Java pod lupou

Ak ste sledovali C++ pod lupou, nový seriál bude písaný rovnakým štýlom, azda len s tým rozdielom, že kým pri opise C++ som sa snažil obmedziť výklad na štandardné a implementačne nezávislé črty, pri Jave je takýto postup zbytočný. Jej špecifikácia a vývoj je pod taktovkou jedinej firmy (hm, žeby monopol?) a každá implementácia Javy, ktorá nesúhlasí so špecifikáciou, je chybná. Okrem toho vždy je k dispozícii takpovediac referenčná implementácia Java platformy s názvom Java Runtime Environment, pochádzajúca priamo od tvorcov Javy, firmy Sun Microsystems, ktorá je k dispozícii zadarmo. Seriál sa bude zaoberať Javou ako celkom, teda opisom jednak jazyka, jednak knižnice tried, ktoré sú k dispozícii.

Príklady, ktoré nájdete v jednotlivých častiach seriálu, nebudú obmedzené na konkrétne vývojové prostredie. Väčšina komerčných vývojových prostredí je založená na Java Development Kite od firmy Sun Microsystems (na voľné stiahnutie v poslednej verzii 1.3 na adrese http://java.sun.com/j2se/1.3). Ak nevlastníte licenciu na nijaké komerčné prostredie (ako Borland JBuilder, IBM Visual Age for Java a pod.), JDK na preklad a spúšťanie programov úplne postačuje. Azda len tie komerčné prostredia sú výrazne komfortnejšie. Ale ani v prípade, že máte prístup iba k JDK, nemusíte zúfať – na internete je k dispozícii niekoľko pomerne šikovných nadstavieb. Pre jednotnosť budem ďalej mlčky predpokladať, že používate JDK.

V prípade, že pocítite potrebu doplniť si informácie z iných zdrojov, najlepšie bude začať na oficiálnych stránkach (na adrese http://java.sun.com/docs). Okrem tohto oficiálneho zdroja existuje nespočetné množstvo ďalších stránok a diskusných skupín, ktoré sa venujú Jave. Takisto sa môžete obrátiť aj na tlačenú literatúru, tá však podlieha času a knihy prekladané z anglických či iných originálov môžu byť v čase vydania už mierne zastarané. Ak máte k dispozícii kreditnú kartu, obráťte sa priamo na niektorý zo zahraničných internetových obchodov – tak sa dajú zohnať aj najnovšie knihy s aktuálnymi informáciami. Cena bude vyššia, ale žijeme v informačnej spoločnosti a treba si zvykať, že informácie sú najdrahším obchodným artiklom.

Základné pojmy

Skôr než sa pustíme do Javy, bude vhodné ozrejmiť si niekoľko základných pojmov, s ktorými sa v tomto seriáli stretnete.

Základnou jednotkou informácie je **bit**. Premenná s veľkosťou jedného bitu (takisto logická alebo boolovská premenná) môže nadobúdať dva stavy: 0 a 1, resp. false a true. Postupnosť ôsmich bitov sa nazýva **bajt**. Historicky je bajt najmenšou adresovateľnou jednotkou počítačovej pamäte. Premenná s veľkosťou jeden bajt môže nadobúdať 2^8 rôznych hodnôt. Používané násobky bajtov vychádzajú z binárnej sústavy: 1 kilobajt (1 KB) = 2^{10} bajtov, 1 megabajt (1 MB) = 2^{20} bajtov atď. Premenné v počítačových jazykoch mávajú ďalej šírku 16 bitov (**slovo**), 32 bitov (**dvojslovo**), 64 bitov (**štvorslovo**). Ich rozsahy sú po rade 2^{16}, 2^{32} a 2^{64} rôznych hodnôt.

Pamäť počítača je postupnosť pamäťových buniek s rozsahom jedného bajtu. V moderných operačných systémoch musíme rozlišovať *fyzickú pamäť*, ktorá je úplne pod správou operačného systému a programy ju obyčajne neadresujú priamo, a virtuálnu pamäť, ktorú si môžeme predstaviť ako nezávislý pamäťový priestor, pridelený každému procesu, rozdelený na jednotlivé stránky. Z nich sa vo fyzickej pamäti nachádzajú len tie, ktoré proces práve používa, ostatné sú odložené na disku v stránkovacom súbore. Koncept virtuálnej pamäte umožňuje programom využívať podstatne väčšiu pamäť, ako je fyzicky k dispozícii.

Štandardný vstup a **štandardný výstup** sú pojmy pochádzajúce z operačného systému UNIX. Stretneme sa s nimi aj v Jave – ide o štandardné rozhrania na vstup údajov do programu a na výstup údajov z programu. Štandardný vstup je implicitne prepojený s klávesnicou, štandardný výstup smeruje na obrazovku. Obe rozhrania je možné pomocou prostriedkov operačného systému presmerovať napríklad na súbory.

Ľahký úvod do OOP

Základné pojmy sme si objasnili, všetko ostatné sa dozviete v samotnom seriáli – s jedinou výnimkou, ktorou sú princípy objektovo orientovaného programovania. Neodporúčam štúdium Javy bez minimálne základných znalostí OOP, preto si tu uvedieme aspoň fundamentálne princípy. V každom prípade, kto nemá v tejto oblasti jasno, mal by si vedomosti čo najskôr doplniť.

Paradigma OOP je založená na modelovaní objektov reálneho sveta. Objekt je v OOP základnou entitou. Charakterizovaný je svojím stavom (určeným hodnotou tzv. atribútov objektu) a svojím správaním, ktoré definuje, čo objekt dokáže robiť. Stav objektu je navonok neviditeľný – nevieme, čo ho tvorí ani ako je uložený. Toto je prvý základný princíp OOP, hovorí sa mu zapuzdrenie (*encapsulation*). Dôvodom ukrytia stavu objektu pred okolím je snaha zabrániť takým jeho modifikáciám, ktoré by uviedli objekt do nekonzistentného stavu.

Interakcia medzi objektmi sa realizuje pomocou posielania správ. Každý objekt dáva k dispozícii zoznam správ, na ktoré dokáže reagovať (to je to spomínané správanie objektu). Napríklad v bankovej aplikácii by mohol objekt Účet reagovať na správy „Vypočítaj úrok“, „Preveď sumu“ a podobne. Dôležité je, že iniciátor správy sa nemusí starať o to, akým spôsobom príjemca jeho správu spracuje. Príjemca sa, naopak, nestará o to, kto mu správu poslal. V súvislosti so zapuzdrením je zrejmá zjavná výhoda OOP oproti klasickému štýlu: pokiaľ je stav objektu pred svetom ukrytý a prístupné je iba komunikačné rozhranie, môžeme kedykoľvek objekt vymeniť za iný s rovnakým rozhraním. Pokiaľ sa nezmení funkčnosť objektu, okolité objekty nemusia ani len tušiť, že k nejakej výmene došlo.

Druhým dôležitým konceptom OOP je *polymorfizmus*. Táto vlastnosť znamená, že viacero objektov dokáže reagovať na jednu a tú istú správu, každý však inak, svojím vlastným spôsobom. Klasickým prípadom môže byť správa „Serializuj sa“. Serializácia objektu je zakódovanie jeho stavu pre uloženie napríklad do súboru, z ktorého bude v budúcnosti možné objekt rekonštruovať. Je zrejmé, že každý objekt sa bude serializovať inak; to nás však nemusí zaujímať, stačí, keď pošleme príslušnú správu všetkým objektom.

Zapuzdrenie a polymorfizmus sú naozaj základnými princípmi OOP. Bez nich by celá koncepcia stratila zmysel. Často sa však môžete stretnúť ešte s pojmami trieda a dedičnosť. Triedy sú v klasickom OOP šablónami, na základe ktorých je možné generovať nové objekty. Dôležité je, že sú to takisto objekty (lepšie povedané metaobjekty). V mnohých jazykoch (napríklad v C++, ale Java nie je výnimkou) však nie je možné vytvoriť objekt bez toho, aby sme najprv definovali jeho triedu. V takýchto jazykoch sú triedy iba jazykovými konštrukciami.

Posledný pojem – dedičnosť – predstavuje možnosť usporadúvať objekty, resp. ich triedy do hierarchických úrovní na základe vzťahu generalizácie/špecializácie. Napríklad objekt TermínovanýÚčet je špeciálnym prípadom objektu Účet, čo môžeme s úspechom využiť a v definícii objektu TermínovanýÚčet uviesť len odlišnosti oproti generickému Účtu.

Java je zameraná takmer výhradne objektovo. V niektorých oblastiach zachádza viac do hĺbky ako C++, hoci stále to nie je taký silne objektový jazyk ako napríklad Smalltalk. Ostatne, o všetkom sa budete mať možnosť presvedčiť pri sledovaní tohto seriálu.

Java: nová paradigma

Na začiatok si položme otázku, čo vlastne Java je. Odpoveď nie je triviálna. Java totiž nie je len ďalší z dlhého radu programovacích jazykov, hoci moderný a efektívny, ani módna záležitosť, vyvolaná masívnou propagáciou materskej firmy Sun Microsystems, ktorej divízia JavaSoft má vývoj Javy na starosti, a už vôbec nie zábavná technológia na vytváranie appletov s poskakujúcim textom, oživujúcim až donedávna statické webové stránky. Nie, Java je komplexná platforma pre vývoj a beh širokého spektra aplikácií v heterogénnom sieťovom prostredí.

História Javy je pomerne krátka. Začiatkom 90. rokov zatiaľ ešte tohto storočia skupina výskumných pracovníkov vo firme Sun Microsystems pracovala na projekte vývoja univerzálneho softvéru slúžiaceho na riadenie a programovanie sieťových zariadení a tzv. vnorených systémov (embedded systems), ako napr. čipy v mikrovlnkách, mobilných telefónoch, spotrebnej elektronike a pod. Výskumníci pôvodne rátali s použitím programovacieho jazyka C++, čoskoro však usúdili, že tento jazyk je na ich účely príliš zložitý, nejednoznačný a nespoľahlivý. Krátko nato uzrel svetlo sveta ich vlastný produkt, nazvaný jednoducho Oak. Tvrdí sa, že jeho názov má na svedomí dub, ktorý rástol pred oknami pracovne vedúceho projektu Jamesa Goslinga.

Celý projekt tak trochu predbehol svoju dobu a zrejme by upadol do zabudnutia, nebyť razantného nástupu internetu v roku 1995. Vtedy pán Gosling vycítil, že jeho jazyk je ideálny na použitie v heterogénnych sieťových prostrediach, ako je aj internet. Spolu s výskumnou skupinou pripravili koncept univerzálnej programovacej platformy, orientovanej na sieťové prostredie a nezávislej od použitého hardvéru či operačného systému. Platforma a programovací jazyk dostali názov Java, podľa jednej z historiek vraj preto, lebo ľudia, ktorí majú Javu „na svedomí“, s obľubou pijú kávu (java je americký slangový výraz pre kávu).

Prvá verzia Javy s označením 1.0 obsahovala knižnicu tried pozostávajúcu zo siedmich základných balíkov, ktoré pokrývali najdôležitejšie oblasti použitia, ako práca s reťazcami, matematické funkcie, sieťové funkcie, práca s grafikou a pod. K jej rozšíreniu prispela jednak jej voľná licencia – vývojové prostredie na tvorbu javovských aplikácií Java Development Kit bolo k dispozícii zadarmo –, jednak vytvorenie webového prehliadača HotJava, ktorý ako prvý umožňoval prehliadanie stránok s vloženými mikroaplikáciami, tzv. appletmi (mimochodom, celý bol napísaný v Jave). Applety poskytovali nevídanú mieru interaktivity na dovtedy statických webových stránkach, a preto nečudo, že podporu technológie Java v rýchlom slede ohlásili aj výrobcovia dvoch najznámejších webových prehliadačov, spoločnosti Netscape a Microsoft.

Už v roku 1997 firma Sun uvoľnila ďalšiu verziu Javy s označením Java 1.1. V špecifikácii jazyka došlo k niekoľkým drobným, ale užitočným zmenám, najväčšie rozšírenie však podstúpila knižnica tried. Počet balíkov

sa rozšíril zo siedmich na dvadsaťdva – pribudla lepšia podpora grafiky, možnosť práce s komprimovanými súbormi, s databázami, bezpečnostné funkcie, podpora distribuovaných aplikácií a predovšetkým koncept tzv. Java Beans (zapuzdrené komponenty, niečo ako ActiveX).

Vývoj Javy sa však nezastavil. V priebehu posledných rokov počiatočné nadšenie zo strany tvorcov webových stránok, ktoré sa prakticky prejavovalo tým, že na javovský applet ste narazili pomaly na každej druhej stránke, našťastie opadlo a Java sa postupne pretransformovala na reálne použiteľnú technológiu. Najnovšia verzia Javy s marketingovým označením Java 2 Platform predstavuje skutočne komplexnú programovaciu platformu, mimoriadne vhodnú na nasadenie v heterogénnych podnikových aplikáciách a intranetoch, umožňujúcu úplnú sieťovú interoperabilitu a beh distribuovaných aplikácií podľa súčasných štandardov.

Platformu Java 2 firma Sun pomyselne delí do troch vetiev: Java 2 Standard Edition, Java 2 Enterprise Edition a Java 2 Micro Edition. V seriáli Java pod lupou sa budeme zaoberať prvou vetvou, J2SE, ktorá predstavuje plnohodnotnú platformu pre vývoj desktopových aplikácií, poskytujúcu všetky črty verzie 1.1 s niekoľkými podstatnými zlepšeniami v oblasti používateľského rozhrania, práce so zvukom, podpory distribuovaných aplikácií a transakčného spracovania. Druhá vetva, J2EE, zahŕňa oproti J2SE technológie určené na nasadenie v prostredí podnikových aplikácií, ako Enterprise Java Beans, Java Server Pages a pod. O tejto vetve si azda povieme niečo viac v budúcnosti. Konečne tretia vetva, J2ME, je určená pre zariadenia ako vreckové počítače či mobilné telefóny.

Na stránkach JavaSoftu nájdete okrem týchto troch hlavných vetiev množstvo špecifikácií ďalších technológií súvisiacich s Javou, ktoré však tak trochu presahujú rámec nášho seriálu, a preto sa nimi zaoberať nebudeme. Kto má o ne záujem, môže si príslušné informácie vyhľadať na internete.

Charakteristika Javy

V oficiálnom opise The Java Language Environment (nájdete ho na adrese http://java.sun.com/docs/white/langenv) je Java výstižne charakterizovaná množinou adjektív. I keď ich zoznam pôsobí ako zbierka typických marketingových „buzzwords“, prekvapivo dobre vystihujú podstatné vlastnosti Javy:

• Java je **jednoduchá**. Jednou z najväčších nevýhod jazykov ako C++ je ich zložitosť. Isto mi dáte za pravdu, že na dokonalé zvládnutie C++ potrebujete mať pomerne rozsiahle vedomosti a dostatočne dlhú prax a vaša cesta k poznaniu bude dláždená neuveriteľným množstvom chýb, ktoré stvoríte. Práve preto bola Java od začiatku navrhovaná tak, aby vaša krivka učenia bola pokiaľ možno čo najstrmšia a aby vás netrafil šľak skôr, než niečo užitočné vyprodukujete. Autori jazyka pri návrhu Javy vychádzali z existujúcich objektovo orientovaných jazykov (C++, Smalltalk, Objective C, Eiffel a pod.), z ktorých systematicky odstraňovali príliš zložité črty, až dospeli ku konečnému, minimalistickému stavu. Java je skutočne ukážkovým príkladom jednoduchého a pritom dostatočne výkonného jazyka.

• Java sa vám bude zdať **známa**. Neoddiskutovateľnou prednosťou Javy je jej príbuznosť s C++ – program v Jave vyzerá na pohľad takmer ako program v C++, a ak aspoň trochu ovládate C++, nemali by ste mať so zvládnutím Javy žiadne problémy. Java používa podobné údajové typy, operátory, jazykové konštrukcie i príkazy. Na rozdiel od C++ však oslobodzuje programátora od pamätania si komplikovaných pravidiel, od dodržiavania správnych postupov (napr. pri alokácii a uvoľňovaní pamäte), odstraňuje mnohé síce užitočné, ale pri nesprávnom použití nebezpečné konštrukcie a predovšetkým nepozná ukazovatele, čo je zrejme najčastejšia príčina chýb v programoch C++. Ako niekto priliehavo poznamenal, Java je C++ s menej povrazmi, na ktorých by sa programátor mohol obesiť. Kruté, ale pravdivé.

• Java je **objektovo orientovaná.** Dnes hádam ani netreba zdôrazňovať, že klasický procedurálny a funkcionálny spôsob návrhu aplikácie je prežitok. Objektovo orientovaná analýza, návrh a programovanie vedú k prehľadnejšej štruktúre, lepšej orientácii v programe a k menšej náchylnosti na výskyt chýb. Samozrejme, OO prístup nie je zázračný prútik, ktorého mávnutím nám pod rukami vykvitne perfektná a bezchybná aplikácia. Jeho výhoda je v snahe o napodobenie objektov reálneho sveta, ich stavu a správania – takýto spôsob tvorby softvéru je nám rozhodne bližší ako úporné, procedurálne napodobovanie „myslenia“ počítača. Java podporuje prakticky všetky dôležité princípy OO programovania – zapuzdrenie stavu objektov a jeho ukrytie pred vonkajším svetom, komunikáciu medzi objektmi pomocou správ (ktoré majú za následok vyvolanie príslušných metód objektov), polymorfné správanie objektov, teda rôznu reakciu rôznych objektov na jednu a tú istú správu, triedy (iba ako jazykové konštrukcie, a nie ako plnohodnotné runtimeové metaobjekty), jednoduchú dedičnosť (na simuláciu viacnásobnej dedičnosti slúžia tzv. rozhrania). Podpora OO paradigmy v Jave úzko súvisí s jej orientáciou na sieťové prostredia a distribuované aplikácie.

• Java je **architektonicky nezávislá**. Pri klasickom spôsobe vývoja aplikácií musí softvérový výrobca vytvoriť osobitnú verziu aplikácie pre každú podporovanú platformu a neraz aj pre každý podporovaný operačný systém. Tento prístup je nesporne časovo aj finančne náročný, nehovoriac o nutnosti vyrovnávať sa so špecifikami jednotlivých platforiem. Java vám ponúka šalamúnske riešenie nezávislosti od konkrétneho prostredia: javovské programy nie sú v binárnej forme zložené z natívnych inštrukcií toho-ktorého procesora, ale z inštrukcií tzv. bajt-kódu, platformovo nezávislého inštrukčného súboru. Medzi hardvérom/softvérom príslušnej platformy a programom v Jave leží vrstva tzv. virtuálneho počítača (Java VM), ktorý dokáže inštrukcie bajt-kódu interpretovať. Javovský program je tak možné spustiť na ľubovoľnej platforme, pre ktorú existuje javovský virtuálny stroj. Softvérovému výrobcovi zo začiatku tohto odseku potom stačí vytvoriť jedinú verziu aplikácie, ktorú bude možné spustiť všade. Pre tento koncept má firma Sun obľúbené heslo: „Write Once, Run Everywhere.“

• Java je **portabilná**. Možno máte dojem, že portabilnosť Javy je priamym dôsledkom jej architektonickej nezávislosti. Je za tým však ešte niečo viac: v špecifikácii napríklad C++ je mnoho vecí definovaných ako implementačne závislé. Niečo také v Jave nenájdete. Všetky údajové typy majú presne stanovený rozsah, všetky jazykové konštrukcie sa správajú za každých okolností rovnako, nezávisle od konkrétnej implementácie virtuálneho stroja. Okrem toho pôvodný sunovský virtuálny stroj je napísaný v ANSI C a je maximálne portabilný v zmysle štandardu POSIX.

• Java je **robustná** a **spoľahlivá**. Oproti jazykom ako C++ má Java oveľa prísnejšiu syntax, takže mnoho potenciálnych zdrojov chýb je odstránených už počas fázy prekladu zdrojových textov. Kontrola binárnych tried pokračuje aj vo fáze behu programu. Klasické linkovanie, ako ho poznáme z iných jazykov, sa deje dynamicky, v okamihu, keď je tá-ktorá trieda potrebná. Javovský virtuálny stroj počas behu kontroluje správnosť a konzistentnosť načítavaných tried, správnosť typov parametrov volaných metód a podobne. Iným veľkým rozdielom Javy oproti C++ je neporovnateľne robustnejšia správa pamäte. V Jave sa nemôže stať, že budete pristupovať k nejakej oblasti pamäte iným ako povoleným spôsobom, okrem toho uvoľňovanie nepotrebných objektov z pamäte je automatické, čím sa eliminujú klasické memory-leak problémy.

• Java je **interpretovaná**. Bohužiaľ, chcelo by sa mi povedať. V podstate každý program je počas behu interpretovaný – klasické natívne programy sú však interpretované priamo procesorom. Javovské programy vzhľadom na prítomnosť vrstvy virtuálneho stroja musia byť interpretované softvérovo. (Objavujú sa síce správy o dedikovanom hardvéri, na ktorom bude môcť bežať program v Jave priamo, ale zatiaľ sme nič podobné nevideli.) Softvérová interpretácia má však podstatnú nevýhodu – je jednoducho pomalšia. Našťastie procesory sú čoraz rýchlejšie a virtuálne stroje stále efektívnejšie, takže táto nevýhoda Javy sa dá ako-tak stráviť.

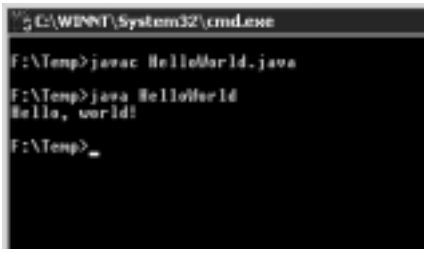
• Java je **dynamická**. Ako bolo povedané, pri vývoji aplikácií v Jave sa klasická linkovacia fáza odkladá až do okamihu, keď je príslušná trieda naozaj potrebná. Kompilátor všetky odkazy na triedy ukladá nie v binárnom, číselnom tvare, ale v symbolickom, mennom. Znamená to okrem iného, že pri zmene definície jednej triedy nie je nevyhnutné opätovne prekladať zdrojové súbory všetkých tých tried, ktoré so zmenenou triedou pracujú. Dokonca je v princípe možné vymeniť binárny súbor s definíciou triedy aj za behu aplikácie, ak táto trieda dosiaľ nebola načítaná. Používané triedy môžu byť naťahované z lokálneho systému, ale aj z akéhokoľvek miesta dostupného cez sieť (presne takto sa správajú applety vo webových stránkach). Poslednou veľkou výhodou je, že triedy v Jave majú svoju vlastnú run-time reprezentáciu, takže je možné za behu programu zisťovať typ akéhokoľvek objektu, dokonca možno vytvoriť inštanciu triedy, ktorej názov bude známy až počas behu programu.

• Java je **bezpečná**. Vzhľadom na predpokladané použitie Javy v implicitne nezabezpečených sieťových prostrediach pri jej návrhu sa kládol veľký dôraz na bezpečnostné opatrenia znemožňujúce potenciálne zneužitie existujúceho kódu na iné ako predpokladané účely. Prvým, pomerne dôležitým predpokladom je samotný pamäťový model Javy. Programátor v zásade nikdy nevie, kde a akým spôsobom sú v pamäti uložené jednotlivé inštancie tried a aká je ich vnútorná reprezentácia. Keďže v Jave neexistujú ukazovatele, nemáte nijakú možnosť pristupovať do ľubovoľnej, explicitne určenej oblasti pamäte. Ďalším ochranným prvkom je mechanizmus načítavania tried (tzv. *class loader*). Ten zabezpečí, že nie je možné štandardné triedy, ktoré sa vždy nachádzajú na lokálnom systéme, nahradiť podvrhnutými verziami, načítavanými zo siete. Okrem toho každá načítaná trieda prejde dôslednou kontrolou bajt-kódu, ktorá nekorektné, nebezpečné či inak neakceptovateľné triedy jednoducho odmietne načítať. Dôležité je, že táto kontrola sa vykonáva iba raz, pri načítavaní triedy a počas behu programu sa interpret Javy nemusí zdržovať napríklad kontrolou zásobníka, typov operandov a podobne.

• Java **podporuje multithreading**. Dnes už hádam netreba vysvetľovať výhody multithreadingu. Ak vám toto slovo nič nehovorí, tak ide o možnosť existencie viacerých threadov (vykonávacích vlákien) v rámci jedného procesu. Všetky thready zdieľajú adresný priestor a prostriedky materského procesu, len z hľadiska plánovača procesov vystupujú ako samostatné plánovateľné entity, to značí, že súťažia o procesor. Výhoda threadov je nesporná – klasickým príkladom môže byť textový procesor. V jednom z threadov editujete dokument, druhý má na starosti napríklad prekresľovanie obrazovky, ďalší tlačenie a ešte ďalší sa môže starať napríklad o kontrolu pravopisu. Procesor je využívaný oveľa efektívnejšie a program vykazuje lepší ohlas. Java je jeden z mála jazykov, ktorý zahŕňa podporu multithreadingu na úrovni jazyka. Priamo pomocou príslušnej triedy a jej metód je možné vytvárať, spúšťať i zastavovať jednotlivé thready. Tieto thready sú v závislosti od použitej hardvérovej a softvérovej platformy úplne preemptívne. V multithreadovom prostredí je tiež nevyhnutná prítomnosť synchronizačných prostriedkov. Java obsahuje podporu pre monitory (vzájomné vylučovanie pre prístup k triede) a pre podmienené premenné (*conditional variables*).

Prvý program

Máme teda vytvorenú predstavu o tom, čo Java je, aké má vlastnosti a ako zhruba funguje. Aby bola prvá časť seriá-



Obr. 1 Preklad a spustenie programu

lu úplná, ukážeme si ešte na záver azda najjednoduchší program, aký sa v Jave dá napísať. Tradície neradno rušiť, preto program vypíše na konzolu obligátny text „Hello, world!“:

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

Program si pravdepodobne budete chcieť vyskúšať. Prepíšte ho teda do súboru, ktorý musíte pomenovať HelloWorld.java. Preklad spustíte najlepšie v konzolovom okne/DOS boxe zadaním riadka javac HelloWorld.java, po jeho skončení v aktuálnom adresári pribudne súbor HelloWorld.class. To je súbor s binárnym bajt-kódom vytvorenej triedy HelloWorld. Spustíte ho jednoducho – zadaním riadka java HelloWorld (**obr. 1**). V prípade, že používate JDK 1.3, je možné, že budete musieť do premennej prostredia CLASSPATH pridať cestu k aktuálnemu adresáru, čo je znak . (bodka). Neviem síce prečo, v predchádzajúcich verziách to nebolo potrebné, ale v operačných systémoch Windows 2000 ani Windows NT (iné som netestoval) bez tejto úpravy nijaký javovský program nespustíte.

Poznámka: *V operačných systémoch Windows 2000/NT sa premenné prostredia definujú v Control paneli, v applete System. V operačných systémoch Windows Me/9x treba pridať príslušný riadok do súboru autoexec.bat.)*

2. časť

Na úvod tejto časti by som rád poznamenal, že je prakticky nemožné takú rozsiahlu tému, akou Java a jej špecifikácia nepochybne je, opísať a vysvetliť lineárne. S trochou zjednodušenia sa dá povedať, že v nej všetko so všetkým súvisí. To vedie k nepríjemnému rozhodovaniu medzi snahou o výklad smerom od absolútnych teoretických základov k prakticky použiteľným informáciám, bohužiaľ, aspoň spočiatku na úkor užitočnosti, a medzi pokusmi podávať informácie v podobe komentárov k praktickým príkladom, čo zase pôsobí menej prehľadne a menej exaktne.

Nový seriál bude inklinovať skôr k prvému spôsobu výkladu, i keď je veľmi pravdepodobné, že sa nájde čitateľ, ktorý by radšej volil druhú cestu. K môjmu rozhodnutiu vo veľkej miere prispeli prevažne kladné ohlasy na seriál C++ pod lupou, v ktorého štýle by som rád pokračoval. Jedinou významnejšou výnimkou bude začiatok tohto pokračovania, pretože na úplnom začiatku jednoducho treba vytvoriť aspoň minimálnu bázu vedomostí.

Láskavý čitateľ mi hádam prepáči tento vzletný úvod, ktorý je v skutočnosti len chabým pokusom o ospravedlnenie sa za prípadné chybné vyriešenie klasického problému, „ako začať“. Plne si uvedomujem veľké časové rozpätie medzi jednotlivými časťami seriálu na pokračovanie a budem sa snažiť do nich zahrnúť maximum použiteľných a hlavne konzistentných informácií. Ale dosť bolo rečí, prejdime konečne k Jave.

Trieda – základ života

Prevažne teoretickú prvú časť seriálu Java pod lupou zakončil jednoduchý program "Hello, world", ktorý po spustení vypísal na obrazovku (správnejšie na štandardný výstup) tradičný text. Na osvieženie pamäti je tu jeho zdrojový text:

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

Pozorný čitateľ si iste spomenie, že zdrojový text tohto programu treba pre úspešné preloženie zapísať do súboru s názvom **HelloWorld.java**. Prečo je to tak, to si hneď povieme.

Java je výrazne objektový jazyk. Nie síce v takom rozsahu ako napríklad v Smalltalku, ale je v tomto smere oveľa dôslednejšia ako C++. Jedným z najdôležitejších rozdielov oproti C++ je fakt, že všetok kód v programe je organizovaný výlučne do tried. Triedy sú v Jave syntaktickými konštrukciami a predstavujú šablóny na tvorbu objektov. Vieme, že objekty sú charakterizované svojím stavom a zoznamom správ, na ktoré dokážu reagovať. Stav objektu v Jave je uložený v členských premenných jeho triedy, zasielanie správ sa realizuje volaním príslušných metód triedy. Každá trieda, resp. jej zdrojový kód preto obsahuje jednak deklaráciu členských premenných, jednak deklaráciu metód. Metódy sú v terminológii C++ členské funkcie tejto triedy.

A propos, termín deklarácia v programovaní znamená v podstate to isté, čo v reálnom živote. Deklarácia je vždy spojená s konkrétnym menom a predstavuje vyhlásenie, ako má prekladač ďalší výskyt tohto mena chápať. Deklarácia členskej premennej **x** tak oznamuje prekladaču, že každý výskyt identifikátora **x** v príslušnej oblasti programu znamená odkaz na túto členskú premennú a na nič iné. Podobne pre deklaráciu metód alebo tried.

Povinná organizácia kódu do tried znamená predovšetkým to, že v Jave neexistujú globálne premenné a nečlenské funkcie. Táto vlastnosť logicky vyplýva z objektovej orientácie Javy a prispieva okrem iného k zníženiu percenta programových chýb spôsobených neočakávanou a nekonzistentnou manipuláciou s premennými z rôznych miest kódu.

Triedy sú v Jave organizované do vyšších celkov, nazvaných balíky. O balíkoch zatiaľ nebudeme hovoriť, dôležité je pre začiatok len to, že triedy, ktoré majú byť viditeľné (a použiteľné) aj mimo svojho balíka, musia byť deklarované ako verejné. A tu sa okľukou dostávame k spomínanému obmedzeniu – v Jave platí pravidlo, že v jednom zdrojovom súbore sa môže nachádzať deklarácia len jedinej verejnej triedy, ktorej meno musí byť navyše totožné s menom súboru bez prípony **.java**.

Neľakajte sa, že zatiaľ nerozumiete, o čom je reč. Zo začiatku budú ukážkové príklady zapísané pomocou jedinej verejnej triedy, uloženej v jedinom zdrojovom súbore. Názov tohto súboru preto nebude z dôvodov úspory miesta odteraz uvádzaný explicitne.

Vráťme sa k pôvodnému programu. Ten pozostáva z jedinej triedy s názvom **HelloWorld**. Ako vidieť, deklarácia tejto triedy sa začína kľúčovým slovom public, ktoré udáva, že pôjde o verejnú triedu. Nad detailmi jeho použitia si zatiaľ netreba lámať hlavu, v zásade bude program fungovať aj bez neho (ale len zatiaľ!). Druhé kľúčové slovo class určuje, že ide o deklaráciu triedy. Za ním nasleduje názov deklarovanej triedy, pre ktorý platia rovnaké pravidlá ako pre pomenovávanie identifikátorov v mnohých programovacích jazykoch: prvým znakom musí byť písmeno, ďalšie znaky môžu byť aj číslice. Za písmeno sa v tomto kontexte považuje aj znak **_** (podčiarknutie).

Deklarácia členov triedy (premenných i metód) je povinne uzavretá do zložených zátvoriek (**{ a }**). Ako uvidíme neskôr, takéto zátvorky sa v Jave používajú aj v iných prípadoch; vo všeobecnosti slúžia na ohraničenie bloku deklarácií, príkazov a pod. Čitateľ znalý C++ si isto všimol, že za uzatváracou zátvorkou sa v Jave nepíše bodkočiarka (nikdy).

Trieda **HelloWorld** obsahuje jediný člen: metódu **main()**. Tento stav nie je ničím výnimočný – triedy môžu obsahovať len členské premenné, len metódy alebo oboje. Dokonca je možné deklarovať triedu, ktorá nebude obsahovať nijaké členy. Praktická použiteľnosť takej triedy je síce obmedzená, ale nájde sa prípad, keď prázdna trieda postačí.

Deklaráciu metód zatiaľ nebudeme podrobne rozoberať. Ako vidieť z príkladu, jej súčasťou môže byť niekoľko kvalifikátorov, z ktorých hneď prvý **(public)** naznačuje, že metóda **main()** bude nejakým spôsobom verejná. Kľúčové slovo static v tomto kontexte určuje, že metódu **main()** bude možné zavolať aj v prípade, že dosiaľ nebude existovať žiadna inštancia triedy **HelloWorld**. (Inštancia triedy je objekt vytvorený na základe tejto triedy; v Jave je to jednoducho premenná typu tejto triedy.) Metódy tohto typu sa nazývajú metódy triedy (class methods), na rozdiel od metód inštancií (instance methods).

Metódy sú v podstate funkciami. Funkcie môžu mať v programovacích jazykoch (rovnako ako v matematike) argumenty a typicky vracajú návratovú hodnotu. Jediným miestom, kde možno určiť počet a typ argumentov a typ návratovej hodnoty metód, je ich deklarácia. Metóda **main()** v uvedenom príklade napríklad pri svojom volaní akceptuje jeden argument pomenovaný args, ktorého typ je „pole reťazcov“ **(String[])**. Po skončení táto metóda nevracia nijakú hodnotu, čo vyjadruje kľúčové slovo void.

Kód, ktorý tvorí metódu, sa nazýva telo metódy. Toto telo musí byť uzavreté do známych zložených zátvoriek. V tele funkcie sa typicky nachádzajú deklarácie lokálnych premenných a príkazy, ktoré realizujú požadovanú funkčnosť. V príklade **HelloWorld** metóda **main()** akýmsi – zatiaľ záhadným – spôsobom zabezpečí výstup požadovaného textu na obrazovku. Bez nároku na vysvetlenie prosím čitateľa o zapamätanie si tohto spôsobu, pretože ho nájde v mnohých ukážkových príkladoch.

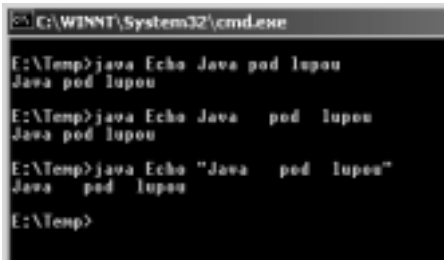
Je potrebné spomenúť, že metóda **main()** má v Jave podobné výsadné postavenie ako v C++. Predovšetkým jej deklaráciu (presnejšie jej prototyp – teda názov, počet a typ argumentov a typ návratovej hodnoty a dokonca aj oba kvalifikátory **public** a **static**) treba vždy zapísať tak ako v uvedenom príklade. Dôvod je prozaický: interpret Javy túto metódu musí vedieť nájsť, pretože práve ňou začína vykonávanie každej klasickej javovskej aplikácie (pozor, pri appletoch či servletoch je to ináč). Preto je ostatne táto metóda deklarovaná ako statická (**static**) – v dobe spustenia programu neexistuje nijaký objekt, nad ktorým by mohla byť zavolaná.

Do poľa reťazcov **args**, ktoré má metóda **main()** dostať ako argument, vloží interpret Javy argumenty programu, zadané na príkazovom riadku. I keď tu dochádza k drobnej kolízii termínov argument funkcie a argument programu, verím, že čitateľ nebude zmätený.

Na záver stručného opisu, ako vyzerá deklarácia tried a ich obsahu, je tu krátky program, ktorý možno nájsť v mnohých učebniciach Javy. Pozostáva z jedinej triedy **Echo** a po spustení vypíše jednotlivé argumenty príkazového riadka, oddelené medzerou:

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

Kto dával pozor, vie, že zdrojový kód tohto programu treba uložiť do súboru **Echo.java**. Spôsob prekladu a spustenia programu bol opísaný v predchádzajúcej časti seriá-



Obr. 2

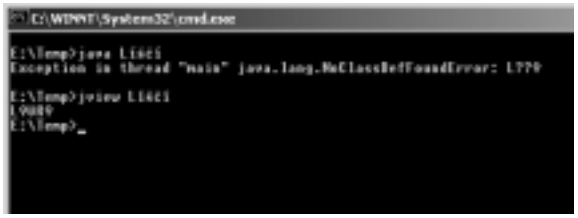
lu, ale tu je pre istotu ešte raz: program preložíme pomocou nástroja **javac**, ktorého argumentom musí byť názov prekladaného súboru, takže na príkazový riadok zapíšeme **javac Echo.java**. Výstupom prekladu bude súbor s binárnym obrazom triedy, **Echo.class**. Vykonanie programu má na starosti nástroj **java**. Jeho argumentom je vždy názov triedy obsahujúcej metódu **main()**, ale pozor: bez prípony **.class!** Program Echo preto spustíme zadaním java Echo a požadovaných argumentov. Ukážka je na **obr. 2**. Povšimnite si, prosím, že argumenty príkazového riadka sú definované ako súvislé úseky textu, oddelené bielymi medzerami. Ak chceme ako argument odovzdať reťazec obsahujúci medzery, musíme ho uzavrieť do úvodzoviek.

Lexikálna štruktúra Javy

Po praktickom začiatku prejdeme teraz k oveľa teoretickejším informáciám. Názov tohto odseku naznačuje, že sa v ňom bude hovoriť o Jave z hľadiska lexikálnej analýzy. Lexikálnu analýzu vykonáva prekladač akéhokoľvek jazyka pri preklade programu ako úplne prvú a jej cieľom je transformácia pôvodne neštruktúrovanej postupnosti znakov, uloženej v zdrojovom súbore a tvoriacej text programu, na postupnosť symbolov (lexikálnych jednotiek), ktorým prekladač rozumie. Súčasťou tejto transformácie je, samozrejme, kontrola správnosti programu z hľadiska „pravopisu“ (to keď omylom napíšeme napríklad publyc). Po lexikálnej analýze nasleduje syntaktická analýza, ktorá kontroluje dodržiavanie gramatiky jazyka a zabraňuje zápisu neplatných konštrukcií.

Java je veľmi mladý jazyk, a preto nikoho neprekvapí, že podporuje štandard **Unicode**. Čitateľ, ktorému je tento pojem neznámy, nájde bližšie informácie na http://www.unicode.org. V skratke možno povedať, že ide o štandard pre jednotné šestnásťbitové kódovanie znakov de facto všetkých existujúcich svetových jazykov, resp. ich abecied. Unicode je štandardizovaný organizáciou ISO, takže sa s ním možno stretnúť aj pod názvom ISO 10646. V praxi Unicode predstavuje náhradu ASCII a znamená koniec rôznym proprietárnym kódovaniam národných znakov a večným problémom s diakritikou. Na ďalších riadkoch čitateľ nájde zmienku o unikódoch niekoľkých špecifických znakov. Nemusí sa však ľakať, prvých 256 znakov Unicode je totožných s kódom ASCII (pravdepodobne ISO 8859-1).

Podpora Unicode v Jave je podľa špecifikácie veľmi dôsledná. Kdekoľvek v programe je dovolené používať unikódové znaky – dokonca aj v názvoch premenných, metód či samotných tried. V praxi táto podpora silne závisí od konkrétnej implementácie, pretože nie každé runtime prostredie si s unikódovými znakmi dokáže poradiť stopercentne. O tom je ostatne možné presvedčiť sa pokusom vypísať v ukážkovom príklade **HelloWorld** reťazec obsahujúci neanglické unikódové znaky. Po spustení programu v konzolovom okne (resp. DOS boxe) dostaneme namiesto vytúženého textu s diakritikou akési čudné znaky, šťaby výsledok nepochopiteľného prevodu do historickej kódovej stránky CP 852 (testované vo Windows 2000 s nastavením slovenčiny ako systémového jazyka). Preklad zdrojového textu triedy s ľubozvučným slovenským názvom **Líščí** sa síce podaril, ale pokus o jej spustenie sa skončil v prípade použitia JDK výnimkou oznamujúcou, že požadovaná trieda sa nenašla. Microsoftovská implemen-



Obr. 3

tácia Java VM je na tom lepšie, tej sa triedu Líšči podarilo úspešne nájsť, ale výstup unikódového textu na konzolu dopadol rovnako neslávne (***obr. 3***).

Čo z toho vyplýva? Unikódovým znakom v názvoch premenných, metód či tried sa treba radšej vyhnúť. Ostatne je pomerne rozšíreným zvykom používať pri ich pomenovávaní angličtinu, ktorá sa bez diakritiky našťastie zaobíde. S nesprávnym výstupom na konzolu si netreba ktovieako lámať hlavu, dnes mávajú programy v prevažnej väčšine grafické používateľské rozhranie. O podpore slovenských znakov v komponentoch GUI si určite ešte povieme, lebo ani tam situácia nie je ktovieaká ružová, ale dá sa aspoň čiastočne ovplyvniť.

Vráťme sa k lexikálnej analýze. Ako vyplýva z doteraz povedaného, zdrojový text programu je reprezentovaný postupnosťou unikódových znakov. Jej transformácia na postupnosť lexikálnych symbolov sa realizuje v troch krokoch.

Prvým krokom je preklad unikódových escape sekvencií na zodpovedajúce unikódové znaky. Unikódové escape sekvencie slúžia na zápis znakov pomocou explicitného určenia ich kódu a vo všeobecnosti vyzerajú takto: \uhhhh, kde hhhh je hexadecimálna hodnota znaku. Ako vidieť, sú podobné escape sekvenciám z jazyka C++ (ktoré v Jave tiež nájdeme), líšia sa však v mimoriadne dôležitej vlastnosti: možno ich použiť **kdekoľvek** v programe, nielen v rámci textových reťazcov. Ak v ľubovoľnom mieste programu použijeme unikódovú escape sekvenciu, výsledný efekt bude rovnaký, ako keby sme namiesto nej zapísali priamo príslušný znak. Príklad: predpokladajme, že silou-mocou chceme nazvať nejakú premennú gréckym písmenom α (alfa). V prípade, že dokážeme zapísať znak α z klávesnice, napíšeme jednoducho do programu niečo ako α = 1. Ak grécku klávesnicu nemáme k dispozícii, budeme si musieť vypomôcť escape sekvenciou: **\u03B1 = 1** (znak α má v unikóde hodnotu **03B1**). Oba zápisy sú úplne ekvivalentné a v skutočnosti sa počas prekladu v prvom kroku lexikálnej analýzy transformuje šesť znakov \, **u**, **0**, **3**, **B** a **1** pôvodného zdrojového textu na jediný znak α. Táto transformácia je, samozrejme, z hľadiska programátora úplne transparentná. Treba však podotknúť, že raz transformované znaky nepodliehajú ďalšej transformácii, a preto sa zápis **\u005Cu03B1** transformuje na šesť znakov \, **u**, **0**, **3**, **B**, **1** (**005C** je kód znaku \), ktoré sa už ďalej nenahradzujú a v tejto podobe vstupujú do ďalšieho kroku lexikálnej analýzy.

V tomto druhom kroku sa postupnosť unikódových znakov rozdelí na jednotlivé riadky. Java je v tomto smere pomerne nezávislá od platformy a úspešne sa vie vyrovnať s oddeľovačmi riadkov platforiem DOS/Windows (kombinácia CR+LF), Unix/Linux (LF) i Macintosh (CR). Pre úplnosť: znak CR (carriage return) má v Unicode hexadecimálny kód 000D, znak LF (line feed) kód 000A.

Tretím a posledným krokom je rozloženie jednotlivých riadkov na lexikálne symboly. Aby toho bol prekladač schopný, musia byť tieto symboly niečím oddelené. To niečo sú buď tzv. biele medzery, alebo komentáre.

Termín *biela medzera* sa používa v mnohých programovacích jazykoch a za svoj názov vďačí faktu, že pri tlačenom výstupe zdrojového textu na biely papier biele medzery vyzerajú iba ako biele miesta. V Jave sa za bielu medzeru považujú znaky medzera (SP, unikód 0032), horizontálny tabulátor (HT, unikód 0009), znak „posun strany“ (form feed, FF, unikód 000C) a takisto všetky oddeľovače riadkov.

Komentáre sú v Jave prakticky totožné s komentármi C++. Povolené sú dva typy: prvý sa začína dvojicou znakov **/*** a končí sa znakmi ***/**. Všetko, čo je medzi týmito znakmi, prekladač ignoruje. Tento typ nie je povolené vnárať do seba – prvý výskyt znakov ***/** ukončuje celý komentár. Druhý typ je vhodný na kratšie, jednoriadkové komentáre, začína sa dvojicou znakov **//** a za komentár sa považujú všetky nasledujúce znaky až po prvý výskyt oddeľovača riadkov. Znaky **//** nemajú nijaký význam vnútri komentára začínajúceho sa na **/***.

V mnohých učebniciach Javy možno nájsť zmienku o treťom type komentára, tzv. dokumentačnom komentári, začínajúcom sa znakmi **/****. V skutočnosti špecifikácia Javy ako jazyka nič také nepozná a prekladač nerozlišuje medzi komentármi začínajúcimi sa na **/*** a na **/****. Dokumentačný komentár je pomôcka na automatizované vytváranie dokumentácie k jednotlivým triedam, ich premenným a metódam, spracovať ho však dokáže iba samostatný nástroj **javadoc** (k dispozícii v rámci JDK). K tomuto nástroju sa ešte v priebehu seriálu vrátime.

Mnoho programátorov význam komentárov podceňuje. Je možné písať kód samodokumentujúcim spôsobom, keď sa používajú skutočne zmysluplné názvy premenných či funkcií a zdrojový text je formátovaný tak, aby bol maximálne prehľadný. To však nič nemení na skutočnosti, že človek je tvor nedokonalý a detaily implementácie projektu, na ktorom pracuje, dokáže po jeho skončení veľmi rýchlo zabudnúť. Problém nastáva, ak sa treba k starému kódu vrátiť, nedajbože ho ešte prepracovať.

Malo by byť jednou z najdôležitejších zásad programátora, že kód, ktorý tvorí, dostatočne okomentuje. V takej Jave to minimálne znamená opísať, na čo je ktorá trieda určená, aké sú vstupy a výstupy jednotlivých metód a na čo slúžia jednotlivé premenné. Programové konštrukcie je vhodné komentovať v prípade, že môže v budúcnosti dôjsť k nejasnostiam; úplne nevyhovujúcim je z tohto hľadiska komentár typu „x priradí hodnotu 1“ – takúto informáciu dokážeme zistiť aj zo samotného kódu. Komentáre majú vždy predstavovať pridanú hodnotu, v tomto prípade informačnú hodnotu.

Program HelloWorld s doplnenými komentármi by teda mohol vyzerať napríklad takto:

```
/*
 * HelloWorld
 *
 * Ukážkový príklad výstupu textu na konzolu
 */
public class HelloWorld
{
    //----------------------------------------
    // main()
    //
    // vstup:  args - argumenty príkazového riadka
    // výstup: žiaden
    //----------------------------------------
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

Lexikálne symboly

Výstupom lexikálnej analýzy pri preklade programu je postupnosť lexikálnych symbolov jazyka. Tieto symboly možno v Jave rozdeliť na päť druhov: kľúčové slová, identifikátory, literály, operátory a oddeľovače.

Kľúčové slová sú reťazce znakov, ktoré majú v Jave rezervovaný význam a nemožno ich použiť inak, napríklad ako názov premennej. S niekoľkými kľúčovými slovami sme sa už stretli: public, void, class.

Identifikátory sú svojím spôsobom vlastne mená. Mená tried, rozhraní, premenných či metód. Každá údajová či kódová entita v Jave musí mať svoje pomenovanie, ano-

Sekvencia	Unikód	Význam
\b	\u0008	backspace (BS)
\t	\u0009	horiz. tabulátor (HT)
\n	\u000A	nový riadok (LF)
\f	\u000C	nová strana (FF)
\r	\u000D	návrat vozíka (CR)
\"	\u0022	úvodzovky
\'	\u0027	apostrof
\\	\u005C	spätná lomka
\ooo	\u0000÷\u00FF	

nymné objekty (ako napríklad anonymné uniony v C++) nie sú povolené. Pravidlo pre vytváranie platných identifikátorov sme už v tomto článku uviedli, teda pre zopakovanie: identifikátorom môže byť ľubovoľná postupnosť písmen alebo číslic, prvý znak však povinne musí byť písmeno. Jazyk Java rozlišuje medzi veľkými a malými písmenami (a to nielen v identifikátoroch, ale aj v kľúčových slovách), podobne ako C++. Pod termín písmeno v rámci identifikátorov patria aj znaky **_** (podčiarknutie, unikód 005C) a **$** (unikód 0024).

Ako sme už spomenuli, v názve identifikátorov je možné použiť ľubovoľné písmená, ktoré sú zahrnuté v štandarde Unicode. Tu si však treba dávať pozor: Unicode obsahuje znaky, ktoré vyzerajú rovnako, ale kód majú odlišný. Príkladom môže byť napríklad veľké latinkové písmeno **A** (unikód 0041) a veľké grécke písmeno **A** (čítané ako alfa, unikód 0391). Z pohľadu prekladača Javy ide o dva rozdielne znaky.

Operátory sú postupnosti špeciálnych znakov, určené na vyjadrenie matematických, logických či iných operácií. Zatiaľ sme sa so žiadnym z nich nestretli, ako príklad možno uviesť notoricky známy operátor priradenia **=**.

Oddeľovače sú znaky, ktoré v súlade so svojím názvom oddeľujú úseky kódu. Ich paralelou v reálnych jazykoch sú interpunkčné znamienka. V Jave sú oddeľovačmi všetky druhy zátvoriek (obyčajné **()**, hranaté **[]**, zložené **{}**) a znaky bodka (**.**), čiarka (**,**) a bodkočiarka (**;**).

Literály

Literály možno iným slovom nazvať konštanty. Ide o konkrétne hodnoty údajových typov, priamo zapísané do programu. Ak napríklad máme niekde v kóde reťazec **a = 1**, potom **a** je identifikátor premennej a **1** je literál, konštanta celočíselného typu.

Každý literál v Jave má jednoznačne priradený svoj typ. Na základe tohto typu môžeme literály rozdeliť na celočíselné, s plávajúcou čiarkou, boolovské, znakové, reťazcové a špeciálny literál **null**.

Celočíselné literály reprezentujú hodnoty z oboru celých čísel. Java, podobne ako C++ podporuje zápis celočíselných konštánt v desiatkovej, šestnástkovej i osmičkovej sústave. Desiatkový literál je jednoducho postupnosť desiatkových číslic **0** až **9**, ktorá sa však nesmie začínať nulou. Jedinou výnimkou je samostatný literál **0**.

Šestnástkový literál sa začína dvojicou znakov **0x**, resp. **0X** a pokračuje postupnosťou šestnástkových číslic **0** až **9** a **A** až **F**, resp. **a** až **f.** Osmičkový literál sa začína znakom **0** a pokračuje postupnosťou osmičkových číslic **0** až **7**. Literály **0**, **0x0** a **00** sú v princípe odlišné, hoci všetky reprezentujú tú istú hodnotu.

Celočíselné literály sú v zásade dvoch typov: **int** a **long** (zatiaľ ešte nehovoríme o typoch premenných, hoci názvy typov sú totožné). Literály typu int majú rozsah 32 bitov. Najväčší desiatkový literál typu int je teda 2147483648 ($2^{31} - 1$). Všimnite si, že o záporných hodnotách sa tu nehovorí, na zmenu znamienka slúži totiž unárny operátor –, ktorý nie je súčasťou literálu. Najväčší kladný šestnástkový literál je **0x7FFFFFFF**, osmičkový **017777777777**, najväčší záporný šestnástkový je **0x80000000**, osmičkový potom **020000000000**. Pri iných ako desiatkových literáloch je „zápornosť“ vzhľadom na použitý znamienkový bit ich súčasťou. Rozsah týchto literálov je, pochopiteľne, $-2^{31} \div 2^{31} - 1$.

Literály typu long majú rozsah 64 bitov. Keďže v prípade hodnôt menších ako 2^{31} nie je možné jednoznačne určiť ich typ, musia sa takéto literály typu long končiť znakom **l** alebo **L**. Čo sa týka maximálnych hodnôt, platí v podstate to isté, čo pri type int, len namiesto hranice 2^{31} treba uvažovať hranicu 2^{63}. Rozsah literálov typu long je $-2^{63} \div 2^{63} - 1$.

Príklady celočíselných literálov: **0, 123, 0xCAFEBABE, 0377** (pre typ **int**), **0L**, **2000L**, **0xFF00L**, **0177777L** (pre typ **long**).

Ďalším typom sú literály s *plávajúcou čiarkou* (tzv. floating-point literály). Tieto literály slúžia na zápis neceločíselných, desatinných hodnôt. Ich zápis sa vo všeobecnosti skladá z celej časti, desatinnej bodky (znak **.**), desatinnej časti a exponentu. Celá aj desatinná časť je postupnosťou desiatkových číslic **0** až **9**, exponent sa skladá z písmena **E** alebo **e**, nasledovaného číslom s prípadným znamienkom. Zo zápisu je možné vynechať celú časť alebo desatinnú časť, ale nie obe naraz. Exponent je nepovinný, ak je však prítomný, možno vynechať desatinnú časť aj desatinnú bodku.

Reprezentácia hodnôt s plávajúcou čiarkou sa riadi štandardom IEEE 754. Zápis s použitím exponentu sa nazýva aj vedeckou notáciou a umožňuje samostatné vyjadrenie mantisy, pozostávajúcej z platných číslic daného čísla, a exponentu, určujúceho polohu desatinnej čiarky. Tvar **1.2345E11** tak predstavuje číslo **1.2345 × 1011**.

Aj literály s plávajúcou čiarkou sa delia na dva typy: **float** a **double**. Typ float má šírku 32 bitov a rozsah hodnôt $1{,}4 \times 10^{-45} \div 3{,}4 \times 10^{38}$ v kladnej oblasti (v zápornej oblasti je rozsah rovnaký, len s opačným znamienkom). Typ double má šírku 64 bitov a rozsah hodnôt $4{,}9 \times 10^{-324} \div 1{,}8 \times 10^{308}$ v kladnej oblasti. Na explicitné rozlíšenie hodnôt oboch typov slúžia prípony **f**, **F** (float) a **d**, **D** (double). V prípade, že príponu neuvedieme, implicitne sa berie typ double. Výhodou použitia explicitnej prípony je možnosť vynechať desatinnú bodku, desatinnú časť i exponent súčasne.

Príklady literálov s plávajúcou čiarkou: **0.0F, .78F, 3e-9F** (typ **float**), **2.99e8, 99D** (typ **double**).

Boolovské literály sú iba dva, **true** a **false**. Predstavujú logické hodnoty používané v dvojstavovej boolovskej logike. Literál true reprezentuje logickú jednotku, literál **false** logickú nulu.

Znakové literály reprezentujú znaky štandardu Unicode. Každý znakový literál musí pozostávať z jediného znaku uzavretého v apostrofoch. Jeho šírka je 16 bitov. Na vyjadrenie špeciálnych znakov slúžia escape sekvencie. (Pozor, nejde o unikódové escape sekvencie, ktoré sú ekvivalentným spôsobom zápisu ľubovoľného znaku zdrojového textu programu.) Zoznam escape sekvencií je v tabuľke. Sekvencia v poslednom riadku tabuľky slúži na explicitné vyjadrenie kódu znaku pomocou osmičkovej hodnoty. Jej rozsah je však obmedzený na prvých 256 znakov Unicode, preto sa na tieto účely odporúča použiť unikódové escape sekvencie.

Súčasťou znakového literálu nesmie byť nijaký oddeľovač riadka. Je preto chybou uviesť do programu literál **'\u000A'**, pretože vnorená escape sekvencia sa nahradí znakom LF ešte počas lexikálnej analýzy a výsledný efekt bude rovnaký, ako keby na danom mieste bol vložený klasický prechod na nový riadok. Na tieto účely slúžia práve štandardné escape sekvencie, takže správny zápis je **'\n'**.

Príklady znakových literálov: **'z'**, **'#'**, **'\t'**, **'\177'**, **'\u0427'**, **'?'**.

Reťazcové literály sú postupnosti unikódových znakov, uzavreté v úvodzovkách. V rámci reťazcových literálov je povolené používať tie isté escape sekvencie ako v znakových literáloch. Ani v reťazcových literáloch nie je povolené používať oddeľovače riadkov.

Príklady reťazcových literálov: **""**, **"hello"**, **"Prvý riadok\nDruhý riadok"**.

Konečne posledným typom literálu je samostatný literál **null**. Tento literál v zásade nereprezentuje nijakú reálnu hodnotu a používa sa pri referenčných premenných objektového typu na vyjadrenie skutočnosti, že daná premenná neodkazuje na nijaký existujúci objekt, namiesto toho ukazuje „nikam“.

Zhrnutie

Úspešne sme sa prepracovali na záver druhej časti seriálu Java pod lupou, ktorá bola aspoň spolovice výrazne teoretická. V nasledujúcej časti prídu na rad okrem iného premenné a ich typy.

3.časť: Typy, výrazy, operátory

V predošlej časti sme sa dozvedeli o tom, že program v Jave pozostáva z kolekcie tried, na základe ktorých počas behu programu vznikajú objekty. Tie medzi sebou navzájom komunikujú pomocou správ. Konkrétny spôsob realizácie správ je v prípade Javy klasické volanie funkcií, ktoré v tomto prípade nazývame metódami. Každá metóda, ako aj každá premenná je súčasťou nejakej triedy.

Druhá polovica predchádzajúcej časti opisovala program z lexikálneho hľadiska – komentáre, oddeľovače a predovšetkým klasifikáciu lexikálnych jednotiek, stavebných jednotiek programu, akoby korálikov, ktoré jeden po druhom navlieka programátor na šnúru, tvoriac tak použiteľný program.

V tretej časti seriálu o Jave sa čitateľ dozvie o údajových typoch, ktoré v Jave možno používať, a o spôsobe ich použitia vo forme výrazov. Takisto tu nájde opis najjednoduchších operátorov, realizujúcich základné matematické operácie, ako sčítanie, násobenie a pod.

Premenné

Skôr než začneme hovoriť o typoch jazyka Java, bolo by nanajvýš vhodné uviesť niekoľko praktických poznámok o *premenných*. Interpretácia pojmu premenná je v programovacích jazykoch máličko odlišná od klasickej matematickej predstavy. Premenná v programe predstavuje miesto na uchovanie údajov (ang. storage location). Každej premennej možno priradiť niekoľko atribútov: predovšetkým je to jej názov, ktorý ju jednak odlišuje od iných premenných, jednak poskytuje programátorovi jednoznačný spôsob, ako sa na ňu kdekoľvek v programe odvolať. Medzi ďalšie atribúty, o ktorých bude podrobnejšie reč v budúcich častiach, patrí napríklad doba trvania, rozsah platnosti a podobne.

Aby čitateľ mohol sledovať príklady uvedené v nasledujúcich odsekoch, treba aspoň v stručnosti ukázať, akým spôsobom do programu zapíšeme, že chceme pracovať s takou a takou premennou, takého a takého typu. Základná schéma je veľmi jednoduchá: deklarácia premennej je tvorená trojicou typ – názov – inicializátor. Ak teda budeme chcieť deklarovať napríklad premennú `val` typu `int` (hneď si o tomto type povieme viac) a súčasne jej priradiť počiatočnú hodnotu 123, vložíme do programu riadok:

```
int val = 123;
```

Bodkočiarka za deklaráciou je povinná. Znalosť tejto schémy nám postačí do doby, keď sa k deklaráciám premenných (a nielen tých) vrátime.

Údajové typy v Jave

Nech si zoberieme akýkoľvek aspoň trochu zmysluplný počítačový program, vždy dospejeme k zisteniu, že základom jeho činnosti je práca s údajmi. Ako sa dá predpokladať, vo všeobecnosti to budú rôznorodé údaje: číselné či textové, štruktúrované i neštruktúrované. Pojem *údajový typ* v programovacích jazykoch túto rôznorodosť kvalifikuje – o rôznych údajoch hovoríme, že sú rôzneho typu. Každý údajový typ má svoje špecifické vlastnosti,

predovšetkým je nevyhnutné poznať množinu hodnôt, ktoré obsiahne, a súbor operácií, ktoré nad týmito hodnotami možno vykonávať.

Hneď na úvod treba podotknúť, že Java je jazyk s prísnou typovou kontrolou. Každá premenná a každý výraz má svoj typ a tento typ je vždy možné stanoviť už počas prekladu. Takýto prístup pomáha minimalizovať počet chýb, ktoré sa prejavia až počas používania programu. Množstvo chýb, plynúcich z nekorektnej práce s údajmi (ako sa to, žiaľ, stáva v C++), dokáže zachytiť a ohlásiť už kompilátor.

Údajové typy v Jave možno rozdeliť do dvoch kategórií: *primitívne typy a referenčné typy*. Špeciálnym typom zostáva nulový typ, ktorý nemá nijaké meno, nie je ho možné použiť na deklaráciu premenných a jedinou platnou hodnotou tohto typu je nulový odkaz, reprezentovaný literálom **null**. Ako sme sa dozvedeli minulý mesiac, hodnotu null obsahuje premenná referenčného (objektového) typu, ktorá neukazuje na nijaký existujúci objekt. Nulový typ nie je totožný s typom void z jazyka C++. Java obsahuje kľúčové slovo void, no to sa používa bez výnimky na deklaráciu neexistujúcej návratovej hodnoty metód.

Primitívne typy

Prvou skupinou údajových typov v Jave sú primitívne typy. Medzi ne patria všetky číselné typy a typ boolean. Číselné typy možno rozdeliť na celočíselné a s pohyblivou čiarkou. Celočíselných typov je päť: **byte**, **short**, int, long a char, typy s pohyblivou čiarkou sú dva: float a double.

Typ **byte** má šírku 8 bitov, najvyšší bit je znamienkový. Rozsah hodnôt, ktoré tento typ obsiahne, je -128 až 127. Jeho ekvivalentom v C++ je typ **unsigned char**.

Príklad deklarácie premennej typu byte:

```
byte b = 0x3F;
```

Ďalší typ **short** je v Jave široký vždy 16 bitov (opäť so znamienkom). Hodnoty tohto typu patria do intervalu -32768 až 32767. Podobný typ **short int** v C++ je oproti tomu implementačne závislý, i keď na väčšine implementácií má takisto 16 bitov. Príklad deklarácie:

```
short k = -1500;
```

Typ int má šírku 32 bitov, čo mu umožňuje uchovávať hodnoty v rozsahu od -2^{31} po $2^{31}-1$. Najvyšší bit aj v tomto prípade slúži na zobrazenie znamienka. Rovnako pomenovaný typ v C++ má podľa tradície šírku totožnú so šírkou slova procesora, čo je dnes zatiaľ ešte 32 bitov (to, samozrejme, neplatí v archaickom DOS-e, tam sa musíme uspokojiť so 16 bitmi). Príklad premennej typu int:

```
int n = 12345678;
```

Najširším číselným typom v Jave je typ **long** – celých 64 bitov. Aj tento typ umožňuje ukladanie záporných hodnôt a jeho rozsah je logicky -2^{63} až $2^{63}-1$. V C++ nájdeme typ s rovnakým názvom, len s menším, 32-bitovým rozsahom. Príklad deklarácie premennej typu **long**:

```
long l = -256L;
```

Čitateľ si isto spomenie, ako sme v predošlej časti seriálu okrem iného hovorili o celočíselných literáloch. Tie sa v porovnaní s vymenovanými celočíselnými typmi obmedzujú len na typy **int** a **long**. Na osviežení pamäti – celočíselnému literálu môžeme typ **long** explicitne priradiť pomocou prípony **l** či **L**. Literály typu **byte** alebo **short** v Jave nenájdeme; ak treba, vždy môžeme použiť literál typu **int**, len musíme dodržať povolený rozsah hodnôt. Nasledujúca deklarácia je teda chybou:

```
byte b = 500;
```

Piatym celočíselným typom, stojacim trochu mimo predchádzajúcich štyroch, je typ **char**. Ako naznačuje jeho názov, použijeme ho na uloženie znakových údajov. Z prvej časti si pamätáme, že Java podporuje štandard **Unicode**, šírka typu **char** je z tohto dôvodu 16 bitov. Na rozdiel od štyroch znamienkových typov je typ **char** bez znamienka a jeho rozsah je teda **0** až **65535** alebo presnejšie '**\u0000**' až '**\uFFFF**'. Premenné typu **char** je, samozrejme, možné inicializovať celočíselným i znakovým literálom. Oba nasledujúce príklady sú preto ekvivalentné, i keď z praktických dôvodov sa preferuje ten prvý:

```
char c = 'Z';
char c = 90;
```

Celočíselné typy sú vhodné len na reprezentáciu celočíselných hodnôt. Naproti tomu typy s pohyblivou čiarkou použijeme v prípade, že potrebujeme pracovať s číselnými údajmi mimo oboru celých čísel. Je pomerne bežnou chybou domnievať sa, že pomocou typov s pohyblivou čiarkou zobrazujeme reálne čísla. Vzhľadom na konečnú presnosť zobrazenia ide, pochopiteľne, len o racionálne čísla.

Java poskytuje dva typy s pohyblivou čiarkou: **float** a **double**. Oba typy vyhovujú štandardu IEEE 754 – prvý poskytuje jednoduchú presnosť a šírku 32 bitov, druhý dvojnásobnú presnosť a šírku 64 bitov. V súlade so zmieneným štandardom umožňujú oba typy reprezentovať aj určité špeciálne hodnoty: kladnú a zápornú nulu, kladné a záporné nekonečno a zvláštnu hodnotu NaN (z anglického Not a Number), ktorá sa používa na reprezentáciu určitých neplatných operácií, ako napríklad delenie nuly nulou. V programe sa môžeme na túto hodnotu odvolávať pomocou konštantných premenných **Float.NaN** a **Double.NaN**.

Rozsahy hodnôt typov **float** a double sa rozprestierajú na obe strany od začiatku číselnej osi. Každá hodnota s pohyblivou čiarkou je vnútorne reprezentovaná dvojicou mantisa – exponent (výsledná hodnota je daná výrazom **mantisa × 2^exponent**). V type **float** je pre mantisu vyhradených 24 bitov a pre exponent 8 bitov, takže celkový rozsah je rádovo v rozmedzí -2^{127} až -2^{-126} a 2^{-126} až 2^{127}. Pre ilustráciu, číslo 2^{127} je rádovo približne rovné 10^{38}. Typ double poskytuje pre mantisu 53 bitov a pre exponent 11 bitov, čo dáva rozsah v rozpätí -2^{1023} až -2^{-1022} a 2^{-1022} až 2^{1023}. Číslo 2^{1023} je pri dekadickom základe približne rovné 10^{308}.

Binárna reprezentácia čísel s pohyblivou čiarkou počíta s tzv. *normalizovanou mantisou*. Normalizáciu môžeme ilustrovať na nasledujúcom príklade: ak by sa na ukladanie hodnôt používala desiatková sústava, číslo **123 × 10^4** by bolo treba pred uložením normalizovať na tvar **1,23 × 10^6**. Java okrem normalizovaných čísel vyžaduje v súlade so štandardom IEEE 754 podporu aj pre tzv. denormalizované hodnoty. Týmto názvom označujeme čísla také malé, že ich už normalizovať nemožno. V dekadickom ponímaní by išlo napríklad o hodnotu **0,00123 × 10^-308**. Zmena mantisy z **0,00123** na **1,23** by si totiž v tomto prípade vyžadovala zníženie exponentu na hodnotu **-311**, čo, samozrejme, nejde.

Príklad deklarácie premennej typu **float** i typu **double**:

```
float f = 1.234F;
double d = -3.894E-21;
```

Pozorný čitateľ si spomenie, že literály typu **float** explicitne určuje prípona **f**, resp. **F**.

Posledný primitívny typ **boolean** slúži na reprezentáciu logických hodnôt. Najjednoduchší zo všetkých typov, pozostáva len z jediného bitu. Rozsah typu **boolean** tvoria len dve hodnoty, reprezentované literálmi **true** a **false**. Na rozdiel od jazyka C++ typ **boolean** v Jave nemožno len tak jednoducho pretypovať na celočíselný typ. O tom si však ešte povieme. Príklad deklarácie premennej typu **boolean**:

```
boolean flag = false;
```

Špecifickou vlastnosťou primitívnych typov v Jave je skutočnosť, že každá premenná či výraz primitívneho typu predstavuje samostatnú hodnotu. Tento na pohľad možno triviálny fakt možno demonštrovať nasledujúcim jednoduchým príkladom:

```
public class Test
{
   public static void main(String[] args)
   {
      int a = 1;
      int b = a;
      a = 2;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }
}
```

Po preložení a spustení tohto krátkeho programu by ste mali na obrazovke dostať nasledujúci výstup:

```
a = 2;
b = 1;
```

Je teda očividné, že hoci sme premennú b inicializovali pomocou premennej a (zápisom **int b = a**), obe premenné sú inak samostatné a navzájom nezávislé, takže neskoršia zmena jednej z nich neovplyvní druhú. Toto je fundamentálna vlastnosť primitívnych typov; o chvíľu uvidíme, že pri referenčných typoch je situácia odlišná.

Referenčné typy

Druhú skupinu typov v Jave tvoria *referenčné typy*. Sem patria triedy, rozhrania a polia. Všetky tri druhy referenčných typov považujeme za objektové typy (v zmysle OOP). O triedach sme si základné informácie povedali v predchádzajúcich častiach, širší pohľad na ne príde na rad v blízkej budúcnosti. *Rozhrania* sú zvláštnym konceptom, používaným najčastejšie ako náhrada za neexistujúcu jazykovú podporu pre viacnásobnú dedičnosť. V určitom zmysle možno javovské rozhrania chápať ako vzdialeného príbuzného abstraktným triedam v C++. Toto prirovnanie však nie je celkom presné, pretože ako uvidíme, triedy možno explicitne označiť ako abstraktné aj v Jave. O rozhraniach bude reč neskôr.

Zostáva teda tretí referenčný typ, a to **pole**. Na rozdiel od C++, kde je prakticky celá kontrola korektnej práce s prvkami poľa ponechaná na programátorovi, je v Jave typ **pole** značne sofistikovaný a výrazne bezpečnejší. Každé pole je vytvárané dynamicky, pri prístupe k jednotlivým jeho prvkom program (presnejšie run-time prostredie) kontroluje, či nie je zadaný index mimo hraníc, a práca s viacrozmernými poľami je oveľa prehľadnejšia a flexibilnejšia.

Poliam bude venovaná samostatná časť seriálu, preto nasleduje len krátka ukážka spôsobu deklarácie poľa a práce s jeho prvkami:

```
int[] a = { 1, 2, 3, 4, 5 };
a[2] = 333;
System.out.println("a[4] = " + a[4]);
```

Výstup tohto miniprogramu bude nasledujúci:

```
a[4] = 5
```

V uvedenom fragmente deklarujeme pole celých čísel s názvom **a**, obsahujúce päť prvkov. Indexovanie jednotlivých prvkov sa realizuje pomocou hranatých zátvoriek, prvky číslujeme od nuly. Na príklade vidieť, že v princípe sa deklarácia poľa spojená s inicializáciou nelíši od toho, čo sme videli doteraz. Rozdiel nastane v prípade, že sa rozhodneme inicializačnú časť deklarácie vynechať. Ale o tom neskôr. Za pozornosť ešte stojí, že typ „pole celých čísel typu **int**“ označujeme veľmi intuitívne ako **int[]**. Je však možné použiť aj deklaráciu v štýle C++:

```
int a[] = { 1, 2, 3, 4, 5 };
```

Pozornému čitateľovi isto neunikla skutočnosť, že dosiaľ sme nehovorili o type, ktorý by reprezentoval textové reťazce. V jazyku C++ je reťazec znakov realizovaný ako pole znakov, ukončené nulovou hodnotou. Tento spôsob je síce pomerne efektívny, ale vzhľadom na nijako neošetrovanú prácu s poľami aj značne náchylný na veľmi ťažko odhaliteľné chyby. A to už vôbec nehovoríme o praktickej nutnosti používať pri práci s reťazcami (a poľami všeobecne) ukazovatele, ktoré sa pri len trochu zložitejších údajových štruktúrach rýchlo stanú nočnou morou menej skúseného programátora.

Java prináša zničenému vývojárovi vytúženú úľavu v podobe realizácie textových reťazcov ako objektov, konkrétne inštancií triedy **String**. Táto trieda okrem faktu, že oslobodzuje programátora od starostí o detaily skutočnej implementácie, poskytuje aj širokú škálu metód na prácu s reťazcami. Trieda **String** má určitým spôsobom výnimočné postavenie, pretože sa od ostatných tried líši v drobných detailoch použitia. Konkrétne každý výskyt reťazcového literálu v programe automaticky predstavuje inštanciu triedy **String** a navyše je na spájanie reťazcov (a nielen reťazcov medzi sebou, ale aj reťazca a ľubovoľnej hodnoty; pozri uvedené príklady) možné použiť operátor **+**.

Tu je príklad práce s premennou typu **String**:

```
String s = "5 * 13 = ";
int x;
x = 5 * 13;
System.out.println(s + x);
```

Program po spustení vypočíta súčin hodnôt 5 a 13 a vypíše výsledok:

```
5 * 13 = 65
```

To, čo odlišuje referenčné typy od primitívnych, je fakt, že každá premenná alebo výraz referenčného typu obsahuje v skutočnosti odkaz na objekt uložený kdesi inde v pamäti. Opäť si môžeme vypomôcť analógiou s C++ – referenčné typy sú v zásade totožné s referenciami jazyka C++, s tým rozdielom, že v C++ možno deklarovať aj referencie na typy, ktoré v Jave považujeme za primitívne.

Zhrňme teda rozdiely: v premennej primitívneho typu je uložená priamo požadovaná hodnota. Ak máme napríklad premennú typu **long**, bude táto premenná v pamäti uložená ako postupnosť 64 bitov, obsahujúcich binárnu reprezentáciu jej obsahu. Naproti tomu v premennej referenčného typu je uložená len adresa miesta v pamäti, kde nájdeme objekt, ktorý táto premenná „obsahuje“. Inými slovami, nezávisle od svojho typu bude každá referenčná premenná v pamäti zaberať rovnaké miesto, so šírkou závisiacou od šírky pamäťovej adresy na danej implementácii Java VM. Vzhľadom na to, že akýkoľvek prístup k referenčnej premennej automaticky pracuje s odkazovaným objektom, nie je možné a z bezpečnostného hľadiska ani žiaduce uloženú adresu zistiť.

Praktickým dôsledkom odlišnosti medzi primitívnymi a referenčnými typmi je skutočnosť, že sa dve rôzne referenčné premenné môžu odkazovať na rovnaký objekt v pamäti. Ukážme si to na príklade:

```
public class Test
{
   public static void main(String[] args)
   {
      int[] a = { 1, 2, 3 };
      int[] b = a;
      a[1] = 222;
      System.out.println("a[1] = " + a[1]);
      System.out.println("b[1] = " + b[1]);
   }
}
```

Ak skúsite preložiť a spustiť tento program, dostanete nasledujúci výstup:

```
a[1] = 222;
b[1] = 222;
```

Je očividné, že premenné a aj b odkazujú na to isté pole – pri zmene prvku **a[1]** súčasne došlo k zmene prvku **b[1]**.

Výrazy a operátory

Spomínali sme už, že prakticky každý program nejakým spôsobom pracuje s údajmi. Každý krok spracovania údajov treba nejakým spôsobom zapísať. Ak napríklad počítame korene kvadratickej rovnice, musíme do programu vložiť jednak postup výpočtu diskriminantu, jednak výpočet samotných koreňov a pri skutočne univerzálnom programe nezabudneme zistiť, či rovnica náhodou nemá komplexné korene. Všetky tieto matematické (a iné) operácie zapisujeme prakticky v každom programovacom jazyku pomocou výrazov.

Každý výraz sa skladá z operátorov a ich operandov. Operátor je symbol, ktorý určuje typ operácie vykonávanej s príslušnými údajmi, ako napríklad **+** pre sčítanie. Údaje, ktoré do operácie vstupujú, sa nazývajú operandy daného operátora. Operátory môžu mať jeden, dva či tri operandy. Podľa toho ich delíme na unárne, binárne a ternárne. Operátor **+** je napríklad binárny operátor, pretože ním sčítavame dve čísla.

Výskyt akéhokoľvek výrazu v programe vedie k jeho vyhodnoteniu. Pod vyhodnotením výrazu sa rozumie postupná aplikácia jednotlivých operátorov na ich operandy, až kým sa výraz nezredukuje na jedinú hodnotu, ktorá bude hodnotou celého výrazu. Tak napríklad vo výraze **2 + 3 * 4** sa aplikuje operátor * na operandy **3** a **4**, čím dostaneme hodnotu **12**, a v druhom kroku sa operátor **+** aplikuje na operandy **2** a **12**. Hodnotou výrazu **2 + 3 * 4** bude teda **14**.

Poradie, v ktorom sa operátory aplikujú, je dané ich prioritou. Tabuľku priorít jednotlivých operátorov si uvedieme po ich postupnom prebratí. Explicitne môžeme hierarchiu priorít zmeniť pomocou zátvoriek, podobne ako v matematike. Operandy každého operátora sa v Jave vyhodnocujú ***vždy*** zľava doprava. Tento princíp možno ilustrovať nasledujúcim príkladom:

```
int i = 3;
int j = (i = 2) * i;
System.out.println(j);
```

Po preložení a spustení program vypíše:

```
4
```

V príklade chceme premennej **j** priradiť hodnotu výrazu **(i = 2) * i**. Vidíme, že máme vo výraze operátor násobenia * s dvoma operandmi. Vzhľadom na uvedené pravidlo sa najprv vyhodnotí operand **(i = 2)**, ktorý priradí premennej **i** hodnotu **2**. Výslednou hodnotou tohto operandu bude priraďovaná hodnota, teda **2**. Až potom sa začne vyhodnocovať druhý operand **i**, ktorého hodnota bude triviálne hodnotou premennej **i**. Ale tej sme predsa pred chvíľou priradili hodnotu **2**! Preto výslednou hodnotou druhého operandu bude **2** a výsledok celého výrazu bude rovný číslu **4**.

Aditívne operátory

Prvými dvoma operátormi, o ktorých si povieme, sú tzv. aditívne operátory **+** a **-**. Oba operátory sú binárne a slúžia na realizáciu tých najjednoduchších matematických operácií – sčítania a odčítania. Majú rovnakú prioritu a asociujú sa zľava doprava. Asociatívnosť operátorov vyjadruje postupnosť vyhodnocovania operátorov s rovnakou prioritou. Napríklad výraz **2 + 3 + 4** sa vzhľadom na asociatívnosť zľava doprava operátora **+** vyhodnotí tak, akoby sme ho zapísali **(2 + 3) + 4**. Keby sa operátor **+** asocioval opačne, sprava doľava, výraz by sa vyhodnocoval ako **2 + (3 + 4)**. Zátvorky, pochopiteľne, explicitne určujú prioritu.

Pri sčítaní a odčítaní celých čísel môže dôjsť k pretečeniu či podtečeniu povoleného rozsahu. Tento stav sa nijako nesignalizuje a je len na programátorovi, aby s ním v prípade potreby počítal. Ako príklad možno uviesť takýto kód:

```
int a = 1000000000;
int b = 2000000000;
System.out.println(a + b);
```

Výstupom programu bude:

```
-1294967296
```

Skutočný výsledok sčítania hodnôt premenných **a** a **b** je mimo rozsahu typu **int**. Výsledok, ktorý dostaneme ako výstup programu, vznikne ignorovaním pretečených bitov. Podobná situácia nastáva pri odčítaní.

Pre sčítanie čísel s pohyblivou čiarkou platia určité pravidlá (podľa štandardu IEEE 754):

- Ak je niektorým operandom hodnota NaN, výsledok bude takisto NaN.
- Súčtom dvoch nekonečien s rovnakým znamienkom je nekonečno s týmto znamienkom.
- Súčtom dvoch nekonečien s opačným znamienkom je NaN.
- Súčtom nekonečna a konečnej hodnoty je to isté nekonečno.
- Súčtom dvoch núl s rovnakým znamienkom je nula s týmto znamienkom.
- Súčtom dvoch núl s opačným znamienkom je kladná nula.
- Súčtom nuly a konečnej hodnoty je tá istá konečná hodnota.
- Súčtom dvoch konečných hodnôt s rovnakou absolútnou hodnotou a opačným znamienkom je kladná nula.

Tieto pravidlá sa súčasne vzťahujú na odčítanie, pretože pre číselné typy bez výnimky platí, že výraz **a – b** dáva rovnaký výsledok ako **a + (-b)**.

Oba operandy operátorov **+** aj **-** musia byť primitívne číselné typy, s jedinou výnimkou, o ktorej sme už čiastočne hovorili. Ak je aspoň jedným z operandov operátora **+** typ **String**, prevedie sa prípadne druhý operand takisto na typ **String** a výsledkom bude spojenie oboch reťazcov.

Multiplikatívne operátory

Do tejto triedy patria operátory *, **/** a **%**, slúžiace po rade na výpočet súčinu, podielu a zvyšku po delení dvoch čísel. Všetky tri operátory majú rovnakú prioritu a asociujú sa zľava doprava. Operandy týchto operátorov musia byť povinne číselného primitívneho typu.

Pre násobenie čísel s pohyblivou čiarkou platia tieto pravidlá:

- Ak je niektorým operandom operátora * hodnota NaN, výsledok bude takisto NaN.
- Ak výsledkom násobenia nie je hodnota NaN, znamienko výsledku bude podobne ako v matematike kladné, ak majú oba operandy rovnaké znamienko, a záporné, ak majú znamienko opačné.
- Súčinom nekonečna a nuly je NaN.
- Súčinom nekonečna a konečnej hodnoty je nekonečno.

Pri výpočte súčinu môže dôjsť k pretečeniu či podtečeniu; pretečené bity sa opäť ignorujú.

Operátor delenia sa správa trochu rozdielne v závislosti od typu svojich operandov. Výsledkom delenia dvoch ***celých*** čísel je podobne ako v C++ opäť celé číslo, ktoré vznikne zo skutočného podielu odstránením desatinnej časti. Ukážka:

```
int a = 13;
int b = 3;
System.out.println(a / b);
```

Tento úsek programu vypíše:

```
4
```

Ďalej, ak sa pokúsime deliť akékoľvek ***celé*** číslo nulou,

dôjde k chybe, tzv. výnimke. (Výnimky v Jave slúžia na signalizáciu chybových stavov. Povieme si o nich viac v budúcnosti.) Ak však budeme deliť nulou číslo s pohyblivou čiarkou, k chybe nedôjde. To je ostatne jedným z dôsledkov nasledujúcich pravidiel:

- Ak je niektorým operandom operátora **/** hodnota NaN, výsledok bude takisto NaN.
- Ak výsledkom delenia nie je hodnota NaN, znamienko výsledku bude kladné, ak majú oba operandy rovnaké znamienko, a záporné, ak majú znamienko opačné.
- Delenie nekonečna nekonečnom dáva výsledok NaN.
- Delenie nekonečna konečnou hodnotou dáva opäť nekonečno.
- Delenie konečnej hodnoty nekonečnom dáva nulu.
- Výsledkom delenia dvoch núl je NaN, výsledkom delenia nuly a konečnej hodnoty je opäť nula.
- Konečne delenie konečnej hodnoty nulou dáva ako výsledok nekonečno.

Pretečenie či podtečenie sa opätovne ignoruje.
Tretí operátor % slúži na výpočet zvyšku po delení dvoch čísel. Na rozdiel od C++ jeho operandmi môžu byť aj čísla s pohyblivou čiarkou. Výsledkom aplikácie tohto operátora je hodnota, ktorá vyhovuje rovnosti **(a / b) * b + (a % b) = a**. Znamienko zvyšku je zhodné so znamienkom delenca (t. j. ľavého operandu).

Pri snahe o výpočet zvyšku po delení celého čísla nulou nastane výnimka, pri výpočte zvyšku po delení čísel s pohyblivou čiarkou platia tieto pravidlá:

- Ak je niektorým operandom operátora % hodnota NaN, výsledok bude takisto NaN.
- Ak výsledkom výpočtu zvyšku nie je hodnota NaN, znamienko výsledku je zhodné so znamienkom delenca.
- Ak je delencom nekonečno alebo ak je deliteľom nula (alebo v oboch prípadoch), výsledok bude NaN.
- Ak je delencom konečná hodnota a deliteľom nekonečno, výsledok je rovný delencu.
- Ak je delencom nula a deliteľom konečná hodnota, výsledok bude nulový.

Pri výpočte zvyšku k pretečeniu či podtečeniu nemôže dôjsť. Príklad všetkých kombinácií kladného a záporného operandu:

```
System.out.println( 13 %  3);
System.out.println( 13 % -3);
System.out.println(-13 %  3);
System.out.println(-13 % -3);
```

Výstupom takéhoto úseku programu budú čísla:

```
1
1
-1
-1
```

Unárne plus a mínus
Ďalšie dva operátory sú veľmi triviálne: ide o unárne plus **(+)** a unárne mínus **(-)**. Oba operátory sa asociujú sprava doľava. Ich jediný operand musí mať číselný primitívny typ. *Unárne plus* vo svojej podstate nerobí nič okrem toho, že podobne ako iné aritmetické operátory automaticky rozširuje typ svojho operandu. Táto konverzia je však prirodzená a ešte si o nej budeme hovoriť. Praktické použitie unárneho **+** sa obmedzuje azda len na explicitné zdôraznenie kladnosti nejakej konštanty a pod.

Unárne mínus v zásade mení znamienko svojho operandu. To platí tak pre celočíselné typy, ako aj pre typy s pohyblivou čiarkou. Špeciálne hodnoty, ako nula či nekonečno, nevykazujú v tomto prípade nijaké odlišnosti. Jediným rozdielom je aplikácia unárneho - na hodnotu NaN, vtedy je výsledkom opäť NaN.

Operátory ++ a —
Posledné dva operátory, o ktorých bude v tejto časti reč, slúžia na inkrementáciu **(++)** a dekrementáciu **(—)** premenných. Inkrementovať premennú znamená zvýšiť jej hodnotu o **1**, dekrementácia predstavuje, naopak, zníženie hodnoty o **1**. Oba operátory sú unárne a existujú v dvoch variantoch: prefixovom (operátor je pred operandom) a postfixovom (operátor je za operandom). Ich jediným operandom musí ***povinne*** byť premenná číselného typu.

Výsledný efekt aplikácie prefixového a postfixového variantu operátora **++**, resp. **--** je rovnaký: inkrementácia, resp. dekrementácia. Oba varianty však sa líšia výslednou hodnotou výrazu obsahujúceho príslušný operátor. Pri použití postfixového variantu je výsledkom výrazu hodnota premennej ***pred*** inkrementáciou/dekrementáciou, pri použití prefixového variantu je výsledkom výrazu hodnota premennej ***po*** inkrementácii/dekrementácii. Ukážme si to na príklade:

```
int a = 3;
int b = 3;
System.out.println(a++);
System.out.println(++b);
System.out.println(a);
System.out.println(b);
```

Uvedený fragment kódu po spustení vypíše na obrazovku čísla:

```
3
4
4
4
```

Ako vidieť, výsledná hodnota premenných **a** aj **b** bude po inkrementácii **4**. Výraz **a++** však vráti hodnotu a pred inkrementáciou, čo je **3**, výraz **++b** vráti hodnotu b po inkrementácii a to je už spomínaných **4**. Rovnaký príklad si možno vyskúšať pre operátor dekrementácie **--**.

Operátory **++** a **--** v Jave vykazujú jeden podstatný rozdiel oproti C++: v C++ vracia prefixový variant oboch operátorov l-hodnotu, predstavujúcu inkrementovanú/dekrementovanú premennú, ktorej je tak možné priradiť inú hodnotu. V Jave je výsledkom aplikácie operátorov vždy hodnota, nikdy nie premenná, preto výraz **++a = 3** je v Jave chybný.

Prefixové varianty oboch operátorov sa asociujú sprava doľava, to znamená, že výraz **-++a** sa vyhodnotí ako **-(++a)**. Pri postfixových variantoch o asociácii nemá zmysel hovoriť, dôležité je vedieť, že tieto operátory majú pomerne veľkú prioritu. Ale k tomu sa ešte dostaneme.

4.časť: Operátory (pokračovanie)

Vo štvrtej časti seriálu, poslednej v tomto storočí, pokračujeme opisom operátorov, ktoré jazyk Java poskytuje programátorovi na prácu s údajmi. Dosiaľ sme prebrali základné aritmetické operátory **+**, **-**, *****, **/** a **%** a dva operátory inkrementácie/dekrementácie **++** a **--**.

Relačné operátory
Ďalšou skupinou operátorov sú tzv. relačné operátory. Pojem relácia bude laikovi pravdepodobne evokovať nejaký vzťah. Skutočne, relačné operátory v Jave slúžia na zistenie, či sú dva výrazy v požadovanom vzťahu. Relácie, ktoré tieto operátory rozpoznávajú, sú dobre známe z matematiky: ide o rovnosť a ostrú a neostrú nerovnosť.

Operátory testujúce nerovnosť dvoch výrazov sú štyri: **<**, **<=**, **>**, **>=**. Vzťahy, ktoré reprezentujú, sú po rade „menší“, „menší alebo rovný“, „väčší“ a „väčší alebo rovný“. Znaky operátorov neostrej nerovnosti **<=** a **>=** treba zapisovať v uvedenom poradí, nikdy nie naopak.

Všetky štyri operátory sú binárne. Ich operandy musia byť povinne primitívne číselné typy a výsledok je vždy typu boolean. Z toho vyplýva, že i keď sa asociujú zľava doprava, nemá asociácia v tomto prípade nijaký význam, pretože zápis **a < b < c** povedie k chybe pri preklade (**a < b** je typu **boolean**, ktorý nemožno porovnávať s premennou c, nech by bola akéhokoľvek typu).

V prípade, že porovnávame výrazy typu float alebo double, je výsledkom NaN, ak je aspoň jeden z operandov NaN. Na účely porovnávania prostredníctvom operátorov **<**, **>**, **<=** a **>=** sú čísla „kladná nula“ a „záporná nula“ rovné a výraz **-0.0 < +0.0** vráti false.

Ďalšie dva relačné operátory, **==** a **!=**, zisťujú rovnosť, resp. nerovnosť dvoch výrazov. Oba sú navzájom komplementárne a vždy platí, že **a!=b** má rovnakú hodnotu ako **!(a==b)**. Aj tieto operátory sú binárne a ich výsledok je vždy typu boolean. Čo sa však týka typu ich operandov, máme k dispozícii trochu širšiu oblasť použitia.

Predovšetkým môžu byť operandmi **==** a **!=** primitívne číselné typy. Pri porovnávaní výrazov typu float/double platí, že ak je aspoň jeden z operandov NaN, výsledok **==** je false, výsledok **!=** je true. Kladná a záporná nula sú považované za navzájom rovné.

Druhá možnosť je porovnávať operandy typu **boolean**. Za pozornosť stojí, že v takom prípade je výsledok operátora **!=** totožný s výsledkom operátora **^** (pozri ďalej).

Konečne môžeme pomocou operátorov **==** a **!=** porovnávať výrazy referenčných typov. Treba si rýchlo spomenúť na predchádzajúcu časť seriálu, kde sme uviedli, že premenné referenčných typov obsahujú iba adresu, odkaz na skutočné údaje, uložené niekde inde v pamäti. (Na zopakovanie: referenčnými typmi sú v Jave objekty [inštancie tried] a polia). Po krátkej úvahe dospejeme k záveru, že operátor **==** aplikovaný na dve premenné referenčného typu vráti true len vtedy, ak obe premenné budú ukazovať ***na jeden a ten istý*** objekt či pole. Operátor **!=** v takom prípade vráti, pochopiteľne, false.

Relačné operátory sa používajú najmä v podmienkových výrazoch príkazov **if**, **for** či **while**. Príkazom Javy budeme venovať samostatnú časť seriálu, preto na demonštráciu aplikácie operátorov použijeme len takýto jednoduchý príklad:

```
int a = -5;
int b = 14;
System.out.println("(a <  b) = " + (a <  b));
System.out.println("(a <= b) = " + (a <= b));
System.out.println("(a >  b) = " + (a >  b));
System.out.println("(a >= b) = " + (a >= b));
System.out.println("(a == b) = " + (a == b));
System.out.println("(a != b) = " + (a != b));

boolean c = true;
boolean d = false;
System.out.println("(c == d) = " + (c == d));
System.out.println("(c != d) = " + (c != d));

int pa[] = { 1, 2, 3 };
int pb[] = pa;
int pc[] = { 1, 2, 3 };
System.out.println("(pa == pb) = " + (pa == pb));
System.out.println("(pa == pc) = " + (pa == pc));
```

Výsledkom aplikácie jednotlivých operátorov budú po rade literály **true**, **true**, **false**, **false**, **false**, **true** (prvá časť programu), **false**, **true** (druhá časť programu) a **true**, **false** (tretia časť programu). Ako vidieť, premenné **pa** a **pb** ukazujú na to isté pole, preto sa z pohľadu operátora **==** navzájom „rovnajú“. Premenné **pa** a **pc** sa však „nerovnajú“ napriek tomu, že polia, ktoré reprezentujú, majú rovnaké hodnoty prvkov. Vyskúšanie aplikácie operátorov na iné hodnoty ponechávam na čitateľovi.

Aby bolo vymenovanie relačných operátorov úplné, nesmieme vynechať operátor **instanceof**, ktorý pomôže pri zisťovaní skutočného typu výrazu. To však platí len pre referenčné typy, výrazy primitívnych typov takto identifikovať nemožno. Operátor instanceof je binárny, ľavým operandom musí byť výraz referenčného typu, pravým operandom je literál predstavujúci vyšetrovaný typ. Ak je typ ľavého operandu zhodný s pravým operandom, výsledkom je hodnota **true**, inak je výsledkom **false**.

Typickým príkladom použitia môže byť obslužná rutina nejakej udalosti grafického používateľského rozhrania, ktorá zisťuje, ktorý prvok GUI má udalosť „na svedo-

mí“:

```
if (sender instanceof Button)
{
   // ...
}
```

Predpokladáme, že premenná sender obsahuje odkaz na objekt predstavujúci prvok GUI, ktorý udalosť spôsobil. Trieda Button by mohla reprezentovať prvky typu „tlačidlo“, potom bude uvedený príkaz if testovať, či je zdrojom udalosti niektoré tlačidlo.

Bitové a logické operátory

Podobne ako C++ možno nájsť v Jave niekoľko operátorov, ktoré pracujú s bitovou reprezentáciou čísel. Jednotlivé bity celočíselných premenných či hodnôt môžeme považovať za logické premenné a aplikovať na ne základné funkcie boolovskej algebry: logický súčin (AND), logický súčet (OR), exkluzívny logický súčet (takisto nonekvivalencia – XOR) a negáciu (NOT). Pre tieto štyri logické funkcie sú v Jave určené štyri operátory: **&**, **|**, **^** a **~**. V tabuľke nájde čitateľ funkčné hodnoty logických funkcií pre jednotlivé kombinácie bitových hodnôt.

		AND	OR	XOR	NOT
a	**b**	**a & b**	**a | b**	**a ^ b**	**~a**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Logické funkcie

Operátory majú prívlastok bitové z toho dôvodu, že realizujú logické funkcie bit po bite, vždy medzi zodpovedajúcimi bitmi svojich operandov (výnimkou je, samozrejme, operátor negácie, ktorý je unárny a má len jediný operand).

Operátor bitového logického súčinu & slúži na výpočet logického súčinu dvoch celočíselných hodnôt. Ide teda o binárny operátor, ktorý sa asociuje podobne ako aditívne či multiplikatívne operátory zľava doprava. Príklad logického súčinu dvoch premenných typu **int**:

```
int a = 0x3CA8;
int b = 0xCAFE;
int c = a & b;
System.out.println("a & b = " + c);
```

Výstupom tohto programu bude číslo **2216**, hexadecimálne **0x08A8**, ktoré je skutočne výsledkom logického súčinu hodnôt **0x3CA8** a **0xCAFE**, o čom sa môžeme ľahko presvedčiť prepísaním hodnôt premenných **a** a **b** a výsledku **c** do dvojkovej sústavy.

Argumentmi operátora **&** môžu byť okrem celočíselných hodnôt či premenných aj premenné a hodnoty typu **boolean**. Tie, ako vieme, sú široké len jeden bit. Logický súčin sa realizuje nad týmto jediným bitom a výsledky sú v zhode s tabuľkou 1, len s tým rozdielom, že namiesto číselných hodnôt **0** a **1** uvažujeme pri logických premenných literály **false** a **true**. O tom ostatne svedčí nasledujúci úsek programu:

```
System.out.println("true & false = " + (true & false));
```

ktorého výstupom bude, pochopiteľne, literál **false**.

Operátor bitového logického súčtu **|** realizuje výpočet logického súčtu jednotlivých bitov svojich dvoch operandov. Opäť ide o binárny operátor s asociáciou zľava doprava. Použitie ukážeme na modifikácii predchádzajúceho príkladu:

```
int a = 0x3CA8;
int b = 0xCAFE;
int c = a | b;
System.out.println("a | b = " + c);
```

Tentoraz je výstupom hodnota **65278**, hexadecimálne **0xFEFE**. O správnosti výsledku sa opätovne možno presvedčiť prepisom hodnôt do dvojkovej sústavy.

Aj operátor **|** dokáže spracovať logické premenné typu boolean. Výstupom nasledujúceho riadka:

```
System.out.println("true | false = " + (true | false));
```

je teda literál true.

Operátor *bitového exkluzívneho logického súčtu* **^** nás už iste ničím neprekvapí. Slúži na výpočet funkcie XOR, ktorá na rozdiel od funkcie OR (tzv. inkluzívneho logického súčtu) vylučuje súčasnú platnosť oboch svojich argumentov. (To je ostatne jediný bod, v ktorom sa XOR líši od OR.) Aj tento operátor je binárny a asociuje sa zľava doprava. Obligátny príklad:

```
int a = 0x3CA8;
int b = 0xCAFE;
int c = a ^ b;
System.out.println("a ^ b = " + c);
```

Program poskytne výsledok **63062**, hexadecimálne **0xF656**.

Ani operátor **^** nám neodoprie možnosť aplikovať ho na hodnoty typu **boolean**:

```
System.out.println("true ^ true = " + (true ^ true));
```

Ako uvidíme po spustení, exkluzívnym logickým súčtom dvoch hodnôt **true** je presne podľa predpokladov hodnota **false**.

Operátor *bitovej negácie* **~** našťastie trochu narúša doterajší stereotyp. Ako vyplýva z povahy negácie, tento operátor bude unárny a asociovať sa bude podobne ako iné unárne operátory sprava doľava. Aplikovať ho však môžeme ***výhradne*** na premenné celočíselného typu:

```
int a = 0x3CA8;
int b = ~a;
System.out.println("~a = " + b);
```

Výstupom bude bitová negácia hodnoty **0x3CA8**, čo je dekadicky **-15529** (typ int je znamienkový, spomínate si?) a hexadecimálne **0xFFFFC357** (pretože typ int má šírku 32 bitov).

Veľký pozor si treba dať pri pokuse o negáciu premenných typu **boolean**. V takom prípade totiž nemožno použiť operátor bitovej negácie **~**. Našťastie Java je v tomto smere ohľaduplná a ako náhradu ponúka operátor logickej negácie !. Pre tento operátor inak platí všetko, čo sme uviedli o operátore **~**. Ešte krátky príklad:

```
System.out.println("!true = " + !true);
```

Ako sa môžeme presvedčiť, negáciou hodnoty **true** je (prekvapujúco :) hodnota **false**.

V Jave podobne ako v C++ existujú okrem bitových operátorov logického súčinu **&** a logického súčtu **|** ešte iné dva logické operátory **&&** a **||**, realizujúce na pohľad rovnaké logické funkcie. Oba operátory sú binárne a asociujú sa zľava doprava, ich operandy však ***musia byť*** typu boolean. Okrem toho sa operátory **&&** a **||** líšia od svojich jednoznakových náprotivkov v pomerne zásadnej veci: spomeňme si, že operandy operátora sa v Jave vyhodnocujú vždy zľava doprava. Ak sa však ľavý operand operátora **&&** vyhodnotí ako **false**, výsledok už nemôže byť iný ako **false**, preto sa pravý operand ani nevyhodnocuje. Podobne operátor **||**: ak má jeho ľavý operand hodnotu **true**, výsledok bude za každých okolností **true** a pravý operand sa nevyhodnotí.

Na demonštráciu uvedených vlastností operátorov **&&** a **||** slúži nasledujúci príklad:

```
int i = 10;
boolean never = false;
boolean r1 = (i < 0) && (never = true);
boolean r2 = (i > 0) || (never = true);
System.out.println("r1 = " + r1 + ", r2 = " + r2);
System.out.println("never = " + never);
```

Výstup programu:

```
r1 = false, r2 = true
never = false
```

nám potvrdí, že pri vyhodnocovaní výrazov priraďujúcich hodnoty premenným **r1** a **r2** sú výsledky známe už po vyhodnotení príslušných podmienok a k priradeniu **true** do premennej **never** nikdy nedôjde.

Čitateľ ovládajúci C++ si azda v súvislosti s operátorom **&&** všimne zásadný rozdiel medzi C++ a Javou. V C++ môžu byť operandy **&&** ľubovoľného číselného typu. Zoberme si teda napríklad čísla **5** a **10**. Výraz **5 && 10** je vyhodnotený ako pravdivý (jeho hodnota je **1**), zatiaľ čo výraz **5 & 10** je vyhodnotený ako **0**, pretože oba operandy majú spodné štyri bity „na preskáčku“ (**0101** a **1010**). V Jave operandy **&&** môžu byť iba typu boolean a výsledok aplikácie operátora **&&** bude rovnaký ako výsledok aplikácie **&**, rozdiel je len v prípadnom skrátenom vyhodnocovaní. Zatiaľ čo teda v C++ je potrebné podmienky reťaziť operátorom **&&** (ako napr. **a >= 100 && a < 200**), v Jave môžeme použiť aj operátor **&**. Pozor treba dať iba na to, aby výsledok aplikácie **&** bol typu **boolean**, pretože podmienkové výrazy v príkazoch **if**, **while** a pod. musia byť výhradne typu **boolean**.

Praktická aplikácia bitových operátorov

Bitové operátory **&**, **|** a **^** možno s úspechom použiť pri manipulácii s jednotlivými bitmi celočíselných premenných. V praxi sa obyčajne vyskytne potreba vybrané bity určitej premennej nastaviť, vynulovať alebo preklopiť do opačného stavu. Ostatné bity by sme radi ponechali v pôvodnom stave. Riešenie tejto úlohy je pomerne jednoduché, stačí dobre ovládať vlastnosti logických funkcií.

Pre *nastavenie vybraných bitov* s úspechom využijeme funkciu logického súčtu OR. Ako vyplýva z tabuľky jej funkčných hodnôt (tab. 1), logický súčet s jednotkovým bitom je vždy jednotka, logický súčet s nulovým bitom pôvodný bit nemení. Ako druhý operand operátora **|** teda použijeme číslo, v ktorom sú jednotkové tie bity, ktoré chceme nastaviť, a nulové všetky ostatné. Príklad:

```
byte b = 0x29;
System.out.println("b | 0x07 = " + (b | 0x07));
```

Chceme nastaviť spodné tri bity, použijeme teda číslo **0x07**. Výsledok bude podľa očakávania číslo **0x2F** (resp. 47 dekadicky).

Na *nulovanie vybraných bitov* sa hodí funkcia logického súčinu AND. Logický súčin ľubovoľnej hodnoty s nulou je vždy nula, logický súčin s jednotkou pôvodný bit zachová. Tentoraz teda použijeme ako druhý operand operátora **&** číslo, v ktorom sú nulové tie bity, ktoré potrebujeme vynulovať; ostatné bity zadáme jednotkové. Príklad:

```
byte b = 0x29;
System.out.println("b & 0xF8 = " + (b | 0xF8));
```

Tentoraz chceme vynulovať spodné tri bity, vhodným operandom bude číslo **0xF8**. Výsledkom je hodnota **0x28** alebo dekadicky 40.

Konečne *negáciu vybraných bitov* realizujeme pomocou funkcie exkluzívneho logického súčtu XOR. Tá má tú vlastnosť, že ak XORujeme ľubovoľný bit s jednotkou, výsledkom je presne opačná hodnota. Nula pôvodný bit nemení. Operand operátora **^** vytvoríme rovnakým spôsobom ako pri nastavovaní bitov. Príklad:

```
byte b = 0x29;
System.out.println("b ^ 0x07 = " + (b ^ 0x07));
```

Opätovne použijeme číslo **0x07**, výsledkom negácie spodných troch bitov hodnoty **0x29** bude číslo **0x2E,** dekadicky 46.

Číslo, ktoré používame ako druhý operand operátorov **&**, **|** a **^**, sa nazýva maska. Je vhodné uložiť si masku do samostatnej premennej (prípadne konštantnej premennej – uvidíme neskôr, ako ju deklarujeme). Potom ju môžeme používať veľmi elegantným spôsobom:

```
final byte MASK = 0x07;
b = b | MASK;   // nastavenie
b = b & ~MASK;  // nulovanie
b = b ^ MASK;   // negácia
```

Všimnite si, ako dosiahneme príslušnú masku na nulovanie bitov – jednoducho masku uloženú v premennej MASK negujeme pomocou operátora **~**. Výsledkom negácie **0x07** je, samozrejme, hodnota **0xF8**.

Nakoniec treba podotknúť, že uvedené príklady pracovali s osembitovými hodnotami typu byte. Pri použití iných typov (**short**, **int**, **long**) treba rátať so širším rozsahom, ale princíp zostáva rovnaký.

Operátory bitového posunu

Pri operátoroch, ktoré berú do úvahy bitovú reprezentáciu čísel, ešte chvíľu zostaneme. V praxi občas nastane situácia, že potrebujeme zmeniť hodnotu nejakej premennej takým spôsobom, aby jej jednotlivé bity zmenili svoju polohu o jednu či viac pozícií doľava alebo doprava.

Operátory bitového posunu sú tri: operátor posunu vľavo **<<**, operátor znamienkového posunu vpravo **>>** a operátor neznamienkového posunu vpravo **>>>**. Všetky tri operátory sú binárne a asociujú sa zľava doprava. Ľavým operandom je posúvaná hodnota, pravým operandom je počet pozícií, o ktorý sa bity posunú. Typy oboch operandov musia byť celočíselné.

Pri posune vľavo bity, ktoré z čísla „vyliezajú“ na ľavom konci, miznú do nenávratna, sprava sa prisúvajú nuly. Pri posune vpravo so znamienkom (operátor **>>**) končia svoju existenciu bity vychádzajúce na pravom konci, zľava sa prisúva bit zhodný s najvyšším, t. j. znamienkovým bitom pôvodnej hodnoty (iný pohľad: najvyšší bit zostáva na svojom mieste a do posunu posiela svoju kópiu). Posun vpravo bez znamienka (operátor **>>>**) sa líši od znamienkového posunu iba tým, že zľava sa prisúvajú vždy nuly.

Ukážeme si aplikáciu všetkých troch operátorov na praktickom príklade:

```
int b = 0xCAFEBABE;
System.out.println("b <<  3 = " + (b <<  3));
System.out.println("b >>  3 = " + (b >>  3));
System.out.println("b >>> 3 = " + (b >>> 3));
```

Po spustení príkladu dostaneme tri hodnoty, ktoré po prevode do šestnástkovej sústavy budú po rade **0x57F5D5F0**, **0xF95FD757** a **0x195FD757**. Prepis týchto hodnôt v binárnom tvare nás presvedčí o tom, že sa bity pôvodnej hodnoty **0xCAFEBABE** skutočne posunuli o tri pozície doľava či doprava. Ako tiež vidíme, operátor **>>** zachoval záporné znamienko pôvodného čísla, zatiaľ čo operátor **>>>** vrátil kladné číslo (s nulovým najvyšším bitom).

Pozorný čitateľ si isto uvedomil, že zmena pozície jednotlivých bitov má na svedomí zmenu príslušného exponentu pri prevode celého čísla do a z dvojkovej sústavy. Skutočne, posun vľavo o n bitov je totožný s násobením čísla hodnotou **2n**. Naproti tomu znamienkový posun vpravo o n bitov zodpovedá celočíselnému deleniu hodnotou **2n**. Neznamienkový posun vpravo je ekvivalentný deleniu mocninou dvojky len pre kladné čísla.

V súvislosti s operátormi bitového posunu treba spomenúť fakt, že posúvané čísla sú vnútorne konvertované na jeden z typov **int** alebo **long** (podľa rozsahu), takže výsledkom posunu premennej typu **byte**, obsahujúcej hodnotu **0xFF**, vľavo o štyri bity nebude očakávaná hodnota **0xF0**, ale hodnota **0x00000FF0** typu **int** (v prípade, že potrebujeme výsledok šírky typu **byte**, musíme ho orezať pomocou operátora **&**). S ohľadom na šírku typov **int** a **long** sa z pravého operandu, vyjadrujúceho „vzdialenosť“ posunu, uvažuje len spodných päť, resp. šesť bitov.

Podmienkový operátor ?:

Ďalší javovský operátor **?:** bude dobre známy čitateľovi znalému C++. Na rozdiel od doteraz prebraných operátorov je operátor **?:** ternárny, pracuje s tromi operandmi. Asociuje sa sprava doľava, výraz **a?b:c?d:e** sa preto vyhodnotí ako **a?b:(c?d:e)**.

Sémantika operátora **?:** je nasledujúca: vyhodnotí sa prvý operand, ktorý musí byť typu **boolean**. Týmto prvým operandom je obyčajne nejaká podmienka. Ak je hodnota prvého operandu **true**, vyhodnotí a vráti sa druhý operand, ak **false**, vyhodnotí a vráti sa tretí operand. Druhý a tretí operand musia byť rovnakého typu. Je dôležité vedieť, že v závislosti od splnenej či nesplnenej podmienky sa vyhodnotí buď druhý operand, alebo tretí operand, nikdy nie oba naraz.

Príklad použitia – chceme zistiť najväčšiu hodnotu z troch čísel **a, b** a **c**:

```
int a = 123, b = 17, c = -229;
int max = a>b ? (a>c ? a : c) : (b>c ? b : c);
System.out.println("max(a, b, c) = " + max);
```

V príklade sa najprv porovnajú hodnoty **a** a **b**. Ak a je väčšie ako **b**, **b** už nemôže byť maximom. Porovnáme preto **a** s **c** a väčšie z oboch vrátime ako maximum. V prípade, že **a** nie je väčšie ako **b**, musí byť maximom jedna z hodnôt **b** alebo **c**. Ktorá, to zistíme treťou aplikáciou operátora **?:**. V našom prípade bude výsledkom hodnota **123**, uložená v premennej **a**. Na zamyslenie dávam do pozornosti, prečo určenie maxima bude fungovať aj v prípade, že niektoré dve hodnoty (či priamo všetky tri) budú zhodné.

Pre ilustráciu vyhodnotenia buď druhého, alebo tretieho operandu **?:** je tu ešte tento krátky príklad:

```
int i = 2;
System.out.println(i > 0 ? (i = 3) : (i = -5));
System.out.println(i < 0 ? (i = -2) : (i = 1));
System.out.println(i);
```

V druhom riadku je podmienka **i > 0** splnená, vyhodnotí sa výraz **i = 3**, takže premenná i bude obsahovať hodnotu **3**. Na obrazovku sa vypíše hodnota výrazu **i = 3**, čo je, samozrejme, **3**. V treťom riadku podmienka splnená nebude, vyhodnotí sa výraz **i = 1**, ktorého hodnota **1** sa vypíše na obrazovku. Premenná **i** bude mať na konci programu hodnotu **1**, o čom nás presvedčí štvrtý riadok. Ani v jednom prípade sa nevyhodnotil výraz priraďujúci premennej i zápornú hodnotu.

Priraďovacie operátory

Poslednými operátormi, o ktorých si povieme, sú operátory priradenia. Skutočne je namieste použitie množného čísla, pretože okrem klasického priraďovacieho operátora **=**, o ktorom sme dosiaľ akosi mlčky predpokladali, že jeho významom je zrejmý, a podľa toho sme ho používali v príkladoch, poskytuje Java podobne ako C++ jedenásť ďalších, zložených priraďovacích operátorov **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=** a **>>>=**.

Jednoduchý priraďovací operátor **=** je binárny. Jeho ľavým operandom musí byť výraz, ktorý sa vyhodnotí ako premenná (t. j. to, čo v C++ nazývame l-hodnota), pravý operand predstavuje priraďovanú hodnotu. Ak nie je možné pravý operand konvertovať na typ ľavého operandu, dôjde pri preklade, prípadne pri behu programu k chybe. Operátor **=** sa asociuje sprava doľava, preto výraz **a = b = c** sa grupuje ako **a = (b = c)**. Hodnotou priraďovacieho výrazu je priraďovaná hodnota, takže do premennej a sa priradí tá istá hodnota ako do premennej b, menovite obsah premennej **c**. Toto viacnásobné priradenie je ostatne známe z C++. Pozor, hodnota priraďovacieho výrazu už nie je premennou, zápis **(a = b) = c** je preto chybný.

Zložené priraďovacie operátory majú bez výnimky tvar **op=**, kde **op** je vždy niektorý z doteraz prebraných binárnych operátorov. Sémantika zložených operátorov priradenia je nasledujúca: výraz **E1 op= E2** je ekvivalentný výrazu **E1 = E1 op E2**. Výsledok **E1 op E2** sa pred priradením späť do **E1** konvertuje na typ **E1**. Zložené operátory sú takisto binárne a asociujú sa rovnako ako jednoduchý operátor **=**, ich operandy však musia byť primitívnych typov. Jedinou výnimkou je operátor **+=**, ktorého ľavý operand môže byť typu **String**, pravý operand v takom prípade môže mať ľubovoľný typ.

Na použitie operátora **=** azda netreba uvádzať príklad, skôr príde vhod nasledujúca ukážka:

```
int x = 2;
x += 3.14;
System.out.println("x = " + x);
```

Ako sa môžeme presvedčiť spustením tohto programu, premenná **x** bude mať po priradení pomocou operátora **+=** hodnotu **5**, pretože výsledok súčtu **x + 3.14**, čo je **5.14**, sa prevedie naspäť na typ **int**.

Pri vyhodnocovaní zložených priraďovacích operátorov **op=** sa hodnota ľavého operandu pred vyhodnotením pravého operandu zapamätá, takže výsledok bude naozaj súčtom pôvodnej hodnoty ľavej strany s výsledkom pravej strany priradenia. O tom nás presvedčí tento krátky príklad:

```
int d = 11;
d *= (d = 3) + (d << 2);
System.out.println("d = " + d);
```

Po priradení bude v premenej **d** uložená hodnota **165**. Prečo? Nuž, pôvodná hodnota **11** sa zapamätá. Následné vyhodnotenie pravej strany uloží do **d** hodnotu **3**, ktorú sčíta s hodnotou **d << 2**, čo je **12**. Výsledok, číslo **15**, sa vynásobí s uloženou hodnotou **11**, čím dostaneme konečných **165**.

5.časť: Príkazy

Vitajte pri ďalšom pokračovaní nášho rozprávania o Jave. V piatej časti sa dozviete, ako pomocou príkazov jazyka vysvetlíme počítaču, čo má presne robiť pri vykonávaní nášho programu. Skôr však, než začneme, treba uzavrieť ešte tému operátory. Podľa môjho pôvodného úmyslu k tomu malo dôjsť ešte v predošlej časti, ale vzhľadom na určité problémy pri výrobe časopisu sme po dohode s redakciou štvrtú časť Javy pod lupou o niečo skrátili. Chýbajúci záver teda dostávate do rúk teraz.

Pretypovanie

Poslednou témou, ktorá spadá do časti venovanej operátorom, je špeciálny typ výrazu, tzv. *pretypovávací výraz*. Jeho syntax je jednoduchá – pred výraz, ktorému chceme explicitne zmeniť typ, napíšeme tento výsledný typ do okrúhlych zátvoriek. V C++, vlastne ešte v C blahej pamäti, sa tomuto zápisu hovorí aj operátor pretypovania. V Jave sa dáva prednosť pojmu pretypovávací výraz (cast expression). V praxi sa obyčajne tento výraz používa pri explicitnom pretypovaní číselných výrazov či pri explicitnej zmene typu referenčnej premennej, ktorá ukazuje na nejaký objekt (to ide s ohľadom na dedičnosť). Príklad:

```
byte b = 0x3C;
byte c = (byte)(b << 4);
```

Ak chceme uložiť výsledok posunu premennej **b** typu **byte** o štyri bity vľavo naspäť do inej premennej **c** typu **byte**, musíme použiť explicitné pretypovanie, inak prekladač ohlási chybu „Possible loss of precision“ a program nepreloží.

Výsledkom pretypovania nikdy nie je premenná, zápis z C++:

```
(double)x = 1.23;
```

je teda chybou. V Jave ostatne takýto zápis pravdepodobne nikdy nebudeme potrebovať.

Pretypovávať možno medzi sebou len niektoré typy, napríklad pokus pretypovať primitívny typ na referenčný sa skončí chybou už počas prekladu. Niektoré pretypovania uspejú za každých okolností (napr. pretypovanie potomka na predka v objektovej hierarchii tried), správnosť iných treba overiť až za behu programu.

Operátor	Význam	Asociativita
+ +	inkrementácia (postfix)	–
– –	dekrementácia (postfix)	
+ +	inkrementácia (prefix)	P ® Ľ
– –	dekrementácia (prefix)	
+	unárne plus	
–	unárne mínus	
~	bitová negácia	
!	logická negácia	
(typ)	pretypovanie	
*	násobenie	Ľ ® P
/	delenie	
%	zvyšok po delení	
+	sčítanie	Ľ ® P
–	odčítanie	
< <	bitový posun vľavo	Ľ ® P
> >	bitový posun vpravo so znam.	
> > >	bitový posun vpravo bez znam.	
<	menší	Ľ ® P
< =	menší alebo rovný	
>	väčší	
> =	väčší alebo rovný	
instanceof	objekt daného typu	
= =	rovný	Ľ ® P
! =	nerovný	
&	bitový logický súčin	Ľ ® P
I	bitový logický súčet	
^	bitový exkluzívny logický súčet	
& &	logický súčin	Ľ ® P
I I	logický súčet	
? :	podmienené vyhodnotenie	P ® Ľ
=	priradenie	P ® Ľ
op =	zložené priradenie	

Priorita operátorov

Aby bolo rozprávanie o operátoroch úplné, treba ešte uviesť poradie, v ktorom sa budú vyhodnocovať operátory vo výraze, ktorý neobsahuje zátvorky. Jednotlivé operátory totiž majú rôznu prioritu a klasickým spôsobom zľava doprava sa budú vyhodnocovať len operátory s rovnakou prioritou. Zoznam operátorov zoradený podľa ich klesajúcich priorít je v tabuľke. Najvyššiu prioritu má, samozrejme, explicitné uzátvorkovanie príslušných častí výrazu. Zátvorky sa odporúča použiť vždy, keď by mohlo dôjsť k omylu či zmätku pri riešení poradia vyhodnocovania (samozrejme, omylu zo strany programátora, počítač stanovené poradie vždy dodrží).

Krátka rekapitulácia

Zhrňme si teraz v krátkosti, čo sme dosiaľ prebrali. Vieme, že Java stavia na princípoch objektovo orientovaného programovania. Všetok kód v Jave je organizovaný do tried, čo sú v princípe šablóny na tvorbu objektov. Každý existujúci objekt v programe je inštanciou nejakej triedy. Objekty medzi sebou komunikujú pomocou zasielania správ, čo sa zo syntaktického hľadiska realizuje volaním metód. Metódy sú ekvivalentmi céplusplusovských členských funkcií tried. Okrem metód, ktoré opisujú správanie objektov, nájdeme v triedach členské premenné, v ktorých je uložený stav objektov. Na úrovni zdrojového kódu je trieda syntaktickou konštrukciou, ktorá obsahuje opis členských premenných a metód. Tomuto opisu inak hovoríme deklarácia. Deklarovať vo všeobecnosti treba všetky premenné, ktoré jednotlivé objekty budú používať, čo okrem členských premenných predpokladá aj lokálne premenné, používané pri vykonávaní kódu metód. Pri deklarácii premennej je nevyhnutné uviesť jej typ a názov a prípadne aj inicializačnú hodnotu. Deklarácia metódy oznamuje prekladaču okrem názvu metódy počet a typ jej argumentov a typ návratovej hodnoty. Metódy, pochopiteľne, nemusia mať argumenty ani vracať nijakú konkrétnu hodnotu. Čo sa týka typov údajov, s ktorými program pracuje, možno ich rozdeliť na primitívne a referenčné. Prvá množina typov zahŕňa číselné typy (celočíselné i s desatinnou čiarkou) a logický typ. Druhá množina predstavuje v zásade štruktúrované typy, menovite objekty (inštancie tried) a polia. Špeciálnym referenčným typom sú rozhrania. Spôsob spracovania údajov v programe sa vyjadruje spôsobom takmer matematickým, pomocou výrazov. Každý výraz je tvorený kombináciou operátorov a ich operandov. Operátory sú symboly vyjadrujúce požadovaný druh operácie, ktorú treba vykonať na operandoch. Výsledkom vyhodnotenia výrazu je vždy nejaká hodnota. Jednotlivé operátory možno roztriediť do skupín podľa priority vyhodnocovania.

Intermezzo: praktický príklad

Dosť bolo teórie, ukážeme si teraz krátky, ale praktický príklad, ktorým bude program na výpočet koreňov kvadratickej rovnice. Aby bol program reálne použiteľný, budeme chcieť vypočítať korene aj v prípade, že riešenie bude ležať mimo množiny reálnych čísel, v komplexnej oblasti. Bohužiaľ, dosiaľ sme nehovorili o podmienenom vykonávaní príkazov, preto prosím čitateľov o prepáčenie, že v programe nájdu zatiaľ neznámy príkaz **if**. Jeho použitie je však pomerne zrozumiteľné a okrem toho bližšie informácie sa nachádzajú o niekoľko odsekov ďalej.

Takže tu je program (výnimočne kompletný):

```
import java.io.*;

class Quadratic
{
   public static void main(String[] args) throws
IOException
 {
   BufferedReader in =
                          new      BufferedReader(new
InputStreamReader(System.in));
   double a, b, c;

   System.out.print("a = ");
   a = Double.valueOf(in.readLine()).doubleValue();
   System.out.print("b = ");
   b = Double.valueOf(in.readLine()).doubleValue();
   System.out.print("c = ");
   c = Double.valueOf(in.readLine()).doubleValue();

   double D = discr(a, b, c);
   if (D > 0)
   {
    System.out.println("\nTwo roots:");
    System.out.println("  x1 = " + ((-b + Math.sqrt(D))
/ 2 / a));
    System.out.println("  x2 = " + ((-b - Math.sqrt(D)) /
2 / a));
   }
   else if (D == 0)
   {
    System.out.println("\nOne double root:");
    System.out.println("  x = " + (-b / 2 / a));
   }
   else
   {
    System.out.println("\nTwo complex roots:");
    System.out.println("  x1 = " + (-b / 2 / a) +
            " + " + (Math.sqrt(-D) / 2 / a) + "i");
    System.out.println("  x2 = " + (-b / 2 / a) +
            " - " + (Math.sqrt(-D) / 2 / a) + "i");
   }
 }

 private static double discr(double a, double b, double c)
 {
  return b * b - 4 * a * c;
 }
}
```

Zdrojový text treba, samozrejme, prepísať do súboru **Quadratic.java**. Ak program preložíte a spustíte, vypýta si od vás tri koeficienty kvadratickej rovnice **a, b** a **c**. S ohľadom na jednoduchosť sa nikde nekontroluje, či zadáte kvadratický koeficient a nenulový (program v tom prípade poskytne kuriózne výsledky – presvedčte sa sami). Takisto nie je nijako ošetrené zadanie nečíselných hodnôt. Spôsob zadávania vstupu vyzerá veľmi komplikovane, ale zatiaľ ho čitateľ bude musieť akceptovať bez vysvetlenia.

Na výpočet diskriminantu je v triede Quadratic zavedená súkromná metóda **discr()**. Spočítaná hodnota diskriminantu sa testuje voči nule. V prípade, že je diskriminant kladný, program vypočíta a vypíše dva reálne korene, nulový diskriminant znamená jeden dvojnásobný koreň a konečne záporná hodnota diskriminantu spôsobí výpočet dvoch komplexných koreňov.

Verím, že sa na mňa nebude ani jeden čitateľ hnevať za použitie anglických textov v programe. Angličtina, ako lingua franca počítačového sveta (internet nevynímajúc), dnes totiž patrí medzi nevyhnutné, i keď rozhodne nie postačujúce znalosti úspešného programátora.

Hor' sa na príkazy

Ako sme už niekoľkokrát uviedli, bežiaci program v Jave si možno predstaviť ako množinu vytvorených objektov, ktoré medzi sebou komunikujú prostredníctvom správ. Na syntaktickej úrovni je poslanie správy realizované ako zavolanie konkrétnej metódy konkrétneho objektu. Ako však virtuálny počítač vie, čo má robiť pri vyvolaní danej metódy? Nuž, povie mu to obsah tela metódy, ktoré, ako sme videli, je uzavreté do zložených zátvoriek ({}). Telo každej metódy je tvorené postupnosťou príkazov.

Ak sú teda stavebnými jednotkami programu objekty so svojimi atribútmi a metódami, možno povedať, že stavebnými jednotkami jednotlivých metód sú príkazy. Každý príkaz opisuje jeden malý krok algoritmu, ktorý daná metóda realizuje. V uvedenom príklade sme mali metódu **discr()**, ktorá obsahovala jediný príkaz na výpočet hodnoty diskriminantu. Mohli sme, pravda, tento výpočet rozdeliť do viacerých krokov, a teda do viacerých príkazov, ale bolo by to zbytočné.

Každá metóda obsahuje žiaden, jeden či viacero príkazov, ktoré sú navzájom oddelené bodkočiarkou. Áno, metóda môže obsahovať aj nulový počet príkazov – takáto prázdna metóda síce nerobí nič užitočné, ale môže slúžiť ako akýsi zástupca (angl. placeholder, ktorý doslova „drží miesto“) do doby, kým potrebné príkazy do jej tela doplníme (nič neobyčajné pri vývoji väčších programov).

Deklaračné a výrazové príkazy

Prvé dva typy príkazov už dobre poznáte, ani o tom neviete. Každá deklarácia lokálnej premennej je príkazom. V tele metódy, samozrejme, nemožno deklarovať nič iné ako lokálne premenné (a takisto lokálne triedy, ale o tom ešte budeme hovoriť). Zoberme si príklad:

```
double a, b, c;
```

Tento zápis je príkazom, ktorého účinok je zrejmý: prekladač odteraz až do skončenia príslušného rozsahu platnosti bude identifikátory **a**, **b** a **c** považovať za premenné typu double a podľa toho bude s nimi aj narábať.

Iným typom príkazu je výrazový príkaz. Takmer každý výraz, ktorý je správne zapísaný podľa syntaktických pravidiel Javy, môžeme použiť ako samostatný príkaz. Slovko takmer je namieste – Java totiž na rozdiel od C++ povoľuje ako príkazy použiť len výrazy, ktoré spôsobujú nejaký trvalý efekt. Medzi ne patrí volanie metódy (možno zavolať aj metódu, ktorá „nič nemení“, ako napr. naša metóda **discr()**, ktorá len vráti vypočítanú hodnotu), priraďovací výraz či výrazy s operátormi **++** a **–**. Pri pokuse použiť ako príkaz výraz, ktorý iba vracia hodnotu, prekladač ohlási chybu. Tu je príklad výrazového príkazu, ktorý má účinok – zmení obsah premennej **x**:

```
x *= (x - 1);
```

A pre úplnosť je tu aj chybný výrazový príkaz, ktorý „nič nerobí“:

```
x * (x - 1);  // CHYBA!
```

Výsledná hodnota výrazu použitého vo výrazovom príkaze sa ignoruje.

Podmienený príkaz if

S týmto príkazom sme sa už stretli v príklade o niekoľko odsekov vyššie. Príkaz **if** slúži na vyjadrenie podmieneného vykonávania určitej časti programu v závislosti od splnenia či nesplnenia zadanej podmienky. Jeho syntax je nasledujúca:

```
if ( podmienka )
   príkaz
```

Príkaz **if** má ešte jeden možný zápis:

```
if ( podmienka )
   príkaz1
else
   príkaz2
```

V prvom, kratšom tvare sa príkaz vykoná vtedy a len vtedy, keď výraz podmienka bude mať hodnotu true. Na rozdiel od C++ musí byť podmienkový výraz typu **boolean**, inak dôjde k chybe pri preklade. V prípade, že podmienka nie je splnená, teda výraz má hodnotu **false**, príkaz sa preskočí. Druhý zápis umožňuje vybrať z dvojice príkazov podľa toho, či je podmienka pravdivá alebo nie. Ak bude výsledkom výrazu podmienka hodnota **true**, vykoná sa príkaz1, ak sa podmienka vyhodnotí na **false**, vykoná sa príkaz2.

Bolo by dosť nešikovné, keby sme v rámci príkazu **if** mohli skutočne použiť iba jeden príkaz, vykonávaný v závislosti od splnenia či nesplnenia podmienky. Preto je v Jave podobne ako v C++ možné na mieste takmer každého príkazu použiť tzv. zložený príkaz, resp. blok príkazov. Takýto blok vytvoríme jednoducho: uzavrieme požadované príkazy do zložených zátvoriek. Za zloženým príkazom na rozdiel od jednoduchého nepíšeme nikdy bodkočiarku.

I keď použitie príkazu if je zrejmé z programu na riešenie kvadratickej rovnice, ukážeme si ešte nejaký príklad:

```
if (a == 0)
  System.out.println("a is zero");
else
  System.out.println("a is non-zero");
```

Takýto úsek kódu vypíše, či obsah premennej a je nulový alebo nenulový. Príslušné vetvy príkazu if obsahujú len po jednom príkaze, preto sú zložené zátvorky zbytočné. Ak však budeme mať nasledujúci príklad:

```
if (a != 0)
{
  b = 5 / a;
  System.out.println("5 / a = " + b);
}
else
  System.out.println("cannot divide");
```

zložené zátvorky okolo dvoch príkazov v „true“ vetve príkazu **if** sú nevyhnutné.

V súvislosti s príkazom **if** treba spomenúť jeden z najznámejších zádrhov v programovacích jazykoch. Obsahom kladnej vetvy príkazu **if** môže byť opäť príkaz **if**. Ako však vyriešiť nasledujúcu situáciu?

```
if (x != 0)
  if (y != 0)
    z = 1 / (x * y);
  else
    z = 1 / x;
```

Patrí časť else prvému či druhému príkazu **if**? V tomto príklade odsadenie kódu nasvedčuje druhej možnosti. Skutočne, prakticky vo všetkých programovacích jazykoch, ktorých sa tento problém týka, existuje pravidlo, podľa ktorého časť else prislúcha vždy najbližšiemu neuzavretému príkazu **if**, v našom prípade druhému, vnorenému. Pozor teda, ak chceme mať časť else len pri vonkajšom **if**, musíme použiť zložené zátvorky:

```
if (x != 0)
{
  if (y != 0)
    z = 1 / (x * y);
}
else
  z = 1;
```

Mimochodom, v ukážkovom programe na riešenie kvadratickej rovnice je použité iné združovanie príkazov **if**, ktoré nie sú vnorené jeden do druhého, ale sú takpovediac zreťazené – ak nie je splnená jedna podmienka, testuje sa druhá, prípadne tretia a ďalšie. Ak by sme chceli dodržiavať pravidlá pre „pekné“ formátovanie zdrojového textu, rýchlo by zdrojový kód začal pretekať cez pravý okraj okna editora. V praxi teda obyčajne namiesto zápisu:

```
if (D > 0)
{
  // ...
}
else
  if (D == 0)
  {
    // ...
  }
  else
    if (D < 0)
    {
      // ...
    }
```

použijeme úspornejší zápis:

```
if (D > 0)
{
  // ...
}
else if (D == 0)
{
  // ...
}
else if (D < 0)
{
  // ...
}
```

Poznámka: ak je čitateľ zvyknutý z C++ testovať nenulovosť či nulovosť premennej (napr. **x**) takto:

```
if (x) { ... }
```

resp.

```
if (!x) { ... }
```

má v Jave smolu, takéto zápisy sú chybné. Podmienkové výrazy totiž musia byť typu boolean a treba použiť tvary:

```
if (x != 0) { ... }
```

prípadne

```
if (x == 0) { ... }
```

Podmienený príkaz switch

Ďalším príkazom, pomocou ktorého možno vetviť vykonávanie programu v závislosti od určitej podmienky, je príkaz **switch**. Ako napovedá jeho názov, ide doslova o prepínač, ktorý vyberie vetvu programu na základe hodnoty nejakého výrazu. Takýto kód možno naprogramovať aj pomocou zreťazených príkazov **if**, ale pri použití príkazu switch ušetríme dačo miesta a ešte k tomu zabránime viacnásobnému vyhodnocovaniu testovaného výrazu (čo oceníme v prípade, že tento výraz má nejaké vedľajšie účinky).

Ako vyzerá príkaz **switch**? Jeho syntax je nasledujúca:

```
switch ( výraz )
{
case hodnota1 :
   príkazy
case hodnota2 :
   príkazy

   // ...

default:
   príkazy
}
```

Pozrime sa na túto schému podrobnejšie. Za kľúčovým slovom **switch** nájdeme v okrúhlych zátvorkách testovaný výraz. Ten musí mať niektorý z typov **char**, **byte**, **short** alebo **int**. Za výrazom nasleduje v zložených zátvorkách uzavretý blok príkazov. V tomto bloku sa na určitých miestach vyskytujú tzv. návestia v tvare case hodnota. Jednotlivé hodnoty v návestiach musia byť rôzne a kompatibilné s typom výrazu.

Keď program počas behu dospeje k príkazu **switch**, určí hodnotu výrazu a postupne bude prehľadávať hodnoty uvedené v návestiach v bloku príkazov. Ak nájde zhodu, začne vykonávať príkazy za nájdeným návestím až do konca tela príkazu **switch**. V prípade, že ani jedno návestie neobsahuje hľadanú hodnotu, program pokračuje príkazmi za návestím **default**. Toto návestie nie je povinné, a ak sa v tele príkazu **switch** nenachádza, program jednoducho pri neúspešnom hľadaní zhody príkaz **switch** preskočí.

Podobne ako v C++ sa v okamihu nájdenia zhody ďalšie návestia ignorujú a program takpovediac „prepadáva“ cez jednotlivé návestiami oddelené úseky príkazov, až kým dospeje na koniec príkazu **switch**. To ilustruje nasledujúci príklad:

```
class OneTwo
{
  public static void main(String[] args)
  {
    show(3);
    show(2);
    show(1);
  }
  static void show(int n)
  {
    switch (n)
    {
    case 1:
      System.out.print("one ");
    case 2:
      System.out.print("two ");
    default:
      System.out.print("other");
    }
    System.out.println("");
  }
}
```

Tento program po spustení vypíše riadky:

```
other
two other
one two other
```

pretože po odskoku na návestie **case 1**, resp. **case 2**, a vypísaní príslušného textu program jednoducho pokračuje ďalšími príkazmi v tele **switch**. Ak tomu chceme zabrániť, treba použiť iný príkaz break, ktorý v tele príkazu **switch** spôsobí, že program z tohto tela vyskočí a bude pokračovať prvým príkazom za celým blokom **switch**.

Upravená metóda **show()** bude potom vyzerať takto:

```
static void show(int n)
{
  switch (n)
  {
  case 1:
    System.out.print("one");
    break;
  case 2:
    System.out.print("two");
    break;
  default:
    System.out.print("other");
```

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

S takouto verziou **show()** už program vypíše správny výstup:

```
other
two
one
```

Príkaz cyklu while

Dva predchádzajúce príkazy, **if** a **switch**, patria medzi príkazy selekcie, resp. alternatívy, takisto sa im hovorí podmienkové príkazy. Druhý okruh príkazov, na ktorých je založená teória štruktúrovaného programovania, sú príkazy cyklu.

Na realizáciu jednoduchého cyklu s testovaním podmienky na začiatku slúži príkaz **while**. Syntax tohto príkazu je:

```
while ( podmienka )
  príkaz
```

Výraz podmienka musí byť podobne ako v príkaze **if** typu **boolean**, iný typ prekladač odmietne. Na mieste príkazu môže stáť aj zložený príkaz, teda blok príkazov, uzavretý v zložených zátvorkách.

Príkaz **while** sa vykonáva týmto spôsobom: najprv sa vyhodnotí podmienkový výraz. Ak je jeho hodnota pravdivá, teda **true**, vykoná sa príkaz a opätovne sa riadenie programu vráti akoby pred príkaz **while**. Ak bude hodnota podmienky nepravdivá, cyklus **while** sa končí, príkaz sa ďalej vykonávať nebude. Treba si uvedomiť, že ak bude už na začiatku cyklu podmienka nesplnená, telo cyklu sa nevykoná ani raz.

Slovo „while“ v angličtine znamená „kým, pokiaľ“ a podľa toho sa aj cyklus **while** správa: vykonáva príkaz, kým je splnená určitá podmienka. Podmienka sa testuje pred vykonaním tela cyklu.

Ukážme si príklad na tento cyklus. Chceme nájsť najmenšieho spoločného deliteľa dvoch čísel uložených v premenných **a** a **b**. Použijeme známy Euklidov algoritmus:

```
while (a != b)
{
  if (a > b)
    a = a - b;
  else
    b = b - a;
}
```

Cyklus bude „cykliť“ dovtedy, kým budú hodnoty premenných **a** a **b** rôzne. Po skončení cyklu bude v oboch premenných **a** aj **b** ich najmenší spoločný deliteľ (ukončovacou podmienkou je rovnosť premenných **a** a **b**). Program, pochopiteľne, nefunguje pre záporné hodnoty.

V cykle **while**, podobne ako v príkaze **switch**, môžeme použiť príkaz **break**. V tele cyklu však tento príkaz spôsobí okamžité prerušenie vykonávania cyklu a vyskočenie programu za koniec tohto cyklu. Jedným z častých príkladov použitia je cyklus, v ktorom testujeme podmienku niekde „uprostred“:

```
while (true)
{
  a++;
  if (a > 100)
    break;
  b--;
}
```

(Nehľadajte v tomto úseku kódu hlbší zmysel, nie je tam.) Za povšimnutie stojí, že ako podmienku cyklu **while** sme uviedli literál **true**. Takáto podmienka bude vždy splnená, a preto by sa cyklus nikdy neskončil, nebyť toho, že v určitom momente (v tomto prípade, keď premenná a prekročí stovku) príkazom **break** cyklus prerušíme.

Príkaz cyklu do

Pri príkaze **while** sa testuje iteračná podmienka na začiatku cyklu. Môže sa však stať, že budeme potrebovať túto podmienku testovať až na konci cyklu, pretože napríklad jej obsah závisí od toho, čo sa v tele cyklu vykonalo. Pre tento variant máme v Jave k dispozícii príkaz **do** s nasledujúcou syntaxou:

```
do
  príkaz
while ( podmienka );
```

Podmienka musí byť opäť typu **boolean** a príkaz môže byť zložený.

Pri realizácii cyklu **do** sa vykoná príkaz a otestuje sa hodnota podmienkového výrazu. Ak je **true**, cyklus sa bude opakovať, ak **false**, program pokračuje ďalším príkazom za cyklom **do**. Na rozdiel od cyklu **while** sa telo cyklu do vykoná vždy aspoň raz. Na predčasné opustenie tela cyklu môžeme použiť už spomínaný príkaz **break**.

Príklad cyklu do:

```
double d;
do {
  d = Math.random();
}
while (d >= 0.1);
```

Knižničná metóda **Math.random()** vracia náhodné číslo z intervalu **<0; 1>**. Cyklus sa teda bude opakovať, kým táto metóda nevygeneruje číslo menšie ako **0,1**. Výsledkom bude náhodné oneskorenie programu (na súčasných počítačoch, samozrejme, neveľmi badateľné).

Dokončenie nabudúce

V budúcom pokračovaní tému príkazov uzavrieme univerzálnym príkazom cyklu **for** a pomocnými príkazmi **break** a **continue**.

6.časť: Príkazy (dokončenie)

Príkaz cyklu for

Tretí príkaz cyklu, príkaz **for**, je značne univerzálny. Syntax má trošku zložitejšiu:

```
for ( init ; podmienka ; iterácia )
  príkaz
```

Všimnime si podrobnejšie obsah zátvorky. Skladá sa z troch častí oddelených bodkočiarkou. Prvá časť, nazvaná init, predstavuje jeden alebo viac výrazových príkazov (ak ich je viac, oddelíme ich čiarkou). Táto časť sa vykoná pred začiatkom cyklu **for** – môžeme povedať, že to je inicializačná časť. Dávame do nej obyčajne inicializáciu riadiacich premenných cyklu.

Druhá časť, výraz podmienka, má podobný význam ako v cykle **while**. Teda telo cyklu **for** sa bude vykonávať dovtedy, kým bude táto podmienka splnená. Len čo sa podmienka stane nepravdivou, cyklus sa končí. Podmienkový výraz sa vyhodnocuje pred telom cyklu.

Tretia časť, označená ako iterácia, obsahuje výrazy, ktoré sa majú vykonať po každom prechode cyklom. Obyčajne sa tu podľa potrieb menia riadiace premenné. Telo cyklu je tvorené príkazom, ktorý opätovne môže byť aj zložený.

Použitie príkazu for si ukážeme na úseku programu, ktorý spočíta prvých N prirodzených čísel, kde N je nejaké vopred známe číslo, uložené v premennej N:

```
int N = 100;
int i, sum = 0;
for (i = 1; i <= N; i++)
  sum += i;
System.out.println(sum);
```

Spočítavané čísla z intervalu **<1; 100>** uchovávame v premennej **i**. Táto premenná je tzv. riadiacou premennou cyklu, čo sa dá preložiť aj tak, že obsah premennej i sa vhodne mení pri každom prechode telom cyklu. Inicializačná časť príkazu **for**, teda príkaz **i = 1**, nastaví hodnotu premennej **i** na prvú spočítavanú hodnotu, teda na jednotku. Podmienka cyklu znie: **i <= N**. Cyklus sa teda bude opakovať dovtedy, kým premenná i nepresiahne koncovú hodnotu **N**. V tele cyklu vykonávame triviálnu operáciu, pripočítanie aktuálnej hodnoty **i** k premennej sum, kde uchovávame priebežný súčet. Túto premennú nesmieme zabudnúť na začiatku vynulovať. Konečne pravidelné zvyšovanie hodnoty **i** o jednu zabezpečí tretí výraz, **i++**.

Nikde nie je napísané, že riadiacu premennú cyklu musíme v tele cyklu použiť. Keby sme chceli vypísať desaťkrát hoci reťazec „Java pod lupou“ (síce neviem načo, ale ako príklad to postačí), napíšeme cyklus:

```
for (int i = 0; i < 10; i++)
  System.out.println("Java pod lupou");
```

Pozorný čitateľ si isto všimne, že v inicializačnej časti cyklu sa nachádza deklaračný príkaz **int i = 0**. To je dovolené, ale pozor, takto deklarovaná premenná **i** je lokálna pre daný cyklus a po jeho skončení zaniká. Okrem toho nesmieme v inicializácii kombinovať deklaračné a výrazové príkazy. Inak by totiž mohlo dôjsť k dvojznačnosti, napr. pri takomto cykle:

```
for (int i = 0, j = 10; ... )
```

kde sú v inicializačnej časti dva príkazy, by nebolo jasné, či druhý príkaz **j = 10** priraďuje hodnotu **10** existujúcej premennej **j** alebo deklaruje novú premennú **j** typu **int a** nastavuje ju na hodnotu **10**. Z tohto dôvodu platí pravidlo, že ak je prvý príkaz deklaračný, za deklaračné sa považujú aj ostatné.

Riadiacu premennú cyklu nemusíme nevyhnutne pri každom prechode cyklom inkrementovať. Dajme tomu, že potrebujeme vypísať mocniny dvojky od **20** po **210**. Nič ľahšie:

```
for (int i = 1; i <= 1024; i *= 2)
  System.out.println(i);
```

Premenná **i** sa po každej iterácii cyklu zdvojnásobí pomocou výrazu **i *= 2**.

Ak sa zamyslíme nad spôsobom vykonávania cyklu **for**, dôjdeme k záveru, že ho môžeme teoreticky prepísať takýmto spôsobom:

```
init ;
while ( podmienka )
{
  príkaz ;
  iterácia ;
}
```

Tento prepis platí okrem jednej malej výnimky, o ktorej si povieme v ďalšom odseku. K príkazu **for** zostáva už len dodať, že ak chceme cyklus ukončiť predčasne, použijeme známy príkaz **break**.

Príkazy break a continue

Posledné dva príkazy, o ktorých bude reč, sú príkazy **break** a **continue**. Prvý z nich, **break**, sme už niekoľkokrát spomínali. Tento príkaz možno použiť buď vo vetviacom príkaze **switch** na okamžité vyskočenie von z bloku, alebo v príkazoch cyklu **while**, **do** a **for** na okamžité ukončenie cyklu. Inde jeho použitie vedie k chybe pri preklade.

Príkaz **break** však možno použiť aj s jedným argumentom, ktorým je tzv. návestie. O návestiach sme sa už bavili v súvislosti s príkazom **switch**. V Jave však môže byť každý zložený príkaz, resp. každý z príkazov **switch**, **while**, **do**, **for**, ktoré obsahujú zložený príkaz, označený pomocou návestia. Návestím je obyčajný reťazec znakov (platia rovnaké pravidlá ako pre názvy premenných) ukončený dvojbodkou, ktorý umiestnime pred príslušný zložený príkaz. Výskyt príkazu **break** s návestím potom spôsobí okamžité vyskočenie zo zloženého príkazu, ktorý je týmto návestím označený. Je zrejmé, že sa tento break musí v danom zloženom príkaze nachádzať.

Typicky sa príkaz **break** s návestím používa na vysko-

čenie z vnoreného cyklu, čo sa v C++ s obľubou robí pomocou príkazu goto. Tento azda najpolemizovanejší príkaz nepodmieneného skoku v Jave totiž nenájdeme. Ukážme si príklad:

```
outer:
for (int i = 0; i < 10; i++)
{
  for (int j = 0; j < 10; j++)
  {
    if (i == j)
      break;
    if (i == 8)
      break outer;
    System.out.print(i + "-" + j + " ");
  }
  System.out.println("");
}
```

V príklade máme dva vnorené cykly, pomocou ktorých prechádzame pomyselnú súradnicovú mriežku, súradnice máme v premenných **i** a **j**. V každom prechode vnútorným cyklom vypíšeme súradnice aktuálneho bodu mriežky na obrazovku. V prípade, že sa nachádzame na diagonále, kde sú obe súradnice rovnaké (**i == j**), tento a nasledujúce body na aktuálnom riadku „preskočíme“. Okrem toho, keď sa dostaneme na predposledný riadok, pre ktorý bude v **i** hodnota **8**, vyskočíme pomocou break outer z celého cyklu, a teda body na riadkoch so súradnicami **8** a **9** vôbec nevypíšeme. Všetko bude oveľa jasnejšie po spustení programu.

Druhý príkaz, continue, sa používa iba v príkazoch cyklu, teda **while**, **do** a **for**. Pokiaľ ho použijeme bez návestia, spôsobí okamžité ukončenie práve prebiehajúcej iterácie cyklu a skok na pomyselný začiatok cyklu, čo spôsobí nové vyhodnotenie cyklovacej podmienky a podľa jej výsledku prípadné pokračovanie či nepokračovanie v cykle. V príkaze **for** sa pred týmto opätovným testovaním podmienky ešte vykonajú všetky výrazy tretej, iteračnej časti (pozri vyššie). To je ten spomínaný malý rozdiel medzi **for** a jeho náhradou pomocou **while**.

Aj príkaz continue možno použiť s návestím, význam je analogický: ukončí sa práve prebiehajúca iterácia toho cyklu, ku ktorému dané návestie patrí. Modifikujme predchádzajúci príklad tak, že príkazy break nahradíme príkazmi continue:

```
outer:
for (int i = 0; i < 10; i++)
{
  for (int j = 0; j < 10; j++)
  {
    if (i == j)
      continue;
    if (i == 8)
      continue outer;
    System.out.print(i + "-" + j + " ");
  }
  System.out.println("");
}
```

Tentoraz sa z výpisu vynechajú iba body ležiace na diagonále a celý riadok so súradnicou 8.

Prázdny príkaz

Príkazov v Jave nájdeme ešte o niečo viac, ale tie sa vzťahujú buď na výnimky (**try**, **finally**, **catch**, **throw**), alebo na synchronizáciu v multithreadovom prostredí (synchronized), takže si o nich povieme až neskôr. Na záver sa patrí spomenúť tzv. prázdny príkaz, ktorým je jednoducho „nič“ nasledované bodkočiarkou:

```
;
```

Prázdny príkaz možno použiť napríklad ako telo cyklu **for**, ktorého celú náplň sme vyjadrili buď v podmienkovom, alebo v iteračnom výraze.

Zahráme sa na kombajny

A v budúcom pokračovaní seriálu sa vrhneme na polia a prácu s nimi. Teším sa na stretnutie opäť o mesiac.

7.časť: Polia

Polia v Jave sú samostatnou kapitolou. Na jednej strane ich nemožno zaradiť medzi primitívne typy, pretože už svojou podstatou patria medzi agregované údajové typy. Na druhej strane nejde ani o plnohodnotné objekty, tak ako o nich bola reč v krátkom úvode do OOP, predovšetkým preto, lebo polia nie sú tvorené na základe vopred zadanej šablóny (ktorej sa hovorí trieda).

Skôr než sa dostaneme k opisu polí v Jave, zopakujme pre menej zdatných, čo je vôbec to pole. Takže najprv učená definícia: pole je postupnosť prvkov rovnakého typu. Dôležité je slovko „rovnakého“. Typickou situáciou, na ktorej možno ukázať opodstatnenosť existencie polí, je nutnosť vykonať nejakú operáciu na netriviálnom množstve údajov. Mohli by sme síce pre každý údaj vytvoriť samostatnú premennú, ale jednak by sme museli zaradom pre tieto premenné vymýšľať mená (a každé iné!), jednak by sme zamýšľanú operáciu museli do programu zadávať znova a znova, hoci zakaždým nad inou premennou. Čo je, pochopiteľne, neúnosné a vo väčšine prípadov prakticky nerealizovateľné riešenie.

Zavedenie poľa oba problémy vyrieši šmahom ruky. Prvky poľa sú zastrešené pod jedným názvom – názvom poľa – a prístup k nim realizujeme prostredníctvom tzv. indexu (ktorým je v Jave celé číslo). No a generovať postupne celé čísla v požadovanom rozsahu vieme – použijeme cyklus, teda napríklad príkaz for. V tele cyklu vyjadríme požadovanú operáciu nad i-tym prvkom poľa a premennú i jednoducho necháme prebehnúť cez všetky indexy poľa.

Vytvorenie poľa

Skôr než pole začneme používať, treba ho nejakým spôsobom vytvoriť. Podobne ako pri primitívnych typoch musíme najprv deklarovať premennú, pomocou ktorej budeme na pole odkazovať. Čitateľ si isto spomenie, že v deklarácii premennej je nevyhnutné zadať okrem názvu aj jej typ. Ten pri poliach vytvoríme jednoducho: pridaním hranatých zátvoriek **[]** za niektorý z primitívnych typov, ktorý sme si vybrali ako typ prvkov poľa. Ak chceme napríklad pole celých čísel, bude deklarácia vyzerať takto:

```
int[] ai;
```

Od tejto chvíle je ai premennou typu „pole prvkov typu int“. Nanešťastie to nie je všetko. Polia v Jave totiž na rozdiel hoci od C++ nikdy nie sú statické a vždy ich treba vytvoriť dynamicky, za behu programu. V jednej z prvých častí seriálu sme hovorili o rozdiele medzi primitívnymi typmi a referenčnými typmi. Polia ležia niekde na pomedzí, ale bližšie majú k referenčným typom; dôležité v tomto momente je, že premenná typu pole (teda aj naša premenná **ai**) obsahuje v skutočnosti len odkaz, referenciu na skutočný objekt poľa. A tento objekt sme zatiaľ nevytvorili. Premenná ai zatiaľ obsahuje len špeciálnu hodnotu **null**, informujúcu o tom, že **ai** neukazuje nikam.

Samotný objekt, obsahujúci jednotlivé prvky poľa, môžeme vytvoriť dvojakým spôsobom. Prvý je založený na jednoduchom vymenovaní hodnôt všetkých prvkov; tieto hodnoty oddelíme čiarkami, uzavrieme do zložených zátvoriek a doplníme do deklarácie za operátor **=**:

```
int[] ai = { 11, 22, 33, 44 };
```

Týmto riadkom sme naraz deklarovali premennú ai typu „pole celých čísel“, vytvorili objekt "pole štyroch celých čísel" s uvedenými hodnotami a do premennej **ai** sme uložili odkaz na tento novovytvorený objekt. Za povšimnutie stojí, že premenná ai nijako vopred nevie, aký bude počet prvkov poľa, na ktoré bude ukazovať. To je veľmi pozitívna vlastnosť, ktorá programátorovi ušetrí kôpku starostí.

Druhý spôsob vytvorenia objektu poľa je blízky C++. Použijeme kľúčové slovo **new**, za ktoré doplníme typ poľa podľa uvedeného vzoru, tentoraz však s doplneným počtom prvkov medzi hranatými zátvorkami. Príklad:

```
int[] ai = new int[4];
```

Takto vytvorené pole nie je však nijako inicializované a všetky jeho prvky obsahujú implicitné hodnoty, teda nuly. Ak chceme dostať do poľa nejaké rozumné údaje, spravíme to buď v cykle:

```
for (int i = 0; i < 4; i++)
  ai[i] = i * 11;
```

alebo použijeme nasledujúci zápis, ktorý spája vytvorenie pomocou new s inicializáciou:

```
int[] ai = new int[] { 11, 22, 33, 44 };
```

Tentoraz počet prvkov v hranatých zátvorkách nesmieme uviesť. Na pohľad sa celá vec zdá komplikovaná, ale pravidlo je jednoduché: ak zadáme zoznam inicializačných hodnôt, neuvádzame explicitne počet prvkov (ten je zrejmý zo zadaného zoznamu) a naopak.

A teraz si predstavme inú situáciu. Máme deklarovanú premennú typu pole (teda napríklad premennú ai z nášho príkladu) a chceme jej priradiť odkaz na iné pole, hoci aj s inou dĺžkou. V Jave nijaký problém, zapíšeme buď:

```
ai = new int[3];
```

a pole naplníme v cykle, alebo použijeme tvar:

```
ai = new int[] { 111, 222, 333 };
```

Pôvodné štvorprvkové pole je od tejto chvíle stratené, zabudnuté a odsúdené na neľútostný koniec v pažeráku garbage-collectora.

Prístup k prvkom poľa

Keď už máme pole vytvorené, môžeme s jeho prvkami pracovať. **i-ty** prvok poľa **ai** sprístupníme zápisom:

```
pi[i]
```

teda za názov premennej, ktorá odkazuje na dané pole, dáme do hranatých zátvoriek príslušný index. Indexy poľa, ktoré má N prvkov, sú vždy v rozsahu 0 až N-1. Ak by sme zadali index záporný alebo väčší, než je dĺžka poľa, dôjde počas behu programu k výnimke. Hľa, aký rozdiel oproti C++! Tam sme mohli prepisovať pamäť, ako sa nám zapáčilo. Už je teraz jasnejšie tvrdenie, že Java je bezpečnejšia ako C++?

Konštanta alebo premenná použitá ako index musí byť typu int. Môžeme, samozrejme, použiť aj premenné typu **byte**, **short** a **char**, pretože tie sa automaticky konvertujú na typ **int**; nie je však dovolené indexovať polia pomocou premenných (a konštánt) typu **long**.

Ukážme si teraz krátky príklad, v ktorom vytvoríme pole, naplníme ho náhodnými hodnotami a potom spočítame súčet prvkov poľa a ich aritmetický priemer:

```
int[] a = new int[20];
for (int i = 0; i < a.length; i++)
{
  a[i] = (int)(Math.random() * 100);
  System.out.print(a[i] + " ");
}
int sum = 0;
for (int i = 0; i < a.length; i++)
  sum += a[i];
double avg = (double)sum / a.length;
System.out.println("\nSum = " + sum);
System.out.println("Avg = " + avg);
```

Na generovanie náhodných čísel používame metódu **Math.random()**, ktorá vracia pseudonáhodné číslo typu double z intervalu **<0; 1)**. To vynásobíme číslom 100 a výsledok pretypujeme na **int**, takže každý prvok poľa

bude náhodným číslom z intervalu 0 až 99.

V príklade sa vyskytuje ešte jeden nový prvok, a to výraz **a.length**. Polia v Jave sú objektmi a objekty, ako vieme, obsahujú atribúty. V každom poli takto máme k dispozícii konštantný atribút **length**, ktorý udáva počet prvkov poľa. Čo viac si môžeme želať? Ak posielame pole ako argument nejakej funkcii, netreba spolu s ním posielať jeho dĺžku, pretože tú si funkcia zistí sama z poslaného poľa!

Viacrozmerné polia

Prvkami polí nemusia byť len premenné primitívneho typu, rovnako dobre môžeme deklarovať polia objektov. A keď je každé pole samo osebe takým „skoroobjektom“, prečo by sme nemohli deklarovať pole, ktorého prvkami budú opäť polia?

Pojem viacrozmerné polia nie je celkom presný, rovnako ako nebol celkom presný v C++. Viacrozmerné polia nájdeme v Pascale, ale v C++ alebo v Jave ide v skutočnosti o polia, ktorých prvky sú takisto polia. A prvkami týchto polí môžu byť opäť polia a tak ďalej, do hĺbky, ktorú potrebujeme.

Ukážme si teraz na príklade, ako by sme deklarovali „dvojrozmerné“ pole celých čísel. Potrebujeme deklarovať pole prvkov typu „pole celých čísel“. Tento typ už poznáme, je to **int[]**. A poznáme aj pravidlo, ako vytvoriť typ poľa prvkov nejakého typu – pridaním hranatých zátvoriek. Tak dostaneme výsledný **typ int[][]**. Príklad deklarácie:

```
int[][] aai;
```

Takto deklarovaná premenná **aai** opäť neukazuje nikam a obsahuje len hodnotu **null**. Jediné, čo o nej môžeme povedať, je, že ak bude na niečo odkazovať, tak to niečo bude pole polí celých čísel. Ak ju chceme spolu s deklaráciou inicializovať, použijeme napríklad takýto zápis:

```
int[][] aai = { { 1, 2 }, { 3, 4 } };
```

(Všimnite si rekurzívnosť inicializačného zoznamu. Ten je tvorený postupnosťou inicializačných hodnôt oddelených čiarkou a uzavretých do zložených zátvoriek. Ak inicializačná hodnota inicializuje vnorené pole, bude sama osebe podobným zoznamom.)

Teraz premenná **aai** odkazuje na dvojprvkové pole, ktorého každý prvok je opäť dvojprvkové pole celých čísel. Prvé pole, **aai[0]**, obsahuje prvky **aai[0][0]** a **aai[0][1]**, druhé pole, **aai[1]**, obsahuje zase prvky **aai[1][0]** a **aai[1][1]**. V pamäti existujú teda tri polia: prvé pole je to, na ktoré odkazuje premenná **aai**. Toto pole má dva prvky, ktorými sú dve premenné **aai[0]** a **aai[1]** odkazujúce každá na svoje samostatné dvojprvkové pole.

Premennú aai môžeme inicializovať aj takýmto spôsobom:

```
int[][] aai = { null, null };
```

Tentoraz sa v pamäti vytvorilo iba jediné pole, na ktoré odkazuje premenná **aai**. Oba jeho prvky majú hodnotu null, a teda neukazujú nikam. Aby sme ich prinútili ukazovať na nejaké reálne objekty polí, musíme ich inicializovať explicitne:

```
aai[0] = new int[] { 1, 2 };
aai[1] = new int[] { 3, 4, 5 };
```

A teraz dávajte dobrý pozor! Zatiaľ čo premenná **aai[0]** ukazuje podobne ako v predchádzajúcom príklade na dvojprvkové pole obsahujúce hodnoty **1** a **2**, premenná **aai[1]** odkazuje na pole troch prvkov, čísel **3, 4** a **5**! Vo výsledku sme teda dostali dvojrozmerné pole, ktorého druhý rozmer nie je pevne daný, ale sa mení podľa indexu prvej úrovne. (Keď sa to tak vezme, podobnú konštrukciu možno vytvoriť aj v C++, ale trochu menej elegantne a možno aj menej prehľadne. A okrem toho javovské runtime prostredie automaticky kontroluje, či sme zadali správne indexy a či sa náhodou nesnažíme pristupovať „mimo“ poľa.) Mimochodom, uvedené nepravidelné pole zvládneme vytvoriť aj jediným riadkom:

```
int[][] aai = { { 1, 2 }, { 3, 4, 5 } };
```

Ktorý spôsob si vyberieme, závisí od toho, či poznáme inicializačné hodnoty vopred, alebo nie.

Dosiaľ sme ukázali len spôsoby, pomocou ktorých sme najvyššiu úroveň poľa aai inicializovali zoznamom hodnôt. Pochopiteľne, podobné výsledky môžeme dosiahnuť použitím operátora new:

```
int[][] aai = new int[2][2];
```

Takto deklarovaná a inicializovaná premenná **aai** ukazuje na pole dvoch polí, obsahujúcich implicitné hodnoty, teda nuly. Inicializáciu môžeme trochu obmedziť a zapísať:

```
int[][] aai = new int[2][];
```

Teraz premenná aai ukazuje na pole, obsahujúce dve premenné typu **int[]**, ale zatiaľ neinicializované, a teda obsahujúce hodnoty **null**. Explicitne ich inicializujeme napríklad takto:

```
aai[0] = new int[2];
aai[1] = new int[3];
```

Jednotlivé prvky polí druhej úrovne budú v takomto prípade obsahovať, samozrejme, nuly.

Celá teória okolo viacrozmerných polí bude zrejme na prvý pohľad vyzerať zložito a zamotane, hlavne pre začiatočníka, ale len málo vecí je takých, aké sa nám zdajú na prvý pohľad. Stačí sa dobre zamyslieť, vziať si ceruzku a papier a pokúsiť sa celú situáciu nakresliť. A takisto si treba sadnúť k počítaču, napísať zopár deklarácií a snažiť sa zisťovať, ktorá premenná má akú hodnotu, čo je definované a čo pri pokuse o výpis vyhodí výnimku.

Zhrňme všetky doteraz uvedené poznatky do niekoľkých bodov:

- ak v deklarácii uvedieme zoznam inicializačných hodnôt, neuvádzame pre danú úroveň počet prvkov poľa;
- hociktorý inicializačný zoznam na niektorej vnorenej úrovni môžeme nahradiť hodnotou **null**, potom bude táto časť poľa neinicializovaná a príslušné podpolia budeme musieť vytvoriť explicitne;
- ak použijeme na vytvorenie poľa na ľubovoľnej úrovni kľúčové slovo new bez uvedenia inicializačného zoznamu, musíme uviesť počet prvkov tohto poľa;
- pri vytváraní poľa pomocou new môžeme sprava vynechať ľubovoľný počet rozmerov polí nižších úrovní – tieto polia potom budeme musieť vytvoriť explicitne.

Na dokonalé ozrejmenie týchto pravidiel je tu ešte na záver príklad poľa s tromi rozmermi. Použijeme štyri rozličné spôsoby vytvorenia poľa s použitím kľúčového slova new. Prvý spôsob vytvorí naraz celé pole 3 × 2 × 4 prvkov:

```
int[][][] aaai = new int[3][2][4];
```

Jednotlivé prvky budú mať nulové hodnoty.

Druhý spôsob vytvorí naraz prvú a druhú úroveň, tretiu úroveň vytvoríme explicitne a „zubato“:

```
int[][][] aaai = new int[3][2][];
aaai[0][0] = new int[2];
aaai[0][1] = new int[5];
aaai[1][0] = new int[4];
aaai[1][1] = new int[3];
aaai[2][0] = new int[6];
aaai[2][1] = new int[2];
```

Polí tretej úrovne je, ako vidíme z príkladu, spolu šesť (3 × 2) a každé bude mať inú dĺžku.

V treťom spôsobe vytvoríme pomocou new len najvyššiu úroveň poľa, druhú a tretiu úroveň vytvoríme explicitne (ale naraz!)

```
int[][][] aaai = new int[3][][];
aaai[0] = new int[2][4];
aaai[1] = new int[3][3];
aaai[2] = new int[4][2];
```

Ako vidieť z príkladu, prvé dvojrozmerné podpole bude mať 2 × 4 prvky, druhé 3 × 3 prvky a tretie 4 × 2 prvky. Všetky prvky budú inicializované nulovými hodnotami.

Modifikáciou tretieho spôsobu môžeme vytvoriť ešte štvrtý spôsob, v ktorom by sa druhá a tretia úroveň vytvárali oddelene:

```
int[][][] aaai = new int[3][][];
aaai[0] = new int[2][];
aaai[0][0] = new int[2];
aaai[0][1] = new int[5];
aaai[1] = new int[3][];
aaai[1][0] = new int[4];
aaai[1][1] = new int[3];
aaai[1][2] = new int[2];
aaai[2] = new int[4][];
aaai[2][0] = new int[6];
aaai[2][1] = new int[2];
aaai[2][2] = new int[5];
aaai[2][3] = new int[3];
```

Je očividné, že ide v podstate o skombinovanie druhého a tretieho spôsobu. Výsledné pole bude mať **(2 + 5) + (4 + 3 + 2) + (6 + 2 + 5 + 3)**, čo je spolu 32 prvkov.

Špecifiká polí v Jave

Nasledujúci odsek je určený predovšetkým tým čitateľom, ktorí ovládajú jazyk C a nedajbože trpia utkvelou predstavou, že Java je také zjednodušené céčko. Tak predovšetkým v céčku neexistuje samostatný typ pre reťazec znakov. Na prácu so znakovými reťazcami sa používa pole znakov, teda typ **char[]** a v podstate vzhľadom na známu schizofréniu polí v céčku aj typ **char***. V Jave nič takéto neplatí. Hoci je možné vytvoriť pole premenných typu char, bude nám poväčšine nanič, pretože na prácu s reťazcami slúži vstavaný objektový typ **String**, resp. **StringBuffer**. Ich súčasťou je množstvo metód, ktoré na prácu s reťazcami treba, takže nemá prakticky nijaký význam snažiť sa tvrdošijne operovať so znakovými poľami. A nehovoriac o tom, že v Jave nie sú reťazce ukončené nulovým znakom.

Druhá skutočnosť je pre ortodoxných céčkarov oveľa priaznivejšia. Pri deklarácii poľa totiž netreba uvádzať hranaté zátvorky len za názov typu, môžeme ich podobne ako v C/C++ doplniť za názov deklarovanej premennej, dokonca je dovolené oba spôsoby kombinovať. Premennú typu „pole polí celých čísel“ teda môžeme deklarovať až tromi rôznymi spôsobmi:

```
int[][] aai;
int[] aai[];
int aai[][];
```

Všetky tri spôsoby sú rovnocenné.

A zopár príkladov

Skôr než tému polí uzavrieme, ukážeme si ešte dva príklady. V prvom z nich vytvoríme zvláštnu konštrukciu trojuholníkovej „matice“ – dvojrozmerné pole, ktorého počet stĺpcov bude postupne rásť v závislosti od poradového čísla riadka matice:

```
int[][] mat = new int[10][];
for (int i = 0; i < mat.length; i++)
  mat[i] = new int[i+1];

for (int i = 0; i < mat.length; i++)
  for (int j = 0; j < mat[i].length; j++)
    mat[i][j] = (int)(Math.random() * 10);

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

Ponajprv maticu vytvoríme. V cykle **for**, kde sa dynamicky vytvárajú riadky matice, vidieť, že počet prvkov i-teho riadka má byť **i + 1**. Skutočne teda vytvoríme konštruk-

ciu v tvare trojuholníka. V nasledujúcej dvojici cyklov for maticu naplníme náhodnými hodnotami a nakoniec tieto hodnoty vypíšeme na obrazovku. Za pozornosť stojí, že všetky cykly, ktoré s premennou mat pracujú, si zistia dĺžku príslušných polí pomocou výrazu **mat.length**, resp. **mat[i].length**. Netreba dávať pozor, či cykly pracujú nad celým poľom, netreba zavádzať žiadne konštanty, jednoducho, programátorské selanky :-).

V druhom príklade predvedieme násobenie dvoch matíc **A** a **B**, výsledok uložíme do matice **C**. Z algebry vieme, že násobiť možno iba také dve matice, z ktorých prvá má rovnaký počet stĺpcov ako druhá riadkov a výsledok „zdedí“ počet riadkov prvej matice a počet stĺpcov druhej matice:

```
int[][] A = new int[3][4];
int[][] B = new int[4][2];
int[][] C = new int[3][2];

for (int i = 0; i < A.length; i++)
  for (int j = 0; j < A[i].length; j++)
    A[i][j] = (int)(Math.random() * 10);

for (int i = 0; i < B.length; i++)
  for (int j = 0; j < B[i].length; j++)
    B[i][j] = (int)(Math.random() * 10);

for (int i = 0; i < C.length; i++)
  for (int j = 0; j < C[i].length; j++)
    for (int k = 0; k < A[i].length; k++)
      C[i][j] += A[i][k] * B[k][j];

System.out.println("A:");
for (int i = 0; i < A.length; i++)
{
  for (int j = 0; j < A[i].length; j++)
    System.out.print(A[i][j] + " ");
  System.out.println("");
}

System.out.println("\nB:");
for (int i = 0; i < B.length; i++)
{
  for (int j = 0; j < B[i].length; j++)
    System.out.print(B[i][j] + " ");
  System.out.println("");
}

System.out.println("\nA.B:");
for (int i = 0; i < C.length; i++)
{
  for (int j = 0; j < C[i].length; j++)
    System.out.print(C[i][j] + " ");
  System.out.println("");
}
```

Prvé dva cykly for naplnia matice **A** a **B** náhodnými číslami od **0** po **9**. V treťom cykle počítame súčin oboch matíc a výsledok ukladáme do prvkov matice **C**. S výhodou využijeme fakt, že jednotlivé prvky sú inicializované nulami. Keby to tak nebolo, treba na vhodné miesto doplniť riadok **C[i][j] = 0**. Kam, to už ponechávam na čitateľovi. Zvyšok programu sa postará už len o výpis všetkých troch matíc.

Žatva sa skončila…

… a my necháme polia poľami. V budúcej časti prejdeme k triedam a práci s objektmi, dovtedy dovidenia.

8.časť: Objekty a triedy

Na popud istého čitateľa bude počnúc týmto číslom PC REVUE možné nájsť na webových stránkach PC REVUE v sekcii Programujeme kompletné zdrojové texty preberaných príkladov vo forme, na akú v časopise (žiaľ) nie je miesto, a podľa situácie aj niekoľko príkladov navyše. Toľko úvodná informácia, môžeme sa vrátiť k Jave. V ôsmom pokračovaní budeme hovoriť o deklarácii tried.

Java je výrazne objektovo orientovaný jazyk. Javovský program je tvorený množinou objektov, ktoré medzi sebou komunikujú. Komunikácia sa deje prostredníctvom zasielania správ, ktoré sa v Jave realizuje ako klasické volanie procedurálnych jednotiek, tzv. metód jednotlivých objektov. Každý objekt je tak charakterizovaný jednak množinou svojich členov, ktoré opisujú jeho okamžitý stav, a jednak množinou svojich metód, ktoré opisujú možné správanie objektu. Na zoznam metód možno pohliadať ako na exportovaný zoznam správ, na ktoré je objekt schopný reagovať.

(Ešte poznámka k termínu člen: Použil som takúto formuláciu, pretože preklad pôvodného anglického slova field, teda pole, by mohol kolidovať s poľom ako údajovým typom, ktoré sa pre zmenu v angličtine povie array.)

V Jave v zásade nie je možné vytvárať objekty **ad hoc**, čírym vymenovaním ich obsahu (existujú anonymné triedy, ale tie nie je možné používať na najvyššej úrovni). Prv než môžeme začať používať nejaký objekt, treba opísať schému, na základe ktorej sa tento objekt vygeneruje. Takejto schéme sa hovorí deklarácia triedy objektu. V pôvodnom chápaní **OO** prístupu je trieda samostatným metaobjektom, ktorý dokáže generovať nové objekty. Čiastočne je to tak aj v Jave – počas behu programu existuje run-time reprezentácia každej triedy, na základe ktorej bol vytvorený nejaký objekt –, ale koncept triedy je orientovaný viac syntakticky; trieda je teda syntaktická konštrukcia, ktorá opisuje polia a metódy objektov. Výhoda existencie tried je zrejmá: na základe jednej šablóny možno jednoducho vytvárať objekty s rovnakou štruktúrou.

Ako teda vlastne vyzerá zdrojový kód javovského programu? Je zvykom zapisovať deklaráciu každej triedy do samostatného súboru s príponou **.java**. Pri verejných (public) triedach je to dokonca nutnosť a názov každého takéhoto súboru sa musí zhodovať s názvom verejnej triedy v ňom deklarovanej. Pri preklade programu sa súbory **.java** transformujú na súbory s príponou **.class**, ktoré obsahujú binárnu reprezentáciu jednotlivých tried. Ešte treba vyriešiť otázku, ako má program začať. Ako sme už spomínali, jedna z tried musí obsahovať statickú metódu **main()**, ktorú možno vyvolať bez toho, aby existoval konkrétny objekt príslušnej triedy. Táto metóda sa potom musí postarať o vytvorenie zodpovedajúcich objektov (ak je to naším cieľom).

Triedy a ich deklarácia

Deklaráciou triedy zavádzame do programu nový referenčný typ; samotná deklarácia opisuje jeho implementáciu. Triedy možno deklarovať na rôznych úrovniach, podľa toho ich delíme na triedy na najvyššej úrovni (top-level) a na triedy vnorené (nested). Na začiatku sa budeme zaoberať výhradne triedami top-level, vnorené triedy (vnútorné, lokálne či anonymné) prídu na rad až neskôr.

Pre názornosť uvedieme hneď na začiatku kompletnú deklaráciu jednoduchej triedy, na ktorú budeme v ďalšom výklade odkazovať. Triedu nazveme Point, bude totiž predstavovať bod v dvojrozmernom priestore. Taký bod je úplne opísaný svojimi dvoma súradnicami, ktoré budú predstavovať stav objektu. Náš bod bude schopný presunúť sa na určené súradnice a ďalej sa dokáže určitým obmedzeným spôsobom „zobraziť“.

Tu je zdrojový kód triedy **Point**, umiestnený v súbore **Point.java**:

```
public class Point
{
  private int x;    // x-ová súradnica
  private int y;    // y-ová súradnica

  // konštruktory
  Point()
  {
    this(0, 0);
  }

  Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  // move() – presun na nové súradnice
  public void move(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  // getX() – vrátenie x-ovej súradnice
  public int getX()
  {
    return x;
  }

  // getY() – vrátenie y-ovej súradnice
  public int getY()
  {
    return y;
  }

  // show() – „zobrazenie“ bodu
  public void show()
  {
    System.out.println("Point at [" + x + ", " + y + "]");
  }
}
```

Všeobecný zápis deklarácie triedy vyzerá s určitými zjednodušeniami takto:

modifikátory class *MenoTriedy*
{
  telo
}

Pre meno triedy platia rovnaké pravidlá ako pre identifikátor. Je zvykom začínať meno triedy (a pokiaľ je toto meno viacslovným spojením, aj jednotlivé zložky mena) veľkým písmenom, ako napr. **LineBuffer**, **TextFrame** a i.

Pred kľúčovým slovom class sa v deklarácii vyskytujú tzv. modifikátory triedy, ktoré určitým spôsobom upravujú vlastnosti novo deklarovanej triedy. Ako modifikátory triedy možno použiť kľúčové slová **public, protected, private, abstract, static** či **final**. Tri z týchto modifikátorov, menovite **protected, private** a **static**, možno použiť len pre triedy deklarované v rámci inej triedy. Modifikátory **abstract** a **final** dostanú zmysel v okamihu, keď budeme hovoriť o vzťahu dedičnosti medzi triedami, a konečne modifikátor **public** robí triedu verejnou, čo však bližšie vysvetlíme až pri rozprávaní o balíkoch.

Telo deklarácie triedy tvoria deklarácie členov triedy a jej metód. Okrem nich možno v triede nájsť ešte deklarácie vnorených tried a rozhraní, ale tie zatiaľ preskočíme.

Deklarácia členov triedy

Prvou časťou deklarácie triedy je deklarácia jej členov, v ktorých bude uložený stav objektu. Deklarácia členov triedy sa nijako výrazne nelíši od deklarácie lokálnych premenných, samozrejme, ak neberieme do úvahy fakt, že členské premenné takto deklarované sú trvalou súčasťou príslušného objektu, zatiaľ čo lokálne premenné zanikajú po opustení tela príslušného bloku kódu.

Všeobecný zápis deklarácie členov triedy vyzerá teda takto:

modifikátory typ zoznam-deklarátorov ;

kde zoznam deklarátorov je čiarkami oddelená postupnosť deklarátorov v tvare

identifikátor

alebo

identifikátor = inicializátor

Ak sa pozrieme do triedy **Point**, nájdeme tam deklaráciu dvoch členov, **x** a **y**, ktoré obsahujú x-ovú a y-ovú súradnicu príslušného bodu. Deklarácia člena **x** vyzerá takto:

```
private int x;
```

V tejto deklarácii sa nachádza jediný deklarátor, identifikátor **x**. Typom člena **x** bude **int**. Okrem toho je súčasťou deklarácie jeden modifikátor, kľúčové slovo private, ktoré

hovorí o tom, že člen x bude súkromným členom triedy Point a nikto iný k nemu nebude mať prístup. Deklarácia člena y vyzerá podobne.

Obe deklarácie by sme mohli spojiť do jednej:

```
private int x, y;
```

V takejto deklarácii je zoznam deklarátorov tvorený dvoma identifikátormi, **x** a **y**. Dôležité je zapamätať si, že modifikátory a typ sa vzťahujú na všetky identifikátory v zozname deklarátorov.

Ak by sme pociťovali potrebu oba členy inicializovať nejakou hodnotou, stačí deklarácie upraviť na tvar:

```
private int x = 0;
private int y = 0;
```

resp.

```
private int x = 0, y = 0;
```

Inicializovanie premenných (hocijakých) nulovou hodnotou je však zbytočné, to sa robí automaticky a okrem toho je trieda Point napísaná tak, že pokiaľ pri vytváraní novej inštancie (inštancia triedy = objekt vytvorený na základe tejto triedy) neuvedieme počiatočné hodnoty súradníc, implicitne sa budú uvažovať nulové.

V deklarácii jednej triedy sa nesmú vyskytovať dva členy s rovnakým názvom. Je to logické, pretože by pri prístupe k nim dochádzalo k nejednoznačnosti. Je však možné, i keď neveľmi žiaduce, pomenovať člen triedy rovnako ako niektorú z metód triedy.

Keďže jediné, čo odlišuje deklaráciu členských premenných od deklarácie lokálnych premenných, sú modifikátory, zameriame sa teraz na ne. Ako modifikátory členov triedy možno použiť kľúčové slová public, protected, private, static, final, transient a volatile.

Prvé tri modifikátory ovplyvňujú spôsob prístupu k členom. Kľúčovým slovom public sa označujú členy verejné. K takýmto členom môže pristupovať ktokoľvek: samotné metódy objektu, metódy prípadných odvodených objektov, ale aj metódy ľubovoľného iného objektu. Verejné členy treba používať opatrne, pretože v zásade nie je možné zabrániť nekonzistentnej manipulácii s nimi bez toho, aby o tom samotný objekt niečo vedel. Kľúčové slovo private znamená presný opak: takto označené členy sa považujú za súkromné pre daný objekt. Okrem metód objektu k nim nemá prístup nikto, ani metódy odvodených tried. Takéto členy sú úplne konformné s jedným zo základných pilierov OOP, konceptom zapuzdrenia. Jediná možnosť, ako meniť hodnotu súkromných členov, je prostredníctvom metód objektu, a tie sú, samozrejme, úplne pod kontrolou objektu. Konečne posledný modifikátor protected definuje členy ako tzv. chránené; takéto členy sú prístupné na rozdiel od súkromných členov aj odvodeným triedam, nie však metódam ostatných objektov.

Je chybou, ak pri deklarácii člena triedy uvedieme viac ako jeden z modifikátorov public, protected alebo private. Je však možné neuviesť ani jeden z týchto modifikátorov, v takom prípade sa členy správajú ako chránené, s tým rozdielom, že k nim majú prístup metódy všetkých tried nachádzajúcich sa v tom istom balíku. Viac na túto tému neskôr.

Kľúčové slovo static označuje tzv. statické členy. Rozdiel medzi nestatickými a statickými členmi je pomerne zásadný. Zatiaľ čo nestatický člen triedy existuje samostatne v každej kópii objektu, vytvoreného podľa tejto triedy, statický člen existuje pre každú triedu práve jeden. Inak povedané, nestatické členy sú súčasťou každej vytvorenej inštancie triedy a zmena nestatického člena jedného objektu nijako neovplyvňuje hodnotu rovnako pomenovaného člena iného objektu tej istej triedy. Statický člen je, naopak, zdieľaný všetkými objektmi jednej triedy a de facto nie je súčasťou jednotlivých objektov, ale samotnej triedy. Je teda možné s ním pracovať aj vtedy, ak dosiaľ nebola vytvorená nijaká inštancia tejto triedy. Pre nestatické členy sa v angličtine používa aj termín instance variables, statické sú potom class variables. Z oboch pomenovaní je okamžite zrejmé, komu daný člen patrí.

Klasickým príkladom použitia statického člena by mohlo byť počítadlo vytvorených objektov. Každé vytvorenie novej inštancie by spôsobilo inkrementáciu počítadla. Deklarácia člena by mohla vyzerať takto:

```
public static long counter = 0;
```

A v tele konštruktora triedy (čo to je, to sa dozvieme o chvíľu) by sa nachádzal riadok:

```
counter++;
```

Kompletný príklad takejto triedy nájdete na webovej stránke PC REVUE v sekcii Programujeme.

Ďalší modifikátor final hovorí o tom, že člen takto označený svoju hodnotu nebude meniť a bude sa správať ako pomenovaná konštanta. Je pochopiteľné, že final členy treba inicializovať priamo v deklarácii. Často je výhodné deklarovať finálne členy súčasne ako statické, potom máme možnosť pristupovať k nim bez nutnosti vytvárať nový objekt. Ako príklad možno uviesť takúto deklaráciu:

```
public class Const
{
  public static final double PI = 3.1415926;
}
```

Trieda **Const** obsahuje jediný člen **PI**, ktorý je statický a zároveň konštantný. V programe by sme ho mohli sprístupniť jednoduchým zápisom **Const.PI** a používať namiesto literálu **3.1415926**.

Modifikátor transient označuje členy, ktoré nie sú súčasťou perzistentného stavu objektu. Inak povedané, keby sme chceli objekt serializovať a uložiť na perzistentné médium (napríklad do súboru na disku), alebo hoci poslať po sieti v rámci distribuovanej aplikácie, tranzientné členy by sa v takomto prípade neukladali – stav objektu by bolo možné rekonštruovať aj bez hodnôt týchto členov.

Posledný členský modifikátor volatile sa používa na označenie členov, ktorých hodnotu treba pri použití člena vždy nanovo čítať z príslušnej oblasti pamäte. Pri multithreadových aplikáciách je bežné, že si jednotlivé thready uchovávajú lokálne kópie určitých premenných, ktoré sa s hlavnou kópiou synchronizujú len v určitých okamihoch. Modifikátor volatile túto synchronizáciu vynucuje pri každom použití príslušnej premennej.

Deklarácia metód triedy

Metódy na rozdiel od členov opisujú správanie objektov, a preto budú obsahovať kód. Deklarácia metódy vyzerá vo všeobecnosti takto:

```
modifikátory výsledok názovMetódy ( parametre )
{
  telo
}
```

Každá metóda má svoj názov, pre ktorý platia známe pravidlá. Podobne ako pri pomenovávaní tried je zvykom začínať jednotlivé slová vo viacslovnom pomenovaní veľkým písmenom, názvy metód sa však obyčajne začínajú malým písmenom, ako napr. **moveTo**, **convertToBinary** a pod.

Metóda sa z hľadiska vykonávania programu správa rovnakým spôsobom ako klasická céčkovská funkcia. Možno jej poslať parametre a môže volajúcemu objektu vrátiť nejakú hodnotu. Typ výsledku sa v deklarácii zapisuje pred názov metódy. Zoznam formálnych parametrov metódy uvedieme do okrúhlych zátvoriek za názov metódy. Jednotlivé parametre majú tvar klasickej deklaračnej dvojice typ – názov a sú oddelené čiarkou. Názvy jednotlivých parametrov možno používať v tele metódy ako lokálne premenné, ktoré sa naplnia príslušnými hodnotami pri volaní metódy.

Zoberme si ako príklad metódu **move()** triedy **Point**. Tu je pre úplnosť jej deklarácia:

```
public void move(int x, int y)
{
  this.x = x;
  this.y = y;
}
```

Vidíme, že táto metóda prijíma dva parametre, ktoré sme pomenovali **x** a **y** a oba sú typu **int**. Zhodou okolností sa aj oba členy triedy Point volajú **x** a **y**, ale akýkoľvek samostatný výskyt identifikátorov **x** a **y** v metóde move() tieto dva členy zakryje. Preto sme v tele metódy museli použiť zápis **this.x = x**. Jednoduché, nekvalifikované meno **x** napravo od operátora priradenia predstavuje formálny parameter metódy. Naľavo od operátora priradenia sa nachádza výraz **this.x**, ktorý sprístupňuje zakrytý člen **x** triedy **Point**. Kľúčové slovo **this** použité vnútri metódy predstavuje referenciu na objekt, nad ktorým bola metóda vyvolaná. Viac si povieme v odseku venovanom prístupu k členom triedy.

Metóda **move()** nevracia nijakú hodnotu. To je vyjadrené kľúčovým slovom void, použitým ako typ návratovej hodnoty v deklarácii. Na rozdiel od C++ je toto jediná situácia, v ktorej je možné kľúčové slovo void použiť.

Pozrime sa teraz na deklaráciu inej metódy, napr. **getX()**. Vidíme, že tá je deklarovaná s návratovým typom int. V tele metódy musíme preto zabezpečiť, aby sa nejaká hodnota typu int vrátila volajúcemu objektu. Ako vyplýva z názvu metódy, chceme zrejme vrátiť x-ovú súradnicu príslušného bodu, ktorá je uložená v člene **x**. Na určenie návratovej hodnoty použijeme príkaz **return x**. Tento príkaz v ľubovoľnej metóde spôsobí jej okamžité ukončenie a vrátenie príslušnej hodnoty. Je chybou použiť príkaz return bez argumentu v metóde, ktorá bola deklarovaná s návratovým typom iným ako **void**, rovnako je chybou použiť **return** s argumentom v metóde deklarovanej ako **void**.

Formálne parametre metódy môžu byť deklarované s modifikátorom **final**, v takom prípade sa v tele metódy považujú za konštantné a nemožno ich meniť.

Podobne ako pri deklarácii členov môžu byť súčasťou deklarácie metód triedy modifikátory. Pri metódach možno použiť modifikátory **public**, **protected**, **private**, **abstract**, **static**, **final**, **synchronized** a **native**. Prvé tri majú rovnakú funkciu ako pri členoch, teda definujú úroveň prístupu k metódam. Modifikátory **abstract** a **final** súvisia s mechanizmom dedičnosti, takže si o nich povieme až nabudúce.

Popri statických členoch tried možno pomocou modifikátora static ako statické definovať aj metódy. Takéto metódy sa dajú zavolať aj v prípade, že nemáme k dispozícii nijaký objekt príslušnej triedy. V analógii s posielaním správ si možno predstaviť, že príslušnú správu neposielame konkrétnej inštancii triedy, ale samotnému metaobjektu triedy. Pre statické metódy sa vžilo aj pomenovanie class methods na rozdiel od bežných instance methods. Príklad statickej metódy možno opäť nájsť na webovej stránke PC REVUE.

Metódy označené ako synchronized slúžia na zabezpečenie vzájomného vylučovania pri prístupe k objektu (v podstate realizujú údajovú metaštruktúru monitor). Možnostiam synchronizácie v Jave bude venovaná samostatná časť.

Konečne metódy deklarované s modifikátorom native sú implementované pomocou platformovo závislého kódu. Takéto metódy neobsahujú telo, namiesto neho je deklarácia ukončená bodkočiarkou, ako napr.:

```
public native void fileOpen(String name);
```

Deklarácia konštruktorov

Špeciálnou množinou metód sú tzv. konštruktory. Deklaráciu konštruktora spoznáme ľahko: neobsahuje špecifikáciu návratového typu a jeho názov sa presne zhoduje s názvom triedy. Ako napovedá jeho názov, konštruktor je zodpovedný doslova za „skonštruovanie“ objektu pri jeho vytváraní. Konštruktorov môže byť viac, musia sa však líšiť počtom a/alebo typmi argumentov. Konštruktory okrem jednej výnimky nikdy nevoláme klasickým volaním metódy, vyvolávajú sa automaticky pri vzniku novej inštancie objektu.

Od bežných metód sa konštruktory líšia predovšetkým tým, že môžeme k ich deklarácii pripojiť len modifikátory prístupu public, protected či private. Menovite konštruktory nemôžu byť abstraktné ani finálne a už vôbec nie statické.

V triede Point sa nachádzajú dva konštruktory. Jeden akceptuje dva argumenty typu int, ktoré bežným spôsobom skopíruje do členov **x** a **y**, druhý je bez argumentov. Všimnime si telo tohto konštruktora:

```
Point()
{
  this(0, 0);
}
```

Prvým (a v tomto prípade jediným) príkazom v tele konštruktora je **this(0, 0)**, ktorý spôsobí explicitné zavolanie iného konštruktora, ktorý dokáže akceptovať dve celé čísla ako argumenty. Takýto konštruktor „zhodou okolností“ v triede **Point** máme, takže vytvorenie inštancie triedy **Point** bez argumentov vytvorí v skutočnosti bod s nulovými súradnicami. Tento prístup je pomerne bežný – najkomplikovanejší konštruktor sa implementuje celý, zjednodušené verzie konštruktora potom volajú hlavný konštruktor s určitými implicitnými hodnotami. Výhoda je zrejmá: pri zmene spôsobu konštrukcie objektu stačí prepísať kód jediného konštruktora.

V prípade, že trieda neobsahuje nijaký konštruktor, vytvorí prekladač tzv. implicitný konštruktor bez argumentov, ktorého jedinou úlohou bude zavolať konštruktor nadradenej triedy (ale o tom až nabudúce).

Práca s objektmi

Skôr než s objektmi začneme pracovať, musíme ich nejakým spôsobom vytvoriť. To sa deje v dvoch krokoch. Predovšetkým treba deklarovať premennú príslušného referenčného typu, v našom prípade teda napríklad:

```
Point p;
```

Podobne, ako to bolo pri poliach, ani teraz premenná **p** neobsahuje samotný objekt, dokonca dosiaľ žiadna inštancia triedy **Point** vytvorená nebola (**p** obsahuje hodnotu **null**).

Explicitné vytvorenie objektu typu **Point** a priradenie odkazu na tento objekt do premennej **p** zabezpečíme pomocou nasledujúceho príkazu:

```
p = new Point(10, -40);
```

Novú inštanciu teda vytvoríme pomocou kľúčového slova new, za ktoré uvedieme názov príslušnej triedy a do zátvoriek argumenty pre príslušný konštruktor. Vytvorenie inštancie sa vykoná v nasledujúcich krokoch:

1. alokuje sa miesto v pamäti
2. inicializujú sa členy objektu na implicitné (nulové) hodnoty
3. vyhodnotia sa argumenty konštruktora
4. vyberie sa a vykoná sa vhodný konštruktor
5. referencia na nový objekt sa vráti ako hodnota celého výrazu

V prípade, že by sme chceli vytvoriť bod s implicitnými súradnicami, použijeme príkaz:

```
Point q = new Point();
```

Len čo je objekt vytvorený, možno s ním pracovať. Členy aj metódy sprístupňujeme pomocou klasickej bodkovej notácie:

```
objekt.člen
objekt.metóda
```

Keby niektorý z členov triedy **Point** bol verejne prístupný, mohli by sme napísať niečo ako:

```
p.x = 20;   // chyba
p.y = 50;   // chyba
```

ale vzhľadom na to, že oba členy sú privátne, uvedený zápis povedie k chybe pri preklade. Môžeme však objektu p poslať správu move alebo show (obe sú verejné):

```
p.move(20, 50);
p.show();
```

Použitie this

Na záver si ešte objasníme význam kľúčového slova **this**. V ľubovoľnej nestatickej metóde predstavuje this fiktívnu lokálnu premennú, ktorej obsahom je referencia na ten objekt, nad ktorým práve vykonávaná metóda pracuje. Ak teda napríklad objektu, na ktorý odkazuje prv definovaná premenná **p**, pošleme správu **move**, zavolá sa jeho metóda **move()a** v jej tele bude **this** obsahovať referenciu na tento objekt (inými slovami, bude platiť podmienka **p == this).**

Je nadovšetko zrejmé, že odkaz na volaný objekt nemá zmysel v statických metódach, takže v takýchto metódach povedie akékoľvek použitie kľúčového slova this k neodvratnej prekladovej chybe.

Nabudúce

V májovom čísle budeme v rozprávaní o triedach pokračovať – na rad príde dedičnosť, prekrývanie metód, abstraktné triedy a kadečo iné. Dovtedy dovidenia.

9.časť: Dedičnosť

Byl pozdní večer, první máj… hm, ako ten čas letí. No, ale naspäť k Jave. V minulej časti sme hovorili o deklarácii tried a vytváraní objektov. Tento mesiac sa budeme zaoberať spájaním tried do hierarchie dedičnosti a princípom fungovania polymorfizmu.

Vzťah dedičnosti

Pri objektovo-orientovanom návrhu analytik často prichádza k zisteniu, že objekt, ktorý práve navrhol, sa do značnej miery podobá inému objektu, dokáže reagovať na rovnaké správy (i keď nie nutne rovnakým spôsobom) a naviac pridáva svoju vlastnú funkčnosť. Bolo by preto dobré, keby sa nový objekt dal vytvoriť na základe pôvodného, s explicitným vymenovaním nových vlastností a s tým, že všetky nezmenené vlastnosti nový objekt automaticky „zdedí“. Tento jav, nazývaný dedičnosť, patrí medzi základné, hoci nie fundamentálne princípy OOP. Je takisto možné sa stretnúť s iným pomenovaním vzťahu dedičnosti – môžeme povedať, že odvodený objekt, tzv. potomok, je špeciálnym prípadom nadradeného objektu, predka a naopak, predok je všeobecnejší prípad potomka. Vzťah medzi oboma objektmi potom nazývame vzťah špecializácie/generalizácie.

V Jave sa dedičnosť realizuje úpravou deklarácie tried. Ak chceme, aby nová trieda bola potomkom inej, existujúcej triedy, vyjadríme to pomocou kľúčového slova extends. Tu je príklad:

```
// Point.java
public class Point
{
  protected int x, y;

  Point()
  { this(0, 0); }

  Point(int x, int y)
  { this.x = x; this.y = y; }

  public void move(int x, int y)
  { this.x = x; this.y = y; }

  public void show()
  { System.out.println("Point at ["
      + x + ", " + y + "]"); }
}

// ColorPoint.java
public class ColorPoint extends Point
{
  protected int c;

  ColorPoint()
  { this(0, 0, 0); }

  ColorPoint(int x, int y, int c)
  { super(0, 0); this.c = c; }

  public void show()
  { System.out.println("ColorPoint at ["
      + x + ", " + y + "], color " + c); }
}
```

(Z dôvodu úspory miesta je výpis mierne „zhustený“. Čitateľovi sa neodporúča tento spôsob zápisu používať vo svojich programoch.)

Zdrojový text triedy **Point** je prakticky rovnaký ako minule. Nová trieda **ColorPoint** predstavuje špeciálny prípad všeobecného bodu: farebný bod, ktorý okrem svojich dvoch súradníc obsahuje naviac tretiu zložku – farbu. Pozrime sa na deklaráciu triedy **ColorPoint**. Za názvom triedy objavíme spomínané kľúčové slovo extends, za ktorým nasleduje názov nadradenej triedy, čo je v našom prípade **Point**. Nadradená trieda môže byť len jedna, pretože Java na rozdiel od C++ nepodporuje viacnásobnú dedičnosť. V tele triedy sa nachádza len deklarácia polí a metód, ktoré sú „iné“ oproti nadradenej triede. V triede **ColorPoint** konkrétne pribudol jeden údajový člen **c**, ktorý obsahuje informáciu o farbe. Členy **x** a **y** triedy Point sa do novej triedy preberajú automaticky (hovoríme, že sa dedia). Ďalej v triede **ColorPoint** nájdeme dva konštruktory. Tie musíme doplniť vždy, pretože konštruktor, ako špeciálny typ metódy, sa nikdy nededí. Konečne poslednou metódou, ktorej deklaráciu v triede **ColorPoint** nájdeme, je metóda **show()**. Táto metóda by sa za normálnych okolností prebrala z triedy **Point**, ale my chceme, aby sa farebný bod „zobrazoval“ iným spôsobom ako obyčajný bod. Preto sme telo metódy **show()** prepísali vlastnou verziou. Príklad vytvorenia a práce s objektom **ColorPoint** možno nájsť na web stránke PC REVUE v sekcii Programujeme.

Implementácia

Princíp dedenia členov a metód je implementačne veľmi jednoduchý. Objekt potomka v sebe obsahuje akoby kópiu objektu predka, so všetkými jeho členskými premennými. Inštancia našej triedy **ColorPoint** tak napríklad v sebe obsahuje kompletnú inštanciu triedy **Point** a naviac jeden „vlastný“ údajový člen **c**. Členy a metódy objektu môžeme teda rozdeliť na dve skupiny: vlastné (deklarované v danej triede) a zdedené (deklarované v triede či triedach nadradených).

Hierarchia dedičnosti sa neobmedzuje na jednu úroveň. Ak by sme potrebovali vytvoriť špeciálnu verziu farebného bodu, odvodili by sme novú triedu, ktorá by bola potomkom triedy **ColorPoint**. Takto môžeme zostrojiť celý hierarchický strom tried. Koreňom tohto stromu je v Jave preddefinovaná trieda **Object**, ktorá je priamym, či nepriamym predkom každej existujúcej javovskej triedy. Ak pri deklarácii triedy neuvedieme jej predka, automaticky sa nová trieda stáva potomkom triedy Object.

Táto superrodičovská trieda obsahuje niekoľko univerzálnych metód, spoločných všetkým triedam a okrem toho umožňuje neobyčajnú flexibilitu pri narábaní s inštanciami objektov. V Jave, podobne ako vo väčšine objektových jazykov, platí dôležité pravidlo: všade tam, kde sa očakáva inštancia nejakej triedy, môžeme vždy použiť inštanciu triedy odvodenej. Nasledujúci kód je teda korektný:

```
Point[] pts = new Point[3];
pts[0] = new Point(10, 40);
```

```
pts[1] = new ColorPoint(30, 5, 12);
pts[2] = new ColorPoint();
```

Hoci sú jednotlivé prvky poľa pts typu **Point**, priraďujeme im tiež odkazy na objekty typu **ColorPoint**. Prekladač ani run-time chybu neohlási, pretože ako sme hovorili vyššie, každý objekt typu **ColorPoint** v sebe v skutočnosti obsahuje skrytý objekt typu **Point**. Opačný spôsob použitia v zásade nie je možný a vedie k chybe pri preklade či pri behu programu. Ak potrebujeme takto uložiť do poľa alebo inej údajovej štruktúry nepríliš súvisiace objekty, vždy môžeme použiť ako spoločný typ **Object**.

Finálne triedy

V minulom pokračovaní seriálu sme hovorili okrem iného o tom, že každá trieda môže byť deklarovaná s modifikátorom final. Takáto trieda sa považuje za „finálnu" – nie je totiž od nej možné odvodiť žiadneho potomka. Pokus o deklaráciu triedy, ktorá by rozširovala finálnu triedu, skončí chybou pri preklade.

Zmysel existencie finálnych tried vyplýva z práve spomenutého pravidla „potomok môže vždy zastúpiť predka". Ako finálnu obyčajne definujeme takú triedu, pri ktorej sa potrebujeme spoľahnúť na jej funkčnosť a chceme zabrániť možnosti nahradiť ju inou triedou (odvodenou).

Skrývanie členov

Odvodená trieda môže obsahovať údajový člen s rovnakým menom, ako má niektorý člen v nadradenej triede, rovnakého alebo aj odlišného typu. V takejto situácii hovoríme, že člen potomka skrýva člen predka. Termín skrývanie je na rozdiel od nasledujúceho prípadu presným prekladom pôvodného anglického výrazu „hiding". Skrytý člen nie je prístupný bežným spôsobom; ak s ním chceme pracovať, musíme ho explicitne sprístupniť pomocou kľúčového slova super. Zoberme si príklad:

```
class A { protected int x = 1; }

class B extends A
{
  protected int x = 2;
  public void print()
  {
    System.out.println("super.x = " + super.x);
    System.out.println("x = " + x);
  }
}
```

Ak vytvoríme objekt triedy B a zavoláme jeho metódu **print()**, dostaneme takýto výstup:

```
super.x = 1
x = 2
```

Názov **x** v metóde print() predstavuje celočíselný člen triedy B. K členu **x** zdedenému z triedy A sa dostaneme pomocou výrazu **super.x**. Podobne ako **this** sprístupňuje aktuálnu inštanciu, kľúčové slovo **super** predstavuje odkaz na nadradenú inštanciu.

V prípade, že sa potrebujeme dostať k zakrytému členu z niektorého nepriameho predka (t. j. viac ako o jednu úroveň smerom „nahor"), logické by bolo použiť zápis **super.super.x**. To bohužiaľ nefunguje, je nutné explicitne pretypovať this na príslušnú nadtriedu:

```
public class C extends B
{
  protected boolean x;
  public void anotherPrint()
  { System.out.println(((A)this).x); }
}
```

K členu **x** z triedy A sa v metódach triedy C dostaneme pomocou výrazu **((A)this).x**. V prípade, že by zakrytý člen bol statický, možno použiť aj výraz **A.x,** resp. **B.x**.

Pozrime sa ešte, čo sa stane, ak pristupujeme k objektu potomka pomocou premennej typu predka:

```
B b = new B();
A a = b;
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
```

Máme dve premenné **a** a **b**, jednu typu A, druhú typu B. Obe však odkazujú na tú istú inštanciu triedy B. Horeuvedený kód vypíše na obrazovku toto:

```
a.x = 1
b.x = 2
```

Je zrejmé, že výraz **a.x** sprístupňuje skrytý člen **x**, ktorý trieda B zdedila od triedy A. Dôvod je jednoduchý – premenná **a** je typu A a v Jave pri prístupe k údajovým členom triedy nezáleží na skutočnom type objektu, ale len na type výrazu, pomocou ktorého sa na objekt odvolávame. Tento typ je známy už pri preklade. Ak by sme potrebovali pracovať s členom tej triedy, ktorej inštanciou objekt skutočne je, bude nutné použiť iný spôsob, ako uvidíme z nasledujúceho odseku.

Skrývanie a predefinovanie metód

V minulej časti sme hovorili o tom, že dve metódy jednej triedy môžu mať rovnaký názov, ak sa líšia počtom a/alebo typmi svojich argumentov. Tomuto javu sa v angličtine hovorí „method overloading". Vhodný preklad termínu „overloading" akosi neexistuje – často používaný kalk „preťažovanie" má od skutočného významu pomerne ďaleko. Snáď by sa dal použiť výraz prekrývanie metód.

Iná situácia nastane v prípade, že deklarujeme v odvodenej triede metódu s rovnakým názvom, ako má niektorá zdedená metóda. O takejto metóde hovoríme, že predefinuje metódu nadradenej triedy. V angličtine sa používa termín „method overriding". Metóda v odvodenej triede nesmie mať iný typ návratovej hodnoty ako predefinovaná metóda.

Predefinovanie metód používame vtedy, keď nám reakcia nadradenej triedy na príslušnú správu nevyhovuje a chceme, aby sa odvodená trieda správala ináč. V prípade triedy **ColorPoint** zo začiatku článku napríklad chceme, aby sa farebný bod „zobrazoval" iným spôsobom ako obyčajný, bezfarebný bod. Z toho dôvodu v triede **ColorPoint** predefinujeme metódu show(). Naproti tomu metóda **move()** predefinovaná nie je; dedí sa bezo zmeny z triedy **Point**, pretože jej telo nám vyhovuje – farebný bod sa „pohybuje" tak isto, ako obyčajný bod.

Pri predefinovaní metódy sa môže stať, že potrebujeme predsa len zavolať normálne neprístupnú metódu nadradenej triedy. V takej situácii opäť pomôže kľúčové slovo super, ako vidno aj z nasledujúceho príkladu:

```
class A
{
  public void foo()
  { System.out.println("Calling A.foo()"); }
}

class B extends A
{
  public void foo()
  { System.out.println("Calling B.foo()"); }

  public void print()
  {
    foo();
    super.foo();
  }
}
```

Ak nad objektom triedy **B** zavoláme metódu **print()**, dostaneme na obrazovke takýto výpis:

```
Calling B.foo()
Calling A.foo()
```

Výraz **super.foo()** teda vyvolal predefinovanú metódu **foo()** triedy **A**.

V prípade, že v odvodenej triede deklarujeme statickú metódu s rovnakou signatúrou ako niektorá metóda nadradenej triedy, nehovoríme o predefinovaní, ale, podobne ako pri údajových členoch, o skrytí metódy. Aj skrytú metódu možno sprístupniť prostredníctvom kľúčového slova super.

Nie je možné statickú metódu predefinovať nestatickou a takisto nie je možné nestatickú metódu skryť prostredníctvom statickej metódy, oboje vedie k chybe pri preklade.

Pri prístupe k metódam potomka prostredníctvom premennej typu predka dochádza oproti údajovým členom k závažnej odlišnosti. Viac ukáže príklad:

```
B b = new B();
A a = b;
b.foo();
a.foo();
```

Po prebehnutí tohto úseku programu by sme po predchádzajúcej skúsenosti očakávali, že sa v prvom prípade zavolá metóda triedy **B** a v druhom prípade metóda triedy **A.** Dostaneme však nasledovný výstup:

```
Calling B.foo()
Calling B.foo()
```

Čo sa stalo? V oboch prípadoch sa zavolala metóda **foo()** triedy **B**, hoci premenná a je typu **A**. V Jave sa totiž (na rozdiel od prístupu k údajovým členom) pri volaní metódy určí skutočný, run-time typ objektu, ktorého metódu voláme a podľa tohto typu sa vyberie správna verzia metódy z príslušnej triedy.

Ak porovnáme Javu s C++, zistíme, že zatiaľ čo v C++ sú členské funkcie implicitne nevirtuálne, so statickou väzbou (early binding – správna funkcia sa vyberá už počas prekladu), v Jave sú naopak metódy automaticky „virtuálne", dynamicky viazané (late binding – konkrétna metóda sa vyberá až počas behu programu, podľa skutočného typu objektu). Ak v C++ existuje možnosť zmeniť funkciu z nevirtuálnej na virtuálnu, kľúčovým slovom virtual, malo by v Jave byť možné deklarovať metódu ako nevirtuálnu. To sa zabezpečí kľúčovým slovom final. Metóda deklarovaná ako finálna nemôže byť v odvodenej triede predefinovaná a pri jej volaní sa nevyhodnocuje dynamický typ objektu.

Mimochodom, práve dynamickému vyhodnocovaniu metód vďačíme za polymorfné správanie objektov v Jave. Polymorfizmus znamená v preklade mnohotvarosť; rôznym polymorfným objektom môžeme poslať jednu a tú istú správu a ony na ňu zareagujú každý „po svojom". V Jave sa posielanie správ realizuje volaním metód. Programátorovi teda môže byť úplne jedno, nad akým objektom metódu volá; jej správna verzia sa totiž nájde počas behu programu na základe skutočného typu objektu.

Konštruktory a dedičnosť

Na začiatku článku sme spomínali, že konštruktory sa automaticky nededia. Pre každú odvodenú triedu preto musíme napísať konštruktor nanovo. Môže sa však stať, že konštruktor odvodenej triedy už z princípu bude čiastočne opakovať kód konštruktora nadradenej triedy.

Zoberme si náš farebný bod. Ten zrejme budeme konštruovať rovnako ako obyčajný bod, len naviac do jeho stavu uložíme informáciu o farbe. Bolo by preto nanajvýš efektívne, keby sme mali možnosť povedať, že farebný bod sa má skonštruovať presne tak isto ako jeho predok, obyčajný bod, ale naviac si má ešte zapamätať svoju farbu. Našťastie v Jave túto možnosť máme – ako prvý príkaz konštruktora môžeme uviesť kľúčové slovo super nasledované prípadnými argumentmi v okrúhlych zátvorkách. To spôsobí zavolanie príslušného konštruktora nadradenej triedy. (Ak žiaden s vhodným počtom a typmi argumentov neexistuje, dôjde samozrejme pri preklade k chybe.) Volanie superkonštruktora nie je povinné, ale ak ho neuvedieme, resp. ho neumiestnime ako prvý príkaz konštruktora, doplní sa namiesto neho automaticky volanie bezargumentového konštruktora (t. j. **super();**). Problém by mohol nastať v prípade, že nadradená trieda nemá konštruktor bez argumentov. Ak však k takejto situácii dôjde, je „niečo prehnité" v návrhu tried.

Jedinou výnimkou, kedy nie je volanie nadradeného konštruktora nutné a ani sa automaticky nedopĺňa, je prípad, keď z jedného konštruktora voláme iný v rámci tej istej triedy. O tom sme hovorili minule – ako jediný príkaz konštruktora v takomto prípade bude kľúčové slovo this nasledované vhodnými argumentmi v okrúhlych zátvorkách.

Použitie kľúčového slova super pri volaní konštruktora nadradenej triedy vidno v triede ColorPoint. Jej konštruktor s tromi argumentmi prvé dva (to sú súradnice bodu) pekne krásne pošle nadradenému konštruktoru a ani v najmenšom sa nestará, čo sa s nimi bude diať. Zaujíma ho až tretí argument, ktorý si uloží „pre vlastnú potrebu“. Ako vidno, pri takomto prístupe je konštruovanie objektu rozdelené do jednotlivých hierarchických stupňov a každá trieda, resp. jej konštruktor, inicializuje len to, čo sama do objektu pridala. No a ak by sa náhodou stalo, že by bolo nutné zmeniť spôsob inicializácie inštancií triedy Point, trieda ColorPoint zostane bezo zmien.

Riadenie prístupu

Teraz, keď už vieme, ako je to s odvodzovaním nových tried pomocou dedenia, môžeme doplniť informácie z minulej časti seriálu o modifikátoroch prístupu. Ako si čitateľ iste spomenie, deklaráciu každého člena alebo metódy môžeme doplniť jedným z kľúčových slov private, protected alebo public. Pri nastavovaní prístupu nezáleží na tom, či sú deklarované členy alebo metódy statické alebo nestatické.

Členy a metódy deklarované ako private sú pre danú triedu privátne, súkromné. Prístup k nim majú len metódy tejto triedy a nikto iný. Nemožno s nimi pracovať „zvonka“ a nemajú k nim prístup ani triedy odvodené. Privátne členy realizujú princíp zapuzdrenia, skrytia stavu objektu pred jeho okolím. Ešte na vyjasnenie: v originálnej špecifikácii Javy sa hovorí, že privátne členy a metódy sa nededia. To je pravda, ale myslí sa tým, že ich odvodené triedy „nevidia“ a nemôžu k nim nijako pristupovať. Napriek tomu sa však tieto členy v inštanciách odvodených tried fyzicky nachádzajú (inak by to ani nešlo a celý mechanizmus dedičnosti by bol nanič).

Členy a metódy s modifikátorom **protected** („chránené“) sú takisto zvonka neprístupné, na rozdiel od privátnych členov k nim však majú prístup aj metódy odvodených tried. Pokiaľ teda máme objektovú hierarchiu navrhnutú tak, že odvodené triedy musia pracovať s členmi nadradených tried, je nutné tieto členy deklarovať ako chránené. Príkladom môžu byť triedy **Point** a **ColorPoint** z úvodu článku. Údajové členy **x** a **y** sú v triede **Point** deklarované ako protected, vďaka čomu ich metóda **show()** triedy **ColorPoint** môže použiť na výpis súradníc bodu. Treba si však uvedomiť, že v zásade je takmer vždy možné mať všetky členy privátne a sprístupňovať ich pomocou metód, takže voľba medzi private a protected prístupom je viac-menej otázkou požadovanej efektivity implementácie.

Konečne modifikátor public deklaruje členy a metódy ako verejné, takže k nim má prístup ktokoľvek, odkiaľkoľvek. Z pochopiteľných dôvodov nie je veľmi rozumné deklarovať v triede verejné údajové členy – môže sa totiž stať, že ich ktorýsi objekt modifikuje nesprávnym, nekonzistentným spôsobom a už je narušená integrita programu. Verejné členy majú zmysel predovšetkým v triedach, ktoré slúžia ako jednoduché štruktúrované typy (analógia struct v C++).

Aby sme boli presní, existuje ešte jedna úroveň prístupu – ak neuvedieme ani jeden zo spomenutých troch modifikátorov, člen alebo metóda má tzv. implicitnú alebo balíkovú úroveň prístupu. Ale o tom až nabudúce.

Od rozhraní k balíkom

V ďalšom pokračovaní prídu na rad abstraktné triedy, rozhrania a balíky. Dovidenia opäť o mesiac.

10.časť: Abstraktné triedy, rozhrania, balíky

Rozprávanie o triedach by nebolo úplné bez zmienky o abstraktných triedach a abstraktných metódach. S abstraktnými triedami úzko súvisí zvláštny údajový typ – rozhrania.

Abstraktné triedy

Do deklarácie ľubovoľnej triedy môžeme doplniť modifikátor **abstract**. O takej triede potom hovoríme, že je abstraktná, a chceme tým vyjadriť, že je nejakým spôsobom „nedokončená“ alebo sa aspoň za nedokončenú má považovať. Z abstraktných tried nemožno vytvárať inštancie, možno však od nich odvodzovať nové triedy, preto abstraktnú triedu veľmi pravdepodobne nájdeme na niektorom medzistupni či priamo na vrchole objektovej hierarchie. Takisto môžeme deklarovať premennú, ktorej typom bude abstraktná trieda, a uložiť do nej odkaz na reálnu inštanciu niektorej z odvodených tried.

Ako príklad by sme mohli uviesť hierarchiu grafických objektov v kresliacom programe. Na vrchole objektového stromu sa zrejme bude nachádzať nejaká univerzálna trieda, nazvime ju napríklad **GrObject**, implementujúca minimálnu množinu spoločných vlastností každého grafického objektu. V programe však sotva nastane situácia, keď by sme vytvorili reálnu inštanciu tejto triedy – pravdepodobne vytvoríme pole typu **GrObject[]**, ktorého jednotlivé prvky budú ukazovať na inštancie *potomkov* triedy GrObject. Univerzálna trieda je teda výborným kandidátom na „zabstraktnenie“.

Abstraktné metódy

Abstraktná trieda môže byť s výnimkou modifikátora abstract v zásade totožná s neabstraktnou triedou, v takom prípade jediným rozdielom je nemožnosť vytvoriť objekt tejto triedy. (Predpokladá sa, že od abstraktnej triedy budeme odvodzovať potomkov; ak chceme ***len*** zabrániť vytváraniu inštancií, je správnejšie deklarovať konštruktor triedy ako privátny – členy triedy by však potom mali byť výhradne statické.) Pokiaľ nepovieme inak, je ľubovoľný potomok takejto triedy neabstraktný.

Abstraktná trieda však môže obsahovať aj jednu alebo viac *abstraktných* metód. To sú metódy deklarované s modifikátorom abstract. Deklarácia abstraktných metód informuje o názve, počte a typoch argumentov a o type návratovej hodnoty, neobsahuje však telo metódy, t. j. jej implementáciu (deklaráciu treba ukončiť bodkočiarkou). Každá trieda odvodená od takejto abstraktnej triedy potom musí implementovať všetky zdedené abstraktné metódy, inak sa sama stáva abstraktnou. Odvodená trieda môže zdedenú abstraktnú metódu okrem implementácie (t. j. predefinovania plnohodnotnou, neabstraktnou metódou) predefinovať inou, vlastnou abstraktnou metódou.

Ako abstraktné sa obyčajne deklarujú metódy, ktoré je nereálne alebo nemá zmysel naprogramovať pre všeobecný typ objektu. Príkladom v našej kolekcii grafických objektov môže byť metóda show(), ktorá prikazuje objektu, aby sa zobrazil. Zatiaľ čo o každej konkrétnej triede - grafickom objekte, ako napr. bod, čiara, kružnica – dokážeme povedať, ako sa má zobraziť, univerzálna supertrieda GrObject bude mať metódu show() deklarovanú ako abstraktnú – sotva by sme vedeli určiť, ako sa má objekt GrObject zobraziť.

Abstraktnou nemôže byť hociktorá metóda. Prekladač ohlási chybu pri pokuse o abstraktnú deklaráciu metódy s modifikátorom private, static alebo final. Privátnu metódu nie sme schopní implementovať v odvodenej triede, pretože sa nededí, finálna metóda má predefinovanie zakázané a pri statických metódach abstraktnosť akosi stráca zmysel. Ani trieda ako celok nemôže byť deklarovaná ako abstraktná a finálna zároveň.

Abstraktné metódy nemajú telo. Je preto chybou snažiť sa o ich vyvolanie z inštancie odvodenej triedy prostredníctvom kľúčového slova super.

Trieda, ktorá obsahuje abstraktné metódy, či už vlastné alebo zdedené, musí byť deklarovaná s modifikátorom abstract, inak dôjde pri preklade k chybe.

Príklad abstraktnej triedy

Ako príklad si uvedieme veľmi zjednodušenú deklaráciu spomínanej abstraktnej triedy **GrObject**, od ktorej odvodíme už v predošlých častiach použitú triedu **Point**, ktorá bude reprezentovať bod, a ešte jednu triedu **Circle**, ktorá bude implementovať kružnicu. S ohľadom na obmedzený rozsah časopisu sú deklarácie čiastočne oklieštené – kompletnú verziu nájdete na webovej stránke PC REVUE:

```
public abstract class GrObject
{
  public abstract void show();
}

public class Point extends GrObject
{
  private int x, y;

  public void show()
  { System.out.println("Point at ["
    + x + ", " + y + "]"); }
}

public class Circle extends GrObject
{
  private int x, y, r;

  public void show()
  { System.out.println("Circle at ["
    + x + ", " + y + "]"); }
}
```

(V triedach chýbajú predovšetkým konštruktory – ale tie si šikovný čitateľ dokáže doplniť už aj sám.)

Rozhrania

S abstraktnými metódami úzko súvisí koncept rozhraní. *Rozhranie* (interface) v jazyku Java predstavuje referenčný typ. Je blízkym príbuzným triedy, nie je však možné vytvárať jeho inštancie a jeho členmi môžu byť len statické finálne údajové členy a abstraktné metódy. Ktorákoľvek trieda môže o sebe povedať, že dané rozhranie *implementuje* – tým sa zaväzuje, že implementuje všetky abstraktné metódy uvedené v rozhraní. Tento záväzok nemusí nevyhnutne splniť, ale potom sa stáva abstraktnou a musí byť tak aj deklarovaná. Rozhranie sa teda v podstate správa ako (špeciálna) abstraktná trieda.

Súčasťou rozhrania môžu byť údajové členy, ktoré sa automaticky považujú za verejné, finálne a statické (je dovolené príslušné modifikátory vynechať). Takéto konštantné členy môže potom implementujúca trieda používať ako svoje vlastné (t. j. ako keby ich zdedila).

Deklarácia rozhrania je značne podobná deklarácii triedy:

modifikátory **interface** *MenoRozhrania* ↵
extends *Rozhranie*, *Rozhranie …*
{
telo
}

Modifikátormi rozhrania môžu byť kľúčové slová **public, protected, private, abstract** a **static**. Prvé z nich hovorí o tom, že deklarované rozhranie je verejné a prístupné komukoľvek, odkiaľkoľvek. Modifikátory protected, private a static sa používajú iba pri vnorených rozhraniach, ktoré zatiaľ preskočíme. Posledný modifikátor abstract je redundantný, pretože každé rozhranie je implicitne abstraktné. Považuje sa za zastaraný a neodporúča sa ho používať.

Rozhrania možno podobne ako triedy zoraďovať do hierarchie dedičnosti. Rozdiel oproti triedam je v tom, že roz-

hranie môže byť potomkom viacerých superrozhraní. Nadradené rozhrania sa uvádzajú v deklarácii rozhrania spolu so známou klauzulou extends. Zoznam rozhraní oddeľujeme čiarkou. Ak trieda implementuje rozhranie, ktoré je potomkom iných rozhraní, musí táto trieda implementovať okrem abstraktných metód samotného rozhrania všetky zdedené abstraktné metódy jeho predkov.

Telo rozhrania obsahuje deklaráciu jednotlivých abstraktných metód a/alebo údajových členov. Všetky členy i metódy sú implicitne verejné. Inak pre ne platia v zásade rovnaké pravidlá ako pri triedach – nemožno deklarovať dva členy s rovnakým názvom, člen potomka skrýva rovnako pomenovaný člen predka, abstraktná metóda potomka predefinuje abstraktnú metódu predka s rovnakou signatúrou.

Údajové členy musia v deklarácii obsahovať inicializátory (konštanty, ktorých hodnotu nepoznáme, by nám, pochopiteľne, boli na nič). Metódy môžeme deklarovať s modifikátormi **public** a **abstract**, ale je to zbytočné a neodporúča sa ich používať. Metódy rozhraní nesmú byť statické, natívne ani synchronizované.

V súvislosti s viacnásobnou dedičnosťou môže dôjsť k situácii, že rozhranie zdedí dva členy s rovnakým názvom. Nie je to chyba; k tej dôjde až pri pokuse o prístup k tomuto členu použitím nekvalifikovaného mena. Na viacnásobné členy preto treba vždy odkazovať pomocou úplného mena v tvare *Rozhranie.člen*. Ďalej sa môže stať, že rozhranie zdedí jeden a ten istý člen viackrát (cez rôzne cesty v strome dedičnosti). V takom prípade sa tento člen bude v rozhraní nachádzať iba raz (podobne, ako keby sme v C++ použili kľúčové slovo virtual).

Implementácia rozhraní

Trieda, ktorá sa zaväzuje implementovať niektoré rozhranie, musí túto skutočnosť explicitne oznámiť vo svojej deklarácii pomocou klauzuly **implements**, nasledovanej zoznamom implementovaných rozhraní. Táto klauzula sa uvádza za klauzulu **extends**, určujúcou predka triedy. Úplná schéma deklarácie triedy potom vyzerá takto:

```
modifikátory class MenoTriedy ↵
  extends MenoPredka ↵
  implements Rozhranie, Rozhranie …
{
   telo
}
```

Ako vidieť, trieda môže implementovať aj viac ako jedno rozhranie, vďaka čomu je možné do určitej miery zabezpečiť viacnásobné dedenie, ktoré na rozdiel od C++ v tradičnej forme v Jave nenájdeme. Autori Javy zastávajú názor, že nutnosť viacnásobnej dedičnosti je známkou nedokonalého návrhu objektovej hierarchie, ale našťastie doplnením rozhraní do návrhu jazyka umožnili programátorom zariadiť sa podľa svojich potrieb.

Rozhrania, ktoré trieda implementuje, sú jej superrozhraniami (alebo by bolo lepšie nadradenými rozhraniami? …ach, tá slovenčina). Trieda môže mať aj nepriame superrozhrania – to sú jednak predkovia priamych superrozhraní triedy, jednak superrozhrania niektorého z predkov triedy (a aj rôzne iné kombinácie predkov a superrozhraní). Jednoduché, nie?

Trieda môže pri implementácii rozhraní jedinou deklaráciou metódy zabiť dve a viac múch jednou ranou – to v prípade, keď jej superrozhrania obsahujú metódy s rovnakou signatúrou. Tieto metódy sa však nesmú líšiť typom návratovej hodnoty, inak nebude existovať nijaký spôsob, ako implementovať obe metódy a súčasne dodržať pravidlo, že v jednej triede sa nesmú nachádzať dve metódy s rovnakou signatúrou.

Praktická ukážka

Aby sme neomieľali len suchú teóriu, ukážeme si deklaráciu a implementáciu rozhrania na príklade. Do našej kolekcie grafických objektov doplníme rozhranie Resizable, ktoré bude od objektov požadovať implementáciu metódy na zmenu ich veľkosti, a rozhranie **Colorable**, v ktorom sa budú nachádzať metódy na zmenu farby objektu. (Mimochodom, je zvykom rozhrania pomenovávať tak, aby sa končili príponou *-able*.)

```
public interface Resizable
{
  void resize(int factor);
}

public interface Colorable
{
  int RED = 1, GREEN = 2, BLUE = 4;
  void setColor(int color);
  int getColor();
}
```

Pre verejné rozhrania platí rovnaké pravidlo ako pre verejné triedy – ich zdrojový text sa musí nachádzať v súbore s rovnakým názvom a príponou **.java**. Naše dve rozhrania teda treba uložiť do súborov **Resizable.java** a **Colorable.java**.

Rozhranie Resizable obsahuje jedinú metódu resize(), ktorá slúži na zmenu veľkosti objektu podľa hodnoty argumentu **factor**. Druhé rozhranie, **Colorable**, zahŕňa dve metódy na nastavenie farby **(setColor())** a zistenie jej hodnoty (**getColor()**) a tri konštanty RED, GREEN a BLUE.

Vytvoríme ďalej novú triedu **ColorPoint**, ktorá bude potomkom triedy **Point** a navyše bude implementovať rozhranie **Colorable**:

```
public class ColorPoint extends Point
   implements Colorable
{
  private int color;

  public void setColor(int color)
  { this.color = color; }

  public int getColor()
  { return color; }
}
```

(V triede opätovne chýba konštruktor, kompletný zdrojový text nájdete na webe.)

Trieda **ColorPoint** obsahuje implementáciu metód **setColor()** a **getColor()**, takže skutočne implementuje rozhranie **Colorable**.

Ako príklad implementácie rozhrania **Resizable** upravíme deklaráciu triedy **Circle**:

```
public class Circle extends GrObject
   implements Resizable
{
  private int x, y, r;

  public void resize(int factor)
  { r *= factor; }

  public void show()
  { System.out.println("Circle at ["
     + x + ", " + y + "]"); }
}
```

Zmenu veľkosti kružnice dosiahneme jednoduchým vynásobením jej polomeru parametrom **factor**.

Ako sme spomínali, rozhrania v Jave sú samostatnými referenčnými typmi. Je teda možné deklarovať premennú typu niektorého rozhrania. Takáto premenná potom môže (a vlastne musí) odkazovať na inštanciu triedy, ktorá dané rozhranie implementuje. Nasledujúce dva príklady sú teda korektné:

```
Colorable c = new ColorPoint(1, 2);
c.setColor(RED);

Resizable r = new Circle(0, 0, 10);
r.resize(3);
r.show();
```

Balíky

Ako sme niekoľkokrát naznačili v predchádzajúcich častiach, triedy v Jave sú zoskupované do tzv. *balíkov* (packages). Balík je množina tried, ktoré zvyčajne spolu nejakým spôsobom súvisia. Okrem toho je vďaka balíkom možné deklarovať dve a viac tried či rozhraní s rovnakým názvom – pokiaľ sa budú nachádzať v rôznych balíkoch.

Každý balík má svoj názov, ktorý by mal pozostávať výhradne z malých písmen. Členmi balíka môžu byť triedy, rozhrania a podriadené balíky, usporiadanie je teda hierarchické. Možno, samozrejme, vytvoriť balík, ktorý neobsahuje žiadne triedy/rozhrania, ale čisto len podbalíky. Úplné (tzv. plne kvalifikované) meno balíka Q, ktorý patrí do balíka P, vytvoríme klasickou bodkovou metódou ako P.Q. Plne kvalifikované meno ľubovoľnej triedy, ktorá je prvkom niektorého balíka, dostaneme bodkovým spojením úplného mena tohto balíka a mena triedy.

Zoberme si ako príklad triedy štandardného aplikačného programového rozhrania Java API. Všetky nepriamo patria do hlavného balíka s názvom java (alebo v novších verziách Javy aj do balíka javax). Balík java neobsahuje žiadne triedy, zato však v ňom nájdeme mnoho podbalíkov, ako napríklad **awt, lang, net, util** a pod. Každý z týchto podbalíkov je určený pre iný okruh úloh: **java.awt** napríklad poskytuje triedy na realizáciu grafického používateľského rozhrania, triedy balíka **java.net** sú určené na prácu so sieťou atď. V balíku **java.util** sa nachádza okrem množstva ďalších tried aj trieda **Vector**. Jej úplné meno je podľa prv uvedených pravidiel **java.util.Vector**.

Vráťme sa ešte na chvíľu k deklarácii údajových členov a metód tried. V prípade, že v deklarácii neuvedieme nijaký modifikátor prístupu, bude daný člen (či metóda) prístupný v rámci tejto triedy, všetkých jej potomkov, a aj v rámci všetkých tried toho istého balíka, nebude však prístupný *mimo* balíka. Typy prístupu sú v Jave teda štyri: private, protected, implicitný (takisto balíkový) prístup a public.

Implementácia balíkov

Každý javovský virtuálny stroj musí vedieť, ako nájsť binárny obraz každej triedy na základe jej plne kvalifikovaného mena. Jednou z najjednoduchších a súčasne najpoužívanejších metód je prevedenie hierarchickej štruktúry balíkov a tried do hierarchickej štruktúry adresárov a súborov. Balíku java tak bude zodpovedať adresár java, ktorý musí obsahovať podadresáre ako **applet, awt, io, lang, net, util** a pod. Obsah každého podadresára je ekvivalentný obsahu každého balíka. V podadresári java/util by sme teda mali nájsť binárny obraz triedy **Vector**, súbor **Vector.class**. Zdrojový kód každej triedy sa zvykne umiestňovať do rovnakého adresára ako jej binárny tvar, takže v spomínanom podadresári nájdeme pravdepodobne aj súbor **Vector.java**. Teraz je už zrejmé, ako JVM nájde ľubovoľnú triedu – jednoducho zoberie jej plne kvalifikované meno, bodky v ňom nahradí oddeľovačmi adresárov pre danú platformu (t. j. obyčajne **/** alebo ****, prípadne ešte **:**) a k výsledku pripojí príponu **.class**.

Je očividné, že virtuálny stroj musí vedieť, kde je koreň celej hierarchie. To sa väčšinou dozvie z premennej prostredia **CLASSPATH**. Jej obsahom býva postupnosť ciest k adresárom, ktoré potom slúžia ako štartovacie body na hľadanie príslušnej triedy. Je logické, že implementácie umožňujú mať týchto koreňov niekoľko – potom netreba miešať štandardné javovské triedy s triedami, ktoré tvoria vlastný program.

Spomínaný systém ukladania tried v súborovom systéme má jednu drobnú nevýhodu: môže zaberať pomerne dosť veľa miesta na disku, pretože ide o množstvo malých súborov a na niektorých systémoch v takom prípade dochádza k plytvaniu miestom vzhľadom na internú fragmentáciu (to vtedy, keď je veľkosť alokačnej jednotky zbytočne veľká). Riešenie je našťastie triviálne: celý strom binárnych obrazov tried sa skomprimuje do jedného súboru.

Ďalším spôsobom organizácie tried je ich ukladanie do

databázy. Tento spôsob používajú niektoré vývojové prostredia.

Deklarácia package

Vysvetlíme si najprv pojem *kompilačná jednotka*. V prípade ukladania tried do súborového systému je kompilačnou jednotkou súbor so zdrojovým textom jednej alebo viacerých tried. Každá trieda (a pochopiteľne aj rozhranie) môže byť deklarovaná s modifikátorom public alebo bez neho. Pokiaľ je trieda deklarovaná ako verejná (teda s kľúčovým slovom **public**), musí sa jej zdrojový kód nachádzať v súbore s rovnakým názvom, ako je meno tejto triedy, a s príponou **.java**. Z toho vyplýva, že v každej kompilačnej jednotke sa smie nachádzať deklarácia najviac jednej verejnej triedy.

Verejná trieda je viditeľná a prístupná všetkým triedam z toho istého balíka, ako aj každej triede z ľubovoľného iného balíka. Naproti tomu trieda deklarovaná bez modifikátora **public** je prístupná len v rámci svojho balíka, mimo balíka ju používať nemožno. Taká trieda sa správa ako privátna pre svoj balík. Jej deklaráciu možno uložiť do súboru s ľubovoľným názvom.

V každej kompilačnej jednotke sa môže nachádzať informácia o tom, do ktorého balíka náležia triedy v nej deklarované. Túto informáciu zabezpečuje kľúčové slovo package, nasledované plne kvalifikovaným menom balíka. Ak sa deklarácia **package** v súbore nachádza, musí byť uvedená ako prvá, ináč prekladač ohlási chybu.

Deklaráciu **package** môžeme aj vynechať. Robí sa to obyčajne v prípade jednoduchých či pomocných programov alebo v raných štádiách vývoja nejakého projektu. Triedy z takejto kompilačnej jednotky sa stávajú súčasťou tzv. nepomenovaného balíka. Takto boli, mimochodom, definované všetky naše doterajšie triedy.

Deklarácia import

Pri použití ľubovoľnej triedy v programe je v zásade potrebné uviesť jej úplné meno, čo je v prípade zložitejšej balíkovej hierarchie značne frustrujúce. Java preto poskytuje príkaz **import**, pomocou ktorého môžeme požadované triedy sprístupniť pod svojimi krátkymi menami. Tento príkaz, ak ho použijeme, sa musí nachádzať v kompilačnej jednotke medzi príkazom **package** a začiatkom deklarácie tried.

Príkaz import má dva spôsoby použitia. V prvom uvedieme za kľúčové slovo **import** plne kvalifikované meno triedy, ako napríklad:

```
import java.util.Vector;
```

Odteraz sa môžeme na triedu **java.util.Vector** v programe odvolávať použitím výrazne kratšieho **Vector**. Pochopiteľne, nie je možné takto „importovať" dve triedy s rovnakým krátkym menom **Vector** ani nemôžeme v danej kompilačnej jednotke deklarovať inú triedu **Vector**. Tento spôsob takisto neumožňuje importovať podbalíky – nemôžeme napísať import **java.util**; a použiť potom zápis **util.Vector**.

Druhý spôsob je vhodnejší v prípade, že chceme sprístupniť viacero tried z jedného balíka. Ak použijeme napríklad takýto zápis:

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

môžeme na všetky triedy balíka **java.util** odkazovať ich krátkymi menami, ako **Vector, Set, HashMap** a pod.

V našich doterajších programoch sme používali pre výstup textu na obrazovku konštrukciu **System.out.println(...)**. Šikovný čitateľ by už mal byť schopný tento zápis analyzovať a zistiť, že ide o volanie metódy **println()** nad objektom out, ktorý je statickým členom triedy **System**. A tu je malá nezrovnalosť. O niekoľko riadkov vyššie sme predsa tvrdili, že triedy treba sprístupňovať ich úplným menom a tým je pre triedu **System** meno **java.lang.System**. Pravidlo, samozrejme, platí – okrem jednej malej výnimky. Triedy balíka **java.lang** sú natoľko používané, že sa importujú automaticky – každá kompilačná jednotka „obsahuje" fiktívny príkaz **import java.lang.*;**.

Jednoznačné mená balíkov

V praxi pri realizácii projektu rôznymi tímami, prípadne pri využívaní cudzích knižníc tried môže nastať nepríjemná situácia, keď je potrebné do jedného programu integrovať dva balíky s rovnakými názvami. To, samozrejme, možné nie je, preto existuje odporúčanie, ako prideľovať balíkom jednoznačné, unikátne mená.

Postup je jednoduchý. V rámci nášho projektu si vytvoríme hierarchiu a pomenovanie balíkov, ako sa nám hodí. Potom vezmeme svoju doménovú internetovú adresu (musíme, pochopiteľne, nejakú mať), usporiadame v nej subdomény v opačnom poradí a výsledok pripojíme na začiatok názvu každého nášho balíka. Ak by teda napríklad časopis PC REVUE (ktorého doménová adresa je pcrevue.sk) vyvíjal nejaké javovské triedy, ich plne kvalifikované mená by sa začínali reťazcom sk.pcrevue. Povedzme, že by to boli triedy na prácu so zvukom a s grafikou, usporiadané do dvoch balíkov sound a graphics. Úplné mená týchto balíkov by teda zneli **sk.pcrevue.sound** a **sk.pcrevue.graphics**.

Nabudúce

Aby sme konečne prešli od nevyhnutnej, ale nie veľmi záživnej teórie k praxi, v budúcej časti sa pozrieme na zúbky niektorému zo štandardných balíkov Java API. Dovidenia o mesiac.

11.časť: Výnimky

V každom programe sa nachádza chyba. Možno diskutovať, či je táto stará programátorská axióma pravdivá alebo nie, rozhodne však počas behu programu dochádza k situáciám, ktoré nie v súlade s predpokladaným správaním programu. Zoberme si napríklad program, ktorý má naformátovať disketu. Takýto program môže perfektne fungovať s disketou v mechanike, ale čo v prípade, že je mechanika prázdna? Je neprijateľné, aby používateľ dostal správu typu *Invalid drive specification* – program namiesto neľútostného odchodu do binárnych večných lovísk musí vzniknutý chybový stav detegovať a vhodne ošetriť. Iné príklady chybových stavov v programe: nedostatok pamäte, nepripravenosť externých zariadení, pokus o delenie nulou a pod.

Výnimky v Jave

Práca s chybovými stavmi je pre programátora v Jave značne zjednodušená vďaka existencii mechanizmu tzv. *výnimiek*. Pojem výnimka reprezentuje akúkoľvek situáciu, ktorá narúša normálny beh programu a na ktorej výskyt treba náležite reagovať. Súčasne pod výnimkou rozumieme konkrétny objekt (teda inštanciu triedy), ktorý nesie informáciu o vzniknutej chybe.

Každá výnimka v Jave musí byť inštanciou triedy **Throwable**, resp. niektorého jej potomka. Táto trieda má štandardne dve podtriedy: **Error** a **Exception**. Výnimky typu **Error** reprezentujú vážne, obyčajne nenapraviteľné chyby - vnútorná chyba virtuálneho stroja, chyba pri linkovaní tried a pod. Od programu sa neočakáva, že bude na výnimky tohto typu reagovať. Výnimky triedy **Exception** naproti tomu predstavujú bežné chybové situácie, ktoré rozumný program dokáže detegovať a ošetriť. Od oboch tried je, pochopiteľne, odvodená celá hierarchia ďalších, špecifických tried (napr. trieda **IOException**, ktorá reprezentuje chyby pri vstupno-výstupných operáciách).

Ku vzniku výnimky môže dôjsť z troch príčin:

1. výnimka vznikne synchrónne, pri vyhodnocovaní výrazu, pri chybe počas nahrávania a linkovania triedy, alebo pri vyčerpaní dostupných zdrojov (napr. pamäte)

2. výnimka vznikne ako dôsledok vykonania príkazu **throw** (pomocou ktorého môže programátor vyvolať výnimku explicitne)

3. výnimka vznikne asynchrónne (obyčajne ako dôsledok chyby virtuálneho stroja)

Výnimky sa ďalej delia na *kontrolované* a *nekontrolované*. Každá metóda, v ktorej môže dôjsť k neošetrenej kontrolovanej výnimke, to musí explicitne oznámiť vo svojej deklarácii pomocou klauzuly **throws** (pozri ďalej). Nekontrolované výnimky takto deklarovať netreba – buď sa môžu vyskytnúť v ľubovoľnom mieste programu, alebo sú natoľko časté, že ich deklarácia by program zbytočne zneprehľadňovala. Medzi nekontrolované výnimky patria výnimky triedy **Error** a jej potomkov a aj výnimky triedy **RuntimeException** a jej potomkov. Trieda **RuntimeException** je podtriedou triedy **Exception**, reprezentujúcou chyby, ku ktorým môže dôjsť všeobecne počas behu programu, viac-menej nezávisle od konkrétnej operácie (príklad: výnimka **NullPointerException**, ktorá vzniká pri pokuse o dereferenciu hodnoty **null**).

Ošetrenie výnimiek

Pokiaľ v ľubovoľnom mieste programu dôjde k výnimke, okamžite sa ukončí práve vykonávaný príkaz a začne sa hľadať vhodný *obslužný blok* (v angl. origináli exception handler). V prípade, že sa takýto blok nenájde v aktuálnej metóde, táto metóda sa predčasne ukončí a hľadanie sa presunie do volajúcej metódy (hovoríme, že výnimka sa rozšírila do volajúcej metódy). V nej pokračuje hľadanie rovnakým spôsobom od bodu volania metódy, v ktorej výnimka vznikla. Výnimka sa postupne šíri metódami podľa spätného poradia ich volania, až kým sa nenájde vhodný handler. Ak sa stane, že sa výnimka nezachytí ani v metóde **main()**, program oznámi výskyt neošetrenej výnimky a predčasne sa skončí.

Na ošetrenie výskytu výnimiek slúži príkaz **try**, ktorého úplný tvar je nasledujúci:

```
try {
   strážený blok
}
catch ( form-arg )
{
   ošetrenie výnimky
}
finally
{
   finálny blok
}
```

Kľúčové slovo **try** uvádza tzv. strážený blok príkazov. Za **try** blokom sa nachádzajú klauzuly **catch**, ktoré predstavujú jednotlivé obslužné bloky výnimiek. Po kľúčovom slove **catch** v každom bloku nasleduje deklarácia formálneho argumentu (presne tak isto, ako je to v deklarácii metód), ktorý predstavuje zachytenú výnimku. Tento formálny argument musí byť typu **Throwable** alebo niektorého z jeho potomkov. Celý príkaz ukončuje blok **finally**.

Mechanizmus zachytenia výnimky je nasledujúci:

· Ak v bloku **try** nedôjde k žiadnej výnimke, vykoná sa celý tento blok a po ňom všetky príkazy bloku **finally**.

· V prípade, že sa v ktoromkoľvek bode bloku **try** vyskytne výnimka, preskočia sa všetky zvyšné príkazy až po koniec tohto bloku a začne sa hľadať vhodný handler. Pri hľadaní sa porovnáva skutočný typ vyhodenej výnimky s typom formálneho argumentu každej klauzuly **catch**. *Výnimku dokáže zachytiť prvý handler, ktorého typ formálneho argumentu je zhodný s typom výnimky alebo je mu nadradený* (v zmysle dedičnosti; platí tu ostatne rovnaké pravidlo – potomok môže vždy zastúpiť predka).

· Ak sa vhodný handler nájde, inicializuje sa jeho formálny argument inštanciou výnimky, vykoná sa telo handlera a po ňom sa celý príkaz dokončí vykonaním bloku **finally**.

· V prípade, že ani jeden prítomný handler nevyhovuje, výnimka sa rozšíri „o poschodie vyššie". No prv, než

dôjde k opusteniu aktuálnej metódy, vykonajú sa príkazy bloku **finally**.

Ako vidno, blok **finally** sa vykoná za každých okolností, preto doň obyčajne dávame „upratovacie“ príkazy, ktoré uvoľnia alokované prostriedky a vrátia veci do konzistentného stavu.

Na pamäti treba mať fakt, že v prípade výskytu neošetrenej výnimky sa blok **try** nedokončí a celý zvyšok metódy sa preskočí, pretože dôjde k nelokálnemu skoku do volajúcej metódy. A ak volajúca metóda výnimku nedokáže zachytiť, ani ona sa nedokončí a výnimka pokračuje stále vyššie a vyššie. Riadenie sa programu vráti až v okamihu, keď sa nájde vhodný handler – program po spracovaní výnimky pokračuje za tým príkazom **try**, ku ktorému tento handler patrí.

V príkaze **try** sa nemusí nachádzať nijaký obslužný blok **catch**. V takom prípade sa výnimka vždy rozšíri do volajúcej funkcie, ale ešte predtým sa vykoná blok **finally**. Ak chceme blok finally vynechať (to môžeme, nie je povinný), musíme uviesť aspoň jednu klauzulu **catch**.

Ako sme už spomínali, metóda, ktorá vie, že sa z nej môže rozšíriť nejaká výnimka, musí to explicitne deklarovať pomocou klauzuly **throws**. Tá sa nachádza v deklarácii medzi ukončujúcou zátvorkou zoznamu formálnych argumentov metódy a samotným telom funkcie a pozostáva z kľúčového slova **throws** a čiarkou oddeleného zoznamu povolených výnimiek (resp. ich tried). Pozor však, metóda odvodenej triedy nemôže takto deklarovať viac výnimiek ako prekrytá metóda nadradenej triedy.

V zásade možno povinnosť deklarovať typ vyhadzovaných výnimiek obísť tým, že do každej metódy sa doplní deklarácia **throws Throwable**, čím sa postihnú všetky možné výnimky. Toto riešenie sa však neodporúča, pretože sa ním programátor dobrovoľne vzdáva jednej z možností kontroly správnosti programu.

Väčšinu preddefinovaných výnimiek vyhadzujú metódy štandardných tried Java API. Programátor si však môže definovať vlastné triedy výnimiek, odvodené obyčajne od triedy **Exception**. Takéto vlastné výnimky (ale nielen tie) potom môže na vhodnom mieste programu explicitne vyvolať pomocou príkazu **throw**. Jeho syntax je jednoduchá:

throw *výraz* **;**

Uvedený výraz sa musí vyhodnotiť na inštanciu triedy **Throwable** alebo niektorej z jej potomkov. Príkaz **throw** je správne použitý pri splnení aspoň jednej z nasledujúcich troch podmienok:

1. nachádza sa v bloku **try**, ktorý dokáže zachytiť uvedenú výnimku
2. nachádza sa v metóde, ktorá deklaruje, že môže vyhodiť uvedenú výnimku
3. vyhadzuje nekontrolovanú výnimku

Výrazom v príkaze **throw** môže byť už vopred existujúci objekt, ale najčastejšie sa príslušná inštancia výnimky vytvára až v okamihu jej vyhodenia, takže sa obyčajne stretneme so zápisom:

throw new Exception();

Príklad

Na demonštráciu práce s výnimkami poslúži nasledujúci príklad (je pomerne jednoduchý, zložitejší príklad nájde čitateľ ako vždy na webových stránkach PC REVUE):

```
// TestException.java
public class TestException extends Exception
{
  TestException()
  { super(); }
  TestException(String s)
  { super(s); }
}

// Exceptions.java
public class Exceptions
{
  public static void main(String[] args)
  {
    for (int i = 0; i < 4; i++)
    {
      try {
        throwIt(i);
        System.out.println(i + ": no exception");
      }
      catch (Exception e)
      {
        System.out.println(i + ": " + e.getClass()
                           + ": " + e.getMessage());
      }
    }
  }
  static void throwIt(int i) throws TestException
  {
    try {
      switch (i)
      {
      case 0:  i = i/i;
      case 1:  String s = null;
               i = s.length();
      case 2:  throw new TestException("just test");
      default: return;
      }
    }
    finally
    {
      System.out.println("-- throwIt(" + i + ") done --");
    }
  }
}
```

Trieda **TestException** predstavuje našu vlastnú triedu výnimiek. Je odvodená od triedy **Exception**, preto patrí medzi kontrolované výnimky. Dva (zvyčajné) konštruktory len volajú konštruktory nadradenej triedy. Pri vytvorení inštancie výnimky je možné zadať pomocný, vysvetľujúci text.

Metóda **main()** hlavnej triedy **Exceptions** volá metódu **throwIt()** postupne s argumentom **0**, **1**, **2** a **3**. Toto volanie je uzavreté do bloku **try**. Metóda **throwIt()** v závislosti od svojho argumentu vyvoláva rôzne typy výnimiek (prípadne aj žiadnu). Všimnime si, že pomocou klauzuly **throws** táto metóda deklaruje len výnimku **TestException**, ostatné dve (**ArithmeticException** a **NullPointerException**) patria medzi nekontrolované. Žiadna výnimka v metóde nie je ošetrená, takže sa z nej rozšíri von, najprv sa však vďaka bloku **finally** vypíše patričná správa. V metóde **main()** sa nachádza jediný handler, ktorý zachytáva všetky výnimky typu **Exception** a jeho potomkov – to znamená, že zachytí čokoľvek, čo môže „prísť“ z metódy **throwIt()**. Úlohou handlera je vypísať informácie o výnimke pomocou jej metód **getClass()** (zdedená z triedy **Object** – vráti triedu výnimky, ktorá sa v uvedenom výraze automaticky konvertuje na reťazec obsahujúci jej názov) a **getMessage()** (vracia doplnkovú informáciu o výnimke, ktorú sme v prípade triedy **TestException** vytvorili sami – to je ten text „just test“). Výstup programu je takýto:

```
-- throwIt(0) done --
0: class java.lang.ArithmeticException: / by zero
-- throwIt(1) done --
1: class java.lang.NullPointerException: null
-- throwIt(2) done --
2: class TestException: just test
-- throwIt(3) done --
3: no exception
```

Ako vidieť, program sa napriek výskytu výnimočných stavov neskončil chybovým hlásením, namiesto toho výnimky korektne „ošetril“.

Balík java.lang

V druhej polovici tejto časti a v niekoľkých ďalších pokračovaniach seriálu sa postupne prejdeme štandardnými balíkmi Java API a ich najčastejšie používanými triedami a rozhraniami. Nie je, pochopiteľne, možné zapodievať sa na stránkach časopisu každou triedou podrobne, čitateľ sa preto v prípade hlbšieho záujmu bude musieť obrátiť na oficiálnu dokumentáciu k JDK. Ešte upozornenie: opis balíkov je v súlade s aktuálnou verziou JDK 1.3.1 – v prípade použitia staršej verzie môžu v dokumentácii niektoré metódy dosiaľ „neexistovať“.

Prvým na rane je balík **java.lang**, ktorý obsahuje triedy úzko súvisiace s návrhom programovacieho jazyka Java. Veľmi dôležitou súčasťou tohto balíka je trieda **Object**, priamy či nepriamy predok ktorejkoľvek deklarovanej javovskej triedy. **Object** obsahuje niekoľko metód, na ktoré by mal reflektovať prakticky každý objekt. Metóda **clone()** slúži na vytvorenie plytkej kópie objektu (shallow copy, t. j. člen po člene), ale len v prípade, že príslušná trieda implementuje rozhranie **Cloneable** (ktoré je takisto súčasťou balíka **java.lang**). Toto rozhranie neobsahuje nijaké metódy, slúži len na indikáciu, že je možné klonovanie previesť. Ďalšia metóda **equals()** testuje zhodu objektu s ľubovoľným iným objektom. Podmienku zhody dvoch objektov si môže každá trieda definovať sama. Metóda **toString()** vracia textovú reprezentáciu objektu, **getClass()** vracia objekt triedy **Class** (poyri ďalej) nesúci informácie o triede objektu, ďalšia metóda **hashCode()** vypočítava hash hodnotu objektu, ktorú používa napríklad údajová štruktúra **Hashtable**. Metódy **wait()**, **notify()** a **notifyAll()** slúžia na implementáciu synchronizácie procesov a threadov – tejto téme bude venovaná samostatná časť seriálu.

Ďalšia trieda **Class** je nositeľom informácií o triedach programu. Nemá verejný konštruktor, pretože pre každú zavedenú triedu, rozhranie, pole, dokonca aj pre primitívne typy sa vytvára jedna jej inštancia automaticky. Odkaz na túto inštanciu dostaneme volaním spomínanej metódy **getClass()** alebo pomocou statickej metódy **Class.forName()**, ktorej parametrom je názov triedy. Trieda **Class** obsahuje niekoľko metód na zisťovanie typu inštancie (ako **isArray()**, **isInstance()** a pod.) a množstvo metód s názvom v tvare **get...()**, pomocou ktorých sa dajú získať kompletné informácie o zložení triedy, o obsahu jej deklarácie a podobne. Ako príklad možno uviesť metódu **getFields()**, vracajúcu pole objektov **Field** opisujúcich údajové členy triedy. Trieda **Field** a ďalšie opisné triedy, ako **Method** či **Constructor**, sa nachádzajú v podbalíku **java.lang.reflect**.

Trieda **String** predstavuje reťazcový údajový typ. V porovnaní s jazykom C sa v Jave s reťazcami pracuje omnoho komfortnejšie, k dispozícii je množstvo metód na pohodlnú manipuláciu. Nové inštancie triedy String možno vytvoriť klasickým objektovým spôsobom:

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

alebo aj takto (je to špeciálny prípad, iné objekty týmto spôsobom nevytvoríme):

```
String s = "Hello";
```

Nový reťazec možno vytvoriť na základe iného, už existujúceho reťazca, alebo pomocou poľa bajtov či poľa znakov. Metódy triedy **String** umožňujú sprístupňovať jednotlivé znaky (**charAt()**), porovnávať reťazce (**compareTo()**), spájať ich (**concat()**), určovať ich dĺžku (**length()**), hľadať jeden reťazec v druhom (**indexOf()**, **lastIndexOf()**), meniť veľkosť znakov (**toLowerCase()**, **toUpperCase()**) a iné. Dva reťazce možno okrem metódy **concat()** spojiť aj pomocou operátora **+**:

```
String s = "He" + "llo";
```

V triede **String** sa nachádza množstvo prekrytých verzií statickej metódy **valueOf()**, pomocou ktorej možno prevádzať primitívne typy na reťazce. Dá sa však použiť aj takýto zápis:

```
double pi = 3.14159;
String s = "pi = " + pi;
```

Reťazec **s** bude mať hodnotu "**pi = 3.14159**".

Inštancie triedy **String** predstavujú reťazce s nemennou dĺžkou. V prípade, že potrebujeme reťazec konštruovať dynamicky, pridávať k nemu či uberať z neho znaky, treba použiť triedu **StringBuffer**. V nej nájdeme metódy ako **append()**, **delete()**, **insert()**, **replace()**, **substring()** a podobne, ktorých funkciu azda ani netreba vysvetľovať.

V praxi občas nastane situácia, keď by bolo vhodné, aby sa premenná primitívneho typu správala ako objekt. Triviálnym príkladom môže byť snaha vrátiť z metódy viac ako jednu hodnotu, prípadne odovzdať primitívny argument metóde „odkazom“, a nie hodnotou, ako to v Jave štandardne je. Na tieto účely nájdeme v balíku **java.lang** súbor tried **Boolean**, **Byte**, **Character**, **Double**, **Float**, **Integer**, **Long**, **Short** a **Void**. Uvedené triedy predstavujú objektové zapuzdrenie primitívnych typov a s výnimkou tried **Boolean**, **Character** a **Void** sú odvodené od abstraktnej triedy **Number**. Táto trieda deklaruje niekoľko metód s názvom v tvare **xxxValue()** (kde **xxx** je názov primitívneho typu), určených na konverziu „objektového“ čísla na iný typ. Prakticky všetky spomínané triedy sa dokážu skonštruovať aj na základe reťazca. V prípade nemožnosti prevodu generujú výnimku **NumberFormatException**. Medzi ďalšími metódami nájdeme **compareTo()** na porovnávanie zapuzdrených „čísel“, rôzne (statické) klasifikačné metódy, ako **isInfinite()** či **isNaN()** v triedach **Float** a **Double**, a extrémne užitočné statické metódy typu **parse...()**, ktoré slúžia na prevod reťazca na číslo (teda nie zapuzdrený objekt) príslušného typu. Príklad takéhoto prevodu:

```
String s = "13.98";
double d;
try {
  d = Double.parseDouble(s);
}
catch (NumberFormatException e)
{
  d = 0.0;
}
```

(Výnimku **NumberFormatException** treba buď zachytiť, alebo deklarovať v klauzule **throws** obklopujúcej funkcie.) V triede **Character** navyše nájdeme kolekciu klasifikačných metód na zisťovanie typu znakov, ako napr. **isDigit()**, **isLetter()**, **isLowerCase()**, **isWhitespace()** a iné.

Ďalšou významnou triedou balíka **java.lang** je trieda **Math**. Nie je možné vytvárať jej inštancie, všetky jej metódy sú statické a slúžia na realizáciu najrozmanitejších matematických operácií. Ich názvy sú dostatočne opisné: **abs()**, **acos()**, **asin()**, **atan()**, **cos()**, **exp()**, **log()**, **max()**, **min()**, **random()** (tú sme už v minulosti použili!), **round()**, **sin()**, **sqrt()** či **tan()**.

Skupina tried **Thread**, **ThreadGroup**, **ThreadLocal** a **InheritableThreadLocal** v súčinnosti s rozhraním **Runnable** nájde uplatnenie pri programovaní multithreadových aplikácií. Ich použitie spolu s praktickou ukážkou spolupráce viacerých threadov príde na rad neskôr. Multithreading ostatne významne súvisí so synchronizáciou, pretože paralelne bežiace thready často súťažia o prostriedky, ako čas procesora, pamäť a podobne.

Trieda **System** realizuje väzbu programu na hostiteľský systém. Takisto z nej nemožno tvoriť inštancie. Trieda obsahuje tri dôležité členy **in**, **out**, **err**, predstavujúce štandardný vstup, štandardný výstup a štandardný chybový výstup vo forme objektových prúdov, o ktorých bude reč nabudúce. Metóda **exit()** slúži na ukončenie bežiaceho programu, **currentTimeMillis()** vracia aktuálny čas v milisekundách od polnoci 1. 1. 1970 (t. j. klasický unixový čas), **getProperty()**/**setProperty()** a **getProperties()**/**setProperties()** sprístupňujú systémové nastavenia (properties; vzdialená analógia registrov či INI súborov), **setSecurityManager()** umožňuje aktiváciu zvolených bezpečnostných politík.

Trieda **SecurityManager** slúži na implementáciu bezpečnostných politík. Bližší opis, žiaľ, presahuje rámec tohto článku. Trieda **Runtime** sprístupňuje run-time prostredie, v ktorom program beží. Ďalšia trieda **Process** sa používa pri vytváraní natívnych procesov prostredníctvom metód **Runtime.exec()**. Inštancie triedy **Package** obsahujú informácie o javovských balíkoch (názov, dodávateľa, verziu a pod.).

Konečne poslednou významnou triedou balíka **java.lang** je v úvode článku spomínaná trieda **Throwable**, ktorá je rodičovskou triedou všetkých chýb a výnimiek v Jave. Trieda obsahuje okrem iného metódy **getMessage()** a **getLocalizedMessage()**, vracajúce bližší opis výnimky, a metódu **printStackTrace()**, ktorá dokáže vypísať aktuálny stav zásobníka volaných metód v okamihu výskytu výnimky. Podtriedy **Error** a **Exception** ani ich potomkovia nijaké ďalšie metódy nepridávajú – celá hierarchia výnimkových tried slúži predovšetkým na jednoznačnú identifikáciu vzniknutého chybového stavu. Ako sme už uviedli, trieda **Error** a jej podtriedy reprezentujú nezotaviteľné chyby, pri ktorých sa neočakáva, že ich bude aplikácia zachytávať a ošetrovať. Naproti tomu Exception a jej potomkovia indikujú výskyt bežných, ošetriteľných chýb. Z tých, s ktorými sa programátor stretne najčastejšie, možno vybrať **ClassNotFoundException**, oznamujúcu neprítomnosť binárneho obrazu triedy, ktorej inštanciu treba vytvoriť, **InterruptedException**, ktorú je potrebné ošetrovať v prípade použitia metódy **wait()**, ďalej **IOException** a jej podtriedy, ako napr. **FileNotFoundException** či **MalformedURLException**, ktoré indikujú nejakú chybu pri práci so vstupno-výstupnými triedami, ako aj množstvo potomkov triedy **RuntimeException** (tú, ako vieme, netreba deklarovať ani zachytávať), ako **ArithmeticException**, **IllegalArgumentException**, **IndexOutOfBoundsException**, **NoSuchElementException** či obľúbenú **NullPointerException**, ktorých názvy hovoria samy za seba.

Balík **java.lang** toho obsahuje ešte o niečo viac, ale drinu spojenú s podrobným štúdiom dokumentácie si už musí každý programátor „oddrieť“ sám. Nabudúce sa pozrieme na zúbky hlavne balíku **java.io**, v ktorom nájdeme triedy na realizáciu mnohých vstupno-výstupných operácií. Dovidenia o mesiac.

12.časť: Vstup a výstup údajov

V dvanástom pokračovaní seriálu sa oboznámime s triedami, ktoré Java poskytuje na realizáciu vstupných a výstupných operácií. Na príklady, bohužiaľ, tentoraz nezostalo miesto, niekoľko ukážkových programov však možno nájsť na webe PC REVUE v sekcii *Programujeme*.

Triedy File a FileDescriptor

Trieda **File** navzdory svojmu menu neslúži na reprezentáciu súborov, ale na abstraktnú, systémovo nezávislú reprezentáciu ciest k súborom a/alebo adresárom. Cesta sa skladá z nepovinného prefixu (napr. **/** v Unixe, **C:** alebo **** vo Windows) a z postupnosti mien adresárov. Posledné meno môže predstavovať adresár alebo súbor. Oddeľovač adresárov je systémovo závislý a získať ho možno zo statického člena **separator** (typu **String**), resp. **separatorChar** (typu **char**) triedy **File.** Ďalšie dva statické členy, **pathSeparator** a **pathSeparatorChar**, obsahujú znak oddeľujúci zoznam ciest (napríklad v premennej prostredia **PATH**). Cesta môže byť absolútna i relatívna a môže obsahovať špeciálne adresáre **.** a **..** (t. j. aktuálny adresár a nadradený adresár).

Konštruktory triedy **File** akceptujú jeden alebo dva reťazcové argumenty, ktoré obsahujú cestu (pri dvoch argumentoch sa cesty jednoducho spoja – ak pracujeme so súbormi v nejakom adresári, môže byť vďaka tomu cesta k adresáru a cesta ku konkrétnemu súboru uložená každá v samostatnej premennej). V tretej verzii konštruktora možno namiesto prvého z reťazcov použiť inštanciu triedy **File**.

V deklarácii triedy **File** ďalej nájdeme niekoľko informačných metód so samovysvetľujúcimi názvami: **getName()**, **getParent()**, **getPath()**, **getAbsolutePath()**, **getCanonicalPath()** a iné (kanonická cesta je cesta, ktorá neobsahuje adresáre **.** a **..**). Pomocou metód **canRead()** a **canWrite()** môžeme zistiť, či má program právo na čítanie/zápis z/do súboru. Metódy **exists()**, **isDirectory()**, **isFile()**, **isHidden()** zisťujú, či daná cesta existuje, či reprezentuje adresár, resp. súbor a či je skrytá (skrytá cesta má vo Windows nastavený atribút Hidden, v Unixe sa začína bodkou). Metóda **lastModified()** vracia čas poslednej modifikácie daného objektu, **length()** zase jeho dĺžku v bajtoch.

Metóda **delete()** slúži na vymazanie súboru/adresára, **renameTo()** na ich premenovanie. Nový adresár vytvoríme pomocou metód **mkdir()** alebo **mkdirs().** Metódy **list()** a **listFiles()** poskytujú zoznam všetkých súborov/adresárov, ktoré sa nachádzajú vo vybranom adresári (tento výber možno vhodne filtrovať), pomocou metód **setLastModified()** a **setReadOnly()** zmeníme čas poslednej modifikácie súboru či nastavíme príznak read-only. Statická metóda **createTempFile()** príde vhod, ak potrebujeme vytvárať dočasné súbory; o ich automatické zmazanie po skončení programu sa postará metóda **deleteOnExit()**.

Trieda **FileDescriptor** je objektovým zapuzdrením tradičného súborového deskriptora (celé číslo, identifikujúce otvorený súbor na úrovni operačného systému). Inštanciu triedy **FileDescriptor** možno získať volaním metódy **getFD()** nad objektom niektorej z tried **FileInputStream**, **FileOutputStream** alebo **RandomAccessFile** (pozri nižšie). Trieda obsahuje metódu **valid()**, ktorá zisťuje platnosť deskriptora súboru, a metódu **sync()**, ktorá vynúti synchronizáciu systémových vyrovnávacích pamätí so záznamovým médiom.

I/O triedy

V Jave sa vstupné/výstupné operácie realizujú podobne ako v C/C++ pomocou prúdov (streams). *Prúd* je údajový tok, ktorý smeruje od zdroja údajov do programu (vstupný prúd), resp. z programu do spotrebiča údajov (výstupný prúd). Zdrojom/spotrebičom údajov môže byť diskový súbor, konzola (= klávesnica/obrazovka), IPC rúra, sieťový soket a podobne. Výhodou prúdov je unifikovaný spôsob ich obsluhy. Koreňmi objektovej hierarchie I/O tried sú štyri abstraktné triedy: **InputStream**, **OutputStream**, **Reader** a **Writer**. Prvá dvojica predstavuje spoločný základ pre ***bajtové*** prúdy (t. j. postupnosť bajtov), druhá dvojica pre ***znakové*** prúdy (postupnosť znakov).

Trieda InputStream

Trieda **InputStream** obsahuje abstraktnú metódu **read()** v troch obmenách, ktorá slúži na čítanie bajtu alebo poľa bajtov zo vstupného prúdu. Odvodené triedy túto metódu musia vhodne implementovať. Ak v prúde nie sú k dispozícii žiadne údaje, volanie **read()** je blokujúce (okrem prípadu, keď sa dosiahne koniec prúdu). Pomocou metódy **skip()** možno určitý počet bajtov v prúde „preskočiť“. Metóda **available()** indikuje, či sa v prúde nachádzajú nejaké neprečítané dáta, a pomocou metód **mark()** a **reset()** si možno „poznačiť“ pozíciu v prúde a neskôr sa k nej vrátiť. Túto funkčnosť však nemusí poskytovať každý prúd. Poslednou metódou je **close()**, ktorá slúži na korektné ukončenie práce s prúdom.

Pozrime sa teraz na potomkov triedy **InputStream**. Trieda **ByteArrayInputStream** realizuje čítanie bajtov z bajtového poľa. Trieda **FileInputStream** umožňuje sekvenčne čítať bajty z diskového súboru. Argumentom jej konštruktora je cesta k súboru (typu **String** alebo **File**). Trieda **ObjectInputStream** slúži na deserializáciu objektov – táto téma je s ohľadom na svoju zložitosť mimo rozsahu seriálu. Pomocou triedy **PipedInputStream** možno

vytvoriť rúru (pipe) na medziprocesovú (resp. medzithreadovú) komunikáciu. Druhú stranu rúry, čo musí byť inštancia triedy **PipedOutputStream**, možno zadať ako argument konštruktora alebo ako argument samostatnej metódy **connect()**. Trieda **SequenceInputStream** umožňuje zreťaziť dva a viac vstupných prúdov do jedného.

Dôležitým potomkom triedy **InputStream** je trieda **FilterInputStream**, ktorá umožňuje podľa potreby filtrovať iný vstupný prúd (ten sa jej pošle ako argument konštruktora). Samotná trieda **FilterInputStream** prepúšťa všetky bajty bez zmeny, skutočnými filtrami sú až jej potomkovia: trieda **BufferedInputStream**, ktorá obsahuje vyrovnávaciu pamäť (buffer) a tým zefektívňuje čítanie údajov zo zdrojového prúdu, trieda **PushbackInputStream**, ktorej metódu **unread()** môžeme použiť na „vrátenie“ posledného načítaného bajtu späť do prúdu, a predovšetkým trieda **DataInputStream**, pomocou ktorej možno čítať z prúdu primitívne javovské typy strojovo nezávislým spôsobom (existuje jednoznačné mapovanie každého primitívneho typu na postupnosť bajtov a naopak), prostredníctvom metód **readBoolean()**, **readByte()**, **readUnsignedByte()**, **readShort()**, **readUnsignedShort()**, **readChar()**, **readInt()**, **readLong()**, **readFloat()**, **readDouble()** a **readUTF()**.

Uvedené triedy možno medzi sebou vhodne kombinovať, takže ak chceme napríklad čítať hodnoty primitívnych typov zo súboru a využívať pritom vyrovnávaciu pamäť, použijeme nasledujúci zápis:

```
FileInputStream f = new FileInputStream("file.dat");
BufferedInputStream b = new BufferedInputStream(f);
DataInputStream d = new DataInputStream(b);
```

Každý ďalší prúd slúži ako wrapper (v slovenskom preklade azda „zapuzdrovač“) predchádzajúceho prúdu. S finálnym objektom d pracujeme štandardným spôsobom.

Trieda OutputStream

Trieda **OutputStream** je výstupným náprotivkom triedy **InputStream**. Je takisto abstraktná a obsahuje kľúčovú metódu **write()**, realizujúcu zápis bajtu alebo poľa bajtov do výstupného prúdu, metódu **flush()** na vyprázdnenie prípadnej vnútornej vyrovnávacej pamäte a metódu **close()**, ktorá výstupný prúd uzavrie.

Triedy odvodené od **OutputStream** sú poväčšine výstupnými ekvivalentmi potomkov triedy InputStream. Trieda **ByteArrayOutputStream** ako spotrebič údajov využíva pole bajtov, ktoré sa dokáže dynamicky zväčšovať. Pomocou metódy **writeTo()** možno toto pole odoslať do iného prúdu a pomocou **toByteArray()** previesť na samostatný objekt typu **byte[]**. Trieda **FileOutputStream** slúži na zápis bajtov do diskového súboru. Argumentom jej konštruktora môže byť reťazec, objekt triedy **File** alebo objekt triedy **FileDescriptor**. V prvom prípade možno pomocou dodatočného argumentu špecifikovať, či sa má súbor otvoriť v móde **append** (t. j. zapisované bajty sa budú pridávať na koniec už existujúceho súboru. Pomocou triedy **ObjectOutputStream** možno realizovať serializáciu objektov. Na vytvorenie zápisovej zložky medziprocesovej komunikácie je k dispozícii trieda **PipedOutputStream**. K vytváraniu rúr na komunikáciu medzi threadmi sa ešte v seriáli vrátime pri rozprávaní o threadoch – na príklade budeme demonštrovať, ako sa rúra vytvorí a ako sa cez ňu posielajú údaje.

Aj medzi výstupnými prúdovými triedami nájdeme „filtrovaciu“ triedu **FilterOutputStream**. Tá sama osebe prepúšťa údaje bez zmeny. Medzi jej potomkov patrí trieda **BufferedOutputStream**, ktorá implementuje výstupný prúd s vyrovnávacou pamäťou, a takisto trieda **DataOutputStream**, pomocou ktorej možno do výstupného prúdu zapisovať hodnoty primitívnych typov (v binárnom formáte) a reťazcov – na to máme k dispozícii metódy **writeBoolean()**, **writeByte()**, **writeShort()**, **writeChar()**, **writeInt()**, **writeLong()**, **writeFloat()**, **writeDouble()**, **writeBytes()**, **writeChars()** a **writeUTF()**. Poslednou filtrovacou triedou je **PrintStream**, vďaka ktorej môžeme realizovať zápis hodnôt primitívnych typov (a reťazcov) v ľudskom, čitateľnom tvare, prostredníctvom metód **print()** a **println()**. Druhá z metód ukončuje výpis znakom prechodu na nový riadok, čo je obyčajne '**\n**'. Na ilustráciu rozdielu medzi **DataOutputStream** a **PrintStream**: v prvom prípade sa číslo **123** (nech je typu **byte**) zapíše do prúdu ako jeden bajt s hexadecimálnou hodnotou **0x7B**, v druhom prípade ako tri ASCII znaky **1**, **2** a **3**, teda tri hexadecimálne hodnoty **0x31, 0x32** a **0x33**.

Trieda Reader

Trieda **Reader** je prakticky totožná s triedou **InputStream** – je takisto abstraktná a obsahuje rovnaké metódy (s výnimkou metódy **ready()**, ktorá je však inak ekvivalentná metóde **available()**), jediným rozdielom je, že táto trieda z prúdu číta znaky. Tie, ako vieme, majú rozsah 16 bitov, čo sú dva bajty. Trieda **Reader** preto plne podporuje všetky unikódové znaky a nie je viazaná na vybrané osembitové kódovanie národných abecied.

Trieda **Reader** má množstvo potomkov, čiastočne podobných potomkom triedy **InputStream**. Nájdeme tu tak triedu **CharArrayNumber**, ktorá ako zdroj údajov používa pole znakov, triedu **StringReader**, ktorá podobným spôsobom číta znaky z objektu typu **String,** i triedu **PipedReader**, ktorá umožňuje čítanie znakov z rúry IPC. Trieda **InputStreamReader** funguje ako most medzi bajtovými a znakovými prúdmi. Argumentom jej konštruktora je inštancia niektorého vstupného prúdového typu, z ktorého sa bajty čítajú a prekladajú na znaky podľa vybraného kódovania znakov. Táto trieda sa využije pri čítaní znakov zo štandardného vstupu, pretože statický člen **System.in** je typu **InputStream**. Na zjednodušenie práce so súbormi je od triedy **InputStreamReader** odvodená trieda **FileReader**, ktorá je v podstate spojením objektov typu **InputStreamReader** a **FileInputStream**.

Z filtrovacích tried tu nájdeme triedu **FilterReader**, ktorá opätovne slúži len ako bázová trieda, a jej potomka **PushbackReader**, schopného prostredníctvom metódy **unread()** vrátiť posledný načítaný znak späť do prúdu. Ďalšia trieda **BufferedReader** obsahuje vnútornú vyrovnávaciu pamäť, vďaka čomu môžeme naraz čítať celé riadky znakov – na to slúži metóda **readLine()**. Od triedy **BufferedReader** je odvodená jej špeciálna verzia **LineNumberReader**, ktorá prostredníctvom metódy **getLineNumber()** umožňuje zistiť poradové číslo aktuálneho riadka. Číslovanie možno zmeniť pomocou metódy **setLineNumber()**.

Trieda Writer

Posledná zo štvorice, trieda **Writer**, je znakovým ekvivalentom triedy **OutputStream**. Obsahuje rovnaké metódy – menovite metódu **write()**, ktorá slúži na zápis znaku, poľa znakov alebo reťazca do výstupného prúdu, a metódy **flush()** a **close()** na vyprázdnenie, resp. uzavretie prúdu.

Medzi potomkami triedy **Writer** logicky nájdeme triedu **CharArrayWriter**, ktorá zapisuje znaky do znakového poľa (s možnosťou poslať toto pole do iného znakového prúdu pomocou metódy **writeTo()** alebo ho konvertovať na typ **char[]** pomocou metódy **toCharArray()**), podobnú triedu **StringWriter**, využívajúcu namiesto poľa znakov objekt typu **String**, ako aj triedu **PipedWriter**, ako protipól triedy **PipedReader** pri komunikácii prostredníctvom rúry. Prepojenie s bajtovými prúdmi zabezpečuje trieda **OutputStreamWriter**, ktorá zapisované znaky prekladá pomocou zvoleného kódovania na postupnosť bajtov a tie posiela na zadaný bajtový prúd. Táto trieda nájde pre zmenu uplatnenie pri práci so štandardnými výstupnými prúdmi **System.out** a **System.err**. Od triedy **OutputStreamWriter** je odvodená trieda **FileWriter**, ktorá je spojením objektov typu **OutputStreamWriter** a **FileOutputStream**.

Posledné tri triedy sú opäť filtrovacie: trieda **FilterWriter** (ako základ pre tvorbu vlastných filtrov), trieda **BufferedWriter**, zahŕňajúca vyrovnávaciu pamäť, a nakoniec trieda **PrintWriter**, ktorá má rovnakú funkciu ako trieda **PrintStream**, teda umožňuje prostredníctvom metód **print()** a **println()** zapisovať do výstupného prúdu formátovanú reprezentáciu hodnôt primitívnych typov a reťazcov.

Ďalšie triedy

Je len málo programov, ktoré počas svojho behu nepracujú s diskovými súbormi. Triedy z predchádzajúcich odsekov, pomocou ktorých možno čítať a zapisovať z/do súborov, umožňujú len sekvenčný prístup. Pomocou metód **mark()/reset()** a **skip()** toto obmedzenie možno aspoň čiastočne obísť, ale pre pohodlnú prácu so súbormi tak, ako sme zvyknutí napríklad z C/C++, je to málo. Preto v Jave nájdeme samostatnú triedu **RandomAccessFile**, ktorá implementuje možnosť práce so súborom, t. j. čítanie a zápis, formou náhodného prístupu. Slovko náhodný v tomto kontexte, pochopiteľne, znamená skôr ľubovoľný.

Inštanciu triedy **RandomAccessFile** si možno predstaviť ako virtuálne pole bajtov, z ktorého možno čítať a do ktorého možno zapisovať údaje (aj za koniec súboru, lebo v takom prípade sa súbor začne jednoducho predlžovať). Inštancia si pamätá aktuálnu pozíciu v súbore – tú, z ktorej sa bude nasledujúcou operáciou čítať alebo na ktorú sa bude zapisovať.

Argumenty konštruktora triedy **RandomAccessFile** sú dva: prvý typu **String** alebo **File** reprezentuje súbor, s ktorým sa bude pracovať, druhý typu **String** udáva režim práce: "**r**" znamená čítanie, "**rw**" je čítanie + zápis. Iné režimy neexistujú. Pri vytvorení inštancie sa súčasne vytvorí deskriptor súboru, objekt typu **FileDescriptor**, ktorý možno získať pomocou metódy **getFD()**.

Na univerzálne čítanie bajtov zo súboru slúžia metódy **read()** a **readFully()**. Prvá z nich sa správa presne tak isto ako metóda **read()** triedy **InputStream**. Druhá metóda **readFully()** sa od **read()** líši v tom, že čaká, kým nie je k dispozícii presne požadovaný počet bajtov. Metóda **skipBytes()** umožňuje „preskočiť“ stanovený počet bajtov. Univerzálny zápis bajtov do súboru má na starosti metóda **write()**, analogická metóde **write()** triedy **OutputStream**. Na prácu s ukazovateľom aktuálnej pozície slúžia metódy **getFilePointer()** (zistenie) a **seek()** (nastavenie). Zistenie a nastavenie dĺžky súboru zabezpečujú metódy **length()** a **setLength()**. Súbor sa uzavrie volaním metódy **close()**.

Metódy **read()** a **write()** sa na pohodlnú prácu príliš nehodia. Trieda **RandomAccessFile** preto obsahuje rovnaké metódy typu **read*****()** a **write*****()**, ako triedy **DataInputStream** a **DataOutputStream** – pozri vyššie.

V balíku **java.io** sa nachádza ešte jedna užitočná trieda, **StreamTokenizer**, ktorá umožňuje parsing (toto slovo sa mi nedarí vhodne preložiť – rozbor, analýzu?) vstupného údajového toku a jeho rozklad na postupnosť tzv. *tokenov*. (Čitateľ si azda spomenie na jednu z prvých častí tohto seriálu, v ktorej bola reč o lexikálnych jednotkách Javy. Postupnosť lexikálnych jednotiek v zdrojovom súbore je výborný príklad na postupnosť tokenov.) Na bližšie oboznámenie sa s triedou **StreamTokenizer** však niet v seriáli miesta. V budúcnosti sa stretneme s používanejšou triedou **StringTokenizer** – tam bude k dispozícii aj príklad.

Výnimky a rozhrania

Spoločným predkom všetkých výnimiek deklarovaných v balíku **java.io** je trieda **IOException**, ktorá signalizuje, že došlo k nejakej, bližšie nešpecifikovanej chybe pri I/O operácii. Medzi jej potomkov patria triedy **CharConversionException** (chyba pri konverzii znakov), **EOFException** (pri čítaní údajov zo súboru/prúdu signalizuje jeho koniec), **FileNotFoundException** (nastane pri pokuse o otvorenie neexistujúceho súboru), **InterruptedIOException** (pri ukončení threadu vykonávajúceho I/O operáciu signalizuje jej prerušenie),

ObjectStreamException (bázová trieda pre výnimky týkajúce sa serializácie objektov – sama osebe má množstvo ďalších potomkov), **SyncFailedException** (chyba pri synchronizácii počas volania metódy **sync()**), **UnsupportedEncodingException** (pokus o použitie nepodporovaného kódovania znakov) a nakoniec **UTFDataFormatException** (vyskytne sa pri načítaní neplatnej UTF-8 sekvencie zo vstupného prúdu; UTF-8 je jeden zo spôsobov kódovania šestnásťbitových unikódových znakov pomocou osembitových hodnôt).

V balíku **java.io** možno nájsť aj deklaráciu niekoľkých rozhraní: **DataInput** obsahuje všetky tie **read***()** metódy, ktoré implementujú triedy **DataInputStream** a **RandomAccessFile**, podobne rozhranie **DataOutput** obsahuje všetky metódy **write***()**, implementované triedami **DataOutputStream** a **RandomAccessFile**. Obe rozhrania majú aj potomkov, **ObjectInput** a **ObjectOutput**, zahŕňajúcich metódy pre čítanie a zápis objektov, polí a reťazcov. Ďalšie rozhrania **Serializable**, **Externalizable**, **ObjectInputValidation** a **ObjectStreamConstants** sa týkajú serializácie objektov, no a posledné dve rozhrania **FileFilter** a **FilenameFilter** obsahujú metódu **accept()** a používajú sa pri filtrovaní zoznamu súborov v metódach **list()** a **listFiles()** triedy **File**.

13.časť: Balík java.util

V ďalšej časti seriálu sa pozrieme, čo ponúka Java v balíku **java.util**. Ako naznačuje jeho názov, balík **java.util** obsahuje zbierku rozličných užitočných tried, prevažne údajových štruktúr.

Údajové štruktúry

Podstatnú časť balíka **java.util** tvoria triedy a rozhrania tzv. kolekčného rámca (collection framework). Základom tohto rámca je šestica rozhraní **Collection, List, Set, SortedSet, Map** a **SortedMap**.

Rozhrania kolekčného rámca

Rozhranie **Collection** reprezentuje všeobecný pojem *kolekcie* (zbierky) údajov. Kolekciou rozumieme ľubovoľnú skupinu údajov, usporiadanú či neusporiadanú. Jednotlivé údaje nazveme *prvkami* danej kolekcie. Kolekcia môže i nemusí dovoľovať duplicitu prvkov. Java neposkytuje žiadne triedy, ktoré by priamo implementovali rozhranie **Collection**.

Rozhranie **Collection** obsahuje nasledujúce metódy: **size()** vracia počet prvkov v kolekcii, **isEmpty()** zisťuje, či je kolekcia prázdna, contains() testuje prítomnosť objektu v kolekcii, **iterator()** vracia iterátor (pozri ďalej) pre prístup k prvkom kolekcie, **toArray()** prevádza kolekciu na pole objektov, **add()/remove()** pridáva/odoberá prvok do/z kolekcie, **containsAll()** testuje prítomnosť všetkých prvkov kolekcie v inej kolekcii, **addAll()/removeAll()** pridáva/odoberá všetky prvky kolekcie do/z inej kolekcie, **retainAll()** odoberá z kolekcie všetky prvky, ktoré sa nenachádzajú v inej kolekcii, a **clear()** odstraňuje všetky prvky z kolekcie.

Rozhranie **Collection** má niekoľko potomkov. Prvým z nich je rozhranie List, ktoré reprezentuje *zoznam* – usporiadanú postupnosť údajov. Každý prvok zoznamu má priradený celočíselný index (číslovanie sa začína od nuly). Rozhranie **List** pridáva niekoľko nových metód: **get()** vracia prvok na určenej pozícii, **set()** nahrádza prvok na určenej pozícii novým prvkom, **add()** vkladá nový prvok do zoznamu na určenú pozíciu, **remove()** odoberá zo zoznamu prvok na určenej pozícii, **indexOf()/lastIndexOf()** vracia index prvého/posledného výskytu zadaného prvku v zozname, **listIterator()** vracia špeciálny iterátor pre zoznamy a **subList()** vracia podzoznam zoznamu.

Druhý potomok rozhrania **Collection** má názov **Set** a predstavuje matematický pojem *množiny* – neusporiadané kolekcie údajov bez duplicitných prvkov. Rozhranie **Set** nezavádza žiadne nové metódy. Má však potomka, rozhranie **SortedSet**, ktoré reprezentuje *usporiadanú množinu*. Takáto množina má svoje prvky usporiadané buď „prirodzeným“ spôsobom, alebo na základe objektu implementujúceho rozhranie **Comparator**, obsahujúce dve metódy, **compare()** a **equals(),** ktoré vhodným spôsobom definujú reláciu úplného (sic!) usporiadania.

V rozhraní **SortedSet** nájdeme zopár nových metód: **comparator()** vracia príslušný objekt **Comparator**, **subSet()** vracia vybranú podmnožinu danej množiny, **headSet()** vracia všetky prvky „menšie“ ako zadaný prvok, **tailSet()** vracia všetky prvky „väčšie alebo rovné“ ako zadaný prvok a **first()/last()** vracia „najmenší/najväčší“ prvok množiny.

Piate údajové rozhranie **Map** (nie je potomkom rozhrania **Collection**) predstavuje *mapu* – objekt, ktorý realizuje transformáciu zadaného kľúča na hodnotu. Mapa teda obsahuje množinu dvojíc kľúč-hodnota. Usporiadanie mapy je v zásade určené usporiadaním týchto dvojíc (presnejšie poradím, v akom sú tieto dvojice sprístupňované iterátormi).

Rozhranie **Map** obsahuje niekoľko metód významovo zhodných s metódami rozhrania **Collection**, menovite **size(), isEmpty(), clear()**, a ďalej vlastné špecifické metódy: **containsKey()** testuje prítomnosť zadaného kľúča, **containsValue()** testuje prítomnosť zadanej hodnoty, **get()** vracia hodnotu patriacu zadanému kľúču, **put()** priraďuje zadanému kľúču hodnotu, **remove()** odstraňuje kľúč z mapy, **putAll()** kopíruje do mapy všetky dvojice kľúč-hodnota z inej mapy, **keySet()** vracia množinu všetkých kľúčov (ako objekt typu **Set**), **values()** vracia množinu všetkých hodnôt (ako objekt typu **Collection**) a **entrySet()** vracia množinu všetkých dvojíc (ako objekt typu **Set**). Dvojice kľúč-hodnota, ktoré vracia posledná z vymenovaných metód, sú typu **Map.Entry**, čo je statické rozhranie deklarované v rozhraní **Map**. Na prácu s dvojicami toto rozhranie poskytuje metódy **getKey()**, **setKey()** a **setValue()** so samovysvetľujúcimi názvami.

Posledným údajovým rozhraním je **SortedMap**, teda *usporiadaná mapa*. Usporiadanie sa týka kľúčov a je podobne ako pri usporiadanej množine dané buď „prirodzene“, alebo prostredníctvom objektu typu **Comparator**. Nové metódy triedy **SortedMap**: **comparator()** vracia zadaný komparátor, **subMap()** vracia vybranú podmapu danej mapy, **headMap()** vracia časť mapy obsahujúcu kľúče menšie ako zadaný kľúč, **tailMap()** vracia časť mapy obsahujúcu kľúče väčšie alebo rovné ako zadaný kľúč a **firstMap()/lastMap()** vracia najmenší/najväčší kľúč mapy.

Iterátory

Iterátory sú údajové metaštruktúry, ktoré umožňujú k prvkom kolekcie pristupovať opakovane, v jednotlivých iteráciách cyklu. Balík **java.util** deklaruje dve iteračné rozhrania.

Prvé z nich, **Iterator**, sa používa pri iterovanom prístupe ku kolekciám. Obsahuje tieto metódy: **hasNext()** testuje, či možno sprístupniť ďalší prvok, **next()** sprístupní ďalší prvok a **remove()** odstráni prvok, vrátený posledným volaním **next()**, z kolekcie.

Druhé rozhranie, **ListIterator**, je potomkom rozhrania Iterator a poskytuje iterovaný prístup k prvkom zoznamu. Obsahuje niekoľko ďalších metód: **hasPrevious()** testuje, či možno sprístupniť predchádzajúci prvok, **previous()** sprístupní predchádzajúci prvok, **nextIndex()/previousIndex()** vracia index prvku, ktorý by bol sprístupnený nasledujúcim volaním **next()/previous()**, **set()** nahrádza prvok vrátený posledným volaním **next()/previous()** novým prvkom a konečne **add()** pridáva do zoznamu nový prvok, *pred* prvok, ktorý by bol vrátený nasledujúcim volaním **next()** a za prvok, ktorý by bol vrátený nasledujúcim volaním **previous()**.

K prvkom zoznamov tak možno pristupovať buď náhodným spôsobom, pomocou ich indexov, alebo s použitím iterátorov, ktoré implementujú sekvenčný prístup. Príklad oboch spôsobov nájde čitateľ na webovej stránke PC REVUE.

Triedy kolekčného rámca

V balíku **java.util** sa okrem uvedených rozhraní nachádzajú aj ich rozličné implementácie.

Trieda **AbstractCollection** predstavuje skelet pre implementáciu rozhrania **Collection**. Slúži ako bázová trieda dvom ďalším triedam, **AbstractList** a **AbstractSet**. Tieto dve triedy sú analogicky skeletmi pre implementácie rozhraní **List** a **Set**. Do počtu chýba už len trieda **AbstractMap**, ktorá predstavuje skeletálnu implementáciu rozhrania **Map**.

Pozrime sa teraz na potomkov triedy **AbstractList**. Prvým z nich je ďalšia abstraktná trieda **AbstractSequentialList**. Aj táto trieda predstavuje kostru pre implementáciu rozhrania **List**, ale na rozdiel od triedy **AbstractList**, vhodnejšej na náhodný prístup k prvkom prostredníctvom indexu (takéto zoznamy sú obyčajne implementované pomocou poľa), je trieda **AbstractSequentialList** šitá na mieru sekvenčnému prístupu pomocou iterátorov. Triedy od nej odvodené môžu na implementáciu použiť napríklad zreťazený zoznam. Jedným takýmto potomkom je trieda **LinkedList**, ktorá realizuje dvojito zreťazený zoznam. Vďaka niekoľkým pridaným metódam ju možno efektívne použiť na implementáciu zásobníka, frontu i obojstranného frontu. Ide o tieto metódy: **getFirst()/getLast()** vracia prvý/posledný prvok zoznamu, **removeFirst()/removeLast()** odstraňuje a vracia prvý/posledný prvok zoznamu a **addFirst()/addLast()** pridáva nový prvok na začiatok/koniec zoznamu.

Ďalším potomkom triedy **AbstractList** je trieda **ArrayList**. Z jej názvu možno vydedukovať, že ide o spomínanú implementáciu zoznamu pomocou poľa. Každá inštancia triedy **ArrayList** má svoju kapacitu, ktorá sa zadáva ako argument konštruktora. V prípade, že sa celá kapacita poľa zaplní prvkami zoznamu, pole sa automaticky zväčšuje. Okrem známych metód nájdeme v triede **ArrayList** dve nové metódy: **trimToSize()** zmenší kapacitu poľa tak, aby bola rovná aktuálnej dĺžke zoznamu a **ensureCapacity()** explicitne zväčší kapacitu poľa na požadovanú veľkosť.

Nasledujúca trieda **Vector** (ešte stále hovoríme o potomkoch triedy **AbstractList**) je okrem malých detailov v zásade zhodná s triedou **ArrayList**. Dôvodom tejto duplicity je skutočnosť, že trieda **Vector** je súčasťou Javy prakticky „od začiatku“ a až od verzie 1.2 bola upravená tak, aby zapadla do kolekčného rámca. Vďaka tomu obsahuje oproti triede **ArrayList** aj niekoľko vlastných metód: **copyInto()** kopíruje prvky vektora do externého poľa, **setSize()** nastaví požadovanú veľkosť vektora (ak treba, dopĺňa prvky s hodnotou **null**, prípadne odstraňuje nadbytočné prvky), **capacity()** vracia aktuálnu kapacitu vektora, **elements()** vracia prvky vektora ako enumeráciu (pozri ďalej), **elementAt()** vracia prvok na danej pozícii, **firstElement()/lastElement()** vracia prvý/posledný prvok vektora, **setElementAt()** nastavuje hodnotu prvku na danej pozícii, **removeElementAt()** odstraňuje prvok na danej pozícii, **insertElementAt()** vkladá nový prvok do vektora na požadovanú pozíciu, **addElement()** pridáva prvok na koniec vektora, **removeElement()** odoberá požadovaný prvok z vektora a konečne **removeAllElements()** odstraňuje z vektora všetky prvky.

Vsuvka: *enumerácia* je objekt, ktorý dokáže postupne poskytovať objekty z nejakej skupiny. Takýto objekt musí implementovať rozhranie **Enumeration**, ktoré obsahuje dve metódy: **hasMoreElements()**, testujúcu, či je k dispozícii ešte nejaký prvok, a **nextElement()**, vracajúcu nasledujúci prvok skupiny.

Trieda **Vector** má jedného potomka – triedu **Stack**, implementujúcu zásobník, t. j. údajovú štruktúru s **LIFO** stratégiou prístupu. Na zjednodušenie práce táto trieda poskytuje nasledujúce metódy: **empty()** testuje, či je

zásobník prázdny, **peek()** vracia objekt na vrchole zásobníka, **pop()** odstráni objekt z vrcholu zásobníka a vracia ho volajúcej metóde, push() vkladá objekt do zásobníka a **search()** vracia pozíciu daného objektu v zásobníku.

Toľko k zoznamom, pozrime sa na potomkov triedy **AbstractSet**, triedy **HashSet** a **TreeSet**. Prvá z nich, **HashSet**, je implementáciou množiny, založenou na hešovacej tabuľke. V konštruktore triedy možno zadať počiatočnú kapacitu i faktor vyťaženia tabuľky. Druhá trieda **TreeSet** implementuje usporiadanú množinu (t. j. **SortedSet**) pomocou tzv. **Red-Black** stromu.

Aby bol kolekčný rámec úplný, zostáva ešte spomenúť potomkov triedy **AbstractMap**. Tými sú triedy **HashMap**, **TreeMap** a **WeakHashMap**. Prvá z nich predstavuje mapu založenú na hešovacej tabuľke. Podobne ako pri **HashSet** možno zadať počiatočnú kapacitu aj faktor vyťaženia tabuľky. Trieda **TreeMap** realizuje usporiadanú mapu pomocou **Red-Black** stromu. Tretia z tried, **WeakHashMap**, je variantom triedy **HashMap** s tzv. slabými kľúčami – v prípade, že sa niektorý kľúč mapy už nepoužíva (mimo mapy naň neukazuje žiadna premenná), bude spolu so svojou asociovanou položkou z mapy odstránený.

Iné triedy

Medzi triedami realizujúcimi údajové štruktúry nájdeme v balíku **java.util** aj také, ktoré nie sú navrhnuté ako súčasť kolekčného rámca (predovšetkým preto, že sú staršie). Ide o triedy **BitSet**, **Dictionary**, **Hashtable** a **Properties**.

Trieda **BitSet** predstavuje vektor bitov (bitovú množinu), ktorý dokáže podľa potreby meniť svoju veľkosť. Jednotlivé bity (hodnoty typu **boolean**) sú prístupné pomocou celočíselného indexu. Trieda obsahuje tieto metódy: **length()** vracia „logickú“ dĺžku vektora, t. j. index najvyššieho nastaveného bitu **+ 1**, **set()** nastavuje vybraný bit, **clear()** nuluje vybraný bit, **andNot()** vynuluje všetky bity, ktoré sú jednotkové v inej bitovej množine, **get()** vracia hodnotu vybraného bitu, **and()/or()/xor()** vykoná logickú operáciu **AND/OR/XOR** s inou bitovou množinou a **size()** vracia počet bitov v množine.

Ďalšia trieda **Dictionary** je abstraktná a predstavuje údajovú štruktúru schopnú mapovať kľúče na hodnoty. Považuje sa za zastaranú – na odvodzovanie nových tried sa odporúča použiť rozhranie **Map**. Jej potomok, trieda **Hashtable**, bola podobne ako **Vector** upravená tak, aby implementovala rozhranie **Map**. Vzťah medzi **Hashtable** a **HashMap** je zhruba rovnaký ako medzi **Vector** a **ArrayList**.

Trieda **Properties** je odvodená od **Hashtable** a reprezentuje perzistentnú množinu tzv. vlastností (to sú v podstate opäť dvojice kľúč-hodnota, kde hodnoty sú typu **String**). Parametrom konštruktora triedy **Properties** môže byť iná inštancia tejto triedy – ako „implicitný“ zoznam vlastností – ktorá sa prehľadá v prípade neúspešného hľadania kľúča v primárnom zozname. Nové metódy triedy: **setProperty()/getProperty()** nastavuje/vracia hodnotu vybranej vlastnosti, **load()** načíta zoznam vlastností zo vstupného prúdu, **store()** zapíše zoznam vlastností do výstupného prúdu, **propertyNames()** vracia enumeráciu všetkých kľúčov v zozname a **list()** vypíše celý zoznam na zadaný výstupný prúd. Trieda **Properties** sa používa aj pri práci so systémovými vlastnosťami JVM.

Posledné dve triedy balíka **java.util**, ktoré súvisia s údajovými štruktúrami, sú triedy **Arrays** a **Collections**. Obe obsahujú výhradne statické metódy, určené na prácu s poľami a s kolekciami. Trieda **Arrays** poskytuje tieto metódy: **sort()** usporiada pole alebo jeho časť buď vzostupne, na základe „prirodzeného“ usporiadania, alebo podľa objektu typu **Comparator**, **binarySearch()** slúži na binárne vyhľadanie hodnoty v (usporiadanom!) poli, **equals()** porovnáva dve polia, **fill()** vyplní pole alebo jeho časť zadanou hodnotou a **asList()** vráti pole vo forme objektu typu **List**.

V triede **Collections** nájdeme metódy **sort()**, **binarySearch()**, **fill()**, ktoré sú zoznamovým ekvivalentom rovnakých metód triedy **Arrays**, a ďalej tieto metódy: **reverse()** obráti poradie prvkov v zozname, **shuffle()** vyrobí náhodnú permutáciu prvkov zoznamu, **copy()** skopíruje jeden zoznam do druhého, **min()/max()** vracia najmenší/najväčší prvok zoznamu, **unmodifiableXXX()/synchronizedXXX(),** kde **XXX** je jedno zo šiestich kolekčných rozhraní, vracia nemodifikovateľnú/synchronizovanú verziu príslušnej údajovej štruktúry, **singleton()/singletonList()/singletonMap()** vracia množinu/zoznam/mapu obsahujúcu len zadaný objekt, **nCopies()** vracia zoznam pozostávajúci z požadovaného počtu kópií zadaného objektu, **reverseOrder()** vracia komparátor špecifikujúci obrátené usporiadanie k prirodzenému usporiadaniu a nakoniec **enumeration()** vracia enumeráciu nad danou kolekciou.

Dátum a čas

Ďalším okruhom, ktorý balík **java.util** pokrýva, sú triedy na prácu s dátumovými a časovými údajmi.

Trieda **Date** reprezentuje časový bod (určený dátumom a časom) univerzálneho času UTC s presnosťou na milisekundy. Má dva konštruktory – prvý bez argumentov vytvorí objekt reprezentujúci aktuálny systémový čas, druhý akceptuje jeden argument typu **long**, ktorý udáva počet milisekúnd od polnoci 1. 1. 1970 GMT (ďalej „unixový čas“). Metódy triedy: **getTime()** vracia ekvivalentný unixový čas, **setTime()** umožňuje nastaviť objekt Date na základe unixového času, **before()** testuje, či sa časový bod nachádza *pred* iným časovým bodom, **after()** podobne testuje, či sa časový bod nachádza za iným časovým bodom a **compareTo()** porovnáva časový bod s iným časovým bodom. Metóda **toString()**, zdedená z triedy **Object**, konvertuje dátum/čas na reťazec tvaru:

Sun Jul 22 20:07:47 GMT+02:00 2001.

V bežnom živote sa časové údaje vyjadrujú pomocou rokov, mesiacov, dní, hodín, minút a sekúnd. Na konverziu medzi takýmto vyjadrením a objektom typu **Date** slúži abstraktná trieda **Calendar** a jej potomok **GregorianCalendar**, reprezentujúci gregoriánsky kalendárny systém. Opis týchto dvoch tried a práce s nimi by zabral pomaly aj samostatný článok, preto sa uspokojíme s uvedením najdôležitejších metód: **getTime()** vracia ekvivalentný objekt **Date**, **setTime()** nastaví dátum/čas na základe zadaného objektu **Date**, **get()** vracia vybranú zložku dátumu/času, **set()** umožňuje nastaviť vybranú zložku dátumu/času, **before()/after()** sa správajú podobne ako v triede **Date**, **add()** umožňuje posun časového bodu o zadaný interval, **roll()** je podobná metóde **add()**, nemení však hodnotu „vyšších“ zložiek. Na ilustráciu rozdielu medzi **add()** a **roll()**: posunom dátumu 1. 11. 2001 o tri mesiace dopredu pomocou **add()** dostaneme dátum 1. 2. 2002, posunom pomocou **roll()** dátum 1. 2. 2001 (t. j. zmenil sa len mesiac).

S dátumom a časom čiastočne súvisia aj ďalšie triedy: **TimeZone**, **SimpleTimeZone** a **Locale**. Prvé dve z nich slúžia na opis časových zón, vyjadrený odchýlkou od GMT (do úvahy sa berie aj letný čas, DST) – **SimpleTimeZone** je implementáciou abstraktnej triedy **TimeZone**. Tretia trieda **Locale** reprezentuje národné prostredia s kódmi podľa štandardu ISO. Ich bližší opis by bol istotne zaujímavý, ale, bohužiaľ, nemáme naň miesto.

Ostatné triedy

Zo zvyšných tried v balíku **java.util** medzi dôležitejšie patria ešte triedy **Observable**, **Random**, **StringTokenizer**, **Timer** a **TimerTask**.

Trieda **Observable** predstavuje základ pre objekty, ktoré z nejakého dôvodu chceme pozorovať a v okamihu zmeny ich stavu o tom upozorniť iné objekty. „Pozorovacie“ objekty musia implementovať rozhranie **Observer**, obsahujúce jedinú metódu **update()**, ktorá sa vyvolá ako reakcia na zmenu pozorovaného objektu. Medzi metódy triedy Observable patria: **addObserver()/deleteObserver()** pridáva/odoberá pozorovateľov, **notifyObservers()** „obvolá“ všetkých pozorovateľov a oznámi im, že sa stav pozorovaného objektu zmenil, **setChanged()/clearChanged()** nastavuje/maže príznak zmeny objektu, **hasChanged()** indikuje, či sa objekt zmenil, a **countObservers()** vracia počet aktuálne pripojených pozorovateľov.

Inštancie triedy **Random** generujú prúd pseudonáhodných čísel. Na získanie náhodného čísla požadovanej bitovej šírky slúži metóda **next()** a niekoľko jej variantov, ktoré dokážu vrátiť náhodné číslo zadaného typu, prípadne aj so zadaným rozdelením (rovnomerné, normálne).

Trieda **StringTokenizer** sa používa na „rozbitie“ reťazca na postupnosť tokenov. Pri konštruovaní možno zadať oddeľovače, implicitne sa predpokladajú biele znaky. **StringTokenizer** implementuje rozhranie **Enumeration**, nájdeme v nej teda metódy **hasMoreElements()** a **nextElement()** (vracia typ **Object**), ako aj vlastné metódy **hasMoreTokens()**, **nextToken()** (vracia typ **String**) a **countTokens()**. Názvy metód sú samovysvetľujúce.

Posledné dve triedy **Timer** a **TimerTask** slúžia na implementáciu plánovača úloh. Ich činnosť čiastočne súvisí s threadmi, preto ich zatiaľ preskočíme. V budúcnosti sa k nim ešte vrátime.

Výnimky

V balíku **java.util** nájdeme deklaráciu niekoľkých výnimiek. Z tých podstatných vyberáme: **ConcurrentModificationException** indikuje pokus o súčasnú modifikáciu údajovej štruktúry dvoma rôznymi threadmi, **EmptyStackException** sa vyskytne pri snahe vyberať údaje z prázdneho zásobníka, **NoSuchElementException** dostaneme pri pokuse o sprístupnenie neexistujúceho prvku enumerácie.

14.časť: Balíky java.util* a java.net

Opis balíka **java.util** by nebol úplný, keby sme vynechali jeho dva podbalíky **java.util.zip** a **java.util.jar**. Tretím balíkom, ktorému sa v tomto čísle pozrieme na zúbky, je **java.net**.

Balík java.util.zip

Tento balík v zhode so svojím názvom obsahuje triedy pre čítanie a zápis štandardných súborov ZIP a GZIP, triedy pre kompresiu a dekompresiu dát pomocou algoritmu *DEFLATE* (použitého v súboroch ZIP a GZIP) a nakoniec triedy pre výpočet kontrolného súčtu podľa algoritmov *CRC-32* a *Adler-32* (špecifikácie formátov a algoritmov možno nájsť v dokumentoch RFC 1950-1952).

Začneme od konca. Triedy **Adler32** a **CRC32** slúžia na výpočet kontrolného súčtu postupnosti bajtov. Práca s nimi spočíva v opakovanom volaní metódy **update()**, ktorej argumentom je bajt alebo pole bajtov. Po odoslaní všetkých bajtov postupnosti získame kontrolný súčet metódou **getValue()**. Tretia metóda **reset()** kontrolný súčet „reštartuje“ na úvodnú hodnotu a dovoľuje tak opätovne použiť existujúcu inštanciu triedy. Schéma použitia oboch tried vyzerá približne takto:

```
CRC32 crc = new CRC32();
while ( ... )
{
  byte b = ... ;
  crc.update(b);
}
long chksum = crc.getValue();
```

Triedy implementujú rozhranie **Checksum**, ktoré obsahuje všetky uvedené metódy.

Dve ďalšie triedy **CheckedInputStream** a **CheckedOutputStream** predstavujú vstupný, resp. výstupný prúd s pridanou schopnosťou počítať kontrolný súčet dát,

ktoré ním prechádzajú. Obe triedy sú odvodené od „filtračných“ tried **FilterInputStream** a **FilterOutputStream** (pozri predminulú časť seriálu) a navyše obsahujú metódu **getChecksum()**, ktorá vracia aktuálnu hodnotu kontrolného súčtu.

Kompresiu dát použitím populárnej knižnice *ZLIB* zabezpečuje trieda **Deflater**. Metódami **setDictionary()**, **setStrategy()** a **setLevel()** môžeme nastaviť počiatočný slovník, komprimačnú stratégiu a úroveň kompresie (trieda **Deflater** obsahuje niekoľko preddefinovaných úrovní vo forme statických konštánt). Údaje, ktoré chceme komprimovať, posielame triede pomocou metódy **setInput()**, skomprimované údaje získame volaním metódy **deflate()**. Pomocou metódy **needsInput()** zistíme, či treba dodať ďalšie vstupné údaje, metódou **finish()** oznámime triede, že už stačilo, ďalšie údaje jej nedodáme, a aby kompresiu vhodne ukončila. Tento stav indikuje metóda **finished()** vrátením hodnoty true. Vďaka metódam **getTotalIn()**, **getTotalOut()** a **getAdler()** máme k dispozícii celkový počet bajtov, ktoré „pretiekli“ triedou, a ich kontrolný súčet (*Adler-32*). Konečne metódy **reset()** a **end()** slúžia na reinicializáciu, resp. ukončenie práce s triedou **Deflater**.

Opačnú operáciu, dekompresiu dát, má na starosti trieda **Inflater**. Aj tu vstupné údaje zadávame pomocou metódy **setInput()**, ich dekomprimovanú podobu získame volaním metódy **inflate()**. Na zadanie počiatočného slovníka použijeme metódu **setDictionary()** (či je to nevyhnutné, zistíme metódou **needsDictionary()**). Metódy **needsInput()**, **finished()**, *getTotalIn()*, **getTotalOut()**, **getAdler()**, **reset()** a **end()** majú rovnaký význam ako pri predošlej triede. Či vo vstupnom bufferi po dekompresii zostali nejaké nespracované dáta, zistíme pomocou metódy **getRemaining().**

Triedu **Deflater** použijeme obyčajne takýmto spôsobom:

```
Deflater dft = new Deflater();
byte[] data = new byte[...];
byte[] cdata = new byte[...];
...
dft.setInput(data);
dft.finish();
dft.deflate(cdata);
```

a triedu Inflater zase takto:

```
Inflater ift = new Inflater();
ift.setInput(cdata);
ift.inflate(data);
```

Triedy **Deflater** a **Inflater** sú vhodne skombinované so vstupnými/výstupnými prúdmi v triedach **DeflaterOutputStream** a **InflaterInputStream** (opäť sú to „filtračné“ triedy). Prvá z oboch tried disponuje schopnosťou priebežne komprimovať údaje zapisované pomocou štandardnej metódy **write()**, druhá, naopak, dokáže dekomprimovať údaje čítané pomocou metódy **read()**. Na detekciu konca vstupného prúdu InflaterInputStream použijeme metódu **available()**.

Štvorica tried **ZipEntry**, **ZipFile**, **ZipOutputStream** a **ZipInputStream** značným spôsobom uľahčuje prácu so súbormi ZIP. **ZipEntry** reprezentuje jeden položku súboru ZIP (t. j. jeden zo súborov v archíve). Každá položka má svoje meno, ktoré sa zadáva ako argument konštruktora a zistiť ho možno pomocou metódy **getName()**. Ďalšie metódy triedy **ZipEntry** slúžia na nastavenie/sprístupnenie rôznych atribútov položky: **setTime()**, **setSize()**, **setCompressedSize()**, **setCrc()**, **setMethod()**, **setExtra()**, **setComment()** a ich **get****()** ekvivalenty – názvy metód sú dostatočne opisné. Pomocou metódy **isDirectory()** zistíme, či je daná položka adresárom (názvy adresárov sa končia znakom /).

Ďalšia trieda **ZipFile** slúži na čítanie položiek zo súboru ZIP. Argumentom jej konštruktora je reťazec s cestou k súboru alebo objekt triedy **File**. Cestu k súboru môžeme spolu s menom zistiť dodatočne volaním metódy **getName()**. Metóda **size()** vracia počet položiek v súbore, metóda **entries()** vracia položky ako enumeráciu (typ **Enumeration**). Pomocou **getEntry()** sa dostaneme k vybranej položke a pomocou **getInputStream()** získame vstupný prúd, ktorý možno použiť na čítanie údajov z položky. Metóda **close()** otvorený súbor korektne uzavrie.

Trieda **ZipOutputStream** je potomkom **DeflatedOutputStream** a predstavuje výstupný prúdový filter na zápis do súborov ZIP. Komprimačnú metódu, úroveň kompresie a prípadný komentár môžeme špecifikovať pomocou metód **setMethod()**, **setLevel()** a **setComment()**. Novú položku v súbore ZIP začneme metódou **putNextEntry()**. Zápis sa realizuje klasickou metódou write(), položku uzavrieme volaním metódy **closeEntry()**. Súbor (a prúd) uzavrieme metódou **close()**.

Komplementárna trieda **ZipInputStream** slúži ako vstupný filter na čítanie zo súborov ZIP. Je potomkom triedy InflaterInputStream a pracuje sa s ňou podobne ako s predchádzajúcou triedou: nasledujúcu položku „začneme“ volaním **getNextEntry()**, čítanie obsahu položky obstará metóda **read()** a položku nakoniec uzavrieme metódou **closeEntry()**. Metóda **createZipEntry()** slúži na vytvorenie objektu **ZipEntry** pre položku so zadaným menom.

Posledné dve triedy, **GZIPOutputStream** a **GZIPInputStream**, predstavujú prúdové filtre pre zápis a čítanie do/z GZIP súborov. Tie nemajú žiadnu vnútornú štruktúru (preto sa v UNIX-like operačných systémoch musí použiť najprv TAR archivátor), takže sú predefinované len metódy **read()** a **write()** (triedy sú odvodené od **DeflaterOutputStream** a **InflaterInputStream**).

Balík **java.util.zip** obsahuje aj dve triedy výnimiek: **DataFormatException**, ktorá signalizuje chybu formátu údajov pri ich rozbaľovaní, a **ZipException**, ktorá indikuje chybu pri práci so súborom ZIP.

Balík java.util.jar

Druhý podbalík poskytuje triedy pre prácu so súbormi JAR (= **J**ava **AR**chive), ktoré sú založené na štandardnom formáte ZIP a navyše voliteľne obsahujú tzv. manifestačný súbor (manifest file, ďalej len „manifest“). Tento súbor obsahuje metainformácie o obsahu súboru JAR vo forme položiek a k nim priradených množín atribútov (pod atribútom treba rozumieť dvojicu kľúč-hodnota).

Upozornenie: nasledujúce riadky budú čitateľovi omnoho jasnejšie po preštudovaní dokumentácie k JDK, kde sa nachádza opis súborov JAR i manifestu.

Prvou z tried balíka **java.util.jar** je **Attributes**, ktorá funguje podobne ako mapovacie triedy balíka **java.util** – mapuje názvy atribútov na reťazcové hodnoty pridružené k nim (trieda **Attributes** v skutočnosti implementuje rozhranie **java.util.Map**). Hodnotu vybraného atribútu zistíme pomocou metódy **getValue()**. Meno atribútu môžeme zadať ako reťazec alebo ako objekt typu **Attributes.Name**, čo je pomocná trieda na reprezentáciu názvov atribútov s množstvom preddefinovaných statických konštánt (bližšie pozri dokumentáciu k JDK). Opačnú operáciu – nastavenie hodnoty atribútu – realizuje metóda **putValue()**. Ostatné metódy boli opísané v predchádzajúcej časti.

Samotný manifest reprezentuje trieda **Manifest**. Pomocou metódy **getMainAttributes()** získame tzv. hlavné atribúty manifestu (týkajú sa celého manifestu) a pomocou **getAttributes()** sa dostaneme k atribútom jednotlivých položiek. Zoznam položiek (objekt typu **Map**) získame volaním metódy **getEntries()**. Metóda **clear()** zruší celý obsah manifestu, metódy **read()** a **write()** slúžia na načítanie/zápis obsahu manifestu z/do zadaného prúdu.

Nasledujúce štyri triedy – **JarEntry**, **JarFile**, **JarOutputStream** a **JarInputStream** – sú významom prakticky totožné so svojimi náprotivkami v balíku **java.util.zip** (ba čo viac, sú ich potomkami). Trieda **JarEntry** reprezentuje položku súboru JAR (pozor, nie položku manifestu!) a obsahuje navyše metódu **getAttributes()**, ktorá vracia atribúty danej položky, a **getCertificates()** – tá vracia digitálne certifikáty položky.

Trieda **JarFile** slúži na čítanie obsahu súboru JAR. Je možné prikázať, aby sa pri otváraní súboru skontroloval jeho digitálny podpis. Medzi pridanými metódami nájdeme **getManifest()** na prístup k manifestu súboru JAR a **getJarEntry()** na sprístupnenie zadanej položky súboru.

Triedy **JarOutputStream** a **JarInputStream** celkom logicky slúžia ako prúdové filtre na zápis/čítanie súborov JAR. V prvej triede, **JarOutputStream**, nenájdeme nič nové, len predefinovanú metódu **putNextEntry()**. Druhá trieda, **JarInputStream**, obsahuje dve nové metódy: **getManifest()** a **getNextJarEntry()**. Ich význam je zrejmý.

V balíku **java.util.jar** nájdeme jednu výnimku, **JarException**, odvodenú od **ZipException** a indikujúcu problémy pri čítaní alebo zápise súboru JAR.

Balík java.net

Z názvu tohto balíka je na prvý pohľad jasné, že bude obsahovať triedy pre podporu sieťových aplikácií. Komunikácia sa realizuje prostredníctvom soketov – Java mlčky predpokladá prítomnosť TCP/IP (čo je však úplne logické, majorita súčasných sietí používa protokoly rodiny TCP/IP).

Na reprezentáciu adries IP (protokolu IPv4) je k dispozícii trieda **InetAddress**. Čo je zvláštne, táto trieda nemá nijaký konštruktor (teda aspoň nie verejne prístupný), na vytvorenie objektu typu **InetAddress** treba použiť jednu zo statických metód **getLocalHost()**, **getByName()**, **getAllByName().** Prvá z metód vracia objekt reprezentujúci lokálnu adresu IP, druhá a tretia vracajú objekt, resp. pole objektov predstavujúcich jednu adresu IP alebo množinu všetkých možných adries IP počítača špecifikovaného pomocou reťazca s doménovým menom alebo s adresou IP v bodkovom zápise (ako napr. "**195.168.1.4**").

Medzi ďalšie metódy triedy **InetAddress** patria: **isMulticastAddress()** – slúži na rozpoznanie multicast adries (to sú adresy triedy D, začínajúce sa bitmi 1110), **getHostName()** – vracia doménové meno patriace k danej adrese IP, **getHostAddress()** – vracia adresu IP ako reťazec v bodkovom zápise a **getAddress()** – vracia adresu IP ako pole štyroch bajtov.

Význam triedy URL je zrejmý. Na zopakovanie: URL (= Uniform Resource Locator) predstavuje spôsob, ako na internete možno odkazovať na rôzne prostriedky (ako napr. stránky, súbory, mailové schránky). URL pozostáva zo špecifikácie protokolu, názvu cieľového počítača, voliteľného čísla portu, na ktorom prebieha komunikácia, a cesty, ktorá jednoznačne identifikuje prostriedok. Trieda URL obsahuje viacero konštruktorov, novú inštanciu možno zostrojiť napríklad takto:

```
URL url = new URL("http", "www.niekde.sk", 80,
"/index.htm");
```

ale aj takto:

```
URL url = new
URL("http://www.niekde.sk:8080/index.htm");
```

Metódy **getQuery()**, **getPath()**, **getUserInfo()**, **getAuthority()**, **getPort()**, **getProtocol()**, **getHost()**, **getFile()** a **getRef()** vracajú rozličné zložky URL (pod *query* sa rozumie časť cesty za prípadným znakom **?**, authority je zloženina *host:port*, resp. *user@host:port*, a *ref* je časť cesty za prípadným znakom **#**). Metódou **sameFile()** môžeme zisťovať zhodu dvoch URL s vylúčením zložky ref. Pomocou metódy **toExternalForm()** získame URL v tvare súvislého reťazca. Dôležitá metóda **openConnection()** vráti objekt **URLConnection** (pozri ďalej), ktorý reprezentuje sieťové spojenie so zadaným URL. Podobne metóda **openStream()** vytvorí spojenie s URL a vráti vstupný prúd na čítanie z tohto spojenia. Pomocou metódy **getContent()** získame obsah URL vo forme javovského objektu.

Komunikačné spojenie medzi javovskou aplikáciou a URL zabezpečuje abstraktná trieda **URLConnection**. Konštruktor tejto triedy je chránený, nové objekty získame volaním metódy **openConnection()** nad objektom typu URL. Trieda obsahuje množstvo metód, z tých významnejších vyberáme: **connect()** – fyzicky zrealizuje požadované spojenie (teda prepojí obe komunikujúce strany), **getContentLength(), getContentType(), getContentEncoding(), getExpiration(), getDate()** a **getLastModified()** – vracajú rovnako pomenované polia hlavičky, nachádzajúcej sa na začiatku odpovede vzdialeného počítača (spomeňme si hoci na polia štandardnej hlavičky HTTP), veľmi dôležitá metóda **getContent()** – vracia samotný dátový obsah URL, **getInputStream()/getOutputStream()** – vracajú prúdy na čítanie alebo zápis z/do spojenia, a mnoho ďalších.

Java poskytuje dve inkarnácie triedy **URLConnection** – pre HTTP spojenia je to **HttpURLConnection** a pre spojenia k archívom JAR zase **JarURLConnection**. Obe triedy sú abstraktné, takže skutočná implementácia spojení je záležitosťou konkrétneho JVM. V triede **HttpURLConnection** nájdeme množstvo statických finálnych členov, reprezentujúcich stavové kódy protokolu HTTP a niekoľko špecifických metód: **setFollowRedirects()** – zapína či vypína automatické presmerovanie (pri vrátení kódu **3xx**), **setRequestMethod()** – nastavuje prístupovú metódu (GET, POST a pod.), **getResponseCode(), getResponseMessage()** – vracajú návratový kód, resp. návratovú správu HTTP protokolu, ako aj niekoľko ďalších, menej významných metód.

Druhá trieda **JarURLConnection** sa dostáva k slovu pri prístupe k súborom JAR – URL má vtedy formát *jar:http://www.niekde.sk/archiv.jar!/sk/niekde/foo/bar.class* (ako vidieť, za výkričníkom sa nachádza cesta ku konkrétnej triede – položke súboru JAR). Z metód triedy spomenieme **getEntryName()** – vracia názov sprístupňovanej položky, **getJarFile(), getJarEntry(), getManifest(), getAttributes()** – vracajú objekty typu **JarFile**, **JarEntry**, **Manifest**, **Attributes** (pozri balík **java.util.jar**).

Príklad, ako vypísať HTML obsah vybranej webovej stránky (obsah, pochopiteľne, nie je nijako interpretovaný):

```
URL url = new URL("http://www.pcrevue.sk/");
URLConnection uc = url.openConnection();
uc.connect();
BufferedReader br =
   new BufferedReader(
   new InputStreamReader(uc.getInputStream()));

while (true)
{
  String line = br.readLine();
  if (line == null)
    break;
  System.out.println(line);
}
```

Všimnite si, že sa vôbec nestaráme o to, akého typu je objekt **uc** v skutočnosti.

Okrem komunikácie so zadaným URL možno v Jave vytvoriť spojenie priamo medzi dvoma soketmi (soket je koncový komunikačný bod, identifikovaný adresou IP a číslom portu). Trieda **Socket** reprezentuje klientsky soket pre prúdovú komunikáciu pomocou TCP. Pri jej konštrukcii môžeme zadať cieľovú adresu a port (soket sa potom automaticky spojí so svojím náprotivkom) a aj lokálnu adresu a port (to keď chceme „vysielať“ z konkrétneho portu). Metódami **getInetAddress(), getLocalAddress(), getPort()** a **getLocalPort()** zistíme vlastnosti soketu, pomocou **getInputStream()** a **getOutputStream()** získame prúdy na čítanie a zápis z/do soketu. Metóda **close()** ukončí komunikáciu a soket uzavrie. Niekoľko ďalších metód slúži na nastavenie parametrov komunikácie. Zaujímavé je, že samotnú komunikáciu realizuje objekt typu **SocketImpl** (abstraktná trieda). Pokiaľ si však nechceme napísať vlastnú implementáciu soketov, nie je táto trieda zaujímavá.

Ak programujeme serverovú časť aplikácie, použijeme triedu **ServerSocket**. Argumentom jej konštruktora je číslo portu, na ktorom bude soket počúvať. Najdôležitejšou metódou je **accept()**, ktorej volanie je blokujúce – metóda čaká, až sa na soket pripojí klient „zvonka“. V tom okamihu vracia objekt typu **Socket**, pomocou ktorého možno s klientom komunikovať štandardným spôsobom. Z ďalších metód medzi tie zaujímavejšie patrí **getInetAddress(), getLocalPort()** a **close()** s významom rovnakým ako v triede **Socket**.

Ako vyzerá kostra serverovej aplikácie, možno lepšie pochopiť z nasledujúceho príkladu:

```
ServerSocket ss = new ServerSocket(12345);
while (true)
{
  Socket cs = ss.accept();
  BufferedReader br =
     new BufferedReader(
     new InputStreamReader(cs.getInputStream()));
  while (true)
  {
    String ln = br.readLine();
    if (ln == null)
      break;
    System.out.println(ln);
  }
  cs.close();
}
```

(Kompletnejšiu verziu nájde čitateľ spolu s ostatnými príkladmi na webe PC REVUE.)

V príklade vytvoríme serverový soket **ss**, počúvajúci na porte č. 12345. V nekonečnom cykle pomocou metódy **accept()** získame klientsky soket **cs**, z ktorého čítame riadky textu a vypisujeme na konzolu. Na otestovanie je potrebné pripojiť sa k počítaču, na ktorom program beží, napríklad pomocou telnetu (ale na porte 12345!). Posielané riadky sa budú postupne objavovať na konzole. Program sa ukončí stlačením Ctrl+C.

Na datagramový prenos pomocou protokolu UDP (bez spojenia, nezabezpečený) slúži trieda **DatagramSocket**. Pri jej konštruovaní môžeme zadať číslo lokálneho portu soketu. Pripojenie k zadanej vzdialenej adrese (pozor, nie vytvorenie sieťového spojenia ako pri TCP) má na starosti metóda **connect()**, odpojenie metóda **disconnect()**. Na príjem a odosielanie údajov slúžia metódy **send()** a **receive()**, soket uzavrieme volaním **close()**. Ostatné metódy sú podobné ako v triede **Socket**, implementáciu soketov zabezpečuje trieda **DatagramSocketImpl**.

Keďže v prípade UDP nejde o prúdový prenos, nepracujeme s prúdmi, ale s objektom triedy **DatagramPacket** (ten je ostatne argumentom oboch metód – **send()** aj **receive()**). Pri jeho konštruovaní zadávame pole bajtov, ktoré bude fungovať ako prijímací alebo vysielací buffer, a pri odosielaných datagramoch aj cieľovú adresu. Pomocou metód triedy **DatagramPacket** môžeme nastavovať a zistiť zdrojovú, resp. cieľovú adresu a port, dĺžku údajov a samotné údaje.

Na záver spomeňme ešte výnimky: **MalformedURLException** (chyba v syntaxi URL), **ProtocolException** (chyba na úrovni prenosového protokolu), **SocketException** s potomkami (chyba pri práci so soketmi), **UnknownHostException** (chyba pri preklade pomocou DNS) a **UnknownServiceException** (pokus o prácu s neznámym MIME typom).

Žiaľ, na viac z balíka **java.net** nemáme miesto. Zvyšné triedy slúžia napríklad na implementáciu multicast soketov (**MulticastSocket**), zabezpečenie prístupových práv (**SocketPermission, NetPermission, Authenticator**), prácu s URL a interpretáciu obsahu (**ContentHandler**, **URLStreamHandler**) či konverziu medzi typom **String** a MIME typom *x-www-form-urlencoded* (**URLEncoder**, **URLDecoder**). Záujemca si viac informácií isto nájde priamo v dokumentácii.

15.časť: Balík java.awt

AWT je skratka názvu *Abstract Windowing Toolkit*. Súbor tried združených v balíku **java.awt** a v jeho podbalíkoch je určený na tvorbu používateľského rozhrania a prácu s grafikou. Dnes sa skôr dáva prednosť komponentom *Swing*, ktoré sú univerzálnejšie, flexibilnejšie a poskytujú väčšie možnosti. Napriek tomu nie je AWT úplne odsúdený na zánik; pre jednoduchšie aplikácie postačí (a práca s ním je v určitých ohľadoch o niečo jednoduchšia). Okrem toho *Swing* stavia na základoch vybudovaných pomocou AWT.

Vzhľadom na rozsiahlosť celého balíka nebude jeho opis taký vyčerpávajúci ako v predchádzajúcich častiach. Niekoľko väčších príkladov nájde čitateľ ako vždy na internetových stránkach PC REVUE.

Hierarchia komponentov

Jadrom balíka **java.awt** je hierarchia komponentov používateľského rozhrania (ďalej len UI). Používa sa tradičný okienkový systém, niektoré komponenty môžu v sebe obsahovať ďalšie komponenty. Interakcia používateľa s komponentmi sa transformuje na *udalosti* (ako stlačenie klávesu, kliknutie myši a podobne). Komponenty dokážu na výskyt udalostí vhodne reagovať, štandardným spôsobom alebo na základe správania špecifikovaného programátorom.

Trieda Component

Koreňom hierarchie je abstraktná trieda **Component**, predstavujúca univerzálny objekt, ktorý má nejakú grafickú reprezentáciu, môže byť zobrazený na obrazovke a dokáže interagovať s používateľom. Komponenty UI sú obyčajne implementované pomocou natívnych komponentov hostiteľskej platformy, ale v prípade potreby môže programátor odvodiť vlastných potomkov triedy **Component** a vytvoriť tak tzv. *lightweight* komponenty, ktoré nie sú implementované natívnymi objektmi – je očividné, že potom treba naprogramovať úplne všetko – od zobrazovania komponentu v jeho rôznych stavoch po definovanie spôsobu interakcie s používateľom.

V triede **Component** nájdeme obrovské množstvo metód. Nie je možné ich tu všetky vymenovať, tak sa pozrieme aspoň na tie najdôležitejšie (zápis **(get|set)*()** reprezentuje dvojicu metód **get*()** a **set*()** na získanie a nastavenie hodnoty nejakého atribútu): **(get|set)Name()** – názov komponentu, **getParent()** vracia rodičovský komponent, **getToolkit()** vracia objekt triedy **Toolkit** (pozri ďalej), **isVisible()/isEnabled()** zisťuje, či je komponent viditeľný a prístupný, **setVisible()/setEnabled()** nastavuje viditeľnosť a prístupnosť komponentu, **(get|set)Foreground()** a **(get|set)Background()** – farba popredia, resp. pozadia komponentu, **(get|set)Font()** – font použitý v komponente, **(get|set)Location()** – poloha komponentu, **(get|set)Size()** – rozmery komponentu, **getX()/getY()** – súradnice ľavého horného rohu komponentu, **getWidth()/getHeight()** – šírka/výška komponentu, **getPreferredSize()/getMinimumSize()/getMaximumSize()** – predpokladaná (požadovaná), minimálna a maximálna veľkosť komponentu, **getGraphics()** vracia objekt typu **Graphics** (pozri ďalej), **(get|set)Cursor()** – typ kurzora priradený komponentu, **paint()** vykreslí komponent, **update()** obnoví obsah komponentu (zmaže pozadie a zavolá metódu **paint()**), **repaint()** spôsobí, že sa čo najskôr zavolá metóda **update()** (možno zadať, ktorá časť komponentu sa má prekresliť a za aký čas), **contains()** zisťuje, či komponent obsahuje zadaný bod, **getComponentAt()** zisťuje, či samotný komponent alebo niektorý z jeho podkomponentov obsahuje zadaný bod.

Nasledujúca séria metód pripojí ku komponentu, resp. odoberie od neho špeciálny objekt, tzv. *listener*, ktorý bude dostávať správy o udalostiach týkajúcich sa komponentu. Metódy majú tvar **(add|remove)*Listener()**, kde

namiesto hviezdičky môže byť niektoré zo slov **Component** (udalosti týkajúce sa komponentu), **Focus** (získanie alebo strata fokusu), **Hierarchy** a **HierarchyBounds** (zmena hierarchie komponentov), **Key** a **InputMethod** (udalosti klávesnice), **Mouse** a **MouseMotion** (udalosti myši) a **PropertyChange** (zmena fontu, farby popredia alebo farby pozadia). Metóda **getListeners()** vracia pole všetkých listenerov pripojených ku komponentu.

Zo zvyšných metód medzi významnejšie patria ešte: **requestFocus()** žiada o presun vstupného fokusu na komponent, **transferFocus()** presúva fokus na ďalší komponent, **hasFocus()** zisťuje, či má komponent fokus, **add()/remove()** pridáva/odoberá kontextové (popup) menu ku komponentu, **list()** „vypíše“ obsah komponentu do zadaného výstupného prúdu (vhodné na ladenie).

Kontajnerové komponenty

Dôležitým potomkom triedy **Component** je trieda **Container**. Inštancie triedy **Container** dokážu fungovať ako kontajnery – môžu v sebe obsahovať iné komponenty. (Máme na mysli vizuálne „obsiahnutie“ – okno napríklad *obsahuje* ovládacie prvky, ako tlačidlá, textové polia a podobne.) Komponenty sú v kontajneri uložené v určitom poradí, ktoré určuje ich prípadné vzájomné prekrývanie. Usporiadanie komponentov v kontajneri zabezpečuje layout manager, čo je objekt implementujúci rozhranie **LayoutManager** (viac o layoutoch pozri ďalej).

Z nových metód triedy **Container** k významnejším patria: **getComponent()|getComponents()** sprístupňuje vybraný komponent (alebo všetky vo forme poľa), **getInsets()** vracia objekt Insets, definujúci veľkosť okrajov kontajnera, **add()|remove()|removeAll()** pridáva, resp. odstraňuje komponenty do/z kontajnera, **(get|set)Layout()** zisťuje/nastavuje layout manager pre kontajner, **validate()** spôsobí, že sa komponenty kontajnera znovu rozmiestnia podľa aktuálneho layoutu (vhodné napríklad po pridaní nových komponentov), **(add|remove)ContainerListener()** pripája/odoberá kontajnerový listener, **findComponentAt()** je podobná metóde **getComponentAt()**, ale na rozdiel od nej prehľadáva všetky dcérske komponenty.

Trieda **Container** nie je implementovaná natívnym komponentom, takže sa v praxi stretneme skôr s jej potomkami. Najjednoduchším je **Panel**, komponentový kontajner bez akejkoľvek pridanej funkčnosti. Trieda Panel sa hodí napríklad v prípade, že chceme v jednom kontajneri použiť rôzne layouty. Ďalší potomok, **ScrollPane**, predstavuje panel s pridanou možnosťou rolovania obsahu pomocou posuvníkov. **ScrollPane** smie obsahovať len jediný dcérsky komponent, ktorému poskytuje schopnosť rolovania. Z metód za zmienku stojí len dvojica **(get|set)ScrollPosition()**, pomocou ktorej možno pracovať s aktuálnou pozíciou dcérskeho komponentu v rámci panela.

Dôležitým kontajnerom je trieda **Window**, predstavujúca okno bez okrajov a bez menu. Ak chceme vytvoriť objekt triedy **Window**, musíme zadať jeho rodičovské okno (iný objekt typu **Window** alebo niektorého z odvodených typov **Frame** a **Dialog**). Podstatné metódy triedy **Window**: **pack()** zmení veľkosť okna tak, aby sa doň práve vošli všetky vnorené komponenty, **show()|hide()** zobrazuje/skrýva okno, **dispose()** okno zruší, **toFront()|toBack()** presúva okno do popredia/do pozadia, **getOwner()** vracia vlastníka okna, **getOwnedWindows()** vracia vlastnené okná, **(add|remove)WindowListener()** pridáva/odoberá „okenný“ listener.

Praktickejšími potomkami triedy **Window** sú dve už spomínané triedy **Frame** a **Dialog**. Trieda **Frame** predstavuje okno najvyššej úrovne s titulkom a okrajom. Väčšina nových metód umožňuje čítať a meniť atribúty okna: **(get|set)Title()** – titulok okna, **(get|set)IconImage()** – ikona okna, **(get|set)MenuBar()** – hlavné menu okna, **(get|set)State()** – stav okna (či je minimalizované). Statická metóda **getFrames()** vracia pole všetkých objektov typu **Frame**, ktoré práve bežiaca aplikácia dosiaľ vytvorila.

Druhá trieda **Dialog** reprezentuje dialógové okná, ktorým pri konštrukcii treba určiť vlastníka. Dialógové okná môžu byť modálne (kým sú zobrazené, nemožno pracovať s inými top-level oknami aplikácie) alebo nemodálne. Z metód spomenieme **(is|set)Modal()** na zistenie alebo nastavenie modality dialógu a **(is|set)Resizable()** na zistenie alebo nastavenie schopnosti meniť veľkosť dialógu. Špeciálnym potomkom triedy **Dialog** je trieda **FileDialog**, predstavujúca dialóg na výber názvu súboru. Táto trieda poskytuje metódy **(get|set)Directory**, **(get|set)File()**, **(get|set)FilenameFilter()** na zistenie či nastavenie názvu a cesty vybraného súboru, ako aj metódy **(get|set)Mode()** na špecifikovanie, či je vybraný súbor určený na čítanie alebo zápis.

Ovládacie prvky

Zostávajúci potomkovia triedy **Component** predstavujú klasické ovládacie prvky. Triedu **Label** použijeme na zobrazenie needitovateľného textu. Pri jeho konštrukcii môžeme zadať, či má byť zarovnaný na ľavý okraj, na stred alebo na pravý okraj. S textom pracujeme pomocou metód **(get|set)Text()**, so zarovnaním pomocou **(get|set)Alignment()**.

Trieda **Button** reprezentuje tlačidlo, ktoré obsahuje voliteľný text, a ktoré možno stlačiť pomocou myši alebo klávesnice. S textom na tlačidle pracujeme pomocou metód **(get|set)Label()** a ohlas na stlačenie tlačidla zabezpečí príslušný listener (môže ich byť aj viac), ktorý pridávame a odoberáme metódami **(add|remove)ActionListener()**.

Trieda **Checkbox** predstavuje dvojstavové zaškrtávacie pole s voliteľným textom. Tento text sprístupňuje dvojica metód **(get|set)Label()**, stav poľa zase dvojica metód **(get|set)State()**. Zmenu stavu poľa môže registrovať listener, ktorý pridávame a odoberáme metódami **(add|remove)ItemListener()**. Objekty **Checkbox** môžu byť združované do skupín, reprezentovaných objektmi triedy **CheckboxGroup** (pozor, nie je potomkom triedy **Component**!) – vtedy sa z nich stávajú prepínače (radio buttons). Pomocou metód **(get|set)SelectedCheckbox()** máme prístup k aktuálne zvolenému prepínaču.

Rozbaľovací (drop-down) zoznam reťazcových položiek reprezentuje trieda **Choice**. Položky sa do zoznamu pridávajú metódami **add()** a **insert()**, odoberajú sa pomocou **remove()**. Položku s daným indexom môžeme získať metódou **getItem()**. Index aktuálne vybranej položky vracia metóda **getSelectedIndex()**, vybranú položku ako reťazec získame pomocou **getSelectedItem()**. Metódou **select()** môžeme nastaviť, ktorá položka bude vybraná, a dvojica metód **(add|remove)ItemListener()** slúži na pridanie/odobranie listenerov.

Klasický (teda nerozbaľovací) zoznam vytvoríme pomocou triedy **List**. Pri konštrukcii zoznamu môžeme zadať počet zobrazených riadkov (implicitne 4), ako aj to, či je možné vybrať viac ako jednu položku (multiselect list). Metódy triedy **List** sú podobné metódam triedy **Choice**. Z nových metód spomeňme **replaceItem()**, pomocou ktorej môžeme zmeniť vybranú položku, **getSelectedIndexes()** a **getSelectedItems()**, ktoré vracajú indexy vybraných položiek alebo priamo položky vo forme poľa (ináč to pri multiselect zoznamoch ani nejde), **deselect()** na zrušenie výberu zadanej položky, **isIndexSelected()** na zistenie, či je položka s daným indexom vybraná.

Trieda **Scrollbar** predstavuje vodorovný alebo zvislý posuvník (nemýliť si s triedou **ScrollPane**, na ktorú sa v zásade môžeme pozerať ako na kombináciu objektu **Panel** a dvoch objektov typu **Scrollbar**). Pri konštrukcii posuvníka alebo neskôr pomocou samostatných metód môžeme zadať orientáciu, povolený rozsah hodnôt, ktoré bude ťahadlo posuvníka reprezentovať, a veľkosť prírastkov pri kliknutí na šípku alebo na plochu posuvníka. Metód je priveľa na to, aby sme ich všetky vymenovali (ide vždy o get/set dvojice), takže za všetky spomenieme iba metódu **getValue()**, ktorá vracia aktuálnu hodnotu, ktorú posuvník (resp. jeho ťahadlo) reprezentuje.

Ďalšia trieda **TextComponent** je spoločným predkom komponentov určených na editáciu textu. S textom ako celkom možno pracovať pomocou metód **(get|set)Text()**, metóda **getSelectedText()** vracia aktuálne vybranú (obyčajne inverzne zvýraznenú) časť textu. Ak chceme programovo vyznačiť úsek textu, použijeme metódu **select()**. Volaním **setEditable()** sa dá komponent „zamknúť“, metódy **(set|get)CaretPosition()** nastavujú/zisťujú polohu textového kurzora. K textovému komponentu môžeme pomocou **(add|remove)TextListener()** pripojiť/odobrať príslušný listener.

Od triedy **TextComponent** sú odvodené dve ďalšie triedy: **TextField** a **TextArea**. Prvá z nich predstavuje klasické jednoriadkové textové pole. Pri jeho konštrukcii môžeme zadať počiatočný obsah a šírku poľa v znakoch. Z užitočných metód pribudla predovšetkým dvojica **(get|set)EchoChar()**, umožňujúca nastaviť/zistiť znak, ktorý sa bude zobrazovať namiesto jednotlivých písmen textu (typicky to býva hviezdička pri textových poliach, do ktorých sa zadáva heslo). Druhá trieda, **TextArea**, reprezentuje viacriadkové textové editačné pole s možnosťou automatického zobrazovania posuvníkov, ak je obsah dlhší ako rozmery poľa. Pri konštrukcii môžeme zadať rozmery poľa (v riadkoch a znakoch). Pomocou metód **insert()** a **append()** môžeme text do poľa pridávať, metóda **replaceRange()** poskytuje možnosť nahradiť vybraný úsek textu iným reťazcom.

Poslednou komponentovou triedou je **Canvas**. Táto trieda slúži ako základ pre vlastné, používateľsky kreslené komponenty, ktoré na tento účel musia predefinovať metódu **paint()**. Pohybujeme v oblasti programovania riadeného udalosťami, nemôžeme si preto do okna kresliť, kedy sa nám zachce, ale iba vtedy, keď nám to systém dovolí. Filozofia je našťastie veľmi jednoduchá: v okamihu, keď treba komponent prekresliť, zavolá sa jeho metóda **paint()**. Ak ju nahradíme vlastnou verziou, máme proces kreslenia „pod palcom“.

Usporiadanie komponentov

Ako sme už spomenuli, usporiadanie komponentov v rámci kontajnerov majú na starosti zvláštne objekty, ktoré implementujú rozhranie **LayoutManager**. Toto rozhranie obsahuje niekoľko metód, ktoré sú volané kontajnerom (a pokiaľ nebudeme písať vlastný manažér, nemusíme sa nimi zapodievať) a má jedného potomka, rozhranie **LayoutManager2**, pridávajúce schopnosť rozmiestňovať komponenty na základe zadaných obmedzení.

V balíku **java.awt** nájdeme päť manažérov. Najjednoduchší je zrejme **FlowLayout**, ktorý umiestňuje komponenty do kontajnera v riadkoch zľava doprava. Keď sa komponent na riadok nezmestí, „pretečie“ do ďalšieho riadka. Pri zmene veľkosti kontajnera sa komponenty prelievajú z riadka na riadok podľa aktuálnych rozmerov. Pri konštrukcii objektu **FlowLayout** môžeme zadať spôsob zarovnania komponentov v rámci riadka a veľkosť medzery medzi komponentmi. Kontajnery triedy Panel majú implicitne nastavený manažér **FlowLayout** (zopakujme, že layout manager kontajnera sa nastavuje pomocou metódy **setLayout()**).

Ďalší manažér, trieda **GridLayout**, usporiadava komponenty do pravouhlej siete. Pri konštrukcii možno zadať počet stĺpcov a riadkov siete i rozstup komponentov. Každý komponent zaberá jednu „bunku“ siete (nejde teda o umiestňovanie na zadanú pozíciu, ako je to bežné v iných prostrediach).

Trieda **BorderLayout** umiestňuje komponenty do jednej z piatich oblastí, pomenovaných príznačne sever, juh, západ, východ, stred. Opätovne možno zadať veľkosť medzier medzi komponentmi.

Manažér triedy **CardLayout** zoradí všetky komponenty nad seba a zobrazí vždy len jeden z nich (vytvára teda akoby balíček kariet). To, ktorý komponent bude navr-

chu, nastavíme pomocou metód **first(), last(), next(), previous()** a **show().**

Najzložitejším a súčasne najmocnejším manažérom je **GridBagLayout**. Nanešťastie nemáme miesto na jeho podrobnejší opis, v skratke platí, že pre každý komponent môžeme (ba musíme) zadať množstvo obmedzení (vo forme objektu triedy **GridBagConstraints**). Tieto obmedzenia opisujú, kde sa komponent bude nachádzať (v rámci pomyselnej mriežky – podobne ako pri **GridLayout**), či bude zaberať viacero buniek mriežky, či sa zväčší tak, aby zaplnil celú oblasť, ktorú má k dispozícii, veľkosť medzier okolo komponentu, dokonca môžeme stanoviť, ktoré komponenty budú meniť svoje rozmery pri zmene veľkosti kontajnera a ktoré zostanú nemenné.

Ak potrebujeme v rámci jedného kontajnera použiť viacero layoutov, musíme skupiny komponentov umiestniť na samostatné panely s potrebnými layout manažérmi a tieto panely vložiť do kontajnera.

Práca s grafikou

Ak chceme v programe zobrazovať kreslený výstup (napríklad v prípade vlastných komponentov), celkom isto prídeme do styku s triedou **Graphics**. Táto abstraktná trieda predstavuje grafický kontext (podobne ako napr. vo Windows), v rámci ktorého existuje niekoľko charakteristík: komponent, do ktorého sa bude kresliť, poloha a farba pera, font použitý pri kreslení textu a podobne. Inštanciu typu **Graphics** pre požadovaný komponent získame volaním metódy **getGraphics()**.

V triede **Graphics** nájdeme mnoho metód na zisťovanie a nastavovanie uvedených charakteristík, nastavenie orezávacieho obdĺžnika (clipping rectangle – oblasť, mimo ktorej sa prípadný grafický výstup bude ignorovať), kreslenie čiar, plných aj prázdnych obdĺžnikov, elíps, oblúkov, polygónov, textových reťazcov, obrázkov a podobne. Trieda **Graphics2D**, ktorá je potomkom triedy **Graphics**, tieto možnosti ešte rozširuje.

Pomocné triedy

Z množstva ďalších tried spomenieme Point na reprezentáciu dvojrozmerných bodov s metódami na absolútnu (**move()**) aj relatívnu (**translate()**) zmenu súradníc, triedu **Color** na reprezentáciu farieb v modeli RGBA (teda s podporou priehľadnosti), ktorá obsahuje aj preddefinované konštanty pre niekoľko najbežnejších farieb, jej potomka **SystemColor** na reprezentáciu systémových farieb, ďalej triedu **Cursor** na prácu s preddefinovanými typmi kurzorov, pomerne komplexnú triedu Font na reprezentáciu fontov, ktoré sú v systéme k dispozícii, spolu s triedou **FontMetrics**, ktorá opisuje rozmerové vlastnosti fontov, triedu Insets, ktorá sa používa na opis veľkosti okrajov kontajnera alebo komponentov umiestňovaných pomocou manažéra **GridBagLayout**, a triedy **Rectangle** a **Polygon**, predstavujúce obdĺžniky a polygóny, s metódami na rôzne geometrické operácie (zjednotenie, prienik, test polohy bodu voči objektu a pod).

16.časť: Balík java.awt (dokončenie)

V predchádzajúcom pokračovaní sme zabudli spomenúť ešte dve významnejšie pomocné triedy. Prvá z nich, **Dimension**, predstavuje zapuzdrenie šírky a výšky komponentu do jedného objektu. Druhá s názvom **Toolkit** je podstatne komplexnejšia a dá sa povedať, že reprezentuje implementáciu AWT. Nájdeme v nej množstvo metód na vytváranie natívnych komponentov, ale aj užitočné metódy **getScreenSize()** a **getScreenResolution()** na zistenie parametrov obrazovky, **getImage()**, **createImage()**, **prepareImage()** a **checkImage()** na prácu s obrázkami, metódu **beep()**, ktorá generuje systémové „pípnutie“ a mnoho ďalších.

Menu komponenty

Na prácu s menu slúži skupina tried so spoločným predkom, abstraktnou triedou **MenuComponent**. Každý menu komponent má svoj názov, ktorý môžeme zistiť a nastaviť dvojicou metód **(get|set)Name()**. Metóda **getParent()** vracia rodičovský kontajner, v ktorom je zadaný komponent obsiahnutý, metódy **(get|set)Font()** slúžia na zistenie a nastavenie fontu použitého v menu komponente.

Obyčajné, ďalej nerozbaliteľné položky menu predstavuje trieda **MenuItem**. Pri konštrukcii položky zadávame ako parameter textový reťazec, ktorý sa zobrazí v menu. Tento reťazec môžeme neskôr zisťovať a nastavovať metódami **(get|set)Label()**. Položky môžu byť nastavené ako neaktívne (a obyčajne zobrazené svetlou, nevýraznou farbou) volaním metódy **setEnabled()**. Pomocou metód **(get|set)ActionListener()** môžeme k položke pripojiť príslušný listener.

Špeciálnym prípadom položky menu je trieda **CheckboxMenuItem** (odvodená od **MenuItem**). Tá obsahuje navyše binárny stav, indikovaný zaškrtávacou značkou. Stav môžeme zisťovať a nastavovať pomocou metód **(get|set)State()**.

Základným typom menu kontajnera je trieda **Menu**, ktorá reprezentuje vertikálne menu. Jednotlivé položky menu musia byť typu **MenuItem**, a keďže samotná trieda **Menu** je potomkom triedy **MenuItem**, nie je problém vytvoriť hierarchickú štruktúru menu jednoduchým skladaním. Objekt triedy **Menu** má svoj názov a môže predstavovať oddeliteľné (tear-off) menu. Túto skutočnosť indikuje metóda **isTearOff().** Z ďalších metód **getItemCount()** vracia počet položiek v kontajneri, **getItem()** sprístupňuje jednotlivé položky, **add(), insert()** a **remove()** slúžia na pridávanie alebo odoberanie položiek. Špeciálnym typom položiek sú oddeľovacie pruhy, ktoré pridávame pomocou metód **addSeparator()** a **insertSeparator()**.

Vodorovný pruh s položkami menu implementuje trieda **MenuBar**. Jednotlivé položky musia byť typu **MenuItem** (plus všetkých jeho potomkov). Objekt triedy **MenuBar** sa pripája ku komponentom typu **Frame** pomocou metódy **Frame.setMenuBar()**. Na pridávanie a odoberanie položiek menu máme k dispozícii metódy **add()** a **remove()**, počet položiek v menu zistíme volaním **getMenuCount().**

V AWT možno vytvárať aj kontextové menu, ktoré nemusia byť súčasťou vodorovného menu okna, ale môžu byť zobrazené na zadanej pozícii. Takéto menu sú objektmi triedy **PopupMenu**, ktorá je odvodená od **Menu**; zobrazenie menu má na starosti metóda **show()**.

Ukážme si ešte krátky príklad, ako zostrojiť základné menu s dvoma podmenu *File* a *Help* a s niekoľkými položkami (predpokladáme, že uvedený kód je súčasťou konštruktora okna – kompletný príklad možno tradične nájsť na webe):

```
Menu mnFile = new Menu("File");
mnFile.add(new MenuItem("New"));
mnFile.add(new MenuItem("Open..."));
mnFile.add(new MenuItem("Save"));
mnFile.add(new MenuItem("Save as..."));
mnFile.addSeparator();
mnFile.add(new MenuItem("Exit"));

Menu mnHelp = new Menu("Help");
mnHelp.add(new MenuItem("About..."));

MenuBar mb = new MenuBar();
mb.add(mnFile);
mb.add(mnHelp);

setMenuBar(mb);
```

Rozhrania java.awt

Z rozhraní balíka **java.awt** za zmienku stoja nasledujúce: **Adjustable** reprezentuje objekty, ktoré umožňujú zmenu číselnej hodnoty v stanovenom rozsahu. Z balíka **java.awt** je takým objektom trieda **Scrollbar**. Rozhranie **ItemSelectable** predstavuje objekty obsahujúce zoznam položiek, ktoré možno nejakým spôsobom „vybrať“ – ako napríklad triedy **List**, **Checkbox** či **Choice**.

Rozhranie **MenuContainer** implementujú všetky triedy, ktoré fungujú ako menu kontajnery. Okrem **Menu** a **MenuBar** je to tiež trieda **Frame** a na účely prípadného rozširovania aj trieda **Component**.

Všetky triedy, ktoré dokážu rozmiestňovať komponenty v kontajneroch, implementujú rozhranie **LayoutManager**. Potomkom tohto rozhrania je **LayoutManager2**, opisujúci triedy, ktoré vedia rozmiestňovať komponenty aj na základe zadaných obmedzení.

Chyby a výnimky

Vážnu, neopraviteľnú chybu AWT indikuje objekt triedy **AWTError**, odvodenej od triedy **Error**. Tá, ako vieme, predstavuje chyby, pri ktorých sa nepredpokladá obnovenie normálnej činnosti programu a nie je nevyhnutné ich zachytávať. Bežné výnimky AWT sú naproti tomu signalizované triedou **AWTException**. Okrem nej nájdeme v balíku **java.awt** ešte výnimky **FontFormatException** (chyba vo formáte súboru s definíciou písma) a **IllegalComponentStateException** (indikuje neplatný stav komponentu).

Triedy udalostí

Ďalej sa budeme venovať podbalíku **java.awt.event**, ktorý obsahuje triedy a rozhrania určené na obsluhu udalostí GUI. Spoločným predkom pre triedy udalostí v AWT je trieda **AWTEvent** (patriaca však do balíka **java.awt**). Inštancie tejto triedy, resp. jej potomkov, nevytvára programátor, ale samotný systém ako reakciu na výskyt rôznych udalostí. Každý typ udalosti je identifikovaný celým číslom, uloženým v chránenom člene **id**. Jeho hodnotu zistíme pomocou metódy **getID()**. Veľmi často je nevyhnutné určiť konkrétny typ udalosti práve na základe jej identifikačného čísla.

„Akčné“ udalosti, ako napríklad kliknutie na objekt **Button**, reprezentuje trieda **ActionEvent**. Pomocou metódy **getActionCommand()** môžeme získať príkazový reťazec, asociovaný k akcii (rôzne tlačidlá budú mať obyčajne priradené rôzne reťazce, podľa čoho môžeme jednoducho identifikovať, ktoré tlačidlo bolo stlačené); metóda **getModifiers()** vracia stav modifikačných klávesov (Shift a pod.) v okamihu výskytu udalosti.

Trieda **AdjustmentEvent** predstavuje udalosti generované triedami, ktoré implementujú rozhranie **Adjustable**, ako napríklad trieda **Scrollbar**. Pomocou metódy **getAdjustable()** získame objekt, ktorý udalosť spôsobil, **getValue()** vracia aktuálnu „hodnotu“ objektu (pri posuvníku je to poloha jazdca) a **getAdjustmentType()** použijeme na bližšie určenie typu udalosti.

Ďalšia trieda **ComponentEvent** zastupuje udalosti, ako posun, zmenu veľkosti a viditeľnosti komponentu a podobne. Tieto udalosti sú určené len na informačné účely – systém sa postará o zodpovedajúce prekreslenie komponentov sám.

Trieda **ComponentEvent** má niekoľko potomkov. Prvým z nich je trieda **ContainerEvent**, ktorá predstavuje udalosti týkajúce sa kontajnerov (opäť je určená len na informačné účely). **FocusEvent** reprezentuje získanie a stratu fokusu (ak má komponent **fokus**, znamená to, že doň budú smerované všetky vstupy z klávesnice; **fokus** sa obyčajne znázorňuje vizuálne – ako bodkovaný obdĺžnik či silnejšie orámovanie).

Ďalší potomok **InputEvent** slúži ako spoločný predok pre udalosti vstupu z klávesnice alebo myši. Pomocou niekoľkých metód môžeme zistiť stav modifikačných klávesov v okamihu výskytu udalosti. Udalosti klávesnice bližšie špecifikuje odvodená trieda **KeyEvent**. Tieto udalosti sú troch typov: stlačenie klávesu, uvoľnenie klávesu a „vstup znaku“. Tretí typ udalosti býva dôsledkom výskytu prvých dvoch typov, v rôznych kombináciách (napríklad udalosti „vstup znaku A“ predchádzajú udalosti „stlačenie Shift“, „stlačenie a“, „uvoľnenie a“ a „uvoľ-

nenie Shift“). Príslušný znak a jeho kód získame volaním metód **getKeyChar()** a **getKeyCode()**, slovný ekvivalent kódu znaku (ako napríklad „F1“) volaním metódy **getKeyText()**. Ďalšie metódy **setKeyChar()**, **setKeyCode()** a **setModifiers()** umožňujú zmeniť kód znaku, ako keby bol stlačený úplne iný kláves.

Udalosti myši, ako jej pohyb, prechod nad komponentom, stlačenie a uvoľnenie tlačidiel, sú reprezentované triedou **MouseEvent**. Polohu myši zistíme pomocou metód **getX()** a **getY()**, počet kliknutí vracia metóda **getClickCount()** (jednoduché i dvojité kliknutie majú rovnaký identifikátor **MOUSE_CLICKED**, líšia sa v návratovej hodnote tejto metódy). Metóda **isPopupTrigger()** indikuje, či daná udalosť slúži ako povel na zobrazenie kontextového menu pre danú platformu.

Trieda **PaintEvent** existuje v súvislosti s prekresľovaním komponentov, používa ju však systém, nie programátor. Poslednou komponentovou udalosťou je trieda **WindowEvent**, indikujúca zmeny stavu okna (otvorenie, zavretie, maximalizáciu, minimalizáciu a pod.).

Udalosti súvisiace so zmenou hierarchie komponentov reprezentuje trieda **HierarchyEvent**, určená opätovne len na informačné účely. Udalosti označenia a odznačenia položky v objekte typu **ItemSelectable** (takým je napríklad trieda **List**) reprezentuje trieda **ItemEvent**. Príslušnú položku získame volaním metódy **getItem()**. Poslednou udalostnou triedou je **TextEvent**, ktorá indikuje zmenu textového obsahu v komponente.

Zachytávanie udalostí

Ako sme hovorili minule, jednotlivé komponenty AWT generujú rôzne množiny udalostí a iným objektom poskytujú možnosť reagovať na výskyt týchto udalostí. Objekt, ktorý sa chce dozvedieť o každom výskyte nejakého typu udalosti, pripojí k príslušnému komponentu tzv. listener. Sú v zásade dva spôsoby, ako vytvoriť listener: implementáciou príslušného „počúvacieho“ rozhrania alebo odvodením od niektorej adaptérovej triedy.

„Počúvacích“ rozhraní je spolu pätnásť a všetky sú odvodené od rozhrania **EventListener** (neobsahuje žiadne metódy, slúži len ako spoločný predok). Nebudeme ich tu všetky menovať, v zásade majú názvy korelujúce s metódami typu **(add|remove)*Listener()** jednotlivých komponentov. Jeden príklad za všetky: rozhranie **ActionListener**. Toto rozhranie musia implementovať všetky listenery, ktoré majú zachytávať „akčné“ udalosti, ako napríklad kliknutie na tlačidlo. Rozhranie tvorí jediná metóda **actionPerformed()**, ktorá je volaná v okamihu výskytu udalosti. Parametrom metódy actionPerformed() je objekt typu **ActionEvent**.

Treba mať na pamäti, že metóda **actionPerformed()** je súčasťou listenera a volá ju komponent, na ktorom k udalosti došlo. Vo všeobecnosti si každý komponent udržuje zoznam všetkých zaregistrovaných listenerov a pri výskyte niektorej udalosti ich všetkých informuje zavolaním ich príslušnej metódy.

Podobne vyzerajú aj ostatné „počúvacie“ rozhrania, môžu však obsahovať aj viac ako jednu metódu, na indikáciu rôznych podtypov udalostí (napríklad **WindowListener** obsahuje sedem metód, zodpovedajúcich siedmim podtypom udalosti **WindowEvent**). Určitou nevýhodou tohto spôsobu zachytávania udalostí môže byť nutnosť implementovať všetky metódy príslušného rozhrania, aj keď máme záujem iba o jeden podtyp udalosti. Malý príklad, ako pripojiť k tlačidlu akčný listener:

```
import java.awt.*;
import java.awt.event.*;

...

Button b = new Button("OK");
b.addActionListener(new ButtonWatcher());

...

class ButtonWatcher implements ActionListener
{
  void actionPerformed(ActionEvent e)
  {
    // ošetrenie udalosti
  }
}
```

(Dva riadky kódu, na ktorých sa vytvára nový objekt typu **Button** a pripája sa k nemu listener, budú, pochopiteľne, súčasťou nejakej metódy).

Druhým spôsobom zachytávania udalostí je použitie niektorej z adaptérových tried. Ide o osem abstraktných tried, ktoré implementujú vybrané počúvacie rozhrania. Metódy týchto tried sú implicitne prázdne. Ak od niektorej z tried odvodíme vlastný adaptér, postačí reimplementovať len tie metódy, o ktoré máme záujem (na rozdiel od počúvacích rozhraní, kde sme povinní implementovať *všetky* metódy). Názvy adaptérových tried sú podobné názvom počúvacích rozhraní, len namiesto slova **Listener** je **Adapter**. Adaptérové triedy existujú len pre rozhrania s viac ako jednou metódou.

Príklad použitia triedy **WindowAdapter**:

```
// nech f je inštanciou Frame
f.addWindowListener(new WindowWatcher());

...

class WindowWatcher extends WindowAdapter
{
  void windowClosing(WindowEvent e)
  {
    // reakcia na zavretie okna
  }
}
```

Treba mať na pamäti, že v praxi sa namiesto samostatných tried (ako sú **ButtonWatcher** a **WindowWatcher** v našich príkladoch) častejšie používajú triedy vnorené alebo dokonca anonymné, takže sa stretneme napríklad so zápisom:

```
f.addWindowListener(new WindowAdapter
{
  void windowClosing(WindowEvent e)
  {
    // reakcia na zavretie okna
  }
});
```

K anonymným triedam sa však dostaneme až neskôr.

Práca s obrázkami

Ďalší podbalík **java.awt.image** poskytuje triedy určené na vytváranie a úpravu obrázkov. Ako abstraktná reprezentácia grafického obrázka slúži trieda **Image** (ktorá patrí do **java.awt**). Z dôležitých metód spomeňme **getWidth()** a **getHeight()** na zistenie rozmerov obrázka a **getSource()** na získanie objektu, ktorý má na starosti vygenerovanie jednotlivých pixelov obrázka.

Na začiatku článku sme spomenuli triedu **Toolkit**, ktorá poskytuje niekoľko metód na vytvorenie a prácu s obrázkami. V balíku **java.awt.image** sa nachádzajú triedy určené pre komplexnejšie operácie. Práca s obrázkami je postavená na prúdovom modeli, ktorý zahŕňa „producenta“ obrázka (implementuje rozhranie **ImageProducer**), voliteľné filtre a „spotrebiteľa“ obrázka (implementuje rozhranie **ImageConsumer**).

Základom obrázkových filtrov je trieda **ImageFilter**, ktorá sa sama osebe javí ako spotrebiteľ obrázka. Táto trieda predstavuje prázdny filter a slúži len ako bázová implementácia. Praktickejšia je trieda **CropImageFilter**, ktorá dokáže pôvodný obrázok orezať na požadovanú veľkosť. Na jednoduchú zmenu rozmerov obrázka možno použiť triedu **ReplicateScaleFilter**, ktorá používa najjednoduchší možný algoritmus: opakuje riadky (pri zväčšovaní obrázka) alebo ich vynecháva (pri zmenšovaní). O niečo lepšie výsledky dáva trieda **AreaAveragingScaleFilter**, ktorá pri zmene rozmerov počíta priemerné hodnoty pixelov. Najzaujímavejšia je však abstraktná trieda **RGBImageFilter**, ktorá umožňuje filtrovať obrázky na základe zložiek RGB jednotlivých pixelov. Vzhľadom na jej abstraktnosť je jasné, že konkrétny filter treba naprogramovať reimplementáciou príslušných metód.

Na prepojenie producenta obrázka a niektorého z filtrov použijeme triedu **FilteredImageSource**. Tá sa sama tvári ako producent, takže v prípade nutnosti použiť viacero filtrov stačí triedy vhodne zreťaziť rovnakým spôsobom ako pri filtroch údajových prúdov.

Ostatné triedy spomeňme len stručne: **ColorModel** je abstraktná trieda na reprezentáciu farebných modelov. Jej konkrétne implementácie tvoria triedy **ComponentColorModel** (zložkový model, jednotlivé farebné zložky sú samostatnými údajmi), **IndexColorModel** (paletový systém, jednotlivé farby sú indexmi do zadanej farebnej palety) a **DirectColorModel**, ktorá je odvodená cez medzipotomka **PackedColorModel** (opäť zložkový model, jednotlivé zložky sú však skombinované do jedného údaja – napríklad model RGB565 v 16-bitových farebných režimoch).

Na reprezentáciu pixelového rastra slúži trieda **Raster**. Nad rastrami možno vykonávať rôzne operácie, ako afinné transformácie, konvolúcie a podobne. Údaje rastra sú uložené v objekte typu **DataBuffer**.

Ďalšie podbalíky

V balíku **java.awt** možno nájsť niekoľko ďalších podbalíkov, ale s ohľadom na rozsah článku sa im nebudeme podrobnejšie venovať. Takže len informatívne: balík **java.awt.color** obsahuje triedy na prácu s farebnými profilmi ICC. Balík **java.awt.datatransfer** poskytuje rozhrania a triedy na výmenu dát medzi aplikáciami (pomocou schránky – clipboardu). V balíku **java.awt.dnd** sa nachádzajú triedy potrebné na implementáciu mechanizmu drag-and-drop. Ďalší balík **java.awt.font** v súlade so svojím názvom poskytuje triedy a rozhrania na prácu s písmami a ich opisnými súbormi (fontami). Balík **java.awt.geom** obsahuje množstvo tried na prácu s dvojrozmernou grafikou. Balíky **java.awt.im** a **java.awt.im.spi** umožňujú zápis netradičných znakov (typicky japonských, čínskych či kórejských) pomocou tzv. vstupných metód. No a konečne v poslednom balíku **java.awt.print** nájdeme rozhrania a triedy na prácu s tlačiarňou.

Pri grafike zostaneme

Java 2 obsahuje okrem AWT ešte jeden grafický rámec s poetickým názvom **Swing**. Ten je o niečo bohatší a prispôsobiteľnejší. Viac si o ňom povieme v budúcom pokračovaní.

17.časť: Thready a synchronizácia

Napriek sľubu v závere predošlej časti sa balíku *Swing* ani ďalším balíkom v tomto seriáli nebudeme venovať. Pri zložitejších témach totiž vôbec nie je jednoduché efektívne zhrnúť obrovské množstvo informácií do jednej či dvoch častí seriálu. Ak sa to podarí, výsledok pripomína skôr referenčnú príručku a o tom seriál predsa nie je. Čitateľ chtivý podrobnejších informácií preto bude musieť hľadať aj inde, najlepšie na internete.

Multithreading

Sotva by sa dnes uživila programovacia platforma, ktorá by neumožňovala paralelné vykonávanie úloh. Podľa úrovne, na ktorej paralelizmus pozorujeme, môžeme hovoriť o *multitaskingu* alebo o *multithreadingu*. Prvý pojem označuje schopnosť súčasného behu viacerých procesov. Pod procesom si predstavme samostatne vykonávanú exekučnú jednotku, ktorá má pridelený adresový priestor, disponuje vnútorným stavom a môže vlastniť

prostriedky OS. Skrátka, proces je „bežiaci program“. Jeden binárny program môže bežať súčasne vo viacerých kópiách – každá kópia je potom samostatným procesom. Multitasking sa dnes považuje za samozrejmosť.

Pri multithreadingu rozdelíme jeden proces na niekoľko menších podprocesov, tzv. *threadov*. Názov pochádza z angličtiny, kde znamená „vlákno“. O threadoch preto niekedy hovoríme ako o „exekučných vláknach“. Thready medzi sebou zdieľajú adresový priestor, môžu však obsahovať vlastné, privátne údaje. Dôvodom, prečo sa uchyľujeme k rozdrobeniu aplikácie na thready, je obyčajne snaha o zlepšenie používateľského ohlasu. V takom textovom editore sa jeden thread môže starať o editáciu textu, ďalší o obnovu obrazovky, kontrolu pravopisu či tlač na pozadí. Keby toto všetko mal robiť jeden thread, nebolo by možné pri tlačení upravovať text, priebežne kontrolovať pravopis a podobne. Existujú však aj typy úloh, pri ktorých je výhodné použiť thready z iného dôvodu – napríklad na zjednodušenie algoritmu alebo implementáciu zložitejšieho správania.

Ak máme dosiahnuť paralelný beh viacerých procesov či threadov na jednoprocesorovom systéme, musíme sa uchýliť k nejakému triku. Rýchlym prepínaním medzi jednotlivými threadmi môžeme vytvoriť ilúziu súčasného behu viacerých úloh. V operačnom systéme zvyčajne existuje samostatný modul, plánovač (scheduler), ktorý sa stará o prideľovanie procesora jednotlivým threadom na základe vhodnej stratégie. Ak je plánovač schopný po uplynutí vopred stanoveného časového kvanta thread pozastaviť a spustiť iný, hovoríme o *preemptívnom plánovaní*. (V ére šestnásťbitových Windows, ktoré neposkytovali preemptívny multitasking, sa musel každý bežiaci proces explicitne a dobrovoľne zrieť tohto svojho výsadného postavenia. Ak sa stalo, že sa program zasekol, spolu s ním išiel pod kytičky aj operačný systém.)

Niektoré thready sú pre beh systému dôležitejšie. Aby sa zabezpečila plynulosť ohlasu, majú thready priradenú celočíselnú hodnotu *priority*. Plánovač threadov preferuje pri prideľovaní procesora thready s vyššou prioritou. Thready sa môžu nachádzať v rôznych stavoch, najhrubšie rozdelenie je na thready bežiace (môže ich byť aj viac ako jeden, napríklad na multiprocesorových systémoch), thready pripravené na naplánovanie a thready „spiace“, ktoré čakajú na výskyt nejakej udalosti. Väčšina threadov je počas svojho života v tomto treťom stave, pretože čaká na vstup z klávesnice, na pohyb myši, na vykreslenie obrázka, na dodanie údajov z disku a podobne. Len thready, ktoré intenzívne počítajú, dokážu využiť svoje časové kvantum naplno.

Thread môže zmeniť svoj stav aj inak ako vypršaním prideleného času – môže sa dobrovoľne zrieť procesora alebo môže vyjadriť svoju potrebu čakať na splnenie nejakej podmienky. Spiaci thread môže byť zobudený systémom alebo iným threadom, ktorý sa postaral o splnenie podmienky.

Thready a Java

Java poskytuje dva spôsoby, ako vytvoriť thread. Prvým je odvodenie vlastnej podtriedy systémovej triedy **java.lang.Thread**. Jadrom threadu je metóda **run()** (treba ju, pochopiteľne, predefinovať), ktorá sa začne vykonávať po spustení threadu. To dosiahneme zavolaním metódy **start()**. Thread beží, kým je v metóde **run()** čo robiť. Len čo sa v metóde vyskytne príkaz return či nezachytená výnimka a aj keď tok riadenia dospeje ku koncovej zátvorke, thread končí a systém ho po čase zlikviduje. Ukážka vytvorenia a spustenia nového threadu:

```
class MyThread extends Thread
{
  // ...
}

MyThread mt = new MyThread();
mt.start();
```

V príklade vytvárame objekt **MyThread**, ktorý sme odvodili od triedy **Thread**, a voláme jeho metódu **start()**.

Druhý spôsob spočíva v implementácii rozhrania **Runnable**, ktoré obsahuje metódu **run()**. Objekt implementujúci toto rozhranie potom môžeme zadať ako argument konštruktora na vytvorenie objektu typu **Thread**. Pri tomto spôsobe sa po zavolaní metódy **start()** začne vykonávať metóda **run()** objektu **Runnable**. Ukážka:

```
class MyThread implements Runnable
{
  // ...
}

MyThread mt = new MyThread();
Thread tt = new Thread(mt);
tt.start();
```

Ako vidno, tentoraz vytvárame dva objekty; jeden typu **MyThread** (to je ten, ktorý v metóde **run()** obsahuje kód nového threadu) a druhý typu **Thread** – ten funguje ako „obálka“ prvého objektu.

Pri vytváraní nového threadu môžeme voliteľne zadať jeho meno; to sa môže hodiť napríklad pri ladení projektu. Okrem toho môžeme threadu prideliť skupinu, do ktorej bude patriť (pozri ďalej).

Trieda **Thread** poskytuje niekoľko veľmi potrebných metód. Pomocou metódy **yield()** sa napríklad thread môže vzdať procesora. Zavolaním metódy **sleep()** thread na stanovený čas „zaspí“. Násilne prerušiť thread možno metódou **interrupt()**. Pomocou metódy **join()** môžeme prikázať aktuálnemu threadu, aby čakal na skončenie iného threadu. Metóda **isAlive()** indikuje, či daný thread žije, t. j. či bol spustený a dosiaľ beží. Názov a prioritu threadu zisťujeme a nastavujeme dvojicami **(get|set)Name()** a **(get|set)Priority()**.

Javovský program sa končí buď explicitným volaním **System.exit()**, alebo v okamihu, keď svoj beh skončia všetky thready. Pomocou metódy **setDaemon()** môžeme vybrané thready premeniť na „démonov“ (a naopak). Systém pri rozhodovaní, či ukončiť program, na „démonické“ thready neberie ohľad. Metóda run() týchto threadov obyčajne obsahuje nekonečnú slučku. Na zistenie, či je vybraný thread démonom, použijeme metódu **isDaemon()**.

Skupiny threadov

Thready možno zlučovať do skupín, ktoré tvoria stromovú štruktúru. Každá skupina okrem počiatočnej má pridelenú rodičovskú skupinu. Pri vytvorení nového threadu možno určiť, do ktorej skupiny bude thread zaradený.

Na reprezentáciu skupiny threadov je určená trieda **ThreadGroup**. Pri vytvorení jej možno prideliť meno a rodičovskú skupinu. Ak rodičovskú skupinu nezadáme, stane sa ňou automaticky skupina, do ktorej patrí aktuálny thread. Pomocou metódy **setMaxPriority()** nastavujeme maximálnu hodnotu priority, akú môžu mať thready patriace do skupiny. Z ďalších metód vyberáme **setDaemon()** na transformáciu všetkých threadov v skupine na démonov a naopak, **interrupt()** na prerušenie všetkých threadov v skupine, **getName()**, **getParent()**, **getMaxPriority()**, **isDaemon()**, **isDestroyed()** na zistenie atribútov skupiny či **activeCount()/activeGroupCount()** na zistenie počtu aktívnych threadov a aktívnych skupín patriacich do zadanej skupiny.

V súvislosti s threadmi spomeňme ešte triedu **ThreadLocal**. Táto trieda predstavuje zvláštny druh objektov, líšiacich sa od bežných premenných v tom, že každý thread, ktorý s nimi pracuje, má k dispozícii vlastnú, nezávisle inicializovanú kópiu. Objekty **ThreadLocal** zvyčajne bývajú privátnymi statickými zložkami tried, ktoré s threadmi nejakým spôsobom súvisia.

Na čítanie a zápis hodnoty premenných typu **ThreadLocal** použijeme metódy **get()** a **set()**, ktoré pracujú s typom **Object**, takže uložená hodnota môže byť v podstate ľubovoľná. Počiatočná hodnota premenných je null; ak potrebujeme použiť inú inicializačnú hodnotu, musíme si odvodiť vlastnú triedu a predefinovať chránenú metódu **initialValue()**.

Synchronizácia

Nevyhnutným dôsledkom zavedenia multitaskingu a multithreadingu je celkom nová trieda problémov, ktoré v jednopoužívateľských a jednoúlohových prostrediach prakticky neexistovali. Zoberme si ako príklad dva thready, ktoré pracujú s rovnakými údajmi. V prípade, že každý thread číta a zapisuje údaje bez ohľadu na ten druhý, ľahko môže dôjsť k narušeniu integrity údajov. Majme nasledujúci kód:

```
void run()
{
  for (int i = 0; i < 10; i++)
    x = x + 1;
}
```

Nech oba thready vykonávajú tento kód, ktorého účelom je desaťkrát inkrementovať premennú x, ktorá je prístupná obom threadom (napríklad je statickým členom tej triedy, do ktorej patrí metóda **run()**). Predpokladajme, že počiatočná hodnota **x** je rovná nule. Ak najprv spustíme jeden thread a po jeho skončení druhý, dostaneme výsledok **20**. Nechajme teraz oba thready bežať naraz. V závislosti od okolností bude ležať konečný výsledok niekde v rozsahu od **10** po **20**. Ako je to možné?

Jadrom problému je riadok **x = x + 1** (zámerne nie je použitý operátor **++**). Pri vyhodnocovaní tohto výrazu thread načíta obsah premennej **x** do dočasnej pamäte (registra), zvýši ho v nej o jednotku a uloží naspäť do hlavnej pamäte. Toto vyhodnotenie však nie je atomické (nedeliteľné) a v prípade, že threadu vyprší časové kvantum niekde „uprostred“, v premennej **x** zatiaľ zostane stará hodnota. Druhý thread si odteraz s premennou môže robiť, čo chce, jeho úpravy budú onedlho zabudnuté; len čo sa totiž k slovu opäť dostane prvý thread, dokončí vyhodnotenie výrazu a prepíše momentálnu hodnotu **x** v hlavnej pamäti hodnotou, ktorá odpočívala v pomocnom registri.

Poznámka: Je pochopiteľné, že pri praktickej realizácii k opísanému problému vôbec nemusí dôjsť. Použitá implementácia JVM môže vyhodnotenie výrazu vykonať atomicky alebo budú okolnosti natoľko priaznivé, že k žiadnej chybe nedôjde. To však vo všeobecnosti očakávať nemôžeme – vždy treba rátať s najhorším možným prípadom. Programátor, ktorý sa spolieha na náhodu, si nezaslúži nič iné, len svoje programy za trest používať.

Ak chce zvedavý čitateľ vidieť efekt vzájomného „lezenia do kapusty“ na vlastné oči, stačí inkriminovaný výraz rozpísať napríklad takto:

```
int x_tmp = x;
yield();
x = x_tmp + 1;
```

Uvedené tri riadky simulujú vyhodnotenie pôvodného výrazu s vynúteným preplánovaním na druhý thread medzi načítaním starej a uložením novej hodnoty **x**. Výsledná hodnota **x** bude pravdepodobne **10**.

Tento príklad len jemne naznačuje celú sféru problémov, ktoré môžu vzniknúť pri súbežnej činnosti viacerých threadov a procesov. Riešenie spočíva v použití niektorej zo synchronizačných metód. Rozsah článku, bohužiaľ, nedovoľuje zaoberať sa týmito metódami podrobnejšie, pretože ide o veľmi rozsiahlu (ale o to zaujímavejšiu) tému. Pozrime sa preto len na možnosti, ktoré na vyriešenie synchronizačných problémov poskytuje Java.

Príkaz synchronized

Úsek kódu, ktorý by sa mal vykonávať atomicky, t. j. maximálne jedným threadom súčasne, nazveme v súlade s tradíciami *kritickou sekciou*. Vylúčiť súčasnú prítomnosť

viacerých threadov v kritickej sekcii je jednoduché: nepovolíme do nej vstup v prípade, že sa tam nachádza iný thread. Je, samozrejme, nevyhnutné, aby thready pred vstupom do kritickej sekcie svoj úmysel dali najavo.

Existuje mnoho spôsobov, ako zabezpečiť toto tzv. *vzájomné vylučovanie* (mutual exclusion). Java používa zámky (locks) a z nich vychádzajúcu implementáciu monitorov (pozri ďalej). Zámok je metaobjekt, ktorý nie je priamo prístupný programátorovi a viaže sa vždy na existujúci objekt (to, mimochodom, znamená, že pomocou primitívnych typov zamykať nemožno). Pre každú kritickú sekciu v programe by mal existovať samostatný zámok.

Pred vstupom do kritickej sekcie sa thread pokúsi získať zámok nad vybraným objektom. Ak sa mu to podarí, zámok sa uzamkne a thread môže vykonávať kód kritickej oblasti. Ak sa mu to nepodarí, pretože zámok je už zamknutý, znamená to, že v kritickej oblasti je niekto iný. Thread bude preto pozastavený a opäť sa rozbehne až po odomknutí zámku. Nevstupuje však do kritickej sekcie automaticky, ale opakuje svoj pokus o získanie zámku. Medzi okamihom, keď thread prešiel zo spiaceho stavu do stavu pripravenosti, a okamihom, keď mu bol pridelený procesor, totiž mohol do kritickej sekcie vstúpiť ďalší thread (hoci aj ten istý, čo v nej bol predtým). Že to nie je spravodlivé? Nuž, v paralelnom prostredí si príliš nenavyberáme…

V zdrojovom kóde kritickú sekciu musíme „obaliť“ do príkazu synchronized. Ten má nasledujúcu syntax:

```
synchronized ( výraz )
{
  telo (kritická sekcia)
}
```

Výsledkom *výrazu* musí byť nenulová referencia na existujúci objekt. Pred začatím vykonávania tela príkazu sa thread pokúsi získať zámok nad týmto objektom. Ako zamykací objekt použijeme niektorý z objektov, s ktorými pracujeme v kritickej sekcii, alebo si vypomôžeme pomocným objektom:

```
Object lock = new Object();
synchronized (lock)
{
  ...
}
```

Sémantika zamykania objektov v Jave dovoľuje threadu, ktorý vlastní zámok nad nejakým objektom, získať tento zámok ešte raz. To je dôležité napríklad v takomto prípade:

```
synchronized (lock)
{
  synchronized (lock)
  {
    // ...
  }
}
```

Thread sa nezasekne na druhom príkaze **synchronized**, ale plynule prejde do kritickej sekcie.

Dôležitá poznámka: Je nevyhnutné zabezpečiť, aby sa k premenným a objektom, s ktorými pracujeme v kritickej sekcii, dalo v programe pristupovať len cez príkazy synchronized, ktoré navyše musia pracovať s rovnakým zamykacím objektom. V opačnom prípade synchronizácia stráca zmysel.

Monitory

V paralelnom programovaní pojem *monitor* opisuje údajovú metaštruktúru, ktorú by sme mohli pripodobniť k javovskej triede: monitor má vnútorný stav (údajové členy) a poskytuje operácie, ktoré možno nad monitorom vykonávať (metódy). Podstatným rozdielom oproti triedam je zabezpečenie vzájomného vylučovania pri prístupe k monitoru. Inak povedané: z metód monitora môže byť súčasne volaná najviac jedna.

Implementácia monitorov v Jave je logickým dôsledkom použitia mechanizmu zámkov. Ak sa zamyslíme nad tým, ako by sa dalo vzájomné vylučovanie pri prístupe k metódam triedy realizovať, rýchlo prídeme na najjednoduchšiu možnosť: obaliť kód metódy do príkazu **synchronized** a ako zamykací objekt použiť **this**. Ide to však ešte jednoduchšie. Ak pri deklarácii metódy použijeme ako jeden z modifikátorov kľúčové slovo **synchronized**, dosiahneme tým, že pri volaní metódy sa aktuálny thread najprv pokúsi získať zámok nad objektom, nad ktorým metódu volá. V prípade, že ide o statickú metódu, v rámci ktorej objekt **this** neexistuje, ako zamykací objekt sa použije objekt typu **Class** reprezentujúci danú triedu.

Ako jednoduchý príklad monitora si ukážme triedu, ktorá vyrieši náš predchádzajúci problém s paralelnou inkrementáciou premennej **x**:

```
class MutExVar
{
  private int value;
  public MutExVar(int _val)
  { value = _val; }
  public synchronized void set(int _val)
  { value = _val; }
  public synchronized int get()
  { return value; }
  public synchronized void inc()
  { value ++; }
}
```

Oba thready budú namiesto príkazu **x = x + 1** v cykle volať metódu **x.inc()** (predpokladáme, že **x** je teraz inštanciou triedy **MutExVar**). Ďalšie synchronizované metódy **set()** a **get()** slúžia na nastavenie a získanie uloženej hodnoty. Konštruktor z pochopiteľných dôvodov nemusí byť synchronizovaný.

Problém s deadlockom

Pri práci so zámkami sa treba vyvarovať situácie, ktorá sa bežne označuje anglickým termínom *deadlock*, po slovensky uviaznutie. Okolo problému deadlocku existuje celá teória, takže len stručne: stav deadlocku znamená, že jeden alebo viacero threadov neobmedzene dlho čaká na vstup do kritickej sekcie (a nikdy sa do nej nedostane).

Ako môže k deadlocku dôjsť? Predstavme si dva thready, ktoré na vstup do kritickej sekcie potrebujú získať dva zámky, ale budú to robiť v navzájom opačnom poradí. Prvý thread napríklad takto:

```
synchronized (lock1)
{
  synchronized (lock2)
  {
    // ...
  }
}
```

a druhý takto:

```
synchronized (lock2)
{
  synchronized (lock1)
  {
    // ...
  }
}
```

Ak sa prvému threadu podarí získať lock1 a druhému **lock2**, v tom okamihu sú odsúdení na večné čakanie, pretože ani jeden nemôže získať druhý zámok (má ho ten druhý). Vyriešenie tohto konkrétneho problému je jednoduché – stačí zameniť poradie príkazov **synchronized** v niektorom z threadov. Univerzálny liek však na odstránenie nebezpečenstva deadlocku neexistuje a pri písaní programov treba používať predovšetkým zdravý rozum a riadnu dávku predstavivosti.

Podmienečné premenné

Java poskytuje ešte jeden synchronizačný prostriedok, ktorý sa najviac podobá konceptu *podmienečných premenných* (conditional variables). Tie si môžeme predstaviť ako metaobjekty, ktoré predstavujú synchronizačné body viazané na splnenie nejakej podmienky. Poskytujú dve operácie s tradičnými názvami ***wait*** a ***signal***, ktorým v Jave zodpovedajú metódy **wait()** a **notify()**. Ide o metódy triedy **Object**, a teda ich môžeme zavolať nad každou inštanciou objektového typu.

Aká je sémantika týchto dvoch operácií? Ak thread zistí, že podmienka, ktorá je pridružená k podmienečnej premennej, nie je splnená, vykoná operáciu ***wait***. Tým sa zaradí do zoznamu čakateľov na danú podmienečnú premennú, prejde do stavu „spiaci“ a naďalej nesúťaží o pridelenie procesora. Zo spánku ho môže zobudiť iný thread, ktorý nad podmienečnou premennou vykoná operáciu ***signal***. Dôsledkom tejto operácie je výber jedného z čakateľov a jeho zobudenie. Zobudený thread sa presúva do stavu „pripravený“ a čaká na pridelenie procesora. Ak je zoznam čakateľov prázdny, operácia signal nemá nijaký efekt.

Pozrime sa teraz na vec na úrovni zdrojového kódu. Operácii ***wait*** zodpovedá zavolanie metódy **wait()** nad ľubovoľným objektom (ktorý nahrádza spomínanú podmienečnú premennú). Aktuálny thread musí nad týmto objektom vlastniť zámok. V rámci vykonávania metódy je thread pozastavený a zámok sa mu odoberie (to je nevyhnutný krok, pretože inak by spiaci thread blokoval akýkoľvek prístup k synchronizačnému objektu). Pri volaní metódy **wait()** možno ako voliteľný argument zadať maximálny čas, počas ktorého thread bude čakať na zobudenie.

Druhej operácii ***signal*** zodpovedá metóda **notify()**, ktorú, pochopiteľne, voláme nad rovnakým objektom ako metódu **wait()**. Thread, ktorý metódu volá, tak obyčajne činí preto, lebo zabezpečil splnenie podmienky pridruženej k objektu. Ako dôsledok volania metódy notify() sa zobudí jeden z čakajúcich threadov (ktorý to bude, je vecou implementácie). Ak nikto nečaká, metóda je bez efektu. Niekedy príde vhod zobudiť *všetky* čakajúce thready – vtedy použijeme metódu **notifyAll()**, ktorá sa inak správa ako metóda **notify()**. Metódy **notify()** a **notifyAll()** by mal volať len thread, ktorý je momentálne vlastníkom zámku nad synchronizačným objektom.

Zobudený thread nezačne bežať okamžite. Je zaradený do radu threadov čakajúcich na naplánovanie a po pridelení procesora sa najprv snaží získať zámok, ktorý mu bol pred pozastavením odobraný. Po získaní zámku ukončí volanie metódy **wait()** a pokračuje nasledujúcim príkazom. V prípade, že thread volal metódu **wait()** ako dôsledok nesplnenia nejakej podmienky, je veľmi dôležité test podmienky *zopakovať*, pretože nie je zaručené, že sa od okamihu zobudenia threadu po jeho opätovné naplánovanie táto podmienka nezmenila. Namiesto kódu:

```
synchronized ( obj )
{
  if (! podmienka )
    obj.wait();
}
```

treba bezpodmienečne použiť kód:

```
synchronized ( obj )
{
  while (! podmienka )
    obj.wait();
}
```

A ešte jedna poznámka na záver: Threadu sa počas vykonávania metódy **wait()** odoberá ***len*** zámok na aktuálny objekt. Prípadné ďalšie zámky mu zostávajú a sú tak dobrým kandidátom na vznik deadlocku.

Príklady
Zdrojové texty k príkladom možno tradične nájsť na webovej stránke PC REVUE. Nachádza sa tam okrem iného riešenie klasického problému synchronizácie prístupu ku konečnému bufferu (inak známeho aj ako problém producentov/konzumentov).

18.časť: Dodatky

Posledná časť seriálu patrí niekoľkým témam, ktoré nie sú natoľko bežné (vnorené a anonymné triedy) alebo už nie sú také populárne (applety), bolo by však škoda nevenovať im aspoň niekoľko riadkov.

Applety
Applet je zdrobneninou slova *application* (teda nijaké „jabĺčko“, ako sa možno niekde dočítať). V súčasnosti applety ustupujú do pozadia, ale ešte nedávno boli nadmieru populárne ako súčasť webových stránok. Až natoľko, že to zaváňalo gýčom a zlým vkusom.

Applet je javovská aplikácia, ktorá nie je určená na samostatné fungovanie – na svoj beh potrebuje hosťovské prostredie, ktorým je typicky, ale nie výhradne internetový prehliadač. Na rozdiel od klasických aplikácií applet nemusí obsahovať metódu **main().** Jadrom appletu je trieda odvodená od štandardnej systémovej triedy **java.applet.Applet**. Trieda **Applet** je potomkom grafického komponentu **java.awt.Panel**, z čoho môžeme dedukovať, že applety disponujú grafickým rozhraním. S grafikou sa v rámci appletu pracuje rovnako ako pri iných komponentoch AWT: applet má definovaný manažér layoutu, možno doň pridávať iné komponenty (**java.awt.Panel** je kontajnerom), definovať obsluhu udalostí a pod. Na pamäti treba mať iba skutočnosť, že hlavné okno appletu, ako súčasť hosťovského prostredia, obyčajne nemôže meniť svoju polohu. Na zmenu veľkosti nájdeme v triede **Applet** metódu **resize()**, ktorá však nezaručuje, že túto zmenu hosťovské prostredie povolí.

Životný cyklus appletu možno rozdeliť do štyroch metód: **init(), start(), stop()** a **destroy()**. Prvú metódu, **init()**, volá prostredie po natiahnutí appletu do pamäte. Metóda je volaná práve raz a je vhodné do nej vložiť kód na inicializáciu appletu – alokáciu prostriedkov, vytvorenie pomocných objektov, threadov a pod. „Spustenie“ appletu má na starosti metóda **start()**, ktorú prostredie volá, keď je applet zobrazený a prístupný používateľovi. Ak sa applet dostane mimo zobrazenej plochy (napríklad odrolujeme webovú stránku) alebo je neprístupný iným spôsobom, prostredie zavolá metódu **stop()**. Logicky si domyslíme, že metódy **start()** a **stop()** sa najlepšie hodia na rozbehnutie a pozastavenie prípadných bežiacich threadov. Poslednú metódu **destroy()** volá prostredie pred odstránením appletu z pamäte (opäť práve raz). Treba však vziať do úvahy, že spôsob, ako a kedy hosťovské prostredie volá spomenuté metódy, je úplne otázkou jeho implementácie.

Applet disponuje metódami na zisťovanie informácií o prostredí, v ktorom beží. **Metóda getDocumentBase()** vracia URL adresu dokumentu, v ktorom je applet vložený. Iná metóda, **getCodeBase()**, vracia URL adresu samotného appletu (môže byť odlišná od dokumentu URL). Pomocou metódy **getParameter()** môže applet zisťovať svoje parametre, zadané ako súčasť dokumentu, v ktorom je applet vložený. Metódu **showStatus()** applet využije na zmenu textu v stavovom riadku hosťovského prostredia. Ďalšie dve metódy, **getImage()** a **getAudioClip()**, slúžia na natiahnutie obrázka alebo zvukového súboru zo zadanej adresy URL. Zvukový súbor možno prehrať pomocou metódy **play()**. Posledná metóda **getAppletContext()** vracia objekt implementujúci rozhranie **java.applet.AppletContext**, ktorého metódy slúžia na interakciu s hosťovským prostredím.

Ukážkový applet
Ukážme si príklad jednoduchého appletu, ktorý vo svojom okne zobrazí text zadaný pomocou parametra:

```
public class Display extends java.applet.Applet
{
  private final String defaultText = "Hello, world!";
  private String text;

  public void init()
  {
    text = getParameter("DisplayText");
    if (text == null)
      text = defaultText;
  }

  public void paint(java.awt.Graphics g)
  {
    setBackground(new java.awt.Color(0xFFECC0));
    g.drawString(text, 20, 30);
  }
}
```

Text, ktorý sa má zobraziť v okne appletu, zistíme pomocou metódy **getParameter()**. Vhodným miestom na jej použitie je metóda **init()**. Ak náhodou parameter nie je zadaný, metóda **getParameter()** vráti null a applet zobrazí preddefinovaný text „Hello, world!“.

Stránka s appletom môže vyzerať napríklad takto:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">
<html>
<head>
<title>Display Applet</title>
</head>
<body>

<p>Here comes the Display applet:</p>

<object classid="java:Display.class" width="140px"
height="60px">
<param name="DisplayText" value="Any other text">
</object>

</body>
</html>
```

V starších materiáloch sa možno stretnúť s vkladaním appletov do stránky pomocou elementu **APPLET**. Ten je už pekných pár rôčkov považovaný za zastaraný (deprecated) a podľa špecifikácie HTML 4.01 je na vkladanie objektov určený element **OBJECT**. Syntax tohto elementu nebudeme opisovať, informácie možno nájsť na adrese http://www.w3.org/TR/html401. Parameter appletu zadáme pomocou elementu PARAM.

Čitateľ sa môže pokúsiť prerobiť applet tak, aby aj farbu jeho pozadia bolo možné zadať pomocou parametra. Metóda **getParameter()** však vracia reťazec, ktorý je potrebné vhodne konvertovať, podľa použitého konštruktora triedy **Color**.

Obmedzenia appletov
Na applety sa vzťahujú určité obmedzenia. Applet napríklad nemôže vytvoriť sieťové spojenie s iným počítačom, než z ktorého bol natiahnutý jeho kód. V prípade, že applet používa ďalšie triedy, musia byť tieto triedy umiestnené na rovnakom serveri. Je, samozrejme, možné mať stránku a applet na rôznych serveroch – umiestnenie appletu potom treba spresniť pomocou atribútu **codebase** elementu **OBJECT**. Ak sa počas behu appletu používa viac ako jedna trieda, je vhodné celý balík transformovať do archívu JAR a v stránke sa odvolávať na tento archív (ušetríme dačo z prenosovej kapacity, pretože sa bude sťahovať menší objem dát).

Pri ladení appletov je dobré mať na pamäti, že prehliadače si natiahnutý kód ukladajú do pamäte cache a po opätovnom preklade zdrojového textu triedy obyčajne nestačí znovu natiahnuť stránku s appletom; treba prehliadač ukončiť a znovu spustiť.

Statické inicializátory
Za skonštruovanie inštancie triedy je zodpovedná špeciálna metóda, konštruktor. Kto sa však postará o inicializáciu statických zložiek triedy? Dosiaľ sme mlčky predpokladali, že tieto zložky inicializujeme triviálnym spôsobom ako súčasť deklarácie:

```
static int x = 1;
```

Niekedy však jeden riadok kódu nestačí: potrebovali by sme nejakú analógiu konštruktora, ale pre statické členy. Takýto „statický konštruktor“ nazývame statickým inicializátorom. V deklarácii triedy sa môže *statický inicializátor* nachádzať aj viackrát a poznáme ho podľa kľúčového slova **static** nasledovaného zloženým príkazom:

```
static
{
  // inicializácia...
}
```

Kód statických inicializátorov sa nachádza na rovnakej deklaračnej úrovni ako kód ostatných metód. Aktivuje sa po natiahnutí triedy do pamäte pri prvom pokuse o vytvorenie inštancie triedy. Ak trieda obsahuje viac statických inicializátorov, ich kódy sa vykonajú sekvenčne, v takom poradí, v akom sa inicializátory nachádzajú v zdrojovom texte.

Statické vnorené triedy
Až do tejto chvíle sme v príkladoch pracovali s triedami, ktoré boli deklarované na súborovej (najvyššej) úrovni. V ďalšom texte ich budeme označovať ako top-level triedy. Java však dovoľuje používať aj vnorené triedy (nested), deklarované v rámci inej triedy alebo dokonca v rámci metódy.

Pri triedach deklarovaných v rámci inej triedy hrá rolu prítomnosť či neprítomnosť modifikátora **static**. Vnorené triedy, deklarované ako statické, sa v zásade nelíšia od top-level tried. Používame ich rovnakým spôsobom a na ich obsah sa nevzťahujú nijaké obmedzenia. Jediným rozdielom je ich plne kvalifikovaný názov: medzi „balíkovú“ identifikáciu a názov vnorenej triedy musíme doplniť ešte názov tzv. obklopujúcej (enclosing) triedy, t. j. triedy, v ktorej je vnorená trieda deklarovaná. Príklad:

```
package my.pkg;
class TopLevel
{
  // ...
  public static class Nested
  { ... }
}
```

Vnorená trieda **Nested** je súčasťou obklopujúcej triedy **TopLevel**. Plne kvalifikovaný názov vnorenej triedy je **my.pkg.TopLevel.Nested**. Ak vhodne použijeme príkaz **import** alebo sa pohybujeme v rámci implicitného balíka, úplný názov triedy je **TopLevel.Nested**.

Statická vnorená trieda nesmie byť deklarovaná inde ako v top-level triede. Iné triedy môžu k vnorenej triede pristupovať podľa toho, ktorý modifikátor prístupu (**public, protected, private**) pri deklarácii tejto triedy použijeme. Obklopujúca trieda má k vnorenej triede vždy plný prístup. Statická vnorená trieda môže pristupovať len k statickým premenným a metódam obklopujúcej triedy.

Členské triedy
Vnorené triedy, deklarované bez kľúčového slova **static**, sa nazývajú *členskými triedami*. Členské triedy majú na rozdiel od statických vnorených tried prístup ku všetkým členom a metódam obklopujúcej triedy. To je možné vďaka obmedzeniu, podľa ktorého každá inštancia členskej triedy je zviazaná práve s jednou inštanciou obklopujúcej triedy (tzv. obklopujúcou inštanciou). Pri vytváraní inštancie členskej triedy dostane jej konštruktor refe-

renciu na obklopujúcu inštanciu prostredníctvom skrytého parametra. Logicky môžeme usúdiť, že vytvorenie inštancie členskej triedy je možné len v niektorej z metód obklopujúcej triedy.

Prístup k členským triedam „zvonka“ je riadený podobne ako v predchádzajúcom prípade modifikátormi prístupu. V rámci členských tried nemožno deklarovať statické premenné, metódy ani inicializátory, s výnimkou statických konštánt (deklarovaných s modifikátorom **final**).

Lokálne triedy

Deklarácia vnorenej triedy môže byť takisto súčasťou tela metódy (nazvime ju obklopujúcou metódou). Takejto triede hovoríme *lokálna*. Inštancia lokálnej triedy je viditeľná len v rámci tela obklopujúcej metódy. Lokálna trieda má prístup ku všetkým premenným a metódam obklopujúcej triedy (rovnaký princíp ako pri členských metódach). Okrem toho môže lokálna trieda pristupovať k lokálnym premenným a parametrom obklopujúcej metódy za podmienky, že sú tieto premenné či parametre deklarované s modifikátorom **final**. Doba platnosti lokálnych premenných sa predlžuje na neurčito, pokiaľ existuje inštancia lokálnej triedy, ktorá sa na ne odvoláva.

Lokálne triedy možno vhodne použiť pri obsluhe udalostí ako tzv. adaptérové triedy. Vráťme sa o jedno či dve čísla časopisu naspäť a spomeňme si, že komponenty GUI umožňujú iným objektom „prihlásiť sa“ na odber oznámení o výskyte udalostí. Interakcia je zabezpečená pomocou listenerov, čo sú špeciálne objekty implementujúce vybrané rozhranie. Komponent GUI pri výskyte „svojej“ udalosti postupne zavolá príslušné metódy všetkých prihlásených listenerov. Objekty pracujúce v úlohe listenerov sú obyčajne inštanciami vnorených tried: členských, lokálnych či anonymných (pozri ďalej).

Príklad použitia lokálnej triedy pri reakcii na stlačenie tlačidla:

```
public void setUpButton()
{
  final Button b;
  b = new Button("Any text");
  b.addActionListener(new ButtonListener());
  add(b);

  class ButtonListener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    { b.setLabel("Clicked!"); }
  }
}
```

Metóda **setUpButton()** slúži na vytvorenie tlačidla (trieda Button) a jeho pridanie do kontajnera (metóda **add()**; treba si predstaviť, že metóda **setUpButton()** je súčasťou objektu kontajnera). Pomocou metódy **addActionListener()** pridáme k tlačidlu listener – inštanciu lokálnej triedy **ButtonListener**. V rámci obsluhy udalosti sa vyvolá metóda **actionPerformed()** tejto triedy. Metóda má prístup k objektu tlačidla pomocou finálnej premennej **b**, čo demonštrujeme zmenou textu tlačidla.

Anonymné triedy

Variáciou na lokálne triedy sú *anonymné* triedy. Nemajú vlastný názov, nesmú obsahovať konštruktor a sú určené na ad hoc použitie spájajúce deklaráciu s vytvorením inštancie. Deklarácia anonymnej triedy nahradzuje názov typu vo výraze pre alokáciu nového objektu. Predchádzajúci príklad s využitím anonymnej triedy bude vyzerať takto:

```
public void setUpButton()
{
  final Button b;
  b = new Button("Any text");
  b.addActionListener(new ActionListener()
  {
    public void actionPerformed(ActionEvent e)
    { b.setLabel("Clicked!"); }
  });
  add(b);
}
```

Všimnime si, že za kľúčovým slovom **new** sa nachádza názov rozhrania **ActionListener**. Anonymná trieda bude potomkom triedy **Object** implementujúcim toto rozhranie. Ak namiesto rozhrania uvedieme triedu, anonymná trieda bude potomkom tejto triedy.

Implementácia vnorených tried

Binárny kód javovských tried je uložený v súboroch s príponou **.class**. Názov každého súboru je (bez prípony) totožný s názvom príslušnej triedy. Pre vnorené triedy však tento mechanizmus treba upraviť, pretože dve vnorené triedy v rámci jedného balíka sa môžu volať rovnako, ak sú deklarované v rôznych triedach.

Riešenie je jednoduché: názov súboru vznikne spojením mena obklopujúcej triedy a vnorenej triedy. Obe mená sú oddelené znakom **$**. Trieda **TopLevel.Nested** z vyššie uvedeného odseku bude napríklad uložená v súbore s názvom **TopLevel$Nested.class**. Rovnaké pravidlo platí pre triedy členské i lokálne.

Malý háčik predstavujú anonymné triedy: tie nemajú meno, preto ich prekladač pri preklade očísluje vo vzostupnom poradí od jednotky a tieto čísla použije ako „názvy“ tried na účely vytvorenia mena binárneho súboru. Ak teda budeme mať napríklad triedu **MyClass** a v nej dve anonymné triedy, prekladač vygeneruje tri binárne súbory:

```
MyClass.class
MyClass$1.class
MyClass$2.class
```

Ak je vnorená trieda súčasťou inej vnorenej triedy, uvedené pravidlo sa uplatní rekurzívne. Môžeme sa potom stretnúť hoci so súborom **Outer$FirstLevel$SecondLevel$Inner.class**.

Koniec

Seriál je na konci. V Jave ako jazyku toho veľa nezostáva, a ak, tak len najpokročilejšie témy, v praxi málokedy používané. Väčší priestor ponúka Java ako platforma - to sme si však vysvetlili na úvod predošlej časti: opis ďalších balíkov by zabral priveľa miesta a sotva by bol dostatočne užitočný, azda len ako slovenský preklad originálnej dokumentácie. Zostáva teda veriť, že čitateľ za svoje peniaze dostal adekvátnu protihodnotu a že programátorov, ktorí sa vedia orientovať v Jave, je vďaka seriálu o niečo viac.

Vladimír Klimovský