čínštině.

```
> mi prosím sdělit, jestli je již dostupný k vyzkoušení a/nebo jestli
> je k němu na FTP dostupné nějaké odborné pojednání?

Vyzkoušejte http://java.sun.com
(Oak se přejmenoval na Java a WebRunner se přejmenoval na
 HotJava, aby byli právníci spokojeni)
```

Stáhl jsem si HotJava a začal jsem ji zkoušet. Zpočátku jsem byl trochu skeptický k tomuto modernímu jazyku, který vypadal jako zkomolenina jazyků C/C++. Napsal jsem testovací a ukázkové programy, několik jsem jich najednou vložil do adresáře, který jsem označil jako *javasrc*, abych uchovával tyto soubory odděleně od mých zdrojových kódů v C (protože programy měly mnohdy stejný název). Jakmile jsem však trochu pronikl do tajů jazyka Java, začal jsem objevovat její výhody pro celou řadu různých činností, například pro automatické obnovení paměti a eliminaci kalkulací ukazatele. Můj adresář *javasrc* se stále zvětšoval, a když jsem napsal kurzy Javy pro společnost Learning Tree*, dostal se do bodu, kdy bylo potřeba vytvořit podadresáře. I poté bylo stále obtížnější něco v adresáři najít, takže brzy bylo zřejmé, že budu potřebovat nějaký typ dokumentace.

Tato kniha je v určitém smyslu výsledkem rychlé kolize mezi mým adresářem *javasrc* a dokumentačním rámcem zřízeným pro další nově vznikající jazyk. Tom Christiansen a Nathan Torkington vytvořili v knize nakladatelství O'Reilly *Perl Cookbook* velmi úspěšnou strukturu, v níž se předkládá materiál v krátkých, účelově zaměřených článcích označovaných jako „návody" neboli jakési „recepty". Výchozím modelem této knihy je pochopitelně všem důvěrně známá kuchařka. Použití pojmu „kuchařka" pro označení výčtu návodů, „jak něco udělat" na počítači, není žádnou žhavou novinkou. Analogii „kuchařky" aplikoval na straně softwaru Donald Knuth ve své knize *The Art of Computer Programming* (Addison Wesley), poprvé vydané v roce 1968. Na straně hardwaru napsal Don Lancaster knihu *The TTL Cookbook* (Sams, 1974). (TTL neboli tranzistorově tranzistorová logika představovala v té době malé stavební bloky elektronických obvodů.) Tom a Nathan vypracovali úspěšnou variaci tohoto konceptu a jejich knihu doporučuji každému, kdo se chce dozvědět více o jazyce Perl. No a práce, kterou právě čtete, se snaží být knihou pro lidi, kteří se chtějí dozvědět více o jazyce Java.

Kód v každém návodu by měl být do značné míry samostatný; jakékoli části příkladů si můžete bez obav vypůjčit pro využití ve vlastních projektech. Kód je distribuován s berkeleyovskými autorskými právy, pouze abychom zabránili obchodní reprodukci.

Pro koho je tato kniha určena

Předpokladem k úplnému pochopení výkladu knihy je, že máte základní znalosti o jazyce Java. Nebudeme vám vysvětlovat, jak současně vypsat pomocí `println` řetězec a číslo nebo jak napsat třídu, která rozšiřuje `Applet` a v okně vypíše vaše jméno. Předpokládáme, že jste absolvovali kurz o Javě nebo jste studovali nějakou knihu vysvětlující základy Javy, například *Head First Java*, *Learning Java* nebo *Java in a Nutshell* z vydavatelství O'Reilly. V kapitole 1 si však vysvětlíme určité techniky, které byste nemuseli moc dobře znát a které jsou nezbytné k pochopení později uváděných informací. Neobávejte se přeskakovat jednotlivé části! Jak tištěná, tak elektronická verze knihy obsahuje množství křížových odkazů.

* Jedna z předních technologicky vyspělých nezávislých vzdělávacích společností na světě; viz *http://www.learningtree.com/*.

Co v této knize najdete?

Narozdíl od mých kolegů Toma a Nathana nemusím tolik času věnovat zvláštnostem a idiomům jazyka: Java je příjemně novým způsobem zbavena zvláštních kliček. To však neznamená, že její dobré zvládnutí je jednoduché! V tom případě by totiž nebylo třeba této knihy. Naše hlavní koncepce spočívá tedy v zaměření na aplikační rozhraní Javy. Pomocí příkladů vás naučíme, jaké jsou aplikační rozhraní API a k čemu jsou dobrá.

Java je podobně jako Perl jazyk, na který si zvyknete a který poroste s vámi. A osobně musím připustit, že v současnosti většinou používám Javu. Věci, které bych dříve psal v C, jsou nyní – s výjimkou ovladačů zařízení a zastaralých systémů – udělány v Javě.

Java je však vhodná pro jiný rozsah úloh než jazyk Perl. Perl (a další skriptovací jazyky, například awk a Python) se hodí zejména pro „one-liner" obslužné úlohy. Jak Tom a Nathan ve své knize předvádí, Perl je vynikají pro úlohy jako např. tisk 42. řádku ze souboru. Tyto věci lze pochopitelně udělat i v Javě, nicméně jelikož Java je kompilovaný, objektově orientovaný jazyk, zdá se být vhodnější pro „vývoj ve velkém" nebo vývoj podnikových aplikací. Na tento typ vývoje se ve skutečnosti zaměřila řada aplikačních rozhraní API doplněných v Java 2. Spoustu technik si však nevyhnutelně předvedeme na kratších příkladech, a dokonce fragmentech kódu. Buďte si jistí, že každý řádek kódu, který v této knize uvidíte, byl zkompilován a spuštěn.

Celá řada delších příkladů v této knize jsou nástroje, které jsem původně napsal k automatizaci nějaké světské úlohy nebo něčeho jiného. Program `MkIndex` (popsaný v kapitole 17) například načítá nejvyšší adresář místa, kde uchovávám všechny své příklady zdrojových kódů v Javě, a pro tento adresář sestaví z hlediska prohlížeče přívětivý soubor *index.html*. U jiného příkladu bylo tělo prvního vydání zčásti sestaveno v XML, zjednodušení, které se spoléhá na léta zkušeností v SGML (původní standard, který vedl ke značkové syntaxi jazyka HTML). V tomto bodě není zřejmé, jestli bude XML primárně užitečné jako publikační formát nebo jako formát na zpracování dat, případně jestli jeho rozmach bude dále zastírat tento rozdíl, ačkoli se zdá, že zastírání rozdílů je pravděpodobnější. V této knize však formát XML používáme k vepsávání a vyznačování původního textu některých kapitol. Text byl poté převeden do publikačního formátu pomocí programu `XmlForm`. Tento program rovněž ošetřuje – pomocí dalšího programu, `GetMark` – úplné a částečné vložení kódu ze zdrojového adresáře. O `XmlForm` budeme hovořit v návodu 21.7.

Nyní si stručně projděme uspořádání této knihy. Nejprve si v kapitole 1, *Začínáme: kompilace, spuštění a ladění*, popíšeme určité metody kompilování vašeho programu na různých platformách, spouštění programů v různých prostředích (v prohlížeči, na příkazovém řádku, v okně na pracovní ploše) a ladění. V kapitole 2, *Komunikace s prostředím*, přejdeme od kompilování a spouštění vašeho programu k jeho přizpůsobování okolnímu prostředí – jiným programům, které jsou na vašem počítači.

Následujících pět kapitol pojednává o základním aplikačním rozhraní. V kapitole 3, *Řetězce*, se zaměříme na jeden z nejzákladnějších, ale výkonných datových typů v Javě. Ukážeme si, jak sestavit, rozložit, porovnat a přeuspořádat to, co byste jinak mohli považovat za obyčejný text.

V kapitole 4, *Porovnávání vzorů pomocí regulárních výrazů*, vás naučíme, jak při porovnání řetězců a porovnání vzorů v celé řadě problematických oblastí používat mocné regulární výrazy, technologii pocházející ze systému Unix. Prvním vydáním JDK, které nabízelo tuto moc-

nou technologii, byla verze 1.3; rovněž se zmíníme o několika balících regulárních výrazů třetích stran.

V kapitole 5, *Čísla*, budeme hovořit o vestavěných typech, například `int` a `double`. Také se zmíníme o odpovídajících třídách API (`Integer`, `Double` atd.) a převodových a testovacích schopnostech, jež nabízí. Krátce se rovněž zmíníme o třídách pro „velká čísla". Jelikož programátoři Javy často potřebují zpracovávat data a časy, a to jak v místním, tak v mezinárodním formátu, probereme si toto důležité téma v kapitole 6, *Data a časy*.

Další dvě kapitoly pojednávají o zpracování dat. V Javě, jako ve většině jazyků, jsou *pole* lineární, indexové kolekce podobných druhů objektů, o čemž budeme hovořit v kapitole 7, *Strukturování dat pomocí Javy*. V této kapitole pokračujeme v diskusi o celé řadě tříd „kolekcí": výkonných způsobech uchovávání množství objektů v balíku `java.util`. Do této části byla v druhém vydání přidána nová kapitola. JDK 1.5 představila nový rozměr v pojetí strukturování dat a to převzetím konceptu šablon C++ do Java Collections; výsledek označovaný jako generičnost je hlavním námětem kapitoly 8, *Strukturování dat pomocí generičnosti, foreach a výčtů (JDK 1.5)*.

Navzdory určité syntaktické podobnosti s procedurálními jazyky jako C je Java ve své podstatě objektově orientovaný jazyk. V kapitole 9, *Objektově orientované techniky*, si probereme určité klíčové koncepty OOP (objektově orientovaného programování), jak se týkají Javy včetně obvykle předefinovaných metod třídy `java.lang.Object`, a důležité otázky návrhových vzorů.

Několik dalších kapitol pojednává o aspektech tradičního vstupu a výstupu. V kapitole 10, *Vstup a výstup*, si podrobně vysvětlíme pravidla pro načítání a zapisování do souborů. (Nepřeskakujte tuto kapitolu, pokud si myslíte, že soubory jsou nudné, protože některé informace budete potřebovat v nadcházejících kapitolách: v kapitole 12 budete číst a zapisovat na sériových nebo paralelních portech a na soketovém síťovém připojení v kapitole 16!) V kapitole 11, *Operace nad adresáři a systémem souborů*, si ukážeme všechno ostatní o souborech – například hledání jejich velikost, a času poslední modifikace – a o čtení a modifikaci adresářů, vytváření dočasných souborů a přejmenovávání souborů na disku. V kapitole 12, *Programování externích zařízení: sériové a paralelní porty*, vám ukážeme, jak můžete používat API `javax.comm` pro čtení-zapisování na sériových a paralelních portech pomocí standardního API Javy.

V kapitole 13, *Grafika a zvuk*, se podíváme na vývoj grafického uživatelského rozhraní (GUI). Tato kapitola představuje směsici informací o činnostech na nižší úrovni, například kreslení grafiky a nastavování typu písma a barev, ale také činností na velmi vysoké úrovni, například ovládání videoklipů nebo filmů. V kapitole 14, *Grafická uživatelská rozhraní*, se budeme zabývat aspekty GUI na vyšší úrovni, například tlačítky, návěštími, nabídkami a dalšími – předefinovanými komponentami GUI. Jakmile získáte GUI (ve skutečnosti dříve, než jej opravdu napíšete), budete si chtít přečíst kapitolu 15, *Podpora cizích jazyků a lokalizace*, protože vaše programy mohou fungovat v Akbaru, Afganistanu, Alžíru, Amsterdamu nebo Angleterru stejně tak dobře jako v Albertě, Arkansasu nebo Alabamě...

Jelikož Java byla původně publikována jako „programovací jazyk pro Internet", je pouze spravedlivé, že nějaký čas věnujeme práci se sítí. V kapitole 16, *Síťoví klienti*, budeme pojednávat o základech síťového programování ze strany klienta se zaměřením na sokety. Poté se v kapitole 17, *Serverová Java: Sockety*, přesuneme na stranu serveru. V kapitole 18, *Síťoví klienti II: Aplety a weboví klienti*, vás naučíme další techniky prováděné na straně klienta. Programy na Internetu často musí generovat e-maily, takže tato část končí kapitolou 19, *Java a elektronická pošta*.

V kapitole 20, *Databázový přístup*, si vysvětlíme základy balíků JDBC (Java Database Connectivity) a JDO (Java Data Objects). Ukážeme vám, jak se můžete připojit k místním nebo vzdáleným relačním databázím, ukládat a vybírat data a zjišťovat informace o výsledcích dotazu nebo o databázi.

Další tvar ukládání a výměny data nabízí jazyk XML. V kapitole 21, *XML*, si probereme formáty XML a některé operace, které můžete použít pomocí dvou standardních API Javy: SAX a DOM.

V kapitole 22, *Distribuovaná Java: RMI*, pokročíme s konceptem distribuce o krok vpřed a vysvětlíme si RMI (Remote Method Invocation), mechanismus vzdáleného volání procedur Javy. RMI vám dovolí sestavovat klienty, servery, a dokonce scénáře „zpětného volání" pomocí standardního mechanismu Javy – rozhraní – pro popis dohody mezi klientem a serverem.

V kapitole 23, *Balíky a balíkování*, si ukážeme, jak vytvářet balíky tříd, které vzájemně spolupracují. V této kapitole budeme také hovořit o „implementaci" nebo distribuci a instalaci vašeho softwaru.

V kapitole 24, *Zpracování vláken v Javě*, se naučíme psát třídy, které vypadají, že dělají více úloh najednou a dovolí vám využít výkonu víceprocesorového hardwaru.

V kapitole 25, *Pohled dovnitř neboli introspekce*, vám odhalíme tajemství, jak mechanicky psát dokumenty API s křížovými odkazy („staňte se ve svém volném čase věhlasnými autory knihy o Javě!") a jak jsou webové prohlížeče schopny načítat jakýkoli starý aplet – aniž by kdy předtím spatřily tuto specifickou třídu – a spustit ho.

Někdy již máte kód napsaný a pracujete v jiném jazyce, který za vás může udělat část práce nebo chcete použít Javu jakou součást většího balíku. V kapitole 26, *Použití Javy s jinými jazyky*, vám ukážeme, jak spustit externí program (kompilovaný nebo skript) a také přímo komunikovat s „nativním kódem" v C/C++ v jiných jazycích.

V této 800stránkové knize není místo na všechno, co bych vám rád o Javě sdělil. V *Doslovu* zmiňujeme několik závěrečných myšlenek a odkaz na moje on-line shrnutí aplikačních rozhraní Javy, o nichž by měl vědět každý vývojář Javy.

Žádní dva programátoři nebo autoři se neshodnou na nejvhodnějším pořadí uvedení všech témat Javy. Abychom vám pomohli vždy najít to, co právě hledáte, zahrnuli jsme do knihy četné křížové odkazy, v nichž se odvoláváme na příslušnou část většinou podle čísla návodu.

Poznámky k platformě

Doposud Java prošla pěti hlavními verzemi. První oficiální verzí byla JDK 1.0 a poslední opravnou verzí byla 1.0.2. Druhou hlavní verzí je Java JDK 1.1 a poslední opravnou verzí je 1.1.9, ačkoli v době, kdy budete číst tuto knihu, může být toto číslo již vyšší. V prosinci 1998 vyšla třetí hlavní verze s označením JDK 1.2, ale někdo ze společnosti Sun ji v době jejího vydání neočekávaně přejmenoval na Java 2 a implementace se nazývá jako Java 2 SDK 1.2. Aktuální verzí v době psaní prvního vydání této knihy byla Java 2 SDK 1.3 (JDK 1.3), jež byla publikována v roce 2000.

V době, kdy šlo první vydání této knihy do tisku, se objevila verze Java 2 SDK 1.4; při dokončování knihy vyšla beta verze (kterou společnost Sun označila za „prvotní přístup"), proto jsem ji mohl uvést pouze stručně. Zdá se, že druhé vydání této knihy má lepší načasování; verze Java 2 SDK 1.5 je při provádění aktualizace knihy ve stadiu beta-verze.

V této knize se zaměříme na pátou verzi J2SE (Java 2 Standard Edition), čili verzi 1.5. Předpokládáme, že v době publikace knihy budou všechny vývojové projekty Javy používat JDK 1.4 s velmi malým množstvím spojení na předchozí verze z historických důvodů. K testování kódu jsem kvůli přenositelnosti používal několik platforem. Testy jsem prováděl pomocí Linux JDK od Sunu. Pro masový trh jsem otestoval celou řadu programů na implementaci Win32 (Windows 2000/XP/2003) od Sunu. A „pro nás ostatní" jsem provedl velkou část mého nedávného vývoje pomocí Mac OS X verze 10.2x a novějších verzí od Apple. Jelikož je však Java přenositelná, očekávám, že drtivá většina příkladů bude fungovat na jakékoliv platformě podporující Javu, kromě platforem vyžadujících zvláštní API. Všechny příklady nebyly testována na každé platformě, ale všechny byly testovány alespoň na jedné platformě – a většina příkladů na více platformách.

Java API se skládá ze dvou částí: základní API (Core API) a doplňkové API. Základní API je samozřejmě to, které je obsaženo v JDK, jež si zdarma stáhnete z webových stránek *http://java.sun.com/*. Všechno ostatní je doplňkové. Přesto však toto „základní" API není vůbec malé: čítá kolem 50 balíků, hodně přes 2 000 veřejných tříd a na každou třídu v průměru připadá asi 12 veřejných metod. Programy, které se drží tohoto základního API, mají přiměřeně zajištěnou přenositelnost na jakoukoli platformu Javy.

Doplňkové API se dále dělí na standardní rozšíření a nestandardní rozšíření. Všechny názvy balíčků standardních rozšíření mají prefix `javax.`* (a Sun poskytuje referenční implementace). K implementaci každého standardního rozšíření se nevyžaduje licence Javy (jako například Apple nebo IBM), ale pokud vyžadována je, mělo by se dodržovat rozhraní standardního rozšíření. V této knize upozorňujeme na jakýkoli kód, který závisí na standardním rozhraní. Malé množství kódu zde, kromě kódu uvedeného v samotné knize, závisí na nestandardních rozšířeních. Můj vlastní balík `com. darwinsys` obsahuje některé obslužné třídy, které se tu a tam používají; jeho import uvidíte v horní části každého souboru, jenž používá třídy z tohoto balíku.

Navíc jsou standardizovány další dvě *platformy*, J2ME a J2EE. J2ME (Java 2 Micro Edition) se stará o malá zařízení jako kapesní počítače (PalmOS a jiné), mobilní telefony, faxová zařízení a podobné přístroje. V rámci J2ME existují různé „profily" pro různé třídy zařízení. J2EE (Java 2 Enterprise Edition) se naopak stará o vytváření velkých, škálovatelných, distribuovaných aplikací. Součástí J2EE jsou API Servlets, JavServer Pages, JavaServer Faces, CORBA, RMI, JavaMail, Enterprise JavaBeans™ (EJB), Transactions a další API. Balíky J2ME a J2EE mají běžně prefix „javax", protože nejsou základními balíčky J2SE. V této knize se nebudeme vůbec zabývat platformou J2ME, nicméně je zde obsaženo několik málo API J2EE, která jsou užitečná na straně klienta, například RMI a JavaMail. Jak jsme se dříve zmínili, pojednání o servletech a JSP z prvního vydání bylo odstraněno, protože nyní již existuje kniha *Servlet and JSP Cookbook*.

Jiné knihy

Do této knihy jsme napěchovali množství užitečných informací. V důsledku rozsahu námětů však není možné poskytnout jakémukoli jednomu tématu pojednání o délce samostatné knihy. Kvůli tomu kniha rovněž obsahuje odkazy na celou řadu webových stránek a dalších knih. To je v souladu s naším cílovým publikem: člověkem, který se chce dozvědět více informací o Javě.

* Všimněte si, že všechny balíky s názvy javax. představují rozšíření: javax.swing a jeho podbalíky (balíky Swing GUI) bývaly rozšířeními, ale nyní jsou v základním API.

Podle mého názoru publikuje nejlepší výběr knih o Javě na trhu nakladatelství O'Reilly. Své pokrytí rozšiřuje současně s rozšiřováním API. Nejnovější verze a informace o objednávání můžete najít v on-line sekci knih o Javě nakladatelství O'Reilly na stránkách *http://java.oreilly.com* a knihy si můžete zakoupit ve většině knihkupectví a to jak kamenných, tak elektronických. Rovněž si knihy můžete číst on-line přes službu placeného přihlášení; viz *http://safari.oreilly.com*. I když celá řada publikací je zmíněna na příslušných místech v této knize, některé z nich si zaslouží, abychom je uvedli i zde.

Nejprve a především bych chtěl zmínit knihu *Java in Nutshell* od Davida Flanagana, která nabízí stručné shrnutí jazyka a API a podrobné odkazy na nejzákladnější balíky. Taková kniha by měla mít své místo vedle vašeho počítače. Kniha *Head First Java* nabízí mnohem rozmarnější uvedení do jazyka a doporučuje se méně zkušeným vývojářům.

Definitivní (a monumentální) popis programování grafického uživatelského rozhraní Swing představuje *Java Swing*, kterou napsali Marc Loy, Robert Eckstein, Dave Wood, James Elliott a Brian Cole.

Java Virtual Machine od Jona Meyera a Troye Downinga bude fascinovat čtenáře, kteří chtějí proniknou hlouběji pod povrch. Tato kniha se již neprodává, ale lze ji sehnat použitou a v knihovnách.

Knihy *Java Network Programming* a *Java I/O* (obě napsal Elliotte Rusty Harold) a *Database Programming with JDBC and Java* od George Reeseho rovněž představují užitečné reference.

Existuje mnohem více knih, jejichž aktualizovaný seznam najdete na webových stránkách vydavatelství O'Reilly.

Další knihy o Javě

Neměli byste ani uvažovat o vydávání aplikací GUI, pokud jste nečetli oficiální knihu od společnosti Sun *Java Look and Feel Design Guidelines* (Addison Wesley). Tato práce představuje pohledy velké skupiny lidských faktorů a expertů na uživatelské rozhraní ve společnosti Sun, kteří pracovali na balíku Swing GUI od jeho vzniku; poradí vám, jak docílit, aby dobře fungoval.

Knihy o obecném programování

Inspirací pro celé generace studentů informatiky byla kniha Donalda E. Knutha *The Art of Computer Programming*, poprvé publikovaná nakladatelstvím Addison Wesley v roce 1968. První díl pojednává o základních algoritmech, díl 2 je o seminumerických algoritmech a třetí díl se zabývá řazením a hledáním. Zbývající čtyři díly v plánované sérii stále čekají na své dokončení. I když příklady z této knihy jsou docela vzdálené od Javy (autor si pro své příklady vymyslel hypotetický jazyk symbolických instrukcí), celé řady pojednání o algoritmech – o tom, jak by měly být počítače užívané k řešení skutečných problémů – jsou stejně tak relevantní dnes, jako byly před lety.*

Kniha od Kernighana a Plaugera, *The Elements of Programming Style*, ačkoli dnes již poněkud zastaralá, nastavuje styl (doslova) pro generování programátorů pomocí příkladů z různých strukturovaných programovacích jazyků. Kernighan a Plauger také napsali dvojici knih, *Software Tools* a *Software Tools in Pascal*, které uvádí natolik dobré rady o programování, že jsem všem programátorům radil, aby si tyto knihy přečetli. Nyní jsou však tyto tři knihy již zastaralé;

* S výjimkou rozhodnutí algoritmů, které jsou kvůli pozoruhodným změnám v dnešním dostupném výpočetním výkonu méně relevantní.

mnohokrát jsem chtěl napsat nějaké volné pokračování v modernějším jazyce, ale raději to přenechám zkušenějším. Kniha *The Practice of Programming* je Brianovo pokračování řady *Software Tools*, které napsal společně s Robem Pikem a navazuje na tradici dokonalosti softwarových učebnic od společnosti Bell Labs (nyní část Lucentu). V návodu 3.13 jsem dokonce z jejich knihy převzal jednu část kódu.

Rovněž se podívejte na knihu *The Pragmatic Programmer* od Andrew Hunta a Davida Thomase (Addison Wesley).

Knihy o návrhu

Kniha *Java Design* od Petera Coada (PTR-PH/Yourdon Press) se výslovně zaměřuje na problémy objektově orientovaných analýz a návrhu u Javy. Coad je poněkud kritický k implementaci paradigmatu pozorovatelného pozorovatele (observable-observer) Javy a místo něj nabízí vlastní náhradu.

Jednou z nejproslulejších knih o objektově orientovaném návrhu za poslední roky je *Design Patterns* od autorů Gamma, Helm, Johnson a Vlissides (Addison Wesley). Těmto autorům se často souhrnně říká „skupina čtyř" a následkem toho se někdy označuje jako „kniha GOF" (Gang of four – skupina čtyř). Jeden z mých kolegů ji označil za „vůbec nejlepší knihu o objektově orientovaném designu" a já s ním souhlasím; přinejmenším patří mezi nejlepší.

Kniha *Refactoring* od Martina Fowlera pojednává o celé řadě technik „čištění programového kódu", pomocí nichž můžete zlepšit čitelnost a údržbu kódu. Podobně jako kniha GOF i tato publikace uvedla novou terminologii, která pomáhá vývojářům a ostatním při komunikaci o tom, jak má být kód navržen. Fowlerova kniha poskytovala slovník pro diskusi o zlepšování. Celá řada „refaktorování" se nyní objeví v nabídce Refactoring integrovaného vývojového prostředí Eclipse (viz návod 1.3).

V současné době jsou v oběhu dva důležité směry teorií metodologie. První směr se souhrnně označuje jako metody Agile a jeho nejznámějším členem je extrémní programování (XP). Metodologie XP (nejedná se o verzi operačního systému společnosti Microsoft) je představena v řadě malých, krátkých, čitelných textů pod vedením jejího designéra Kenta Becka. Dobré shrnutí všech metod Agile nabízí kniha *Agile Software Development Ecosystems* od Highsmitha. První knihou v řadě XP je *Extreme Programming Explained*.

Další skupinou důležitých knih o metodologii, které se zabývají tradičnějším objektově orientovaným návrhem, je řada UML vedená „třemi Amigos" (Boochem, Jacobsonem a Rumbaughem). Mezi hlavní práce těchto autorů patří *UML User Guide*, *UML Process* a další. Menší a přístupnější knihou ve stejné sérii je *UML Distilled* od Martina Fowlera.

Konvence používané v knize

V této knize budeme používat následující konvence.

Programovací konvence

Nejprve si trochu vysvětleme pojmy, s kterými se budete v knize setkávat. Slovem *program* označujeme buď aplet, servlet nebo aplikaci. *Aplet* se používá v prohlížeči. *Servlet* je podobný jako aplet, ale používá se na serveru. A *aplikace* je jakýkoli jiný typ programu. Desktopová aplikace (alias *klient*) komunikuje s uživatelem. Serverový program komunikuje s uživatelem nepřímo obyčejně prostřednictvím síťového připojení.

Příklady jsou uváděny ve dvou variantách. Kompletní příklady, začínající žádným nebo několika příkazy import, komentářem Javadoc a příkazem veřejné třídy a výtahy začínající deklarací nebo proveditelným příkazem. Zkompilovány a spuštěny však přirozeně byly kompletní verze těchto výtahů, které najdete v online zdrojových kódech k této knize.

Návody jsou číslovány podle kapitol a pořadového čísla, takže například návod 7.5 odkazuje na pátý návod v kapitole 7.

Konvence sazby

V knize se používají následující typografické konvence:

Kurziva

Slouží k vyznačení příkazů, názvů souborů a adres URL. Pomocí kurzivy rovněž definujeme nové pojmy při jejich prvním výskytu v textu.

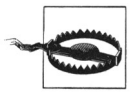
`Písmo se stejnou šířkou znaků`

Používá se v příkladech pro částečné nebo úplné znázornění zdrojových kódů programů v Javě. `Neproporcionální písmo` se rovněž používá pro názvy tříd, názvy metod, názvy proměnných a jiné fragmenty kódu v jazyce Java.

`Tučné písmo se stejnou šířkou znaků`

Slouží k vyznačení uživatelského vstupu, jako například příkazů, které zadáváte na příkazovém řádku.

Tato ikona značí tip, doporučení nebo obecnou poznámku.



Tato ikona značí varování nebo upozornění.

Celou řadu programů doprovází příklad spouštěný z příkazového řádku, který ukazuje jejich použití v praxi. V příkladech je obvykle uveden prompt končící buď symbolem $ pro Unix nebo > pro Windows v závislosti na počítači, s kterým jsem právě pracoval. Text před tímto symbolem můžete ignorovat; buď se, opět v závislosti na systému, jedná o cestu nebo název počítače.

Komentáře a otázky

Jak jsem se dříve zmínil, veškerý kód jsem testoval přinejmenším na jedné z uváděných platforem a ve většině případů hned na několika platformách. Přesto některé programy mohou být závislé na platformě, nebo dokonce můžete narazit na chyby v kódu či v nějaké důležité implementaci Javy. Jakékoli chyby, které najdete, a také podněty pro další vydání oznamte a zašlete na adresu:

O'Reilly Media, Inc.
1005 Gravenstein Highway North

Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

Máte-li dotazy technické povahy nebo nějaké technické komentáře týkající se knihy, zašlete e-mail na:

bookquestions@oreilly.com

Seznam tiskových chyb v knize, příklady a jakékoli další informace najdete na webových stránkách nakladatelství O'Reilly:

http://www.oreilly.com/catalog/javacook2/

Moje osobní webové stránky věnované knize najdete na:

http://javacook.darwinsys.com/

Na obou zmíněných webech najdete seznam tiskových oprav a plány týkající se budoucích vydání. Také jsou zde ke stažení zdrojové kódy ke všem příkladům v Javě; *prosím* neplýtvejte svým časem jejich přepisováním! Přečtěte si zvláštní instrukce v následující části.

Poznámka českého vydavatele: I nakladatelství Computer Press, které pro vás tuto knihu přeložilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press
redakce PC literatury
náměstí 28. dubna 48
635 00 Brno-Bystrc

nebo

knihy@cpress.cz

Stažení zdrojových kódů

Zdrojový kód si můžete stáhnout z mých webových stránek *http://javacook.darwinsys.com*, stačí klepnout na odkaz Downloads a vybrat si jednu ze tří možností:

1. Stáhnout celý zdrojový archiv jako jediný velký soubor zip.
2. Stáhnout jednotlivé zdrojové soubory indexované abecedně rovněž podle kapitol.
3. Stáhnout binární soubor JAR pro balíček `com.darwinsys.` potřebný pro kompilaci celé řady jiných programů.

Většina lidí si vybere možnosti 1 nebo 2, ale každý, kdo chce kompilovat moje kódy, bude muset sáhnout po třetí volbě. Informace o použití těchto souborů najdete v návodu 1.5.

Stažením celého zdrojového archívu získáte velký soubor zip se všemi soubory z knihy (a dalšími). Tento archív lze rozbalit pomocí *jar* (viz návod 23.4), volně dostupného programu zip od Info-ZIP, komerčních programů WinZip nebo PKZIP či libovolného kompatibilního nástroje. Soubory jsou uspořádány v podadresářích podle tématu, jeden podadresář pro řetězce (kapitola 3), regulární výrazy (kapitola 4), čísla (kapitola 5) a podobně. Soubory v archívu jsou indexovány podle názvu a podle kapitol, takže můžete jednoduše najít soubory, které potřebujete.

Stažení jednotlivých souborů je rovněž snadné: stačí následovat odkazy buď podle názvu souboru/podadresáře nebo podle kapitoly. Jakmile v prohlížeči uvidíte hledaný soubor, použijte

příkaz Soubor – Uložit nebo ekvivalentní techniku, případně pouze zkopírujte a vložte soubor z prohlížeče do editoru nebo nástroje IDE.

Soubory se pravidelně aktualizují, takže nebuďte překvapeni, pokud najdete rozdíly mezi tím, co je vytištěno v knize, a tím, co jste si stáhli. Díky tomu se můžete poučit z minulosti.

Stažení českých zdrojových kódů

Nakladatelství Computer Press pro vás nechalo připravit české zdrojové kódy, tak jak jsou použity v textu knihy. Všechny balíčky kódu si můžete stáhnout z webových stránek knižního obchodu nakladatelství Computer Press ze sekce určené této knize. Zdrojové kódy naleznete na adrese http://knihy.cpress.cz/K1223.

Poděkování

Můj život mnohokrát ovlivnilo řízení osudu, díky kterému jsem se ve správnou dobu setkal se správným člověkem, aby mi ukázal správnou věc. K počítačům mě přivedl Steve Munroe, s nímž jsem již dlouho nebyl v kontaktu – zejména k IMB 360/30 v Torontském školním výboru, který byl větší než obývák, měl 32 nebo 64 KB paměti a výkon snad PC/XT – v roce 1970. Na Univerzitě v Torontu mě vzal pod svá křídla Herb Kugel, když jsem se učil o větších sálových počítačích IBM, které přišly později. Terry Wood a Dennis Smith na Univerzitě v Torontu mě převedli k mini- a mikropočítačům dříve, než spatřilo světlo světa IBM PC. Po večerech a o víkendech mi torontský obchodní klub Toastmasters International *(http://www.toastmasters.org)* a kanadská škola SCUBA Al Lamberta umožnily vyvíjet můj veřejný projev a vychovatelské schopnosti. Funkcionality a výhody operačního systému Unix mě naučilo několik lidí na Univerzitě v Torontu, ale zejména Geoffrey Collyer, a to v době, kdy jsem byl připraven se ho učit.

První kurz společnosti Learning Tree, na který jsem docházel, vyučoval Greg Davidson z UCSD a ten mě také přivítal jako lektora Learning Tree. O několik let později, když měl být na webových stránkách Sunu vydán jazyk Oak, mě Greg podnítil, abych napsal Jamesi Goslingovi a něco se o něm dozvěděl. Jamesova odpověď z 29. března 1995, že právníci přejmenovali jazyk na Javu a že byla „právě teď" dostupná ke stažení, představuje cenný první záznam v mé poštovní schránce vyčleněné pro Javu. Mike Rozek mě přijal jako autora kurzů Learning Tree pro unixový kurz a dva javové kurzy. Po odchodu Mika ze společnosti, obstarávali produktový management těchto kurzů Francesco Zamboni, Julane Marx a Jennifer Urick. Jennifer rovněž pro mě obstarala povolení „znovu použít" v této knize určité zdrojové kódy, který byly dříve použity v mých poznámkách. Nakonec děkuji celé řadě lektorů Learning Tree a studentům, kteří mi ukázali, jak zlepšovat mé prezentace. Stále vyučuji pro Learning Tree a jejich kurzy doporučuji všem vytíženým vývojářům, kteří se v průběhu čtyřech dnů chtějí podrobně zaměřit na jedno téma. Jejich webové stránky jsou na *http://www.learningtree.com*.

Vraťme se ale blíže k tomuto projektu. Tim O'Reilly věřil v „malou knihu Lint", když byla pouze ve stadiu vzorku kapitoly, což mi umožnilo proniknout do kruhu autorů vydavatelství O'Reilly. O několik let později mě Mike Loukides podnítil, abych zkusil najít nápady pro knihu o Javě, na které bychom mohli oba spolupracovat. A naléhal na mě, když jsem nestíhal konečný termín. Mike si rovněž přečetl celý rukopis a udělal spoustu praktických komentářů, z nichž některé přivedly poletování fantazie zpět na zem. Jessamyn Read zase přeměnila celou řadu faxových a e-mailových špatně čitelných podkladů na kvalitní ilustrace, které máte možnost vidět v této knize. A mnozí další a další talentovaní lidé v nakladatelství O'Reilly pomohli uvést tuto knihu do stavu, v němž se vám dostala do rukou.

Musím také poděkovat mým prvotřídním recenzentům za první vydání, především mé drahé ženě Betty Cerar, která přesto, že ví více o kofeinových nápojích, které jsem při programování vypil, než o programovacím jazyku, s nímž pracuji, tolik prospěla během naše společného života mému psaní, a to díky svému nadšení pro čisté vyjádření a správnou gramatiku. Jonathan Knudsen, Andy Oram a David Flanagan okomentovali koncept, když byl ještě ve fázi rozšířeného seznamu kapitol a návodů a již byli schopni vidět, jakým typem knihy by se mohla stát, a doporučit věci, které dopomohou k jejímu zlepšení. Lektor společnosti Learning Tree, Jim Burgess, přečetl značnou část prvního vydání s velmi kritickým pohledem na vyjadřování, formulaci a kód. Bil Lewis a Mike Slinn *(mslinn@mslinn.com)* vytvořili prospěšné komentáře na více návrzích knihy. Ron Hitchens *(ron@ronsoft.com)* a Marc Loy pečlivě přečetli celý konečný návrh prvního vydání. Za povzbuzování a podporu během psaní jsem vděčný Miku Loukidesovi. Redaktorka Sue Miller pomohla přivést rukopis přes poněkud energické konečné fáze produkce. Sarah Slocombe přečetla celou kapitolu o XML a vytvořila spoustu bystrých námětů; bohužel čas mi nedovolil zahrnout všechny z nich do prvního vydání. Každý ze zmíněných lidí zlepšil tuto knihu v mnoha směrech, zejména prostřednictvím doporučení na další návody nebo na revize stávajících návodů. Chyby, které zůstaly, jsou mé vlastní.

Při přípravě, kompilaci a testování prvního vydání jsem používal různé nástroje a operační systémy. Poděkování si zaslouží vývojáři OpenBSD *(http://www.openbsd.org)* „proaktivně zabezpečeného unixového systému“, za vytvoření stabilního a bezpečného unixového klonu, který je také blíže k tradičnímu Unixu než ostatní freewarové systémy. Při vkládání původního rukopisu v XML jsem používal editor *vi* (*vi* na OpenBSD a *vim* na Windows) a na formátování dokumentů Adobe FrameMaker. Každá z těchto aplikací je svým způsobem vynikající nástroj, ale musím upozornit na jednu maličkost týkající se aplikace FrameMaker. Než jsem začal revizní cyklus, společnost Adobe měla od vydání systému OS X čtyři roky na to, aby vytvořila aktuální verzi FrameMakeru pro Macintosh. Nestalo se tak, čímž jsem byl nucen provádět revizi ve stále více prastarém prostředí Classic. V průběhu tohoto období však jejich prodeje Mac neustále klesaly, až se během konečné fáze produkce této knihy dostaly na hranici, kdy společnost Adobe oficiálně oznámila, že již nebude vytvářet žádné verze tohoto vynikajícího publikačního softwaru pro Macintosh.

Žádná kniha o Javě by nebyla kompletní bez quadriumu*, takže díky patří Jamesovi Goslingovi za vymyšlení prvního unixového Emacsu, tabulkového procesoru *sc*, systému NeWS a Javy. Díky také jeho zaměstnavateli, společnosti Sun Microsystems (NASDAQ SUNW), za vytvoření nejenom jazyka Javy, ale neuvěřitelného množství nástrojů pro Javu a knihoven API, které jsou volně dostupné na Internetu.

Díky Tomu a Nathanovi za knihu *Perl Cookbook*. Bez nich bych nikdy nepřišel s tímto formátem knihy.

Willi Powellovi ze společnosti Apple Canada za zpřístupnění Mac OS X v počátcích systému OS X; nyní mám vlastní notebook Apple. Rovněž děkuji Apple za nasazení OS X na BSD Unix a za to, že se stala největší komerční unixovou společností na světě.

Díky Dunutsu Tima Hortona v Boltonu, Ontario za skvělé kafe a za nevyžadování 20minutového tabulkového limitu na chlapíka s počítačem.

Všem a každému z vás patří moje upřímné díky.

* Je dobře, že James vymyslel pouze čtyři hlavní technologie, nikoli pět. Jinak bych musel přeformulovat celou větu, abych se vyhnul porušení ochranné známky Intelu.

Kapitola 1

Začínáme: překlad, spuštění a ladění

V této kapitole vám vysvětlím určité základní úlohy, jejichž zvládnutí je nezbytné pro další práci s knihou – říká se, že nejprve musíte umět lézt a teprve poté můžete začít chodit, nebo že než budete moci řídit kolo, musíte umět chodit. Takže než si cokoli v této knize vyzkoušíte, musíte umět přeložit a spustit váš kód v Javě. Nejprve si tedy přečtěte tuto kapitolu, v níž si ukážeme několik metod, jak překládat a spouštět programy: metodu s využitím sady JDK (Java Development Kit – vývojářská sady Javy), s využitím nástroje Ant nebo nějakého IDE (Integrated Development Environment – integrované vývojové prostředí). Poté se budeme zabývat dalším problémem, s kterým se lidé setkávají, a tím je správné nastavení proměnné CLASSPATH. Následně si povíme něco o apletech – to pro případ, že s nimi budete pracovat. Dále si probereme varování o použití zavrženého kódu, protože na něj pravděpodobně při údržbě „starého“ kódu Javy narazíte.* Kapitolu zakončíme obecnými informacemi o podmíněném překladu, testech jednotek, asercích a ladění.

Jste-li se svým vývojovým prostředím spokojeni, možná budete chtít tuto kapitolu částečně nebo zcela přeskočit. Informace jsou zde však obsaženy proto, aby všichni uměli překládat a ladit své programy dříve, než postoupíme k dalšímu výkladu.

1.1 Překlad a spuštění programů v Javě: JDK

Problém

Potřebujete přeložit a spustit svůj program v Javě.

Řešení

Jedná se o jednu z mála oblastí, kde operační systém počítače zasahuje do přenositelnosti Javy, proto je nejprve třeba trochu podrobnější vysvětlení.

JDK

Práce s vývojářskou sadou nástrojů JDK (Java Development Kit) na příkazovém řádku může být nejlepší způsob, jak držet krok s nejnovějšími vylepšeními společnosti Sun. Nejedná se o nejrychlejší překladač dosažitelný všemi prostředky; překladač je napsaný v Javě a interpre-

* Obrat „starý kód Javy“ je myšlen s trochou nadsázky, což by mělo být zřejmé, když zjistíte, že Java nebyla v době prvního vydání této knihy v oběhu ani pět let.

tován v době překladu, což z něj dělá rozumné samozaváděcí řešení, avšak z hlediska rychlosti vývoje nemusí bezpodmínečně představovat optimální řešení. Pokud se i přesto rozhodnete pro používání JDK od Sunu, program přeložíte zadáním příkazu `javac` a spustíte příkazem `java` (ve Windows můžete navíc využít i příkazu `javaw`, kterým program spustíte bez okna konzoly). Například:

```
>javac HelloWorld.java

>java HelloWorld
Ahoj, všichni

>
```

Jak si můžete z (chybějícího) výstupu překladače všimnout, tento překladač prosazuje filozofii Unixu „dobrá zpráva je žádná zpráva“: když je program schopen udělat přesně to, co po něm požadujete, neměl by se obtěžovat se sdělením, že to provedl. Takový překladač, případně nějaký jeho klon, používá hodně lidí.

Existuje systémová proměnná (a také parametr příkazového řádku) s názvem CLASSPATH (budeme o nich hovořit v návodu 1.4), která nastavuje cesty, na nichž Java hledá třídy. Je-li systémová proměnná CLASSPATH nastavena, používá ji jak javac, tak java. Ve starších verzích Javy jste museli nastavit CLASPATH tak, aby obsahovala „.“, dokonce i pro spuštění jednoduchého programu z aktuálního adresáře; to již pro aktuální implementace Javy neplatí. Možná to však bude platit pro některé klony.

Alternativy příkazového řádku

Překladač *javac* od Sunu je oficiální referenční implementací. Je však napsán v Javě, a tudíž se musí interpretovat za běhu. Některé jiné překladače Javy jsou napsány v jazycích C/C++, takže jsou o něco rychlejší než referenční překladač. V zájmu zrychlení svých překladů jsem používal nástroj Jikes, který je rychlý (C++), zdarma a dostupný pro Windows i pro Unix. Také se snadno instaluje a je součástí balíku Mac OS X Developer Tools. Aktuální spustitelnou verzi pro Windows, Linux a další unixové systémy můžete najít na webových stránkách IBM věnovaných překladači Jikes. Pracujete-li se systémy OpenBDS, NetBSD nebo FreeBSD, mělo by vám stačit zadat:

```
cd /usr/ports/lang/jikes; sudo make install
```

nebo podobný příkaz, případně si stáhnout soubor balíku a k instalaci použít *pkg_add*. Podívejte se na stránky *http://jikes.sourceforge.net/*, na kterých najdete podrobnější informace o programu Jikes, a rovněž si můžete odtud tento překladač stáhnout.

Co se mi na Jikes skutečně líbí, je, že ve srovnání s překladačem JDK poskytuje mnohem lepší chybová hlášení. Upozorní vás například na nepatrně nesprávně napsané názvy. Hlášení Jikes jsou často trochu rozvláčná, ale pomocí volby +E můžete nastavit stručnější formát. Jikes nabízí spoustu dalších voleb příkazového řádku, z nichž celá řada koresponduje s volbami překladače JDK, nicméně některé jej přesahují. Podrobné informace najdete v online dokumentaci k programu Jikes.

Další alternativní technologií je program Kaffe, který je licencován společností Transvirtual, ale je také dostupný ve formě open-souce (na webu http://www.kaffe.org) pod licencí GNU GPL (General Public License). Kaffe se snaží být plnohodnotnou náhradou vývojářské sady JDK, nicméně jeho vývoj postupuje dost pomalu a aktuální klon Javy nebyl v době psaní této knihy ještě úplně hotov. Kaffe je opět k dispozici pro systémy BSD Unix a Linux ve formátu RPM. Aktuální informace o programu Kaffe najdete na jeho webových stránkách.

Mezi další freewarové programy patří Japhar – klon virtuálního stroje Javy (dostupný na webových stránkách *http://www.hungry.com/old-hungry/products/japhar/*) a IBM Jikes Runtime (dostupný na stejných webových stránkách jako Jikes).

Chcete-li zkusit opravdu něco neobvyklého, prozkoumejte JNODE (Java New Operating System Dessing Effort) na webu *http://www.jnode.org/*. JNODE je kompletní operační systém napsaný v Javě – jakési prokázání principu. V tento okamžik nepředstavuje JNODE systém, který byste pravděpodobně chtěli používat na svém hlavním počítači – nabootoval jsem jej pouze pod Virtual PC na Mac OS X – ale v budoucnu by jím být mohl.

Mac OS X

JDK představuje pouhý příkazový řádek. Na druhém konci spektra z pohledu klávesnice-zobrazení máme platformu Apple Macintosh. O tom, jak skvělé je uživatelské rozhraní platformy Macintosh, byly napsány celé knihy a do této debaty nebudeme vstupovat. Systém Mac OS X (Mac OS verze 10.x) představuje novou technologickou základnu, která je postavena na systémech BSD Unix. Jako taková má normální příkazový řádek (aplikaci Terminal skrytou v adresáři */Applications/Utilities*) a také tradiční nástroje Macintoshů. Obsahuje plnou implementaci Javy včetně dvou balíčků GUI: Swing od Sunu a vlastního balíčku Cocoa od Apple. V době psaní této knihy byla pro Mac OS 10.3 vydána JDK 1.4.2; nejnovější verze je vždy k dispozici na Software Update.

Uživatelé Mac OS X mohou požívat nástroje příkazového řádku sady JDK, jak jsme se dříve zmínili, nebo nástroj Ant (viz návod 1.7). Přeložené třídy lze zabalit do „aplikací spustitelných poklepáním“ pomocí nástroje Jar Packeger, o němž budeme hovořit v návodu 23.7. Případně mohou příznivci platformy Macintosh použít nějaké z celé řady vývojových prostředí IDE, o nichž budeme hovořit v návodu 1.3.

1.2 Upravování a překlad pomocí editoru s podporou barevného zvýrazňování syntaxe

Problém

Unavují vás nástroje příkazového řádku, ale nejste připraveni pracovat v prostředí IDE.

Řešení

Použijte editor, který barevně zvýrazňuje kód.

Diskuse

Editory s podporou Javy sice nenabízejí tolik, co vývojová prostředí IDE (viz následující návod), ale jsou lepší než příkazový řádek. Nástroje jako TextPad *(http://www.textpad.com)*, Visual Slick Edit a další jsou levné editory (zejména pro Windows), které mají vestavěnou schopnost do určité míry rozpoznat kód Javy a překládat v rámci editoru. TextPad rozpozná docela velké množství typů souborů včetně dávkových souborů, shellových skriptů, C, C++, Java, JSP, JavaScript a mnoho dalších. U všech těchto typů znázorní pomocí barevného zvýraznění, která část upravovaného souboru obsahuje klíčová slova, komentáře, řetězce v apostrofech atd. Barevné zvýrazňování je velmi užitečné z hlediska zjištění, která část vašeho kódu byla pohlcena neukončeným komentářem `/*` nebo chybějícími uvozovkami. I když to není totéž jako důkladné porozumění jazyku Java, kterým mohou disponovat IDE, zkušenosti ukáza-

ly, že tyto editory rozhodně napomáhají produktivitě programátora. TextPad rovněž nabízí příkazy „přeložit Javu“ a „spustit externí program“. Výhodou obou zmíněných příkazů je, že shromáždí celý výstup příkazu do okna, v němž lze snáze rolovat než v oknech příkazového řádku na některých platformách. Na druhé straně nevidíte výsledky příkazu, dokud se program neukončí, což není zrovna příhodné, pokud grafické uživatelské rozhraní vaší aplikace vyvolá výjimku dříve, než otevře své hlavní okno. I přes tuto drobnou nevýhodu je TextPad velmi užitečný nástroj. K dalším editorům, které nabízí barevné zvýrazňování, patří *vim* (rozšířená verze unixového nástroje *vi*, k dispozici je pro platformy Windows a Unix na stránkách *http://www.vim.org*), neustále populární editor Emacs a celá řada dalších.

Zmíníme se o editoru Emacs, jelikož je ve své podstatě tak rozšiřitelný, že lidé pro něj sestavili rozšířené schopnosti pro Javu. Příkladem těchto schopností je JDEE (Java Development Environment for Emacs), „hlavní režim“ Emacs (režim `jde-mode` založený na `c-mode`) s množinou položek menu – například Generate Getters/Setters. Mohli byste tvrdit, že JDEE představuje něco mezi používáním editoru s barevným zvýrazňováním a vývojovým prostředím IDE. Webové stránky věnované JDEE najdete na adrese *http://jdee.sunsite.dk/*.

Emacs má i bez JDEE funkcionalitu `dabbrev-expand`, která dokončuje názvy tříd a metod. Její funkce je však založena na tom, co máte ve svých aktuálních editačních bufferech, takže nezná třídy ve standardním API nebo v externích souborech Jar. Chcete-li dosáhnout takové úrovně funkcionality, musíte se poohlédnout po plnohodnotných IDE a právě o nich budeme hovořit v návodu 1.3.*

1.3 Překlad, spouštění a testování pomocí IDE

Problém

Práce s několika samostatnými nástroji není příliš pohodlná.

Řešení

Používejte integrované vývojové prostředí (IDE).

Diskuse

Většina programátorů brzy zjistí, že práce s několika samostatnými nástroji – textovým editorem, překladačem, programem na spouštění, a to se ani nezmiňuji o nástroji na ladění (viz návod 1.13) – není příliš pohodlná. Všechny tyto programy spojuje do jedné sady nástrojů s (doufejme konzistentním) grafickým uživatelským rozhraním *integrované vývojové prostředí* (IDE**).

K dispozici je řada IDE sahající až po úplně integrované nástroje s vlastními překladači a virtuálními stroji. Domnělou snadnost použití množiny funkcí těchto nástrojů završují prohlížeče tříd a další funkce IDE. Mnohokrát se polemizovalo, zda IDE skutečně zvyšují produktivitu vývojářů nebo je práce s nimi pouze zábavnější. Dokonce i vývojáři společnosti Sun připouští (možná kvůli podpoře svých inzerentů), že IDE je často produktivnější, přestože skrývá spou-

* Poznámka českého vydavatele: Můžeme doporučit například český editor PSPad, který je k dispozici zdarma.

** Vyslovit nebo napsat pojem integrované vývojové prostředí trvá příliš dlouho, proto budeme dále používat pouze zkratku IDE. Vím, že si zkratky dobře pamatujete, zejména TLA (zkratky o třech písmenech).

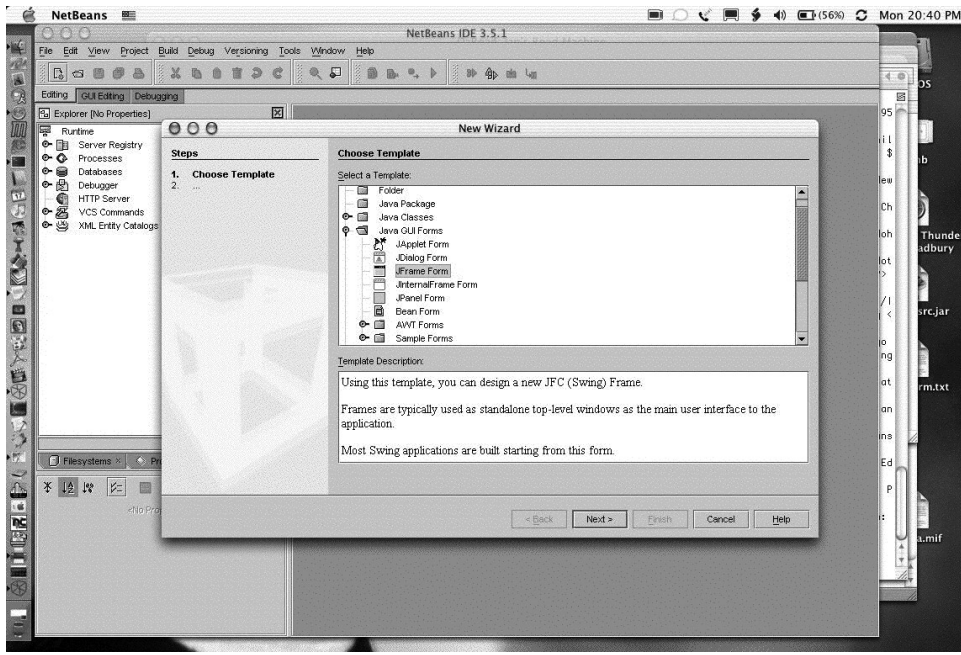
stu implementačních údajů a má sklon generovat kód, který vás spoutává s konkrétním IDE. Na Java Jumpstart CD od Sunu (součást *Developer Essentials*) je uvedeno:

> Software JDK obsahuje minimální sadu nástrojů, proto se doporučuje, aby vývojáři používali mimo softwaru JDK i profesionální integrované vývojové prostředí. Klepnutím na některý z níže uvedených obrázků můžete navštívit externí webové stránky, kde se dozvíte podrobnější informace.

Za tímto sdělením následují nějaké (pravděpodobně placené) reklamní odkazy na různé komerční vývojové sady nástrojů. Zjistil jsem, že IDE s funkcí „přírůstkového překladu", jež zpozoruje a ohlásí chyby překladu v době psaní kódu místo toho, abyste museli čekat, až dopíšete kód, poskytuje většině programátorů poněkud zvýšenou produktivitu. Jinak nehodlám dále polemizovat o výhodách a nevýhodách práce s vývojovými nástroji IDE ve srovnání s nástroji příkazového řádku; osobně používám oba režimy v závislosti na situaci a projektu. Pouze vám ukážu pár příkladů použití několika nástrojů IDE pro Javu.

Jedním IDE, které běží jak na platformě Windows, tak na Unixu, je NetBeans, které si můžete stáhnout zdarma. Toto IDE původně vytvořila společnost NetBeans.com*, avšak bylo tak výborné, že Sun koupil společnost a nyní distribuuje IDE ve dvou verzích, které sdílí převážnou část kódu: NetBeans (dříve označované Forte a distribuované jako open-source) a Sun One Studio (komerční verze). Existuje doplněk API; některé soubory *.nbm* budou fungovat v opensource i ve verzi Studio, zatímco jiné fungují pouze v jedné, nebo druhé verzi. Volně dostupnou verzi a rozšiřující moduly si můžete stáhnout z webových stránek *http://www.netbeans.org*; komerční verzi lze získat na stránkách *http://www.sun.com/*.

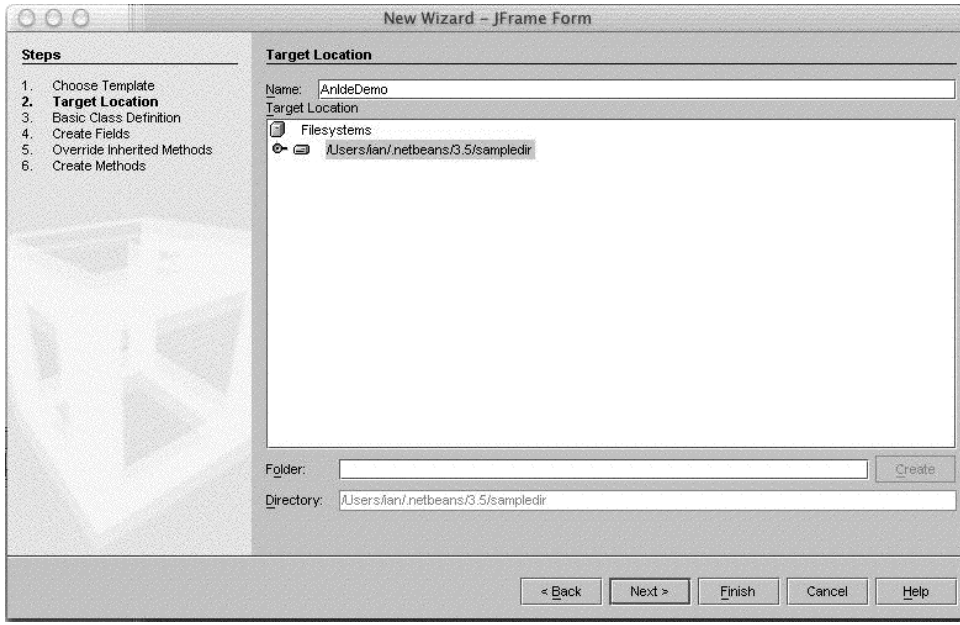
Součástí NetBeans jsou také různé šablony. Na obrázku 1.1 jsem vybral šablonu Swing `JFrame`.



Obrázek 1.1. *NetBeans: Dialog Template z nabídky New*

* Poznámka českého vydavatele: Vývoj produktu NetBeans, jehož autorem je česká společnost stejného jména, stále pokračuje.

Na obrázku 1.2 definuji v NetBeans název třídy a balíku pro nový program, který právě vytvářím.



Obrázek 1.2. *NetBeans: Pojmenování třídy*

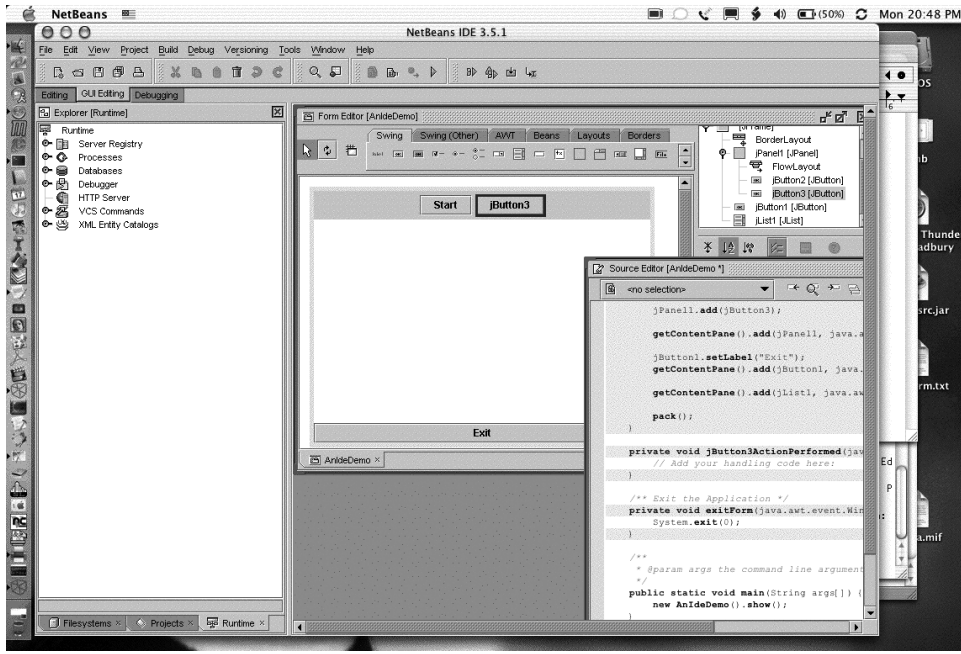
Na obrázku 1.3 vytvářím grafické uživatelské rozhraní (GUI – Graphics User Interface) pomocí sestavovacího nástroje (GUI builder) NetBeans. Vpravo nahoře vyberte vizuální komponentu a ve formuláři klepněte na místo, kam ji chcete umístit. I když v NetBeans existuje několik věcí, které se většině lidí zdají podivné, líbí se mi skutečnost, že standardně používá `BorderLayout`; některá jiná IDE nepoužívají ve výchozím nastavení vůbec layout a výsledné GUI pak nemění svoji velikost dostatečně uhlazeně.

Rovněž se mi líbí způsob, jakým NetBeans zpracovává ovladače událostí grafického uživatelského rozhraní (viz návod 14.4). Jednoduše poklepete na ovládací prvek GUI, jehož události chcete ošetřit, a NetBeans pro něj vytvoří ovladač události a nasměruje vás do editoru, kde zapíšete kód ovladače události. Abychom zjistili, co se stane, úmyslně jsem udělal v tomto případě překlep; když klepnu na položku nabídky Build Project, jasně červenou barvou se zvýrazní řádek obsahující chybu, a to jak ve zdrojovém kódu, tak v chybovém výpisu překladače (viz obrázek 1.4).

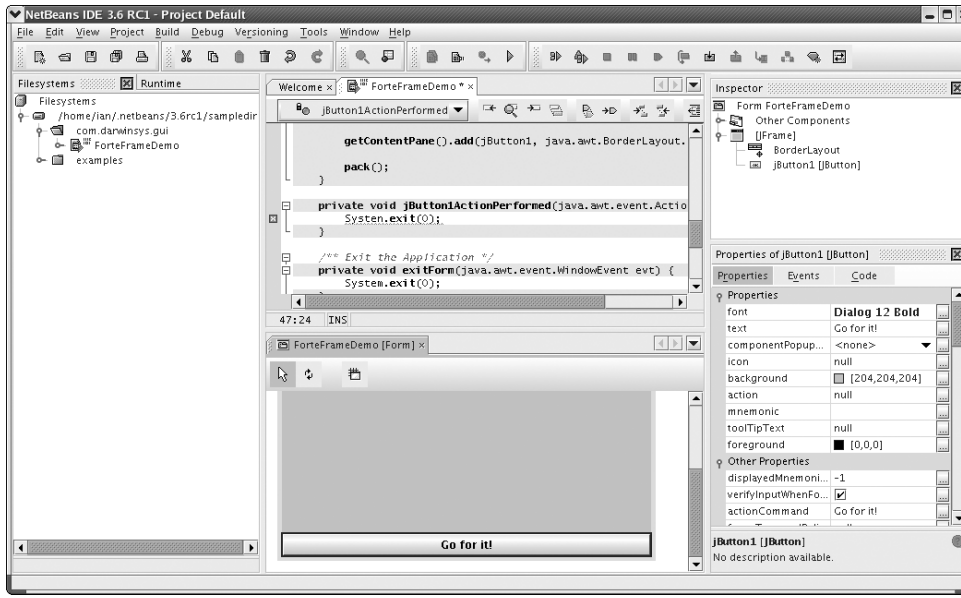
Dalším populárním multiplatformním open-source IDE pro Javu je Eclipse pocházející z dílny IBM. Stejně jako je Sun Studio postaveno na NetBeans, Eclipse se stalo základem pro WebSphere Studio Application Developer od IBM*. Eclipse má tendenci nabízet více voleb než NetBeans; podívejte se například na průvodce New Java Class, jenž je znázorněn na obrázku 1.5. Eclipse rovněž obsahuje celou řadu prostředků pro refaktorování**, viz obrázek 1.6.

* Ve své nápovědě online je WebSphere Studio Application Developer popsáno jako „implementace Eclipse od IBM“ a uvádí odkaz na webové stránky *http://www.eclipse.org/*.

** Poznámka českého vydavatele: Prostředky pro refaktorování obsahují od verze 4.0 i NetBeans.

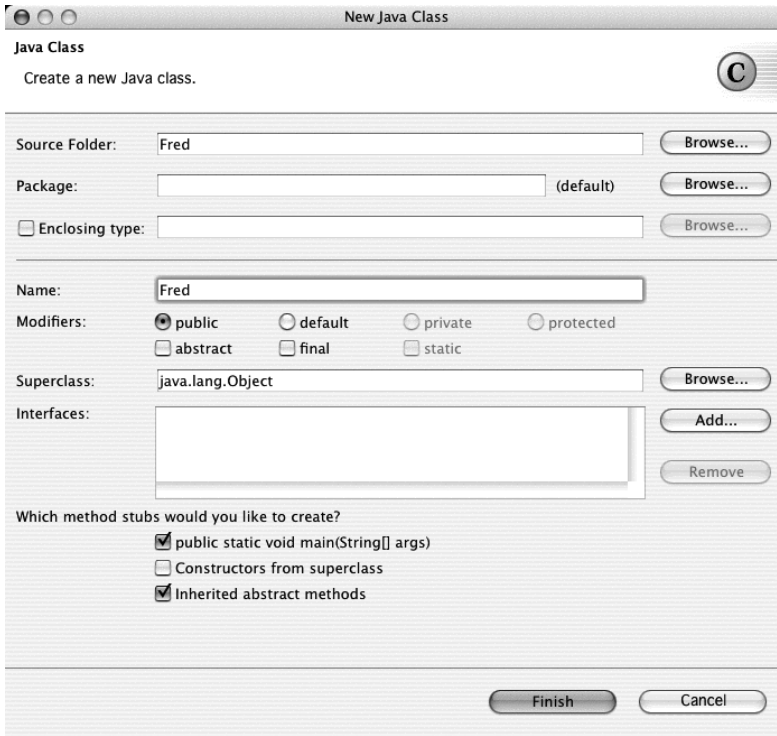


Obrázek 1.3. *NetBeans: sestavování GUI*

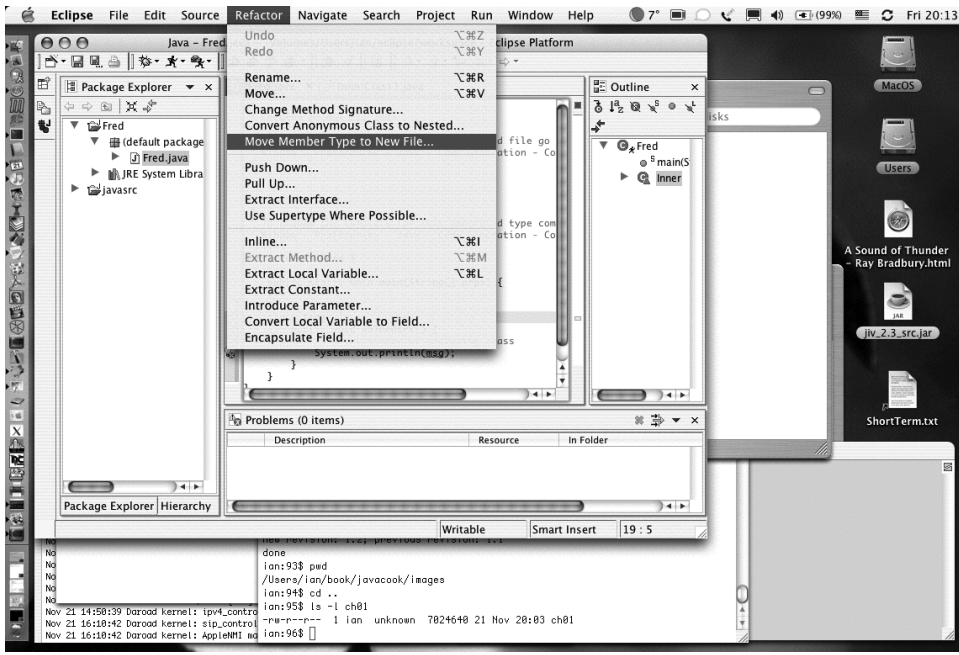


Obrázek 1.4. *NetBeams: Zvýrazněná chyba překladu*

Z těchto dvou hlavních open-source IDE si mnoho vývojářů oblíbilo nástroj NetBeans a jiní zase Eclipse. K dispozici je celá řada dalších IDE zejména pro Windows a téměř každý, kdo



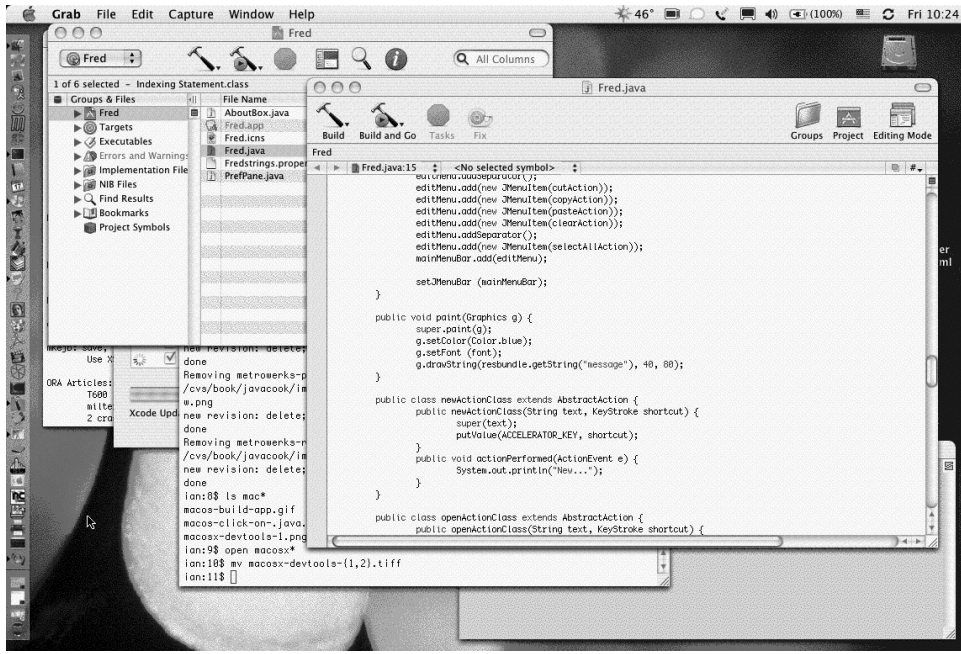
Obrázek 1.5. *Eclipse: Průvodce New Java Class*



Obrázek 1.6. *Eclipse: Refaktorování*

nějaké IDE používá, má svého favorita – například NetBeans, Eclipse, WebGain Visual Cafe, Sun Studio nebo IBM WebSphere Studio Application Developer. Většina z nich má volně dostupnou verzi anebo zkušební verzi či profesionální verzi. Jelikož IDE se aktualizují relativně často, možná se budete chtít podívat do odborných časopisů na aktuální srovnání.

Součástí Mac OS X je Developer Tools od Apple. Ve verzi 10.3 je hlavním IDE program Xcode (znázorněný na obrázku 1.7). IDE od Apple, na rozdíl od většiny ostatních, neobsahuje nástroj pro vizuální sestavování GUI (takzvaný „GUI builder"); tuto úlohu zajišťuje samostatný program s názvem Interface Builder. Oba nástroje lze použít s různými programovacími jazyky včetně C/C++, Objective C (objektové rozšíření jazyka C) a Javy. I když Interface Builder patří k nejpěknějším nástrojům na sestavování GUI, v současné době vytvoří pouze aplikace Cocoa, nikoli aplikace Swing. Na obrázku 1-8 je znázorněno, jak Xcode spouští triviální aplikaci sestavenou prostřednictvím výchozí šablony založené na oknech.



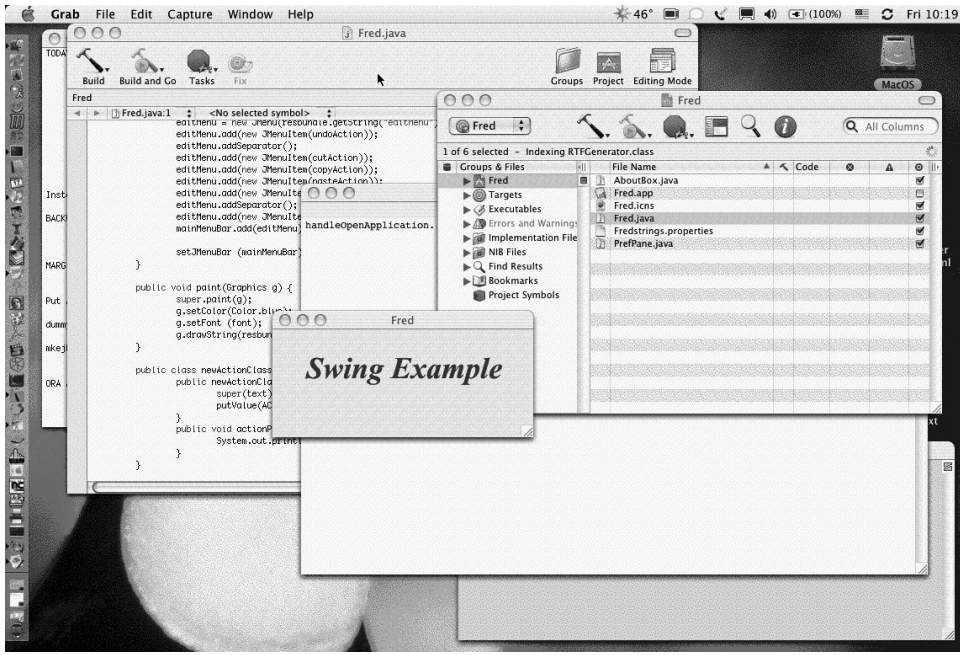
Obrázek 1.7. *Xcode (Mac OS X): Hlavní okna*

Jak si vybrat IDE? Je-li možné stáhnout všechna hlavní IDE (Eclipse, NetBeans) případně zdarma, ale bez zdrojového kódu, nebo alespoň zkušební verzi zdarma, měli byste si jich několik vyzkoušet a zjistit, které nejlépe vyhovuje vašemu způsobu vývoje aplikací. Nezáleží na tom, jakou platformu používáte k vyvíjení kódu Javy, pokud máte runtime Javy, měli byste mít na výběr spoustu IDE.

Přečtěte si také...

Další informace o NetBeans najdete v knize *NetBeans: The Definitive Guide* od Tima Boudreaua, Jesse Glicka, Simeona Greenea, Vaughna Spurlina a Jacka J. Woehreta (vydavatelství O'Reilly). Informace k Eclipse získáte v knize *Eclipse Cookbook* od Steva Holznera (O'Reilly) nebo v knize *The Java Developer's Guide to Eclipse*, kterou napsali Sherry Shavor, Jim D'Anjou,

Scott Fairbrother, Dan Kehn, John Kellerman a Pat McCarthy (Addison Wesley). Obě IDE jsou rozšiřitelná; zajímáte-li se o rozšíření Eclipse, věhlasný teoretik OO Erich Gamma (hlavní autor knihy *Design Patterns*) a Kent Beck (autor titulu *Extreme Programming Explained*) napsali knihu *Contributing to Eclipse: Principles, Patterns, and Plugins* (Addison Wesley).



Obrázek 1.8. *Xcode IDE (Mac OS X): Sestavení a spuštění aplikace*

1.4 Efektivní používání CLASSPATH

Problém

Potřebujete uchovávat své class soubory ve společném adresáři nebo zápolíte s proměnnou CLASSPATH.

Řešení

Natavte proměnnou CLASSPATH na seznam adresářů anebo souborů JAR, které obsahují požadované třídy.

Diskuse

CLASSPATH patří k „nejzajímavějším“ aspektům práce s jazykem Java. Své soubory class můžete uchovávat v libovolném počtu adresářů, souborů JAR nebo souborů zip. Stejně jako slouží proměnná PATH vašeho systému k hledání programů, tak runtime Javy používá proměnnou CLASSPATH k hledání tříd. I když zadáte něco tak jednoduchého jako *java HelloWorld*, interpret Javy hledá ve všech umístěních uvedených ve vaší CLASSPATH, dokud nenajde odpovídající záznam. Ukažme si to na příkladu.

V systémech, které ji podporují (Unix včetně Mac OS X a Windows), lze nastavit CLASSPATH jako proměnnou prostředí. Nastavíte ji stejným způsobem jako jiné proměnné prostředí – například PATH.

Případně můžete pro daný příkaz definovat CLASSPATH ve svém příkazovém řádku:

```
java -classpath \c:\ian\classes MyProg
```

Dejme tomu, že jste svou CLASSPATH nastavili na *C:\classes;.* ve Windows nebo *~/classes:.* na Unixu (v systémech Mac můžete nastavit CLASSPATH pomocí JBindery). A že jste právě přeložili soubor s názvem *HelloWorld.java* na *HelloWorld.class* a zkusili ho spustit. Spustíte-li na Unixu jeden z nástrojů pro sledování jádra (trace, strace, truss, ktrace), pravděpodobně uvidíte, jak program provede open (nebo stat či access) následujících souborů:

- Některých souborů v adresáři JDK
- Následně souboru *~/classes/HelloWorld.class*, který pravděpodobně nebude nalezen
- A souboru *./HelloWorld.class*, který by se našel, otevřel a načetl do paměti

Neurčité tvrzení „nějaké soubory v adresáři JDK" je závislé na verzi. V JDK od Sunu lze ve vlastnostech systému najít:

```
sun.boot.class.path = C:\JDK1.4\JRE\lib\rt.jar;C:\JDK1.4\JRE\lib\i18n.jar;C:\JDK1.4\
JRE\classes
```

Soubor *rt.jar* obsahuje běhový (runtime) materiál; *i18n.jar* je internacionalizace (podpora národních prostředí); a *classes* představuje volitelný adresář, kam můžete umístit další třídy (taková knihovna).

Dejme tomu, že jste si rovněž nainstalovali soubor JAR obsahující podpůrné třídy pro programy v této knize – *darwinsys_cz.jar*. Poté byste mohli nastavit svou proměnnou CLASSPATH na *C:\classes;C:\classes\darwinsys_cz.jar* ve Windows nebo na *~/classes;~/classes\darwinsys_cz.jar* na Unixu. Všimněte si, že soubor JAR musíte na rozdíl od samostatného class souboru explicitně uvést. K jeho zpřístupnění nestačí umístit souboru JAR do adresáře uvedeného ve vaší CLASSPATH .

Všimněte si, že určité specializované programy (například Java Servlety běžící na webovém serveru) nemusí používat ani bootovací cestu, ani CLASSPATH; tyto aplikační servery totiž typicky poskytují vlastní ClassLoader (podrobnější informace o nahrávání tříd najdete v návodu 25.4).

Dalším užitečným nástrojem v JDK je *javap*, který ve výchozím nastavení vypíše externí obraz class souboru, tj. celý jeho název, jeho veřejné metody a pole atd. Kdybyste při sledování jádra spustili příkaz, například *javap HelloWorld*, zjistíte, že se otevřel, rozhlédl se kolem a četl ze souboru *\jdk\lib\tools.jar* a poté by se jako předtím vrátil k hledání vaší třídy HelloWorld. Ve vaší CLASSPATH ještě není záznam pro umístění této třídy. Co se stane? V takovém případě si *javap* interně nastaví svou CLASSPATH tak, aby obsahovala soubor *tools.jar*. Může-li to udělat *javap*, proč nemůžete vy? Můžete, ale není to tak jednoduché, jak byste možná předpokládali. Vyzkoušíte-li oblíbený první pokus, kterým je zavolání metody setProperty s žádostí o nastavení proměnné java.class.path na sebe plus oddělovač a *jdk/lib/tools.jar*, nebudete schopni najít třídu JavaP (sun.tools.java.JavaP); CLASSPATH se totiž na začátku ještě před spuštěním vašeho program nastaví na java.class.path. Můžete si to vyzkoušet ručně a uvidíte, že to funguje, pokud ji nastavíte napřed:

```
>java -classpath /jdk1.4/lib/tools.jar sun.tools.javap.JavaP
Usage: javap <volby> <třídy>...
```

Potřebujete-li nastavit CLASSPATH v aplikaci, můžete nastavit cestu buď v inicializačním skriptu, jak jsme to udělali zde, nebo napsat kód pro spuštění Javy v jazyku C, což si popíšeme v návodu 26.6.

Jak můžete jednoduše ukládat class soubory do adresáře ve vaší CLASSPATH? Příkaz *javac* má volbu -d (destination), která udává, kam by měl směřovat výstup překladače. Pro vložení class souboru HelloWorld do mého adresáře /tridy pomocí -d jsem například zadal:

```
javac -d /tridy HelloWorld.java
```

Dokud bude tento adresář uveden v mé proměnné CLASSPATH, mohu class soubor zpřístupňovat bez ohledu na svůj aktuální adresář. To je jedna z klíčových výhod používání proměnné CLASSPATH.

Správa CLASSPATH však může být ošemetná, zejména když používáte více virtuálních strojů (jako já) nebo když máte více adresářů, v kterých se hledají soubory JAR. Možná budete chtít k ovládání použít nějaký druh dávkového souboru nebo shell-skriptu. Podívejte se na část skriptu, který používám. Byl napsán pro shell Kron na Unixu, ale podobné skripty lze napsat v shellu C nebo dávkovém souboru Dosu.

```
# V naší proměnné classpath musí být následující....
export CLASSPATH=/home/ian/classes/darwinsys.jar:
# Nyní smyčka for testující .jar/.zip nebo [-d ...]
OPT_JARS="$HOME/tridy $HOME/tridy/*.jar
${JAVAHOME}/jre/lib/ext/*.jar
/usr/local/antlr-2.6.0"
for thing in $OPT_JARS
do
      if [ -f $thing ]; then           // Musí být buďto soubor,
            CLASSPATH="$CLASSPATH:$thing"
      else if [ -d $thing ]; then      //nebo adresář
      fi
done
CLASSPATH="$CLASSPATH:."
```

Skript sestaví ze souboru *darwinsys.jar* minimální CLASSPATH, poté projde seznam dalších souborů a adresářů, aby zkontroloval, že se všechny nachází v tomto systému (skript používám na několika počítačích v síti), a ukončí se vložením tečky (.) na konec CLASSPATH.

1.5 Použití tříd API cz.darwinsys z této knihy

Problém

Chcete vyzkoušet můj kód příkladů anebo použít mé pomocné třídy.

Řešení

Ve svém vlastním API jsem vytvořil docela rozsáhlou kolekci opětovně použitelných tříd, které využívám ve vlastních projektech v Javě. V celé knize používám kód příkladů z tohoto API a v mnoha jiných příkladech pracuji s jeho třídami. Chcete-li si tedy stáhnout a přeložit jednotlivé příklady, měli byste si nejprve stáhnout soubor *darwinsys.jar* a začlenit jej do vaší CLASSPATH. Všimněte si, že pokud budete sestavovat celý můj zdrojový kód (jako v návodu 1.6), můžete stahování přeskočit, protože soubor Ant na nejvyšší úrovni začne sestavením souboru JAR pro toto API.

Diskuse

Balík `com.darwinsys.util` z prvního vydání této knihy jsem rozdělil na zhruba tucet balíků `com.darwinsys` uvedených v tabulce 1.1. Rovněž jsem přidal celou řadu nových tříd; tyto balíky nyní obsahují přibližně 50 tříd a rozhraní. Online dokumentaci si můžete prostudovat na stránkách *http://javacook.darwinsys.com/docs/api*.*

Tabulka 1.1. *Balíky cz.darwinsys*

Název balíku	Popis balíku
`cz.darwinsys.io`	Třídy pro vstupní a výstupní operace využívající základní třídy I/O Javy
`cz.darwinsys.lang`	Třídy pro manipulaci se standardními funkcemi Javy
`cz.darwinsys.swingui`	Třídy pomáhající při sestavování a používání grafických rozhraní Swing
`cz.darwinsys.swingui.layout`	Několik zajímavých implementací rozhraní LayoutManager
`cz.darwinsys.util`	Několik různých pomocných tříd

Celá řada těchto tříd slouží jako příklady v této knize; podívejte se na soubory, jejichž první řádek je:

```
package cz.darwinsys._xx_nnn;
```

Rovněž zjistíte, že spousta příkladů obsahuje importy z balíčků `cz.darwinsys`.

1.6 Překlad zdrojového kódu příkladů z této knihy

Problém

Chcete si vyzkoušet příklady uvedené v knize.

Řešení

Stáhněte si nejnovější archiv zdrojových souborů ke knize, rozbalte jej, upravte soubor *build.properties* a program Ant soubory přeloží (viz návod 1.7).**

Diskuse

Aktuální verzi originálního zdrojového kódu všech příkladů v této knize si můžete stáhnout z webových stránek *http://javacook.darwinsys.com/*. Celý zdrojový kód lze získat v podobě

* Poznámka českého vydavatele: Lokalizované verze programů použité v textu jsme umístili do balíčku cz.darwinsys a jeho podbalíčků. Na našem webu si můžete na adrese *http://knihy.cpress.cz/k1223* (ze sekce *Soubory ke stažení*) stáhnout jak původní anglickou verzi, tak lokalizovanou českou verzi.

** Poznámka českého vydavatele: Pro českou lokalizaci není soubor build.properties připraven. Zdrojové kódy jsou k dispozici na webových stránkách nakladatelství *http://knihy.cpress.cz/k1223* (v sekci *Soubory ke stažení*). Celý zdrojový kód lze získat v podobě jednoho velkého souboru s názvem *javakucharka.zip*, který byste měli rozbalit do prázdného adresáře na příhodné místo, kam chcete zdrojový kód uložit. Pokud se v textu objeví zmínky o skriptu napsaném pro Ant, platí pouze pro originální verzi zdrojových kódů a v české lokalizaci ho nenajdete. Výpisy zdrojových kódů a z příkazového řádku platí pro českou lokalizaci.

jednoho velkého souboru s názvem *javacooksrc.jar*, který byste měli rozbalit do prázdného adresáře na příhodné místo, kam chcete zdrojový kód uložit. Následně byste měli upravit soubor *build.properties*, udávající umístění některých souborů jar. Modifikací souboru *build.properties* a následným spuštěním programu `ant` v tomto adresáři se nejprve vytvoří soubor s názvem *darwinsys.jar,** obsahující třídy z balíčku `com.darwinsys`, o němž jsem hovořil v návodu 1.5 (tento soubor budete pravděpodobně chtít přidat do své proměnné CLASSPATH – viz návod 1.4 – nebo do svého adresáře *JDKHOME/jre/lib/ext*). Pak Ant sestaví takové množství dalších příkladů, jaké mu dovoluje dané nastavení v souboru *build.properties* vaší Javy a vašeho operačního systému. Soubory jsou zhruba uspořádány do adresářů podle kapitol, avšak hodně se překrývají a je zde spousta křížových odkazů.

Používáte-li místo JDK 1.5 starší verze 1.3 nebo 1.4, několik souborů se nepřeloží, nicméně překladač vypíše komentář, že je třeba mít verzi 1.4 nebo 1.5. Příklady „nativního kódu" se nemusí vůbec přeložit. Jinak by se mělo všechno ostatní korektně přeložit.

Nepoužíváte-li Ant, měli byste začít! Pokud však nemůžete nebo nechcete, měli byste přeložit, co potřebujete, jakmile nastavíte svou proměnnou CLASSPATH. Budete potřebovat soubor *darwinsys.jar*, takže byste si jej měli stáhnout. V některých adresářích můžete jednoduše zadat *javac *.java* nebo *jikes *.java*, ale v jiných musíte nastavit svou CLASSPATH ručně; pokud se nějaké potřebné soubory nepřeloží, budete muset nahlédnout do souboru Ant *build.xml*, abyste zjistili, jaké soubory jar jsou potřeba. Jelikož se Ant stal pro vývojáře Javy dominantním sestavovacím nástrojem, neposkytuji již Makefiles.

Rovněž mohou nastat situace, kdy si nebudete chtít stáhnout celý archiv – potřebujete-li rychle pouze část kódu – proto můžete na stejných webových stránkách *http://javacook.darwinsys.com/* kdykoli zpřístupnit zmíněné indexové soubory a výsledný adresář.

Upozornění

Jednou z praktik extrémního programování je průběžné refaktorování – schopnost kdykoliv zlepšovat části kódu. Nebuďte proto překvapeni, nebude-li kód v on-line adresáři zdrojů stejný jako kód vytištěný v knize; zřídkakdy se najde týden, kdy neprovedu nějaké zlepšení v kódu, a výsledky těchto zlepšení docela často publikuji na webu.

1.7 Automatický překlad pomocí nástroje Ant

Problém

Unavuje vás neustále psaní příkazů *javac* a *java*.

Řešení

Překládejte pomocí programu Ant.

Diskuse

Komplikovanost Makefiles vedla k vývoji čistě javového řešení automatizace procesu vytváření programu. Ant patří do kategorie svobodného softwaru; je dostupný v podobně zdrojového kódu nebo spustitelných souborů na webových stránkách organizace Apache Foundation

* Máte-li soubor s názvem *com-darwinsys-util.jar*, tento soubor obsahuje staré API popsané v prvním vydání, které nebude fungovat s příklady v této knize.

věnovaných projektu Jakarta: *http://jakarta.apache.org/ant/*. Podobně jako *make* i Ant používá soubor nebo soubory napsané v XML a definující, co se má provést a jak se to má provést (je-li to nutné). Účelem těchto pravidel je, aby nebyla závislá na platformě, ačkoli samozřejmě můžete napsat návody specifické pro nějakou platformu, je-li to potřeba.

K používání programu Ant musíte vytvořit 10 až 30řádkový soubor vymezující různé volby. Tento soubor by měl mít název *build.xml*; pokud jej nazvete nějak jinak, budete muset při každém spuštění Antu udávat speciální argument příkazového řádku. Na příkladu 1.1. je znázorněn sestavovací skript, který slouží k sestavování souborů v adresáři *starting*. Pojednání o syntaxi XML najdete v návodu 21.0. Prozatím si všimněte, že značka `<!--` zahajuje komentář XML, který sahá až po značku `-->`.

Příklad 1.1. *Soubor pravidel pro Ant (build.xml)*

```
<project name="Příklady Java Cookbook" default="compile" basedir=".">

<!-- Nastavení globálních proměnných pro toto sestaveni-->
<propery name="src" value="."/>
<property name="build" value="build"/>
<!--Specifikace překladače.
Překladač jikes se podporuje, ale vyžaduje zahrnutí rt.jar do classpath. -->
<propery name="build.compiler" value="modern"/>

<target name="init">
      <!-- Vytvoření časové známky>
      <tstamp/>
      <!-- Vytvoření adresáře pro přeložené programy -->
      <mkdir dir="${build}"/>
</target>

<!-- Specifikace, co se má překládat. Tímto se vše sestaví -->
<target name="compile" depends="init">

      <!-- Překlad javového kódu z ${src} do ${build} -->
      <javac srcdir="${src}" destdir="${build}"
            classpath="../darwinsys.jar">
</target>

</project>
```

Jakmile spustíte Ant, vytvoří se při jeho provádění přiměřené množství oznámení:

```
$ ant compile
Buildfile: build.xml
Project base dir set to: /home/ian/javasrc/starting
Executing Target: init
Executing Target: compile
Compiling 19 source files to /home/ian/javasrc/starting/build
Performing a Modern Compile
Copying 22 support files to /home/ian/javasrc/starting/build
Completed in 8 seconds
$
```

Přečtěte si také...

Sloupek „make versus Ant“; *Ant: The Definitive Guide* od Jesse E. Tillyho a Erica M. Burkeho (O'Reilly).

make versus Ant

make je nástroj pro překlad a sestavení programu, který se používá na systémech Unix a při vývoji C/C++. Jak *make*, tak Ant mají své výhody; pokusím se zůstat nestranný, ačkoli připouštím, že jsem používal *make* mnohem déle než Ant.

Makefiles jsou kratší a nemají žádný obsah. *make* nepoužívá jazyk XML, avšak má vlastní jazyk – proto může být daleko stručnější. Provádí se rychleji, protože je napsán v C. Ant je však zase schopen souběžně spouštět celou řadu úloh Javy – například vestavěný překladač, *jar/war/tar/zip* soubory a mnohé další. Spuštění několika překladů Javy v jednom procesu Antu může být tedy efektivnější než spuštění stejných překladů pomocí `make`. To znamená, že jakmile běží virtuální stroj JVM, který spouští samotný Ant, spuštění překladače Javy a přeložené třídy vůbec netrvá dlouho. To je Java, jaká by měla být!

Ant však dokáže mnohem víc. Například úloha *javac* v Antu automaticky najde v podadresářích všechny soubory **.java*. U *make se běžně* vyžaduje sub-*make*. Direktiva `include` pro podadresáře není navíc stejná pro GNU *make* a BSD *make*.

Ant má speciální znalosti proměnné CLASSPATH, které v době překladu zjednodušují nastavení CLASSPATH různými způsoby. Viz nastavení CLASSPATH v příkladu 1-1. Možná budete muset v rámci ručního spuštění nebo testování své aplikace duplikovat nastavení jinými způsoby – shell-skripty nebo dávkovými soubory.

make se snáze rozšiřuje, ale není snadné vytvořit přenositelná rozšíření. K získání archívu CVS ze vzdáleného webu můžete napsat jednořádkové pravidlo *make*, ale narazíte na nekompatibilitu mezi GNU *make*, BSD *make* atd. Ant nabízí vestavěnou úlohu pro získání archivu z CVS prostřednictvím Antu. Vývojáři, kteří nepracují s jazykem Java, většinou nikdy neslyšeli o Antu; téměř všichni používají *make*. Většina open-source projektů využívá *make* (s výjimkou těch, které jsou napsány v Javě).

Z počátku je *make* jednodušší. Výhody Ant mají větší význam u rozsáhlejších projektů. V opravdu velkých projektech se však doposud používal pouze *make*. Nástroj *make* se například používá pro zdrojový kód telefonního přepínače, který tvoří stovky tisíc zdrojových souborů obsahujících desítky nebo stovky milionů řádků zdrojového kódu. Naproti tomu server Tomacat 4 má asi 340 000 řádků kódu a server JBoss J2EE asi 560 000 řádků. Ant se však využívá stále více, zejména nyní, když nejpoužívanější vývojová prostředí Javy (Eclipse, NetBeans a další) mají rozhraní pro Ant. Prakticky všechny open-source projekty v Javě používají Ant.

make je součástí většiny systémů Unix a systémů založených na Unixu a dodává se s celou řadou IDE pro Windows. Ant není součástí žádného operačního systému, ale je obsažen v celé řadě open-source balíčků Javy.

Abychom si to zrekapitulovali, ačkoli *make* i Ant jsou dobré nástroje, v nových projektech psaných v Javě bychom měli používat Ant.

1.8 Spouštění apletů

Problém

Chcete spustit aplet.

Řešení

Vytvořte třídu, která rozšiřuje třídu `java.applet.Applet`; napište nějaký kód HTML a nasměrujte na něj prohlížeč.

Diskuse

Aplet je třída, které je potomkem třídy `java.applet.Applet`, od níž dědí funkcionalitu potřebnou pro své zobrazení uvnitř webové stránky ve webovém prohlížeči, který podporuje Javu.* K tomu všemu je nezbytné, aby se stránka HTML odkazovala na aplet. Tato stránka HTML musí obsahovat značku `applet` alespoň se třemi parametry s příslušnými hodnotami: název samotného apletu a jeho šířku a výšku na obrazovce v obrazových bodech neboli pixelech. Nebudu se zde zabývat výukou syntaxe HTML (něco o tom najdete v návodu 18.1), pouze vám ukážu svoji šablonu souboru HTML používajícího aplet. Podobnou stránku vytvoří celá řada IDE (a navíc vám k tomu nabídnou průvodce):

```
<html>
 <head><title>Ukázka</title></head>
 <body>
  <h1>Šablona apletu</h1>
  <applet code="CCC" width="200" height="200">
  </applet>
 </body>
</html>
```

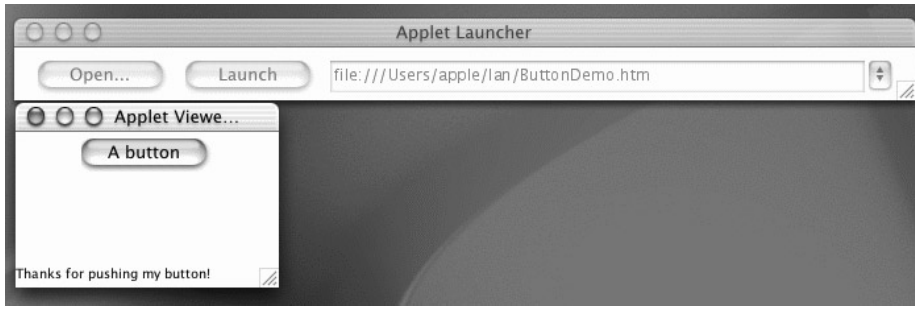
Z této šablony můžete pravděpodobně vytušit, co všechno budete na počátku potřebovat. Podrobnější informace najdete v návodu 18.1. Jakmile vytvoříte uvedený soubor (nahrazením *CCC* za plně kvalifikovaný název třídy svého apletu – např. `code="cz.foo.MujApplet"`) a umístíte ho do adresáře, v kterém je adresář *cz*, stačí pouze sdělit webovému prohlížeči podporujícmu Javu, aby zobrazil stránku HTML, a aplet by se do ní měl vložit.

Dobrá, takže aplet se zobrazil a dokonce skoro fungoval. Proveďte změny ve zdrojovém kódu a znovu jej přeložte. Klepněte v prohlížeči na tlačítko **Obnovit**. Je totiž možné, že stále spouštíte starou verzi! Prohlížeče nejsou příliš vhodné k ladění apletů. Někdy se tomu můžete vyhnout přidržením klávesy **Shift** při klepání na tlačítko **Obnovit**. Pro jistotu však raději používejte `AppletViewer` – miniprohlížeč obsažený v JDK, kterému musíte předat soubor HTML podobně jako normálnímu prohlížeči. AppletViewer od Sunu (zobrazený na obrázku 1.9 v systému Windows) má explicitní tlačítko **Reload** (Obnovit), které skutečně znovu načte aplet. A obsahuje další funkce, například ladění a zobrazení dalších informací. Rovněž nabízí volbu **View – Tag**, díky níž můžete měnit velikost okna, dokud aplet nevypadá co nejlépe, a poté můžete zkopírovat a vložit značku apletu (včetně upravených atributů width a height) do většího dokumentu HTML.

* Mezi prohlížeče podporující Javu patří Netscape/Mozilla, Apple Safari, MS Internet Explorer, KDE Konqueror a spousta dalších.

Obrázek 1.9. *Sun JDK AppletViewer*

Runtime systému Mac OS X obsahuje jak standardní `AppletViewer` od společnosti Sun, tak svou implementaci od Apple (k dispozici jako */Applications/Utilities/Java/Applet Launcher*, viz obrázek 1.10), která je barevnější, ale nepatrně odlišná. Nemá ve svém menu položku Reload (Obnovit); k obnovení zavřete okno apletu a stiskněte tlačítko Launch. Tento prohlížeč vám také dovolí načíst nový soubor HTML zadáním adresy do pole URL (nebo klepnutím na tlačítko Open... a prohledáváním), což je pro změnu HTML souboru efektivnější než zavírání a restartování tradičního AppleVieweru.



Obrázek 1.10. *Aplikace Mac OS X Applet Launcher od Apple*

Ani verze od Sunu, ani verze Apple nepředstavuje úplný apletový runtime; funkce, jako např. přeskočit do nového dokumentu, nefungují. Přesto jsou velmi vhodnými nástroji pro ladění apletů. Naučte se používat AppletViewer, který je součástí vašeho JDK nebo IDE.

Přečtěte si také...

Špatnou zprávou týkající se apletů je, že buď nemohou využívat funkce aktuální verze Javy (prohlížeč ji nemá instalovánu) nebo narazí na obávané problémy nekompatibility prohlížečů. V návodu 23.6 vám ukážeme, jak se těmto problémům vyhnout pomocí instalace zásuvného modulu. V návodu 23.13 budu hovořit o JWS (Java Web Start), relativně nové technice pro distribuci aplikací na webu, která funguje podobným způsobem, jakým se stahují aplety; díky JWS se programy stáhnou pomocí protokolu HTTP, ale uloží a spustí se jako běžné aplikace na místním disku vašeho systému.

1.9 Zpracování varování o zavržení

Problém

Váš kód se dříve bez problémů přeložil, ale nyní generuje varování o používání zavržených metod a/nebo tříd.

Řešení

Musíte přivřít oči. Buď žijte (nebezpečně) s varováními, nebo opravte svůj kód a zavržené metody a třídy nahraďte.

Diskuse

Každé nové vydání Javy obsahuje spoustu nových výkonných metod, ale není to „zadarmo“: během vývoje nového kódu většinou vývojáři najdou i kód, který nebyl vytvořen zcela správně, a neměl by proto být dále používán, protože z prostého důvodu nejde opravit.* Například při sestavování JDK 1.1 vývojáři zjistili, že třída `java.util.Date` měla určité vážné omezení týkající se internacionalizace. Celá řada metod třídy `Date` byla proto označena jako „deprecated“ – zavržené. Podle slovníku *American Heritage Dictionary* je význam slova *deprecate* „vyjádření nesouhlasu; něco odsuzovat“. Vývojáři Javy tedy nedoporučují používat tento kód. Zkuste přeložit následující kód:

```
package cz.darwinsys._01_zaciname;

import java.util.Date;

/** Demonstrace varování deprecated */
public class Deprec {

    public static void main(String[] av) {
        // Vytvoří objekt Date pro datum 5. května 1986.
        // OČEKÁVÁ SE VAROVÁNÍ DEPRECATED
        Date d = new Date(86, 04, 05); // 5. května, 1986
        System.out.println("Datum je " + d);
    }
}
```

Co se stalo? Při překladu kódu jsem obdržel toto varování:

```
>javac cz\darwinsys\_01_zaciname\Deprec.java
Note: cz\darwinsys\_01_zaciname\Deprec.java uses or overrides a deprecated API.
Recompile with "-deprecation" for details.
1 warning
>
```

Takže uposlechněme doporučení překladače a znovu přeložte kód s nastavenou volbou `-deprecation` (používáte-li Ant, zadejte `<javac deprecation='true '...>`), abychom získali podrobné informace:

```
>javac -deprecation cz\darwinsys\_01_zaciname\Deprec.java
cz\darwinsys\_01_zaciname\Deprec.java:10: warnign: constructor Date(int,int,int) in
class java.util.Date has been deprecated
Date d = new Date(86, 04, 05);  // 5. května, 1986
         ^
1 warning

>
```

* Poznámka českého vydavatele: Nejde např. opravit proto, že samotná myšlenka na vytvoření takové metody byla scestná – viz např. zavržené metody ve třídě Thread.

Význam varování je prostý: nedoporučuje se používat konstruktor Date, který přijímá tři argumenty typu int. Jak to opravíte? Odpověď je stejná jako u většiny otázek na použití, nahlédněte do dokumentace dané třídy. V dokumentaci třídy Date je uvedeno:

> Třída Date představuje specifický časový okamžik udávaný s přesností na milisekundy.
>
> Ve verzích před JDK 1.1 měla třída Date dvě další funkce a umožňovala interpretaci data jako hodnoty rok, měsíc, den, hodina, minuta a sekunda. Rovněž umožňovala formátovat a syntakticky analyzovat řetězec data. Bohužel API pro tyto funkce nebylo přizpůsobitelné pro internacionalizaci. Od verze JDK 1.1 by se mělo konvertovat mezi poli data a času pomocí třídy Calendar a k formátování a syntaktické analýze řetězců data by se měla používat třída DateFormat. Použití příslušných metod v Date se nyní zavrhuje.

A v popisu konstruktoru se třemi celočíselnými parametry je konkrétněji uvedeno:

> `Date(int rok, int mesic, int den)`
>
> Zavrženo. Od JDK 1.1 bylo nahrazeno za `Calendar.set(rok + 1900, mesic, den)` nebo `GregorianCalendar(rok + 1900, mesic, den)`.

Jako obecné pravidlo platí, že byste neměli požívat zavržené metody v novém kódu a při údržbě kódu byste se měli snažit varování deprecated eliminovat.

Hlavní oblasti varování deprecated představuje ve standardním API třída `Date` (jak jsme se zmínili), ošetřování událostí JDK 1.0 a nějaké metody – z nichž některé jsou důležité – ve třídě `Thread`.

Zavrhnout můžete i vlastní kód. Stačí do dokumentačního komentáře třídy nebo metody, od jejíhož používání chcete vývojáře odradit, umístit značko *@deprecated* (viz návod 23.2).

1.10 Podmíněné ladění bez #ifdef

Problém

Chcete podmíněný překlad a zdá se, že jej Java neposkytuje.

Řešení

V závislosti na cíli používejte konstanty, argumenty příkazového řádku nebo aserce (návod 1.12).

Diskuse

Některé jazyky (například C, PL/I a C++) nabízí funkci známou jako *podmíněný překlad*. Podmíněný překlad znamená, že v době překladu se na základě určité podmínky musí rozhodnout o začlenění nebo vyloučení části programu. Často slouží k začlenění nebo vyloučení ladicích tiskových příkazů. Když se zdá, že program funguje, vývojáře náhle zasáhne záchvat přílišného sebevědomí a odstraní všechny kontroly chyb. Obvyklejším odůvodněním je, že chce zmenšit velikost výsledného programu – úctyhodný cíl – nebo zrychlit program odstraněním podmíněných příkazů.

Podmíněný překlad?

I když Java postrádá jakoukoli tradiční možnost zadání podmíněného překladu, určitý druh podmíněného překladu nabízí. Všechny překladače Javy musí analyzovat tok programu, aby zajistily, že každé proměnné byla před jejím prvním použitím přiřazena počáteční hodnota, že

metoda, jež má vracet hodnotu, tuto hodnotu doopravdy vrací atd. Představte si, co udělá překladač, když najde příkaz `if`, o jehož hodnotě již v době překladu ví, že není pravdivá. Proč by měl vůbec generovat kód příslušného podmíněného bloku? To je pravda, řeknete si, ale jak může být v době překladu znám výsledek příkazu `if`? Jednoduše: atributy s modifikátory `static final` tuto vlastnost mají. Ví-li překladač, že hodnota podmínky `if` je nepravda, nemusí tělo příkazu `if` vůbec přeložit. Tato vlastnost slouží jako náhražka podmíněného překladu.

```
// IfDef.java
final boolean LADENI = false;
System.out.println("Ahoj všichni");
if (LADENI) {
       System.out.println("Život není cíl, ale cesta");
}
```

Překlad tohoto programu a prozkoumání výsledného class souboru odhalí, že řetězec „Ahoj" se zobrazí, ale nezobrazí se podmíněně tištěný epigram. Ze souboru class byl vypuštěn celý `println`. Java má tedy vlastní mechanismus podmíněného překladu.

```
$ jr IfDef
  jikes +E IfDef.java
  java IfDef
Ahoj světe
$ strings IfDef.class | grep cesta # nenalezeno!
$ javac IfDef.java # zkuste jiný překladač
$ strings IfDef.class | grep cesta # stále nenalezeno!
$
```

Co když budeme chtít použít podobný ladicí kód, avšak tak, abychom mohli zadávat podmínku až při spuštění programu? Pak můžeme hodnotu zadat do systémové proměnné, jejíž hodnotu program přečte pomocí `System.properties` (návod 2.2). V návodu 1.11 se používá moje třída `Debug` jako příklad třídy, jejíž celé chování se ovládá tímto způsobem.

Na tomto místě bych se rád zmínil ještě o další vlastnosti. Jazyky C/C++ mají klíčové slovo `inline`, které dává překladači pokyn, že funkce (metoda) není mimo aktuální zdrojový soubor potřeba. Proto může překladač jazyků C a C++ nahradit volání takovéto funkce vložením jejího těla. Tím eliminuje režii spojenou s vkládáním argumentů do zásobníku, předáváním kontroly, získáváním parametrů a vracením hodnoty. V Javě k tomuto účelu slouží modifikátor `final`. Jím označené metody může překladač v místě jejich použití zavolat anebo místo volání vložit do volající metody jejich tělo. Jedná se však o volitelnou optimalizaci, kterou překladač provádět nemusí, ale může.

Přečtěte si také...

Návod 1.12.

1.11 Ladění výstupů

Problém

Chcete ve svém kódu nechat za běhu zapnuty ladicí příkazy.

Řešení

Použijte moji třídu `Debug`.

Diskuse

Místo používání mechanismu podmíněného překladu z návodu 1.10 možná budete chtít ponechat své ladicí příkazy v kódu, ale aktivovat je pouze tehdy, když se vynoří problém. Tato technika je vhodná pro vše, kromě výpočetně náročných aplikací, protože režie prostého příkazu `if` není tak obrovská. Ukažme si flexibilitu kombinace kontrol za běhu s jednoduchostí příkazu `if` při ladění hypotetické metody `prednes()` (část třídy `Prednes.java`):

```
String hodnota;
String nazev = "báseň";
Prednes p = new Prednes();
if (System.getProperty("ladim.prednes") != null) {
    System.err.println("Predneseno: " + nazev);
}
hodnota = p.prednes(nazev);
```

Poté můžeme kód běžným způsobem přeložit a spustit a za normálních podmínek se ladicí příkaz přeskočí. Spustíme-li ho však s argumentem `-D` pro aktivaci `ladim.prednes`, objeví se kontrolní tisk*:

```
> java Prednes     # Vidíte? Žádný výstup
> java -Dfile.encoding=Cp852 -Dladim.prednes cz.darwinsys._01_zaciname.Prednes
Predneseno: báseň
```

Kdybyste měli psát takovýto podmíněný příkaz častěji, asi by vás to začalo obtěžovat. Proto jsem jej zapouzdřil do třídy `Debug`, která je součástí balíčku `cz.darwinsys.util`. Celý kód třídy `Debug.java` je uveden na konci této kapitoly v návodu 1.17. Třída `Debug` rovněž poskytuje řetězec „debug.“ jako součást argumentu pro metodu `System.getProperty()`, takže předchozí příklad `Prednes` můžeme zjednodušit následovně (kód není součástí doprovodných programů):

```
String nazev = "báseň", hodnota;
Prednes p = new Prednes();
Debug.println("prednes", "Předneseno: " + nazev);
hodnota = p.prednes(nazev);
```

Kód se při spuštění chová stejně jako v původním příkladu `Prednes`:

```
> java cz.darwinsys._01_zaciname.PrednesLadim     # Opět žádný výstup
> java -Dfile.encoding=Cp852 -Ddebug.prednes cz.darwinsys._01_zaciname.PrednesLadim
Předneseno: báseň
>
```

Přečtěte si také...

Určité komplexnější a flexibilnější mechanismy „výstupu ladění“ – včetně výstupů, které lze protokolovat přes síťové připojení – si popíšeme v návodech 17.7, 17.8 a 17.9.

* Poznámka českého vydavatele: Aby fungovala v konzole ve Windows čeština, je nutné nastavit systémovou proměnnou `file.encoding` na CP852.

1.12 Udržování korektnosti programu pomocí asercí

Problém

Chcete ve svém kódu ponechat testy, ale nechcete mít režii kontroly za běhu, dokud ji nepotřebujete.

Řešení

Použijte mechanismus asercí JDK 1.4.

Diskuse

JDK 1.4 zavedl do jazyka nové klíčové slovo: `assert`. Klíčové slovo `assert` přijímá dva argumenty oddělené dvojtečkou (podle analogie s relačním operátorem): výraz, o němž vývojář prohlašuje, že má být pravdivý, a zpráva, která se má zahrnout do výjimky, jež se vyvolá, pokud výraz není pravdivý. V rámci zajištění zpětné kompatibility s programy, které by mohly v předchozích verzích JDK používat „assert" jako název identifikátoru, vyžaduje JDK 1.4 zadat parametr příkazového řádku `-source 1.4`, který zabezpečí, že `assert` bude rozpoznáno jako klíčové slovo. Běžně se plánuje, že aserce zůstanou v kódu (na rozdíl od pomocných tiskových příkazů, které se často vkládají během jednoho testu a poté se odstraní). Vzhledem ke snaze o redukci běhové režie nejsou kontroly asercí standardně aktivovány; musí se výslovně aktivovat pomocí příznaku příkazového řádku `-enableassertions` (nebo `-ea`). Podívejte se na jednoduchý ukázkový program, který znázorňuje použití mechanismu asercí:

```
package cz.darwinsys._01_zaciname;

public class AssertDemo {
    public static void main(String[] args) {
        int i = 4;
        if (args.length == 1) {
            i = Integer.parseInt(args[0]);
        }
        assert i > 0 : "i není kladné číslo";
        System.out.println("Pozdrav za asercí");
    }
}
> javac -source 1.4 AssertDemo.java # bez příznaku 1.4 se nepřeloží*
> java Dfile.encoding=Cp852 cz.darwinsys._01_zaciname.AssertDemo -1
Pozdrav za asercí
>java -ea -Dfile.encoding=Cp852 cz.darwinsys._01_zaciname.AssertDemo -1
Exception in thread "main" java.lang.AssertionError: i není kladné číslo
        at cz.darwinsys._01_zaciname.AssertDemo.main(AssertDemo.java:9)
```

* Poznámka českého vydavatele: Pro verzi JDK 5.0 již není nutné příznak -source uvádět.

1.13 Ladění pomocí JDB

Problém

Použití ladicích výstupů a asercí ve vašem kódu není stále dostatečné.

Řešení

Použijte ladicí nástroj nejlépe takový, který je součástí vašeho prostředí IDE.

Diskuse

JDK obsahuje ladicí nástroj na bázi příkazového řádku – *jdb*. Prakticky všechna IDE nabízí vlastní ladicí nástroje. Zaměřujete-li se na jedno IDE, naučte se pracovat s jeho ladicím nástrojem. Pracujete-li raději s příkazovým řádkem, možná se budete chtít naučit alespoň základní operace s *jdb*.

Následující program záměrně obsahuje celou řadu chyb, abyste mohli vidět jejich účinky v ladicím nástroji:

```
package cz.darwinsys._01_zaciname;

/** Tento program vykazuje nějaké chyby, takže můžeme použít ladicí nástroj. */
public class Buggy {
    static String nazev;

    public static void main(String[] args) {
        int n = nazev.length();    // chyba #1

        System.out.println(n);

        nazev += "; Konec.";       // chyba #2
        System.out.println(nazev); // chyba #3
    }
}
```

Podívejte se na relaci používající *jdb* k nalezení těchto chyb:

```
>java cz.darwinsys._01_zaciname.Buggy
Exception in thread "main" java.lang.NullPointerException
        at cz.darwinsys._01_zaciname.Buggy.main(Buggy.java:8)>jdb
cz.darwinsys._01_zaciname.Buggy
Initializing jdb ...
> run
run cz.darwinsys._01_zaciname.Buggy
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
Exception occurred: java.lang.NullPointerException (uncaught)"thread=main", cz.d
arwinsys._01_zaciname.Buggy.main(), line=8 bci=3
8            int n = nazev.length(); // chyba #1

main[1] list
4    public class Buggy {
5        static String nazev;
6
7        public static void main(String[] args) {
```

```
8 =>         int n = nazev.length();      // chyba #1
9
10            System.out.println(n);
11
12            nazev += "; Konec.";        // chyba #2
13            System.out.println(nazev);  // chyba #3
main[1] print cz.darwinsys._01_zaciname.Buggy.nazev
 cz.darwinsys._01_zaciname.Buggy.nazev = null
main[1] help
** command list **
connectors                -- list available connectors and transports in this VM


run [class [args]]        -- start execution of application's main class

threads [threadgroup]     -- list threads
thread <thread id>        -- set default thread
suspend [thread id(s)]    -- suspend threads (default: all)
resume [thread id(s)]     -- resume threads (default: all)
where [<thread id> | all] -- dump a thread's stack
wherei [<thread id> | all]-- dump a thread's stack, with pc info
up [n frames]             -- move up a thread's stack
down [n frames]           -- move down a thread's stack
kill <thread id> <expr>   -- kill a thread with the given exception object
interrupt <thread id>     -- interrupt a thread

print <expr>              -- print value of expression
dump <expr>               -- print all object information
eval <expr>               -- evaluate expression (same as print)
set <lvalue> = <expr>     -- assign new value to field/variable/array element
locals                    -- print all local variables in current stack frame

classes                   -- list currently known classes
class <class id>          -- show details of named class
methods <class id>        -- list a class's methods
fields <class id>         -- list a class's fields

threadgroups              -- list threadgroups
threadgroup <name>        -- set current threadgroup

stop in <class id>.<method>[(argument_type,...)]
                          -- set a breakpoint in a method
stop at <class id>:<line> -- set a breakpoint at a line
clear <class id>.<method>[(argument_type,...)]
                          -- clear a breakpoint in a method
clear <class id>:<line>   -- clear a breakpoint at a line
clear                     -- list breakpoints
catch [uncaught|caught|all] <class id>|<class pattern>
                          -- break when specified exception occurs
ignore [uncaught|caught|all] <class id>|<class pattern>
                          -- cancel 'catch' for the specified exception
watch [access|all] <class id>.<field name>
                          -- watch access/modifications to a field
unwatch [access|all] <class id>.<field name>
                          -- discontinue watching access/modifications to a fiel
d
trace methods [thread]    -- trace method entry and exit
```

```
untrace methods [thread]  -- stop tracing method entry and exit
step                      -- execute current line
step up                   -- execute until the current method returns to its cal
ler
stepi                     -- execute current instruction
next                      -- step one line (step OVER calls)
cont                      -- continue execution from breakpoint

list [line number|method] -- print source code
use (or sourcepath) [source file path]
                          -- display or change the source path
exclude [<class pattern>, ... | "none"]
                          -- do not report step or method events for specified c
lasses
classpath                 -- print classpath info from target VM

monitor <command>         -- execute command each time the program stops
monitor                   -- list monitors
unmonitor <monitor#>      -- delete a monitor
read <filename>           -- read and execute a command file

lock <expr>               -- print lock info for an object
threadlocks [thread id]   -- print lock info for a thread

pop                       -- pop the stack through and including the current fra
me
reenter                   -- same as pop, but current frame is reentered
redefine <class id> <class file name>
                          -- redefine the code for a class

disablegc <expr>          -- prevent garbage collection of an object
enablegc <expr>           -- permit garbage collection of an object

!!                        -- repeat last command
<n> <command>             -- repeat command n times
help (or ?)               -- list commands
version                   -- print version information
exit (or quit)            -- exit debugger

<class id>: a full class name with package qualifiers
<class pattern>: a class name with a leading or trailing wildcard ('*')
<thread id>: thread number as reported in the 'threads' command
<expr>: a Java(tm) Programming Language expression.
Most common syntax is supported.

Startup commands can be placed in either "jdb.ini" or ".jdbrc"
in user.home or user.dir
main[1] exit
```

K dispozici je celá řada jiných ladicích nástrojů; více se o nich dozvíte v aktuálních časopisech o Javě. Mnohé z těchto nástrojů pracují vzdáleně (tj. ladicí a laděný program máte spuštěné na různých počítačích), protože aplikační rozhraní Java Debugger API, které ladicí nástroje používají, je síťově založené.

1.14 Jednotkové testy: vyhněte se potřebě ladicích nástrojů

Problém

Chcete se vyhnout nutnosti ladit svůj kód.

Řešení

Zkontrolujte všechny třídy při jejich vývoji pomocí jednotkových testů (unit test).

Diskuse

Přestat používat ladicí nástroj je časově náročné: lepší je testovat předem. Metodika *testování jednotek* existuje již delší dobu, ale byla zastíněna novějšími technikami. Testování jednotek je vyzkoušené a skutečně otestuje váš kód v malých blocích. V objektově orientovaných (OO) jazycích, k nimž patří i Java, se jednotkové testování typicky aplikuje na jednotlivé třídy na rozdíl od testování „černé skříňky", při němž se testuje celá aplikace.

Tuto klíčovou metodiku prosazuji již velmi dlouho. Vývojáři softwarové metodiky známé jako extrémní programování (zkráceně XP; viz *http://www.extremeprogramming.org*) prosazují psaní jednotkových testů dokonce ještě *před tím*, než napíšete kód.* Současně prosazují spouštění testů téměř při každém překladu. Tato skupina extrémistů má několik velmi dobře známých vůdců, včetně proslulého autora knihy *Design Patterns* Ericha Gammy a Kenta Becka, autora knihy *Extreme Programming Explained*. S jejich prosazováním jednotkových testů každopádně souhlasím.

Celá řada mých tříd ve skutečnosti obsahuje „vestavěné" jednotkové testy. Třídy, jež samy o sobě nepředstavují hlavní programy, často obsahují metodu `main`, která právě testuje funkcionalitu třídy. Podívejte se na příklad:

```
package cz.darwinsys.demo._01;

/** Jednoduchá třída sloužící k demonstrací jednotkového testování. */
public class Osoba {
    protected String celeJmeno;
    protected String jmeno, prijmeni;

    /** Sestaví objekt třídy Osoba pomocí jména a příjmení.*/
    public Osoba(String jmeno, String prijmeni) {
        this.jmeno = jmeno;
        this.prijmeni = prijmeni;
    }

    /** Získá celé jméno osoby. */
    public String ziskejCeleJmeno() {
        if (celeJmeno != null)
            return celeJmeno;
        return jmeno + " " + prijmeni;
    }
```

* Poznámka českého vydavatele: Metoda, při níž vytváříte nejprve testy, a teprve tak testovaný program, bývá označována jako Test Driven Development (TTD) – vývoj řízený testy.

```
    /** Jednoduchý testovací program. */
    public static void main(String[] argv) {
        Osoba o = new Osoba("Jan", "Dlouhy");
        String c = o.ziskejCeleJmeno();
        if (!c.equals("Jan Dlouhy"))
            throw new IllegalStateException("Porušení řetězení jména");
        System.out.println("Cele jméno " + c + " vypadá dobře");
    }
}
```

Překvapila mě však skutečnost, že předtím, než jsem se setkal s metodikami XP, jsem se domníval, že kontroluji často, ale skutečná inspekce dvou projektů ukázala, že asi pouze třetina mých tříd měla interní nebo externí testy. Nepochybně bylo potřeba použít jednotnou metodiku a tu poskytl program JUnit.

JUnit je knihovna Javy pro přípravu a spouštění testů. JUnit si můžete volně stáhnout z webových stránek *http://www.junit.org*. JUnit je velmi jednoduchý, ale užitečný testovací nástroj. Snadno se používá – pouze napíšete testovací třídu, která má řadu metod, jejichž názvy začínají slovem test. JUnit hledá všechny metody pomocí reflexe (viz kapitola 25) a následně je za vás spustí! Rozšíření JUnit zpracovávají tak odlišné úlohy, jako je testování zátěže a testování komponent EJB (Enterprise JavaBeans); odkazy na tato rozšíření najdete na webových stránkách JUnit.

Jak začít pracovat s programem JUnit? Vše, co k tomu potřebujete, je napsat test. Níže jsem vytáhl test z mé třídy *Osoba* a umístil jsem ho do třídy *OsobaTest*. Všimněte si zřejmého vzoru pro tvorbu názvů.

```
package cz.darwinsys.demo_01;

import junit.framework.*;

/** Jednoduchý test pro třídu Osoba. */
public class OsobaTest extends TestCase
{
    /** Testovací třídy JUnit vyžadují tento konstruktor. */
    public OsobaTest(String jmenoTestu) {
        super(jmenoTestu);
    }

    public void testZiskejCeleJmeno() {
        Osoba o = new Osoba("Jan", "Dlouhy");
        String c = o.ziskejCeleJmeno();
        assertEquals(c, "Jan Dlouhy");
    }
}
```

Ke spuštění testu je potřeba jej pouze přeložit a vyvolat simulační program `junit`*:

```
> javac cz/darwinsys/_01_zaciname/OsobaTest.java
>java junit.textui.TestRunner OsobaTest
.
Time: 0.188

OK (1 tests)
```

* Poznámka českého vydavatele: Aby bylo možné takto spouštět testy z příkazového řádku, je nutné mít v CLASSPATH uvedenu cestu k archivu junit.jar (včetně jména archivu).

Používání úplného názvu třídy je trochu zdlouhavé, takže mám skript s názvem *jtest*, který jej vyvolá; pouze zadám *jtest Osoba* a automaticky se spustí předchozí příkaz.

```
#!/bin/sh

exec java junit.textui.TestRunner ${1}Test
```

Ve skutečnosti je i toto zdlouhavé, takže obvykle ve svých skriptech Ant mám *regresní* cíl. V balíku Antu *Optional Tasks* je úloha *junit*,* jejíž použití je snadné:

```
<target name="regress" depends="build">
      <junit>
            <test name="PersonTest" />
      </junit>
</target>
```

Přečtěte si také...

Dáváte-li přednost zářivějšímu grafickému výstupu, spustí některé varianty JUnit (sestavené pomocí Swing a AWT; viz kapitola 14) testy s grafickým uživatelským rozhraním.

JUnit nabízí třídy pro vytváření komplexních testovacích sad a jeho součástí je rozsáhlá vlastní dokumentace; program si stáhněte z výše zmíněných webových stránek.

Pro testování grafických komponent jsem rovněž vyvinul jednoduchý testovací nástroj, o němž budeme hovořit v návodu 13.2.

Pamatujte: *Testujte zavčas a často!*

1.15 Získejte čitelné zpětné sledování

Problém

Při běhu programu získáváte zásobník sledování výjimek, ale většina nejdůležitějších částí nemá čísla řádků.

Řešení

Ujistěte se, že překládáte se zapnutým laděním. Ve starších systémech vypněte JIT a spusťte je znova nebo použijte technologii HotSpot.

Diskuse

Když program v Javě vyvolá výjimku, výjimka se šíří nahoru zásobníkem návratových adres, dokud se nenajde odpovídající klauzule `catch`. Pokud žádná taková klauzule neexistuje, zachytí výjimku virtuální stroj a vypíše ze zásobníku návratových adres (ZNA) všechna volání metod, která obdržel, od vstupního bodu programu až k místu, kde byla výjimka vyvolána. Toto zpětné sledování si můžete sami vypsat v jakékoli klauzuli `catch`: třída `Throwable` má několik metod označovaných jako `printStackTrace()`.**

Výpis ZNA obsahuje čísla řádků pouze tehdy, pokud byla vložena do přeloženého programu. Překladač `javac` od Sunu to provádí standardně. Používáte-li antovou úlohu `javac`, není to vý-

* Pro některé verze Antu si možná budete muset ještě něco stáhnout, aby tato úloha fungovala.

** Na to, abyste si vypsali ZNA, nemusíte vstupovat do bloku `catch`. Můžete si vytvořit instanci třídy `Throwable`, požádat ji o vypsání zásobníku a pak ji zase zahodit.

chozí nastavení; chcete-li tedy vložit do přeloženého programu čísla řádků, musíte se ujistit, že jste ve svém souboru *build.xml* nastavili `<javac debug="true"...>`.

Při překladu typu JIT (Just-In-Time) zkonvertuje virtuální stroj část vašeho souboru class do strojového jazyka tak, aby mohl běžet maximální rychlostí. To je nezbytný krok k tomu, aby programy v Javě běžely i při interpretaci stále přijatelně rychle. Jednou nevýhodou v počátcích existence Javy bylo, že se při JIT většinou ztrácela čísla řádků. Proto když se váš program ukončil, získali jste ZNA, ale už se nezobrazila čísla řádků, na kterých došlo k chybě. Rychlejší vykonávání programů jsme vyměnili za obtížnější ladění. Moderní verze virtuálního stroje od Sunu však obsahují JIT překladač HotSpot, který netrpí popsanými problémy.

V případě, že stále používáte starší JIT (nebo JIT od jiné společnosti), existuje způsob, jak to obejít. Získává-li program výpis zásobníku a chcete jej udělat čitelným, stačí pouze deaktivovat zpracování JIT. Jak se to udělá, závisí na používané verzi Javy. Ve verzi JDK 1.2 a vyšších stačí pouze pomocí příslušného příkazu *set* nastavit proměnnou prostředí JAVA_COMPILER na hodnotu NONE :

```
C:\ set JAVA_COMPILER=NONE # DOS, Windows
setenv JAVA_COMPILER NONE # Unix Csh
export JAVA_COMPILER=NONE # Unix Ksh, modern sh
```

Chcete-li, aby tato změna byla trvalá, nastavte na svém systému hodnotu v příslušném konfiguračním souboru; v současných verzích Windows můžete také nastavit proměnné prostředí přes Ovládací panely.

Jednodušší způsob, jak dočasně JIT deaktivovat, aniž byste museli měnit nastavení v konfiguračních souborech nebo Ovládacích panelech, představuje volba příkazového řádku `-D`, která aktualizuje vlastnosti systému. Stačí, když v příkazovém řádku nastavíte `java.compiler` na hodnotu NONE:

```
java -Djava.compiler=NONE myapp
```

Všimněte si, že volba příkazového řádku `-D` přehlasuje nastavení proměnné prostředí JAVA_COMPILER.

Jak jsem se zmínil, runtime HotSpot JIT od Sunu, obsažený ve většině moderních verzí Javy, poskytuje zpravidla čísla řádků i při zapnutém režimu JIT.

1.16 Hledání dalších zdrojových kódů v Javě

Problém

Chcete se podívat ještě na další příklady kódu v Javě.

Řešení

Nejlepší poučení skýtají zdrojové kódy.

Diskuse

Zdrojové kódy napsané v Javě jsou všude. Jak jsem se zmínil v předmluvě, veškeré příklady kódů v této knize si lze stáhnout z webu O'Reilly *(http://java.oreilly.com/)*. Co jsem vám neřekl, ale na co byste možná sami přišli, je, že zde jsou také k dispozici zdrojové příklady ke *všem* knihám o Javě, které publikovalo vydavatelství O'Reilly: všechny příklady z *Java Examples in A Nutshell, Java Swing*.

Dalším hodnotným zdrojem je kód aplikačního rozhraní Java API. Možná jste si to neuvědomili, ale zdrojový kód všech veřejných částí Java API je obsažen v každém vydání JDK (Java Development Kit). Chcete zjistit, jak `java.util.ArrayList` ve skutečnosti funguje? Máte zdrojový kód. Máte problémy s chováním `JTable`? JDK od Sunu obsahuje zdrojový kód všech veřejných tříd! Vyhledejte soubor s názvem *src.zip* nebo *src.jar*; některé verze ho rozbalí a jiné zase ne.

Pokud vám to ještě nestačí, můžete zdarma přes Internet získat zdrojový kód celého JDK, pouze odsouhlasíte podmínky licence Sun Java Community Source License a stáhnete si velký soubor. Tento soubor obsahuje zdrojový kód veřejných i neveřejných částí API a také překladače (napsaného v Javě) a velké tělo kódu napsaného v C/C++ (samotný runtime a rozhraní pro nativní knihovnu). Například `java.io.Reader` má metodu s názvem `read()`, která čte bajty dat ze souboru nebo síťového připojení. Tato metoda je napsaná v jazyce C, protože ve skutečnosti volá systémovou metodu `read()`pro Unix, Windows, Mac OS, BeOS nebo pro cokoli jiného. Zdroje sady JDK obsahují zdrojový kód veškerého tohoto materiálu.

A od počátků existence Javy byla zřízena celá řada webových stránek pro distribuci svobodného softwaru nebo open-source Javy, stejně tak jako u většiny jiných moderních „evangelizovaných“ jazyků – například Perlu, Pythonu, Tk/Tcl a dalších. (Pokud ve skutečnosti potřebujete nativní kód pro přenositelné zpracování nějakého podivného mechanismu souborového systému přesahující pojednání v kapitole 11 této knihy, možná by nebylo na škodu se podívat na zdrojový kód těchto runtimových systémů.)

V následujícím výčtu uvádíme několik zajímavých webových stránek s trvalou hodnotou:

- Webové stránky Gamelan jsou v provozu téměř po celou dobu existence Javy. Při poslední kontrole adresa *http://www.gamelan.com* stále fungovala, ale web byl (přirozeně) komercializován a nyní je součástí *http://www.developer.com*.
- Java.Net je společný projekt společnosti Sun a nakladatelství O'Reilly. Najdete zde řadu zajímavých projektů v Javě.
- Giant Java Tree je nejnovějším webem a omezuje se pouze na kód pod licencemi GNU Public License. Tyto stránky obsahují obrovské množství zdrojového kódu, který lze volně stahovat, viz *http://www.git.org*.
- CollabNet open-source tržiště se nezaměřuje přímo na Javu, ale nabízí místo, kde se mohou setkávat lidé, kteří chtějí vytvářet open-souce kód a lidé ochotní financovat jeho vývoj, viz *http://www.collab.net*.
- SourceForge rovněž není vyhrazen přímo pro Javu, ale nabízí zdarma veřejné hostování open-source projektů, viz *http://sourceforge.net*.
- Nakonec autor této knihy spravuje malý web o Javě na adrese *http://www.darwinsys.com/java/*, který může být hodnotný. Obsahuje výpis zdrojů a materiálů v Javě týkajících se této knihy.
- Jako u veškerého svobodného softwaru se prosím ujistěte, že rozumíte rozvětvení různých licenčních schémat. Například kód pod licencí GPL automaticky přenáší licenci GPL na jakýkoli kód, který používá byť jen jeho malou část. A dokonce letmý pohled na podrobnosti implementace Javy od Sunu (licencované stažení zmiňované dříve) vám možná zabrání, abyste někdy pracovali na „čisté“ reimplementaci Javy, svobodného softwaru Kaffe nebo na komerčních implementacích. Poraďte se proto raději s právníkem, můžete mít odlišný názor. Navzdory těmto námitkám je zdrojový kód neocenitelným informačním zdrojem pro člověka, který se chce o Javě něco dozvědět.

1.17 Program: Debug

Většina kapitol v této knize končí podkapitolou *Program*, která předvádí materiál, o němž se v dané kapitole hovořilo. Příklad 1.2 představuje zdrojový kód utility zmiňované v návodu 1.11 – `Debug` .

Příklad 1.2. *Debug.java*

```
package cz.darwinsys.util;

/** Utility pro ladění.
 * @author   Ian Darwin, http://www.darwinsys.com/
 * @version  $Id: Debug.java,v 1.8 2004/01/31 01:26:06 ian Exp $
 */
public class Debug {
    /** Statická metoda pro zjištění,
     * zda je daná kategorie ladění aktivována.
     * Aktivuje se nastavením např.
     * -Ddebug.fileio na ladicí soubor vstupně-výstupních operací.
     * Použijte například:<br/>
     * if (Debug.isEnabled("fileio"))<br/>
     *       System.out.println("Začíná čtení souboru " + nazevSouboru);
     */
    public static boolean isEnabled(String kategorie) {
        return System.getProperty("debug." + kategorie) != null;
    }

    /** Statická metoda pro vypsání dané zprávy, pokud
     * je kategorie aktivována pro ladění.
     */
    public static void println(String kategorie, String zprava) {
        if (isEnabled(kategorie))
            System.out.println(zprava);
    }
    /** Totéž ale pro objekty, které nejsou String (jako optimalizaci
     * zvažte jiný tvar).
     */
    public static void println(String kategorie, Object material) {
        println(kategorie, material.toString());
    }
}
```

Kapitola 2

Komunikace s prostředím

2.0 Úvod

V této kapitole si popíšeme, jak váš program v Javě může jednat se svým bezprostředním okolím, s tím, čemu říkáme *běhové prostředí**. Jeden výklad udává, že všechno, co děláte v programu s téměř jakýmkoli aplikačním rozhraním Javy, se týká prostředí. Intenzivněji se zaměříme na věci, které přímo obklopují program. Přitom narazíme na třídu `System`, která toho hodně ví o vašem konkrétním systému.

Stručnou zmínku si zaslouží další dvě běhové třídy. První z nich je `java.lang.Runtime`, jež se skrývá za celou řadou metod ve třídě `System`. Metoda `System.exit()` například volá právě `Runtime.exit()`. Po technické stránce se jedná o část „prostředí“, ale přímo ji používáme pouze ke spouštění jiných programů, což si vysvětlíme v návodu 26.1. Část prostředí rovněž představuje objekt třídy `java.awt.Toolkit`, o němž budeme hovořit v kapitole 13.

2.1 Získání proměnných prostředí

Problém

Chcete zevnitř svého programu v Javě získat hodnoty „proměnných prostředí“.

Řešení

Nedělejte to (JDK 1.4 a starší). Můžete to zkusit, ale buďte opatrní (JDK 5.0).

Diskuse

Sedmá edice Unixu, publikovaná v roce 1979, obsahovala vzrušující novou funkci známou jako *proměnné prostředí*.* Proměnné prostředí jsou na všech moderních unixových systémech (včetně Mac OS X) a ve většině pozdějších systémů příkazového řádku (jako např. podsystému DOS tvořícím základ systému Windows), ale nejsou na některých starších platformách nebo jiných běhových prostředích Javy. Proměnné prostředí běžně slouží k přizpůsobení samostatného běhového prostředí na počítači uživatele, odtud pochází i jejich název. Vezměme si jeden dobře známý příklad, proměnná prostředí PATH v Unixu nebo DOSu vymezuje místo, kde systém hledá spustitelné programy. Přirozeně tedy vyvstává otázka: Jak získám z mého programu v Javě proměnné prostředí?

* Často se pro ně používá také název *systémové proměnné*.

Odpovědí je, že to můžete zjistit v určitých verzích Javy, ale neměli byste to dělat. Java je navržena tak, aby měla přenositelné běhové prostředí. Jako taková by neměla být závislá na funkcích operačního systému, které neexistují na každé jednotlivé platformě, na níž Javu provozujete. Výše jsem se zmínil pouze o několika platformách Javy, které nemají proměnné prostředí.

Verze 1.4 a starší

Tak dobře, když na tom trváte. Vyzkoušíme si tedy použití metody `static` s názvem `getenv()` ve třídě `java.lang.System`. Ale nezapomeňte, že jste mne k tomu přinutili. Nejprve si ukážeme kód. Všechno, co potřebujeme, je malý program uvedený v příkladu 2.1.

Příklad 2.1. *GetEnv.java*

```
package cz.darwinsys.demo._02_prostredi;

public class GetEnv {
    public static void main(String[] argv) {
        System.out.println("System.getenv(\"PATH\") = "+System.getenv("PATH"));
    }
}
```

Pokusme se program přeložit:

```
>javac cz/darwinsys/_02_prostredi/GetEnv.java
Note: cz/darwisys/_02_prostredi/GetEnv.java uses or overrides a deprecated API.
Recompile with -deprecation for details.
```

Toto hlášení je zřídkakdy vítanou zprávou. Uděláme tedy to, co nám udává:

```
>javac -deprecation GetEnv.java
cz/darwisys/_02_prostredi/GetEnv.java:9: Note: The method java.lang.String
getenv(java.lang.String) in class java.lang.System has been deprecated.
System.out.println("System.getenv(\"PATH\") = " + System.getenv("PATH"));
                                                          ^
Note: cz/darwisys/_02_prostredi/GetEnv.java uses or overrides a deprecated API. Please
consult the documentation for a better alternative.
```

Je to však pouze varování, že? Tak co. Zkusme spustit program!

```
>java cz.darwisys._02_prostredi.GetEnv
Exception in thread "main" java.lang.Error: getenv no longer supported, use properties
and -D instead: PATH
        at java.lang.System.getenv(System.java:602)
        at GetEnv.main(GetEnv.java:9)
```

Nuže, metoda getenv fungovala pouze v JDK 1.1 a pak už ne. Možná se domníváte se, že musíme udělat to, co říká chybové hlášení, tj. že se máme raději naučit pracovat s vlastnostmi a s argumentem -D. Ve skutečnosti se tímto budeme zabývat hned v příštím návodu.

Zpět do budoucnosti: 1.5

V JDK 1.5 již není metoda `getenv()` zavržená, ačkoli stále přináší varování, že by se místo ní měly používat systémové vlastnosti (návod 2.2).* Dokonce i v systémech, které ji podporují, se u názvů proměnných prostředí na některých platformách rozlišuje velikost písmen a na jiných nikoli. Pokud však na tom trváte, spusťte podobný program jako výše zmíněný `GetEnv` a získáte následující výstup (nebo něco podobného):

* Poznámka českého vydavatele: JDK 5.0_05, na němž jsme programy testovali, již žádné varovné hlášení nevydávalo.

```
>java  cz.darwisys._02_prostredi.GetEnv
C:\windows\bin;c:\j2sdk1.5\bin;c:\documents and settings\ian\bin
>
```

Verze 1.5 přidala přetíženou verzi metody `System.getenv()` bez argumentů, která vrací *všechny* proměnné prostředí v podobě nemodifikovatelného seznamu `String Map`. Tento seznam můžete procházet a zpřístupnit tak veškerá uživatelská nastavení nebo získat další nastavení prostředí.

Oba tvary `getenv()` vyžadují, abyste měli oprávnění zpřístupnit prostředí, takže většinou nefungují v omezených prostředích, jako jsou aplety.

2.2 Systémové vlastnosti

Problém

Potřebujete získat informace ze systémových vlastností.

Řešení

Použijte metodu `System.getProperty()` nebo `System.getProperties()`.

Diskuse

Co vůbec je *vlastnost*? Vlastnost představuje pouze dvojici název a hodnota uchovávanou v objektu třídy `java.util.Properties`, o kterém budeme podrobně hovořit v návodu 7.7. Takže kdybyste se pro to rozhodli, mohli byste do objektu `Properties` s názvem `ian` ukládat následující vlastnosti*:

```
jmeno=Ian Darwin
oblibeny_nanuk=cherry
oblibena_rockova skupina=Fleetwood Mac
oblibeny_programovaci_jazyk=Java
barva tuzky=zelena
```

Třída `Properties` má několik tvarů svých vyhledávacích metod. Mohli byste například zadat `ian.getProperty("barva tuzky")` a získat řetězec „zelena“. Rovněž můžete poskytovat výchozí hodnotu: `ian.getProperty("barva tuzky","černá")`, a pokud vlastnost nebyla nastavena, získáte výchozí hodnotu „černá“.

Prozatím se zajímáme o třídu `System` a její úlohu jako úložiště konkrétního objektu `Properties`, který ovládá a popisuje runtime Javy. Třída `System` má statický atribut `Properties`, jehož obsah slučuje specifika operačního systému (například os.name obsahuje jeho název), přizpůsobení systému uživateli (java.class.path) a vlastnosti definované v příkazovém řádku (jak si ukážeme za okamžik). V těchto názvech si všimněte použití teček (např. `os.arch`, `os.version`, `java.class.path` a `java.lang.version`), díky nimž se zdá, že mají hierarchickou strukturu, podobně jak je tomu u názvů tříd. Třída `Properties` však neukládá žádné takové relace: každý klíč je pouze řetězec a tečky nemají pro systém žádný zvláštní význam.

* Poznámka českého vydavatele: Vlastnosti i jejich hodnoty jsou v následujícím příkladu uváděny bez diakritiky. Teoreticky sice Java názvy s diakritkou (nebo např. v japonštině) umožňuje, ale tyto znaky převádí na sekvence, které výslednou podobu textu pro člověka poněkud znepřehledňují, takže jsme pro větší přehlednost a srozumitelnost zůstali raději u identifikátorů a textů bez diakritiky.

K výběru jedné vlastnosti poskytované systémem slouží metoda `System.getProperty()`. Chcete-li však vybrat všechny vlastnosti, použijte metodu `System.getProperties()`. Kdybych tedy chtěl zjistit, jestli objekt `System Properties` obsahuje vlastnost s názvem „barva tužky“, mohl bych zadat:

```
String barva = System.getProperty("barva tuzky")
```

Co však tento příkaz vrátí? Java zajisté není dost chytrá na to, aby znala oblíbenou barvu tužky každého jedince? Správně! Prostřednictvím argumentu `-D` však můžeme jednoduše informovat Javu o barvě naší tužky (a případně sdělit cokoli jiného) .

Argument `-D` slouží k předefinování hodnoty v objektu systémových vlastností. Musí mít název, znaménko rovná se a hodnotu, která se analyzuje stejným způsobem jako ve vlastnostech souboru (viz níže). Na příkazovém řádku lze mezi příkaz java a váš název třídy vložit více definic `-D`. Do příkazového řádku Unixu nebo Windows zadejte (vzhledem k mezeře v názvu vlastnosti jsou uvozovky nutné):

```
>java -D"barva tuzky=zelenomodra" cz.darwinsys._02_prostredi.SysPropDemo
```

Při spouštění programu vložte pomocí prostředí IDE název proměnné a hodnotu do příslušného dialogového okna. Program `SysPropDemo` je krátký; jeho podstatu tvoří následující řádek:

```
System.getProperties().list(System.out);
```

Spustíte-li program tímto způsobem, vypíše okolo 50 řádků, které vypadají přibližně takto:

```
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=C:\Program Files\Java\jre1.5.0_04\bin
java.vm.version=1.5.0_04-b05
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=CZ
......
barva tuzky=zelenomodra
java.specification.name=Java Platform API Specification
java.class.version=49.0
..........
```

Program obsahuje také kód k vytažení pouze jedné nebo několika vlastností, takže můžete zadat:

```
$ java cz.darwinsys._02_prostredi.SysPropDemo os.arch
os.arch = x86
```

Což mi připomíná, že je vhodný čas zmínit se o kódu, který je závislý na systému. V návodu 2.3 budeme hovořit o kódu závislém na verzi a v návodu 2.4 o kódu závislém na operačním systému.

Přečtěte si také...

Podrobnější informace o použití a pojmenovávání vašich souborů `Properties` najdete v návodu 7.7. Přesná pravidla užívaná v metodě `load()` a také další podrobné informace jsou uvedeny v dokumentaci pro `java.util.Properties`.

2.3 Psaní kódu závislého na verzi JDK

Problém

Musíte vytvořit kód, který je závislý na verzi JDK.

Řešení

Nedělejte to.

Diskuse

Ačkoli záměrem Javy je, aby byla přenositelná, její virtuální stroje mají určité podstatné odchylky. Někdy potřebujete pracovat s funkcemi, které nemusí být dostupné ve starších verzích, ale v případě, že jsou, chcete je použít. Takže jako jednu z prvních věcí budete chtít vědět, jak zjistit verzi JDK odpovídajícího virtuálního stroje. Tyto informace získáte pomocí metody `System.getProperty()`:

```
System.out.println(System.getProperty("java.specification.version"));
```

Obdobně můžete chtít otestovat přítomnost nebo absenci určitých tříd. Jeden ze způsobů, jak toho dosáhnout, spočívá v použití metody `Class.forName("třída")`, která při neúspěchu (tj. třídu se nepodaří zavést) vyvolá výjimku – dobré znamení, že knihovna třídu neobsahuje. Tento princip je znázorněn v následujícím kódu z aplikace, která chce zjistit, zda je k dispozici JDK 1.1 nebo pozdější komponenty. Spoléhá na skutečnost, že třída `java.lang.reflect.Constructor` byla doplněna ve verzi 1.1, ale ve verzi 1.0 ještě neexistovala. (Dokumentace standardních tříd uvádí níž pod hlavičkou „Since“ verzi, v níž byla daná třída, atribut či metoda přidána do standardu Javy. Není-li zde žádná taková hlavička, obyčejně to znamená, že třída existovala od začátku – to znamená od verze JDK 1.0.)

```
package cz.darwinsys.demo._02_prostredi;

/** Test na JDK >= 1.1 */
public class TestJDK11 {
    public static void main(String[] a) {
        // Kontroluje JDK >= 1.1
        try {
            Class.forName("java.lang.reflect.Constructor");
        } catch (ClassNotFoundException v) {
            String chyba =
                "Je nám líto, ale tato verze MyApp vyžaduje \n" +
                "Java Runtime založený na Java JDK 1.1 nebo novějších";
            System.err.println(chyba);
            throw new IllegalArgumentException(chyba);
        }
        System.out.println(
            "S potěšením oznamujeme, že se jedná o JDK 1.1 nebo vyšší");
        // Zbytek programu by byl zde
        return;
    }
}
```

Chcete-li ověřit, zda běhová knihovna obsahuje komponenty Swing s jejich konečnými názvy,* mohli byste použít:

```
Class.forName("javax.swing.JButton");
```

Je důležité rozlišovat testování existence v době překladu a při běhu. V obou případech musí být tento kód přeložen na systému obsahujícím třídy, jejichž přítomnost se testuje – JDK 1.1 a Swing. Tyto testy představují pouze snahu pomoci neprogresivním uživatelům, kteří provozují staré verze Javy, spustit vaši moderní aplikaci. Cílem je takovému uživateli poskytnout smysluplnější zprávu než pouze chybu „class not found", kterou předává JVM. Rovněž je důležité si všimnout, že tento test se stane nedosažitelným, pokud jej napíšete uvnitř jakéhokoli kódu, který závisí na kódu, jehož přítomnost testujete. Kontrola přítomnosti knihovny Swing v systému JDK 1.0 se nikdy nespustí, vytvoříte-li ji v konstruktoru podtřídy `JPanel` (popřemýšlejte o tom).** Umístěte test někde na začátek hlavního toku své aplikace dříve, než se sestaví objekty grafického uživatelského rozhraní. Jinak kód na novějších runtimech pouze plýtvá místem a v systémech Java 1.0 se nikdy nespustí.

Zajímá-li vás, co skutečně provádí třída `Class`, vyčkejte si do kapitoly 25.

2.4 Psaní kódu závislého na operačním systému

Problém

Potřebujete vytvořit kód, který závisí na operačním systému.

Řešení

I toho se vyvarujte. Musíte-li to však udělat, použijte `System.properties`.

Diskuse

I když je Java navržena tak, aby byla přenositelná, některé věci bohužel nejsou. A právě k nim patří proměnné jako oddělovač názvu souboru. Každý, kdo používá Unix, ví, že oddělovačem názvu souboru je lomítko (/) a že zpětné lomítko (\) představuje potlačující znak neboli escape znak. V Microsoftu pracovala skupina lidí na Unixu již koncem sedmdesátých let – jejich verze nesla označení Xenix (později ji převzala SCO) – a lidé, kteří pracovali na DOSu, viděli model systému souborů Unixu a líbil se jim. MS-DOS 2.0 neměl adresáře, obsahoval pouze „uživatelská čísla" jako systém CP/M, jehož byl klonem (samotný CP/M vyvinutý v Digital Research byl zase klonem různých jiných systémů). Takže lidé v Microsoftu se pustili do klonování uspořádání systému souborů Unixu. Bohužel již použili znak lomítka jako oddělovač voleb, pro což Unix používal pomlčku (-); a také oddělovač PATH (:) sloužil jako oddělovač „písmene disku", například `C:` nebo `A:`. Takže nyní máme následující příkazy:

* Pamětníci si určitě vzpomenou, že v předběžných verzích swingu se tato třída jmenovala `com.sun.java.swing.JButton`

** Poznámka českého vydavatele: Nápověda – třídu, na které závisíme, natahuje class loader vždy před námi, takže v případě její neexistence zhavaruje dřív, než se dostaneme k tomu, abychom její existenci otestovali.

Systém	Příkaz pro výpis adresáře	Význam	Příklad nastavení PATH
Unix	*ls-R/*	Rekurzivní výpis /, adresáře na nejvyšší úrovni	*PATH=/bin:/usr/bin*
Dos	*dir/s* \	Adresář s volbou podadresářů (tj. rekurzivní) z \, adresář na nejvyšší úrovni (ale pouze aktuální jednotky)	*PATH=C:\windows; D:\mybin*

Co to pro nás znamená? Budeme-li v Javě generovat názvy souborů, musíme vědět, zda vložit / nebo \, případně nějaký jiný znak. Pro tento problém nabízí Java dvě řešení. První je možné aplikovat, když je alespoň *přípustné* se pohybovat mezi Unixem a systémy od Microsoftu (lze použít buď / nebo *) a kód, který s operačním systémem pracuje, vybere správný tvar. V druhém a obvyklejším řešení Java zpřístupní informace týkající se konkrétní platformy tak, aby byly nezávislé na platformě. Třída `java.io.File` (viz kapitola 11) zpřístupňuje statické proměnné obsahující aktuální znak pro oddělovač souborů (a také oddělovač PATH). Protože třída `File` je závislá na platformě, je rozumné zde tyto informace ukotvit. Proměnné jsou:

Název	Typ	Význam
`Separator`	`static String`	Systémově závislý znak oddělovače názvů souborů, např. / nebo \
`separatorChar`	`static char`	Systémově závislý znak oddělovače názvů souborů, např. / nebo \
`pathSeparator`	`static String`	Systémově závislý znak oddělovače cesty, pro zjednodušení znázorněný jako řetězec.
`pathSeparatorChar`	`static char`	Systémově závislý znak oddělovače cesty.

Oddělovače názvu souboru a cesty jsou obyčejné znaky, ale pro usnadnění jsou rovněž dostupné ve formě řetězce.

Druhým a obecnějším mechanismem je systémový objekt `Properties`, o němž byla zmínka v návodu 2.2. Tento objekt můžete používat k určení operačního systému, na kterém pracujete. Podívejte se na zdrojový kód, který jednoduše vyjmenovává systémové vlastnosti; poučné může být spustit tento skript na několika různých implementacích:

```
package cz.darwinsys.demo._02_prostredi;

import java.util.*;
/**
 * Předvádí systémové vlastnosti.
 */
public class SysPropDemo {
    public static void main(String argv[]) {
        System.out.println("Systemové vlastnosti:");
        Properties p = System.getProperties();
        p.list(System.out);
    }
}
```

* Když připravujete řetězce pro práci ve Windows, nezapomeňte u nich zpětná lomítka zdvojit, protože \ je na většině místech kromě příkazového řádku MS-DOS znak escape: `String rootDir = "C:\\";`.

Některé operační systémy například poskytují mechanismus s názvem „nulové zařízení“, s jehož pomocí lze vyřadit výstup (většinou se používá pro účely časování). Podívejte se na kód, který požaduje systémové vlastnosti pro „os.name“ a na jejich základě vytvoří název, který může sloužit k vyřazování dat. Není-li pro danou platformu žádné nulové zařízení známé, vracíme název *junk*, což znamená, že na takových platformách občas vytvoříme odpadové soubory. Takové soubory jednoduše odstraňte, když na ně narazíte.

```
package cz.darwinsys.demo._02_prostredi;

public class SysDep {
    /** Vrací název zařízení /dev/null na platformách, které jej podporují,
     * nebo v opačném případě řetězec "junk".
     */
    public static String getDevNull() {
        String sys = System.getProperty("os.name");
        if (sys==null)
            return "junk";
        if (sys.startsWith("Windows"))
            return "NUL:";
        return "/dev/null";
    }
}
```

V jednom případě je potřeba zkontrolovat operační systém. Mac OS X má řadu užitečných součástí GUI, které lze sice použít pouze v tomto operačním systému, ale přesto by se měly používat, aby vaše aplikace vypadala více jako „nativní“ aplikace platformy Macintosh. Tento problém si podrobněji rozebereme v návodu 14.16. Stručně řečeno, Apple nás nabádá, abychom k určení, zda aplikace běží na OS X, hledali řetězec `mrj.verion`.

```
boolean isMacOS = System.getProperty ("mrj.verion") != null;
```

2.5 Použití rozšíření nebo dalších zapouzdřených API

Problém

Máte soubor tříd JAR, který chcete použít.

Řešení

Jednoduše zkopírujte soubor JAR do adresáře *JDKHOME/jre/lib/ext/*.

Diskuse

Od svého prvního veřejného vydání v roce 1995 roste aplikační rozhraní Javy mílovými kroky. I když je nyní považováno za dostatečně funkční pro psaní stabilních aplikací, stále přibývá oblastí, na které se aplikuje. Některá specializovaná API mohou vyžadovat více prostředků, než máte na dané platformě k dispozici. Spousta nových API od Sunu je k dispozici v podobě *standardních rozšíření*, což signalizuje umístění jejich balíčků do balíčku `javax`. Třídy v balíčcích `java` a `javax` zpracovává webový prohlížeč z hlediska zabezpečení apletu jako vestavěné třídy. Každé rozšíření se distribuuje v podobě souboru JAR (viz návod 23.4).

Máte-li runtime Javy, který nepodporuje tuto funkci (tj. definici adresáře pro umístění rozšiřujících souborů JAR), budete možná muset vložit každý soubor JAR do své proměnné CLASSPATH, viz návod 1.4.

Jakmile nashromáždíte požadovaná volitelná API v souborech JAR, můžete tyto soubory JAR jednoduše umístit do adresáře spravovaného službou Java Extensions Mechanism (většinou něco jako *\jdk1.4\jre\lib\ext*.) a ušetřit si tak uvádění každého souboru JAR ve své proměnné CLASSPATH a přihlížení, jak CLASSPATH roste, roste a roste. Na rozdíl od celé řady jiných systémových změn nemusíte po přidání nového JAR svůj počítač restartovat, protože se tento adresář prohledává při každém spuštění JVM. Budete-li chtít, aby změnu zpozorovaly dlouhodobé programy (například IDE), budete některé z nich muset restartovat. Nejprve to zkuste a uvidíte, které z nich zaznamenají změnu samy.

2.6 Syntaktická analýza argumentů příkazového řádku

Problém

Potřebujete zpracovat volby z příkazového řádku, a Java k tomu neposkytuje API.

Řešení

Nahlédněte do pole `args` předávaného jako argument metodě `main` nebo použijte moji třídu `GetOpt`.

Diskuse

Vývojáři Unixu se museli tímto problémem zabývat déle než kdokoli jiný a přišli v knihovně jazyka C s funkcí `getopt`.* `getopt` zpracuje vaše argumenty příkazového řádku a vyhledá jednoznakové volby zvýrazněné pomocí pomlček a volitelné argumenty. Například příkaz:

```
sort -n -o vystupni_soubor mujsoubor1 vassoubor2
```

spustí standardní program *sort*. Volba `-n` udává, že záznamy nejsou textové, ale číselné, a volba `-o` vystupni_soubor udává, že výstup se má zapsat do souboru s názvem *vystupni_soubor*. Zbývající slova, *mujsoubor1* a *vassoubor2*, se zpracují jako vstupní soubory, které se mají setřídit. Ve Windows jsou někdy argumenty příkazu zvýrazněny lomítkem (`/`). V našem API používáme tvar Unixu – pomlčku, ale klidně můžete změnit kód tak, aby používal lomítka.

Každý syntaktický analyzátor `GetOpt` je sestaven tak, aby rozpoznal určitou množinu argumentů; to je rozumné, protože daný program má běžně pevně stanovenou množinu argumentů, které přijímá. Můžete sestavit pole objektů `GetOpt`, které představují přípustné argumenty. Pro dříve uvedený třídicí program byste mohli použít:

```
GetOptDesc options[] = {
    new GetOptDesc('n', "číselný", false},
    new GetOptDesc('o', "výstupní soubor", true),
});
Map optionsFound = new GetOpt(options).parseArguments(argv);
if (optionsFound.get("n") != null)
    sortType = NUMERIC;
} else  if (optionsFound.get("o")){
...
```

* Unixový svět má několik variací knihovny getopt; moje verze docela důkladně emuluje původní verzi AT&T s několika vylepšeními, jako například argumenty s dlouhými názvy.

Jednoduchý způsob, jak pracovat s `GetOpt`, spočívá ve vyvolání jeho metody `parseArguments`.

Kvůli zpětné kompatibilitě s lidmi, kteří se učili používat unixovou verzi v C, lze používat metodu `GetOpt()` v cyklu `while`. Cyklus se provede jednou pro každou nalezenou platnou volbu a vrací hodnotu nalezeného znaku nebo konstantu `DONE`, když byly zpracovány všechny volby (jsou-li nějaké).

Podívejte se na celý program, který používá moji třídu `GetOpt` pouze k tomu, aby zjistil, jestli se na příkazovém řádku nachází argument `-h` (pro nápovědu).

```
package cz.darwinsys.demo. 02_prostredi;

import cz.darwinsys.lang.GetOpt;

/** Triviální ukázka GetOpt. Je-li přítomna volba -h, vypíše se nápověda.
 */
public class GetOptSimple {
    public static void main(String[] args) {
        GetOpt go = new GetOpt("h");
        char c;
        while ((c = go.getopt(args)) !=GetOpt.DONE) {
            switch(c) {
                case 'h':
                    helpAndExit(0);
                    break;
                default:
                    System.err.println("Neznámá volba v " +
                        args[go.getOptInd(  )-1]);
                    helpAndExit(1);
            }
        }
        System.out.println(  );
    }

    /** Blok pro poskytování nápovědy k používání
     * Zajisté dokážete k tomu vytvořit delší nápovědu.     */
    static void helpAndExit(int navratHodnota) {
        System.err.println("To by vás informovalo, jak používat tento program");
        System.exit(navratHodnota);
    }
}
```

Následující delší demonstrační program má několik voleb:

```
package cz.darwinsys.demo._02_prostredi;

import cz.darwinsys.lang.GetOpt;
import cz.darwinsys.lang.GetOptDesc;
import java.util.*;

/** Předvádí moderní způsob práce s GetOpt.
 * Program povoluje podmnožinu třídicích voleb
 * <pre>Unix: sort -n -o výstupní_soubor vstupsoubor1 vstupsoubor2</pre>
 * což znamená:
 * - seřadit numericky (-n),
 * - zapsat do souboru "výstupní_soubor" (-o výstupní_soubor),
 * - seřadit ze vstupsoubor1 a vstupsoubor2.
 */
public class GetOptDemoNew {
```

```
    public static void main(String[] argv) {
        boolean cisel_volba = false;
        boolean chyby = false;
        String vystupniNazevSouboru = null;

        GetOptDesc options[] = {
            new GetOptDesc('n', "numericky", false),
                new GetOptDesc('o', "výstupní soubor", true),
        };
        GetOpt analyzator = new GetOpt(options);
        Map optionsFound = analyzator.parseArguments(argv);
        Iterator it = optionsFound.keySet().iterator();
        while (it.hasNext()) {
            String key = (String)it.next();
            char c = key.charAt(0);
            switch (c) {
                case 'n':
                    cisel_volba = true;
                    break;
                case 'o':
                    vystupniNazevSouboru = (String)optionsFound.get(key);
                    break;
                case '?':
                    chyby = true;
                    break;
                default:
                    throw new IllegalStateException(
                        "Neočekávaný znak volby: " + c);
            }
        }
        if (chyby) {
            System.err.println("Použití: GetOptDemo [-n][-o soubor][soubor...]");
        }
        System.out.print("Volby: ");
        System.out.print("Numericky: " + cisel_volba + ' ');
        System.out.print("Výstup: " + vystupniNazevSouboru + "; ");
        System.out.print("Vstupy: ");
        List soubory = analyzator.getFilenameList();
        for (int i = 0; i < soubory.size(); i++) {
            System.out.print(soubory.get(i));
            System.out.print(' ');
        }
        System.out.println();
    }
}
```

Vyvoláme-li program několikrát s různými volbami včetně voleb s jedním argumentem i s dlouhými názvy, bude se chovat následovně:

```
> java -Dfile.encoding=Cp852
 cz.darwinsys._02_prostredi.GetOptDemoNew
Volby: Numericky: false Výstup: null; Vstupy:
> java -Dfile.encoding=Cp852
 cz.darwinsys._02_prostredi.GetOptDemoNew -M
Volby: Numericky: false Výstup: null; Vstupy: -M
> java -Dfile.encoding=Cp852
 cz.darwinsys._02_prostredi.GetOptDemoNew -n a b c
Volby: Numericky: true Výstup: null; Vstupy: a b c
```

```
> java -Dfile.encoding=Cp852
 cz.darwinsys._02_prostredi.GetOptDemoNew -numeric a b c
Volby: Numericky: true Výstup: null; Vstupy: a b c
> java -Dfile.encoding=Cp852
 cz.darwinsys._02_prostredi.GetOptDemoNew -numeric -output-file /tmp/foo a b c
Volby: Numericky: true Výstup: /tmp/foo; Vstupy: a b c
```

Delší příklad vykonávající všechny vstupy a výstupy této verze `GetOpt` lze najít v online zdrojích pod *darwinsys/src/regress*. Zdrojový kód samotné třídy `GetOpt` je uveden v příkladu 2.2.

Příklad 2.2. *Zdrojový kód pro GetOpt*

```
package cz.darwinsys.lang;

import cz.darwinsys.util.Debug;

import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

/** Třída pro implementaci syntaktické analýzy unixových (jednoznakových)
 * argumentů příkazového řádku. Původně vytvořena podle (avšak bez použití jeho kódu)
 * unixového programu getopt(3), který byl přetvořen tak, aby lépe vyhovoval jazyku
Java.
 * <p>
 * Třída <em>nepodporuje</em> zpracování více vláken, čili není vláknově bezpečná
(threadsafe); předpokládá se její použití pouze z metody main(  ).
 * <p>
 * Další způsoby práce s příkazovým řádkem viz
 * <a href="http://jakarta.apache.org/commons/cli/">Jakarta Commons
 * Command Line Interface</a>.
 * @author Ian F. Darwin, ian@darwinsys.com
 * @version $Id: ch02,v 1.4 2004/05/04 20:11:12 ian Exp $
 */
public class GetOpt {
    /** Seznam názvů souborů nalezených za argumenty. */
    protected List argumentyNazvySouboru;
    /** Množina znaků pro hledání. */
    protected GetOptDesc[] volby;
    /** Kde se nacházíme ve volbách. */
    protected int volind = 0;
    /** Veřejná konstanta pro "žádné další volby". */
    public static final int HOTOVO = 0;
    /** Interní příznak - jestli jsme provedli všechny volby. */
    protected boolean hotovo = false;
    /** Aktuální argument volby. */
    protected String volarg;

    /** Vybere aktuální argument volby. */
    public String optarg(  ) {
        return volarg;
    }

    /* V poli objektů GetOptDescSestaví syntaktický analyzátor GetOpt
     * daných specifikací voleb. Jedná se o preferovaný konstruktor.
     */
    public GetOpt(GetOptDesc[] volby) {
```

```
    this.volby = volby;
}

/* Sestaví syntaktický analyzátor GetOpt, který uchovává množinu znaků voleb.
 * Jedná se o původní konstruktor pro zpětnou kompatibilitu.
 */
public GetOpt(String vzor) {
    // První fáze: pouze počítá písmena
    int n = 0;
    for (int i = 0; i<vzor.length(  ); i++) {
        if (vzor.charAt(i) != ':')
            ++n;
    }
    if (n == 0)
        throw new IllegalArgumentException(
            "Žádná písmena voleb nebyla nalezena v " + vzor);

    // Druhá fáze: sestaví pole objektů GetOptDesc.
    volby = new GetOptDesc[n];
    for (int i = 0, ix = 0; i<vzor.length(  ); i++) {
        char c = vzor.charAt(i);
        boolean argPrijimaHodnotu = false;
        if (i < vzor.length(  ) - 1 && vzor.charAt(i+1) == ':') {
            argPrijimaHodnotu = true;
            ++i;
        }
        volby[ix++] = new GetOptDesc(c, null, argPrijimaHodnotu);
        Debug.println("getopt",
            "KONSTR: volby[" + ix + "] = " + c + ", " + argPrijimaHodnotu);
    }
}

/** Resetuje tento syntaktický analyzátor GetOpt. */
public void rewind(  ) {
    argumentyNazvySouboru = null;
    hotovo = false;
    volind = 0;
}

/** Pole, které slouží ke konverzi znaku na řetězec. */
private static char[] strConvArray = { 0 };

/**
 * Moderní způsob použití GetOpt: Jednou se zavolá a získá všechny volby.
 * <p>
 * Analyzuje volby a vrací Map, jehož klíče jsou nalezené volby.
 * Normálně poté následuje volání metody getFilenameList(  ).
 * @return Map, jehož klíče jsou řetězce délky 1 (obsahující znak
 * z volby, která se shodovala) a jehož hodnotou je řetězec
 * obsahující hodnotu nebo prázdnou hodnotu pro argument bez volby.
 */
public Map parseArguments(String[] argv) {
    Map volbyAHodnoty = new HashMap(  );
    argumentyNazvySouboru = new ArrayList(  );
    for (int i = 0; i < argv.length; i++) {
        Debug.println("getopt", "parseArg: i=" + i + ": arg " + argv[i]);
        char c = getopt(argv);
```

```
        if (c != HOTOVO) {
            strConvArray[0] = c;
            volbyAHodnoty.put(new String(strConvArray), volarg);
            // Pokud tento argument přijímá volbu, musíme jej zde přeskočit.
            if (volarg != null)
                ++i;
        } else {
            argumentyNazvySouboru.add(argv[i]);
        }
    }
    return volbyAHodnoty;
}

/** Získá seznam argumentů názvů souborů za volbami. */
public List getFilenameList(  ) {
    if (argumentyNazvySouboru == null) {
        throw new IllegalArgumentException(
            "Nedovolené volání getFilenameList(  ) před parseOptions(  )");
    }
    return argumentyNazvySouboru;
}

/** Skutečným jádrem třídy getopt, ať používá starý nebo nový způsob, je, že:
 * vrací jeden argument; volá se opakovaně, dokud nevrací HOTOVO.
 */
public char getopt(String argv[]) {
    Debug.println("getopt",
        "volind=" + volind + ", argv.length="+argv.length);

    if (volind >= (argv.length)-1) {
        hotovo = true;
    }

    // Pokud jsme (nyní) hotovi, poskytne záruku.
    if (hotovo) {
        return HOTOVO;
    }

    // UDĚLAT - dvě fáze, první zkontroluje dlouhé argumenty, druhá ověří
    // znak, mohou být víceznakové jako v "-no výstupní_soubor" ==
    // "-n -o výstupní_soubor".

    // Vybere další argument příkazového řádku; ověří, jestli začíná na "-".
    // Je-li tomu tak, vyhledá jej v seznamu.
    String tentoArg = argv[volind++];
    if (tentoArg.startsWith("-")) {
        volarg = null;
        for (int i=0; i<volby.length; i++) {
            if ( volby[i].argLetter == tentoArg.charAt(1) ||
                (volby[i].argName != null &&
                 volby[i].argName == tentoArg.substring(1))) { // Nalezen
                // Pokud potřebuje argument volby, dostane jej.
                if (volby[i].takesArgument) {
                    if (volind < argv.length) {
                        volarg = argv[volind];
                        ++volind;
                    } else {
```

```
                            throw new IllegalArgumentException(
                                "Volba " + volby[i].argLetter +
                                "potřebuje hodnotu, ale byl dosažen konec seznamu"
                                + "argumentů");
                        }
                    }
                    return volby[i].argLetter;
                }
            }
            // Začíná na "-", ale neshoduje se, takže to musí být chyba.
            return '?';
        } else {
            // V argumentech nalezeno slovo, které není volba ani argument: konec voleb.
            hotovo = true;
            return HOTOVO;
        }
    }

    /** Vrací volind, index na argumenty poslední volby, na kterou jsme se dívali. */
    public int getOptInd(  ) {
        return volind;
    }

}
```

Přečtěte si také...

`GetOpt` je dostatečný nástroj na zpracování voleb příkazového řádku. Možná přijdete s něčím lepším a přispějete tím do světa Javy; to ponechám jako cvičení pro čtenáře.

Na další způsoby manipulace s argumenty příkazové řádky se podívejte do knihovny CLI (Command Line Interface) Jakarta Commons, kterou lze najít na webových stránkách *http://jakarta.apache.org/commons/cli/*.

Kapitola 3

Řetězce

3.0 Úvod

Nepostradatelnou částí téměř všech programovacích úloh jsou řetězce znaků. Slouží k vypisování zpráv uživateli; k vytváření odkazů na soubory na disku nebo externím médiu; a pro jména lidí, adresy a afilace. Řetězce mají široké, téměř nespočetné použití (pokud ve skutečnosti potřebujete čísla, dostaneme se k nim v kapitole 5).

Přecházíte-li z programovacího jazyka jako C, budete si muset zapamatovat, že v Javě jsou řetězce předdefinovaný typ (třída) `String`. To znamená, že řetězec je objekt, a má tedy metody. Nejedná se o pole znaků, a neměli bychom proto uvažovat o řetězci jako o poli. Běžné jsou operace typu `fileName.endsWith(".gif")` a `extension.equals(".gif")` (a také ekvivalentní `".gif".equals(extension)`).

Pamatujte, že daný objekt `String` je po sestavení neměnný. To znamená, že když jednou zadáme `String s = "Ahoj" + vaseJmeno;` tak právě vytvořený objekt, na který se odkazuje referenční proměnná `s`, nelze nikdy změnit. Proměnné `s` můžete přiřadit odkaz na odlišný řetězec, dokonce i na řetězec odvozený z původního, například `s = s.trim()`. Znaky z původního řetězce můžete vybrat pomocí metody `charAt()`, avšak tato metoda se nejmenuje `getCharAt()`, protože neexistuje a nikdy nebude existovat metoda `setCharAt()`. Řetězec nezmění ani metody jako `toUpperCase()`; tyto metody vždy vrací nový řetězec obsahující převedené znaky. Musíte-li měnit znaky uvnitř řetězce, měli byste raději vytvořit `StringBuilder`* (možná inicializovaný na výchozí hodnotu zpracovávaného řetězce), zpracovat jeho obsah a poté jej na konci převést na `String` pomocí všudypřítomné metody `toString()`.

Jak si můžu být tak jistý, že metoda `setCharAt()` nebude v další verzi doplněna? Protože nezměnitelnost řetězce je jednou ze základních pravidel virtuálního stroje JVM. Pamatujte si, že Java je jazyk, který nebere bezpečnost a souběžné zpracování (vláken) na lehkou váhu. Budete na to myslet? Dobře. Nyní popřemýšlejte o apletech, které jsou chráněny před zpřístupňováním řady místních zdrojů. Zvažte následující scénář: Vlákno A spustí další Vlákno B. Vlákno A vytvoří řetězec s názvem `s`, který obsahuje název souboru, uloží odkaz na řetězec `s` jako `s2` a poté předá `s` nějaké metodě, která vyžaduje oprávnění. Tato metoda bude určitě volat objekt třídy `SecurityManager`** JVM, je-li nějaký nainstalovaný (jakože v prostředí apletu

* `StringBuilder` představuje novinku v JDK 1.5. Jedná se o funkční ekvivalent staršího `StringBuffer`. Podrobně se jím budeme zabývat v návodu 3.3.

** `SecurityManager` je třída, která slouží jako správce zabezpečení, to znamená, že pomocí této třídy se ověří, jestli aktuální aplikace má oprávnění dělat určité věci, například jako otevírat soubory na místním disku, otevírat libovolné síťové připojení atd. Aplety mají více omezujícího správce zabezpečení než normální aplikace.

určitě bude). Následně v rozmezí několika nanosekund mezi časem, kdy `SecurityManager` předá své schválení na pojmenovaný soubor, a časem, kdy se systém I/O skutečně dostane k otevírání souboru, vlákno B změní řetězec odkazovaný pomocí `s2 tak`, aby odkazoval na systémový soubor. Mizera! Kdybyste toto mohli udělat, celá představa o bezpečnosti Javy by byla pouhým vtipem. Avšak vývojáři na to samozřejmě mysleli, takže to udělat nelze. I když můžete na řetězec `s` kdykoli přiřadit nový odkaz `String`, tento nikdy nebude mít žádný účinek na řetězec, na který se `s` odkazuje. Samozřejmě kromě případu, kdy je řetězec `s` jediným odkazem na tento `String`, protože to by byl způsobilý k čištění paměti.

Rovněž si pamatujte, že `String` je v Javě velmi důležitým typem. Na rozdíl od většiny ostatních tříd v jádru API není chování řetězců proměnlivé; třída `String` je konečná (`final`), takže nemůže mít podtřídy. Proto nemůžete deklarovat vlastního potomka třídy `String`. Vývojáři mysleli na to, abyste se nemohli maskovat jako `String`, a přitom poskytovat metodu `setCharAt()`! Pokud mi nevěříte, vyzkoušejte si to:

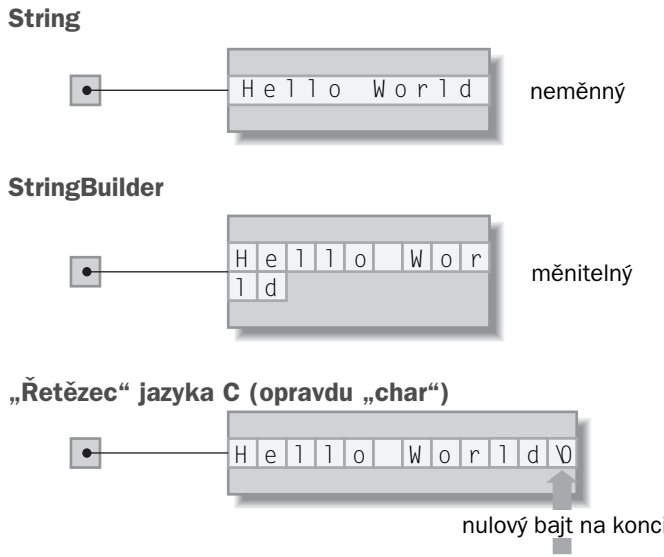
```
package cz.darwinsys.demo._03_retezce;

/** Pokud by bylo možné přeložit třídu s odkomentovanou klauzulí
 * definující odvozenost od třídy java.lang.String,
 * bezpečnost Javy by byla pouhým mýtem.
 */
public class BombaSchovanáVŘetězci
//     extends java.lang.String
{
    public void setCharAt(int index, char novyZnak) {
        // Implementaci této metody ponecháme
        // jako cvičení pro čtenáře.
        // Nápověda: zkompilujte tento kód, jako by byl bez problémů!
    }
}
```

Přesvědčili jste se? Myslete na to!

Samozřejmě potřebujete být schopni modifikovat řetězce. Některé metody extrahují části řetězců; těmito metodami se budeme zabývat v prvních několika návodech této kapitoly. Důležitou množinou tříd, která manipuluje se znaky a řetězci a nabízí řadu metod pro modifikaci obsahu včetně metody `toString()`, je `StringBuilder`. Programátoři přecházející z jazyka C by si měli všimnout, že řetězce Javy nejsou pole znaků jako v C, takže pro operace jako zpracování řetězce po jednom znaku musí používat metody; viz návod 3.4. Na obrázku 3.1 je znázorněno porovnání řetězce `String`, `StringBuilder` a řetězce v jazyku C.

I když jsme zatím podrobně nehovořili o balíku `java.io` (napravíme to v kapitole 10), kvůli některým programům z této kapitoly musíte být schopni číst textové soubory. Dokonce i pokud nejste obeznámeni s balíkem `java.io`, z příkladů, které načítají textové soubory, můžete pravděpodobně zjistit, že `BufferedReader` vám umožní číst „kusy“ dat a že tato třída má velmi příhodnou metodu `readLine()`.

V této kapitole vám neukážu, jak seřadit pole řetězců; obecnější teorií řazení kolekce objektů se budeme zabývat v návodu 7.8.



Obrázek 3.1. *String, StringBuilder a řetězce jazyka C*

3.1 Rozkládání řetězců do podřetězců

Problém

Chcete podle pozice rozdělit řetězec do podřetězců.

Řešení

Použijte metodu `substring()` objektu `String`.

Diskuse

Metoda `substring()` sestaví nový objekt `String` vytvořený ze znaků obsažených někde v původním řetězci, jehož metodu `substring()` jste vyvolali. Název této metody (`substring()`) porušuje stylistické konvence o psaní velkých písmen ve slovech; kdyby byla Java stoprocentně konzistentní, tato metoda by měla název `subString`. Ale není tomu tak – je to `substring`. Metoda `substring` je přetížená: oba tvary vyžadují počáteční index. Metoda s jedním argumentem vrací řetězec od počátečního indexu po konec. Metoda se dvěma argumenty přijímá koncový index (nikoli délku jako v některých jazycích), takže index lze generovat pomocí metod `indexOf()` nebo `lastIndexOf()` třídy `String`. Všimněte si, že hodnota koncového indexu je o 1 větší než pozice posledního znaku!

```
// Soubor SubStringDemo.java
public static void main(String[] av) {
    String a = "Java je skvělá.";
    System.out.println(a);
    String b = a.substring(5);    // b obsahuje řetězec "je skvělá."
    System.out.println(b);
    String c = a.substring(5,7);  // c obsahuje řežezec "je"
```

```
    System.out.println(c);
    String d = a.substring(5,a.length()); // d obsahuje "je skvělá."
    System.out.println(d);
}
```

Při spuštění vypíše program následující:

```
> java -Dfile.encoding=Cp852 cz.darwinsys._03_retezce.SubStringDemo
Java je skvělá.
je skvělá.
je
je skvělá.
>
```

3.2 Rozkládání řetězců pomocí StringTokenizer

Problém

Musíte rozložit řetězec na části oddělené zadanými oddělovači.

Řešení

Svůj řetězec zabalte do instance třídy `StringTokenizer` a aplikujte její metody `hasMoreTokens()` a `nextToken()`. Druhou možností je použití regulárních výrazů (viz kapitola 4).

Metody objektu `StringTokenizer` implementují návrhový vzor `Iterator` (viz návod 7.4):

```
// StrTokDemo1.java
StringTokenizer st = new StringTokenizer("Ahoj světe Javy");

while (st.hasMoreTokens())
    System.out.println("Symbol: " + st.nextToken());
```

`StringTokenizer` rovněž přímo implementuje rozhraní `Enumeration` (také viz návod 7.4), použijete-li však jeho metody, musíte výsledky přetypovat na `String`.

`StringTokenizer` standardně rozdělí řetězec na symboly, které bychom si mohli představit jako slova v evropských jazycích. Někdy však chcete rozdělit řetězec podle jiných oddělovačů. Žádný problém. Při sestavování objektu `StringTokenizer` předejte kromě řetězce, který se má rozložit, i druhý řetězec, který uvádí „oddělovače“. Například:

```
// StrTokDemo2.java
StringTokenizer st = new StringTokenizer("Ahoj, světe|Javy", ", |");

while (st.hasMoreElements())
    System.out.println("Symbol: " + st.nextElement());
```

Ale počkejte, to není všechno! Co když načítáte následující řádky:

```
Jméno|Příjmení|Společnost|Telefonní_číslo
```

a vaše drahá stará teta Begónie nebyla posledních 38 let zaměstnaná? Její pole „Společnost“ bude pravděpodobně prázdné*.

Podíváte-li se velmi důkladně na předchozí příklad kódu, zjistíte, že obsahuje tři oddělovací znaky (čárku, mezeru a svislítko), ale pokud jej spustíte, nejsou zde žádné prázdné symboly představující části řetězce mezi sousedními oddělovači. To znamená, že `StringTokenizer` běž-

* Pokud nejste tak liknaví při aktualizaci osobních údajů jako já.

ně vyřadí sousední po sobě jdoucí oddělovače. U případů jako telefonní seznam, kde musíte zachovat prázdná pole, musíte zvolit jiné řešení. To však s sebou přináší jednu dobrou a jednu špatnou zprávu. Dobrou zprávou je, že to lze provést: jednoduše při sestavování `StringTokenizer` přidáte třetí argument `true`, který znamená, že chcete zobrazit oddělovače jako symboly. Špatnou zprávou je, že uvidíte oddělovače jako symboly, takže musíte sami „oddělit zrno od plev“. Chcete to vidět? Spusťte tento program:

```
// StrTokDemo3.java
StringTokenizer st =
    new StringTokenizer("Ahoj, světe|Javy", ", |", true);

while (st.hasMoreElements())
    System.out.println("Symbol: " + st.nextElement());
```

a získáte následující výstup:

```
>java -Dfile.encoding=Cp852 cz.darwinsys._03_retezce.StrTokDemo3
Symbol: Ahoj
Symbol: ,
Symbol:
Symbol: světe
Symbol: |
Symbol: Javy
```

I když toto chování třídy `StringTokenizer` asi neodpovídá vašim představám o tom, jak by měla fungovat, většinou je použitelná. Příklad 3.1 zpracovává oddělovače a ignoruje ty po sobě jdoucí a vrací výsledek ve formě pole řetězců.

Příklad 3.1. *StrTokDemo4.java (StringTokenizer)*

```
package cz.darwinsys._03_retezce;

import java.util.*;

/** Předvádí práci se třídou StringTokenizer včetně vrácení oddělovačů. */
public class StrTokDemo4 {
    public final static int MAXPOLI = 5;
    public final static String ODDELOVAC = "|";

    /** Zpracuje jeden řetězec; vrací jej jako pole řetězců. */
    public static String[] process(String radek) {
        String[] vysledky = new String[MAXPOLI];

        // Pokud byste nepožádali StringTokenizer, aby poskytoval symboly,
        // prázdné symboly by tiše vyřadil.
        StringTokenizer st = new StringTokenizer(radek, ODDELOVAC,  true);

        int i = 0;
        // Naplňuje každý symbol do aktuálního volného místa v poli.
        while (st.hasMoreTokens()) {
            String s = st.nextToken();
            if (s.equals(ODDELOVAC)) {
                if (i++>=MAXPOLI)
                    // Toto není v pořádku: viz StrTokDemo4b, který používá
                    // Vector, aby povolil jakýkoli počet polí.
                    throw new IllegalArgumentException("Vstupní řádek " +
                        radek + " má příliš mnoho polí");
                continue;
            }
            vysledky[i] = s;
```

```
        }
        return vysledky;
    }

    public static void printResults(String vstup, String[] vystupy) {
        System.out.println("Vstup: " + vstup);
        for (int i=0; i<vystupy.length; i++)
            System.out.println("Výstupem pole " + i + " bylo: " + vystupy[i]);
    }

    public static void main(String[] a) {
        printResults("A|B|C|D", process("A|B|C|D"));
        printResults("A||C|D", process("A||C|D"));
        printResults("A|||D|E", process("A|||D|E"));
    }
}
```

Když spustíte tento příklad, uvidíte, že A je vždy v prvním poli, B (pokud existuje) je v druhém poli atd. Jinými slovy: prázdná pole se zpracují správně:

```
Vstup: A|B|C|D
Výstupem pole 0 bylo: A
Výstupem pole 1 bylo: B
Výstupem pole 2 bylo: C
Výstupem pole 3 bylo: D
Výstupem pole 4 bylo: null
Vstup: A||C|D
Výstupem pole 0 bylo: A
Výstupem pole 1 bylo: null
Výstupem pole 2 bylo: C
Výstupem pole 3 bylo: D
Výstupem pole 4 bylo: null
Vstup: A|||D|E
Výstupem pole 0 bylo: A
Výstupem pole 1 bylo: null
Výstupem pole 2 bylo: null
Výstupem pole 3 bylo: D
Výstupem pole 4 bylo: E
```

Přečtěte si také...

Nyní, když Java obsahuje regulární výrazy (od JDK 1.4), lze nahradit řadu použití třídy `StringTokenizer` regulárními výrazy (viz kapitola 4), a to s podstatně větší flexibilitou. K vytažení všech čísel z řetězce můžete například použít následující kód:

```
Matcher toke = Pattern.compile("\\d+").matcher(vstupniRetezec);
 while (toke.find()) {
        String retezecKurzu = toke.group(0);
        int cisloKurzu = Integer.parseInt(retezecKurzu);
        ...
```

Díky regulárním výrazům může být uživatelský vstup mnohem flexibilnější, než byste mohli jednoduše zpracovat pomocí třídy `StringTokenizer`. Za předpokladu, že čísla představují čísla kurzů v nějaké vzdělávací instituci, vstupy „471,472,570“ nebo „Kurzy 471 a 472, 570, případně pouze „471 472 570“, by měly všechny poskytovat tytéž výsledky.

3.3 Spojování řetězců pomocí operátoru +, StringBuilder (JDK 1.5) a StringBuffer

Problém

Potřebujete (znovu) spojit nějaké řetězce do jediného.

Řešení

Použijte operátor zřetězení (+). Překladač za vás standardně sestaví řetězec v instanci třídy `StringBuilder` za použití její metody `append()`. Je však lepší, když sestavení a práci s metodami provedete sami.

Diskuse

Objekt jedné ze tříd `StringBuilder` v podstatě představuje kolekci znaků. Podobá se objektu `String`, ale, jak jsme se již dříve zmínili, `String` je neměnný.

Obsah objektů `StringBuildery` lze měnit a jsou určeny k sestavování řetězců. Běžně vytvoříte `StringBuilder`, libovolným způsobem vyvoláte metody potřebné k získání posloupnosti znaků a poté zavoláte metodu `toString()` k vygenerování objektu `String`, který reprezentuje stejnou posloupnost znaků a je vyžadován ve většině rozhraní pracujících s textovými řetězci.

Třída `StringBuffer` není v Javě žádnou novinkou – existuje již od JDK 1.1. Některé její metody jsou synchronizované (viz návod 24.5), což v jednovláknových aplikacích způsobuje zbytečnou režii. Ve verzi 1.5 byla tato třída „rozdělena“ na třídu `StringBuffer` (která je nadále synchronizovaná) a novou třídu `StringBuilder` (která není synchronizovaná); tudíž je v jednovláknových aplikacích rychlejší, a proto i vhodnější. Další nová třída `AbstractStringBuilder` je předkem obou tříd. V následujícím pojednání se budu odvolávat na všechny tři třídy pojmem „třídy `StringBuilder`“, protože většinou mají stejné metody. V kódu příkladu používám třídu `StringBuffer` místo `StringBuilder`, protože většina lidí doposud nepřešla na verzi 1.5. Až na skutečnost, že `StringBuilder` není vláknově bezpečný, jsou tyto třídy totožné a lze je vzájemně zaměňovat.

Třídy `StringBuilder` nabízejí různé metody pro vkládání, nahrazování a jiné modifikace daného řetězce. Metoda `append()` obyčejně vrací odkaz na samotný `StringBuilder`, takže se poměrně běžně vyskytují příkazy typu `.append(...). append(...)`. Tento třetí způsob možná dokonce uvidíte i v metodě `toString()`. V příkladu 3.2 jsou znázorněny tři způsoby řetězení řetězců.

Příklad 3.2. *StringBufferDemo.java*

```
package cz.darwinsys.demo._03_retezce;
/**
 * StringBufferDemo: sestaví stejný řetězec třemi různými způsoby.
 */
public class StringBufferDemo {
    public static void main(String[] argv) {
        String s1 = "Ahoj" + ", " + "světe";
        System.out.println(s1);

        // Sestaví objekt StringBuffer a něco k němu připojí.
        StringBuffer sb2 = new StringBuffer();
        sb2.append("Ahoj");
        sb2.append(',');
```

```
        sb2.append(' ');
        sb2.append("světe");

        // Získá hodnotu objektu StringBuffer ve tvaru řetězce a vypíše ji.
        String s2 = sb2.toString();
        System.out.println(s2);

        // Nyní se opět provede všechno, co jsme udělali výše, ale
        // stručnějším způsobem (který se běžně používá v praxi).

        StringBuffer sb3 = new StringBuffer().append("Ahoj").
            append(',').append(' ').append("světe");
        System.out.println(sb3.toString());

        // Cvičení pro čtenáře: proveďte všechno znovu
        // ale bez vytváření JAKÝCHKOLI dočasných proměnných.
    }
}
```

Všechny metody, které ve skutečnosti modifikují více znaků obsahu objektu `StringBulder` – `append()`, `delete()`, `deleteCharAt()`, `insert()`, `replace()` a `reverse()` – vrací v rámci usnadnění tohoto stylu programování odkaz na objekt.

Jako potvrzení toho, že `StringBuilder` je, jak prohlašuje Sun, (vláknově nezabezpečená) náhrada, uvádím příklad `StringBuilderDemo`, což je kopie programu `StringBufferDemo` upravená pro použití `StringBuilderu`. Výstup programu je stejný jako výstup `StringBufferDemo`.

```
package cz.darwinsys.demo._03_retezce;
/**
 * StringBuilderDemo: sestaví stejný řetězec třemi různými způsoby.
 */
public class StringBuilderDemo {

    public static void main(String[] argv) {

        String s1 = "Ahoj" + ", " + "světe";
        System.out.println(s1);

        // Sestaví StringBuffer a něco k němu připojí.
        StringBuilder sb2 = new StringBuilder();
        sb2.append("Ahoj");
        sb2.append(',');
        sb2.append(' ');
        sb2.append("světe");

        // Získá hodnotu StringBuffer ve tvaru řetězce a vypíše ji.
        String s2 = sb2.toString();
        System.out.println(s2);

        // Nyní se opět provede všechno uvedené výše, ale
        // stručnějším způsobem (který se používá v praxi).

        StringBuilder sb3 = new StringBuilder().append("Ahoj").
            append(',').append(' ').append("světe");
        System.out.println(sb3.toString());
    }
}
```

Jako další příklad práce se třídou `StringBuilder` zvažte potřebu převedení seznamu položek na seznam položek oddělených čárkami, jako v následujícím programu:

```
StringBuffer sb = new StringBuffer();
while (st.hasMoreElements()) {
    sb.append(st.nextToken());
    if (st.hasMoreElements())
        sb.append(", ");
}
return sb.toString();
```

Tento vzor spoléhá na skutečnost, že v rozhraní `Enumeration` můžete před získáním dalšího prvku volat informační metodu `hasMoreElements()` vícekrát (nebo v rozhraní `Iterator` metodu `hasNext()`, jak si vysvětlíme v návodu 7.4). Vzor však funguje a zamezuje vkládání nadbytečné čárky za poslední prvek seznamu.

3.4 Zpracování řetězce po jednom znaku

Problém

Chcete zpracovat obsah řetězce po jednom znaku.

Řešení

Použijte cyklus `for` a metodu `charAt()` třídy `String` .

Diskuse

Metoda `charAt()` vybere zevnitř řetězce znak zadaný svým indexem (počáteční znak řetězce má index nula). Pro postupné zpracování všech znaků řetězce použijte cyklus `for` s rozsahem od nuly po `String.length()-1`. Následující kód zpracuje všechny znaky v řetězci.

```
// StrCharAt.java
String a = "Rychlá bronzová liška přeskočila líné hovězí";
for (int i=0; i < a.length(); i++)
    System.out.println("Znak " + i + " je " + a.charAt(i));
```

Kontrolní součet je číslo reprezentující obsah souboru a potvrzující jeho bezchybnost. Přenášíte-li kontrolní součet souboru odděleně od obsahu, příjemce může provést kontrolní součet souboru – za předpokladu, že zná algoritmus – a ověřit, že soubor byl přijat neporušený. V příkladu 3.3 je znázorněn nejjednodušší možný kontrolní součet, který se vypočítá pouze sečtením kódů všech znaků. Na souborech si všimněte, že neobsahují znaky pro přechod na nový řádek; abyste to napravili, vyberte systémovou vlastnost `System.getProperty("line.separator")`; a hodnotu znaku přičtěte k součtu na konci každého řádku. Případně se vzdejte řádkového režimu a čtěte soubor po znacích.

Příklad 3.3. *CheckSum*

```
    /** Kontrolní součet jednoho souboru
     *  zpracovaný pomocí filtru BufferedReader. */
    public int process(BufferedReader is) {
        int soucet = 0;
        try {
            String radekVstupu;

            while ((radekVstupu = is.readLine()) != null) {
```

```
                int i;
                for (i=0; i<radekVstupu.length(); i++) {
                    soucet += radekVstupu.charAt(i);
                }
            }
            is.close();
        } catch (IOException v) {
            System.out.println("IOException: " + v);
        }
        return soucet;
    }
```

3.5 Zarovnání řetězců

Problém

Chcete zarovnat řetězce vlevo, vpravo nebo na střed.

Řešení

Použijte metodu `substring` (návod 3.1) a metodu `stringBuilder` (návod 3.3) nebo pouze moji třídu `StringAlign`, která je založena na třídě `java.text.Format`.

Diskuse

S potřebou vystředit nebo zarovnat text se setkáváme až překvapivě často. Dejme tomu, že chcete vytisknout jednoduchou zprávu s vystředěnými čísly stránek. Nezdá se, že by ve standardním API bylo cokoli, co by mohlo za vás tuto úlohu provést*. Avšak napsal jsem třídu s názvem `StringAlign`, která to udělá. Podívejte se, jak by se mohla použít:

```
/* Zarovná číslo stránky na 70. znak řádku. */
public class StringAlignSimple {

    public static void main(String[] args) {
        // Sestaví "formátovač" pro vystředění řetězců.
        StringAlign formatovac = new StringAlign(70, StringAlign.JUST_CENTER);
        // Vyzkouší to na stránce "i".
        System.out.println(formatovac.format("- i -"));
        // Vyzkouší to na stránce 4. Protože je tento formátovač
        // optimalizovaný pro řetězce, nikoli specificky pro čísla stránek,
        // musíme převést číslo na řetězec.
        System.out.println(formatovac.format(Integer.toString(4)));
    }
}
```

Spustíme-li tuto třídu, vypíše dvě vystředěná čísla řádků:

```
> jikes +E -d . cz/darwinsys/_03_retezce/StringAlignSimple.java
> java cz.darwinsys._03_retezce.StringAlignSimple
                                - i -
                                  4
>
```

* Poznámka českého vydavatele: Java 5.0 zavedla možnost formátovaného výstupu prostřednictvím metody printf, kterou lze aplikovat i na standardní výstup. Samotný textový řetězec lze naformátovat pomocí metody `String.format(String,Object...)`.

Kód třídy `StringAlign` je uveden v příkladu 3.4. Všimněte si, že tato třída je potomkem třídy `Format`. V balíku `java.text` se nachází řada potomků třídy `Format`. Třída `StringAlign` je tak v početné rodině formátovačů, jako například `DateFormat`, `NumberFormat` a dalších, s nimiž se setkáte v následujících kapitolách.

Příklad 3.4. *StringAlign.java*

```
package cz.darwinsys.util;

import java.text.*;
/** Jednoduchý formátovač řetězců (zarovnávač řetězců). */
public class StringAlign extends Format {
    /** Konstanta pro zarovnání vlevo. */
    public static final int VLEVO = 'l';
    /* Konstanta pro vystředění. */
    public static final int VYSTREDIT = 'c';
    /** Konstanta vystředění pro ty, kteří čtou "vystředit" americkým způsobem. */
    public static final int JUST_CENTER = VYSTREDIT;
    /** Konstanta pro řetězce zarovnané vpravo. */
    public static final int VPRAVO = 'r';

    /** Aktuální zarovnání. */
    private int zarovnani;
    /** Aktuální maximální délka. */
    private int maxZnaku;

            /** Sestaví formátovač StringAlign; místo každého volání
            * metody format() se konstruktoru předá délka a zarovnání,
            * protože se předpokládá, že bude běžně sloužit k opakovanému
            * formátování například čísel stránek.
            * @param nZnaku - délka výstupu
            * @param zarovnani - jeden z argumentů  VLEVO, VYSTREDIT nebo VPRAVO
            */
    public StringAlign(int maxZnaku, int zarovnani) {
        switch(zarovnani) {
        case VLEVO:
        case VYSTREDIT:
        case VPRAVO:
            this.zarovnani = zarovnani;
            break;
        default:
            throw new IllegalArgumentException("neplatný argument zarovnání");
        }
        if (maxZnaku < 0) {
            throw new IllegalArgumentException("maxZnaku musí být kladné číslo.");
        }
        this.maxZnaku = maxZnaku;
    }

            /** Formátuje řetězec.
             * @param vstup _ řetězec, který se má zarovnat
            * @param kde - StringBuffer, k němuž se má připojit.
            * @param ignorovat - FieldPosition (může být prázdná, nepoužitá, ale
            * specifikovaná pomocí Format).
            */
    public StringBuffer format(
        Object obj, StringBuffer kde, FieldPosition ignorovat)  {
```

```
        String s = (String)obj;
        String zadany = s.substring(0, Math.min(s.length(), maxZnaku));

        // Dostane mezery na správné místo.
         switch (zarovnani) {
            case VPRAVO:
                pad(kde, maxZnaku - zadany.length());
                kde.append(zadany);
                break;
            case VYSTREDIT:
                int vychPoz = kde.length();
                pad(kde, (maxZnaku - zadany.length())/2);
                kde.append(zadany);
                pad(kde, (maxZnaku - zadany.length())/2);
                // Upraveni kvuli "chybe při zaokrouhlovaní".
                pad(kde, maxZnaku - (kde.length() - vychPoz));
                break;
            case VLEVO:
                kde.append(zadany);
                pad(kde, maxZnaku - zadany.length());
                break;
            }
        return kde;
    }

    protected final void pad(StringBuffer pro, int kolik) {
        for (int i=0; i<kolik; i++)
            pro.append(' ');
    }

    /** Konvenční rutina */
    String format(String s) {
        return format(s, new StringBuffer(), null).toString();
    }

    /** Vyžaduje se metoda ParseObject, ale není zde užitečná. */
    public Object parseObject (String zdroj, ParsePosition poz)  {
        return zdroj;
    }

}
```

Přečtěte si také...

O zarovnávání číselných sloupců pojednává kapitola 5.

3.6 Převody mezi znaky Unicode a řetězci

Problém

Chcete provést konverzi mezi znaky Unicode a objekty `String`.

Řešení

Jelikož hodnoty typu `char` i znaky Unicode jsou 16bitové, může `char` uchovávat jakýkoli znak Unicode.* Metoda `charAt()` třídy `String` vrací znak Unicode. Metoda `append()` třídy `StringBuilder` má tvar, který přijímá `char`. Protože `char` je celočíselný typ, můžete na hodnotách `char` prová-

dět výpočty, ačkoli to není nutné tak často, jako například v jazyce C. Ani se to příliš nedoporučuje, jelikož třída `Character` poskytuje metody, s jejichž pomocí se tyto operace normálně prováděly v jazycích, jako je C. Podívejte se na program, který pro řízení cyklu počítá s kódy znaků a přidává znaky do `StringBuilder` (viz návod 3.3).

```
/**
 * Převod mezi znaky Unicode a řetězci.
 */
public class UnicodeChars {
    public static void main(String[] argv) {
        StringBuffer b = new StringBuffer();
        for (char c = 'a'; c<'d'; c++) {
            b.append(c);
        }
        b.append('\u00a5');    // Symbol japonského Yenu
        b.append('\u01FC');    // Římské AE s ostrým akcentem
        b.append('\u0391');    // Velké řecké písmeno alfa
        b.append('\u03A9');    // Velké řecké písmeno omega

        for (int i=0; i<b.length(); i++) {
            System.out.println("Znak #" + i + " je " + b.charAt(i));
        }
        System.out.println("Dohromady znaky tvoří řetězec " + b);
    }
}
```

Když spustíte tento program, vypíšou se pro znaky ASCII očekávané výsledky. Na mém systému Unix neobsahují výchozí fonty všechny doplňkové znaky, takže se buď vypustí nebo zmapují na náhradní znaky (v návodu 13.3 si ukážeme, jak vypsat text v jiných fontech):

```
Znak #0 je a
Znak #1 je b
Znak #2 je c
Znak #3 je %
Znak #4 je |
Znak #5 je
Znak #6 je )
Dohromady znaky tvoří řetězec abc%|)
```

Dokonce ani v mém systému Windows nemám většinu těchto znaků, ale znaky, o nichž systém ví, že je postrádá, se alespoň vypíšou jako otazníky (fonty systému Windows jsou stejnorodější než fonty různých systémů Unixu, proto je snazší zjistit, co nebude fungovat). Na druhé straně se pokusí vytisknout symbol Yenu jako španělské velké Enye (N nad nímž je ~). Zajímavé je, že pokud ve Windows zapíšu protokol konzoly do souboru a zobrazím ho pod Unixem, symbol Yenu se zobrazí:

```
Znak #0 je a
Znak #1 je b
Znak #2 je c
Znak #3 je ¥
Znak #4 je ?
Znak #5 je ?
Znak #6 je ?
Dohromady znaky tvoří řetězec abc¥???
```

* Poznámka českého vydavatele: Unicode 4.0 zavádí 21bitové znaky a Java 5.0 nabízí mechanismus, jak s těmito znaky pracovat. Podrobnosti o práci s touto rozšířenou sadou znaků najdete např. v naší publikaci *Java 5.0. Novinky jazyka a upgrade aplikací* (Computer Press, 2005).

Přečtěte si také...

Program Unicode v online zdrojích k této knize zobrazí libovolnou 256znakovou sekci znakové sady Unicode. Z webových stránek konsorcia Unicode na adrese *http://www.unicode.org* lze stáhnout dokumentační výpis každého znaku v sadě Unicode .

3.7 Převracení řetězce po slovech nebo znacích

Problém

Chtěli byste převrátit pořadí řetězce po znacích nebo po slovech.

Řešení

Po jednotlivých znacích můžete řetězec jednoduše převrátit pomocí třídy `StringBuilder`. K převrácení řetězce po slovech existuje hned několik technik. Přirozený způsob spočívá v použití tříd `StringTokenizer` a zásobníku `(Stack)`. `Stack` je třída (definovaná v balíku `java.util`; viz návod 7.14), která implementuje jednoduše použitelný objektový zásobník LIFO (last in, first out – poslední dovnitř, první ven).

Diskuse

K převrácení pořadí znaků v řetězci použijte metodu `reverse()` třídy `StringBuilder`:

```
// StringRevChar.java
String sh = "FCGDAEB";
System.out.println(sh + " -> " + new StringBuffer(sh).reverse());
```

Písmena v tomto příkladu představují pořadí křížků ve znělce Západní hudby; v obráceném pořadí uvádí pořadí béček. Pořadí znaků byste mohli eventuálně sami převrátit pomocí režimu „po jednom znaku“ (viz návod 3.4).

Populární *mnemonika* neboli paměťová pomůcka pro pořadí křížků a béček přiřazuje ke každému křížku jedno slovo, které začíná daným písmenem, takže musíme obrátit řetězec po jednotlivých slovech. V příkladu 3.5 se vloží všechny slova do zásobníku (viz návod 7.14) a poté se zpracuje celá posloupnost v pořadí LIFO, čímž se obrátí pořadí slov.

Příklad 3.5. *StringReverse.java*

```
String s = "František Chce Gratulovat Daně A Emil Boženě";


// Vloží řetězec do zásobníku v normálním pořadí.
Stack mujZasobnik = new Stack();
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens()) mujZasobnik.push(st.nextElement());

// Vypíše zásobník pozpátku.
System.out.print('"' + s + '"' + " pozpátku po slovech je:\n\t\"");
while (!mujZasobnik.empty()) {
    System.out.print(mujZasobnik.pop());
    System.out.print(' ');
}
System.out.println('"');
```

3.8 Rozšiřování a komprese tabulátorů

Potřebujete v souboru převést mezeru na znak tabulátoru nebo naopak. Možná budete chtít v rámci úspory místa na disku nahradit mezery za tabulátory nebo upravit soubor pro práci se zařízením nebo programem, který není schopen tabulátory zpracovat.

Řešení

Použijte moji třídu `Tabs` nebo její podtřídu `EnTab`.

Diskuse

V příkladu 3.6 je zdrojový kód programu `EnTab` se vzorovou metodou main. Program zpracovává řádek po řádku. Pro každý znak na řádku, je-li znakem mezera, zjistíme, jestli jej můžeme sloučit s předchozí mezerou pro vypsání jediného znaku tabulátoru. Tento program závisí na třídě `Tabs`, ke které se za okamžik dostaneme. Třída `Tabs` slouží ke zjišťování, které pozice sloupce představují zarážky tabulátorů a které nikoli. Kód rovněž obsahuje několik výpisů `Debug`. (`Debug` jsme si představili v návodu 1.11.)

Příklad 3.6. *Entab.java*

```
package cz.darwinsys._03_retezce;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.io.PrintWriter;

import cz.darwinsys.util.Debug;

/**
 * EnTab: nahradí mezery tabulátory a mezerami. Původně byl přepsán z knihy K&R
 * Software Tools do jazyka C a o několik let později byl znovu přepsán do Javy.
 * Program byl zcela přepracován, aby nezpracovával text po znacích, ale po
 * řádcích.
 *
 * @author Ian F. Darwin, http://www.darwinsys.com/
 * @version $Id: ch03,v 1.3 2004/05/04 18:03:14 ian Exp $
 */
public class EnTab
{
      /** Tabulátory (ovladač logiky tabulátorů). */
      protected Tabs tabulatory;

      /**
       * Deleguje tabulátorům informace o rozmístění.
       *
       * @return
       */
      public int getTabSpacing() {
            return tabulatory.getTabSpacing();
      }

      /**
       * Metoda main: pouze vytvoří objekt EnTab a předá standardní vstup nebo
```

```
 * uvedený soubor(y).
 */
public static void main(String[] argv) throws IOException {
      EnTab et = new EnTab(8);
      if (argv.length == 0) // Provede standardní vstup.
            et.entab(new BufferedReader(new InputStreamReader(System.in)),
                        System.out);
      else
            for (int i = 0; i < argv.length; i++) { // Provede každý soubor.
                  et.entab(new BufferedReader(new FileReader(argv[i])),
                              System.out);
            }
}

/**
 * Konstruktor: pouze uloží hodnoty tabulátorů.
 *
 * @param n Počet mezer, jimiž se nahradí každý tabulátor.
 */
public EnTab(int n) {
      tabulatory = new Tabs(n);
}

public EnTab() {
      tabulatory = new Tabs();
}

/**
 * entab: zpracuje jeden soubor a prázdná místa nahradí tabulátory.
 *
 * @param is BufferedReader otevřený pro soubor, který se má číst.
 * @param out PrintWriter, na který se zašle výstup.
 */
public void entab(BufferedReader is, PrintWriter out) throws IOException {
      String radek;
      int c, col = 0, newcol;

      // Hlavní cyklus: zpracuje celý soubor po řádcích.
      while ((radek = is.readLine()) != null) {
            out.println(entabLine(radek));
      }
}

/**
 * entab: zpracuje jeden soubor a nahradí mezery za tabulátory.
 *
 * @param is BufferedReader otevřený pro soubor, který se má číst..
 * @param out PrintStream, na který se zašle výstup.
 */
public void entab(BufferedReader is, PrintStream out) throws IOException {
      entab(is, new PrintWriter(out));
}

/**
 * entabLine: zpracuje jeden řádek a nahradí mezery za tabulátory.
 *
 * @param radek - řetězec na zpracování
```

```
     */
    public String entabLine(String radek) {
          int N = radek.length(), vystupSl = 0;
          StringBuffer sb = new StringBuffer();
          char z;
          int spotrebovaneMezery = 0;

          for (int vstupSl = 0; vstupSl < N; vstupSl++) {
                z = radek.charAt(vstupSl);
                // Pokud získáme mezeru, spotřebujte ji, neprovádějte její výstup.
                // Pokud nás to zavede k zarážce tabulátoru, proveďte výstup znaku
                // tabulátoru.
                if (z == ' ') {
                      Debug.println("mezera", "Dosažena mezera na " + vstupSl);
                      if (!tabulatory.isTabStop(vstupSl)) {
                            spotrebovaneMezery++;
                      } else {
                            Debug.println("tabulátor", "Dosažena zarážka tabulátoru "
                                        + vstupSl);
                            sb.append('\t');
                            vystupSl += spotrebovaneMezery;
                            spotrebovaneMezery = 0;
                      }
                      continue;
                }

                // Nejsme na mezeře; pokud jsme pouze minuli zarážku tabulátoru,
                // potřebujeme
                // umístit zpět "zbylé" mezery, protože jsme je výše
                // spotřebovali.
                while (vstupSl - 1 > vystupSl) {
                      Debug.println("výplň", "Výplňová mezera na " + vstupSl);
                      sb.append(' ');
                      vystupSl++;
                }

                // Nyní máme prostý znak pro výstup.
                sb.append(z);
                vystupSl++;

          }
          // Pokud je řádek ukončen pomocí koncových (nebo pouze!) mezer,
          // zachovejte je.
          for (int i = 0; i < spotrebovaneMezery; i++) {
                Debug.println("koncové mezery", "Výplňová mezera na konci # " + i);
                sb.append(' ');
          }
          return sb.toString();
    }
}
```

Tento kód byl vytvořen podle klasického díla Kernighana a Plagera *Software Tools*, jak je uvedeno v komentáři. Zatímco jejich verze byla napsaná v jazyce *RatFor* (Rational Fortran), moje verze prodělala od té doby několik překladů. Původní verze ve skutečnosti zpracovávala soubor po jednom znaku a dlouhou dobu se snažila zachovat svou celkovou strukturu. Pro toto vydání knihy jsem konečně přepsal program tak, aby zpracovával najednou celý řádek.

Programem, který postupuje opačným směrem – místo odebírání tabulátorů je raději vkládá dovnitř – je třída `DeTab` znázorněna v příkladu 3.7; níže uvádíme pouze základní metody.

Příklad 3.7. *DeTab.java*

```
package cz.darwinsys._03_retezce;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class DeTab {
    Tabs ts;       // Inicializuje se v konstruktoru

    public static void main(String[] argv) throws IOException {
        DeTab dt = new DeTab(8);
        dt.detab(new BufferedReader(new InputStreamReader(System.in)),
                new PrintWriter(System.out));
    }

    /** Odstraní tabulátory z jednoho souboru (tabulátory nahradí mezerami)
     * @param is - soubor, který se má zpracovat
     * @param out - aktualizovaný soubor
     */
    public void detab(BufferedReader is, PrintWriter out) throws IOException {
        String radek;
        while ((radek = is.readLine()) != null) {
            out.println(detabLine(radek));
        }
    }

    /** Odstraní tabulátory z jednoho řádku (nahradí tabulátory mezerami)
     * @param radek - řádek, který se má zpracovat
     * @return aktualizovaný řádek
     */
    public String detabLine(String radek) {
        char c;
        int sloupec;
        StringBuffer sb = new StringBuffer();
        sloupec = 0;
        for (int i = 0; i < radek.length(); i++) {
            // Buď obyčejný znak nebo tabulátor.
            if ((c = radek.charAt(i)) != '\t') {
                sb.append(c); // Obyčejný
                ++sloupec;
                continue;
            }
            do {  // Tabulátor, provede rozšíření, musí vložit >=1 mezeru.
                sb.append(' ');
            } while (!ts.isTabStop(++sloupec));
        }
        return sb.toString();
    }
}
```

Třída `Tabs` poskytuje dvě metody: `settabpos()` a `istabstop()`. Zdrojový kód třídy `Tabs` je uveden v příkladu 3.8.

Příklad 3.8. *Tabs.java*

```
package cz.darwinsys._03_retezce;

import cz.darwinsys.util.Debug;

public class Tabs {
    /** Tabulátory */
    public final static int VYCHMEZTAB =   8;
    /** Nastavení aktuální zarážky tabulátoru. */
    protected int mezeraTabulatoru = VYCHMEZTAB;
    /** Nejdelší řádek, u kterého se budeme starat o tabulátory. */
    public final static int MAXRADEK  = 250;
    /** Aktuální zarážky tabulátorů */
    protected boolean[] zarazkytabulatoru;

    /** Sestaví objekt Tabs s daným nastavením zarážek tabulátorů. */
    public Tabs(int n) {
        if (n <= 0)
            n = 1;
        zarazkytabulatoru = new boolean[MAXRADEK];
        mezeraTabulatoru = n;
        settabs();
    }

    /** Sestaví objekt Tabs s výchozím nastavením zarážek tabulátorů. */
    public Tabs() {
        this(VYCHMEZTAB);
    }

    /** settabs - nastaví výchozí zarážky tabulátorů */
    private void settabs() {
        for (int i = 0; i < zarazkytabulatoru.length; i++) {
            zarazkytabulatoru[i] = ((i+1) % mezeraTabulatoru) == 0;
            Debug.println("tabulátory",
                   "Tabulátory[" + i + "]=" + zarazkytabulatoru[i]);
        }
    }

    /**
     * @return Vrací mezeraTabulatoru.
     */
    public int getTabSpacing() {
        return mezeraTabulatoru;
    }

    /** isTabStop - je-li daný sloupec zarážkou tabulátoru, vrací true.
     * Je-li aktuální řádek vstupu příliš dlouhý, vloží tabulátory kamkoli,
     * nevyhodí žádnou výjimku.
     * @param sloupec - aktuální číslo sloupce
     */
    public boolean isTabStop(int sloupec) {
        if (sloupec > zarazkytabulatoru.length-1) {
            zarazkytabulatoru = new boolean[zarazkytabulatoru.length * 2];
            settabs();
        }
        return zarazkytabulatoru[sloupec];
    }
}
```

3.9 Ovládání velikosti písmen

Problém

Potřebujete převést řetězec na velká nebo malá písmena, případně porovnat řetězce bez ohledu na velikost písmen.

Řešení

Třída `String` obsahuje celou řadu metod pro zpracování dokumentů s velkými i malými písmeny. Metody `toUpperCase()` a `toLowerCase()` vrací nový řetězec, který je (jak napovídají samotné názvy metod) kopií aktuálního řetězce zkonvertovanou na velká nebo malá písmena,. Každou metodu lze vyvolat buď bez argumentů nebo s argumentem `Locale`, který udává konverzní pravidla; tento argument je nezbytný kvůli internacionalizaci. Java poskytuje podstatně více internacionalizačních a lokalizačních funkcí než obyčejné jazyky. O těchto funkcích budeme hovořit v kapitole 15. Zatímco metoda `equals()` vás informuje o tom, jestli je jiný řetězec naprosto stejný, `equealsIngoreCase()` vám oznámí, jestli jsou všechny znaky stejné bez ohledu na jejich velikost. V následujícím příkladu není použit alternativní argument `Locale`; používá výchozí nastavení systému:

```
String nazev = "Java Cookbook";
System.out.println("Normalní název:\t" + nazev);
System.out.println("Velkými písmeny:\t" + nazev.toUpperCase());
System.out.println("Malými písmeny:\t" + nazev.toLowerCase());
// Jako by se jednalo o identifikátory Javy.
String nazevJavy = "java cookBook";
if (!nazev.equals(nazevJavy))
    System.err.println("metoda equals() správně ohlásí hodnotu false");
else
    System.err.println("metoda equals() nesprávně ohlásí hodnotu true");
if (nazev.equalsIgnoreCase(nazevJavy))
    System.err.println(
            "metoda equalsIgnoreCase() správně ohlásí hodnotu true");
else
    System.err.println(
            "metoda equalsIgnoreCase() nesprávně ohlásí hodnotu false");
    }
```

Po spuštění tohoto programu se nejprve vypíše název knihy převedený na velká a malá písmena a následně program ohlásí, že obě metody fungují podle očekávání.*

```
C:\javasrc>java  -Dfile.encoding=Cp852 cz.darwinsys._03_retezce.Case
Normální název: Java Cookbook
Velkými písmeny:  JAVA COOKBOOK
Malými písmeny:  java cookbook
metoda equals() správně ohlásí hodnotu false
metoda equalsIgnoreCase() správně ohlásí hodnotu true
```

* Poznámka českého vydavatele: Posloupnost řádků nemusí vždy vyjít tak, jak je uvedeno, protože standardní i chybový výstup běží ve vlastních vláknech a chybový výstup má někdy tendenci předbíhat ten standardní. Může se tedy stát, že se na výstupu dozvíte nejdříve, co metody správně ohlásí, a teprve pak se dovypíšou porovnávané texty.

Přečtěte si také...

Při hledání řetězce můžete jednodušeji ignorovat velikost písmen pomocí regulárních výrazů (viz kapitola 4).

3.10 Odsazení textových dokumentů

Problém

Potřebujete odsadit text v dokumentu.

Řešení

Pro odsazení textu buďto vygenerujte řetězec s pevnou délkou a umístěte ho před každý řádek výstupu nebo použijte cyklus `for` a vypište správný počet mezer:

```
// Indent.java
/** Výchozí počet mezer. */
static int nMezer = 10;

while ((radekVstupu = is.readLine()) != null) {
    for (int i=0; i<nMezer; i++) System.out.print(' ');
    System.out.println(radekVstupu);
}
```

Mezery bychom mohli efektivněji generovat, kdybychom si vytvořili dlouhý řetězec mezer a získali potřebný počet mezer pomocí metody `substring()`.

Chcete-li zmenšit odsazení, vytvořte řetězec, který neobsahuje počáteční mezery, pomocí metody `substring`. Dávejte si přitom pozor na vstupy, které jsou kratší než odstraňovaný počet mezer! Vzhledem k oblibě tohoto požadavku vám jeden takový vstup ukážu. Nejprve se však podívejme na ukázku objektu `Undent`, který byl vytvořen zmenšením odsazení o 5, což znamená odstranění až pěti mezer (přitom však nesmíme ztratit nemezerové znaky na prvních pěti pozicích):

```
>java -Dfile.encoding=Cp852 cz.darwinsys._03_retezce.Undent
Ahoj, všichni
Ahoj, všichni
 Ahoj
Ahoj
     Ahoj
Ahoj
      Ahoj
 Ahoj
```

Program jsem otestoval zadáním obvyklého testovacího řetězce „Ahoj, všichni“, který se pěkně vypisuje. Poté zadáním řetězce „Ahoj“ s jednou mezerou, která se odstranila. U řetězce s pěti mezerami se odstranilo přesně pět mezer. U řetězce s šesti a více mezerami se odstraní pouze prvních pět mezer. Prázdný řádek se objeví jako prázdný řádek (tj. bez vyvolání `Exception` nebo jiného bláznivého chování). Myslím tedy, že funguje!

```
package cz.darwinsys._03_retezce;

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

```
public class Undent {
    /** Maximální počet mezer k odstranění. */
    protected int nMezery;

    Undent(int n) {
        nMezery = n;
    }

    public static void main(String[] av) {
        Undent c = new Undent(5);
        switch(av.length) {
            case 0: c.process(new BufferedReader(
                              new InputStreamReader(System.in))); break;
            default:
        for (int i=0; i<av.length; i++)
            try {
                c.process(new BufferedReader(new FileReader(av[i])));
            } catch (FileNotFoundException v) {
                System.err.println(v);
            }
        }
    }

    /** Zpracuje jeden soubor, daný otevřenému BufferedReader. */
    public void process(BufferedReader is) {
        try {
            String radekVstupu;

            while ((radekVstupu = is.readLine()) != null) {
                int kOdstraneni = 0;
                for (int i=0; i<nMezery && i < radekVstupu.length(); i++)
                    if (Character.isSpace(radekVstupu.charAt(i)))
                        ++kOdstraneni;
                System.out.println(radekVstupu.substring(kOdstraneni));
            }
            is.close();
        } catch (IOException v) {
            System.out.println("IOException: " + v);
        }
    }
}
```

3.11 Vkládání netisknutelných znaků

Problém

Potřebujete vložit do řetězců netisknutelné znaky.

Diskuse

V tabulce 3.1 jsou uvedeny nahrazované (escape) řetězce Javy.

* Poznámka českého vydavatele: Metoda `isSpace(char)`, použitá v příkladu, patří mezi zavržené. Doporučuje se místo ní používat metodu `isWhitespace(char)`.

Tabulka 3.1. *Nahrazované řetězce*

Vytvoří	Použití	Poznámky
tabulátor	\t	
posun o řádek (v Unixu nový řádek)	\n	Viz metoda pro získání systémové vlastnosti `System.getProperty("line.separator")`, která vám poskytuje informace o ukončení řádku na dané platformě.
návrat vozíku	\r	
nová stránka	\f	
návrat o jeden znak zpět	\b	
apostrof	\'	
uvozovky	\"	
znak Unicode zadaný svým kódem v šestnáctkové soustavě	\u*NNNN*	Čtyři hexadacimální číslice (nikoli \x jako v C/C++), viz *http://www.unicode.org*
znak zadaný svým kódem v osmičkové soustavě (!)	*NNN*	Kdo dnes používá oktalovou (osmičkovou) soustavu?
zpětné lomítko	\\	

Podívejte se na příklad kódu, který znázorňuje práci s většinou těchto řetězců:

```
// StringEscapes.java
System.out.println("Práce s řetězci v Javě:");
// System.out.println("Zvuková výstraha (cinknutí): \a");    // Nepodporováno
System.out.println("Zvuková výstraha zadaná oktalově: \007");
System.out.println("Klávesa tabulátoru: \t(následující text)");
System.out.println("Nový řádek: \n(následující text)");
System.out.println("Znak Unicode: \u0207");
System.out.println("Znak zpětného lomítka: \\");
```

Potřebujete-li vkládat hodně ne-ASCII znaků, mohli byste zvážit použití vstupních metod, o nichž se stručně zmiňuje dokumentace k JDK.

3.12 Odstraňování mezer z konce řetězce

Problém

Potřebujete pracovat s řetězcem bez přebytečných počátečních nebo koncových mezer, které mohl uživatel napsat.

Řešení

Použijte metodu `trim()` třídy `String`.

Diskuse

V příkladu 3.9 pomocí metody `trim()` odstraníme z řádků zdrojového kódu v Javě jakýkoli počet počátečních mezer anebo tabulátorů, abychom vyhledali znaky `//+` a `//-`. Tyto řetězce

představují zvláštní komentáře Javy, s jejichž pomocí značím části programů v této knize, které chci zahrnout do výtisku.

Příklad 3.9. *GetMark.java (ořezávání a porovnávání řetězců)*

```
package cz.darwinsys._03_retezce;

import java.io.*;

public class GetMark
{
    /** Výchozí počáteční značka. */
    public final String pocatecniZnacka = "//+";

    /** Výchozí koncová značka. */
    public final String koncovaZnacka = "//-";

    /**
     * Nastavte tuto hodnotu na TRUE při vylučovacím režimu (tj. odstraňujete-li
     * části z celkového programu) a na FALSE při vybíracím režimu, při němž vás
     * zajímají naopak právě ty extrahované části.
     */
    public final static boolean START = true;

    /** Proměna má hodnotu true, pokud se nyni nacházime unitř značek. */
    protected boolean vypsat = false;

    /** TRUE požadujete-li čísla řádků */
    protected final boolean number = false;

    // -
    /* Tato část má být vyjmuta! */
    int foo = 42;

    // +

    /**
     * Vyjme označené části ze souboru pomocí zadané instance třídy
     * LineNumberReader. Toto je klíčová část programu a může být použita uvnitř
     * jiných programů pomocí své obálky main.
     */
    public void process(String fileName, LineNumberReader is, PrintStream out) {
        // +
        int nLines = 0;
        try {
            String inputLine;

            while ((inputLine = is.readLine()) != null) {
                if (inputLine.trim().equals(pocatecniZnacka)) {
                    if (vypsat)
                        // Odchází na stderr, takže lze přesměrovat
                        System.err.println("ERROR: START INSIDE START, "
                                + fileName + ':' + is.getLineNumber());
                    vypsat = true;
                } else if (inputLine.trim().equals(koncovaZnacka)) {
                    if (!vypsat)
                        System.err.println("ERROR: STOP WHILE STOPPED, "
                                + fileName + ':' + is.getLineNumber());
```

```
                    vypsat = false;
                } else if (vypsat) {
                    if (number) {
                        out.print(nLines);
                        out.print(": ");
                    }
                    out.println(inputLine);
                    ++nLines;
                }
            }
            is.close();
            out.flush(); // Nesmíme jej zavřít - volající program
                         // jej může ještě potřebovat
            if (nLines == 0)
                System.err.println("ERROR: No marks in " + fileName
                        + "; no output generated!");
            // -
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

3.13 Syntaktická analýza dat oddělených čárkou

Problém

Máte řetězec nebo soubor řádků obsahující hodnoty oddělené čárkami (formát CSV) a potřebujete tyto hodnoty načítat. Formát CSV používá k exportování dat spoustu tabulkových procesorů ve Windows a některé databáze.

Řešení

Použijte moji třídu CSV nebo regulární výrazy (viz kapitola 4).

Diskuse

CSV je ošidný formát. Na první pohled vypadá jednoduše, ale hodnoty mohou (ale také nemusí) být v uvozovkách. Jsou-li hodnoty v uvozovkách, mohou dále obsahovat uvozovky předeslané zpětným lomítkem. To však dalece překračuje schopnosti třídy `StringTokenizer` (návod 3.2). Pro zpracování tohoto formátu je potřeba buď napsat poměrně velký program v Javě nebo použít regulární výrazy. Předvedeme si oba způsoby.

Nejprve se podívejme na program v Javě. Prozatím předpokládejme, že máme třídu s názvem `CSV`, která má konstruktor bez argumentu a metodu s názvem `parse()`, jež přijímá řetězec představující jeden řádek vstupního souboru. Metoda `parse()` vrací seznam polí. Pro větší flexibilitu vrací místo pole seznam (List), z kterého můžete získat `Iterator` (viz návod 7.4). K řízení cyklu a testu možnosti použití metody `next()` pro získání dalšího objektu používáme metodu `hasNext()` rozhraní `Iterator`:

```
package cz.darwinsys._03_retezce;

import java.util.*;

/* Jednoduchý ukázkový program třídy syntaktického analyzátoru CSV.
```

```
 */
public class CSVSimple {
    public static void main(String[] args) {
        CSV analyzator = new CSV();
        List  seznam = analyzator.parse(
            "\"LU\",86.25,\"11/4/1998\",\"2:19PM\",+4.0625");
        Iterator it = seznam.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Syntakticky analyzovaný řetězec vypadá po odstranění zpětných lomítek následovně:

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625
```

Spuštěním programu `CSVSimple` obdržíte následující výstup:

```
> java cz.darwinsys._03_retezce.CSVSimple
LU
86.25
11/4/1998
2:19PM
+4.0625
>
```

Ale co samotná třída `CSV`? Kód uvedený v příkladu 3.10 vznikl jako překlad programu CSV, který napsali Brian W. Kernighan a Rob Pike v C++, a který se objevil v jejich knize *The Practice of Programming* (Addison Wesley). Verze C++ smíchávala vstupní zpracování se syntaktickou analýzou; moje třída `CSV` provádí pouze syntaktickou analýzu, protože vstup by mohl pocházet z nejrůznějších zdrojů. A v průběhu doby byl do značné míry přepracován. Hlavní činnost se provádí v metodě `parse()`, která deleguje zpracování jednotlivých polí na `advquetod()` v případech, že pole začínají uvozovkou; jinak deleguje zpracování na `advplain()`.

Příklad 3.10. *CSV.java*

```
package cz.darwinsys. _03_retezce;

import java.util.*;

import cz.darwinsys.util.Debug;

/**
 * Hodnoty oddělené čárkou (CSV), běžný formát souborů ve Windows. Vzorový
 * vstup: "LU",86.25,"11/4/1998","2:19PM",+4.0625
 * <p>
 * Vnitřní logika byla adaptována z původního programu C++, který byl chráněn
 * autorskými právy Copyright (C) 1999 Lucent Technologies Vyjmuto z knihy
 * 'The Practice of Programming' od autorů Briana W. Kernighana a Roba Pikeho.
 * <p>
 * Včetně oprávnění z webových stránek http://tpop.awl.com/, které prohlašují:
 * "Tento kód můžete používat k libovolným účelům, pokud v něm ponecháte
 * poznámku o autorských právech a odkaz na knihu." Já jsem tak učinil.
 *
 * @author Brian W. Kernighan a Rob Pike (původní program v C++)
 * @author Ian F. Darwin (překlad do Javy a odstranění I/O)
 * @author Ben Ballard (přepsal advQuoted pro ošetření '""'
 *          a upravil z hlediska čitelnosti)
```

```
 */
public class CSV
{

    public static final char VYCHOZI_ODDELOVAC = ',';

    /** Sestaví syntaktický analyzátor CSV s výchozím oddělovačem (','). */
    public CSV() {
        this(VYCHOZI_ODDELOVAC);
    }

    /**
     * Sestaví syntaktický analyzátor CSV s daným oddělovačem.
     *
     * @param oddelovac Jediný znak slouží jako oddělovač (nikoli seznam znaků
     *            oddělovače)
     */
    public CSV(char oddelovac) {
        oddelovacPole = oddelovac;
    }

    /** Pole v aktuálním řetězci */
    protected List seznam = new ArrayList();

    /** Znak oddělovače pro tento syntaktický analyzátor */
    protected char oddelovacPole;

    /**
     * Analyzuje: rozdělí vstupní řetězec do polí
     *
     * @return objekt rozhrani java.util.Iterator obsahující každé pole z
     *         originálu jako řetězec.
     */
    public List parse(String radek) {
        StringBuffer sb = new StringBuffer();
        seznam.clear(); // Provede obnovu na počáteční stav.
        int i = 0;

        if (radek.length() == 0) {
            seznam.add(radek);
            return seznam;
        }

        do {
            sb.setLength(0);
            if (i < radek.length() && radek.charAt(i) == '"')
                i = advQuoted(radek, sb, ++i); // "Přeskočí uvozovku
            else
                i = advPlain(radek, sb, i);
            seznam.add(sb.toString());
            Debug.println("csv", sb.toString());
            i++;
        } while (i < radek.length());

        return seznam;
    }
```

```
    /** advQuoted: uvozené pole; vrací index dalšího oddělovače */
    protected int advQuoted(String s, StringBuffer sb, int i) {
        int j;
        int len = s.length();
        for (j = i; j < len; j++) {
            if (s.charAt(j) == '"' && j + 1 < len) {
                if (s.charAt(j + 1) == '"') {
                    j++;  // Přeskočí escape znak
                } else if (s.charAt(j + 1) == oddelovacPole) { // Další
                                                               // Oddělovač
                    j++;  // Přeskočí koncové uvozovky
                    break;
                }
            } else if (s.charAt(j) == '"' && j + 1 == len) {   // Koncové
                                                               // uvozovky na
                                                               // konci řádku
                break;    // Hotovo
            }
            sb.append(s.charAt(j)); // Normální znak.
        }
        return j;
    }

    /** advPlain: uvozené pole; vrací index dalšího oddělovače */
    protected int advPlain(String s, StringBuffer sb, int i) {
        int j;

        j = s.indexOf(oddelovacPole, i); // Hledá oddělovač
        Debug.println("csv", "i = " + i + ", j = " + j);
        if (j == -1) { // Žádný nenalezen
            sb.append(s.substring(i));
            return s.length();
        } else {
            sb.append(s.substring(i, j));
            return j;
        }
    }
}
```

Mezi doprovodnými programy najdete soubor *CSVFile.java*, který načítá textový soubor a projede jej pomocí metody `parse()`.

O regulárních výrazech jsme doposud nehovořili (napravíme to v kapitole 4), ale celá řada čtenářů je umí obecným způsobem používat. Následující příklad znázorňuje sílu regulárních výrazů a také vám poskytuje kód, který může znovu upotřebit. Všimněte si, že tento program nahrazuje *veškerý* kód,* jak v souboru *CSV.java*, tak v *CSVFile.java*. Klíčem k pochopení regulárních výrazů je skutečnost, že krátká specifikace může odpovídat velkému množství dat.

```
package cz.darwinsys._03_retezce;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
```

* S upozorněním, že nezpracuje různé oddělovače; to by bylo možné přidat pomocí GetOpt a sestavením vzoru kolem oddělovače.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/* Jednuduchý ukázkový program porovnávání CSV pomocí regulárních výrazů.
 * NEPOUŽÍVÁ třídu "CSV", definovanou v knize Java CookBook, ale pracuje se
 * zjednodušeným vzorem regulárních výrazů z kapitoly 7 <em>Mastering Regular
 * Expressions</em> (strana 205, první vydání)
 * @version $Id: ch03,v 1.3 2004/05/04 18:03:14 ian Exp $
 */
public class CSVRE {
    /** Dost spletitý vzor užívaný k porovnávání CSV tvoří tři
     * mutace: první porovnává uvozené pole, druhá neuvozené pole,
     * třetí prázdné pole.
     */
    public static final String CSV_VZOR = "\"([^\"]+?)\",?|([^,]+),?|,";
    private static Pattern csvRE;

    public static void main(String[] argv) throws IOException {
        System.out.println(CSV_VZOR);
        new CSVRE().process(new BufferedReader(new InputStreamReader(System.in)));
    }

    /** Sestaví regulární výraz na základě syntaktického analyzátoru CSV. */
    public CSVRE() {
        csvRE = Pattern.compile(CSV_VZOR);
    }

    /** Zpracuje jeden soubor. Po jednom řádku deleguje metodu parse() */
    public void process(BufferedReader in) throws IOException {
        String radek;

        // Pro každý řádek
        while ((radek = in.readLine()) != null) {
            System.out.println("řádek = `" + radek + "'");
            List l = parse(radek);
            System.out.println("Nalezeny " + l.size() + " položky.");
            for (int i = 0; i < l.size(); i++) {
                System.out.print(l.get(i) + ",");
            }
            System.out.println();
        }
    }

    /** Syntakticky analyzuje jeden řádek.
     * @return Seznam řetězců bez jejich uvozovek
     */
    public List parse(String radek) {
        List seznam = new ArrayList();
        Matcher m = csvRE.matcher(radek);
        // Pro každé pole
        while (m.find()) {
            System.out.println(m.groupCount());
            String odpovida = m.group();
            if (odpovida == null)
                break;
            if (odpovida.endsWith(",")) {  // Upravení koncových znaků
                odpovida = odpovida.substring(0, odpovida.length() - 1);
```

```
            }
            if (odpovida.startsWith("\"")) { // Předpokládá, že timto končí.
                odpovida = odpovida.substring(1, odpovida.length() - 1);
            }
            if (odpovida.length() == 0)
                odpovida = null;
            seznam.add(odpovida);
        }
        return seznam;
    }
}
```

Někdy je „přímo děsivé“, kolik kódu můžete vypustit díky jednomu dobře formulovanému regulárnímu výrazu.

3.14 Program: Jednoduchý formátovač textu

Tento program je velmi jednoduchý formátovač textu, který znázorňuje to, co lidé používali na většině výpočetních platforem před vznikem samostatných grafických textových editorů, laserových tiskáren a případně DTP, textových procesorů a kancelářských balíků. Jednoduše ze souboru načítá slova – dříve vytvořená pomocí textového editoru – a vypisuje je, dokud nedosáhne pravého okraje. Poté vyvolá `println()` pro doplnění konce řádku. Podívejte se na příklad vstupního souboru:

```
Dnes máme ale pěkný
den, že pane Mxyzzptllxy?
Myslím, že bychom měli
jít na procházku.
```

Když program `Fmt` obdrží jako vstup výše uvedený text, vypíše úhledně naformátované řádky:

```
Dnes máme ale pěkný den, že pane Mxyzzptllxy? Myslím, že bychom měli jít na procház-
ku.
```

Jak si můžete všimnout, program uzpůsobí text, který jsme mu předali, okrajům a z původního textu odstraní všechna zalomení řádků. Podívejte se na kód:

```
package cz.darwinsys._03_retezce;

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

/**
 * Fmt - formátuje text (jako Berkeley Unix fmt).
 */
public class Fmt {
    /** Maximální šířka sloupce */
    public static final int SIRKASLOUPCE=72;
    /** Soubor, který načítáme a formátujeme */
    BufferedReader in;

    /** Existují-li soubory, naformátuje všechny, jinak formátuje standardní vstup. */
    public static void main(String[] av) throws IOException {
        if (av.length == 0)
            new Fmt(System.in).format();
        else for (int i=0; i<av.length; i++)
```

```
            new Fmt(av[i]).format();
    }

    /** Sestaví formátovač daného názvu souboru. */
    public Fmt(String nazevs) throws IOException {
        in = new BufferedReader(new FileReader(nazevs));
    }

    /** Sestaví formátovač daného otevřeného proudu. */
    public Fmt(InputStream soubor) throws IOException {
        in = new BufferedReader(new InputStreamReader(soubor));
    }

    /** Formátuje soubor obsažený v sestaveném objektu Fmt. */
    public void format() throws IOException {
        String w, f;
        int sloup = 0;
        while ((w = in.readLine()) != null) {
            if (w.length() == 0) {              // Prázdný řádek
                System.out.print("\n");         // Konec aktuálního řádku
                if (sloup>0) {
                    System.out.print("\n");     // Vypíše prázdný řádek
                    sloup = 0;
                }
                continue;
            }

            // Jinak je to text, takže ho naformátuje.
            StringTokenizer st = new StringTokenizer(w);
            while (st.hasMoreTokens()) {
                f = st.nextToken();

                if (sloup + f.length() > SIRKASLOUPCE) {
                    System.out.print("\n");
                    sloup = 0;
                }
                System.out.print(f + " ");
                sloup += f.length() + 1;
            }
        }
        if (sloup>0) System.out.print("\n");
        in.close();
    }
}
```

Nepatrně důmyslnější verzi tohoto programu, `Fmt2`, najdete v online zdrojích k této knize*. V rámci poskytování omezené kontroly nad formátováním používá „tečkové příkazy“ – řádky začínají tečkami. Rodina formátovačů „tečkových příkazů“ zahrnuje unixové *roff*, *nroff*, *troff* a *groff*, které patří do stejné rodiny jako programy s názvy *runoff* na systémech číslicových zařízení. Tyto formátovače vychází z programu *runoff* od J. Saltzera, který se ponejprv objevil v systému Multics (předchůdce Unixu) a odtud si našel cestu do různých operačních systémů. Abychom alespoň trochu šetřili naše lesy, nebudeme zde program `Fmt2` uvádět; v rámci přidání funkcionality vytvoří podtřídu `Fmt` a překryje metodu `format()`.

* Poznámka českého vydavatele: Programy, které nejsou součástí tištěného textu, nejsou lokalizované.

3.15 Program: Fonetický porovnávač názvů (soundex)

Vývoj algoritmu Soundexu, v němž se každá daná množina souhlásek mapuje na určité číslo, byl inspirován problémovým porovnáváním (amerických) jmen. Algoritmus byl pravděpodobně původně vymyšlen pro Úřad pro sčítání lidu k mapování podobně znějících jmen, protože v té době bylo hodně lidí negramotných a nebyli schopni konzistentně hláskovat své příjmení. Ale své uplatnění najde i dnes – například v telefonním seznamu společností. Jména Darwin a Derwin se například mapují na D650 a Darwent se mapuje na D653, což jej umístí blízko D650. Všechna tato jména představují historické varianty téhož jména. Předpokládejme, že potřebujeme uspořádat řádky obsahující tato jména: kdybychom mohli na začátku každého řádku vypsat čísla Soundexu, bylo by to snadné. Podívejte se na jednoduchou ukázku třídy `Soundex`:

```
package cz.darwinsys._03_retezce;

/** Jednoduchá ukázka použití třídy Soundex.  */
public class SoundexSimple {

    /** hlavní */
    public static void main(String[] args) {
        String[] jmena = {
            "Darwin, Ian",
            "Davidson, Greg",
            "Darwent, William",
            "Derwin, Daemon"
        };
        for (int i = 0; i< jmena.length; i++)
            System.out.println(Soundex.soundex(jmena[i]) + ' ' + jmena[i]);
    }
}
```

Spusťme ji:

```
> jikes +E -d . cz/darwinsys/_03_retezce/SoundexSimple.java
> java cz.darwinsys._03_retezce.SoundexSimple | sort
D132 Davidson, Greg
D650 Darwin, Ian
D650 Derwin, Daemon
D653 Darwent, William
>
```

Jak si můžete všimnout, všechny různé varianty jména Darwin (včetně Daemon Derwin)* se uspořádají vedle sebe a jsou odlišné od jmen Davidson (a Davis, Davies atd.), která se při použití prostého abecedního uspořádání normálně zobrazí mezi jmény Darwin a Derwin. Algoritmus Soundexu splnil svůj úkol.

Podívejte se na samotnou třídu `Soundex`: pro převod jmen na kódy Soundexu využívá objekty `String` a `StringBuilder`. Test JUnit (viz návod 1.14) je rovněž online jako *SoundexTest.java*.

```
package cz.darwinsys._03_retezce;
```

* V terminologii Unixu je *daemon* server. Toto slovo nemá nic společného s démony, ale odkazuje na pomocníka nebo asistenta. Derwin Deamon je ve skutečnosti postava v online kresleném seriálu „Source Wars“ od Susannah Coleman; viz *http://darby.daemonnews.org*.

```
import cz.darwinsys.util.Debug;
/**
 * Soundex - algoritmus Soundexu tak, jak jej popisuje Knuth
 * <p>
 * Tato třída implementuje algoritmus soundexu podle popisu Donalda
 * Knutha ve 3. svazku knihy <I>The Art of Computer Programming</I>.
 * Algoritmus je určen pro hašování slov (zejména příjmení) do
 * malého prostoru pomocí jednoduchého modelu, který přiblíží
 * zvuk slova při jeho vyslovení anglickým mluvčím. Každé slovo se
 * redukuje na čtyřznakový řetězec: první znak bude velké písmeno a
 * zbývající tři znaky budou číslice. Ligatury se
 * složí do jedné číslice.
 *
 * <h2>PŘÍKLADY</h2>
 * Knuthovy příklady různých jmen a kódů soundexu, na které se
 * mapují, jsou:
 * <b>Euler, Ellery -> E460
 * <b>Gauss, Ghosh -> G200
 * <b>Hilbert, Heilbronn -> H416
 * <b>Knuth, Kant -> K530
 * <b>Lloyd, Ladd -> L300
 * <b>Lukasiewicz, Lissajous -> L222
 *
 * <h2>OMEZENÍ</h2>
 * Protože původně byl algoritmus soundexu <B>dlouhou</B> dobu používán
 * ve Spojených státech amerických, pracuje pouze s anglickou
 * abecedou a  výslovností.
 * <p>
 * Jelikož mapuje velký prostor (libovolnou délku řetězců) do
 * malého prostoru (jediné písmeno a 3 číslice), nelze dělat žádné
 * hypotézy o podobnosti dvou řetězců, které končí stejným kódem
 * soundexu.  Například jak "Hilbert", tak "Heilbronn" končí
 * kódem soundexu "H416".
 * <p>
 * Metoda soundex() je statická a neudržuje stavy instancí;
 * to znamená, že nikdy nebudete potřebovat vytvářet instance této třídy.
 *
 * @author Implementaci Perlu vytvořil Mike Stok (<stok@cybercom.net>)
 * z popisu, jejž poskytl Knuth.  Ian Phillips (<ian@pipex.net>) a
 * Rich Pinder (<rpinder@hsc.usc.edu>) dodali nápady a odhalili chyby.
 */
public class Soundex {

    /* Implementuje mapování
     * z: AEHIOUWYBFPVCGJKQSXZDTLMNR
     * na:   00000000111122222222334556
     */
    public static final char[] MAP = {
        //A  B   C   D   E   F   G   H   I   J   K   L   M
        '0','1','2','3','0','1','2','0','0','2','2','4','5',
        //N  O   P   W   R   S   T   U   V   W   X   Y   Z
        '5','0','1','2','6','2','3','0','1','0','2','0','2'
    };

    /** Převede daný řetězec na jeho kód Soundexu.
     * @return prázdnou hodnotu, pokud nelze daný řetězec mapovat na Soundex.
```

```
     */
    public static String soundex(String s) {

        // Algoritmus pracuje s velkými písmeny (éra mainframů).
        String t = s.toUpperCase();

        StringBuffer res = new StringBuffer();
        char c, prev = '?';

        // Hlavní cyklus: najde až 4 znaky, které se mapují.
        for (int i=0; i<t.length() && res.length() < 4 &&
            (c = t.charAt(i)) != ','; i++) {

            // Ověří, zda daný znak je abecedním znakem.
            // Text se již převedl na velká písmena. Algorimtus
            // zpracuje pouze písmena v kódování ASCII;
            // NEPOUŽÍVEJTE Character.isLetter()!
            // Rovněž přeskočí ligatury.
            if (c>='A' && c<='Z' && c != prev) {
                prev = c;

                // První znak se kvůli řazení instaluje nezměněný.
                if (i==0)
                    res.append(c);
                else {
                    char m = MAP[c-'A'];
                    Debug.println("vnitřní", c + " --> " + m);
                    if (m != '0')
                        res.append(m);
                }
            }
        }
        if (res.length() == 0)
            return null;
        for (int i=res.length(); i<4; i++)
            res.append('0');
        return res.toString();
    }
}
```

Kapitola 4

Porovnávání vzorů pomocí regulárních výrazů

4.0 Úvod

Předpokládejte, že jste několik let pracovali s Internetem a veškerou svou korespondenci jste si velmi svědomitě ukládali pro případ, že někdy budete (nebo vaši právníci či státní zastupitelství) potřebovat kopii. Následkem toho máte na disku vyhrazený 50megabajtový oddíl pro uloženou poštu. A matně si vzpomínáte, že někde uvnitř této hromady pošty se nachází e-mailová zpráva od nějaké Anety nebo Anny či snad Andreji? Bohužel si nemůžete vybavit, jak jste zprávu označili a ani kam jste ji uložili. Zřejmě ji budete muset vyhledat.

Ale zatímco někdo z vás půjde a zkusí v textovém editoru otevřít všech 15 000 000 dokumentů, já najdu zprávu pomocí jednoho jednoduchého příkazu. Jakýkoli systém, který poskytuje podporu regulárních výrazů, mi umožní několika způsoby vyhledat text odpovídající zadanému vzoru. Nejsnáze pochopitelné je:

```
Aneta|Anna|Andrea
```

což, jak pravděpodobně tušíte, znamená pouze vyhledat jakoukoli z variant. Stručnější tvar („více přemýšlení, méně psaní“) pro hledání ve všech souborech by byl:

```
An[^ i]
```

Syntaxe vám bude jasná, jakmile si přečtete tuto kapitolu. V krátkosti: písmena „A“ a „n“ odpovídají sama sobě a slouží k nalezení slov, která začínají písmeny „An“, zatímco tajemné `[^ i]` vyžaduje, aby po „An“ následoval jiný znak než mezera (pro vyloučení velmi běžného anglického slova „an“; v případě, že se budou prohledávat i anglicky psané zprávy) a než „i“ (pro vyloučení běžného slova „ani“). Naběhl vám už textový editor? No, to je stejně jedno, protože jsem již našel chybějící soubor. K nalezení odpovědi jsem pouze zadal příkaz:

```
grep 'An[^ i]' *
```

Regulární výrazy neboli zkráceně regexy poskytují stručnou a přesnou specifikaci vzorů, které se mají v textu najít.

Jako další příklad demonstrující sílu regulárních výrazů popřemýšlejte o problému hromadné aktualizace stovek souborů. Když jsem začal s Javou pracovat, odkazy na pole se deklarovaly pomocí syntaxe: `základníTyp názevProměnnéPole[]`. Metoda s polem argumentů, jako metoda `main` každého programu, se například obvykle zapsala jako:

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

Avšak postupem času se stalo zřejmé, že by bylo vhodnější psát deklaraci `základníTyp[] názevProměnnéPole`, tedy například:

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

Jedná se o logičtější styl zápisu v Javě, protože „polo-vost" typu spíše spojuje se samotným typem než s názvem lokálního argumentu. V současnosti připouští kompilátor oba typy zápisu. Všechny výskyty `main` napsané starým způsobem jsem chtěl změnit na nový tvar. K nalezení názvů všech souborů obsahujících starý způsob deklarace `main` jsem prostřednictvím utility *grep*, kterou jsme si popsali dříve, použil vzor `'main(String [a-z]'`. Tento vzor hledá text `main(String` následovaný mezerou a znakem názvu místo počáteční hranaté závorky. Poté jsem chtěl nahradit všechny výskyty `'main(String \([a-z][a-z]*\)\[\]'` za `'main(String[] \1'`. Pro toto nahrazení jsem použil v malém shell-skriptu další unixový nástroj založený na regulárních výrazech – editor *sed* (o použité syntaxi budeme hovořit později v této kapitole). Znovu si připomeňme, že metoda založená na regulárních výrazech byla o řádovou hodnotu rychlejší, než kdybychom totéž dělali interaktivně, dokonce i pomocí přiměřeně výkonného editoru, jakým je `vi` nebo `emacs`. Sami si zkuste použít grafický textový editor.

Při začleňování regulárních výrazů do dalších nástrojů* a dalších jazyků se bohužel změnila jejich syntaxe, takže v předchozích příkladech není uvedena přesně taková syntaxe, jakou byste v Javě použili, ale vyjadřuje stručnost a sílu mechanismu regulárních výrazů.

Jako třetí příklad popřemýšlejte o syntaktické analýze protokolového souboru webového serveru Apache, kde jsou některá pole oddělena pomocí apostrofů, jiná hranatými závorkami a další mezerami. Psaní kódu *ad-hoc*k jeho analýze není rozumné v žádném jazyce, avšak dobře sestavený regulární výraz může v jedné operaci rozdělit řádek na všechna podstatná pole (tento příklad je vyvinut v návodu 4.10).

Stejných časových zisků mohou dosáhnout vývojáři Javy. Java před verzí 1.4 neobsahovala žádné vhodné zařízení pro popisování regulárních výrazů v textu. To je trochu překvapivé, když uvážíme, jak jsou regulární výrazy výkonné, jak jsou všudypřítomné v operačním systému Unix (kde byla Java prvně uvařena) a jak výkonné jsou v moderních skriptovacích jazycích jako *sed*, *awk*, Python a Perl. V tabulce 4.1 je uvedena asi polovina tuctu balíků regulárních výrazů pro Javu.

Dokonce jsem napsal svůj vlastní balík; fungoval dostatečně dobře, ale na produkční použití byl příliš pomalý. Velmi rozšířené jsou také balíky Apache Jakarta RegExp a ORO.

* Příznivci ostatních (neunixových) systémů se nemusí bát, protože pro ně je tady Win32 a jeho balíky. Jedním z nich je open-source balík střídavě označovaný jako CygWin (podle Cygnus Software) nebo GnuWin32 *(http://sources.redhat.com/cygwin/)*. Dalším je vlastní nástroj Microsoftu Unix Services for Windows. Nemáte-li ve svém systému grep, můžete používat můj program `Grep`, uvedený v návodu 4.6,. Mimochodem název *grep* pochází z pradávného příkazu řádkového editoru Unixu `g/RE/p`, příkazu pro globální hledání regulárních výrazů ve všech řádcích editačního bufferu a jejich výstupu – totéž, co dělá program *grep* s řádky v souborech.

Tabulka 4.1. Některé balíky regulárních výrazů Javy

Balík	Poznámky	URL
JDK 1.4 API	Balík `java.util.regex`	`http://java.sun.com/`
Richarda Embersona	Neznámá licence; není udržován	Žádná; balík byl zaslán v roce 1998 na adresu advanced-java@berkeley.edu
regex Iana Darwina	Jednoduchý, ale pomalý. Nekompletní; poučný.	`http://www.darwinsys.com/java/`
Apache Jakarta RegExp (původně jej vytvořil Jonathan Locke)	Licence Apache (jako BSD)	`http://jakarta.apache.org/regexp/`
Apache Jakarta ORO (původně jej vytvořil Daniel Savarese)	Licence Apache, komplexnější než Jakarta RegExp	`http://jakarta.apache.org/oro/`
GNU Java Regexp	Licence GNU LGPL	`http://cacas.org/java/gnu/regexp/`

Od JDK 1.4 je podpora regulárních výrazů přímo vestavěna do standardního runtimu Javy. Výhodou používání balíku JDK 1.4 je jeho integrace s runtimem včetně standardní třídy `java.lang.String` a „nového I/O“ balíku. Kromě této integrace patří balík JDK 1.4 mezi nejrychlejší implementace v Javě. Kód pracující s jinými balíky však stále funguje a po dalších několik let se budeme setkávat se stávajícími aplikacemi, které budou používat některé z těchto balíků, jelikož syntaxe každého z nich je nepatrně odlišná a není nutné ji konvertovat. Avšak při vývoji nových programů byste měli využívat balík regulárních výrazů JDK 1.4.

V prvním vydání této knihy jsme se zaměřili na balík Jakarta RegExp; v tomto vydání se soustředíme na regulární výrazy JDK 1.4 API a nebudeme se zabývat žádným jiným balíkem. Syntaxe samotných regulárních výrazů je vysvětlena v návodu 4.1 a syntaxi API Javy pro použití regulárních výrazů si objasníme v návodu 4.2. Zbývající návody předvádí určité aplikace technologie regulárních výrazů v JDK 1.4.

Přečtěte si také...

Všemi aspekty regulárních výrazů vás provede kniha *Mastering Regular Expressions*, nyní ve svém druhém vydání, kterou napsal Jeffrey E. F. Friedl (O'Reilly). Určité pojednání o regulárních výrazech obsahuje většina knih o Unixu a Perlu; *Unix Power Tools* (O'Reilly) jim věnuje jednu kapitolu.

4.1 Syntaxe regulárních výrazů

Problém

Potřebujete se naučit syntaxi regulárních výrazů JDK 1.4.

Řešení

Podívejte se do tabulky 4.2 na seznam znaků regulárních výrazů.

Diskuse

Tyto vzorové znaky vám umožní specifikovat značně výkonné regulární výrazy. Při sestavování vzorů můžete používat jakoukoli kombinaci běžného textu a *metaznaků* nebo zvláštních znaků uvedených v tabulce 4.2. Lze sestavovat libovolná spojení, která dávají smysl. Například `a+` udává jakýkoli počet výskytů písmene `a`, od jednoho výskytu až po milion nebo nekonečno. Vzor `pan.` odpovídá řetězci `pan` nebo `paní`. A `.*` znamená „jakýkoli počet výskytů libovolného znaku“ a má ve většině interpretů podobný význam jako znak * příkazového řádku. Vzor `\d+` značí jakýkoli počet numerických číslic. `\d{2,3}` značí dvou- nebo tříciferné číslo.

Tabulka 4.2. *Syntaxe metaznaků regulárních výrazů*

Podvýraz	Odpovídá	Poznámky
Obecné		
`^`	Začátek řádku/řetězce	
`$`	Konec řádku/řetězce	
`\b`	Hranice slova	
`\B`	Mimo hranici slova	
`\A`	Začátek celého řetězce	
`\z`	Konec celého řetězce	
`\Z`	Konec celého řetězce (kromě přípustného posledního ukončení řádku)	viz návod 4.9
`.`	Libovolný znak (kromě ukončení řádku)	
`[...]`	„Třída znaků“; libovolný jeden znak z uvedených znaků	
`[^...]`	Libovolný jeden znak, který není uveden	viz návod 4.2
Varianty a seskupování		
`(...)`	Seskupování (skupina)	viz návod 4.3
`|`	Varianty	
`(?:re)`	Nezachytávání závorek	
`\G`	Konec předchozí shody	
`\n`	Zpětný odkaz na *n-tou* skupinu	
Normální (hladové) kvantifikátory		
`{m,n}`	Kvantifikátor pro „*m* až *n* opakování“	viz návod 4.4
`{m,}`	Kvantifikátor pro „*m* nebo více opakování“	
`{m}`	Kvantifikátor pro „přesně *m* opakování“	viz návod 4.10
`{,n}`	Kvantifikátor pro 0 až *n* opakování	
`*`	Kvantifikátor pro 0 nebo více opakování	Zkrácený tvar pro `{0,}`
`+`	Kvantifikátor pro 1 nebo více opakování	Zkrácený tvar pro `{1,}`; viz návod 4.2
`?`	Kvantifikátor pro 0 nebo 1 opakování (např. předloží přesně jednou nebo vůbec)	Zkrácený tvar pro `{0,1,}`

Neochotné (nehladové) kvantifikátory

`{m,n}?`	Neochotný kvantifikátor pro „*m* až *n* opakování"	
`{m,}?`	Neochotný kvantifikátor pro „*m* nebo více opakování"	
`{,n}?`	Neochotný kvantifikátor pro 0 až *n* opakování	
`*?`	Neochotný kvantifikátor: 0 nebo více	
`+?`	Neochotný kvantifikátor: 1 nebo více	viz návod 4.10
`??`	Neochotný kvantifikátor: 0 nebo jedenkrát	

Lačné (velmi hladové) kvantifikátory

`{m,n}+`	Lačný kvantifikátor pro „*m* až *n* opakování"	
`{m,}+`	Lačný kvantifikátor pro „*m* nebo více opakování"	
`{,n}+`	Lačný kvantifikátor pro 0 až *n* opakování	
`*+`	Lačný kvantifikátor: 0 nebo více	
`++`	Lačný kvantifikátor: 1 nebo více	
`?+`	Lačný kvantifikátor: 0 nebo jedenkrát	

Potlačované (escape) znaky a zkratky

`\`	Potlačí znaky (apostrofy): vypne většinu metaznaků, přemění následný abecední znak na metaznak	
`\Q`	Potlačí (opatří apostrofy) všechny znaky až po `\E`	
`\E`	Končí citaci započatou pomocí `\Q`	
`\t`	Tabulátor	
`\r`	Znak návratu (návrat vozíku)	
`\n`	Znak nového řádku	viz návod 4.9
`\f`	Znak nové stránky	
`\w`	Znak ve slově	Pro slovo používejte `\w+`; viz návod 4.10
`\W`	Neslovní znak	
`\d`	Číslice	Pro celé číslo používejte `\d+`; viz návod 4.2
`\D`	Nečíselný znak	
`\s`	Prázdný znak	Mezera, tabulátor apod. podle vymezení v `java.lang.Character.isWhitespace( )`
`\S`	Neprázdný znak	viz návod 4.10

Bloky Unicode (typické vzorky)

`\p{InGreek}`	Znak v řeckém bloku	(jednoduchý blok)
`\P{InGreek}`	Libovolný znak, který není v řeckém bloku	
`\p{Lu}`	Velké písmeno	(jednoduchá kategorie)
`\p{Sc}`	Symbol měny	

Třídy znaků podle standardu POSIX (definované pouze pro US-ASCII)

`\p{Alnum}`	Alfanumerické znaky	`[A-Za-z0-9]`
`\p{Alpha}`	Abecední znaky	`[A-Za-z]`
`\p{ASCII}`	Libovolný znak ASCII	`[\x00-\x7F]`
`\p{Blank}`	Znaky mezery a tabulátoru	
`\p{Space}`	Znaky mezery	`[ \t\n\x0B\f\r]`
`\p{Cntrl}`	Řídicí znaky	`[\x00-\x1F\x7F]`
`\p{Digit}`	Desítkové číslice	`[0-9]`
`\p{Graph}`	Tisknutelné a viditelné znaky (nikoli mezerníky nebo řídicí znaky)	
`\p{Print}`	Tisknutelné znaky	Totéž jako `\p{Graph}`
`\p{Punct}`	Interpunkční znaménka	Některé z `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`
`\p{Lower}`	Malá písmena	`[a-z]`
`\p{Upper}`	Velká písmena	`[A-Z]`
`\p{XDigit}`	Hexadecimální čísla	`[0-9a-fA-F]`

Regulární výrazy hledají shodu kdekoliv v řetězci. Vzory následované *hladovým* kvalifikátorem (jediným typem, který existoval v tradičních unixových regulárních výrazech) spotřebují co možná nejvíce znaků bez kompromitace jakéhokoli následného podvýrazu,* vzory následované *lačným* kvalifikátorem se shodují v nejvyšší možné míře bez ohledu na následující podvýrazy; vzory řízené *neochotným* kvalifikátorem spotřebují k dosažení shody nejmenší možný počet znaků.

Na rozdíl od balíků regulárních výrazů v některých jiných jazycích byl balík JDK 1.4 také od začátku určen ke zpracování znaků Unicode. A standardní escape sekvence Javy `\unnnn` slouží k určení znaku Unicode ve vzoru. K určení vlastností znaků Unicode, například jestli je daný znak mezera, používáme metody `java.lang.Character`.

V rámci pomoci s pochopením principu, jak regulární výrazy fungují, jsem připravil malý program s názvem `REDemo`.**

Do textového pole úplně nahoře (viz obrázek 4.1 dále v textu) zapište vzor regulárního výrazu, který chcete testovat. Všimněte si, že při zadání každého znaku se ověří syntaxe regulárního výrazu, a je-li v pořádku, zobrazí se sousední pole zatržené. Poté můžete nastavit

* Poznámka českého vydavatele: Jinými slovy, snaží se najít nejdelší řetězec vyhovující jím označené části reguláního výrazu, aniž by ohrozil nalezení shody následujících částí výrazu.

** `REDemo` byl inspirován (ale nepoužívá ani kousek jeho kódu) podobným programem, který poskytuje balík Jakarta Regular Expressions, zmíněný v úvodu čtvrté kapitoly. (Poznámka českého vydavatele: Program je umístěn v balíčku s ostatními programy z této kapitoly, avšak není lokalizován.)

přepínač na Match, Find nebo Find All. Match znamená, že celý řetězec musí odpovídat regulárnímu výrazu, kdežto Find znamená, že regulární výraz musí být nalezen někde v řetězci (volba Find All spočítá počet nalezených výskytů). Do pole String zadejte řetězec, s kterým se má regulární výraz porovnat. Vyzkoušejte si to na svém oblíbeném obsahu. Když máte požadovaný regulární výraz, můžete ho vložit do svého programu Javy. Pamatujte, že každý ze znaků, které překladač nebo balík regulárních výrazů JDK 1.4 zpracovává zvláštním způsobem, např. samotné zpětné lomítko, uvozovky a další (viz blok „Pamatujte si!"), musíte označit zpětným lomítkem, aby překladač nezačal daný znak interpretovat, ale opravdu jej do výstupního řetězce vložil.

Pamatujte si!

Pamatujte si, že překladač regulárních výrazů překládá řetězce, které před tím překládal překladač jazyka, a proto obvykle potřebujete dvě úrovně označování (escapování) speciálních znaků včetně zpětného lomítka, uvozovek apod. Například regulární výraz:

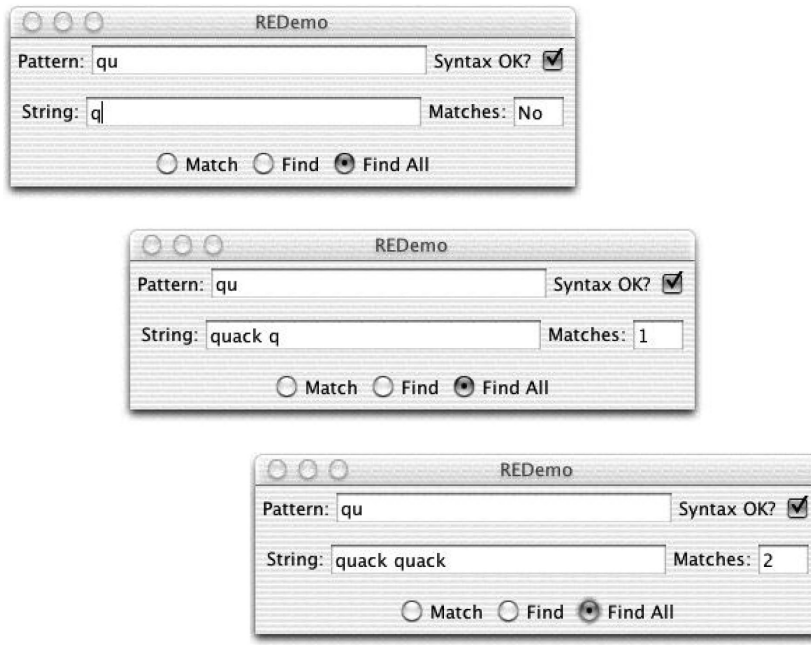
```
"Řekli jste to\."
```

se musí zapsat následovně, aby byl řetězcem jazyka Javy:

```
"\"Řekli jste to\\.\""
```

Ani si již nedokážu přesně vzpomenout, kolikrát jsem u `\d+`, `\w+` a podobných zapomněl přidat další zpětné lomítko!

Na obrázku 4.1 jsem do pole Pattern programu `REDemo` zapsal `do`, což je syntakticky platný regulární výraz: jakékoli obyčejné znaky, které samy vystupují jako regulární výrazy. Tento vý-



Obrázek 4.1. *REDemo s jednoduchým příkladem*

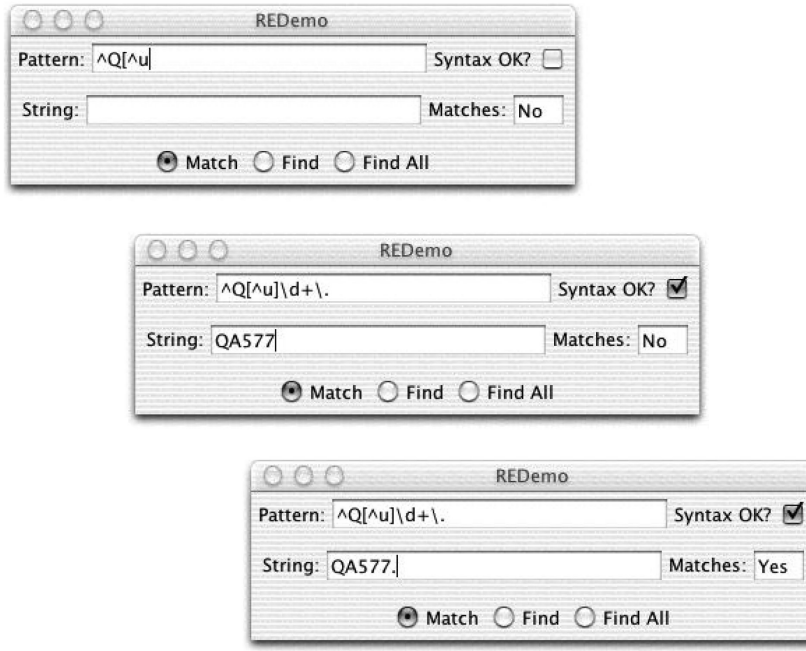
raz tedy hledá písmeno `d` následované písmenem `o`. V okně úplně nahoře jsem zadal do pole pro řetězec (String) pouze písmeno `d`, které danému regulárnímu výrazu neodpovídá. V druhém okně jsem zadal řetězec `doktor` náledovaný znakem `d` přestavujícím začátek druhého slova `doktor`. Protože jsem vybral přepínač Find All, počet ukázal jednu shodu. Jakmile zadám druhé `o`, počet se aktualizuje na dva, jak znázorňuje třetí verze.

Regulární výrazy mohou dělat mnohem víc, než pouze porovnávat znaky. Dvou-znakový regulární výraz `^T` by se například shodoval se začátek řádku (`^`) bezprostředně následovaným velkým písmenem T – tedy s jakýmkoli řádkem začínajícím velkým písmenem T. Je-li na prvním místě velké písmeno T, nezáleží na tom, jestli řádek začíná slovy *Tenké trumpety, Titánské tuby* nebo *Triumfální pozouny* .

Ale to jsme příliš nepokročili. Investovali bychom veškeré toto úsilí do technologie regulárních výrazů pouze proto, abychom mohli dělat něco, co lze provést pomocí metody `startsWith()` třídy `java.lang.String`? Už téměř slyším některé z vás, jak jsou trochu netrpěliví. Zůstaňte klidně sedět! Co když chcete srovnávat nejenom písmeno T na prvním místě, ale také samohlásky (a, e, i, o nebo u) bezprostředně za ním, za nimiž je nějaký počet písmen a vykřičník? Mohli byste to určitě v Javě udělat kontrolováním `startsWith("T")` a `charAt(1) == 'a' || charAt(1) == 'e'` atd.? Ano, ale za dobu než byste to udělali, byste napsali spoustu vysoce specializovaného kódu, který byste nemohli použít v jiné aplikaci. Pomocí regulárních výrazy stačí pouze zadat vzor `^T[aeiou]\w*!`. To znamená, že stejně jako v předchozím příkladě byste zadali `^` a `T`, následované *třídou znaků* uvádějící samohlásky, libovolným počtem slovních znaků (`\w*`) a vykřičníkem.

„Ale počkejte, to ještě není všechno!“, jak říkával můj bývalý velký šéf Yuri Rubinsky. Co když chcete hledaný vzor měnit *za běhu*? Pamatujete si celý tento kód, který jste právě napsali, pro porovnávání `T` ve sloupci 1, samohlásek, nějakých slovních znaků a vykřičníku? Dobře, nadešel čas jej vyhodit. Protože dnes ráno musíme porovnávat `Q`, následované jiným písmenem než `u`, několika číslicemi a tečkou. Zatímco někteří z vás se pustí do psaní nové funkce, ostatní se pouze doloudají k baru s regulárními výrazy, objednají si u barmana vzor `^Q[^u]\d+\.` a jdou pryč.

Dobře `[^u]` znamená „odpovídá libovolnému jednomu znaku kromě znaku `u`”, `\d+` představuje jednu nebo více číslic, `+` je *multiplikátor* nebo *kvalifikátor* značící jeden nebo více výskytů toho, co po něm následuje a `\d` je libovolná jedna číslice. Takže `\d+` znamená jedno, dvou nebo více ciferné číslo. A nakonec `\.`? No, sám o sobě je to metaznak. Většina samostatných metaznaků se vyruší jejich uvozením zpětným lomítkem –zpětné lomítko bývá proto označováno jako únikový nebo-li „escape“, znak. Uvozením metaznaku (například jako .) zpětným lomítkem se vyruší jeho zvláštní význam. Je-li několik vybraných abecedních znaků (např. `n`, `r`, `t`, `s`, `w`) uvozeno zpětným lomítkem, přemění se na metaznaky. Na obrázku 4.2 je znázorněna činnost regulárního výrazu `^Q[^u]\d+\...` Do prvního okna jsem zadal část regulárního výrazu ve tvaru `^Q[^u` a jelikož zde nejsou uzavřeny hranaté závorky, indikátor správnosti syntaxe se vypne. Jakmile dokončíme zápis regulárního výrazu, indikátor se znovu zapne. V druhém okně jsem dokončil regulární výraz a zapsal řetězec `QA577` (u něhož byste mohli předpokládat, že bude zadanému vzoru odpovídat, avšak nebude, protože jsem nenapsal na konci tečku). Do třetího okna jsem napsal tečku, takže signalizace Matches (Shoduje se) ukazuje Yes (Ano).



Obrázek 4.2. *REDemo s příkladem ^Q[^u]\d+\.*

Na regulární výrazy bychom měli nahlížet jako na „malý jazyk“ pro porovnávání vzorů s textem v řetězcích. Pokud jste již rozpoznali návrhový vzor *Interpret*, udělte si prémiové body. API regulárních výrazů je interpret pro porovnávání regulárních výrazů.

Takže nyní byste měli mít alespoň základní představu o tom, jak v praxi regulární výrazy fungují. Ve zbývajících částech této kapitoly vám poskytnu další příklady a objasníme si některá výkonnější témata, jako je například zachycení skupin. Zainteresované čtenáře, které zajímá teorie regulárních výrazů (existuje velké množství teoreticky zajímavých detailů a rozdílů mezi jednotlivými druhy regulárních výrazů), odkážu na knihu *Mastering Regular Expressions*. Mezitím se začněme učit, jak psát v Javě programy, které používají regulární výrazy.

4.2 Použití regulárních výrazů v Javě: Test na vzor

Problém

Jste připraveni začít používat regulární výrazy ve svém kódu k testování, vyskytuje-li se daný vzor v zadaném řetězci.

Řešení

Použijte balík regulárních výrazů Javy `java.util.regex`.

Diskuse

Dobrou zprávou je, že aplikační rozhraní Javy pro regulární výrazy je skutečně jednoduše použitelné. Potřebujete-li pouze zjistit, jestli se daný regulární výraz shoduje s řetězcem, mů-

žete použít příhodnou metodu `boolean matches()` třídy `String`, která jako svůj argument přijímá řetězec se vzorem regulárního výrazu.

```
if (vstupniRetezec.matches(regVyraz)) {
    // shoduje se... něco se s ním udělá...
}
```

Jedná se však o pohodlnou rutinu a pohodlnost vždy něco stojí. Bude-li se regulární výraz vyskytovat v programu více než jednou nebo dvakrát, je mnohem efektivnější sestavit a použít `Pattern` a jeho `Matcher(y)`. Níže je uveden celý program, který sestaví `Pattern` a který slouží k porovnávání vzoru:

```
package cz.darwinsys._04_regexp;

import java.util.regex.*;
/**
 * Jednoduchý příklad pracující s třídou regex.
 */
public class RESimple {
    public static void main(String[] argv) throws PatternSyntaxException {
        String sablona = "^Q[^u]\\d+\\.";
        String vstup = "Následující let má číslo QA777. Není hlášeno zpoždění.";
        Pattern p = Pattern.compile(sablona);
        boolean nalezeno = p.matcher(vstup).lookingAt();
        System.out.println("'" + sablona + "'" +
            (nalezeno ? " odpovídá '" : " neodpovídá '") +
            vstup + "'");
        // Nebude odpovídat, protože QA777. není na počátku řádku.
    }
}
```

Balík `java.util.regex` se skládá ze dvou tříd: `Pattern` a `Matcher`, které poskytují veřejné API uvedené v příkladu 4.1.

Příklad 4.1. *Veřejné API regex*

```
/** Hlavní veřejné API balíku java.util.regex.
 * Připravil  Ian Darwin pomocí javap.
 */

package java.util.regex;

public final class Pattern {
    // Hodnoty příznaků ('nebo' společně).
    public static final int
        UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE,
        DOTALL, UNICODE_CASE, CANON_EQ;
    // Faktorové metody (žádné veřejné konstruktory).
    public static Pattern compile(String sab);
    public static Pattern compile(String sab, int priznaky);
    // Metoda pro nalezeni shody s šablonou z této třídy (Pattern).
    public Matcher matcher(CharSequence vstup);
    // Informativní metody.
```

```
    public String pattern()();
    public int flags();
    // Konvenční metody
    public static boolean matches(String sablona, CharSequence vstup);
    public String[] split(CharSequence vstup);
    public String[] split(CharSequence vstup, int max);
}

public final class Matcher {
    // Akce: najde nebo srovná metody.
    public boolean matches();
    public boolean find();
    public boolean find(int start);
    public boolean lookingAt();
    // "Informace o předchozích metodách  match".
    public int start();
    public int start(int kteraSkupina);
    public int end();
    public int end(int kteraSkupina);
    public int groupCount();
    public String group();
    public String group(int kteraSkupina);
    // Metody resetování
    public Matcher reset();
    public Matcher reset(CharSequence novyVstup);
    // Metody nahrazení
    public Matcher appendReplacement(StringBuffer kde, String novyText);
    public StringBuffer appendTail(StringBuffer kde);
    public String replaceAll(String novyText);
    public String replaceFirst(String novyText);
    // Informativní metody
    public Pattern pattern();
}

/* Třída String znázorňující pouze metody související s regulérními výrazy. */
public final class String {
    public boolean matches(String regv);
    public String replaceFirst(String regv, String novyRet);
    public String replaceAll(String regv, String novyRet)
    public String[] split(String regv)
    public String[] split(String regv, int max);
}
```

Toto API je dostatečně velké na to, abychom si ho trochu vysvětlili. Při porovnávání regulárních výrazů v produkčním programu se běžně používá následující postup:

1. Vytvořit vzor (instance třídy `Pattern`) voláním statické tovární metody `Pattern.compile()`.
2. Pro každý `String` (nebo jiný `CharSequence`), který chcete prohledat, vyžádat od vzoru vyhledávač (instance třídy `Matcher`) voláním `pattern.matcher(CharSequence)`.

3. Zavolat (jednou nebo vícekrát) jednu z vyhledávacích metod získaného vyhledávače (zmíním se o nich později v tomto oddíle).

Rozhraní `CharSequence`, které bylo doplněno v JDK 1.4 do `java.lang`, poskytuje jednoduchý přístup (pouze pro čtení) k objektům obsahujícím kolekci znaků. Standardními implementacemi jsou `String` a `StringBuffer` (popsané v kapitole 3) a nová vstupně-výstupní třída `java.nio.CharBuffer`.

Regulární výraz můžete samozřejmě porovnávat i jinými způsoby, například konvenčními metodami v `Pattern`, nebo dokonce v `java.lang.String`. Například:

```
// StringConvenience.java  -- znázorní konvenční rutinu String  pro "match".
String sablona = ".*Q[^u]\\d+\\..*";
String radek = "Objednat QT300. Nyní!";
if  (radek.matches(sablona)) {
    System.out.println(radek + " odpovida \"" + sablona + "\"");
} else {
    System.out.println("ŽÁDNÁ SHODA");
}
```

Avšak „standardní“ postup pro porovnávání charakterizoval zmíněný tříbodový seznam. V programu, který používá regulární výraz pouze jednou, byste pravděpodobně použili konvenční rutinu `String`; v případě, že s regulárním výrazem pracujeme vícekrát, stojí za to si vyšetřit čas na jeho překlad, protože přeložená verze se provádí rychleji.

Stejně tak má i `Matcher` několik vyhledávacích metod, které poskytují větší flexibilitu než konvenční rutina `match()` třídy `String`. Metody `Matcher` jsou:

`match()`
: Slouží k porovnávání celého řetězce se vzorem; metoda je stejná jako rutina v `java.lang.String`. Před a za vzor jsem musel vložit `.*`, protože porovnává celý `String`.

`lookingAt()`
: Slouží k vyhledání vzoru pouze na začátku řetězce.

`find()`
: Slouží k vyhledání vzoru v řetězci (nemusí se bezpodmínečně jednat o první znak v řetězci) od počátku řetězce nebo, pokud byla metoda volána dříve a uspěla, od prvního znaku, který se neshodoval při předchozím porovnání.

Každá z těchto metod vrací logickou hodnotu pravda (`true`) v případě, že byla nalezena shoda, nebo hodnotu nepravda (`false`), pokud shoda nalezena nebyla. Chcete-li ověřit, jestli daný řetězec odpovídá danému vzoru, stačí pouze zapsat následující kód:

```
Matcher m = Pattern.compile(sab).matcher(radek);
if (m.find()) {
    System.out.println(radek + " odpovida " + sab)
}
```

Možná však také chcete, aby se odpovídají text extrahoval, a to je právě námětem dalšího návodu.

Následující návod pokrývá použití tohoto API. Jako vstupní zdroj příklady zpočátku využívají pouze argumenty typu `String`. Práci s jinými typy `CharSequence` probereme v návodu 4.5.

4.3 Hledání odpovídajícího textu

Problém

Musíte najít text, který odpovídá regulárnímu výrazu.

Řešení

Někdy potřebujete zjistit více, než jen jestli regulární výraz odpovídá řetězci. V editorech a celé řadě jiných nástrojů chcete přesně vědět, které znaky se shodovaly. Pamatujte si, že u kvalifikátorů, jako například `*`, nemusí mít délka odpovídajícího textu žádnou souvislost s délkou vzoru, s nímž se text porovnával. Nepodceňujte mocný vzor `.*`, který odpovídá tisícům nebo milionům znaků, je-li přípustný. Jak jste viděli v předchozím návodu, jestli dané srovnání uspělo, můžete zjistit pouze pomocí metod `find()` nebo `matches()`. Ale v jiných aplikacích budete chtít získat znaky, které se shodují se vzorem.

Po úspěšném vyvolání jedné z výše zmíněných metod, můžete pro získání informací o shodě použít následující „informativní“ metody:

`start(), end()`

Vrací pozici počátečního, resp. poposledního (tj. následujícího za posledním) znaku nalezeného řetězce v původním řetězci.

`groupCount()`

Vrací počet ozávorkovaných skupin, existují-li nějaké; pokud nebyly použity žádné skupiny, vrací hodnotu 0.

`group(int i)`

Vrací znaky, které odpovídají *i-té* skupině, přičemž *i* musí být menší nebo rovno návratové hodnotě `groupCount()`. Skupina 0 představuje celý vzor, takže `group(0)` (nebo pouze `group()`) vrací celý nalezený řetězec.

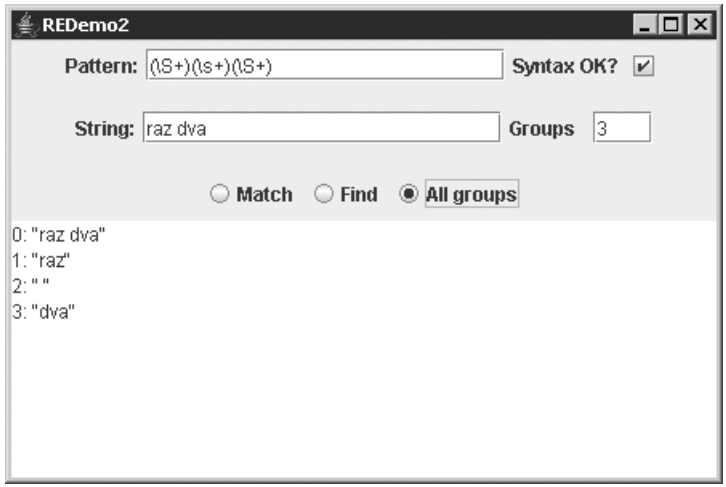
Představa o závorkách nebo „zachytávacích skupinách“ je pro zpracování regulárních výrazů ústřední. Regulární výrazy lze vnořovat do jakékoli úrovně složitosti. Metoda `group(int)` vám umožní vybrat podřetěz odpovídající dané skupině v závorkách. I když nepoužijete žádné závorky, můžete pracovat s celým nalezeným textem jako s nultou zachycenou skupinou. Například:

```
// Část REmatch.java
String sab = "Q[^u]\\d+\\.";
Pattern r = Pattern.compile(sab);
String radek = "Objednat QT300. Nyní!";
Matcher m = r.matcher(radek);
if (m.find()) {
    System.out.println(sab + " odpovídá \"" + m.group(0) +
                       "\" v \"" + radek + "\"");
} else {
     System.out.println("ŽÁDNÁ SHODA");
}
```

Při spuštění programu se vypíše následující:

```
Q[^u]\d+\. odpovídá "QT300." v "Objednat QT300. Nyní!"
```

Rozšířená verze programu `REDemo` uvedená v návodu 4.2 s názvem `REDemo2` nabízí zobrazení všech skupin zachycení v daném regulárním výrazu; jeden příklad je znázorněn na obrázku 4.3.



Obrázek 4.3. *Ukázka funkce programu REDemo2*

Rovněž je možné získat počáteční a koncové indexy a délku textu, který odpovídá vzoru (Připomeňme si, že kvalifikátory, jako např. `\d+` v tomto příkladu, mohou odpovídat libovolnému počtu znaků v řetězci).

Ty můžete společně s metodami `String.substring()`používat následovně:

```
// Část regexsubstr.java -- Vypíše přesně totéž jako REmatch.java2.
Pattern r = Pattern.compile(sab);
String radek = "Objednat QT300. Nyní!";
Matcher m = r.matcher(radek);
if (m.find()) {
    System.out.println(sab + " odpovídá \" +
        radek.substring(m.start(0), m.end(0)) +
        "\" v \" + radek + "\");
    } else {
         System.out.println("ŽÁDNÁ SHODA");
    }
}
```

Dejme tomu, že potřebujete z řetězce vytáhnout několik položek. Pokud je vstup:

```
Soucek, Jan
Andrysek, Jiri
```

a chcete vypsat:

```
Jan Soucek
Jiri Andrysek
```

použijte:

```
// z REmatchTwoFields.java
// Sestaví regulární výraz s rodiči pro "uchopení" jak pole1, tak pole2 .
Pattern r = Pattern.compile("(.*), (.*)");
Matcher m = r.matcher(vstupniRadek);
 if (!m.matches())
    throw new IllegalArgumentException("Špatný vstup: " + vstupniRadek);
System.out.println(m.group(2) + ' ' + m.group(1));
```

4.4 Nahrazení odpovídajícího textu

V předchozím návodu jste viděli, že vzory regulárních výrazů obsahující kvalifikátory mohou s velmi malým počtem metaznaků zachytávat celou řadu znaků. Nyní potřebujeme najít způsob, jak nahradit text, který odpovídá regulárnímu výrazu, aniž by se změnil jiný text před nebo za ním. Mohli bychom to udělat ručně pomocí metody `substring()` třídy `String`. Protože se však jedná o běžný požadavek, API regulárních výrazů JDK 1.4 poskytuje určité substituční metody. Ve všech těchto metodách předáváte text pro nahrazení nebo „pravou stranu“ nahrazení (tento pojem má historické kořeny: v substitučním příkazu textových editorů na bázi příkazového řádku je na levé straně vzor a pravá strana představuje text pro nahrazení). Mezi substituční metody patří:

`replaceAll(novýŘetězec)`

Nahradí všechny shodující se výskyty novým řetězcem.

`appendReplacement(BufferřetězceNovýŘetězec)`

Kopíruje text až po pozici před první shodou a pak přidá daný `novýŘetězec`.

`appendTail(StringBuffer)`

Připojí text za poslední shodu (běžně se používá po `appendReplacement`).

Práce s těmito třemi metodami znázorňuje příklad 4.2.

Příklad 4.2. *ReplaceDemo.java*

```
// Třída ReplaceDemo
// Krátká demonstrace nahrazení: opraví "demon" a jiné
// Pravopisné varianty k opravení neďábelského "daemona".
// Vytvoří vzor regulárního výrazu srovnávající téměř všechny tvary
// (deamon, demon atd.).
String sab = "d[ae]{1,2}mon";
      //libovolná kombinace jednoho či dvou 'a' nebo 'e'

// Testovací řetězec
String vstup = "Unix měl v sobě demony a deamony !";
System.out.println("Vstup: " + vstup);

// Spusťte ho z instance regulárního výrazu a uvidíte, že funguje.
Pattern r = Pattern.compile(sab);
Matcher m = r.matcher(vstup);
System.out.println("Nahradit vše: " + m.replaceAll("daemon"));

// Předvede metodu appendReplacement.
m.reset();
StringBuffer sb = new StringBuffer();
System.out.print("Připojovací metody: ");
```

```
while (m.find()) {
    m.appendReplacement(sb, "daemon");
      // Kopíruj k první shodě, a přidej slovo "daemon".
}
m.appendTail(sb);                       // kopíruj zbytek
System.out.println(sb.toString());
```

Když program spustíme, určitě udělá to, co se od něj očekává:

```
Vstup: Unix měl v sobě demony and deamony!
Nahradit vše: Unix měl v sobě daemony and deamony!
Přípojné metody: Unix měl v sobě daemony and deamony!
```

4.5 Tisk všech výskytů vzoru

Problém

Potřebujete najít všechny řetězce, které odpovídají danému regulárnímu výrazu v jednom nebo více souborech či jiných zdrojích.

Řešení

Tento příklad načítá soubor po jednom řádku. Každou nalezenou shodu vytáhnu z řádku a vypíšu.

Kód vezme metody `group()` z návodu 4.3, metodu `substring` z rozhraní `CharacterIterator` a metodu `match()` z regulárního výrazu a jednoduše je všechny seskládá. Naprogramoval jsem jej tak, aby z daného souboru vytáhl všechny „názvy"; program za chodu vypisuje texty „import", „java", „dokud", „regulární výraz" a tak dále:

```
> jikes +E -d . cz/darwinsys/_04_regexp/ReaderIter.java> java  cz.darwinsys._04_re-
gexp.ReaderIter  cz/darwinsys/_04_regexp/ReaderIter.java
package
cz
darwinsys
demo
regexp
import
java
util
regex
import
java
io
Vyp
ze
souboru
echny
```

V tomto místě jsem běh programu přerušil, abych šetřil papírem. Program lze napsat dvěma způsoby: pomocí tradičního vzoru „po řádku", znázorněného v příkladu 4.3, a pomocí kompaktnějšího tvaru, který používá „nový I/O", znázorněný v příkladu 4.4 (o „novém I/O" balíku budeme hovořit v kapitole 10).

Příklad 4.3. *ReaderIter.java*

```
package cz.darwinsys._04_regexp;

import java.util.regex.*;
import java.io.*;

/**
 * Vypíše ze souboru všechny řetězce, které odpovídají danému vzoru.
 */
public class ReaderIter {
    public static void main(String[] args) throws IOException {
        // Vzor regulárního výrazu.
        Pattern sab = Pattern.compile("[A-Za-z][a-z]+");
        // FileReader (viz kapitola Vstup a výstup).
        BufferedReader r = new BufferedReader(new FileReader(args[0]));

        // Pokusí se srovnat každý řádek vstupu.
        String radek;
        while ((radek = r.readLine()) != null) {
            // Vytáhne a vypíše ho pro každou shodu v řádku.
            Matcher m = sab.matcher(radek);
            while (m.find()) {
                // Nejjednodušší metoda:
                // System.out.println(m.group(0));

                // Získá výchozí pozici textu
                int start = m.start(0);
                // Získá konečnou pozici.
                int konec = m.end(0);
                // Vypíše všechno, co odpovídá.
                // System.out.println("start=" + start + "; konec=" + konec);
                // Použije CharSequence.substring(offset, konec);
                System.out.println(radek.substring(start, konec));
            }
        }
    }
}
```

Příklad 4.4. *GrepNIO.java*

```
package cz.darwinsys._04_regexp;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.regex.*;

/* Program na bázi Grepu používající NIO, ale NENÍ ŘÁDKOVĚ ZALOŽENÝ.
 * Vzor a název souboru musí být  v příkazovém řádku.
 */
public class GrepNIO {
    public static void main(String[] args) throws IOException {

        if (args.length < 2) {
            System.err.println("Použití: GrepNIO sablonovy soubor [...]");
            System.exit(1);
        }
```

```
        Pattern p = Pattern.compile(args[0]);
        for (int i=1; i<args.length; i++)
            process(p, args[i]);
    }

    static void process(Pattern sablona, String jmenoSouboru) throws IOException {

        // Z daného souboru získá FileChannel.
        FileChannel fc = new FileInputStream(jmenoSouboru).getChannel();

        // Zmapuje obsah souboru.
        ByteBuffer buf = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());

        // Dekóduje ByteBuffer na CharBuffer.
        CharBuffer cbuf = Charset.forName("ISO-8859-1").newDecoder().decode(buf);

        Matcher m = sablona.matcher(cbuf);
        while (m.find()) {
            System.out.println(m.group(0));
        }
    }
}
```

Verze NIO, kterou předkládá příklad 4.4, se spoléhá na skutečnost, že NIO `Buffer` může sloužit jako `CharSequence`. Tento program je obecnější v tom, že argument vzoru se vezme z argumentu příkazového řádku. Zadá-li se v příkazovém řádku vzor z předchozího programu, vypíše stejný výstup jako předešlý příklad:

```
>java cz.darwinsys._04_regexp.GrepNIO "[A-Za-z][a-z]+" cz/darwinsys/_04_regexp/
ReaderIter.java
```

Možná vás napadlo použít jako vzor `\w+`. Šlo by to. Jediný rozdíl by byl v tom, že můj vzor hledá správně kapitalizovaná slova, kdežto `\w+` by zahrnulo mezi zachycená slova i identifikátory s velkými písmeny uprostřed názvu jako `mujNazevPromenne`.

Rovněž si všimněte, že verze NIO bude pravděpodobně účinnější, protože na každém řádku vstupu neresetuje `Matcher` na nový vstupní zdroj, jak to dělá `ReaderIter`.

4.6 Vypisování řádků obsahujících vzory

Problém

V jednom nebo více souborech potřebujete vyhledat řádky odpovídající danému regulárnímu výrazu.

Řešení

Napište jednoduchý program podobající se *grepu*.

Diskuse

Jak už bylo zmíněno, když máte balík regex, můžete psát programy na bázi *grepu*. Příklad unixového programu *grep* jsem uvedl dříve; volá se s několika volitelnými argumenty následovanými požadovaným vzorem regulárního výrazu a libovolným počtem názvů souborů. Program vypíše jakýkoli řádek, který obsahuje vzor, což se liší od návodu 4.5, který vypíše pouze samotný shodující se text. Například:

```
grep "[dD]arwin" *.txt
```

hledá řádky obsahující buď darwin nebo Darwin na každém řádku každého souboru, jehož název končí příponou *txt*.*

Příklad 4.5 představuje zdrojový kód první verze programu s názvem `Grep0`. Program načítá řádky ze standardního vstupu a nepřijímá žádné volitelné argumenty, ale zpracovává úplnou množinu regulárních výrazů, které implementuje třída `Pattern` (proto není stejný jako stejnojmenný unixový program). O balíku `java.io` pro vstup a výstup jsme ještě nehovořili (viz kapitola 10), ale jeho použití v tomto příkladu je dostatečně jednoduché na to, abyste jej pravděpodobně mohli vytušit. V on-line zdrojích najdete program `Grep1`, který dělá totéž, ale je lépe strukturovaný (a proto také delší). Dále je v této kapitole v návodu 4.12 uveden program `Grep2`, který pro syntaktickou analýzu voleb příkazového řádku používá moji `GetOpt` (viz návod 2.6).

Příklad 4.5. *Grep0.java*

```
package cz.darwinsys._04_regexp;

import java.io.*;
import java.util.regex.*;

/** Grep0 - Porovná řádky z stdin se vzorem v příkazovém řádku.
 */
public class Grep0 {
    public static void main(String[] args) throws IOException {
        BufferedReader is =
            new BufferedReader(new InputStreamReader(System.in));
        if (args.length != 1) {
            System.err.println("Použití: Grep0 vzor");
            System.exit(1);
        }
        Pattern sab = Pattern.compile(args[0]);
        Matcher porovnavac = sab.matcher("");
        String radek = null;
        while ((radek = is.readLine()) != null) {
            porovnavac.reset(radek);
            if (porovnavac.find()) {
                System.out.println("ODPOVÍDÁ: " + radek);
            }
        }
    }
}
```

4.7 Ovládání velikosti písmen v regulárních výrazech

Problém

Chcete najít text bez ohledu na velikost písmen.

* Shell nebo interpret příkazového řádku na Unixu pochopí, že **.txt* znamená „názvy všech souborů s příponou txt“, ale na systémech, kde není shell dostatečně aktivní nebo chytrý, to za vás udělá normální interpret Javy.

Řešení

Při vtváření vzoru (`Pattern`) zadejte jako druhý argument příznak `Pattern.CASE_INSENSITIVE`, který udává, že porovnání by nemělo brát zřetel na velikost písmen („sloučí" nebo ignoruje rozdíly ve velikosti písmen). V případě, že by váš kód mohl být spouštěn v různých lokalitách (viz kapitola 15), přidejte `Pattern.UNICODE_CASE`. Bez těchto příznaků se porovnávání standardně provádí s ohledem na velikost písmen. Tento příznak (a další) se předají metodě `Pattern.compile()` jako v:

```
// CaseMatch.java
Pattern reNaVelikostiPismenNezalezi =
Pattern.compile(sablona, Pattern.CASE_INSENSITIVE |
Pattern.UNICODE_CASE);
reNaVelikostiPismenNezalezi.matches(vstup); // Provede porovnání bez ohledu na
                                            velikost písmen
```

Tento příznak je třeba předat již při vytváření vzorů; protože objekty `Pattern` jsou neměnné, takže jakmile jsou jednou sestaveny, nelze je změnit.

Celý zdrojový kód tohoto příkladu najdete v on-line zdrojích v programu *CaseMatch.java*.

Příznaky Pattern.compile()

Jako druhý argument metody `Pattern.compile()` lze předat osm příznaků. Je-li třeba předat více hodnot, lze je společně zadat pomocí bitového operátoru OR (`|`). Příznaky seřazené podle abecedy jsou:

`CANON_EQ`

Aktivuje tzv. „kanonickou rovnost", což znamená, že znaky s diakritikou se mohou porovnávat podle svého základního znaku následovaného samostatným diakritickým znakem – např. řetězec „u\u02DA" může odpovídat jak jednoznakovému řetězci „ů", tak dvojznakovému řetězci „u˚" (viz návod 4.8).

`CASE_INSENSITIVE`

Zapne porovnávání nerozlišující velikost písmen (viz návod 4.7).

`COMMENTS`

Způsobí, že prázdná místa a komentáře (od # po konec řádku) budou ve vzoru ignorovány.

`DOTALL`

Dovolí, aby tečka (`.`) odpovídala libovolnému normálnímu znaku nebo novému řádku; standardně znak konce řádku tomuto metaznaku neodpovídá (viz návod 4.9).

`MULTILINE`

Specifikujete víceřádkový režim (viz návod 4.9).

`UNICODE_CASE`

Aktivuje nerozlišování velikosti písmen v kódování Unicode (viz návod 4.7).

`UNIX_LINES`

Učiní `\n` jedinou platnou posloupností „nového řádku" pro režim `MULTILINE` (viz návod 4.9).

4.8 Porovnávání „akcentovaných“ nebo složených znaků

Problém

Chcete, aby se znaky porovnávaly bez ohledu na tvar, v kterém jsou zapsány.

Řešení

Pro „kanonickou rovnost“ přeložte `Pattern` předávající jako hodnotu druhého argumentu příznak `Pattern.CANON_EQ` .

Diskuse

Znaky s diakritikou lze zapsat v různých tvarech. Jako příklad zvažte písmeno `e` s čárkou. Tento znak lze najít v různých tvarech v textu Unicode, například jako jediný znak `é` (znak Unicode \u00e9) nebo jako dvouznaková posloupnost `e´` (e následované znakem pro čárku nad písmenem – \u0301). Balík regulárních výrazů má proto volbu pro „kanonické porovnávání“, která zpracuje všechny tvary jako ekvivalenty, abyste mohli porovnávat takové znaky bez ohledu na to, jaký z možných rozmanitých, „plně rozložených” tvarů byl k jeho zapsání použit. Tato volba se aktivuje předáním `CANON_EQ` jako (jednoho) z příznaků ve druhém argumentu `Pattern.compile()`. Následující program znázorňuje, jak lze pomocí `CANON_EQ` porovnávat několik tvarů:

```
package cz.darwinsys._04_regexp;

import java.util.regex.*;

/**
 * CanonEqDemo - předvádí použití Pattern.CANON_EQ porovnáváním různých způsobů
 * zápisu španělského slova pro "rovná se" a pomocí regulárních výrazů
 * zjistí, zda jsou považovány za stejné.
 */
public class CanonEqDemo {
    public static void main(String[] args) {
        String retSablony = "\u00e9gal"; // égal
        String[] vstup = {
                "\u00e9gal",  // égal - tento se lépe shodoval :-)
                "e\u0301gal", // e + "čárka nad písmenem vpravo"
                "e\u0300gal", // e + "čárka nad písmenem vlevo"
                "e'gal",      // e + apostrof
                "e\u00b4gal", // e + "čárka vpravo z Latin-1"
        };
        Pattern sablona = Pattern.compile(retSablony, Pattern.CANON_EQ);
        for (int i = 0; i < vstup.length; i++) {
            if (sablona.matcher(vstup[i]).matches()) {
                System.out.println(retSablony + " odpovídá vstupu " + vstup[i]);
            } else {
                System.out.println(retSablony + " neodpovídá vstupu " + vstup[i]);
            }
        }
    }
}
```

Když spustíte tento program na JDK 1.4 nebo novějších, správně se shoduje znak é a znak e následovaný čárkou nad písmenem vpravo a ostatní znaky se zamítnou. Některé ze zamítnutých znaků však bohužel na tiskárně vypadají jako čárka nad písmenem (viz poslední řetězec), ale nejsou považovány za diakritická znaménka.

```
égal odpovídá vstupu égal
égal odpovídá vstupu e?gal
égal neodpovídá vstupu e?gal
égal neodpovídá vstupu e'gal
égal neodpovídá vstupu e´gal
```

Podrobnější informace viz mapy znaků na webových stránkách kódování Unicode *http://www.unicode.org/*.

4.9 Porovnávání nových řádků v textu

Problém

Potřebujete najít konce řádků v textu.

Řešení

Použijte `\n` nebo `\r`.

Podívejte se rovněž na konstantu příznaků `Pattern.MULTILINE`, při jejímž zadání odpovídá metaznak ^ začátku řádku a $ konci řádku. Nezadáte-li tuto konstantu v parametru, budou uvedené znaky odpovídat pouze začátku a konci celé sekvence.

Diskuse

I když řádkově orientované nástroje z Unixu, například *sed* a *grep,* porovnávají regulární výrazy řádek za řádkem (tj. každý řádek zvlášť), všechny nástroje to nedělají. Textový editor *sam* od Bell Laboratories byl prvním interaktivním nástrojem, o kterém jsem se dozvěděl, že dovoluje víceřádkové regulární výrazy; krátce poté následoval skriptovací jazyk Perl. V API Javy standardně nemá znak konce řádku žádný zvláštní význam. Metoda `BufferedReader readLine()` běžně vyjímá jakékoli znaky konce řádku, které najde. Načítáte-li hodně znaků pomocí nějaké jiné metody než `readLine()`, možná máte ve svém textovém řetězci určité množství posloupností `\n`, `\r`, nebo `\r\n`.* Všechny tyto posloupnosti se běžně zpracují jako ekvivalenty `\n`. Chcete-li, aby odpovídalo pouze `\n`, předejte metodě `Pattern.compile()` příznak `UNIX_LINES`.

V Unixu se `^` a `$` běžně používá k označení začátku a konce řádku (v tomto pořadí). Metaznaky regulárních výrazů `^` a `$` v API Javy ignorují ukončení řádků a odpovídají pouze začátku a konci celého řetězce. Předáte-li však metodě `Pattern.compile()` příznak `MULTILINE`, tyto výrazy označují místo bezprostředně po nebo bezprostředně před ukončením řádku; `$` také odpovídá úplnému konci řetězce. Protože ukončení řádku je pouze běžný znak, můžete jej porovnávat pomocí . nebo podobným výrazem a pokud chcete přesně vědět, kde se nachází, označíte jej ve vzoru pomocí `\n` nebo `\r`. Jinými slovy pro API Javy je znak nového řádku pouze dalším znakem bez zvláštního významu; viz oddíl Příznaky `Pattern.compile()`. Vyhledání konce řádku je znázorněno v příkladu 4.6.

* Nebo několik souvisejících znaků Unicode včetně znaků konce řádku (`\u0085`), oddělovače řádků (`\u2028`) a oddělovače odstavců (`\u2029`).

Příklad 4.6. *NLMatch.java*

```
package cz.darwinsys._04_regexp;

import java.util.regex.*;

/**
 * Znázorňuje porovnávání ukončení řádků pomocí třídy regex.
 * @author Ian F. Darwin, ian@darwinsys.com
 * @version $Id: ch04,v 1.4 2004/05/04 20:11:27 ian Exp $
 */
public class NLMatch {
    public static void main(String[] argv) {

        String vstup = "Po celý den sním o strojích\nvíce strojích";
        System.out.println("VSTUP: " + vstup);
        System.out.println();

        String[] sab = {
            "strojích.více strojích",
            "strojích$"
        };

        for (int i = 0; i < sab.length; i++) {
            System.out.println("VZOR " + sab[i]);

            boolean nalezeno;
            Pattern p1l = Pattern.compile(sab[i]);
            nalezeno = p1l.matcher(vstup).find();
            System.out.println("STANDARDNI porovnani " + nalezeno);

            Pattern pml = Pattern.compile(sab[i],
                Pattern.DOTALL|Pattern.MULTILINE);
            nalezeno = pml.matcher(vstup).find();
            System.out.println("Vice radkove porovnani " + nalezeno);
            System.out.println();
        }
    }
}
```

Spustíte-li tento kód, první vzor (se zástupným znakem) se vždy shoduje, zatímco druhý vzor (s `$`) se shoduje, pouze když se nastaví `MATCH_MULTILINE`.

```
> java  -Dfile.encoding=Cp852 cz.darwinsys._04_regexp.NLMatch
VSTUP: Po celý den sním o strojích
více strojích

VZOR strojích.více strojích
STANDARDNI porovnani false
Vice radkove porovnani true

VZOR strojích$
STANDARDNI porovnani true
Vice radkove porovnani true
```

4.10 Program: Syntaktická analýza protokolového souboru serveru Apache

Webový server Apache je celosvětovým leaderem na poli webových serverů a jeho existence sahá téměř až k samotnému vzniku webu. Jedná se o jeden z nejznámějších open-source projektů a patří k mnoha projektům, které podporuje Apache Foundation. Název Apache je však slovní hříčka vztahující se k původu serveru; vývojáři Apache nejprve pracovali s volně dostupným serverem NCSA a upravovali nebo záplatovali ho tak dlouho, dokud nefungoval podle jejich představ. Když byl dostatečně odlišný od původního serveru, bylo třeba vymyslet nový název. A tehdy vznikl název Apache, protože šlo o „záplatovaný" server (v angličtině „a patchy" server). Jedním místem, kde prostupuje tato záplatovanost, je formát protokolového souboru (log file). Popřemýšlejte o následujícím záznamu:

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)"
```

Formát souboru byl očividně určen pro prohlížení lidským okem, avšak nikoli pro jednoduchou syntaktickou analýzu. Problém spočívá v použití různých oddělovačů: hranaté závorky pro datum, uvozovky pro řádek požadavku a mezer, které jsou rozházeny po celém souboru. Popřemýšlejte o využití třídy `StringTokenizer`; možná byste ji byli schopni zprovoznit, ale stálo by vás to hodně času. Následující podivuhodný regulární výraz* však zjednoduší syntaktickou analýzu:

```
^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(.+?)" (\d{3}) (\d+) "([^"]+)"
"([^"]+)"
```

Možná shledáte poučným nahlédnutí zpět do tabulky 4.2 a prozkoumání celé syntaxe uvedeného výrazu. Zejména si všimněte použití nehladového kvantifikátoru `+?` v `\"(.+?)\"` k porovnání uvozeného řetězce; nemůžete použít pouze `+`, protože tím by se porovnávalo příliš mnoho (až k uvozovce na konci řádku). V příkladu 4.7 je znázorněn kód k vytáhnutí různých polí, například IP-adresy, požadavku, odkazující URL a verze prohlížeče.

Příklad 4.7. *LogRegExp.java*

```
package cz.darwinsys._04_regexp;

import java.util.regex.*;

/**
 * Syntakticky analyzuje protokolový soubor pomocí regulárních výrazů.
 */
public class LogRegExp implements LogExample {

    public static void main(String argv[]) {

        String logVstupniSablona =
            "^([\\d.]+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\]" +
            " \"(.+?)\" (\\d{3}) (\\d+) \"([^\"]+)\" \"([^\"]+)\"";

        System.out.println("Použití vzoru regulárního výrazu:");
        System.out.println(logVstupniSablona);
```

* Při pohledu na výraz byste si mohli myslet, že drží nějaký druh světového rekordu za složitost v soutěži regulárních výrazů, ale jsem si jistý, že byl mnohokrát překonán.

```
        System.out.println("\nVstupní řádek je:\n" + logVstupniRadek + "\n");

        Pattern p = Pattern.compile(logVstupniSablona);
        Matcher porovnavac = p.matcher(logVstupniRadek);
        if (!porovnavac.matches() ||
                POCET_POLI != porovnavac.groupCount()) {
            System.err.println("Neplatný záznam protokolu " +
                    "(nebo potíže s regulárním výrazem?):");
            System.err.println(logVstupniRadek);
            return;
        }
        System.out.println("IP-adresa:      " + porovnavac.group(1));
        System.out.println("Datum a čas:    " + porovnavac.group(4));
        System.out.println("Požadavek:      " + porovnavac.group(5));
        System.out.println("Odezva:         " + porovnavac.group(6));
        System.out.println("Odeslané bajty: " + porovnavac.group(7));
        if (!porovnavac.group(8).equals("-"))
            System.out.println("Referer:    " + porovnavac.group(8));
        System.out.println("Prohlížeč:      " + porovnavac.group(9));
    }
}
```

Klauzule `implements` je pro rozhraní, které pouze definuje vstupní řetězec; v ukázce sloužila k porovnání režimu regulárních výrazů s použitím `StringTokenizer`. Zdrojový kód obou verzích najdete v on-line zdrojích pro tuto kapitolu. Spuštění programu na výše uvedený vzorový vstup vytvoří následující výstup (regulární výraz i vstupní řádek jsou zalomeny přes několik řádků):

```
Použití vzoru regulárního výrazu:
^([\d.]+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(.+?)" (\d{3}) (\d+) "([^"]+)"
"([^"]+)"

Vstupní řádek je:
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0"
200 10450 "-" "Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)"

IP-adresa:      123.45.67.89
Datum a čas:    27/Oct/2000:09:27:09 -0400
Požadavek:      GET /java/javaResources.html HTTP/1.0
Odezva:         200
Odeslané bajty: 10450
Prohlížeč:      Mozilla/4.6 [en] (X11; U; OpenBSD 2.8 i386; Nav)
```

Program pomocí jednoho volání `matcher.matches()` úspěšně analyzoval celý formát protokolového souboru.

4.11 Program: Dolování dat

Dejme tomu, že já jako autor chci sledovat, jak se má kniha prodává ve srovnání s ostatními. Tyto informace lze získat zdarma – stačí klepnout na stránku pro mou knihu na nějakém z hlavních internetových knihkupectví, přečíst z obrazovky čísla hodnocení prodeje a zapsat údaje do souboru – to je však příliš zdlouhavé. Jak jsem napsal v knize, která se hledá v tomto příkladu: „Na získávání relevantních informací ze souborů máme počítače; lidé by neměli být nuceni dělat takové světské úkoly.“ Tento program používá k výběru hodnoty ze stránky HTML na hypotetických webových stránkách *RychleKnihkupectvi.web* API regulárních výrazů a zejména vyhledávání konců řádku. Program rovněž načítá z objektu URL (viz návod 18.7).

Vzor pro hledání se podobá následujícímu (mějte na paměti, že kód HTML se může kdykoli změnit, proto chci udržet vzor relativně obecný):

```
<b>RychleKnihkupectvi.web-Hodnocení prodeje: </b>
26,252
</font><br>
```

Protože se vzor může rozšířit přes více řádků, nepracuji s tradičním modelem *řádek-po-řádku*, ale načítám celou webovou stránku z URL do jediného dlouhého řetězce pomocí mé metody `FileIO.readerToString()` (viz návod 10.8). Poté prostřednictvím externího programu (viz návod 26.1) nakreslím graf; ten by se mohl (a měl) změnit tak, aby používal grafické programy Javy (určitá vodítka najdete v návodu 13.13). Celý program je uveden v příkladu 4.8.

Příklad 4.8. *BookRank.java*

```
package cz.darwinsys._04_regexp;

// Nejsou znázorněny standardní importy.
import com.darwinsys.io.FileIO;
import com.darwinsys.util.FileProperties;

/** Graf hodnocení prodeje knihy v daném internetovém knihkupectví.
 * @author Ian F. Darwin, http://www.darwinsys.com/, autor knihy Java Cookbook,
 *    původně docela doslovně přeložen z Perlu do Javy.
 * @author Patrick Killelea <p@patrick.net>: původní verze v jazyce Perl
 *    z druhého vydání jeho knihy "Web Performance Tuning".
 * @version $Id: ch04,v 1.4 2004/05/04 20:11:27 ian Exp $
 */
public class BookRank {
    public final static String DATA_SOUBOR = "book.sales";
    public final static String GRAF_SOUBOR = "book.png";

    /** Získá z webové stránky hodnocení prodeje a zapíše jej. */
    public static void main(String[] args) throws Exception {

        Properties p = new FileProperties(
            args.length == 0 ? "bookrank.properties" : args[1]);
        String nazev = p.getProperty("nazev", "ZADNY NÁZEV VE VLASTNOSTECH");
        // Adresa URL musí úplně na konci obsahovat "isbn=" .
        String url = p.getProperty("url", "http://test.ing/test.cgi?isbn=");
        // Desetimístné číslo ISBN knihy.
        String isbn  = p.getProperty("isbn", "0000000000");
        // Vzor regulárního výrazu (MUSÍ mít JEDNU skupinu zachycení pro číslo).
        String sablona = p.getProperty("vzor", "Hodnocení: (\\d+)");

        // Hledání něčeho takového ve vstupu:
        //     <b>RychleKnihkupectvi.web - Hodnocení prodeje: </b>
        //     26,252
        //     </font><br>

        Pattern r = Pattern.compile(sablona);

        // Otevře URL a získá z ní Reader.
        BufferedReader is = new BufferedReader(new InputStreamReader(
            new URL(url + isbn).openStream()));
        // Načte adresu URL hledající informace hodnocení jako
        // jediný dlouhý řetěz, takže můžete porovnávat regulární výraz ve více řádcích.
```

```
        String vstup = FileIO.readerToString(is);
        // System.out.println(vstup);

        // Je-li nalezen, připojí se k souboru prodejních dat.
        Matcher m = r.matcher(vstup);
        if (m.find()) {
            PrintWriter pw = new PrintWriter(
                new FileWriter(DATA_SOUBOR, true));
            String datum = // 'datum +'%m %d %H %M %S %Y'`;
                new SimpleDateFormat("MM dd hh mm ss yyyy ").
                format(new Date());
            // Paren 1 je číslice (a možná ','s) která se shoduje; odstranit čárku.
            Matcher zadnaCarka = Pattern.compile(",").matcher(m.group(1));
            pw.println(datum + zadnaCarka.replaceAll(""));
            pw.close();
        } else {
            System.err.println("VAROVÁNÍ: sablona `" + sablona +
                "' nenalezen v  `" + url + isbn + "'!");
        }

        // Bez ohledu na to, jestli se data naleznou či nikoli, pomocí
        // externího kreslicího programu se vytvoří graf ze všech údajů
        // Mohli byste použít gnuplot, R, jakýkoli jiný program "porovnat-graf".
        // Ještě lépe: použijte jedno z kreslicích aplikačních rozhraní Javy.

        String gnuplot_prikaz  =
            "set term png\n" +
            "set output \"" + GRAF_SOUBOR + "\"\n" +
            "set xdata time\n" +
            "set ylabel \"Prodejní hodnocení knihy\"\n" +
            "set bmargin 3\n" +
            "set logscale y\n" +
            "set yrange [1:60000] reverse\n" +
            "set timefmt \"%m %d %H %M %S %Y\"\n" +
            "set \"" + DATA_SOUBOR +
                "\" using 1:7 title \"" + title + "\" with lines\n"
        ;

        Process proc = Runtime.getRuntime().exec("/usr/local/bin/gnuplot");
        PrintWriter gp = new PrintWriter(proc.getOutputStream());
        gp.print(gnuplot_prikaz);
        gp.close();
    }
}
```

4.12 Program: Úplný Grep

Ukázali jsme si, jak fungují balíky regulárních výrazů, je tedy čas napsat `Grep2`, plnohodnotnou verzi programu na porovnávání řádků se syntaktickou analýzou voleb. V tabulce 4.3 jsou uvedeny některé typické volby příkazového řádku, které by mohla zahrnovat unixová implementace *grepu*.

Tabulka 4.3. *Volby programu příkazového řádku Grep*

Volba	Význam
`-c`	Pouze počet: nevytiskne řádky, pouze je spočítá
`-C`	Kontext; vytiskne nějaké řádky nad a pod každým řádkem, který se shoduje (v této verzi není implementována; ponechána jako cvičení pro čtenáře)
`-f vzor`	Nepřijímá vzor z příkazového řádku, ale ze souboru uvedeného za `-f`
`-h`	Potlačí tisk názvu souboru před řádky
`-i`	Ignoruje velikost písmen
`-l`	Uvede pouze názvy souborů: nevytiskne řádky, pouze názvy, které v nich najde
`-n`	Vytiskne před odpovídajícími řádky čísla řádku
`-s`	Potlačí tisk určitých chybových hlášení
`-v`	Inverzní: vytiskne pouze řádky, které NEODPOVÍDAJÍ vzoru

O třídě `GetOpt` jsme hovořili v návodu 2.6. V tomto příkladu slouží k ovládání činnosti aplikačního programu. Mohli bychom jako obvykle zrušit předávání spousty informací do konstruktoru, protože `main()` běží ve statickém kontextu, ale hlavní směr naší aplikace nikoli. Pro předávání všech argumentů `true/false` (`true` pro tisk čísel řádků a `false` pro tisk názvů souborů atd.) používáme instanci třídy `BitSet`, protože máme mnoho voleb a nebylo by příhodné udržovat rozšiřující se seznam voleb při přidávání nových funkcionalit do programu. (O kolekcích budeme hovořit v kapitole 7.) Třída `BitSet` se v mnohém podobá třídě `Vector` (viz návod 7.3), ale je určena k uchovávání pouze booleovských hodnot a výborně se hodí pro ošetřování argumentů příkazového řádku.

Program v podstatě pouze čte řádky, porovná v nich vzor, a pokud je nalezena shoda (nebo nenalezena u volby `-v`), vypíše řádek (a volitelně také nějaké další informace). Nyní, když jsme si vše vysvětlili, se můžeme podívat na kód v příkladu 4.9.

Příklad 4.9. *Grep2.java*

```
package cz.darwinsys._04_regexp;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import java.util.BitSet;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import cz.darwinsys.lang.GetOpt;

/** Program  ve stylu programu grepspouštěný z příkazového řádku.
 *  Akceptuje některé volby a očekává vzor a seznam textových souborů.
 */
public class Grep2 {
    /** Vzor, který hledáme */
    protected Pattern vzor;
    /** Hledac predchozího vzoru */
    protected Matcher hledac;
```

```
/** Reader pro aktuální soubor. */
protected BufferedReader d;
/** Netiskne pouze počítá řádky? */
protected boolean pouzePocitat = false;
/** Ignorujeme velikost písmen? */
protected boolean nerozlisovatVelikostPismen = false;
/** Potlačujeme tisk názvů souborů? */
protected boolean netisknoutJmenoSouboru = false;
/** Uvádíme pouze názvy souborů, které se shodují? */
protected boolean pouzeVypis = false;
/** Tiskneme čísla řádků? */
protected boolean cislovano = false;
/** Potlačujeme hlášení chyb? */
protected boolean potichu = false;
/** Tiskneme pouze řádky, které se NESHODUJÍ? */
protected boolean inVert = false;

/** Pro každý vzor sestaví objekt Grep  a spustí ho
 * na všechny vstupní soubory uvedené v argv.
 */
public static void main(String[] argv)
// throws RESyntaxException
{

    if (argv.length < 1) {
        System.err.println("Použití: Grep2 vzor [nazevsouboru...]");
        System.exit(1);
    }
    String sablona = null;

    GetOpt go = new GetOpt("cf:hilnsv");
    BitSet args = new BitSet();

    char c;
    while ((c = go.getopt(argv)) != 0) {
        switch(c) {
            case 'c':
                args.set('C');
                break;
            case 'f':
                try {
                    BufferedReader b = new BufferedReader
                             (new FileReader(go.optarg()));
                    sablona = b.readLine();
                    b.close();
                } catch (IOException e) {
                    System.err.println("Nelze načíst soubor vzoru " +
                             go.optarg());
                    System.exit(1);
                }
                break;
            case 'h':
                args.set('H');
                break;
            case 'i':
                args.set('I');
                break;
```

```
            case 'l':
                args.set('L');
                break;
            case 'n':
                args.set('N');
                break;
            case 's':
                args.set('S');
                break;
            case 'v':
                args.set('V');
                break;
        }
    }

    int ix = go.getOptInd();

    if (sablona == null)
        sablona = argv[ix-1];

    Grep2 pg = new Grep2(sablona, args);

    if (argv.length == ix)
        pg.process(new InputStreamReader(System.in), "(standardní vstup)");
    else
        for (int i=ix; i<argv.length; i++) {
            try {
                pg.process(new FileReader(argv[i]), argv[i]);
            } catch(Exception e) {
                System.err.println(e);
            }
        }
}

/** Sestaví objekt Grep2.
 */
public Grep2(String sab, BitSet args)  {
    // zkompiluje regulární výraz
    if (args.get('C'))
        pouzePocitat = true;
    if (args.get('H'))
        netisknoutJmenoSouboru = true;
    if (args.get('I'))
        nerozlisovatVelikostPismen = true;
    if (args.get('L'))
        pouzeVypis = true;
    if (args.get('N'))
        cislovano = true;
    if (args.get('S'))
        potichu = true;
    if (args.get('V'))
        inVert = true;
    int modRozlisovaniVelikostiPismen =
        nerozlisovatVelikostPismen
            ? Pattern.UNICODE_CASE | Pattern.CASE_INSENSITIVE
            : 0;
    vzor = Pattern.compile(sab, modRozlisovaniVelikostiPismen);
```

```
        hledac = vzor.matcher("");
    }

    /** Provede skenovací práce jednoho souboru
     * @param    vstSoubor   Reader    Reader objekt již otevřen
     * @param    jmenoSouboru String    Název vstupního souboru
     */
    public void process(Reader vstSoubor, String jmenoSouboru) {

        String radek;
        int odpovida = 0;

        try {
           d = new BufferedReader(vstSoubor);

            while ((radek = d.readLine()) != null) {
                hledac.reset(radek);
                if (hledac.lookingAt()) {
                    if (pouzePocitat)
                        odpovida++;
                    else {
                        if (!netisknoutJmenoSouboru)
                            System.out.print(jmenoSouboru + ": ");
                        System.out.println(radek);
                    }
                } else if (inVert) {
                    System.out.println(radek);
                }
            }
            if (pouzePocitat)
                System.out.println(odpovida + " nalezených v " + jmenoSouboru);
            d.close();
        } catch (IOException v) { System.err.println(v); }
    }
}
```

Kapitola 5

Čísla

5.0 Úvod

Čísla tvoří základní část téměř jakéhokoli výpočtu. Používají se pro indexy, pole, teploty, platy, hodnocení a nekonečné množství dalších věcí, ačkoliv práce s nimi není tak jednoduchá, jak by se na první pohled mohlo zdát.

Jaká přesnost čísel s plovoucí desetinnou čárkou je dostatečná? Jak náhodná jsou náhodná čísla? Co ve skutečnosti tvoří číslo u řetězců, které by měly obsahovat číslo? Java obsahuje několik primitivních datových typů, s jejichž pomocí lze znázornit čísla. Tyto typy jsou shrnuty v tabulce 5.1. Všimněte si, že na rozdíl od jazyků, jako je C nebo Perl, které neudávají velikost nebo přesnost číselných typů, Java – kvůli dosažení přenositelnosti – definuje tyto typy přesně a tvrdí, že jsou stejné na všech platformách.

Tabulka 5.1. *Číselné typy*

Primitivní typ	Obalová třída	Velikost primitivního typu (v bitech)	Obsah
byte	Byte	8	celé číslo se znaménkem
short	Short	16	celé číslo se znaménkem
int	Integer	32	celé číslo se znaménkem
long	Long	64	celé číslo se znaménkem
float	Float	32	číslo s plovoucí desetinnou tečkou IEEE-754
double	Double	64	číslo s plovoucí desetinnou tečkou IEEE-754
char	Character	16	znak Unicode bez znaménka

Jak si můžete všimnout, Java poskytuje číselné typy téměř pro všechny účely. Pro znázornění celých čísel s různou velikostí existují čtyři velikosti celých čísel se znaménkem. Pro aproximaci reálných čísel pak existují dvě velikosti čísel s plovoucí desetinnou čárkou. Java má také typ, který je výslovně určen pro znázornění znaků Unicode a operace s nimi.

Když načítáte řetězec z uživatelského vstupu nebo textového souboru, musíte jej převést na příslušný typ. Obalové třídy, uvedené v druhém sloupci, plní několik funkcí, ale jednou z nejdůležitějších je právě poskytování této základní převáděcí funkcionality. Tím nahrazují rodinu funkcí *atoi*/*atof* programátorů v C a číselné argumenty pro *scanf*.

Chcete-li zkusit jiný způsob, můžete převést jakékoli číslo (ve skutečnosti cokoli v Javě) na řetězec pouze pomocí operace zřetězení. Chcete-li mít o trochu větší kontrolu nad formátováním čísel, v návodu 5.8 vám ukážeme, jak používat některé konverzní rutiny obalových typů. Rovněž vám předvedeme práci s `NumberFormat` a souvisejícími třídami, abyste získali úplnou kontrolu nad formátováním.

O*balové třídy*, jak naznačuje jejich název, slouží k „zabalení" čísla do objektu, protože mnoho částí standardního API se definuje na základě objektů. Později si v návodu 10.16 ukážeme použití objektu `Integer` k uložení hodnoty `int` do souboru pomocí *serializace objektu* a následný výběr hodnoty.

Ještě jsme se však nezmínili o problémech týkajících se čísel s plovoucí desetinnou čárkou. Reálná čísla, jak si možná vzpomínáte, jsou čísla se zlomkovou částí. Existuje nekonečně možných reálných čísel. Číslo s plovoucí desetinnou čárkou – s jehož pomocí se počítače snaží přiblížit reálným číslům – není stejné jako reálné číslo. Počet čísel s plovoucí desetinnou čárkou je omezený pouze na 2^32 různých bitových kombinací pro typy `float` a 2^64 pro typy `double`. Většina reálných hodnot tedy přesně neodpovídá hodnotám s plovoucí desetinnou čárkou. Vypsání reálného číslo 0,3 funguje správně, jak můžete vidět v následujícím příkladu:

```
System.out.println("Reálná hodnota 0,3 je " + 0.3);
```

jehož výsledkem bude následující výstup*:

```
Reálná hodnota 0,3 je 0.3
```

Ale rozdíl mezi reálnou hodnotou a hodnotou jejího přiblížení uloženého v čísle s plovoucí desetinnou čárkou může narůstat, pokud se hodnota používá ve výpočtech; tento jev bývá označován jako *zaokrouhlovací chyba*. Budeme-li pokračovat v předchozím příkladu a vynásobíme reálnou hodnotu 0,3 třemi, získáme následující:

```
Reálná hodnota 0,3 krát 3 se rovná 0.89999999999999991
```

Překvapilo vás to? Ještě více překvapující je skutečnost, že tentýž výstup získáte na jakékoli vyhovující implementaci Javy. Spustil jsem příklad na tak různorodých počítačích, jako je Pentium se systémem OpenBSD, Pentium s Windows a JDK od Sunu a na systému Mac OS X s JDK 1.4.1. Vždy tatáž odpověď.

A co náhodná čísla? Jak jsou náhodná? Pravděpodobně jste již zaslechli výraz „pseudonáhodná čísla". Všechny konvenční generátory náhodných čísel bez ohledu na to, jestli jsou napsána v jazyce Fortran, C nebo Java, generují pseudonáhodná čísla. To znamená, že ve skutečnosti se nejedná o náhodná čísla! Skutečnou náhodnost generuje pouze speciálně sestavený hardware: například analogový zdroj růžového šumu** připojený k analogově číslicovému převodníku.*** Tak asi nevypadá vaše průměrné PC!

Generátory pseudonáhodných čísel (zkráceně PRNG – Pseudo Random Number Generators) jsou však pro většinu účelů dostačující, proto je také používáme. Generátor pseudonáhodných

---

* Poznámka českého vydavatele: Připomínáme, že v programech se místo desetinné čárky píše desetinnná tečka. Tato záměna bývá častou začátečnickou chybou.

** Poznámka českého vydavatele: Dokonalý šum bývá označován jako bílý – jsou v něm zastoupeny všechny frekvence obdobně jako v bílém světle. Ve skutečných zařízeních však nemohou růst frekvence do nekonečna. Protože v něm nejvyšší frekvence chybějí, říkají o něm fyzici, že je posunut k červené části spektra. Proto jej neoznačují za bílý, ale pouze za růžový.

*** Hledáte-li levný zdroj opravdové náhodnosti, vyzkoušejte *http://lavarand.sgi.com*. Tito lidé poskytují „hardwarovou" náhodnost pomocí digitalizovaného videa sedmdesátých let „lávové lampy". Legrace!

čísel poskytuje Java v základní knihovně `java.lang.Math` a několika dalších knihovnách; ty si prozkoumáme v návodu 5.13.

Třída `java.lang.Math` obsahuje v jedné třídě celou „matematickou knihovnu“ včetně trigonometrie, konverzí (včetně převádění úhlů na radiány a zpět), zaokrouhlování, zkracování, odmocnin, minima a maxima. Všechno zde najdete – vyhledejte si v dokumentaci třídu `java.lang.Math`.

Balíček `java.Math` obsahuje podporu pro „velká čísla“ – delší než například normální zabudovaná dlouhá celá čísla; viz návod 5.19.

Java pilně pracuje na zajištění spolehlivosti svých programů. Toho si obyčejně můžete všimnout v běžném požadavku na zachycení potenciálních výjimek – v celém API Javy – a v potřebě „přetypovávat” nebo převádět hodnoty, jež by se mohly, ale nemusely vejít do proměnné, do které se je snažíte uložit. Ukážeme si to na příkladech.

Ošetřování číselných dat Javy celkově dobře zapadá do teorie přenositelnosti, spolehlivosti a jednoduchosti programování.

Přečtěte si také...

Specifikaci jazyka Java; dokumentaci k balíčku `java.lang.Math`.

5.1 Kontrola, jestli je řetězec platné číslo

Problém

Potřebujete zkontrolovat, jestli daný řetězec obsahuje platná čísla, a pokud ano, tak je zkonvertovat na binární (interní) tvar.

Řešení

Použijte konverzní rutinu příslušné obalové třídy a zachyťte výjimku `NumberFormatException`. Následující kód přemění řetězec na typ `double`:

```
public class StringToDouble {
      public static void main(String argv[]) {
          String cislo;
          if( argv.length == 0 )
            cislo = "12345";   // Nezadané parametry příkazového řádku.
          else
            cislo = argv[0];    // Nikoli argv[1]
          double vysledek;
          try {
              vysledek = Double.parseDouble(cislo);
          } catch(NumberFormatException vyj) {
              System.out.println("Neplatné číslo " + cislo);
              return;
          }
          System.out.println("Číslo je " + vysledek);
      }
}
```

Diskuse

Takto můžete pochopitelně ověřit pouze platnost čísel ve formátu, s kterým návrháři obalových tříd počítali. Potřebujete-li přijímat odlišnou definici čísel, mohli byste ji vymezit pomocí regulárních výrazů (viz kapitola 4).

V určitých situacích byste také chtěli být informováni, jestli dané číslo je celé číslo nebo číslo s plovoucí desetinnou čárkou. Jednou z metod je zjištění přítomnosti znaků `.`, `d`, `e`, nebo `f` ve vstupu; je-li jeden z těchto znaků obsažen, převeďte číslo na typ `double`. V opačném případě ho převeďte na typ `int`:

```
//  Část GetNumber.java
private static Number NAN = new Double(Double.NaN);

/* Zpracuje jeden objekt String a vrací ho jako instanci Number
 */
public Number process(String s) {
    if (s.matches( ".*[\\.dDeEfF].*")) {
        try {
            double dHodnota = Double.parseDouble(s);
            System.out.println("Jedná se o typ double: " + dHodnota);
            return new Double(dHodnota);
        } catch (NumberFormatException v) {
            System.out.println("Neplatný typ double: " + s);
            return NAN;
        }
    } else // Neobsahuje . d e nebo f, takže zkuste převést na typ int.
        try {
            int iHodnota = Integer.parseInt(s);
            System.out.println("Jedná se o typ int: " + iHodnota);
            return new Integer(iHodnota);
        } catch (NumberFormatException v2) {
            System.out.println("Není číslo:" + s);
            return NAN;
        }
}
```

Přečtěte si také...

Spletitější tvar syntaktické analýzy nabízí třída `DecimalFormat`, o které budeme hovořit v návodu 10.5.

Vývojářská sada JDK 1.5 přináší třídu `Scanner`; viz návod 10.5.

5.2 Ukládání větších čísel do menších čísel

Problém

Máte číslo většího typu a chcete ho uložit do proměnné menšího typu.

Řešení

Přetypujte číslo na menší typ. (*Přetypování* představuje uvedení typu v závorkách před hodnotou, což způsobí, že se daná hodnota zpracuje, jako by byla hodnotou uvedeného typu.)

Přetypování je třeba například pro převod typu `long` na `int`. Také převod typu `double` na `float` vyžaduje přetypování.

Diskuse

To působí nováčkům určité potíže, protože výchozím typem čísla s desetinnou čárkou není `float` ale typ `double`. Takže kód:

```
float f = 3.0;
```

se ani nezkompiluje! Jedná se o totéž, jako kdybyste napsali:

```
double tmp = 3.0;
float f = tmp;
```

Napravit to můžete vytvořením proměnné `f` jako typ `double`, vytvořením 3.0 `float`, přetypováním nebo přiřazením celočíselné hodnoty 3:

```
double f = 3.0;
float f = 3.0f;
float f = 3f;
float f = (float)3.0;
float f = 3;
```

Totéž se týká ukládání `int` do typů `short`, `char` nebo `byte`.

```
// CastNeeded.java
public static void main(String argv[]) {
    int i;
    double j = 2.75;
    i = j;             // OČEKÁVÁ SE CHYBA PŘI PŘEKLADU
    i = (int)j;        // s přetypováním; i bude mít hodnotu 2.
    System.out.println("i =" + i);
    byte b;
    b = i;            // OČEKÁVÁ SE CHYBA PŘI PŘEKLADU
    b = (byte)i;      // s přetypováním; i bude mít hodnotu 2.
    System.out.println("b =" + b);
}
```

Řádky označené komentářem OČEKÁVÁ SE CHYBA PŘI PŘEKLADU se nepřeloží, dokud je buď nezakomentujete nebo nezměníte tak, aby byly korektní. Řádky opatřené poznámkou „s přetypováním“ uvádějí správné tvary.

5.3 Převádění čísel na objekty a naopak

Problém

Potřebujete převést čísla na objekty a objekty na čísla.

Řešení

Použijte obalové třídy typů uvedené v tabulce 5.1 na začátku této kapitoly.*

* Poznámka českého vydavatele: Java 5.0 již provádí všechny převody automaticky. Text v této kapitole využijí spíše ti, kteří musí pracovat se staršími verzemi Javy.

Diskuse

Často máte primitivní číslo a potřebujete ho předat do metody, která vyžaduje `Object`. Často k tomu dochází při použití tříd kolekcí (viz kapitola 7) v JDK 1.4 a starších (v 1.5 můžete použít `AutoBoxing`, o němž budeme hovořit v návodu 8.4).

Chcete-li převést `int` na objekt `Integer` nebo naopak, můžete použít následující:

```
// Převod int na objekt Integer
Integer i1 = new Integer(42);
System.out.println(i1.toString()); // nebo pouze i1

// Převod objektu Integer na int
int i2 = i1.intValue();
System.out.println(i2);
```

5.4 Vytvoření zlomku z celého čísla bez použití čísla s plovoucí desetinnou čárkou

Problém

Chcete násobit celé číslo zlomkem bez převodu zlomku na číslo s plovoucí desetinnou čárkou.

Řešení

Vynásobte celé číslo čitatelem a vydělte jmenovatelem.

Tato technika by se měla používat, pouze když je výkon důležitější než jasnost, protože má sklon zhoršovat čitelnost – a tedy i udržovatelnost – vašeho kódu.

Diskuse

Jelikož se celá čísla a čísla s plovoucí desetinnou čárkou uchovávají odlišně, někdy bude možná z hlediska efektivity žádoucí a rozumné vynásobit celé číslo zlomkovou hodnotou, aniž bychom tuto zlomkovou hodnotu převáděli na číslo s plovoucí desetinnou čárkou a nazpět a bez potřeby přetypování:

```
/**Vypočítá hodnotu 2/3 krát 5 */
public class FractMult {
    public static void main(String u[]) {

      // Programátor si v hlavě převedl 2/3 na 0.666 - rychlé, ale
        double d1 = 0.666 * 5;              // nesrozumitelné a nepřesné => zkonvertovat
        System.out.println("0.666 * 5 = " + d1);

        double d2 = 2/3 * 5;                // Nesprávná odpověď - 2/3 == 0, 0*5 = 0
        System.out.println("2/3 * 5 = " + d2);

        double d3 = 2d/3d * 5;              // "normální"
        System.out.println("2d/3d * 5 = " + d3);

        double d4 = (2*5)/3d;               // Celočíselné násobení (je rychlejší),
        System.out.println("(2*5)/3d = " + d4);   // Téměř stejná odpověď

        int i5 = 2*5/3;                     // Rychlá, přibližná celočíselná odpověď.
```

```
        System.out.println("2*5/3 = " + i5);
    }
}
```

Spuštěním programu se vypíše následující:*

```
$ java  FractMult
0.666 * 5 = 3.33
2/3 * 5 = 0.0
2d/3d * 5 = 3.333333333333333
(2*5)/3d = 3.3333333333333335
2*5/3 = 3

$
```

5.5 Zajištění přenosnosti čísel s plovoucí desetinnou čárkou

Problém

Chcete vědět, jestli výpočet čísla s plovoucí desetinnou čárkou generuje rozumný výsledek.

Řešení

Porovnejte s konstantami `INFINITY` a proveďte kontrolu na „není číslo“ pomocí `isNaN()`.

Operace s pevnou desetinnou čárkou, které mohou provádět úlohy jako dělení nulou, způsobí, že vám program neočekávaně vyvolá výjimku. To je zapříčiněno skutečností, že dělení celého čísla nulou je považováno za *logickou chybu*.

Operace s plovoucí desetinnou čárkou nevyhazují výjimku, protože jsou definovány přes (téměř) nekonečný rozsah hodnot. Dělíte-li nulou kladné číslo s plovoucí desetinnou čárkou, bude výsledkem výpočtu konstanta `POSITIVE_INFINITY`, dělíte-li nulou záporné číslo s plovoucí desetinnou čárkou, bude výsledkem výpočtu konstanta `NEGATIVE_INFINITY`. Generujete-li neplatný výsledek nějakým jiným způsobem, bude hodnotou výrazu konstanta `NaN` (Not a Number, neboli není číslo). Hodnoty pro tyto tři veřejné konstanty jsou definovány v obalových třídách `Float` a `Double`. Hodnota `NaN` má neobvyklou vlastnost a to, že není rovna sama sobě, což znamená: `NaN?!=?NaN`. Tudíž by stěží dávalo smysl porovnávat (možná podezřelé) číslo s `NaN`, protože výraz:

```
x == NaN
```

nemůže být nikdy pravdivý. Místo toho je třeba pracovat s metodami `Float.isNaN(float)` a `Double.isNaN(double)`:

```
// InfNaN.java
public static void main(String argv[]) {
    double d = 123;
    double e = 0;
    if (d/e == Double.POSITIVE_INFINITY)
```

* Poznámka českého vydavatele: Možná by mohla ta pětka na konci výsledku čtvrtého výpočtu někoho zmást a svést k domněnce, že je výpočet na třetím řádku přesnější. Opak je pravdou. Výsledek třetí operace má totiž o celou jednu číslici méně a je asi dvakrát méně přesný (jeho výsledek je o zhruba $3{,}3 \times 10^{-16}$ menší, kdežto čtvrtý je přibližně o $1{,}7 \times 10^{-16}$ větší než skutečná hodnota). Při dělení se totiž ztrácí přesnost, kterou při následném násobení již nezískáte.

```
        System.out.println("Kontrola konstanty POSITIVE_INFINITY funguje");
    double s = Math.sqrt(-1);
    if (s != Double.NaN)
        System.out.println("Srovnání s konstantou NaN vrací false");
    if (Double.isNaN(s))
        System.out.println("Metoda Double.isNaN() vrací true");
}()
```

Poznamenejme, že to samo o sobě k zajištění náležité přesnosti výpočtů s plovoucí desetinnou čárkou nestačí. Následující program například ukazuje smyšlený výpočet – Heronův vzorec pro plochu trojúhelníku – jak v typu `float`, tak v `double`. Hodnoty `double` jsou správné, ale hodnoty s plovoucí desetinnou čárkou vychází kvůli zaokrouhlovacím chybám nulové.* Důvodem je skutečnost, že operace v Javě obsahující pouze hodnoty typu `float` se provedou jako 32bitové. Některé překladače příbuzných jazyků, jako například C, automaticky povýší během počítání tyto hodnoty na `double`, což může eliminovat určitou ztrátu přesnosti.

```
/** Pomocí Heronova vzorce vypočítá plochu trojúhelníku.
 * Kód a hodnoty pochází od prof W. Kahana a Josepha D. Darcyho.
 * Viz http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.
 * Odvozeno z výpisu uvedeného v článku Java Pro od Ricka Grehana  (říjen 1999).
 * Zjednodušil a přeformátoval Ian Darwin.
 */
public class Heron {
    public static void main(String[] args) {
        // Strany trojúhelníku jako float
        float af, bf, cf;
        float sf, plochaf;

        //Totéž v double
        double ad, bd, cd;
        double sd, plochad;

        // Plocha trojúhelníku ve float
        af = 12345679.0f;
        bf = 12345678.0f;
        cf = 1.01233995f;

        sf = (af+bf+cf)/2.0f;
        plochaf = (float)Math.sqrt(sf * (sf - af) * (sf - bf) * (sf - cf));
        System.out.println("Jednoduchá přesnost: " + plochaf);

        // Plocha trojúhelníku v double
        ad = 12345679.0;
        bd = 12345678.0;
        cd = 1.01233995;

        sd = (ad+bd+cd)/2.0d;
        plochad =        Math.sqrt(sd * (sd - ad) * (sd - bd) * (sd - cd));
        System.out.println("Dvojnásobná přesnost: " + plochad);
    }
}
```

* Poznámka českého vydavatele: S takovýmito situacemi se při výpočtech setkáte častěji, než by vám bylo milo. Musíte si hlídat, aby se ve vašich výpočtech neodečítala od sebe skoro stejně veliká čísla – tím se automaticky připravujete o několik řádů přesnosti.

Zkusme program spustit. V rámci zajištění, že zaokrouhlování není výtvorem (artefaktem) implementace, otestujeme program jak pomocí JDK od Sunu, tak pomocí Kaffe:

```
$ java Heron
Jednoduchá přesnost: 0.0
Dvojnásobná přesnost: 972730.0557076167
$ kaffe Heron
Jednoduchá přesnost: 0.0
Dvojnásobná přesnost: 972730.05570761673
```

Jste-li na pochybách, použijte `double`!

K zajištění konzistence výpočtů s velmi velkými veličinami typu double na různých implementacích Javy poskytuje Java klíčové slovo `strictfp`, které lze použít na třídy, rozhraní nebo metody v rámci třídy.* Je-li výpočet Strict-FP, pak musí například vždy vracet hodnotu `INFINITY`, kdyby kalkulace překročila hodnotu `Double.MAX_VALUE` nebo nedosáhla hodnoty `Double.MIN_VALUE`.

Výpočty, které nejsou Strict-FP (standardně), mohou provádět výpočty na větším rozsahu a mohou vracet platný konečný výsledek, který je v rozsahu, i když meziprodukt byl mimo rozsah.** To je docela tajuplné a ovlivňuje to pouze výpočty, které se přibližují hranicím toho, co se vejde do typu `double`.

5.6 Porovnávání čísel s plovoucí desetinnou čárkou

Problém

Chcete porovnat na rovnost dvě čísla s plovoucí desetinnou čárkou.

Řešení

Na základě toho, co jsme si právě pověděli, pravděpodobně nebudete na rovnost porovnávat čísla typu `float` nebo `double`. Mohli byste předpokládat, že obalové třídy čísel s plovoucí desetinnou čárkou, `Float` a `Double`, překryjí metodu `equals()`, což také dělají. Metoda `equals()` vrací hodnotu `true`, jsou-li dvě hodnoty bit za bitem stejné. To znamená, že vrátí true právě tehdy, jsou-li čísla stejná nebo mají obě hodnotu `NaN`. Jinak vrací logickou hodnotu `false` včetně případů, když je předaný argument prázdný nebo když má jeden objekt hodnotu +0,0 a druhý -0.0.

Zní-li to podivně, ale pamatujte si, že spletitost částečně vychází z povahy provádění výpočtů s reálnými čísly na méně přesném hardwaru a částečně ze standardu IEEE 754, který definuje funkcionalitu plovoucí desetinné čárky. Java se snaží držet tohoto standardu tak, aby bylo možno použít případný numerický procesor i tehdy, když se programy interpretují.

V praxi se však tato čísla neporovnávají na shodu, ale počítá se s různými zaokrouhlovacími chybami, a čísla jsou proto považována za shodná, i když se trochu liší. Tato povolená tole-

* Všimněte si, že za Strict-FP je rovněž považován výraz, který se skládá pouze z konstant, jejichž hodnotu lze zjistit již během překladu jako `Math.PI * 2.1e17`.

** Poznámka českého vydavatele: Využívá se faktu, že numerické procesory používají pro vlastní výpočet větší rozsah i přesnost, než je rozsah a přesnost proměnných použitých v programu.

rance bývá označována písmenem *epsilon*. V příkladu 4.1 je znázorněna metoda `equals()`, s jejíž pomocí můžete provádět takové porovnání, včetně porovnávání hodnot `NaN`. Kód při spuštění vypíše, že první dvě čísla jsou v rámci přípustné tolerance stejná:

```
$ java FloatCmp
Čísla jsou stejná při epsilon 1.0E-7
Porovnání dvou hodnot NaN vrací: false
Double(NaN).equal(NaN) vrací: true
$
```

Příklad 5.1. *FloatCmp.java*

```
/** Porovnání čísel s plovoucí desetinnou čárkou */
public class FloatCmp {
    final static double EPSILON = 1e-7;

    public static void main(String[] argv) {
        double da = 3 * .3333333333;
        double db = 0.99999992857;
        // Porovná dvě čísla, u kterých se předpokládá, že budou skoro stejná.
        if (da == db) {
            System.out.println("Java se domnívá, že " + da + "==" + db);
        // Jinak se porovná pomocí našich vlastních rovnostních metod.
        } else if (equals(da, db, 0.0000001)) {
            System.out.println("Čísla jsou stejná při epsilon " + EPSILON);
        } else {
            System.out.println(da + " != " + db);
        }
        // Znázorní, že porovnávání dvou hodnot NaN není dobrý nápad:
        double d1 = Double.NaN;
        double d2 = Double.NaN;
        System.out.println( "Porovnání dvou hodnot NaN vrací: " + (d1==d2) );
        System.out.println( "Double(NaN).equal(NaN) vrací: " +
                            new Double(d1).equals(new Double(d2)) );
    }
    /** Porovná dvě hodnoty double se zadanou přesností eps. */
    public static boolean equals(double a, double b, double eps) {
        if (a==b) return true;
        // Je-li rozdíl menší než epsilon, pokládají se za stejné.
        return Math.abs(a - b) < eps;
    }
    /** Porovná dvě hodnoty double s implicitní přesností EPSILON  */
    public static boolean equals(double a, double b) {
        if (a==b) return true;
        // Je-li rozdíl menší než epsilon, pokládají se za stejné.
        return Math.abs(a - b) < EPSILON * Math.max(Math.abs(a), Math.abs(b));
    }
}
```

Pointou tohoto příkladu s hodnotami `NaN` je, že byste vždy měli zajistit, aby hodnoty nebyly `NaN`, než je svěříte metodě `Double.equals()`.

5.7 Zaokrouhlování čísel s plovoucí desetinnou čárkou

Problém

Potřebujete zaokrouhlit čísla s plovoucí desetinnou čárkou na celá čísla nebo na určitou přesnost.

Řešení

Pokud jednoduše přetypujete plovoucí hodnotu na celočíselnou hodnotu, Java hodnotu ořeže. Například hodnota 3,999999 přetypovaná na `int` nebo `long` bude 3, nikoli 4. Pro správné zaokrouhlování čísel s plovoucí desetinnou čárkou používejte metodu `Math.round()`. Tato metoda má dva tvary: pokud jí předáte `double`, získáte výsledek `long`, předáte-li jí `float`, získáte typ `int`.

Co když se vám nelíbí zaokrouhlovací pravidla používaná metodou `round`? V případě, že byste z nějakého zvláštního důvodu chtěli zaokrouhlovat nahoru čísla větší než 0,54 místo normálního 0,5, mohli byste si napsat vlastní verzi `round()`:*

```
/*
 * Zaokrouhlí čísla s plovoucí desetinnou čárkou na celá čísla
 * @return nejbližší číslo int předaného argumentu
 * @param d Nezáporné hodnoty k zaokrouhlení.
 */
static int round(double d) {
    if (d < 0) {
        throw new IllegalArgumentException("Hodnota nesmí být záporná");
    }
    int di = (int)Math.floor(d);    // Celočíselná hodnota menší (nebo ==) d
    if ((d - di) > THRESHOLD) {
        return di + 1;
    } else {
        return di;
    }
}
```

Potřebujete-li zobrazit číslo s menší přesností, než byste normálně obdrželi, pravděpodobně budete chtít použít objekt `DecimalFormat` nebo objekt `Formatter`.

5.8 Formátování čísel

Problém

Potřebujete formátovat čísla.

Řešení

Použijte podtřídu `NumberFormat`.

* Poznámka českého vydavatele: Posledních 6 řádků těla následujícího programu je možné nahradit řádkem jediným: `return (int)(d + THRESHOLD);`

Dřívější verze Javy tradičně neposkytovaly funkce jazyka C *printf/scanf*, protože mají tendenci velmi nepoddajným způsobem smíchávat formátování a vstup-výstup. Programy pracující s funkcemi *printf/scanf* bylo dříve obtížné lokalizovat, což ovšem opět neplatí pro Javu 5.0.

Java nabízí celý balíček, `java.text`, plný formátovacích rutin tak obecných a flexibilních, jak si jen dokážete představit. Tento balíček, stejně jako *printf*, zahrnuje formátovací jazyk, který se popisuje na stránce Javadoc. Popřemýšlejte o znázornění dlouhých čísel. V Severní Americe se číslo jeden tisíc dvacet čtyři celých dvacet pět zapíše jako 1,024.25, v převážné části Evropy jako 1 024,25 a v některých dalších částech světa se může zapisovat ve tvaru 1.024,25. A to se ani nezmiňuji o formátování různých měn a procent! Proto by bylo poněkud náročné, kdybyste se snažili dohlížet na všechno sami.

Naštěstí balíček `java.text` obsahuje třídu `Locale` a runtime Javy navíc ještě na základě uživatelského prostředí automaticky nastaví výchozí objekt `Locale`, který na platformách Macintosh a Windows vychází z uživatelského nastavení, na Unixu z uživatelské proměnné prostředí. (Jak poskytovat vlastní objekt `locale` se dozvíte v návodu 15.8.) Chcete-li poskytovat formátovače s lokalizovatelnými podobami čísel, měn a procent, využijte statické *tovární metody* třídy `NumberFormat`, které normálně vrací `DecimalFormat` s již konkretizovaným správným vzorem. Příslušný objekt `DecimalFormat` k uživatelskému nastavení prostředí (locale) lze získat z tovární metody `NumberFormat.getInstance()` a zpracovat pomocí metod `set`. Jako jednoduchý způsob generování číselného formátu s počátečními nulami se překvapivě ukázala být metoda `setMinimumIntegerDigits()`. Podívejte se na příklad:

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

/*
 * Formátuje číslo naším způsobem a výchozím způsobem
 */
public class NumFormat2 {
    /** Pole čísel k formátování */
    public static final double data[] = {
        0, 1, 22d/7, 100.2345678
    };

    /** Hlavní (a jediná) metoda v této třídě. */
    public static void main(String av[]) {
        // Získá instanci formátu
        NumberFormat form = NumberFormat.getInstance();

        // Nastaví formát na tvar 999.99[99]
        form.setMinimumIntegerDigits(3);
        form.setMinimumFractionDigits(2);
        form.setMaximumFractionDigits(4);

        // Nyní se pomocí tohoto formátu vypíší čísla
        for (int i=0; i<data.length; i++)
            System.out.println(data[i] + "\tse naformátuje jako " +
                form.format(data[i]));
    }
}
```

Program pomocí instance `form` třídy `NumberFormat` vypíše obsah pole:

```
$ java NumFormat2
0.0    se neformátuje jako 000,00
```

```
1.0    se neformátuje jako 001,00
3.142857142857143       se naformátuje jako 003,1429
100.2345678     se naformátuje jako 100,2346
$
```

Pomocí speciálního vzoru můžete také sestavit objekt `DecimalFormat` nebo můžete vzor dynamicky měnit pomocí metody `applyPattern()`. Některé běžnější znaky vzorů jsou uvedeny v tabulce 5.2.

Tabulka 5.2. *Znaky vzorů DecimalFormat*

Znak	Význam
#	Číslo (potlačí se počáteční nuly)
0	Číslo (včetně počátečních nul)
.	Oddělovač desetinných míst dle národního prostředí (desetinná čárka)
,	Oddělovač skupin dle národního prostředí (v angličtině se používá čárka, u nás mezera)
-	Indikátor záporného čísla dle národního prostředí (znaménko minus)
%	Zobrazí hodnotu jako procenta
;	Odděluje dva formáty první pro kladné hodnoty a druhý pro záporné hodnoty
'	Potlačí zobrazení jednoho z výše uvedených znaků
Vše ostatní	Zobrazí se tak, jak se zapíše.

Program `NumFormatTest` vypisuje pomocí jednoho objektu `DecimalFormat` čísla pouze se dvěma desetinnými místy a pomocí druhého objektu formátuje čísla podle výchozího nastavení národního prostředí:

```
// NumFormatTest.java
      /** Číslo k formátování */
      public static final double cislo1 = 1024.25;
      /** Další číslo pro formátování */
      public static final double cislo2 = 100.2345678;
      /** Námi definovaný formát */
      public static final String format = "##0.###";

      /** Hlavní (a jediná) metoda v této třídě. */
      public static void main(String[] av) {
            NumberFormat vychForm = NumberFormat.getInstance();
            NumberFormat nasForm = new DecimalFormat(format);
            // toPattern() znázorní kombinaci #0., atd.
            // Pomocí které toto konkrétní národní prostředí formátuje.
            System.out.println("Výchozí formát je " +
                ((DecimalFormat)vychForm).toPattern());
            System.out.println("Náš formát je     " +
                ((DecimalFormat)nasForm).toPattern());
            System.out.println(cislo1 + " se ve výchozím naformátuje jako " +
                vychForm.format(cislo1));
            System.out.println(cislo1 + " se v našem naformátuje jako     " +
                nasForm.format(cislo1));
            System.out.println(cislo2 + " se ve výchozím naformátuje jako " +
                  vychForm.format(cislo2) );
            System.out.println(cislo2 + " se v našem naformátuje jako     " +
                nasForm.format(cislo2));
```

Tento program vypíše daný vzor a poté pomocí několika formátů naformátuje stejné číslo:

```
$ java NumFormatTest
Výchozí formát je #,##0.##
Náš formát je     #0.###
1024.25 se ve výchozím naformátuje jako 1 024,25
1024.25 se v našem naformátuje jako     1024,25
100.2345678 se ve výchozím naformátuje jako 100,23
100.2345678 se v našem naformátuje jako     100,235
$
```

Přečtěte si také...

Kapitolu 17 knihy *Java I/O*, kterou napsal Elliotte Rusty Harold a vydalo nakladatelství O'Reilly.

5.9 Převod mezi dvojkovými, osmičkovými, desítkovými a šestnáctkovými čísly

Problém

Chcete zobrazit číslo jako sérii bitů – například při komunikaci s určitými hardwarovými zařízeními. Chcete převést binární číslo nebo šestnáctkovou hodnotu na celé číslo.

Řešení

Řešení nabízí třída `java.lang.Integer`. Celé číslo převeďte na binární hodnotu pomocí `toBinaryString()`. Binární řetězec zkonvertujete na celé číslo pomocí metody `valueOf()`.

```
// BinaryDigits.java
String bin = "101010";
System.out.println(bin + " je v desítkové soustavě " +
    Integer.valueOf(bin, 2));
int i = 42;
System.out.println(i + " je ve dvojkové soustavě " +
    Integer.toBinaryString(i));
```

Tento program vypíše binární číslo jako celé číslo a celé číslo jako binární číslo:

```
$ java BinaryDigits
101010 je v desítkové soustavě 42
42 je ve dvojkové soustavě 101010

$
```

Diskuse

Metoda `Integer.valueOf()` má obecnější použití než binární formátování. Převede na `int` řetězcové číslo zapsané v číselné soustavě o libovolném základu* – ten se zadává jako její druhý argument. Třída `Integer` nabízí metody pro převod čísel na řetězce reprezentující zápis čísla v nejčastěji používaných soustavách: `toBinaryString()` ve dvojkové, `toOctalString()` v osmičkové (oktalové) a `toHexString()` v šestnáctkové (hexadecimální) soustavě.

* Poznámka českého vydavatele: Základ soustavy je ve skutečnosti přece jen omezen – musí být celočíselný od 2 do 36.

Samotná třída `String` obsahuje řadu statických metod `valueOf(int)`, `valueOf(double)` a tak dále, takže poskytuje také výchozí formátování. To znamená, že vrací danou číselnou hodnotu naformátovanou jako řetězec.

5.10 Zpracování skupiny celých čísel

Problém

Potřebujete pracovat s řadou celých čísel.

Řešení

U souvislé řady použijte cyklus `for`.

Diskuse

Pro zpracování souvislé množiny celých čísel poskytuje Java cyklus `for`.* Řízení cyklu `for` probíhá ve třech fázích: inicializace, testování ukončovací podmínky a změna. Je-li ukončovací podmínka již na počátku splněna, tělo cyklu se neprovede ani jednou.

U nesouvislého rozsahu čísel použijte třídu `java.util.BitSet`.

Všechny tyto techniky znázorňuje následující program:

```
import java.util.BitSet;

/** Zpracování řady čísel */
public class NumSeries {
    public static void main(String[] args) {

        // Chcete-li seznam pořadových čísel, použijte smyčku for
        // začínající na hodnotě 1.
        for (int i = 1; i <= mesice.length; i++)
            System.out.println("Měsíc # " + i);

        // Chcete-li množinu indexů pole, použijte smyčku for
        // začínající na hodnotě 0.
        for (int i = 0; i < mesice.length; i++)
            System.out.println("Měsíc " + mesice[i]);

        // Pro nesouvislou množinu celých čísel zkuste objekt třídy BitSet.

        // Vytvořte objekt BitSet a zapněte několik bitů.
        BitSet b = new BitSet();
        b.set(0);    // Leden
        b.set(3);    // Duben

        // Toto by pravděpodobně bylo někde jinde v kódu.
        for (int i = 0; i<mesice.length; i++) {
            if (b.get(i))
                System.out.println("Měsíc " + mesice[i] + " je požadován");
        }
    }
```

* Nachází-li se čísla, s nimiž potřebujete pracovat, v poli nebo kolekci (viz kapitola 7), použijte cyklus „foreach“ (viz návod 8.2).

```
    /** Názvy měsíců. Lepší způsob viz kapitola Data/časy */
    protected static String mesice[] = {
        "Leden", "Únor", "Březen", "Duben",
        "Květen", "Červen", "Červenec", "Srpen",
        "Září", "Říjen", "Listopad", "Prosinec"
    };
}
```

5.11 Práce s římskými číslicemi

Problém

Potřebujete naformátovat čísla jako římské číslice. Možná jste právě napsali další díl *Titaniku* nebo *Hvězdných válek* a potřebujete opravit datum registrace copyrightu nebo potřebujete na vstupních stránkách knihy naformátovat čísla stránek (což je trochu reálnější příklad).

Řešení

Použijte moji třídu `RomanNumberFormat`:

```
// RomanNumberSimple.java
RomanNumberFormat nf = new RomanNumberFormat();
int rok = Calendar.getInstance().get(Calendar.YEAR);
System.out.println(rok + " -> " + nf.format(rok));
```

Jak pracovat s třídou `Calendar` pro získání současného roku si vysvětlíme v návodu 6.1. Spuštění programu `RomanNumberSimple` vypadá následovně:

```
+ jikes +E -d . RomanNumberSimple.java
+ java RomanNumberSimple
2004 -> MMIV
```

Diskuse

Pro formátování římských číslic není ve standardním API žádný nástroj. Třída `java.text.Format` je však navržena tak, aby z ní bylo možno odvozovat podtřídy pro přesně takové nepředpokládané účely. Takže přesně to jsem taky udělal a vyvinul jsem třídu pro formátování čísel na římské číslice. Níže uvádím lepší a kompletní příklad programu, který s pomocí této třídy formátuje aktuální rok. Na příkazovém řádku můžu předat celou řadu argumentů včetně `"-"`, kde chceme zobrazit rok (všimněte si, že tyto argumenty se běžně neuvozují; v rámci zachování jednoduchosti programu musí být `"-"` argumentem). S programem pracujeme následovně:

```
$ java  RomanYear   Copyright (c) - Ian Darwin
Copyright (c) MMIV Ian Darwin
$
```

Kód programu `RomanYear` je jednoduchý, nicméně správně vkládá mezery kolem argumentů:

```
import java.util.Calendar;
import com.darwinsys.util.RomanNumberFormat;/** Vypíše aktuální rok římskými číslicemi */
public class RomanYear {

    public static void main(String[] argv) {

        RomanNumberFormat rf = new RomanNumberFormat();
        Calendar kalendar = Calendar.getInstance();
        int rok = kalendar.get(Calendar.YEAR);
```

```
        // Nemá-li žádné argumenty, vytiskne pouze rok.
        if (argv.length == 0) {
            System.out.println(rf.format(rok));
            return;
        }

        // Jinak mikro-formátovač: nahradí argument "-" za rok, jinak vypíše.
        for (int i=0; i<argv.length; i++) {
            if (argv[i].equals("-"))
                System.out.print(rf.format(rok));
            else
                System.out.print(argv[i]);    // Např. "Copyright"
                System.out.print(' ');
        }
        System.out.println();
    }
}
```

Nyní zde máme kód třídy `RomanNumberFormat`. Vplížili jsme se do další třídy `java.text.FieldPosition`. `FieldPosition` jednoduše představuje pozici jednoho číselného pole v řetězci, který byl naformátován pomocí varianty `NumberFormat.format()`. Třídu sestavíte tak, aby představovala buď celočíselnou část nebo zlomkovou část (římské číslice samozřejmě nemají zlomkové části). Metody `getBeginIndex()` a `getEndIndex()` třídy `FieldPosition` označují, kde se ve výsledném řetězci dané pole zapojilo.

Příklad 5.2 představuje třídu, která implementuje formátování římských číslic. Jak naznačují komentáře, jedním omezením je, že výstupní číslo musí být menší než 4 000.

Příklad 5.2. *RomanNumberFormat.java*

```
package cz.darwinsys._05_čísla;
import java.text.FieldPosition;
import java.text.Format;
import java.text.NumberFormat;
import java.text.ParsePosition;
/**
 * Třída Roman Number není lokalizována, protože latina je mrtvý jazyk
 * a římské číslice budeme zobrazovat ve všech Locale shodně.
 * Naplněný pomocí narychlo nahozených algoritmů.
 */
public class RomanNumberFormat extends Format {

    /** Znaky užívané v "Arabic to Roman", tj. metody  format(). */
    static char A2R[][] = {
            { 0, 'M' },
            { 0, 'C', 'D', 'M' },
            { 0, 'X', 'L', 'C' },
            { 0, 'I', 'V', 'X' },
    };

    /** Naformátuje dané číslo typu double jako římskou číslici;
     *  ořízne hodnotu na typ long a zavolá metodu format(long).
     */
    public String format(double n) {
        return format((long)n);
    }
```

```
/** Naformátuje dané číslo typu long jako římskou číslici.
 *  Vyvolá tvar se třemi argumenty.
 */
public String format(long n) {
    if (n <=  0 || n >= 4000)
        throw new IllegalArgumentException(n + " musí být > 0 && < 4000");
    StringBuffer sb = new StringBuffer();
    format(new Integer((int)n), sb,
           new FieldPosition(NumberFormat.INTEGER_FIELD));
    return sb.toString();
}

/** Naformátuje dané číslo typu long jako římskou číslici;
 * vrací aktualizovaný Stringbuffer a aktualizuje FieldPosition.
 * Tato metoda je SKUTEČNÝ FORMÁTOVACÍ STROJ.
 * Počet parametrů metody je nadměrný, ale s tím se musí třída
 * jako potomek třídy Format smířit.
 */
public StringBuffer format(Object on, StringBuffer sb, FieldPosition fp) {
    if (!(on instanceof Number))
        throw new IllegalArgumentException(on + " musí být objekt Number");
    if (fp.getField() != NumberFormat.INTEGER_FIELD)
        throw new IllegalArgumentException(fp +
                  " musí být FieldPosition(NumberFormat.INTEGER_FIELD");
    int n = ((Number)on).intValue();   // Tady se má zkontrolovat rozsah

    // Nejprve vloží číslice do pomocného zásobníku. Musí být čtyřciferné.
    for (int i=0; i<4; i++) {
        int d=n%10;
        push(d);
        // System.out.println("Vloženo " + d);
        n=n/10;
    }

    // Nyní je vyjme a zkonvertuje.
    for (int i=0; i<4; i++) {
        int ch = pop();
        // System.out.println("Vyjmuto " + ch);
        if (ch==0)
            continue;
        else if (ch <= 3) {
            for(int k=1; k<=ch; k++)
                sb.append(A2R[i][1]);   // I
        }
        else if (ch == 4) {
            sb.append(A2R[i][1]);       // I
            sb.append(A2R[i][2]);       // V
        }
        else if (ch == 5) {
            sb.append(A2R[i][2]);       // V
        }
        else if (ch <= 8) {
            sb.append(A2R[i][2]);       // V
            for (int k=6; k<=ch; k++)
                sb.append(A2R[i][1]);   // I
        }
        else { // 9
```

```
                sb.append(A2R[i][1]);
                sb.append(A2R[i][3]);
            }
        }
        // fp.setBeginIndex(0);
        // fp.setEndIndex(3);
        return sb;
    }

    /** Syntakticky analyzuje generický objekt a vrací Object */
    public Object parseObject(String co, ParsePosition kde) {
        throw new IllegalArgumentException("Syntaktická analýza není implementována");
        // ZDE SE MÁ PROVÉST SYNTAKTICKÁ ANALÝZA
        // Vrací nový Long(0);
    }

    /* Implementuje pomocný zásobník */
    protected int zasobnik[] = new int[10];
    protected int hloubka = 0;

    /* Implementuje pomocný zásobník */
    protected void push(int n) {
        zasobnik[hloubka++] = n;
    }
    /* Implementuje pomocný zásobník */
    protected int pop() {
        return zasobnik[--hloubka];
    }
}
```

Třída vyžaduje několik veřejných metod, protože jsme chtěli, aby byla podtřídou abstraktní třídy `Format`. To vysvětluje některé spletitosti, jako existenci tří různých formátovacích metod.

Všimněte si, že rodičovská třída vyžaduje rovněž implementaci metody `parseObject()`, ale v této verzi ve skutečnosti neimplementuji syntaktickou analýzu. Její doplnění ponechám jako cvičení pro čtenáře.

Přečtěte si také...

Kniha *Java I/O* (O'Reilly) obsahuje celou kapitolu o třídě `NumberFormat` a je zde vyvinuta podtřída `ExponentialNumberFormat`.

V online zdrojích k této knize najdete program `ScaledNumberFormat`, který vypíše čísla s maximálně čtyřmi číslicemi a „počítačovou" jednotku (B pro bajty, K pro kilo, M pro mega atd.).

5.12 Správné formátování množného čísla

Problém

Vypisujete řetězec jako `"Koupili jsme " + n + " položek"`, ale v češtině tvrzení „Koupili jsme 1 položek" není stylisticky správné. Chcete tedy vypsat „Koupily jsme 1 položku".

Řešení

Použijte ekvivalent třídy `ChoiceFormat` nebo obdobný podmíněný příkaz.