

Centrum VTI SR, Bratislava

A-59 6037



Miroslav Virius

II. upravené
vydání

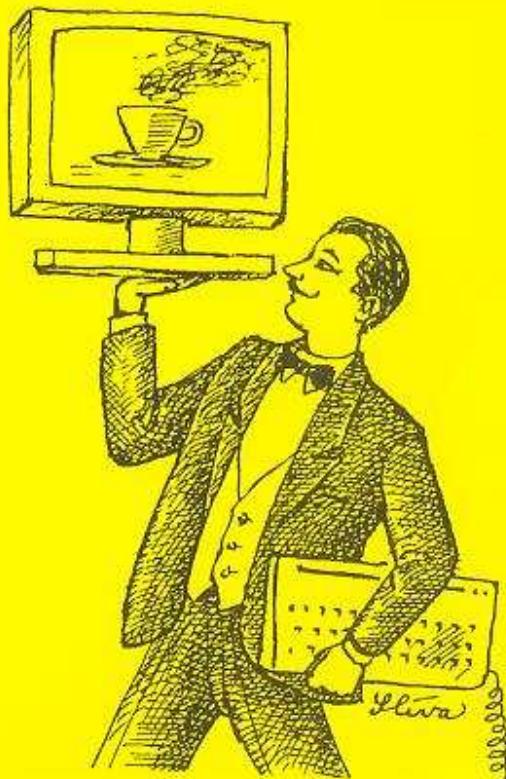
Java

pro zelenáče



- ❖ nenásilný úvod do objektově orientovaného programování
- ❖ moderní programovací jazyk
- ❖ základní programátorské techniky
- ❖ úvod do programování grafického uživatelského rozhraní
- ❖ Java 5 i předchozí verze

NEOCortex®
NAKLADATELSTVÍ A VÝDAVATELSTVÍ





V edici pro zelenáče vyšlo --

Pavel Satrapa **PERL PRO ZELENÁČE** 224 stran, cena 99,- Kč

- ❖ naučte se programovat v Perlu
- ❖ od úplných začátků až k pokročilým technologiím
- ❖ ochočte si regulární výrazy

Pavel Satrapa **PASCAL PRO ZELENÁČE** 256 stran, cena 199,- Kč

- ❖ již třetí vydání tohoto úspěšného titulu
- ❖ naučte se efektivně programovat a používat základní programátorské techniky

Miroslav Virius **C# PRO ZELENÁČE** 256 stran, cena 99,- Kč

- ❖ nenáročný úvod do objektově orientovaného programování
- ❖ moderní programovací jazyk a základní programátorské techniky
- ❖ úvod do programování grafického uživatelského rozhraní

Petr Šaloun **JAZYK C PRO ZELENÁČE** 220 stran, cena 199,- Kč

- ❖ programujte bez ohledu na systém a procesor
- ❖ rychlé a přenositelné programy, jednoduché filtry i rozsáhlé aplikace
- ❖ používejte využitelné programovací jazyky

J. Klán, J. Jindřich **WWW PRO ZELENÁČE** 318 stran, cena 99,- Kč

- ❖ získejte komplexní webovou gramotnost
- ❖ používejte nový XHTML jazyk pro tvorbu WWW stránek
- ❖ oživte stránky pomocí JAVA skriptu, zpracujte data v PHP a MySQL

P. Satrapa **OPENOFFICE.ORG PRO ZELENÁČE** 224 stran, cena 99,- Kč

- ❖ vytvářejte a formátujte dokumenty s obrázky, provádějte výpočty a prezentujte jejich hodnoty, připravujte atraktivní prezentace, kreslete snadno obrázky i schématika
- ❖ využívejte kompatibilitu s programy Microsoft Office na platformách MS Windows a Linux

Martin Snižek **CSS PRO ZELENÁČE** 296 stran, cena 199,- Kč

- ❖ srozumitelně a jasně popisuje CSS, jazyk k navrhování vzhledu webových stránek
- ❖ mnoho příkladů, ukázek a vysvětlujících obrázků
- ❖ přehledné uspořádání a výklad i pro uplného začátečníka

Vydání knihy je možno objednat telefonicky,
nebo přes internet na dobríku s 15% slevou.
Jinak se ptejte u dobrého knihkupce.

Neocortex, s. r. o.

Na Rovnosti 3, 130 00 Praha

tel.: 284 860 682, fax: 284 860 117

e-mail: info@neo.cz, internet: <http://www.neo.cz>

Java

pro zelenáče

II. upravené vydání

Miroslav Virius

Obsah

Úvod	4
1 Nad šálkem kávy	7
1.1 Počítač.....	7
Operační paměť	8
1.2 Datové typy a proměnné	8
1.3 Programy a programovací jazyky	9
Operační systém	11
1.4 Program a algoritmus	11
1.5 Objekty a třídy	14
Zapouzdření	14
Dědění	17
Polymorfismus	19
Dědění versus skládání	20
1.6 Java neboli kafe	21
Kde ji získat	21
Třídy v Javě	23
Java a C++	23
2 První program	25
2.1 Ahoj, lidi	25
Zdrojový text	25
Překlad	26
Běh programu	27
2.2 Co jsme naprogramovali	29
Komentář	29
Deklarace třídy	30
Metoda main()	30
Doplňující poznámky	31
2.3 Vlastní dokumentace	32
3 Jednoduché příklady	34
3.1 Měníme svůj první program	34
Hry s texty	34
3.2 Trocha počítání	36
3.3 Vstup dat	38
3.4 Faktoriál	40
Metoda faktorial()	46
Magická čísla	49
3.5 Pozdrav z Marsu	49
Jak je to s texty	55
4 Složitější příklad	57
4.1 Počítání slov poprvé	57
Třída Analyzer	57
Běh programu: výjimky	57

Kontejner	59
Analýza řádku	60
4.2 Počítání slov podruhé	65
4.3 Počítání slov pořetí, tentokrát v JDK 5	68
Zjednodušujeme čtení	69
Generické třídy	70
4.4 Počítání slov počtvrté	73
Jednosměrně zřetězený seznam	73
Prvek seznamu	73
Implementace seznamu	74
5 Začínáme naostro	77
5.1 Základní pojmy	78
Komentář	78
Identifikátor	78
Klíčová slova	80
Zápis programu	80
5.2 Organizace zdrojového kódu: balíky	81
Jména balíků	81
Deklarace a používání balíku	82
Statický import	84
6 Datové typy, proměnné	86
6.1 Primitivní datové typy	86
Celá čísla	86
Znaky	92
Reálná čísla	94
Logické hodnoty	97
„Prázdný“ typ void	98
Balení a vybalování	98
6.2 Skupina proměnných: pole	100
Deklarace a vytvoření pole	100
Inicializace	102
Kopírování polí	103
Vícerozměrná pole	104
Inicializace vícerozměrných polí	105
Kopírování vícerozměrných polí	105
6.3 Objekty	106
6.4 Výčtové typy	106
Pohled pod pokličku	107
6.5 Automatická správa paměti	108
6.6 Proměnné	109
Deklarace	109
Neinicializované proměnné	110
7 Výrazy a operátory	111
7.1 Vlastnosti operátorů	111
7.2 Aritmetické výrazy	112
Smíšené výrazy	112
7.3 Relační výrazy	114

7.4 Logické výrazy	114
7.5 Přehled operátorů	114
Podmíněný výraz	116
Konverze a přetypování	116
Operátor čárka	117
Přiřazovací operátor	117
8 Příkazy	119
8.1 Blok (složený příkaz)	119
Vnořené bloky	120
8.2 Výrazový příkaz	120
Prázdný příkaz	120
8.3 Rozhodování	121
Příkaz if	121
Příkaz switch (přepínač)	122
Příkaz break	124
8.4 Cykly	125
Příkaz while	126
Příkaz do-while	126
Cyklus for	128
Příkaz for v Javě 5	130
Příkaz break	131
Příkaz continue	132
8.5 Přenos řízení	133
Příkaz return	133
8.6 Aserce (příkaz assert)	133
Poznámky	135
9 Třídy a objekty	137
9.1 Deklarace třídy	137
Modifikátory v deklaraci třídy	138
9.2 Tělo třídy	139
Přístupová práva	139
9.3 Datové složky	141
Nestatické datové zložky	141
Statické datové složky	142
9.4 Metody	143
Deklarace metody	144
Parametry metod	145
Nestatické metody	147
Statické metody	148
Lokální proměnné	149
Třída Bod	150
Rekurze	151
Metoda main()	152
this	153
Konstruktory	154
Metody s proměnným počtem parametrů	155
9.5 Dědění	156

Předefinované metody	157
super	157
Konstruktor potomka	158
Příklad: grafické objekty	158
Abstraktní metody, abstraktní třídy	164
Finální třídy, finální metody	165
10 Výjimky	166
10.1 Výjimka v Javě.....	167
Třídy pro přenos informací o výjimek	167
Vznik výjimky	168
10.2 Ošetřování výjimek	170
Handler	170
Koncovka.....	175
10.3 Výjimky a metody	177
11 Rozhraní.....	179
11.1 Deklarace a implementace rozhraní	180
Deklarace rozhraní.....	180
Implementace rozhraní	181
11.2 Použití rozhraní	181
Příklad: Kontrola konzistence objektů.....	182
Klonování objektů	185
Anonymní třída implementující rozhraní.....	188
12 Parametrizované třídy a metody	189
12.1 Parametrizované třídy	189
12.2 Parametrizovaná rozhraní.....	191
12.3 Parametrizované metody	191
12.4 Meze typových parametrů.....	193
12.5 Pohled pod pokličku	194
Omezení.....	195
Dědění a parametrizované typy	196
12.6 Zástupný typ (žolík)	196
13 Soubory, vstupy a výstupy	198
13.1 Soubor a jeho vlastnosti	200
13.2 Čtení a zápis binárních dat	201
Čtení a zápis primativních typů.....	202
Přenositelnost a nepřenositelnost binárních souborů	204
Ukládání objektů (serializace)	204
13.3 Čtení a zápis znakových dat	207
Zápis do textového souboru.....	207
Čtení z textového souboru	209
Čeština v Javě pod Windows	213
14 Grafické uživatelské rozhraní	216
14.1 První okno	216
Správné ukončení programu	216
Vlastnosti komponent	219
Velikost a umístění okna	220
14.2 Používání komponent.....	224

Vkládáme komponenty do okna	224
Učíme tlačítko fungovat	229
Událostí trochu podrobněji	230
14.3 Piškorky, verze 1.0	232
Požadavky	233
Komponenty	234
Program	235
14.4 Piškorky, verze 2.0	240
Přidáváme nabídku	241
Ukončení programu	242
Zpět	244
Nová hra	245
Okno O programu	245
Volby	246
14.5 Piškorky, verze 2.1	251
A co dál	253
15 Aplet	254
15.1 Struktura apletu	254
Metody apletu	254
Příklad	256
Aplet spuštěný jako aplikace	259
15.2 Spuštění apletu	260
Literatura	262
Rejstřík	263

Úvod

Otevříte knihu, která vás provede základy dnes velice populárního programovacího jazyka Java, a včetně to jeho v současné době nejnovější verze 5. Je určena naprostým začátečníkům; nepředpokládám, že byste měli jakékoli předběžné znalosti o programování.

Jediné, co očekávám, je, že máte chuť se to naučit – a samozřejmě, že umíte s počítadlem alespoň trochu „uživatelsky“ zacházet. To znamená, že umíte spouštět programy z grafického rozhraní a z příkazové řádky.

Co v této knize najdete

Na počátku, v první kapitole, se seznámíme s pojmy, které by měl programátor znát. Povíme si krátce o počítači a jeho součástech, o algoritmech a samozřejmě také o třídách a objektech. Java je totiž tzv. objektově orientovaný jazyk, a proto se s objekty budeme setkávat od samého počátku. Nakonec si řekneme i několik slov o historii Javy.

Ve druhé kapitole si ukážeme, jak napsat jednoduchý program v Javě. Naučíme se vytvořit jeho zdrojový text, přeložit ho, spustit a vytvořit k němu dokumentaci. Zde nám ještě o vlastní programování nepůjde, i když si význam tohoto programu alespoň zhruba vysvětlíme. Ukážeme si i nejběžnější chyby, které můžeme při překladu a spouštění programu udělat.

V dalších dvou kapitolách si ukážeme řadu jednoduchých programů, na kterých se seznámíme se základními konstrukcemi Javy. Výklad v nich není úplný, jeho cílem je vytvořit rámec minimálních znalostí, které nám umožní ilustrovat další výklad alespoň trochu smysluplnými příklady.

Od páté kapitoly začíná výklad Javy „naostro“. Postupně se seznámíme s primitivními datovými typy, s výrazy a příkazy. V deváté kapitole se naučíme vytvářet a používat objektové typy, pak přijdou na řadu tzv. výjimky (způsob ošetřování chyb v programu), rozhraní, generické typy – ti je novinka Javy 5 – a vstupní a výstupní operace.

V posledních dvou kapitolách najdete úvod do programování grafického uživatelského rozhraní programů. Naučíte se vytvářet okna, zpracovávat události, používat předdefinované komponenty JavaBeans. Seznámíte se i se základy tvorby appletů.

Jen JDK

I když to dnes, v době grafických uživatelských rozhraní, vypadá neobvykle, předpokládám, že máte k dispozici pouze JDK, tedy základní nástroje pro práci s Javou, které se spouští z příkazové řádky. (V první kapitole se dočtete, kde je můžete získat.)

K Javě samozřejmě existuje řada vizuálních vývojových prostředí, o žádném z nich ale v této knize nehovořím. Mám pro to dva důvody:

- Popis kteréhokoli z těchto prostředí vydá na samostatnou knihu. Navíc bychom se museli rozhodnout pro jedno konkrétní prostředí a ostatní bychom museli ponechat stranou.
- Tato prostředí automaticky vytvářejí zdrojový kód programů. To je sice skvělé při rutinní práci, pro výuku to ale není to nejlepší. Budete-li na počátku vše psát sami znak po znaku, pochopíte daleko snáze, co váš program dělá – a to je hlavní cíl této knihy.

Co v této knize nenajdete

Především mějte prosím na paměti, že jde o knihu pro zelenáče. To znamená, že nemůže obsahovat vše, co lze o Javě napsat. O řadě témat jsem se nezmínil, protože na ně nezbýlo místo, o jiných proto, že mi připadají pro začátečníka zbytečně komplikovaná (a přitom bez jejich znalosti lze také dobře programovat).

Z konstrukcí jazyka Java jsem vynechal např. vnitřní třídy, reflexi nebo vícevláknové (multithreadové) programování. Nenajdete tu také povídání o programování databázových aplikací, o volání vzdálených metod (RMI), o programování distribuovaných aplikací podle standardu CORBA nebo jiných nebo používání metod napsaných v jiném programovacím jazyce.

Také informace o knihovnách třídách Javy by vydaly na samostatnou knihu, a nijak tenkou. V této knize najdete jen stručné poučení o třídě `ArrayList` a iterátoru na něm, o ostatních kontejnerech se nedozvítíte, než že existují. Pokud jde o vstupy a výstupy, naučíme se pracovat se soubory, pomineme ale operace s rourami (pipe) a jinými prostředky, které lze pomocí této knihovny také používat.

Typografické konvence

identifikátor

Kurziva znamená zdůraznění. Píšeme tak také nově zaváděné terminy. V popisech syntaxe jazyka Java označuje pojmy, které je třeba ještě dále vyjasnit (tzv. neterminální symboly).

- while** Tučně pišeme v popisu syntaxe symboly, které lze přímo opsat do programu (tzv. terminální symboly).
- ArrayList** Neproporcionálním písmem pišeme ukázky programů, výstupy z počítače a také klíčová slova, identifikátory ap.
- ◆ Tento symbol označuje konec příkladu.
-  Ručička upozorňuje na místa, která si zaslouží zvýšenou pozornost.
-  Pavouk upozorňuje na informace, které lze najít na internetu. Zpravidla jde o zdrojové soubory příkladů z této knihy.
-  Tato ikona upozorňuje na poznámky určené především čtenářům, kteří znají programovací jazyk C nebo C++.
- 4, 5** Tyto ikony upozorňují na vlastnosti, které jsou k dispozici počinaje JDK 1.4, resp. JDK 5, tedy které nenajdete v předchozích verzích.
-  Mouha upozorňuje na možné programátorské chyby a na obtížná a problematická místa.

Zdrojové texty příkladů

Úplné zdrojové texty všech příkladů si můžete stáhnout z internetové adresy www.neo.cz. Jsou komprimovány programem PKZIP, a to včetně adresářové struktury. V textu knihy se na ně odolávám slovy „na WWW“; pokud v tomto odkazu uvádím adresář, jde o adresář obsažený v komprimovaném souboru.

Na závěr

Rád bych poděkoval všem, kteří svými radami a připomínkami přispěli ke zdárnému dokončení této knihy, zejména pak lektorovi Doc. Ing. Pavlu Heroutovi, Ph.D., ze Západočeské univerzity v Plzni, který ji pečlivě přečetl a měl k ní řadu cenných připominek.

Jsem si jist, že přes veškerou péči, kterou jsem této knize věnoval, v ní budou chyby. Za ně samozřejmě nesu veškerou odpovědnost já sám. Pokud na nějakou přijdete, pošlete mi prosím upozornění na níže uvedenou adresu; na internetové stránce <http://km1.fjfi.cvut.cz/~virius/errata.htm> uveřejním opravu.

Miroslav Virius
Katedra softwarového inženýrství
FJFI ČVUT
virius@km1.fjfi.cvut.cz

1 Nad šálkem kávy

Než se pustíme do programovacího jazyka Java, musíme si leccos povídět. V této kapitole se seznámíme s řadou potřebných pojmu - od počítače přes programovací jazyky a algoritmy až po objektově orientované programování. Některé z nich nepochybňně znáte; přesto vás prosím, abyste si toto povídání alespoň zběžně přečetli, abychom se mohli shodnout na terminologii.

1.1 Počítač

Pro účely daňového přiznání se počítač označuje za „stroj na zpracování informací“. To je sice pravdivé, ale programátor, a to i začínající, o něm musí přece jen vědět víc. Následující povídání bude samozřejmě silně zjednodušené, pro nás ale naprosto postačující.

Obvykle se říká, že počítač obsahuje čtyři základní části:

- *Procesor* je součást, která opravdu „počítá“, tedy zpracovává informace. Zároveň také řídí činnost všech ostatních částí.
- *Operační paměť* slouží k ukládání dat (informací), která počítač zpracovává, a programu, tedy příkazů, které určují, co má dělat. Vše, co je v této paměti, se při vypnutí počítače ztratí, „zapomene“. Používá se pro ni také anglická zkratka RAM (Random Access Memory, paměť s náhodným přístupem).
- *Vstupní a výstupní zařízení* slouží k výměně informací s okolím. (Sebelepší počítač by nám nebyl nic platný, kdyby nám nemohl předat výsledky své práce.) Typickými příklady vstupních zařízení jsou klávesnice, myš, skener atd. Typickým příkladem výstupních zařízení může být obrazovka monitoru nebo tiskárna. Pro vstupní a výstupní zařízení se používá zkratka V/V nebo I/O (z anglického input/output).
- *Vnější (trvalá) paměť* slouží k trvalému ukládání dat a programů. Data v ní se při vypnutí počítače neztrácejí, má zpravidla mnohonásobně větší kapacitu než operační paměť, ale práce s ní je mnohonásobně pomalejší než práce s operační pamětí. Jako vnější paměť se používají převážně magnetické disky (pevný disk, disketa). Dnes se často používá i kompaktní disk (CD).

Operační paměť

- Bity Základem operační paměti jsou elektronické obvody, které mohou mít dva stavů – např. vypnuto nebo zapnuto. Jeden z těchto stavů obvykle odpovídá číslici 0, druhý číslici 1. Údaje do paměti proto můžeme zapisovat jen pomocí nul a jedniček, v tzv. dvojkové soustavě. Každé místo, na které můžeme zapsat jednu číslici 0 nebo 1, označujeme jako *bit*. Operační paměť je tedy dlouhá řada bitů.
- Bajty Ovšem práce s jednotlivými bity je nepohodlná, a proto se bity sdružují do větších celků. V dnešních počítačích se téměř bez výjimky používají skupiny velikosti 8 bitů, které se nazývají *bajty* (anglicky byte, tj. slabika).
- Adresa Jednotlivé bajty, tvořící operační paměť, jsou očíslovány. Počáteční bajt má číslo 0, následující má číslo 1 atd. Toto pořadové číslo se nazývá *adresa* bajtu. (Na některých počítačích, např. na PC, je záležitost s adresami trochu složitější, ale to nás v souvislosti s Java vůbec nebude zajímat.)

1.2 Datové typy a proměnné

Snadno zjistíme, že nejmenší číslo, které může jeden bajt obsahovat, se skládá z osmi nul a představuje i v desítkové soustavě nulu. Největší takové číslo se bude skládat z osmi jedniček a v desítkové soustavě představuje 255. To je samozřejmě málo – s počítačem, který by znal jen čísla od 0 do 255, bychom si nedokázali ani přepočítat výplatu. Proto se pro ukládání dat obvykle používají různé velké skupiny za sebou následujících bajtů.

Ani to ovšem nestačí. Snadno se přesvědčíme, že kdybychom vzali např. skupinu dvou za sebou následujících bajtů, mohli bychom do ní uložit celá čísla v rozmezí od 0 do 65 535. Ale co když budeme potřebovat záporná čísla? Co když budeme potřebovat reálná čísla? Co když budeme chtít vyjádřit znaky nebo logickou hodnotu (nějaké tvrzení platí nebo neplatí)?

Musíme tedy zajistit nějaký způsob, který nám umožní reprezentovat data různých „druhů“ v paměti počítače. Jinými slovy, musíme najít způsob, jakým určité skupiny bitů přiřadíme hodnotu, kterou tato skupina představuje – jak ji v počítači *zakódovat*.

- Datový typ Můžeme se např. dohodnout, že bajt s hodnotou 01000001 bude představovat znak 'A'. Táž skupina bitů může za jiných okolností ovšem také představovat celé číslo, které v desítkové soustavě má hodnotu 65. Týž bajt ale může být i součástí většího celku s jiným významem.

Předchozí příklady ukazují, že pracuje-li počítač s nějakým kouskem paměti, s nějakou skupinou bajtů na určité adrese, musí vědět, jak je tato skupina velká a jak má její obsah interpretovat. Jinými slovy, musí znát *datový typ* hodnoty, která je tam uložena, musí vědět, zda jde o celé číslo, znak, logickou hodnotu atd.

Podle datového typu se také budou lišit operace, které lze s danou hodnotou provádět. Celá čísla lze například sčítat a odečítat, znaky lze spojovat do řetězců, tedy do souvislého textu.

Proměnná Nyní se na celý problém podivejme trochu jinak. Už víme, že když budeme chtít pracovat s nějakou hodnotou, musíme si ji uložit do paměti. To ale znamená, že si tam pro ni musíme vyhradit místo a říci, jakého typu budou údaje, které bude obsahovat. Takovéto místo pro ukládání hodnoty budeme nazývat *proměnná*.

Abychom s proměnnou mohli zacházet, musíme ji pojmenovat, musíme ji dát *identifikátor*. Tomu se v programování říká *deklarace* proměnné.¹

1.3 Programy a programovací jazyky

Aby mohl počítač nějak zpracovávat data, která mu dáme, musíme mu také říci, co má vlastně dělat – musíme mu dát *program*.

Procesor umí s daty řadu operací, ovšem velice jednoduchých. Lze mu například říci „vezmi celé číslo, která je na adrese 6548, a přiříti k němu celé číslo z adresy 7895“. Protože ovšem do jeho paměti nelze uložit nic jiného než čísla, musí být tyto příkazy vyjádřeny – zakódovány – také čísky. Toto číselné vyjádření instrukcí (příkazů) pro procesor se nazývá *strojový kód* a je to jediná věc, které procesor rozumí. Jedním z problémů je, že různé druhy počítačů používají různé strojové kódy, takže programy ve strojovém kódu nejsou přenositelné mezi počítače s různými procesory.

Vyšší programovací jazyky Jiným – a možná horším – problémem je, že programování ve strojovém kódu je velice namáhavé a nepřehledné. Také to téměř nikdo nedělá; místo toho se používají tzv. *vyšší programovací jazyky* – Pascal, C, C++, Basic, ..., a také Java.

Zápis programu ve vyšším programovacím jazyku se zpravidla skládá z vybraných anglických slov a případně z matematických výrazů zapsaných podobně jako v matematice. Programování ve vyšším jazyku je pochopitelně daleko jednodušší než programování ve strojo-

¹ Později uvidíme, že identifikátor – tedy jméno – mohou mít i jiné části programu, nejen proměnné.

vém kódu. Je tu ovšem jeden háček: Takovýto program nelze přímo spustit, neboť počítač mu nerozumí. Program ve vyšším programovacím jazyku se proto musí buď *přeložit* do strojového kódu nebo *interpretovat*. V obou případech k tomu potřebujeme další program (nebo skupinu programů), které to za nás udělají.

Překlad	Zápis programu ve vyšším programovacím jazyku (přesněji textový soubor, který tento zápis obsahuje) se zpravidla označuje jako <i>zdrojový kód</i> nebo <i>zdrojový program</i> . O jeho překlad do strojového kódu (<i>kompilaci</i>) se postará program zvaný <i>překladač</i> neboli <i>kompilátor</i> . V mnoha případech s ním spolupracuje ještě <i>sestavovací program</i> neboli <i>linker</i> , který může spojit několik nezávisle přeložených částí programu do jednoho celku. Linker také připojí knihovny – části programu, které už někdo naprogramoval předem a které můžeme už jen používat.	
Překladem a sestavením programu vznikne soubor, obsahující strojový kód, který lze na cílovém počítači rovnou spustit. Mezi často používané překládané programovací jazyky patří např. C, C++ nebo Pascal.		
Interpretace	Můžeme však také použít speciální program, který bude číst zdrojový text a provádět příkazy, které v něm najde – interpretovat ho. Typickým interpretovaným jazykem je <i>klasický Basic</i> . ²	
Java	Interpretované programy zpravidla běží výrazně pomaleji než překládané programy. Navíc musíme na cílový počítač spolu s naším programem dodat také interpretační program.	
Java	Java stojí někde mezi interpretovanými a překládanými jazyky. Zdrojový text se sice překládá, nikoli však do strojového kódu určitého počítače, ale do jakéhosi univerzálního jazyka, který se nazývá <i>bajtový kód</i> (byte code). Tento bajtový kód se pak interpretuje programem zvaným <i>javský virtuální stroj</i> (JVM, Java Virtual Machine). Výsledkem je na jedné straně přece jen rychlejší běh programu než u „čistě interpretovaných“ jazyků a na druhé straně možnost přenosu přeloženého programu na libovolný počítač, na němž je instalován JVM.	
	Dodejme, že překladač libovolného programovacího jazyka (i Javy) zároveň kontroluje syntaktickou správnost programu – tedy zda program je napsán podle jistých formálních pravidel, které zaručují, že mu počítač porozumí. (Syntaktická správnost programu bohužel neza-	

² To se netýká Visual Basicu; jeho poslední verze se překládají do strojového kódu.

ručuje věcnou správnost programu, tj. nezaručuje, že program bude dělat to, co si přejeme.)

Operační systém

S počítačem se typicky dodává vždy alespoň jeden program – operační systém. To je program, který startuje automaticky hned po spuštění počítače a běží po celou dobu jeho běhu. „Oživuje“ počítač: Přijímá pokyny uživatele (nerozhoduje, zda jsou vyjádřeny slovem zapsaným v příkazové rádce nebo kliknutím na myši na ikoně), stará se o jejich provedení a informuje uživatele o výsledcích.

Vedle toho ovšem má ještě řadu dalších úloh, z nichž pro nás nejdůležitější je, že poskytuje služby běžicím programům. Stará se o jejich spouštění, o přidělování paměti, poskytuje nástroje pro práci se soubory atd. Chceme-li např. v programu otevřít soubor, program prostě předá náš požadavek operačnímu systému a ten se postará o vše potřebné.

Mezi nejznámější operační systémy na osobních počítačích patří různé verze Windows a Linuxu.

1.4 Program a algoritmus

Java, podobně jako ostatní programovací jazyky, slouží tedy k zápisu programu. Už víme, že program je nějaký soubor instrukcí (příkazů), které počítači říkají, co má dělat.

Program vždy představuje návod k řešení nějakého problému. Přitom musíme mít na paměti, že počítač za nás problém nevyřeší, jen za nás udělá hrubou práci – prohledá obrovské množství záznamů v databázi, vypočítá něco podle složitých vzorců, vykreslí obrázek ap. *My mu ovšem musíme říci, jak to má udělat.* Jinými slovy, musíme mu poskytnout návod, jak dospět k požadovanému výsledku.

Algoritmus Tento návod musí mít určité vlastnosti; některé z nich mohou vypadat jako samozřejmé, ale přesto je zde uvedeme.

- Návod musí vést k požadovanému výsledku.
- Musí se skládat z kroků, kterým počítač rozumí – tzv. elementárních kroků. (To pro nás znamená, že ho musíme umět zapsat v některém programovacím jazyce.)
- Těchto kroků nesmí být nekonečně mnoho. (Nejde o to, že bychom mohli napsat program, který je nekonečně velký; to se vám asi nepodaří. Není ale nic těžkého napsat krátký program, který

nikdy neskončí, protože se v něm bude do nekonečna opakovat určitá skupina instrukcí.)

Návod, který tyto podmínky splňuje, se obvykle označuje jako *algoritmus*. Na program se můžeme dívat jako na zápis algoritmu v programovacím jazyce.

Metoda shora dolů Otázkou je, jak k algoritmu dospět. Je asi jasné, že nejprve musíme umět daný problém vůbec vyřešit sami³, a pak se můžeme snažit zapsat ho jako posloupnost elementárních kroků. Přitom se používá obvykle tzv. metoda shora dolů: Návod se rozkládá na menší a menší úseky, až dospějeme k elementárním krokům. Při tomto zjemňování se samozřejmě může stát, že budeme muset předchozí rozdělení upravit, některé kroky spojit, některé kroky přidat ap.

Příklad: Jako příklad metody shora dolů vezmeme tuto úlohu: Máme skupinu N číselných proměnných, které označíme $A[0], A[1], \dots, A[N-1]$. Tato čísla chceme uspořádat tak, aby platilo

$$A[0] \leq A[1] \leq \dots \leq A[N-1]. \quad (1)$$

Pole, třídění Skupině proměnných stejného typu, se kterými můžeme zacházet jako s jedním celkem, říkáme *pole*. Zde tedy máme pole A s prvky $A[0], A[1], \dots, A[N-1]$. Čísla, označující jednotlivé prvky pole, se nazývají indexy. Úloha „zpřeházení“ hodnoty, uložené v prvcích tohoto pole, aby vyhovovaly podmínce (1), se nazývá *třídění* pole. Naším úkolem tedy je setřídit pole A .

Náš první návrh programu pro třídění⁴ by mohl vypadat takto:

1. Najdi v poli A prvek, který obsahuje nejmenší hodnotu, a vyměň ho s prvkem, který je na prvním místě (na místě s indexem 0).
2. Najdi v poli A prvek, který obsahuje druhou nejmenší hodnotu, a vyměň ho s prvkem, který je na druhém místě.
3. ... atd., dokud není pole setříděné.

Ovšem program nemůže obsahovat nějaké „... atd.“ – musíme najít lepší formulaci.

Přitom si ale stačí uvědomit, že v prvním kroku vyhledáme nejmenší prvek z celého pole a dáme ho na první místo (do prvku pole s inde-

³ Algoritmům a metodám jejich návrhu je věnována řada knih; většina z nich ale předpokládá znalost určitého konkrétního programovacího jazyka, nejčastěji Pascalu (např. [6]). Asi nejznámější jsou knihy D. Knutha [5], které znalost konkrétního programovacího jazyka nepředpokládají.

⁴ Popsaná metoda se nazývá třídění výběrem (selection sort) a je vhodná pro menší pole.

xem 0). Nejmenší hodnota je tedy už na svém místě a nemusíme se o ni starat. Proto nám ve druhém kroku stačí najít nejmenší prvek ze zbylého úseku pole $A[1], \dots, A[N-1]$ a ten dát na druhé místo, ve třetím kroku najít nejmenší hodnotu mezi prvky $A[?], \dots, A[N-1]$ atd. V posledním kroku budeme hledat nejmenší z prvků $A[N-2], A[N-1]$. Pak bude pole seříděné, tj. bude vyhovovat podmínce (1).

To znamená, že opakujeme podobnou akci (vyhledat nejmenší prvek v úseku pole a dát ho na první místo v tomto úseku) se stále menším úsekem pole. Zkoumaný úsek pole začíná postupně prvkem s indexem $0, 1, \dots, N-1$.

Z těchto úvah může vzejít lepší formulace. Bude používat pomocnou proměnnou i , která nám poslouží jako „počítadlo“.

Lepší formulace

1. Do proměnné i vlož hodnotu 0.
2. Mezi prvky $A[i], \dots, A[N-1]$ najdi nejmenší.
3. Vyměň ho s prvkem, který je na i -tém místě.
4. Zvětši hodnotu proměnné i o 1.
5. Je-li $i < N-1$, pokračuj krokem 2, jinak skonči.

Kroky 1, 4 a 5 můžeme považovat za elementární, kroky 2 a 3 je třeba zpřesnit.⁵ Podivejme se nejprve na krok 2. Budeme v něm potřebovat dvě pomocné proměnné. V jedné, kterou pojmenujeme \min , si budeme pamatovat index nejmenšího zatím nalezeného prvku, a druhá, k , nám poslouží jako „počítadlo“ při procházení zkoumaného úseku pole.

2. Mezi prvky $A[i], \dots, A[N-1]$ najdi nejmenší.
 - 2a. Do obou pomocných proměnných, k a \min , ulož i . (To je index počátečního prvku zkoumaného úseku. Zatím jsme prozkoumali jen jeden prvek a tak ho pokládáme za nejmenší.)
 - 2b. Zvětši k o jedničku. (Přejdi na další prvek v poli.)
 - 2c. Je-li $k = N$, skonči.
 - 2d. Je-li $A[k] \leq A[\min]$, ulož do \min hodnotu k .
 - 2e. Přejdi na bod 2b.

Po skončení bude proměnná \min obsahovat index nejmenšího prvku v prohledávaném úseku.

Podobně můžeme ještě upřesnit bod 3:

3. Vyměň prvek $A[\min]$ s prvkem $A[i]$.
 - 3a. Do pomocné proměnné x ulož hodnotu $A[\min]$.
 - 3b. Do $A[\min]$ přesuň hodnotu $A[i]$.
 - 3c. Do $A[i]$ přesuň hodnotu x .

⁵ To prozatím berte jako fakt; z vašich dosavadních znalostí to nevyplývá.

Tyto kroky již můžeme pokládat za elementární. ♦

Metoda zdola nahoru	Občas se také hovoří o tzv. metodě zdola nahoru. Tím se myslí, že budeme svůj počítač postupně učit nové elementární kroky, které poskládáme z kroků, které počítač už umí. Jestliže například zjistíme, že ve svém programu často třídíme pole, naprogramujeme si tento algoritmus jednou pro vždy a později ho budeme už jen používat a budeme ho pokládat za elementární krok.
Knihovna	Poznamenejme, že součástí každého programovacího jazyka jsou knihovny, které obsahují naprogramovanou řadu běžných algoritmů.

1.5 Objekty a třídy

Objektově orientované programování (OOP) je v poslední době velice často skloňovaným pojmem. Protože Java je jazykem čistě objektovým, tzn. vyžadujícím objektový přístup, musíme si o něm povědět. V tomto oddílu si vysvětlíme základní principy. Až budeme znát základy Javy, vrátíme se k nim a povíme si více.

Zapouzdření

Zatím jsme uvažovali o programování a programech převážně z hlediska počítače, hardwaru. Od zobrazování dat v paměti jsme došli k datovým typům, od strojového kódu k programovacím jazykům. Pokusme se nyní na programování podívat z hlediska programátora, který stojí před úkolem něco vyřešit – ať už jde o program pro vedení účetnictví malé či velké firmy, počítačovou hru nebo systém řídicí vstříkování paliva v automobilovém motoru. Program musí v každém případě odrážet vybrané stránky řešené úlohy, musí *být jejím počítacovým modelem*.

Proč objekty? Je asi jasné, že programátorovi se bude lépe pracovat, bude-li moci uvažovat v termínech řešeného problému, než když bude muset uvažovat v termínech datových typů, které jsou jednou pro vždy dány jako součást programovacího jazyka. Jinými slovy, bylo by vhodné, aby si programátor mohl definovat své vlastní datové typy, které by mohl použít k programovému popisu řešené úlohy. V OOP se jim říká *objektové typy* neboli *třídy*, neboť popisují třídy objektů, které se vyskytují v řešeném problému.

Jako příklad si představme, že programujeme grafický editor. Jeho uživatel si v něm bude chtít kreslit obrázky složené ze základních geometrických tvarů – bodů, úseček, kružnic ap.

Programátor bude muset ve svém programu tyto grafické objekty nějak reprezentovat. Bude si muset vytvořit programový popis pro bod, úsečku atd. a bude muset naprogramovat také operace, které lze s grafickými objekty dělat.

Bod jako objekt

Co to znamená? Abychom si to ujasnili, zamysleme se např. nad bodem. Z geometrie si nepochybň ře pamatuji, že bod v rovině (v našem případě na obrazovce monitoru) je určen souřadnicemi – dvojicí reálných čísel. Vedle toho ovšem potřebujeme určit také jeho barvu; tu můžeme v počítači vyjádřit jedním celým číslem. Z toho plynec, že k reprezentaci bodu stačí skupina tří čísel; s touto skupinou budeme chtít často zacházet jako s celkem.

Abychom mohli s bodem snadno pracovat, musíme popsat (naprogramovat) také operace, které s ním chceme dělat. Musíme ho umět vytvořit, zrušit, zobrazit, přemístit, zjistit jeho barvu, změnit jeho barvu, zjistit jeho polohu, změnit jeho polohu atd.

Třída, atribut

V programu budeme určitě potřebovat větší množství bodů. Abychom nemuseli popisovat každý zvlášť, definujeme *bod* jako nový datový typ – jako *třídu*. Jednotlivé body, které nakreslí uživatel našeho grafického editoru, budou v programu představovány proměnnými (budeme také říkat *objekty* nebo *instancemi*) třídy *Bod*. Každá instance třídy *Bod* bude obsahovat *datové složky* (budeme také říkat *atributy*), které popisují jeho polohu a barvu.

Metoda

K zacházení s jednotlivými body budeme používat operace, které jsme k tomu účelu naprogramovali. Také tyto operace budou součástí definice třídy; v OOP se nazývají *metody*.

Při definici nového typu jsme udělali (zatím jen naznačili) dvě věci: Určili jsme

- datovou reprezentaci nového typu,
- operace s ním (metody).

Ukrývání implementace

Jedno z pravidel objektového programování říká, že s datovými složkami bychom měli manipulovat pouze prostřednictvím metod. Tím před zbytkem programu skrýváme, jaká je skutečná datová reprezentace tohoto typu – ukrýváme jeho implementaci. Později si vysvětlíme, proč je to tak důležité.

Zapouzdření

Ukrývání implementace se v objektovém programování označuje jako *zapouzdření* (anglicky *encapsulation*). Někdy se pod tento pojem zahrnuje i společná definice datové reprezentace nového typu a operací s ním.

Objektový program

Objektový program se skládá pouze z objektů, tedy z instancí objektových typů, a tyto objekty si navzájem posílají zprávy. Mohou posílat také zprávy třídám jako celku. To zní asi velice nesrozumitelně, ale v podání jazyka Java to prostě znamená, že volají (spouštějí) své metody.

Vezměme opět grafický editor a podivejme se na něj jako na celek. Celá aplikace bude nejspíš představována instancí třídy `Editor`, jednotlivé součásti obrázku budou představovány instancemi tříd `Bod`, `Usečka` atd. Když uživatel stiskne tlačítko, kterým určí, že chce někde vytvořit nový bod, tedy instanci třídy `Bod`, dozví se to editor. Ten požádá o vytvoření instance třídy `Bod` (pošle třídě `Bod` zprávu, která bude představovat žádost o vytvoření instance určitých vlastností). Bude-li třeba vybraný bod přemístit, pošle editor této instanci zprávu „přemísti se“, tj. zavolá odpovídající metodu.

Modelovací jazyk UML



Skládání objektů

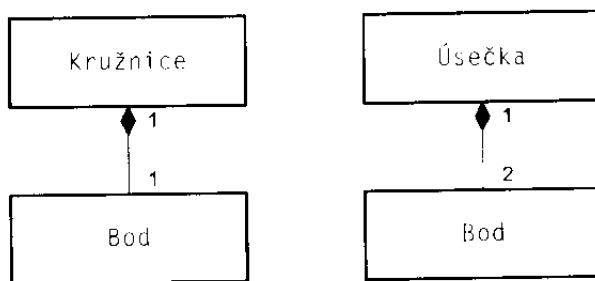
Pojdeme dál. Vedle bodů budeme v programu potřebovat také kružnice. Jak víme, je každá kružnice určena svým středem a poloměrem. Vedle toho ovšem musíme ještě určit tloušťku čáry a její barvu. Mohli bychom tedy tvrdit, že k popisu kružnice nám stačí pětice čísel; ovšem podíváme-li se na to pozorněji, uvědomíme si, že střed kružnice je bod a k jeho popisu můžeme využít už hotový typ `Bod`.

To znamená, že nový typ – třída – kružnice⁶ bude popsán jednou instancí třídy `Bod` a třemi čísly (poloměrem, barvou a tloušťkou čáry).

⁶ V programech se k pojmenování objektů, metod atd. obvykle písmena s háčky a čárkami nepoužívají – většina starších programovacích jazyků to nedovoluje. Java to ovšem umožňuje. Navíc nám to výrazně usnadní pochopení diagramů, které nejsou závislé na žádné programovacím jazyku.

Podobně třída *Úsečka* bude popsána dvěma body, barvou a tloušťkou čáry. Při definici nového typu nám nic nebrání použít jako datovou složku typ, který už existuje. Tomu říkáme *skládání objektů*.

Skládání objektů se v UML vyznačuje čarou zakončenou vyplňeným kosočtvercem. Tato „šipka“ směřuje od třídy představující komponentu ke třídě, která ji používá. Vztah mezi třídami *Bod*, *Úsečka* a *Kružnice* ukazuje obr. 1.1.



Obr. 1.1. Znázornění vztahu mezi třídami *Bod*, *Úsečka* a *Kružnice*. Císla, připojená ke spojnicím, vyjadřují, že *Kružnice* obsahuje jeden *Bod*, zatímco *úsečka* obsahuje dva *Body*.

Dědění

Samotné zapouzdření je sice dobré, zlepšuje přehlednost programu, ale je to málo. Při psaní grafického editoru nejspíš brzy zjistíme, že velice často píšeme podobné úseky programu. V našem editoru bude me např. chít jako jednu z možností nabídnout otáčející se úsečku, třídu *ÚsečkaRotujici*. Ta bude mít stejné vlastnosti jako obyčejná úsečka (bude popsána stejnými datovými složkami), navíc ale bude obsahovat údaj o rychlosti otáčení a metody, které nám umožní tuto rychlosť zjišťovat a měnit.

Bylo by jistě nesmyslné při programování otáčivé úsečky opisovat znova vše, co jsme naprogramovali pro obyčejnou úsečku. OOP proto nabízí mechanizmus nazvaný nepříliš šťastně *dědění*, který umožňuje od existující třídy odvodit třídu novou.⁷

⁷ Pro třídu, od které odvozujeme, se obvykle používá označení *předek*, *rodičovská* nebo *bázová* třída. Pro třídu, kterou od ní odvodíme pomocí dědění, se používá označení *potomek*, *odvozená* nebo *dceřinná* třída. Místo *dědění* se také říká *dědičnost*.



Odvozená třída bude obsahovat všechny datové složky a všechny metody předka; k nim můžeme přidat nové datové složky a nové metody. Můžeme také předefinovat (změnit, překrýt) některou z metod předka. Při dědění nemůžeme žádnou datovou složku ani žádnou metodu odstranit.

Vraťme se k našemu příkladu s otáčející se úsečkou. Třída `UseckaRotujici`, kterou odvodíme jako potomka třídy `Usecka`, bude obsahovat:

- Stejné datové složky jako rodičovská třída. My k nim přidáme novou datovou složku vyjadřující rychlosť otáčení.
- Stejné metody jako rodičovská třída. My k nim přidáme metody pro zjištění a nastavení rychlosti otáčení a změníme metodu pro kreslení, neboť otáčející se úsečka se bude kreslit jinak než obyčejná úsečka.

Ovšem dědění se při práci na grafickém editoru uplatní ještě jinak. Při programování tříd vyjadřujících různé grafické objekty brzy zjistíme, že mnohé části programu jsou stejné nebo hodně podobné:

- Všechny třídy vyjadřující grafické objekty obsahují datovou složku popisující barvu a metody pro její zjištění a nastavení.
- Všechny tyto třídy obsahují metody pro nakreslení a smazání objektu.
- ... a další.

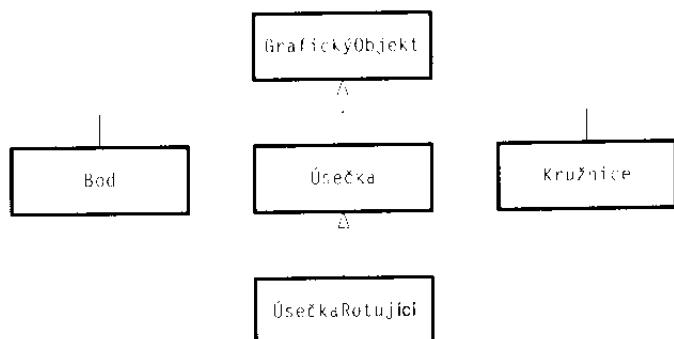
Programovat jednu včetně několikrát, to nikdy není dobré; jednak je to zbytečná práce navíc, jednak rozhodneme-li se něco změnit, musíme to měnit na několika místech a snadno se stane, že na některé místo zapomeneme.

Proto bude rozumné vzít společné vlastnosti všech tříd grafických objektů z našeho programu a přenést je do společného předka. Tuto třídu výstižně nazveme `GrafickyObjekt` a ostatní třídy `Bod`, `Usecka` a další pak odvodíme jako její potomky.

Jestliže bude třída `GrafickyObjekt` obsahovat např. metodu `nakresli`, která objekt nakreslí, bude tuto metodu obsahovat i každá z odvozených tříd. (Je ovšem pravděpodobné, že ji budeme muset v každém z potomků předefinovat; přece jen úsečka se kreslí jinak než kružnice.)

V UML znázorňujeme dědění pomocí šipky, která vede od potomka k předkovi. Třída, která je sama potomkem, může sloužit jako předek jiné třídy. Tak mohou vzniknout *dědické hierarchie*, posloupnosti tříd,

navzájem odvozených děděním. Obrázek 1.2 ukazuje dědičkou hierarchii tříd, o kterých jsme hovořili v příkladu o grafickém editoru.



Obr. 1.2 Dědičká hierarchie tříd z příkladu o grafickém editoru

Polymorfizmus

Zamysleme se nyní nad během našeho grafického editoru. Uživatel si bude kreslit: To znamená, že bude vytvářet úsečky, rotující úsečky, kružnice a další instance jednotlivých tříd představujících grafické objekty. Se všemi se bude v programu zacházet velmi podobně: Bude třeba je vytvořit, určit jejich polohu, barvu atd., bude potřeba je nakreslit, možná smazat atd. Z našich předchozích úvah vyplynulo, že budou obsahovat řadu stejných metod. Bylo by tedy velice pohodlné, kdybychom s nimi mohli také v programu jednotným způsobem zacházet – kdybychom se nemuseli starat o konkrétní typ té které instance, kdyby stačilo vědět, že jde o instanci třídy, která je odvozena od společného předka, třídy *GrafickýObjekt*.

To opravdu v objektově orientovaném programování jde. Platí totiž velice důležité pravidlo: *Potomek může vždy zastoupit předka*. To znamená, že na místě, kde v programu očekáváme instanci předka, můžeme vždy použít instanci potomka.

V našem editoru tedy použijeme pro reprezentaci nakresleného obrázku např. pole instancí třídy *GrafickýObjekt* a budeme do něj ukládat úsečky, kružnice, body – podle toho, co uživatel zrovna vytvoří. O skutečný typ instance se nebudeme starat.

Když budeme chtít celý obrázek nakreslit, vezmeme prostě jednu instanci po druhé a pošleme jí zprávu, že se má nakreslit, tj. zavoláme její metodu *nakresli*. O skutečný typ instance se nebudeme starat: Ať je jakéhokoli typu, metodu *nakresli* určitě obsahuje, neboť ji obsa-

huje společný předek. (V každém z odvozených typů je samozřejmě definována jinak, neboť úsečka se kreslí jinak než kružnice.)

Této vlastnosti se říká *polymorfismus* neboli mnohotvarost. Jeden objekt může mít mnoho tvarů: Např. jedna instance třídy *GrafickýObjekt* může obsahovat podle okolností bod, kružnici, ...

Abstraktní třída

Při úvahách o třídě *GrafickýObjekt*, která představuje obecný grafický objekt, narazíme na malý problém: Jak se vlastně obecný grafický objekt kreslí?

Nijak – kreslit obecný grafický objekt přece nemá smysl, stejně jako nemá smysl třeba chtít kupovat obecnou potravinu nebo se ptát na výšku obecného stromu. *GrafickýObjekt* je tzv. *abstraktní třída*, tj. třída, která představuje pojem natolik abstraktní, že určitá operace, typická pro všechny její podtřídy, pro něj nemá smysl. V našem případě nemá smysl uvažovat o kreslení obecného grafického objektu. Přes-to je jasné, že třída *GrafickýObjekt* bude muset mít metodu *nakresli*, abychom ji mohli použít pro instance odvozené třídy – abychom mohli říci „nakresli tento grafický objekt“. Ve skutečnosti to bude instance některého z odvozených typů, ale v době psaní programu nevíme jakého.

Metodu, která nemá v dané třídě konkrétní smysl, ale kterou musíme definovat, abychom ji mohli volat v odvozených třídách, označujeme také jako *abstraktní*.

Dědění versus skládání

Při vytváření tříd představujících grafické objekty bychom se mohli nechat svést následující úvahou: Kružnice je popsána středem a poloměrem. Střed je bod. Odvodíme tedy třídu *Kružnice* jako potomka třídy *Bod* (a ušetříme tak jednu zbytečnou třídu).

To by mohlo na první pohled fungovat. Jenže ono by to také znamenalo, že můžeme kružnici použít všude, kde program očekává bod – a to může vést k problémům, neboť kružnice *není* bod.

Odvozená třída totiž vždy znamená zvláštní případ bázové třídy. Úsečka je zvláštní případ grafického objektu; rotující úsečka je zvláštní případ úsečky. Proto má smysl odvodit třídu *Úsečka* jako potomka třídy *GrafickýObjekt* a třídu *ÚsečkaRotujici* jako potomka třídy *Úsečka*. Na druhé straně kružnice *není* bod, takže odvodit třídu *Kružnice* jako potomka třídy *Bod* nemá smysl.

Ovšem kružnice *má* bod (střed). Proto má smysl použít skládání.





Předchozí úvahy ukazují něco, čemu se občas říká test *je* – *má* (anglicky *is a – has a*). Jestliže můžeme říci, že třída B je zvláštní případ třidy A, má smysl odvodit třídu B jako potomka třídy A. Jinak je vhodnější použít skládání.

1.6 Java neboli kafe



Java je poměrně nový programovací jazyk; veřejnosti byl představen v roce 1995 a v současné době se uvádí, že ročně přibudou po celém světě asi 4 miliony programátorů, kteří ho používají. Vytvořil ho vývojový tým firmy Sun Microsystems vedený J. Goslingem a původně byl určen k programování pro vestavěné systémy (což v této souvislosti mělo znamenat programové vybavení řídící mikrovlnné trouby a podobná zařízení). Tento jazyk se měl jmenovat Oak, tedy dub – prý podle stromu, který rostl před oknem pana Goslinga. Když se zjistilo, že tento název je již zadán, rozhodli se jeho tvůrci pro název Java, což je jakási americká varianta espresa. Logem se pochopitelně stal kouřící hrnek kávy.

Později se ukázalo, že Javu lze snadno a dobře používat v prostředí internetu, zejména vzhledem k možnostem zabezpečení, a to se stalo základem jejího úspěchu. Dnes se používá v mnoha oblastech. Vedle běžných aplikací (programů) se v ní píší aplety (krátké programy, které lze vkládat do webových stránek), servlety (programy, které běží na webových serverech a vytvářejí dynamické stránky), aplikace tzv. střední vrstvy (součásti distribuovaných aplikací, tj. aplikací, které se skládají z několika programů běžících na různých počítačích a komunikujících spolu prostřednictvím sítě) a mnohé další.

Kde ji získat

Zdarma

Jazyk Java je k dispozici zdarma. Vše, co potřebujete k programování v Javě, lze stáhnout např. z internetové adresy <http://java.sun.com>. Uspořádání webových portálů se průběžně mění, v současné době je třeba použít odkaz [Downloads](#) a na odpovídající stránce vyhledat odkaz na J2SE 5. Tam zpravidla najdete i instrukce pro instalaci. Poznámejme, že vývojové nástroje pro programování v jazyce Java se obvykle označují zkratkou JDK, což znamená Java Development Kit.



Zároveň s JDK si z téhož zdroje stáhněte i dokumentaci (pro JDK 1.5.0 je v souboru `jdk-1_5_0-doc.zip`) a rozbalte ji do podadresáře `doc` adresáře s instalací JDK.

Verze Javy

Od roku 1995 bylo publikováno několik verzí jazyka Java a JDK. Z nich se můžete nejspíš setkat s verzemi JDK 1.0.2, 1.1.8, 1.2.2,

1.3.2, 1.4.2 nebo 5. Verze JDK 1.0 a 1.1 se dnes už příliš nepoužívají; nejspíše se setkáte s JDK 1.3.x nebo 1.4.x. Verze 5 je v době přípravy 2. vydání této knihy žhavou novinkou, na kterou si programátoři teprve zvykají.

Od JDK 1.2 se hovoří o jazyce Java 2; to je samozřejmě především obchodní trik, protože mezi JDK 1.1 a JDK 1.2 nejsou samotném jazyce Java podstatné rozdíly. Podobně po verzi JDK 1.4 nenásleduje JDK 1.5, ale JDK 5 (zkráceně též Java 5). Zde je ovšem v skok číslování opodstatněný, neboť mezi těmito dvěma verzemi byly možnosti jazyka Java velmi výrazně rozšířeny.

Od JDK 1.4 se také rozlišují různé edice. Vedle standardní edice (Java 2, Standard Edition, J2SE), určené pro vývoj běžných aplikací, se můžeme setkat s edici pro podnikové aplikace (Java 2 Enterprise Edition, J2EE), a s edicí pro mobilní zařízení (Java 2 Micro Edition, J2ME). V této knize budeme používat standardní edici, tedy J2SE.

Java 2 přinesla mimo jiné podstatně kvalitnější implementaci některých knihoven, např. tzv. knihovny kontejnerů. Její součástí je také knihovna JFC/Swing, která obsahuje komponenty pro elegantnější vytváření grafického uživatelského rozhraní aplikací, než umožňovala knihovna AWT dodávaná s JDK 1.0. Java 5 přinesla především rozšíření samotného jazyka Java (generické typy, výčtové typy atd.).

Naše kniha je založena na nejnovější verzi, tedy na Javě 5.

Vizuální vývojová prostředí Java je umožňuje programování v různých vizuálních vývojových nástrojích (často označovaných zkratkou RAD – Rapid Application Development). Jejich obrovskou výhodou je, že usnadňují vytváření „okenních“ aplikací. Když v takovémto nástroji „vizuálně“ – tedy myši – sestavíte z předem připravených komponent základ svého programu, vytvoří se automaticky odpovídající zdrojový kód; to je sice velice pohodlné, pro profesionální vývoj programů jsou vizuální vývojová prostředí nedocenitelná, ale pro první seznámení s jazykem se příliš nehodí.

Ve většině této knihy budeme proto předpokládat, že pracujete se samotným JDK, že překlad spouštíte z příkazové řádky atd. Je to sice o něco pracnější, ale na druhé straně nemusíte začínat studiem dokumentace (toho si s Javou užijete i tak dost), nemusíte se starat o tzv. projekt a snáze pochopíte vše, co se ve vašich programech děje.

Nicméně pro úplnost uvádím, že i některá vizuální vývojová prostředí lze získat zdarma. Jde např. o Borland JBuilder Personal Edition Borland. Borland JBuilder Personal Edition lze získat na adrese www.borland.com na stránkách věnovaných produktu Jbuilder, pokud

se zaregistrujete jako členové borlandské vývojářské komunity. Na téže stránce najdete i pokyny pro instalaci.



Jiným podobným nástrojem je vývojové prostředí firmy NetBeans⁸, které lze pro nekomerční účely získat zdarma na internetových stránkách www.netbeans.org.

Speciálně pro výuku Javy je určeno prostředí BlueJ, které lze získat na www.bluej.org.

Je důležité, aby operační systém uměl najít programy, které se nacházejí v podadresáři `bin` adresáře s instalací JDK. Pokud tedy nechcete při každém volání těchto programů uvádět celou cestu, upravte systémovou proměnnou PATH.

Třídy v Javě

V Javě se rozlišují dvě základní skupiny datových typů: tzv. primitivní typy (čísla, znaky, logické hodnoty) a objektové typy (třídy). Jak později uvidíme, zacházení s proměnnými těchto dvou skupin se liší.

Třída Object

V Javě je řada předdefinovaných tříd, které všechny tvoří jednu hierarchii, tj. jsou odvozeny od společného předka – třídy `Object`. V této hierarchii leží i třídy, které definuje programátor ve svém programu. Jestliže v deklaraci nové třídy neuvede předka, doplní si překladač jako předka třídu `Object`. To znamená, že všechny třídy obsahují určité základní metody, a tedy u všech instancí všech tříd lze očekávat jisté základní společné chování. Tyto metody lze v odvozených třídách samozřejmě předefinovat.

Java a C++

C +
Čtenáři, kteří se s jazykem C nebo C++ dosud nesetkali, mohou tento oddíl klidně přeskočit.

Syntax jazyka Java, tedy způsob zápisu programu, je – alespoň na první pohled – velmi podobná syntaxi jazyka C++. To znamená, že programátoři, kteří znají C nebo C++, se velice brzy naučí používat příkazy a některé další konstrukce Javy. *Ovšem pozor: Podobnost Javy a C++ je velice povrchní a brzy zjistíte, že tyto dva jazyky se v mnoha ohledech podstatně liší.* Podivejme se alespoň zhruba na některé jejich zásadní odlišnosti.

⁸ Stojí zato poznamenat, že NetBeans je pražská softwarová firma, nyní plně vlastněná firmou Sun Microsystems. Vývojové prostředí této firmy je považováno za jedno z nejlepších.

- Java je tzv. čistě objektový jazyk. To znamená, že téměř vše v tomto jazyce jsou objekty.
- Na rozdíl od C++ lze pracujeme v Javě pouze s dynamickými objekty (vytváříme je pomocí operátora `new`). O úklid vytvořených objektů se nemusíme starat, Java obsahuje automatickou správu paměti (tzv. garbage collector). To je mechanizmus, který se postará o automatické odstranění nepoužívaných objektů.
- Vzhledem k tomu, že je Java čistě objektový jazyk, má zcela jiné knihovny než C++, které umožňuje i neobjektové programování. Součástí standardních knihoven Javy jsou (od verze 2) i třídy JFC/Swing pro vytváření grafického uživatelského rozhraní.
- V Javě se prakticky všechny metody volají pomocí pozdní vazby. Java neobsahuje analogii klíčového slova `virtual`.
- Java na rozdíl od C++ obsahuje vestavěnou podporu multithreadingu (vícevláknového programování). Podstatně se liší také mechanizmus dynamické identifikace typů, a pokud na to zapomenete, můžete na základě podobnosti s Javou napáchat v C++ poměrně obludeň chyby.
- Nástroje pro generické programování najdeme pouze v Javě 5 a na rozdíl od C++ představují pouze nástroj pro zesílení typové kontroly.
- Také výčtové typy najdeme pouze v Javě 5 a jejich implementace se liší od implementace v C++.
- Součástí Javy jsou nástroje pro vytváření distribuovaných aplikací (RMI), možnost volání metod napsaných v některém jiném programovacím jazyce (rozhraní JNI) a další věci, které nemají v C++ obdobu.

2 První program

Jediný způsob, jak se naučit nějaký programovací jazyk, je psát v něm programy. To ovšem nestačí: Programy musíme umět nejen napsat, ale i přeložit a spustit. V této kapitole se seznámíme s absolutními základy Javy a naučíme se vytvořený program přeložit, spustit a vytvořit k němu dokumentaci.

V této kapitole (a v celé knize) budu předpokládat, že jste si instalovali JDK do adresáře jménem `jdk1.5.0`.

2.1 Ahoj, lidi

Téměř každá učebnice programování začíná jednoduchým programem, který vytiskne nějaký vtipný text – nejlépe něco jako „hello, world“ nebo „ahoj, lidi“ – a skončí. I my se této tradice přidržíme.

Zdrojový text

Editor

Nejprve musíme program napsat, tedy vytvořit jeho zdrojový kód. K tomu potřebujeme libovolný ASCII editor, tj. editor, který vytvoří soubor obsahující pouze znaky, rozčleněné na řádky. Náš editor musí umět pracovat s „dlouhými“ jmény souborů. Pokud pracujete pod 32bitovými Windows, můžete použít třeba notepad (poznámkový blok); pod Linuxem poslouží např. editor emacs nebo vi. Nehodí se například Word nebo Text 602, které text formátují, takže do něj vkládají i dodatečné informace o velikosti a typu písma apod.

V tomto editoru vytvořte soubor `Ahoj.java`, který bude obsahovat následující text, a uložte ho do samostatného adresáře:

```
/* Náš první program v Javě - soubor Ahoj.java */
public class Ahoj {
    public static void main(String[] arg)
    {
        System.out.println("Ahoj, lidi");
    }
}
```



Přitom je důležité, abyste při opisování dodrželi velikost písmen, neboť v Javě se velká a malá písmena důsledně rozlišují. Je také důležité, aby se jméno souboru shodovalo se jménem, uvedeným za slovy

public class ve druhém řádku předchozího výpisu, a to opět včetně velikosti písmen. Jméno souboru musí mít příponu .java.



Tento zdrojový text najdete také na WWW⁹ v adresáři Kap01\01.

Překlad

Abychom mohli vytvořený program spustit, musíme nejprve přeložit jeho zdrojový text do podoby, která bude srozumitelná pro počítač; v případě Javy tedy do bajtového kódu. K použijeme překladač javac.exe, který najdeme v podadresáři bin adresáře s instalací JDK, tedy v jdk1.5.0\bin. Program javac spustíme z příkazové řádky zápisem

```
javac Ahoj.java
```

Příponu .java nelze vynechat. Pokud překlad proběhne bez problémů, nevypíše překladač na konzolu (do příkazové řádky) nic.

Jak to zkazit

Může se stát, že se náš program nepodaří přeložit. V tom případě nás překladač javac nebo operační systém informuje o vzniklých problémech.



- Objeví-li se na obrazovce některé z hlášení Bad command or file name, Chybný příkaz či název souboru nebo 'javac' is not recognized as an internal or external command, operable program or batch file, znamená to, že operační systém nenašel soubor javac.exe. Pokud máme správně nainstalováno JDK, nezná operační systém cestu do podadresáře bin adresáře s instalací JDK. Buď ji zadáme při spouštění, např.

```
C:\jdk1.5.0\bin\javac Ahoj.java
```

nebo uložíme tuto cestu do systémové proměnné PATH.

- Objeví-li se hlášení

```
javac: invalid argument: Ahoj
```

za kterým následuje popis volby, které lze v příkazovém řádku použít, nejspíš jsme ve jménu souboru zapomněli uvést příponu .java.

Syntaktické chyby

Může se také stát, že program bude obsahovat prohřešek proti pravidlům jazyka Java (syntaktickou chybu), takže ho překladač nebude

⁹ Internetová adresa stránky, z níž si můžete stáhnout zdrojové texty příkladů, je uvedena v úvodu knihy na str. 10.

schopen přeložit. Zkuste např. v předchozím příkladu umazat středník na konci pátého řádku, takže bude mít tvar

```
System.out.println("Ahoj, lidi")
```

a takto upravený program zkuste přeložit. Překladač vypíše zprávu

```
Ahoj.java:5: ';' expected
      ^
1 error
```

První řádek říká, že v souboru `Ahoj.java` očekával překladač na pátem řádku středník. Pak následuje výpis chybné řádky; pod místem, kde překladač vidí chybu, je stříška (znak ^). Výpis končí zprávou o celkovém počtu nalezených chyb.

Bohužel, ne vždy jsou chybová hlášení tak přehledná a snadno srozumitelná jako v tomto případě.

Soubory

Jestliže se nyní podíváte do adresáře, kde je uložen zdrojový program, najdete v něm kromě souboru `Ahoj.java` také soubor `Ahoj.class`. Ten obsahuje bajtový kód našeho programu. Budeme-li chtít hotový program přenést na jiný počítač, budeme přenášet právě tento soubor.

Běh programu

Program máme hotový, takže ho spustíme. To by mělo být velice jednoduché, stačí příkaz

```
java Ahoj
```

a nás program vypíše do následujícího řádku

```
Ahoj, lidi
```

Všimněte si, že svůj program vlastně nespouštíme přímo: Místo toho voláme program `java`, tj. interpret bajtového kódu, a jemu předáme soubor `Ahoj.class` jako parametr.¹⁰ (Nicméně příponu `.class` zde uvést *nesmíme*). Tento interpret bývá uložen v podadresáři `bin` adresáře s instalací JDK, podobně jako překladač `javac`.

Když to nefunguje

Je pravděpodobné, že takto jednoduše se vám první program spustit nepodaří, zejména pokud jste si JDK instalovali sami. Podívejme se na typické problémy.

¹⁰ Přesněji, předáváme mu jméno *třídy*, od které má program začít. O tom si ale budeme povídат později; zatím si můžeme mylet, že předáváme jméno souboru.



Pokud operační systém program `java` nenajde, je třeba doplnit cestu do adresáře `bin` do seznamu cest uvedených v proměnné `PATH` nebo tuto cestu zadat při spouštění.

Objeví-li se chybové hlášení

```
Exception in thread "main" java.lang.NoClassDefFoundError:  
Ahoj/class
```

nejspíš jste při spouštění uvedli i příponu `.class`.¹¹ Podobné hlášení

```
Exception in thread "main" java.lang.NoClassDefFoundError:  
Ahoj
```

může mít více příčin, vždy však znamená, že interpret `java` nenašel spouštěný soubor (soubor s třídou, od které má program začít).

- Možná, že jsme nesprávně uvedli jméno třídy (stačí záměna malých a velkých písmen). Zkontrolujte si, zda jste je v příkazovém řádku opravdu napsali stejně jako ve zdrojovém souboru.
- Interpret `java` musí umět najít soubory obsahující přeložený program. Přitom využívá systémovou proměnnou `CLASSPATH`, která by měla obsahovat cesty k potřebným adresářům. Pokud nám bude stačit, když budeme své programy spouštět z aktuálního adresáře, postačí, když bude obsahovat hodnotu „..“ (tečka).
- Pokud nechceme měnit hodnoty systémových proměnných, můžeme cesty k souborům s přeloženým programem zadat také jako parametr `-classpath` v příkazovém řádku, například takto:

```
java -classpath . Ahoj
```

Proměnná **CLASSPATH**

V operačním systému DOS nebo v dosovském okně Windows 95/98 nastavíme hodnotu této proměnné příkazem

```
SFT CLASSPATH=.
```

Tento příkaz můžeme umístit i do souboru `autoexec.bat`.

Pod Windows NT vyvoláme příkazem Nastavení | Ovládací panely okno Ovládací panely. V něm vybereme ikonu Systém; otevře se okno Vlastnosti systému a na kartě Prostředí (Environment) definujeme tuto proměnnou a její hodnotu. Ve Windows 2000 postupujeme podobně, použijeme však kartu Pokročilé (Advanced) a na ní stiskneme tlačítko Proměnné prostředí (Environment Variables).

¹¹ Interpret `java` si ke jménu souboru vždy doplní příponu `.class`. Později uvidíme, že příkaz `java Ahoj.class` vlastně způsobí, že interpret `java` hledá soubor `class.class` v podadresáři `Ahoj`, nikoli soubor `Ahoj.class`.

2.2 Co jsme naprogramovali

Nyní umíme program napsat, přeložit a spustit; je tedy nejvyšší čas, abychom se také podívali, co vlastně znamená. Náš výklad ovšem nebude úplný, v mnoha případech si řekneme jen nejdůležitější a zjednodušené informace. Upřesníme je pak v dalších kapitolách.

Komentář

V prvním řádku najdete text

```
/* Náš první program v Javě - soubor Ahoj.java */
```

To je *komentář* – část programu, která nemá pro překladač vůbec žádný význam, nijak neovlivní chod přeloženého programu. Komentáře slouží programátorem, kteří si do nich zapisují poznámky o významu různých částí programu.

Komentář začíná dvojicí znaků /* zapsaných bezprostředně za sebou a končí dvojicí */. Vše, co je mezi těmito „komentářovými závorkami“, překladač ignoruje. (Přesvědčte se o tom: Smažte komentář v souboru *Ahoj.java*, potom program přeložte a spusťte. Program se přeloží a poběží úplně stejně jako předtím.)

Komentář ohraničený závorkami /* a */ může zabírat i několik řádků. Java ovšem zná i jednofádkový komentář. Ten začíná dvojicí lomítka zapsaných těsně za sebou a končí na konci řádku. Náš program by tedy mohl začínat také takto:

```
// Náš první program v Javě - soubor Ahoj.java
public class Ahoj {
```

V Javě existuje ještě třetí druh komentářů, tzv. dokumentační komentáře. Ty mohou zabírat více řádků, začínají znaky /** a končí */. Vrátíme se k nim později v této kapitole.



Zpočátku vám budou komentáře možná připadat zbytečné – svému programu přece rozumíte. Budete mu ale rozumět také za týden? Za měsíc, za rok? Když po čase budete ve svém programu potřebovat něco změnit, budou pro vás dobré napsané komentáře představovat neocenitelnou pomoc.

Mezi programátory je obvyklé, že komentáře piší bud' anglicky (kdyby to náhodou četl někdo cizí...) nebo alespoň česky (tj. bez háčků a čárek). Důvod je jednoduchý: Asi víte, že v různých prostředích se můžete setkat s různými způsoby kódování češtiny, a při přenosu programu z jednoho prostředí do jiného mohou vzniknout problémy.

V naší knize budeme ovšem psát komentáře důsledně česky, to znamená i s diakritickými znaménky. I zde je důvod jednoduchý: *jak vidíte z teto kratke vetylky, cesky psane komentare byvaji spatne srozumitelne*. Nebudeme si tedy přidělávat zbytečnou práci s luštěním cestiny.

Deklarace třídy

Na dalším rádku začíná deklarace jediné třídy v našem programu. To říká slovo `class`, za kterým následuje jméno (budeme říkat *identifikátor*) této třídy, představované slovem `Ahoj`. Za jménem třídy následuje levá (otevírací) složená závorka `{`. Mezi ní a odpovídající pravou složenou závorkou `}` pak je *tělo třídy*, programový popis třídy. Deklarace třídy `Ahoj` tedy vypadá takto:

```
class Ahoj {  
    // ... Tělo třídy ...  
}
```

V našem programu ale před touto deklarací stojí ještě slovo `public`, které říká, že tuto třídu lze používat i z jiných částí programu, že je veřejně přístupná. (Později si podrobně povíme, co to vlastně znamená.)

Metoda main()

Jak víme, mohou třídy obsahovat jednak datové složky, ve kterých se ukládají data, jednak metody, které popisují operace s instancemi tříd. Třída `Ahoj` neobsahuje žádné datové složky. Obsahuje jedinou metodu, která se jmenuje `main()`.¹² Její deklarace je

```
public static void main(String[] arg)  
{  
    System.out.println("Ahoj, Tidi");  
}
```

Metoda `main()` je nesmírně důležitá, neboť od ní bude program začínat. Ve skutečnosti totiž příkaz

```
java Ahoj
```

kterým jsme program spouštěli, říká: Najdi soubor s třídou `Ahoj`, v této třídě najdi metodu `main()` a tu spust.

První řádek, „hlavička“, metody `main()`, musí vždy vypadat tak, jak to vidíte v našem příkladu. Musí tedy obsahovat slova `public`, `static`

¹² Samotný identifikátor této metody je `main`. Závorky za ním v zápisu `main()` naznačují, že jde o metodu; měly by usnadnit čtení textu.

a `void` v uvedeném pořadí. Identifikátor `main` musí být zapsán malými písmeny a v závorkách za ním musí být `String[] arg`.

Pokud něco z toho změníte nebo vynecháte, dostanete se nejspíš do problémů, neboť interpret `java` metodu `main()` nenajde. (Přesněji, téměř jediné, co můžete v hlavičce metody `main()` změnit, je identifikátor `arg` – můžete zde napsat např. `String[] x`. Vše ostatní musí zůstat stejné, jinak interpret `java` tuto metodu při spuštění programu nenajde.)

Za hlavičkou následuje „tělo“ metody `main()`. To jsou příkazy, které má tato metoda provést, uzavřené mezi složené závorky `{` a `}`. V našem případě obsahuje jediný příkaz,

```
System.out.println("Ahoj, lidi");
```



I když vypadá na pohled komplikovaně, neříká nic jiného, než „vezmi znakový řetězec „Ahoj, lidi“ a vypiš ho do standardního výstupu, tj. na obrazovku (konzolu)“. Vypisovaný text musíme zapsat mezi dvojicí uvozovek. (Tento zápis označujeme jako „řetězcovou konstantu“.)

Bude-li metoda `main()` obsahovat více příkazů, budou se většinou provádět v pořadí, v jakém je zde zapíšeme; o výjimkách si řekneme později. Program skončí, když přejde přes složenou závorku `}`, uzavírající tělo metody `main()`.

Doplňující poznámky



Deklarace metody `main()` začíná slovem `public`, které opět znamená, že je veřejně přístupná, tj. že ji smí spustit – volat – kdokoli. Slovo `static` říká, že jde o tzv. statickou metodu, kterou můžeme volat i v případě, že neexistuje ani jedna instance třídy `Ahoj`. Konečně slovo `void` říká, že metoda `main()` nevrací žádný výsledek.



V závorkách za identifikátorem `main` najdeme ještě `String[] arg`. To je tzv. parametr funkce `main();` budeme o něm hovořit v příští kapitole.



Už jsme si řekli, že příkaz `System.out.println("Ahoj, lidi");` vypisuje zadaný text do standardního výstupu. To většinou znamená na konzolu, tj. buď na obrazovku v textovém režimu nebo do okénka v grafickém prostředí, které se chová jako obrazovka v textovém režimu (s příkazovým rádkem). Tento výstup můžeme příkazem operačního systému přesměrovat, např. do souboru. Spustíme-li program příkazem

```
java Ahoj > text.txt
```

nevypíše se text na obrazovku, ale do souboru `text.txt`.

2.3 Vlastní dokumentace

Firma Sun dodává k JDK poměrně rozsáhlou dokumentaci ve tvaru HTML souborů, zabalených do archivů ZIP nebo JAR. Tato dokumentace je obvykle instalována v domovském adresáři JDK v podadresáři DOC. Vedle obecných informací zde najdeme i podrobné přehledy vlastností jednotlivých knihovních tříd.

V tomto oddílu si ukážeme, jak můžeme dokumentaci pro své třídy vytvářet pomocí nástrojů dodávaných jako součást JDK sami. Základem k tomu jsou dokumentační komentáře, které začínají znaky `/**` a končí `*/`, a standardní program `javadoc.exe`.

Dokumentační komentáře popisují smysl třídy a jejích jednotlivých datových složek a metod. Musí být umístěny bezprostředně před deklarací třídy, složky nebo metody.

Program `javadoc.exe`, dodávaný spolu s JDK, umí tyto komentáře ze zdrojového textu vyjmout a vytvořit z nich HTML dokumentaci ve standardním tvaru. Přitom ignoruje hvězdičky na počátku řádků, pokud tam jsou.

Vezměme náš první program a upravme jej takto:

```
/* Soubor Ahoj.java
 * Příklad použití dokumentačních komentářů
 * k automatickému generování dokumentace
 */

/** Třída Ahoj má za úkol vypsat <B>pozdrav</B>.
 * Je to jediná třída v našem prvním programu a
 * nemá žádný skutečný smysl.
 */
public class Ahoj {
    /** Jediná metoda, <I>obsahuje všechnu funkciálnitu
     * programu</I>.
    */
    public static void main(String[] argn)
    {
        System.out.println("Ahoj, lidi");
    }
}
```

Všimněte si, že dokumentační komentář může obsahovat i značky jazyka HTML (alespoň velkou většinu z nich).

Jestliže nyní spustíme program `javadoc` příkazem

```
javadoc Ahoj.java
```

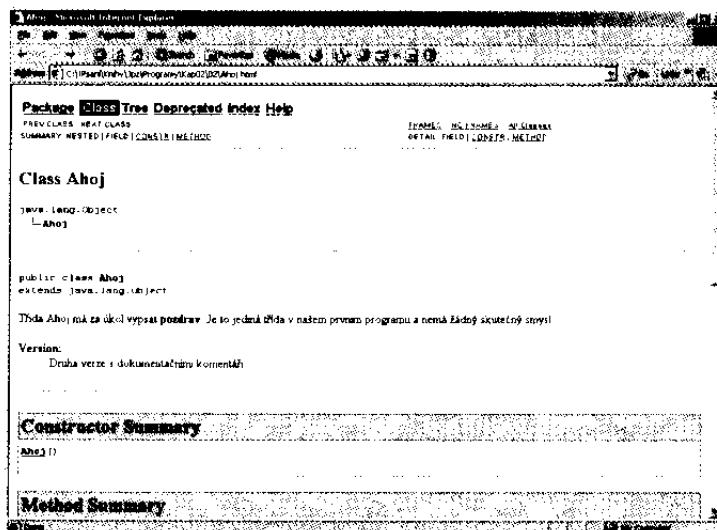
vytvoří se v aktuálním adresáři několik souborů HTML, které si můžete prohlédnout svým internetovým prohlížečem, a jeden soubor se

styly dokumentů (.css). Informace o třídě Ahoj budou v souboru Ahoj.html. Kromě dokumentačních komentářů budou obsahovat informaci o předcích třídy Ahoj až po třídu Object, souhrn zděděných metod, informace o automaticky generovaných metodách a další.

Do dokumentačních komentářů lze také vkládat také příkazy začínající znakem „@“, které způsobí, že program javadoc zařadí do dokumentace další informace. Tak např. @version umožnuje vložit do dokumentace informace o verzi, @see odkaz na jinou třídu, @return informace o výsledku metody atd. Příkaz @deprecated způsobí, metoda bude v dokumentaci označena jako nevhodná (deprecated).

V literatuře se také často uvádí příkaz @author, který by měl vložit do dokumentace informaci o autorovi. Abychom jej mohli použít, musíme v příkazové řádce, kterou spustíme program javadoc, použít přepínač -author. Podobně chceme-li generovat informace o verzi, musíme použít přepínač -version.

Na WWW najdete v souboru Kap01\02 třídu Ahoj s doplněnými dokumentačními komentáři a soubory vytvořené programem javadoc.



Obr. 2.1 Dokumentace ke třídě Ahoj vytvořená programem javadoc

3 Jednoduché příklady

V minulé kapitole jsme viděli nejjednodušší program v Javě a naučili jsme se ho přeložit a spustit. V této kapitole se seznámíme s některými základními konstrukcemi jazyka Java. Náš výklad bude nesystematický; jeho hlavním smyslem je poskytnout vám přehled o základních možnostech, které Java nabízí, a nabídnout vám jakési minimum znalostí nezbytné k tomu, abychom se později mohli věnovat souvisejícímu výkladu a přitom si ukazovat alespoň trochu smysluplné příklady.

To znamená, že si v této kapitole bez nároku na úplnost ukážeme, jak se vytvářejí proměnné, jak se programuje rozhodování a opakování podobných výpočtů a jak se programují metody, které představují dílčí algoritmy. Ukážeme si také, jak vypsat data do standardního výstupu nebo jak je přečíst ze standardního vstupu.

3.1 Měníme svůj první program

V předchozí kapitole jsme napsali, přeložili a spustili svůj první program. Pokud jste vše pečlivě prošli a alespoň trochu tomu rozumíte, můžete začít experimentovat. Některé pokusy si ukážeme, jiné si můžete vymyslet sami.

Doporučuji před započetím každého z pokusů přejít do jiného adresáře a pracovat s kopii původního programu.

Hry s texty

Čeština Nejprve změníme nápis, který náš program vypisuje. Chtěli bychom zjistit, jak je to s češtinou, a proto vypíšeme známou větu o žlutém koni, která obsahuje všechny české znaky s diakritickými znaménky.

```
System.out.println(  
    "Žlutoučký kůň přišerně úpěl d'ábelské ódy.");
```

Pokud takto upravený program vytvoříte v editoru, který pracuje pod Windows, dočkáte se po překladu a spuštění (opět pod Windows, ale i v jiných systémech) nepřijemného překvapení – výstup na konzolu nebude téměř čitelný. V případě, že pracujete pod Windows, dostanete něco jako

Žluťoučký kůň přišerně úpěl d'ábelské ódy.

Problém je v tom, že ve Windows se používá jiné kódování češtiny pro výstup do konzolového okna a jiné pro grafický výstup (to abychom se nenudili...). Obojí kódování se pro jistotu také liší od kódování pod operačním systémem Unix.

Tento problém má řešení, které si ukážeme později, v kapitole věnované vstupům a výstupům. Zatím se spokojíme s výstupy, které budou bez háčků a čárek.



Poznamenejme, že zdrojový text tohoto programu najdete na WWW v adresáři Kap01\01 v souboru ZlutyKun. Třída v něm se jmenuje také ZlutyKun. Pokud jste netrpěliví a chcete si prohlédnout i program, který vypíše češtinu do konzolového okna Windows správně, najdete ho na WWW v souboru kap01\02\ZlutyKun?.java.

Spojování řetězců

Občas se stane, že se nám řetězec nevejde na jeden řádek. Pak nezbývá, než ho rozdělit na několik a dílčí řetězce spojit operátorem +. Dílčí řetězce mohou být i na různých řádcích. Můžeme tedy napsat

```
System.out.println("Ahoj," +
    "lidi");
```

Přechod na nový řádek

Někdy potřebujeme, aby se výstup programu rozdělil na několik řádků. Toho lze dosáhnout různými způsoby. Můžeme například rozdělit text na několik řetězců a každý vypsat samostatným příkazem, neboť metoda System.out.println() za každým výstupem automaticky přejde na nový řádek. Metoda main() by tedy mohla mít tvar

```
public static void main(String[] arg)
{
    System.out.println("Ahoj,");
    System.out.println("lidi");
}
```

Takto upravený program vypíše

```
Ahoj,
lidi
```

Můžeme ale také vložit do řetězce zvláštní znak, který způsobí přechod na nový řádek. Tento znak zapisujeme pomocí tzv. řídicí posloupnosti (escape sequence), která se skládá z obráceného lomítka a malého „n“. Můžeme tedy napsat

```
public static void main(String[] arg)
{
    System.out.println("Ahoj,\nliidi");
}
```

Výstup takto upraveného programu bude stejný jako v předchozím případě.

3.2 Trocha počítání

Proměnné Náš další program se pro změnu pokusí vypočítat a vytisknout součet dvou čísel; k tomu ale budeme potřebovat místo v paměti, kam si tato čísla uložíme. Abychom s nimi mohli pracovat, musíme je deklarovat – tj. musíme programu říci, aby je vytvořil. Začneme s celými čísly, a pro ta se v Javě používá datový typ int. Tvar deklarace proměnných typu int můžeme zjednodušeně popsat takto:

```
int jméno = hodnota;
```

kde *jméno* je identifikátor deklarované proměnné a *hodnota* je hodnota, která bude do této proměnné při vytvoření uložena. (Říkáme, že proměnná bude touto hodnotou inicializována. Inicializaci lze v některých případech vynechat.)

Náš program tedy může vypadat např. takto:

```
/* Sčítání dvou čísel - první pokus
   soubor kap03\03\Soucet.java
*/
public class Soucet {
    public static void main(String[] args) {
        int i = 12, j = 25; // První a druhý sčítanec, výsledek
        k = i+j;           // Vypočítí výsledek
        System.out.println("Soucet cisel " + i +
                           " + " + j + " je " + k);
    }
}
```

První řádek metody `main()` obsahuje deklarace tří proměnných, které jsme pojmenovali *i*, *j* a *k*. První dvě jsme inicializovali hodnotami 12, resp. 25, třetí jsme neinicializovali. (Poznamenejme, že jde o tzv. lokální proměnné. Tyto proměnné existují jen v metodě, ve které jsou deklarovány, nejsou součástí instance třídy `Soucet`. S proměnnými, které jsou součástí instance, tj. s datovými složkami instance, se sejtěme později.)

V následujícím řádku jsme vypočetli výsledek a uložili jsme ho do proměnné *k*. K tomu jsme použili tzv. přířazovací operátor `=`, který říká: Vypočti hodnotu výrazu, který stojí vpravo, a ulož ji do proměnné, která stojí vlevo od symbolu `=`. K sečtení dvou čísel používáme operátor `+` (to je opravdu překvapení). Celá čísla zapisujeme v programech podobně jako v běžném životě.¹³

¹³ Pouze mezi číslicemi nesmíme nikde udělat mezeru – nesmíme tedy napsat např. 12 345, musíme vždy psát 12345.

Podívejme se ještě na čtvrtý řádek metody main(). Napišeme-li v Javě "Součet cisel" + i

zjistí překladač, že se pokoušíme sčítat veličiny dvou různých typů – řetězec a číslo. To ovšem nejde, a proto převede číslo i také na znakový řetězec. Pro znakové řetězce již má operace + smysl: Jak víme, znamená, že se druhý řetězec připojí za první. Výraz ve čtvrtém řádku tedy postupně vytvoří znakový řetězec představující výstup programu.

Tento program po přeložení a spuštění vypíše

Součet cisel 12 + 25 je 37

Postupný výpis

Téhož výsledku bychom mohli dosáhnout i jinak. Místo skládání řetězců pomocí operátora + můžeme použít několik za sebou následujících příkazů k výstupu. Protože však metoda System.out.println() přejde po výstupu vždy na nový řádek, použijeme metodu System.out.print(), která se liší jen tím, že na nový řádek nepřechází. Příkaz k výstupu v metodě main() bychom pak mohli upravit do tvaru

```
System.out.print("Součet cisel ");
System.out.print(i);
System.out.print(" + ");
System.out.print(j);
System.out.print(" je ");
System.out.print(k);
```

(Takto upravený program najdete na WWW v souboru kap03\03\Součet2.java.)

Všimněte si, že metoda System.out.print() – stejně jako System.out.println() – umí vypisovat nejen řetězce, ale i celá čísla. Časem uvidíme, že umí vypisovat i hodnoty mnoha jiných typů.

5

V Javě 5 máme ještě jednu možnost, jak naprogramovat výstup. Instance System.out v této verzi totiž obsahuje metodu printf(), kterou můžeme použít následujícím způsobem:

```
System.out.printf("Součet cisel %d + %d je %d", i, j, k);
```

Prvním parametrem této metody je vždy tzv. formátovací řetězec, který má v našem příkladu tvar "Součet cisel %d + %d je %d". Tento řetězec se vypíše do výstupu, pouze zápisu tvaru %d se v něm nahradí dalšími parametry převedenými na řetězec. To znamená, že první %d se nahradí hodnotou proměnné i, druhé %d se nahradí hodnotou proměnné j atd. Zápis %d se označuje jako „specifikace konverze“ a formátovací řetězec jich může obsahovat libovolné množství. První specifikace konverze odpovídá prvnímu parametru, který násle-

duje za formátovacím řetězcem, druhá specifikace konverze odpovídá druhému parametru za formátovacím řetězcem atd.

Metodu `printf()` lze volat s libovořným počtem parametrů, první z nich však musí být řetězec.

Později poznáme specifikace konverze i pro jiné datové typy a ukážeme si i další možnosti, které dovolují specifikovat podrobnosti formátu vystupující hodnoty.



Zdrojový text programu používajícího metodu `printf()` najdete na WWW v souboru Kap03\03\Soucet3.java.)

3.3 Vstup dat

Pojďme dál. Kdybychom měli psát zvláštní program, kdykoli budeme chtít sečíst nějaká dvě čísla, brzy by nás programování přestalo bavit. Nás program by tato dvě čísla měl umět přečíst z klávesnice.

Přitom ovšem narazíme na problém: formátované vstupy nejsou v Javě až po JDK 1.4 vyřešeny právě nejhodlněji a podrobný popis tohoto problému zatím přesahuje naše možnosti.

JDK 5 nabízí třídu `java.util.Scanner`, jejíž použití je sice sice velmi pohodlné, museli bychom si ale vysvětlit příliš mnoho nových pojmu najednou, a proto se k ní vrátíme až později v následující kapitole. Zatím si ukážeme náhradní postup, který lze použít ve všech verzích JDK.



Toto náhradní řešení, které je naprogramováno v souboru MojeIO.java a najdete ho na WWW v adresáři Kap03\10. Prozatím je berete jako dané. Nás postup při čtení bude následující:

MojeIO

- Do adresáře se zdrojovým souborem svého programu nakopírujte soubor MojeIO.class.
- Ve svém programu pak můžete používat metodu `MojeIO.inInt()`, která ze standardního vstupu načte jedno celé číslo typu `int`. Způsob jejího použití uvidíte v následujícím příkladu.
- Trochu předběhneme a řekneme si, že podobně budete moci používat i metody `MojeIO.inLong()`, `MojeIO.inDouble()` a `MojeIO.inStr()`, které načtou vždy jedno číslo typu `long`, resp. jedno číslo typu `double`, resp. jeden znakový řetězec.
- Budeme-li číst číselnou hodnotu pomocí některé z výše uvedených metod, musíme mít na paměti, že vstup musí obsahovat opravdu jen ono požadované číslo. Před ním a za ním mohou být mezery,

nic více. Pokud zadáte něco jiného, vrátí tyto metody 0 a vypíše upozornění.

- Tyto metody čtou ze standardního vstupu. To znamená, že vstup programu můžeme přesměrovat příkazem operačního systému.

Vraťme se teď k našemu úkolu – napsat program, který přečte z klávesnice dvě čísla a vypíše jejich součet.

Program, který vyžaduje zadání nějakých dat z klávesnice, by měl vždy upozornit, co chce. Měl by také vždy říci, co vypisuje. Proto ho upravíme následujícím způsobem:

```
/* Sčítání dvou čísel - třetí pokus
soubor kap03\04\Soucet4.java
*/
public class Soucet4 {
    public static void main(String[] arg) {
        System.out.print("Zadej první scitanec: ");
        int i = MojeIO.inInt(); // Čti první scitanec
        System.out.print("Zadej druhý scitanec: ");
        int j = MojeIO.inInt(); // Čti druhý scítanec
        System.out.println("Jejich součet je " + (i+j));
    }
}
```

Jestliže tento program přeložíme a spustíme, může naše konverzace s ním vypadat takto:

```
K:\Programy\Kap03\05>java Soucet4
Zadej první scitanec: 45
Zadej druhý scitanec: 45
Jejich součet je 90
```

Jestliže uložíme do souboru data.dta následující dva řádky,

```
123
456
```

a spustíme nás program příkazem

```
java Soucet4 < data.dta,
```

vypíše

```
Zadej první scitanec: Zadej druhý scitanec: Jejich součet je
579
```

Všimněme si použití metody `MojeIO.inInt()`:

- Za jejím jménem musíme zapsat prázdné závorky.
- Její zápis v programu představuje přečtenou hodnotu.



Podívejme se ještě jednou na příkaz, který má na starosti výstup. Závorky kolem výrazu $(i + j)$ vám možná připadají zbytečné; zkusme je tedy vynechat a napsat

```
System.out.println("Jejich součet je " + i + j);
```

Konverzace s programem po této úpravě bude např.

```
C:\Programy\Kap03\05>java Součet3  
Zadej první scitanec: 12  
Zadej druhý scitanec: 34  
Jejich součet je 1234
```

Co se to stalo?

Pořadí operací Překladač uviděl za řetězcem "Jejich součet je " znaménko + a proměnnou i; převedl tedy tuto proměnnou na řetězec a ten spojil s předchozím. Teprve pak se podíval dále, našel opět plus a opět proměnnou, a tak také tuto proměnnou převedl na řetězec a připojil k předchozímu řetězci. To znamená, že výraz

"Jejich součet je " + i + j

zpracovával v pořadí

("Jejich součet je " + i) + j

V programech, podobně jako v matematice, můžeme pořadí, ve kterém se operace ve výrazech provádějí, měnit pomocí kulatých závorek. To znamená, že pokud uzavřeme do závorek výraz $i + j$, pochopí překladač, že má nejprve sečíst tato dvě čísla a teprve jejich součet připojit k řetězci.

Další operace

K počítání můžeme kromě operátoru + použít také - (odečítání), * (násobení) a / (dělení). Operátor násobení musíme vždy zapsat, tj. musíme napsat i*j, nikoli i,j, jak jsme zvyklí z matematiky. Zkusíte-li si dělení, zjistíte, že výsledkem bude celé číslo obsahující celočíselnou část výsledku; zlomková část se „ztratí“. Např. 10 / 4 bude 2, nikoli 2,5 jak byste možná očekávali.

Abychom dostali výsledek i se zlomkovou částí, musel by být alespoň jeden z operandů reálné číslo (typu double nebo float).

3.4 Faktoriál

Nyní si položíme složitější úkol: Napišeme program, který přečte ze vstupu kladné celé číslo n a vypočte součin $1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$ (tzv. faktoriál čísla n , obvykle se označuje symbolem $n!$). Přitom se naučíme programovat rozhodování a cykly (části programu, které se opakují).

Především si ujasníme několik technických detailů:

- Faktoriál záporných čísel není definován.
- Faktoriál čísel 0 a 1 je 1.
- Pro ostatní kladná celá čísla n je $n!$ definován jako součin všech celých čísel od 1 do n .

Cyklus Nejprve budeme předpokládat, že v proměnné n máme kladné celé číslo. Na ověřování této podmínky se podíváme později. Jak vypočteme součin $1 * 2 * \dots * n$, když hodnotu n předem neznáme? Zápis se třemi tečkami nám Java nedovolí; budeme muset postupovat jinak. Vezmeme pomocnou proměnnou s , uložíme do ní 1 a budeme postupně počítat¹⁴ výrazy $s = s * n$; $s = s * (n-1)$; $s = s * (n-2)$ atd. až po $s = s * 2$. Formální zápis tohoto postupu může mít tvar

1. Vytvoříme pomocnou proměnnou s a uložíme do ní hodnotu 1. V této proměnné posupně vypočteme výsledek.
2. Vynásobíme $s * n$ a výsledek uložíme do s .
3. Zmenšíme n o 1.
4. Je-li $n > 1$, přejdeme na krok 2, jinak pokračujeme krokem 5.
5. Proměnná s obsahuje výsledek; konec.

Zde tedy potřebujeme opakovat kroky 2, 3 a 4, dokud je splněna podmínka $n > 1$. (Násobení jedničkou je ve skutečnosti zbytečné, takže ho vynecháme.)

Cyklus: příkaz while Takovéto opakování se nazývá *cyklus* a v Javě ho lze naprogramovat např. pomocí příkazu `while`. Tvar tohoto příkazu je

`while(podminka){příkazy}`

a jeho význam je jednoduchý: Nejprve počítač otěstuje, zda je splněna *podminka* (říkáme také *podminka opakování*). Pokud ano, provede *příkazy* uzavřené mezi složené závorky { a } (tzv. „tělo cyklu“). Pak znova otěstuje, zda je splněna podmínka, atd.

Není-li tato podmínka splněna, tělo počítač cyklu přeskočí a přejde na další příkaz za příkazem `while`.

Podmínka: relační operátory Podmínka je výraz, který je buď *pravdivý* (podmínka je splněna) nebo *nepravdivý* (podmínka není splněna). Obvykle obsahuje porovnání proměnných s danými hodnotami ap. K tomu můžeme použít operátory $>$ a $<$, známé z matematiky, \leq , resp. \geq , které znamenají „menší

¹⁴ Připomeňme si, že operátor = neznamená rovnost, ale přiřazení. Proto výraz $s = s * 3$; znamená „vezmi hodnotu uloženou v s , vynásob ji třemi a výsledek ulož opět do proměnné s “.

nebo rovno“, resp. „větší nebo rovno“, a ==, resp !=, které znamenají „rovná se“ a „nerovná se“. Poznamenejme, že podmínka představuje v Javě hodnotu typu boolean a hodnoty pravda, resp. nepravda jsou vyjádřeny slovy true, resp. false.

Výpočet součinu čísel $n*(n-1)* \dots *1$ můžeme tady naprogramovat např. takto:

```
s = 1;           // Počáteční hodnota
while(n > 1)    // Dokud je n větší než 1
{
    s = s*n;
    n = n-1;
}
```

Zadáme-li jako vstup (hodnotu n) např. číslo 3, bude podmínka $n > 1$ splněna. To znamená, že v těle cyklu se vypočte $s = s*n$, takže proměnná s bude obsahovat hodnotu 3. Dále proběhne příkaz $n = n-1$, který způsobí, že se hodnota uložená v n zmenší o 1, takže n bude obsahovat 2. Pak se program vrátí k podmínce opakování; ta je stále splněna, takže tělo cyklu proběhne ještě jednou. Opět se vypočte výraz $s = s*n$, tj. do s se uloží hodnota 6, a pak se hodnota uložená v n sníží o 1, takže tato proměnná bude obsahovat hodnotu 1.

Program se opět vrátí k podmínce opakování; ta však už není splněna, takže provádění cyklu skončí a program přejde na první příkaz za závorkou ukončující jeho tělo.

Zadáme-li jako n hodnotu menší než 2, tělo cyklu neproběhne ani jednou.

Nás první pokus o program pro výpočet faktoriálu by tedy mohl vyypadat např. takto:

```
public class Fakt {
    public static void main(String[] arg)
    {
        System.out.print("Zadej cele cislo: ");
        int n = Mojel0.inInt(); // Vstup dat
        int s = 1;             // V s bude výsledek
        while(n > 1)          // Cyklus pro výpočet
        {
            s = s*n;          // součinu n*(n-1)*...*
            n = n-1;           // Změna hodnoty n
        }
        System.out.println("Jeho faktorial je " + s);
    }
}
```

Program si nejprve vyžádá jedno celé číslo, to pak přečte do proměnné `n` a pokusí se vypočítat jeho faktoriál. Nakonec vypočtenou hodnotu vypíše.

Zadáme-li záporné číslo, 0 nebo 1, tělo cyklu neproběhne ani jednou a program vypíše 1. Pro 0 a 1 je to správné, pro záporná čísla ovšem nikoli. Pokusíme se to napravit a použijeme nejjednodušší možnost: Zadá-li uživatel programu záporné číslo, vypíšeme zprávu o chybě a program ukončíme.¹⁵ K okamžitému ukončení programu lze kdekoliv použít příkaz `System.exit(1);`.

**Rozhodování:
příkaz if**

Když dostaneme číslo `n`, musíme umět otestovat, zda je záporné, rovno 0 nebo 1 nebo větší než 0, a podle toho použít různé postupy. K tomu používáme v Javě mj. podmíněný příkaz `neboli` příkaz `if`. Jeho ne zcela přesný popis je

`if(podminka) { příkazy_1 } else { příkazy_2 }`

nebo jen

`if(podminka) { příkazy_1 }`

Podminka je opět výraz, který je buď *pravdivý* (podmínka je splněna) nebo *nepravdivý* (podmínka není splněna).

Je-li *podminka* splněna, provede se část programu, zde označená *příkazy_1*, a část *příkazy_2* – pokud jsme ji uvedli – se přeskočí. Není-li *podminka* splněna, přeskočí se část *příkazy_1* a provede se část programu, zde označená *příkazy_2*; Pokud tuto část vynecháme, neprovede se nic. Tyto dvě „větve“ příkazu `if` budeme uzavírat mezi složené závorky (`{ a }`).

Větve *příkazy_1* nebo *příkazy_2* může být i prázdná, tzn. můžeme tu uvést pouze prázdné složené závorky (`{}).`

Vraťme se nyní k výpočtu faktoriálu. Budeme-li tedy mít v proměnné `n` hodnotu, jejíž faktoriál chceme spočítat, můžeme postupovat např. takto:

```
if(n < 0)
{
    System.out.println("Není definovan");
    System.exit(0); // ukončí program
}
else
{
    // ... Vypočítej faktoriál n a vypiš výsledek
}
```

¹⁵ Ukončení programu při chybě je zpravidla velice špatné řešení. Jsou ale situace, kdy je nezbytné, a proto si ho zde ukážeme. Časem (v 10. kapitole) se naučíme řešit podobné situace elegantněji.

Bude-li n obsahovat např. hodnotu -1, provede se první větev a druhá se přeskočí. Bude-li n obsahovat nezáporné číslo, např. 5, přeskočí se v tomto příkazu i f první větev a provedou se příkazy ve druhé větví.

Celý program může vypadat např. takto:

```
/* Soubor Kap03\05\Fakt.java
   výpočet faktoriálu, druhý pokus */

public class Fakt {
    public static void main(String[] args) {
        System.out.print("Adej cele cislo: ");
        int n = Mojel0.readInt(); // Vstup hodnoty
        int s = 1; // Promenná s bude obsahovat výsledek
        if(n < 0)
        {
            System.out.println("Neni definovan");
            System.exit(0);
        } else {
            while(n > 1) // Cyklus, ve kterém se faktoriál vypočte
            {
                s = s*n;
                n = n-1;
            }
            System.out.println("Jeho faktorial je " + s);
        }
    }
}
```

Tento program najdete na WWW v souboru Kap03\05\Fakt.java.

Užitečné operátory

Náš program sice funguje, ale takto by ho žádný programátor v Javě nenapsal. (Každý programovací jazyk má své idiomy – vžití fráze – a patří k hrdosti programátorů je znát a používat.) Podivejme se tedy, jak by se dal upravit. Přitom se seznámíme s několika užitečnými operátory. (V následujícím textu bude znak @ zastupovat některý z operátorů +, -, / atd.)

Operátory @~

Výraz

s = s * n;

obsahuje proměnnou s na obou stranách přiřazení. Java ho umožňuje zapsat ve zkráceném tvaru

s *= n;

který znamená přesně totéž. (Kromě zkráceného zápisu ovšem usnadňuje také překladači vytvoření rychlejšího programu.) Podobně mů-

žeme v Javě používat operátory `+=`, `-=-`, `/=`, `%=` a některé další.¹⁶ Je-li @ některý z operátorů `+`, `-`, `*`, `/`, `%` (a příp. některých dalších), znamená výraz

```
s @= x;
```

vždy totéž co

```
s = s @ x;
```

Cyklus pro výpočet součinu $1 \cdot 2 \cdot \dots \cdot n$ by proto mohl vypadat např.

```
while(n > 1)
{
    s *= n;
    n -= 1;
}
```

Jenže zvětšení nebo zmenšení proměnné o 1 jsou operace natolik časté, že pro ně Java zavádí zvláštní operátory `++` a `--`. Navíc tyto operátory existují ve dvou variantách: jako prefixové (zapisují se před operand, tedy před proměnnou, na kterou mají působit) a jako postfixové (zapisují se za operand). Postfixový operátor způsobí, že se ve výrazu nejprve použije nezměněná hodnota proměnné a teprve pak se tato hodnota zvětší nebo zmenší. Prefixový operátor způsobí, že se hodnota proměnné nejprve zvýší, a pak se tato změněná hodnota použije ve výrazu.

Proto místo `n -= 1` můžeme napsat pouze `n--` nebo `--n`. Zde nezáleží na tom, zda použijeme prefixový nebo postfixový operátor, neboť nám jde jen o změnu hodnoty `n`. Cyklus `while` pak můžeme zapsat

```
while(n > 1)
{
    s *= n;
    n--;
}
```

Můžeme však jit ještě dál: Řekli jsme si, že postfixový operátor `--` způsobí, že se ve výrazu nejprve použije nezměněná hodnota proměnné, a teprve pak se tato hodnota zvětší. To znamená, že tento operátor můžeme přesunout dovnitř výrazu `s *= n`, a to jako postfixový (použije se původní hodnota, teprve pak se `n` zmenší).

Konečný (alespoň prozatím) tvar příkazu `while` tedy bude

```
while(n > 1)
{
    s *= n--;
}
```

¹⁶ Později si samozřejmě řekneme, o které jde.



Zdrojový text programu v této podobě najdete na WWW v souboru Kap03\05\lakt3.java.

Metoda faktorial()

Faktoriál celého čísla tedy už umíme vypočítat. To je skvělé, ale ihned se vnučuje otázka: Ve větším programu budeme faktoriál počítat nejspíš na několika místech. Přece nebudeme pokaždé znova opisovat týž kus programu??

To jistě ne. Kdybychom totiž později chtěli výpočet opravit nebo upravit, museli bychom zasahovat do programu na mnoha místech, a tím by se zvětšovala pravděpodobnost, že někde uděláme chybu. Navíc bychom snadno mohli na některé z těchto míst zapomenout.

Proto si naprogramujeme výpočet faktoriálu jako samostatnou metodu, kterou nazveme `faktorial()`. Nebude to nic těžkého, musíme se pouze naučit dvě věci:

- Jak takové metodě předat hodnotu, se kterou má počítat,
- jak vrátit vypočtený výsledek.

Hlavička metody

Všechny důležité informace o metodě jsou obsaženy v její hlavičce. Její zjednodušený popis je:

specifikátory typ identifikátor(formální parametry)

Zde specifikátory mohou být např. `public static`, jako jsme to viděli u metody `main()` v předchozí kapitole. Typ určuje typ výsledku; v našem případě to bude `int`, neboť faktoriál je celočíselná hodnota. V případě metody `main()` to bylo `void`, neboť tato metoda nevracela žádný výsledek.¹⁷

Identifikátor je jméno metody; v našem případě to bude `faktorial`. Za identifikátorem jsou kulaté závorky a v nich specifikace formálních parametrů. Ta říká, jaké vstupní hodnoty a jakých typů musíme tomuto dílčímu algoritmu předat, aby mohl vypočítat požadovaný výsledek. Specifikace formálních parametrů vypadají podobně jako deklarace proměnných, nesmějí ovšem obsahovat inicializaci. Je-li jich více, oddělíme je čárkami.

¹⁷ Jak metoda `main()`, tak i metoda `faktorial()` představují dílčí algoritmy. Podstatný rozdíl mezi nimi spočívá v tom, že metoda `faktorial()` dostane data, vypočte z nich jednu hodnotu, a zápis metody `faktorial()` v programu bude tuto hodnotu představovat. Chová se tedy podobně jako funkce v matematickém vzorci. Naproti tomu metoda `main()` sice dostane data a něco s nimi udělá, ale jejím výsledkem není žádná hodnota a její zápis žadou hodnotu nepředstavuje.

Hlavička metody faktorial() tedy bude mít tvar

```
public static int faktorial(int n)
```

který říká:

- metoda faktorial() je statická (je označena modifikátorem static – lze ji volat, i když neexistuje žádná instance třídy, ke které patří),
- je veřejně přístupná (public – smí ji volat kdokoli a odkudkoli),
- vrací hodnotu typu int,
- očekává jeden parametr typu int.

Tato hlavička ve skutečnosti sděluje ještě další informace, které prozatím ponecháme stranou. Poznamenejme, že pokud metoda nemá žádné parametry, musíme v její hlavičce uvést prázdné závorky.

Příkaz return

Zbývá vyřešit poslední důležitou drobnost: Jak v těle metody označit hodnotu, která představuje výsledek? Podíváme-li se na výpočet faktoriálu v předchozím příkladu, uvidíme v něm proměnné n a s; musíme počítací nějak sdělit, že skutečným výsledkem je s.

K tomu slouží příkaz return, který má tvar

```
return výraz;
```

a znamená: Zde končí výpočet v těle metody. Vrať se na místo, odkud byla tato metoda volána, a výraz je výsledek.¹⁸

Vlastní výpočet faktoriálu zůstane stejný, a proto si už můžeme zkusit program se samostatnou metodou faktorial() napsat. Abychom si usnadnili další výklad, přejmenujeme proměnnou, deklarovanou v metodě main() – zde se bude jmenovat m.

```
/* Soubor Kap03\05\Fakt3.java
   výpočet faktoriálu, třetí pokus
   samostatná funkce pro výpočet faktoriálu
*/
public class Fakt3 {
    public static int faktorial(int n)
    {
        int s = 1; // Proměnná s bude obsahovat výsledek
        if(n < 0)
        {
            System.out.println("Není definovan");
            System.exit(0);
        }
    }
}
```

¹⁸ V metodách typu void lze použít příkaz return;, který neobsahuje výraz. Znamená pouze „zde provádění metody končí, vrať se na místo, odkud byla volána“.

```

        } else { // Cyklus, ve kterém se faktoriál vypočte
            while(n > 1)
            {
                s *= n--;
            }
        }
        return s;
    }

    public static void main(String[] arg)
    {
        System.out.print("Zadej cele cislo: ");
        int m = MojeIO.readInt(); // Vstup hodnoty
        System.out.printf("Jeho faktorial je %d\n", faktorial(m));
    }
}

```

Všimněte si, že oddělení výpočtu faktoriálu do samostatné metody vedlo i ke zpřehlednění celého programu: Metoda `main()` se nyní stará již jen o vstup a výstup – tedy o komunikaci programu s uživatelem – a k výpočtu používá nástrojů, které jsme implementovali zvlášť.

Poznamenejme, že ve starších verzích JDK musíme poslední příkaz v těle metody `main()` nahradit příkazem

```
System.out.println("Jeho faktorial je " + faktorial(m));
```

Volání metody

Podívejme se nyní na použití metody `faktorial()` v metodě `main()`, na příkaz

```
System.out.printf("Jeho faktorial je %d\n", faktorial(m));
```

Zápis `faktorial(m)` říká: *Vezmi hodnotu m, předej ji metodě faktorial() a na místo tohoto zápisu dosad výsledek. Proměnná m zde vystupuje jako tzv. skutečný parametr, tedy jako hodnota, kterou metodě předáváme a z níž má výsledek vypočítat.*

Metoda `faktorial()` tuto hodnotu dostane ve formálním parametru `n`, vypočte, co se po ní požaduje, a výsledek vrátí. *Tento výsledek je v metodě main() představován zápisem faktorial(m), takže zadáme-li na vstupu hodnotu 5, vytiskne tento program výsledek 120. Zápis této metody tedy představuje vypočtenou hodnotu.*

Formální parametr `n` je vlastně proměnná lokální v metodě `faktorial()`, která bude mít na počátku hodnotu skutečného parametru. Jestliže tuto metodu zavoláme zápisem `faktorial(8)`, bude mít formální parametr `n` počáteční hodnotu 8.

Magická čísla

Budete-li chvíli experimentovat s metodou `faktorial()`, zjistíte, že pro n větší než 12 nedává správné výsledky. Proč, na to se podíváme v 6. kapitole; zatím se omezíme na konstatování, že to tak je, a pokusíme se zajistit, aby se programátor, který tuto metodu používá, dozvěděl, že zadal příliš velký parametr.

Můžeme samozřejmě zařadit jako první v těle této metody příkaz

```
if(n > 12)
{
    System.out.println("Prilis velke n");
    System.exit(0);
}
```

jako jsme to udělali v případě záporné hodnoty. Jistý problém se ale skrývá v čísle 12. Tedy je nám sice jasné, o co jde, ale bude nám to jasné i za týden, až budeme tento program upravovat? Co když zjistíme, jak tuto funkci naprogramovat, aby vracela správné výsledky i pro $n < 20$? Složitější program může obsahovat řadu konstant, a nikde není psáno, že některá z nich nemůže mit také hodnotu 12, i když bude znamenat úplně něco jiného – třeba počet měsíců v roce; pak budeme muset nad každou dvanáctkou zdlouhavě přemýšlet, zda jí máme opravit. (Proto se takové konstanty označují jako „magická čísla“ a programátoři se jim vyhýbají.)

Bude tedy lepší, když tuto konstantu pojmenujeme (přidělíme jí identifikátor) a všude, kde ji potřebujeme, použijeme tento identifikátor. Budeme-li ji chtít později změnit, stačí změnit jediné místo v programu – deklaraci konstanty. Program se bude snáze udržovat a bude přehlednější.

V Javě se konstanty deklarují podobně jako proměnné, pouze se před deklarací připojí slovo (modifikátor) `final`, které říká, že hodnotu této proměnné nelze měnit. Předchozí úsek metody `faktorial()` tedy přepíšeme do tvaru

```
final int HORN1_MEZ = 12;
if(n > HORN1_MEZ)
{
    System.out.println("Prilis velke n");
    System.exit(0);
}
```

3.5 Pozdrav z Marsu

Až dosud nám třídy sloužily jen jako jakýsi „obal“ pro statické metody; to je ale v Javě – a v OOP vůbec – naprostě netypické. Proto si na závěr této kapitoly ukážeme, jak se v Javě pracuje s objekty.

Napišeme program, který bude podobný našemu vůbec prvnímu programu: Bude vypisovat text, bude ovšem mít na vybranou z několika možností. Hlavní třídu svého programu pojmenujeme `Napis` a bude obsahovat pouze metodu `main()`.

Třída Text K uložení textu slouží v Javě předdefinované třídy `String` a `StringBuffer`. Protože si však chceme ponechat možnost některé vlastnosti upravit odlišně od standardní implementace, definujeme si další třídu, kterou nazveme `Text`. Bude obsahovat jednu datovou složku typu `String`, kterou nazveme `text`.¹⁹ Deklarace této třídy bude ve stejném souboru jako deklarace třídy `Napis`; může začínat takto:

```
class Text {  
    private String text;  
    // ...  
}
```

Všimněte si, že jsme zde nepoužili modifikátor `public`. V každém souboru smí být jen jedna třída s modifikátorem `public` a ta se musí jmenovat stejně jako soubor, ve kterém je uložena.

Modifikátorem `private` před složkou `text` říkáme, že tato složka je soukromá, tj. nesmí ji používat nikdo kromě metod třídy `Text`.

Metody Podívejme se ještě na operace, které budeme s instancemi provádět:

- Budeme chtít změnit text uložený v instanci.
- Budeme chtít vytvořit instanci obsahující zadaný text.
- Budeme chtít vypsat do standardního výstupu text uložený v instanci.

Metoda `zmenText()`, která změní text uložený v instanci, bude velice jednoduchá:

```
public void zmenText(String s) { text = s; }
```

(Za chvíli se dozvímme, co se za tímto přiřazením skrývá.)

Konstruktor K vytvoření instance slouží zvláštní metoda, která se nazývá *konstruktor*; má vždy stejné jméno jako třída, jejiž instance vytváří, a nesmíme u něj specifikovat typ vrácené hodnoty (ani `void`). Nesmíme také použít modifikátor `static`.

Konstruktor se obvykle stará o inicializaci (přidělení počátečních hodnot) datových složek instance. To znamená, že konstruktor třídy `Text` by mohl mít např. tvar

¹⁹ Připomeňme si, že `text` a `Text` jsou dva různé identifikátory, neboť se liší velikostí písmen.

```
public Text(String s){ text = s; }
```

Je ale rozumnější neprogramovat jednu věc dvakrát, a proto použijeme už hotovou metodu `zmenText()`:

```
public Text(String s) { zmenText(s); }
```

Bylo by ovšem vhodné mít také konstruktor bez parametrů, který vloží do proměnné `text` nějaký implicitní text. To může na první pohled vypadat jako problém: Jméno konstruktoru je přece předepsáno a jeden konstruktor jsme už definovali.

Přetěžování metod

Přesto to jde. Java totiž umožňuje tzv. *přetěžování metod*: Několik metod téže třídy může mít stejné identifikátory, pokud se liší počtem nebo typem parametrů (nebo obojím). To znamená, že nám nic nebrání definovat ve třídě `Text` ještě jeden konstruktor, např.

```
public Text() { zmenText("Ahoj, lidi"); }
```

Metoda pro vypsání textu, kterou nazveme *vtipně vypis()*, prostě vypíše text do standardního výstupu. Celá definice třídy `Text` tedy může mít tvar

```
/* Soubor Kap03\06\Napis.java  
první pokusy s objekty  
*/  
  
class Text {  
    private String text;  
    public Text() { zmenText("Ahoj, lidi"); }  
    public Text(String s) { text = s; }  
    public void zmenText(String s) { text = s; }  
    public void vypis() { System.out.println(text); }  
}
```

Metoda `vypis()` vám může připadat jako zbytečná zátěž počítače: Proč rovnou nezavolat `System.out.println()`, proč ztrácet čas voláním metody, která to teprve udělá?

Zapouzdření

V tuto chvíli mohou podobné námitky vypadat jako opodstatněné. Jenže tato metoda zvládne v prostředí Windows pouze výpis textu bez diakritických znamének – tedy nejlépe angličtiny.

Ale co když se později rozhodneme, že chceme pracovat s českými texty? (Český uživatel programu má samozřejmý nárok, aby s ním jeho program mluvil česky, takže skutečnost, že to náš program neučí, lze omluvit jedině tim, že jsme začátečníci.) V okamžiku, kdy se rozhodneme předělat náš program tak, aby uměl do konzolového okna Windows psát česky, mám bude stačit přepsat právě tuto metodu, nic více. Kdybychom neměli pro výpis textu samostatnou metodu, museli bychom opravovat všechna místa, kde se tento nápis vypisuje – a na nějaké bychom nejspíš zapomněli, někde bychom to pokazili...

Takto je operace výpisu dobře izolována, její implementace je skryta před uživatelem této třídy. Pokud se nezmění rozhraní (tj. hlavička) metody, nedotkne se změna implementace této metody ostatních částí programu. O tom jsme hovořili již v úvodní kapitole, ve výkladu o zapouzdření.

5 Poznamenejme, že v Javě můžeme v metodě `vypis()` také použít metodu `printf()`:

```
public void vypis() { System.out.printf("%s\n", text); }
```

První parametr, tzv. formátovací řetězec, obsahuje specifikaci konverze `%s`, která říká, že odpovídající parametr představuje odkaz na řetězec (instanci třídy `String`). Za touto specifikací následuje `\n`, řídicí posloupnost pro znak představující přechod na nový řádek.

Statické
a nestatické
metody

Všimněte si také, že ani jedna z metod ve třídě `Text` není statická (nemá modifikátor `static`). To proto, že všechny metody, které zde máme, budou pracovat s jednotlivými instancemi. Statické metody se hodí, když nepracujeme s instancemi – a to je ve skutečnosti málokdy. Jedinou výjimkou je metoda `main()`, neboť ta se volá ještě dříve, než se v programu vytvoří jakákoli instance jakékoli třídy. Musí tedy být statická.

**Vytváření
instancí**

Nyní se podívejme, jak se v Javě vytvářejí instance tříd. Nejprve si musíme říci, že s instancemi pracujeme pomocí tzv. odkazů. Napíšeme-li

```
Text t;
```

nevytvoří se tím instance třídy `Text`, ale pouze odkaz²⁰, který může na instanci třídy `Text` odkazovat, ovšem zatím na žádnou instanci neodkazuje, neboť jsme mu žádnou hodnotu nepřiřadili. Novou instanci vytvoříme pomocí operátoru `new`, za kterým zapíšeme volání konstruktu. Například takto:

```
t = new Text("Mars zdraví");
```



Obr. 3.1 Instance a odkaz na ni

²⁰ Pokud znáte C, C++ nebo Pascal, pak vězte, že jde vlastně o ukazatel. Všechny instance objektových typů jsou v Javě dynamické.

Operátor `new` vytvoří pomocí konstruktoru novou instanci a vrátí odkaz na ni; tento odkaz přiřadíme proměnné `t`. (Můžeme si představovat, že odkaz je vlastně adresa instance.) Situaci znázorňuje obrázek 3.1.

Máme-li odkaz na instanci, můžeme pro ni zavolat některou z metod. Zápis volání metody má tvar

`odkaz_na_instanci.identifikátor(parametry)`

kde `identifikátor` je identifikátor metody. K odkazu na instanci je připojen operátorem tečka, takže se občas hovoří o „tečkové notaci“ nebo o „kvalifikaci“.

Budeme-li chtít vypsat text uložený v instanci, na kterou odkazuje `t`, napišeme

`t.vypis();`

Vynechání kvalifikace To odporuje našim dosavadním zkušenostem: v minulých příkladech jsme metodu volali tak, že jsme prostě zapsali její identifikátor, nic více. Jak to tedy je?

Odkaz na instanci můžeme vynechat, jestliže metodu voláme uvnitř jiné metody téže instance. Podívejme se znova na konstruktor

```
public Text(String s) { zmenText(s); }
```

Také zde jsme napsali jen `zmenText()`. To proto, že voláme jednu metodu uvnitř jiné a obě mají pracovat se stejnou instancí; překladač si tedy jméno instance doplní sám. (Naše předešlé příklady obsahovaly vždy jen jednu třídu, takže kvalifikace jménem instance byla zbytečná.)

Dokončení programu

Nyní už můžeme svůj program dokončit.

```
/* Soubor Kap03\06\Napis.java
   první pokusy s objekty
   dokončení
*/
public class Napis
{
    public static void main(String[] args)
    {
        Text n1 = new Text();
        Text n2 = new Text("Pozdrav z Marsu");
        n1.vypis();
        n2.vypis();
        Text n3 = n1;
        n3.vypis();
        n1.zmenText("Pozor, tunel");
        n1.vypis();
```

```
    r3.vypis();
}
}
```



Tento program najdete na WWW v souboru Kap03\06\Napis.java. Po přeložení a spuštění vypíše

```
Ahoj, lid!
Pozdrav z Marsu
Ahoj, lidí
Pozor, tunel
Pozor, tune!
```

První dva řádky ukazují, že každá z instancí obsahuje svou vlastní datovou složku `text`.

Garbage collector

Pokud znáte C, C++ nebo Pascal, už vás nejspíš napadlo, jak to je s úklidem. Vytvořili jsme dynamické instance, ale kdo je po nás uklidí? Tedy – kdo je zruší, kdo uvolní jejich paměť?

Odpověď se může zdát nezvyklá: O to se postará JVM. Programátor v Javě sice nové instance vytváří, ale neruší je, vůbec takovou možnost nemá. Virtuální stroj si totiž vede evidenci odkazů, a pokud zjistí, že na některý objekt neexistuje žádný odkaz, prostě tento objekt zruší (obvykle ne hned, ale až ve chvíli, kdy se mu to bude hodit.) Toto zařízení se nazývá *garbage collector*, česky se označuje jako *automatická správa paměti* nebo třeba *sběrač neplatných objektů*.

Automatická správa paměti sice může znamenat určité zpomalení běhu programu, ale na druhé straně vylučuje celou řadu chyb, se kterými se běžně potýkají programátoři v jazycích jako je C, C++ nebo Pascal.

Přiřazování odkazů

Podivejme se ještě na přiřazení

```
n3 = n1;
```

ve kterém obě proměnné, `n1` i `n3`, představují odkazy na instanci typu `Text`. Při takovémto přiřazení se nevytvoří kopie instance, na kterou `n1` ukazuje, ale přenese se odkaz. To znamená, že `n1` i `n3` budou po provedení tohoto příkazu odkazovat na tutéž instanci – viz obr. 3.2.

Z toho plyne, že když příkazem

```
n1.zmenText("Pozor, tunel");
```

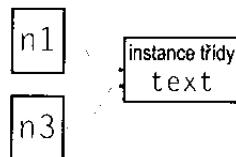
změníme obsah instance, na kterou odkazuje `n1`, změní se i obsah instance, na kterou odkazuje `n3`, a příkaz

```
n3.vypis();
```

vypíše

Pozor, tunel

protože n3 odkazuje na tutéž instanci jako n1.



Obr. 3.2 Přiřazování odkazů



Proměnné typu odkaz mohou také obsahovat zvláštní hodnotu, vyjádřenou slovem null. Tato hodnota znamená, že daný odkaz neukazuje na žádný platný objekt.

Jak je to s texty

Už jsme se zmínili, že String je jedna z předdefinovaných tříd jazyka Java, která slouží k práci se znakovými řetězci. To ale znamená, že deklarace

```
private String text;
```

vytváří opět odkaz na instanci třídy String, nikoli instanci.²¹ Odkaz také předáváme metodě změnText(String s). Proto, když vytvoříme instanci třídy Text příkazem

```
Text n2 = new Text("Pozdrav z Marsu");
```

stane se několik věcí:

- Vytvoří se instance třídy String, obsahující text v uvozovkách.
- Odkaz na tuto instanci se předá konstruktoru jako parametr s.
- V těle konstruktoru se tento odkaz přifadí proměnné (datové složce) text.
- Po ukončení konstruktoru jeho parametr s zanikne.

Jestliže pak zavoláme pro instanci, na kterou odkazuje n2, metodu změnText() příkazem

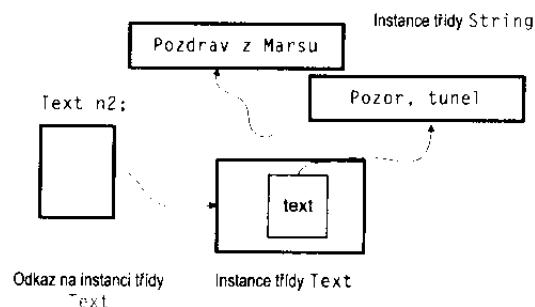
```
n2.změnText("Pozor, tunel");
```

proběhnou následující operace:

²¹ Měli bychom tedy důsledně hovořit o „řetězci, na který odkazuje proměnná text,“ nikoli o „řetězci text“. Nicméně pokud nebude hrozit nedorozumění, budeme si tuto zkratku dovolovat. Podobně budeme často hovořit o „instanci x“ místo o „instanci, na kterou odkazuje x“.

- Vytvoří se instance třídy `String`, obsahující text "Pozdrav, tunel".
- Odkaz na tuto instanci se předá metodě `zmenText()` jako její parametr `s`.
- V těle metodě `zmenText()` se tento odkaz přiřadí proměnné (datové složce `text`); tím se přepíše odkaz na instanci třídy `String`, která obsahovala řetězec "Pozdrav z Marsu". Pokud šlo o jediný odkaz, tato instance zanikne.

Situaci ukazuje obr. 3.3.



Obr. 3.3. Změna textu uloženého v instanci

4 Složitější příklad

V předchozí kapitole jsme na řadě jednoduchých příkladů poznali několik základních programových konstrukcí Javy. Nyní se pustíme do složitějšího příkladu, ve kterém se seznámíme s některými dalšími možnostmi jazyka, především s používáním objektů, a s některými knihovními třídami implementujícími datové struktury. Pokud vám bude tato kapitola při prvním čtení připadat příliš složitá, klidně ji přeskočte a vraťte se k ní později.

4.1 Počítání slov poprvé

Naším prvním úkolem bude napsat program, který přečte ze vstupu textový soubor a vypíše všechna různá slova, která v něm najde. Pro jednoduchost budeme různé gramatické tvary jednoho slova pokládat za různá slova.

Nejprve bychom si měli ujasnit, jak budeme postupovat. Program bude číst data ze standardního vstupu, a to po řádcích; nic jiného totiž zatím neumíme. Každý přečtený řádek pak rozložíme na jednotlivá slova a ta si zapamatujeme. Nakonec jednotlivá slova vypíšeme.

Třída Analyzer

Hlavní třídu svého programu nazveme `Analyzer`. Její metoda `main()` nejprve vytvoří instanci této třídy a pak spustí metodu `beh()`, která se postará o vše potřebné:

```
class Analyzer {  
    // ... deklarace ostatních metod  
  
    public static void main(String[] s)  
    {  
        Analyzer a = new Analyzer(); // Vytvoř instance  
        a.beh(); // a spusť běh  
    }  
}
```

Běh programu: výjimky

Čtení dat Čtení ze souboru zdánlivě nepředstavuje problém; využijeme metodu `MojeIO.inStr()`, která vrací odkaz na `String`, a přesměrování stan-

dardního vstupu. Když metoda `MojeIO.inStr()` narazí na konec vstu-
pu, vrátí null.

Odkaz na přečtený řádek si uložíme do proměnné `radek`.

Výjimky Zde ale narazíme na první problém: Deklarujeme-li v našem programu
metodu

```
void beh()  
{  
    String radek;  
    radek = MojeIO.inStr();  
    // ...
```

ohlásí překladač chybu

`Analyzer.java:36: unreported exception java.io.IOException:
must be caught or declared to be thrown`

To znamená, že překladač zjistil, že v těle této metody může vznik-
nout tzv. výjimka (anglicky exception) a my o tom nevíme.

Oč jde?

Výjimka je chyba Stručně řečeno, výjimka je běhová chyba, která nastane, když např. procedura pro čtení narazí na chybu v souboru, když vydělíme nulou ap. Když k ní dojde, přeruší se přirozený běh programu, vytvoří se objekt s informacemi o této chybě a začne se hledat tzv. handler, místo v programu, které tuto chybu osetří.

Handler nemusí být v téže metodě, ve které chyba vznikla. V tom případě se program vrátí z metody, ve které výjimka vznikla, do metody, která ji zavolala, a v ní pokračuje v hledání handleru. Pokud ho nenajde ani tam, pokračuje v hledání v metodě, která zavolala tuto metodu atd. Tomu se říká *šíření výjimky*. Pokud ho nenajde ani v metodě `main()`, použije implicitní handler JVM, který vypíše zprávu o chybě a ukončí program.

V Javě ovšem platí, že pokud se z nějaké metody může výjimka rozšířit, musíme to v její deklaraci uvést. K tomu nám poslouží slovo `throws`, za kterým uvedeme typ výjimky (je-li jich více, oddělíme je čárkami).

Způsob ošetřování výjimek pomocí handlerů si vysvětlíme později; prozatím se spokojíme s tím, že výjimky „propustíme“, pouze pomocí klíčového slova `throws` ohlášíme, že o nich víme.

Metoda `beh()` tedy bude mít hlavičku

```
void beh() throws java.io.IOException {
```

Podobně doplníme throws java.io.IOException i do hlavičky funkce main(). Pokud dojde při čtení dat k chybě, program vypíše standardní chybové hlášení a skončí.

Kontejner

Další otázka, se kterou se musíme vypořádat, je, jak si zapamatovat jednotlivá nalezená slova; ta si budeme muset ukládat do vhodného *kontejneru*. Kdybychom předem znali počet různých slov, která se mohou v textu vyskytnout, mohli bychom použít pole, tedy skupinu proměnných stejného typu. To ale bohužel neznáme; musíme tedy použít některou z tzv. dynamických datových struktur, jejichž velikost se může měnit v závislosti na množství uložených dat.

ArrayList Jazyk Java (od verze 2) nabízí ve svých knihovnách řadu tříd, které lze k podobným účelům použít. Asi nejčastěji se používá **ArrayList**, kontejner, na který se můžeme dívat jako na pole s délkou, jež roste podle potřeby. Nové prvky můžeme do tohoto kontejneru přidávat pomocí metody **add()**, jejímž parametrem může být odkaz na libovolný objekt (tedy na instanci libovolné jazykové tridy²²).

Třída **ArrayList** je v tzv. balíku **java.util**, a to znamená, že její celé jméno je **java.util.ArrayList**. (To je sice otravně dlouhé, ale časem si ukážeme, jak si psaní zkrátit.)

Obsah tohoto kontejneru vypíšeme velice jednoduše – předáme ho jako parametr metodě **System.out.println()**.

Metoda beh() Přenecháme-li rozklad řádku na jednotlivá slova metodě **analyze()**, můžeme si už ukázat, jak bude metoda **beh()** vypadat.

```
void beh() throws java.io.IOException
{
    java.util.ArrayList slovník = new java.util.ArrayList();
    String radek = MojeIO.inStr(); // Přečti 1. řádek
    while(radek != null) // Dokud se čtení daří
    {
        radek = radek.trim(); // Odstraň okrajové mezery
        analyzuj(radek, slovník);
        radek = MojeIO.inStr(); // Čti další řádek
    }
    System.out.println(slovník); // Vypiš výsledek
}
```

V prvním řádku vytvoříme instanci **slovník** třídy **ArrayList**, ve druhém deklarujeme proměnnou **radek** a uložíme do ní odkaz na instanci

²² Formálním parametrem metody **add()** je odkaz na **Object**.

vytvořenou metodou `MojeIO.inStr()`. Tato instance bude obsahovat načtený řádek.

V následujícím cyklu whilé vždy z řádku nejprve odstraníme úvodní a koncové mezery a pak jej předáme metodě `analyzuj()`. Na závěr cyklu se pokusíme přečíst další řádek. Pokud se to nepodaří, vrátí metoda `MojeIO.inStr()` hodnotu null a cyklus skončí.

Analýza řádku

Další problém, se kterým se musíme vypořádat, bude rozklad textu na jednotlivá slova. Uvidíme však, že třída `String` je velmi důmyslná a udělá řadu věcí za nás.

Třída String: Už víme, že instance této třídy obsahuje znakový řetězec. Celkový počet znaků v řetězci můžeme zjistit pomocí metody `length()`. Jednotlivé znaky jsou očíslovány (indexovány), takže první znak má číslo 0, druhý 1 atd.²³; poslední znak má číslo `length()-1`.

Znak na i-té pozici můžeme zjistit pomocí metody `charAt(i)`. Pomocí metody `indexOf()` můžeme zjistit pozici určitého znaku nebo podřetězce v daném řetězci.

Řetězec, uložený v instanci třídy `String`, ovšem nelze měnit. Pokud něco takového potřebujeme, musíme použít třídu `StringBuffer`.

Metoda equals(): Vyzbrojeni těmito vědomostmi se nyní můžeme pustit do rozkladu řádku na slova. Nejprve se ovšem přesvědčíme, zda jsme nedostali řádek, který obsahuje prázdný řetězec. V takovém případě není co řešit, a proto se ihned vráťme. K porovnání řetězce radek s prázdným řetězcem "" použijeme metodu `equals()`, která porovnává dva řetězce znak po znaku:

```
if(radek.equals("")) { return; }
```

Kdybychom k porovnávání použili operátor ==, tj. kdybychom napsali
`if(radek == "") { return; }`

porovnávaly by se odkazy (tedy vlastné adresy řetězců v paměti). Nám však nejdě o to, zda instance, na kterou odkazuje radek, a instance "" leží na témže místě v paměti, ale o to, zda obsahují stejné znaky (v našem případě zda neobsahují žádné znaky).

Při vlastním rozkladu řádku se budou opakovat dvě operace:

- přeskočení oddělovačů,

²³ Jde vlastně o pole znaků.

- čtení slova znak po znaku, dokud nenarazíme na oddělovač nebo konec řádku, a jeho ukládání do vhodného kontejneru.

Přeskočení oddělovačů naprogramujeme zvlášť, jako metodu `preskočOddělovace()`. Tato metoda bude vracet index znaku, ve kterém začíná další slovo, nebo `-1`, jestliže narazi na konec řádku. Jako parametry dostane odkaz na analyzovaný řádek a index znaku, od kterého má začít. Odpovídající úsek metody `analyzuj()` tedy bude mít tvar

```
int i = 0; // Index znaku v řádku
StringBuffer slovo = null; // Sem uložíme získané slovo
i = preskočOddělovace(radek, i);
while(i >= 0)
{
    // Přenes následující slovo do instance "slovo"
    // Ulož slovo do kontejneru
    i = preskočOddělovace(radek, i);
}
```

Komentáře v těle cyklu zatím nahrazují skutečné operace.

Slova, která v řádku najdeme, budeme znak po znaku přenášet do instance `slovo`: protože však třída `String` nedovoluje měnit uložený řetězec, musíme použít `StringBuffer`. V cyklu nejprve vytvoříme novou instanci třídy `StringBuffer` obsahující prázdný řetězec,

```
slovo = new StringBuffer("");
```

a pak do něj budeme pomocí metody `append()` přidávat znaky jeden po druhém, dokud nenarazíme na oddělovač. Už víme, že znak, který je na `i`-té pozici, zjistíme voláním metody `charAt(i)`. Pozici znaku v řetězci můžeme zjistit pomocí metody `indexOf()`, která – pokud daný znak najde – vrátí nezáporné číslo, jinak vrátí `-1`. Cyklus, ve kterém zpracujeme jedno slovo, tedy můžeme zkoušit naprogramovat takto:

```
while(oddělovace.indexOf(radek.charAt(i)) == -1)
// Dokud to není oddělovač
{
    slovo.append(radek.charAt(i++));
}
```

Na počátku obsahuje `i` index prvního znaku slova. Podmínka

```
oddělovace.indexOf(radek.charAt(i)) == -1
```

testuje, zda nejdě o oddělovač (tj. zda se nevyskytuje v řetězci `oddělovace`). Pokud ne, vstoupí program do těla cyklu, vezme znak na `i`-té pozici a připojí ho k řetězci `slovo`.

To bude fungovat, dokud nedospějeme na konec řádku. Po přečtení posledního znaku se `i` zvětší o jedničku a znova se zavolá metoda

`charAt(i);` její použití za koncem řetězce ovšem způsobí chybu. To znamená, že v podmínce opakování musíme testovat dvě věci: Nejprve se musíme vždy nejprve přesvědčit, že `i` je menší než délka řetězce, a teprve pak, že `i`-tý znak není oddělovač. Chceme-li zjistit, zda platí dvě podmínky zároveň, použijeme v Javě operátor konjunkce `&&`:

```
while((i < radek.length())
      && (oddelovace.indexOf(radek.charAt(i)) == -1))
{
    slovo.append(radek.charAt(i++));
}
```

Neúplné vyhodnocení

Nyní nás možná napadne, že jsme si přiliš nepomohli: Vypadá to, že chyba stejně nastane, neboť `i` když bude `i == radek.length()`, vyhodnotí se i druhá část podmínky a zavolá se metoda `charAt()`.

Naštěstí se však mylím, neboť je-li první část výrazu s operátorem `&&` nepravidlivá, nebude se druhá část vůbec vyhodnocovat, protože to není třeba – výsledek už bude určitě nepravidlivý.²⁴

Poslední část, která nám zbývá, je uložení slova do kontejneru `slovnik`. To je jednoduché: Nejprve vytvoříme z instance třídy `StringBuffer` instanci třídy `String` se stejným obsahem (k tomu poslouží jeden z konstruktorů třídy `String`), pak pomocí metody `indexOf()` zjistíme, zda v instance `slovnik` dané slovo už není, a pokud ne, přidáme ho tam pomocí metody `add()`.

```
String s = new String(slovo);
if(sezn.indexOf(s) == -1) {sezna.add(new String(s));}
```

Na závěr si ukážeme celou metodu `analyzuj()`:

```
void analyzuj(String radek, java.util.ArrayList sezna)
{
    if(radek.equals("")){return;} // Prázdný řádek - vrat' se
    int i = 0; // Index znaku v řádku
    StringBuffer slovo = null; // Sem uložíme získané slovo
    i = preskocOddelovace(radek, i);
    while(i >= 0) // Dokud není konec řádku
    {
        slovo = new StringBuffer("");
        // Přenes následující slovo do instance "slovo"
        while((i < radek.length()) &&
              (oddelovace.indexOf(radek.charAt(i)) == -1))
        {
            // Dokud to není oddělovač
            slovo.append(radek.charAt(i++));
        }
        if(sezn.indexOf(slovo) == -1)
        {
            sezna.add(slovo);
        }
    }
}
```

²⁴ Tomu se říká *neúplné vyhodnocování logického výrazu*: Vyhodnotí se jen část, která je nezbytná k určení výsledku. Setkáme se s ní ještě u operátoru `||`, který znamená disjunkci (logické „nebo“, platí alespoň jedna část).

```

    // Ulož slovo do slovníku
    String s = new String(slovo);
    if(sezn.indexOf(s)==-1){seznam.add(new String(s));}
    i = preskočOddelovace(radek, i);
}

```

Přeskakujeme oddělovače

Posledním úkolem, který musíme ještě vyřešit, je přeskocít oddělovače. Abychom tyto znaky mohli v textu snadno vyhledávat, uložíme si je do zvláštní datové složky třídy Analyzer:

```
String oddelovace = " .,:!?";
```

Metoda preskočOddelovace(String rad, int od) bude mít jako první parametr analyzovaný řetězec a jako druhý parametr index znaku, od kterého má začít.

Prohlédneme-li si pozorně kód metody analyzuj(), zjistíme, že metoda preskočOddelovace() může být volána i v případě, že od je rovno délce řetězce. Proto nejprve zjistíme, zda není od \geq rad.length(), a pokud ano, vrátíme -1.

Pak zjistíme, zda patří od-tý znak mezi oddělovače, a pokud ano, zvětšíme od o jedničku. Pokud od dosáhne hodnoty rad.length(), znamená, to, že jsme prozkoumali celý řetězec a nenašli začátek dalšího slova, a proto vrátíme -1. Jinak vrátíme index prvního znaku, který nebude patřit mezi oddělovače.

Zdrojový kód metody preskočOddelovace() může vypadat takto:

```

int preskočOddelovace(String rad, int od)
{
    if(od >= rad.length()) { return -1; }
    while(oddelovace.indexOf(rad.charAt(od)) >= 0)
    {
        if(++od == rad.length()) { return -1; }
    }
    return od;
}

```

Podmínka oddelovace.indexOf(rad.charAt(od)) \geq 0 vezme znak s indexem od v řetězci rad a pomocí metody indexOf() zjistí, zda patří mezi oddělovače, tj. zda se vyskytuje v řetězci oddelovace. (Připomeňme si, že pokud tento znak není v řetězci oddelovace, vrátí metoda indexOf() hodnotu -1.)

V podmínce příkazu if v těle cyklu nejprve zvětšíme od o 1 (přejdeme na další znak) a pak si ověříme, že jsme nepřekročili horní mez řetězce. Pokud ano, neobsahuje řetězec žádné další znaky, jsme na jeho konci a vrátíme -1.

Jinak cyklus skončí, když v řetězci najdeme znak, který nepatří mezi oddělovače. V tom případě vrátíme hodnotu jeho indexu, která je uložena v od.



Celý program najdete na WWW v souboru Kap04\01\Analyzer.java. Jestliže tento program přeložíte a spustíte příkazem

```
java Analyzer < d.txt  
kde soubor d.txt obsahuje text  
Ahoj lidí  
Ahoj martani
```

vypíše

```
[Ahoj, lidí, martani]
```

Vidíte, že slovo Ahoj se vypsalo jen jednou, i když je ve vstupním souboru dvakrát. Poznamenejme, že takto – v hranatých závorkách, oddělené čárkami – se standardně vypisuje obsah kontejneru typu ArrayList.

5 Přeložíte-li tento program v JDK 5, dostanete varování

Note: Analyzer.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

které říká, že v souboru Analyzer.java jsme použili typově nezabezpečené operace. To ovšem není chyba, pouze nás tím překladač upozorňuje, že jsme nevyužili všech možností, které JDK 5 nabízí. Prozatím budeme toto varování ignorovat. Později v této kapitole si ukážeme, jak lze uvedený program v Javě 5 vylepšit.

String, nebo StringBuffer?

Nyní vás možná napadá, proč jsme do kontejneru slovník neukládali přímo instance třídy StringBuffer. (Jak víme, do předdefinovaných kontejnerů v Javě můžeme ukládat instance jakékoli třídy.) Zkusme tedy nahradit předposlední příkaz v metodě analyzuj() příkazem

```
if(sezn.indexOf(slovo) == -1) {sezn.add(slovo);}
```



a předchozí příkaz, kterým se vytváří pomocná instance s třídy String, vypustit. Program se bez problémů přeloží (viz soubor Kap04\02\Analyzer.java na WWW). Spustíme-li jej však příkazem

```
java Analyzer < d.txt
```

kde soubor d.txt obsahuje týž text, dostaneme výstup

```
[Ahoj, lidí, Ahoj, martani]
```

ve kterém se slovo Ahoj opakuje. Proč?

Metoda `indexOf()`, kterou používáme v metodě `analyzuji()` ke zjištění, zda kontejner už daný řetězec obsahuje, využívá k porovnávání objektů metodu `equals()`.²⁵ Ovšem zatímco metoda `equals()` třídy `String` porovnává znakové řetězce, metoda `equals()` třídy `StringBuffer` porovnává adresy instancí v paměti. Třída `StringBuffer` se nám tedy hodila k postupnému vytváření slova znak po znaku, ale nehodí se nám pro ukládání do kontejneru.

4.2 Počítání slov podruhé

Nyní si předchozí úlohu trochu zkomplikujeme: Budeme chtít náš program upravit tak, aby vypisoval nejen jednotlivá slova, ale i počet jejich výskytů.

Tuto úlohu lze řešit mnoha způsoby; jedním z nich je použití asociačních kontejnerů, které obsahují dvojice klíč–hodnota. V Javě jsou k dispozici implementace těchto tříd (`HashMap`, `TreeMap`), které ovšem vyžadují, aby klíč i hodnota byly objektových typů.

My si ukážeme jinou možnost: Použijeme už známou třídu `ArrayList`, do které budeme ukládat dvojice slovo–počet opakování. Pro tyto dvojice si definujeme pomocnou třídu `Dvojice`; její konstruktor vytvoří dvojici, která bude obsahovat počet opakování rovný 1. Náš první pokus by mohl vypadat takto:

```
class Dvojice
{
    private int pocet;           // Počet výskytů
    private String slovo;        // Řetězec (slovo)
    public void pridej(){ pocet++; } // Zvětší počet výskytů
    public Dvojice(String s){ pocet = 1; slovo = s; }
}
```

Metoda
`equals()`

To ovšem nestačí. Stejně jako předtím řetězce budeme dvojice uložené v instanci třídy `ArrayList` vyhledávat pomocí metody `indexOf()`, která, jak víme, využívá metodu `equals()` uložených objektů. Už jsme si řekli, že tuto metodu obsahuje každá třída v Javě. To znamená, že ji bude obsahovat i třída `Dvojice`, a to i v případě, že ji explicitně nedefinujeme. Ovšem tato implicitní implementace porovnává adresu objektů, nikoli jejich obsah – a to nám není mnoho platné. Musíme ji tedy definovat sami.

²⁵ Tuto metodu obsahuje každá třída v Javě, neboť ji dědí od společného předka, třídy `Object`.

V dokumentaci zjistíme, že metoda `equals()` má ve třídě `Object` hlavičku²⁶

```
boolean equals(Object x)
```

což znamená, že vrací logickou hodnotu (typ `boolean`, jehož hodnoty, jak už víme, vyjadřují pravdivost nebo nepravdivost) a jejímž parametrem je instance libovolné třídy v Javě.²⁷ Tuto metodu musíme předefinovat tak, aby se dvě instance třídy `Dvojice` rovnaly, budou-li obsahovat stejný řetězec. Například takto:

```
public boolean equals(Object x)
{
    return slovo.equals( (Dvojice)x ).slovo ;
}
```

Tato metoda prostě vrátí hodnotu, kterou vrátí metoda `equals()` pro porovnávané řetězce. Nejspíš byste očekávali, že výraz v příkazu `return` bude mít tvar

```
slovo.equals(x.slovo)
```

Přetypování

Ovšem problém je, že parametr `x` je typu `Object` a třída `Object` žádnou složku `slovo` neobsahuje, takže překladač takovýto výraz odmítne. Abychom mohli s parametrem `x` zacházet jako s instancí třídy `Dvojice`, musíme ho *přetypovat*, tj. převést na typ `Dvojice`. K tomu poslouží operátor přetypování, který má tvar `(Dvojice)`. (Jméno typu zapsané v závorkách před hodnotou, kterou chceme přetypovat.) Navíc musíme výraz `(Dvojice)x` uzavřít do závorek, neboť jinak by se překladač snažil nejprve použít operátor tečka a teprve pak operátor přetypování.

Změny ve třídě Analyzer

Podívejme se nyní na změny, které musíme udělat ve třídě `Analyzer`. Budou se týkat především přidávání slov do kontejneru. To bude nyní poněkud složitější, a proto tuto operaci naprogramujeme jako samostatnou metodu `pridej()`. (Pozor, jménuje se stejně jako metoda třídy `Dvojice`.)

```
// Přidání nového slova do seznamu nebo zvýšení počtu výskytů
void pridej(String slovo, java.util.ArrayList sezna)
{
    Dvojice d = new Dvojice(slovo); // Vytvoř dvojici
```

²⁶ Tuto hlavičku musíme dodržet, neboť právě tuto metodu používá metoda `add()`. Kdybychom ji pozmenili, prostě bychom přidali přetiženou metodu, ale metodu zděděnou po třídě `Object` bychom tím nepředefinovali a program by nefungoval správně.

²⁷ Přesněji: Formální parametr `x` je typu `Object`. Ovšem tato třída je předkem všech tříd v Javě, a protože v OOP může potomek vždy zastoupit předka, můžeme jako skutečný parametr použít instanci libovoľného typu.

```

int i = sezn.indexOf(d);           // Zkus ji najít v seznamu
if(i<0)                          // Když tam není
{
    sezn.add(d);                  // tak ji přidej
}
else
{
    d = (Dvojice)sezn.get(i);    // si ji vyndej a
    d.pridej();                  // zvětší počet výskytů
}
}

```

V této metodě nejprve vytvoříme dvojici z daného řetězce a čísla 1. (Na číslu nezáleží, protože dvojice se porovnávají jen podle řetězce.) Pak ji zkusíme v najít v kontejneru `seznam`, a pokud tam není (metoda `indexOf()` vrátí `-1`), přidáme ji tam pomocí `add()`.

Jestliže kontejner obsahuje dvojici s tímto řetězcem, vrátí `indexOf()` její index a my ho budeme mít v proměnné `i`. Pomocí metody `get()` získáme odkaz na tuto dvojici a pomocí metody `pridej()` třídy `Dvojice` zvýšíme hodnotu, kterou obsahuje.

Také zde jsme museli použít přetypování. Standardní kontejnery obsahují pouze odkazy na typ `Object`, a ten vrátí i metoda `get()`. Proto musíme vrácený odkaz přetypovat na odkaz na typ `Dvojice`.

Jestliže nyní takto upravený program přeložíme a spustíme obvyklým příkazem

```

java Analyzer < d.txt
vypíše28
[Dvojice@3179c3, Dvojice@310d42, Dvojice@5d87b2]

```

To sice napovídá, že náš kontejner obsahuje tři položky (jak má), ale že je neumí vypsat. Přesněji, náš program neumí vypsat instanci třídy `Dvojice`. O tom se můžeme přesvědčit, zkuseme-li např. v metodě `main()` zadat příkaz

```
System.out.println(new Dvojice("Nazdar"));
```

kterým se vypisuje jediná dvojice. Dostaneme

```
Dvojice@77d134
```

Jak zabezpečit, aby se tiskly řetězce a čísla, které jsou v jednotlivých dvojicích?

Mohli bychom samozřejmě zjistit pomocí metody `size()` počet prvků uložených v instance `slovnik` a pak v cyklu projít všechny prvky a

²⁸ Čísla za znakem @ se mohou na různých počítačích lišit.

vypsat jejich obsah, museli bychom ale změnit přístupová práva ke složkám dvojic. To se ale v OOP nedělá.

Java nabízí elegantnější řešení. Při výpisu instance libovolného objektového typu pomocí metody `System.out.println()` nebo `System.out.print()` se totiž zavolá metoda `toString()` zděděná od třídy `Object`, která danou instanci převede na řetězec. Její implicitní implementace vypisuje jméno typu a jakési jednoznačné číslo. Nic nám ale nebrání tuto metodu ve třídě `Dvojice` předefinovat:

```
public String toString(){
    return slovo + "(" + pocet + ")";
}
```

Zde prostě vrátíme řetězec `slovo` (uložené slovo) a za ním v závorkách řetězcovou reprezentaci hodnoty `pocet`. To je vše.

Úplný výpis třídy `Dvojice` nyní bude

```
class Dvojice {
    private int pocet;
    private String slovo;
    public void pridej(){pocet++;}
    public Dvojice(String s, int i){pocet = i; slovo = s;}
    public Dvojice(String s){pocet = 1; slovo = s;}
    public boolean equals(Object x)
        {return slovo.equals(((Dvojice)x).slovo);}
    public String toString(){return slovo + "(" + pocet + ");"
}
```

Spuštěme-li takto upravený program, vypíše

```
[Ahoj(2), lidi(1), martani(1)]
```

 Zdrojový text tohoto programu najdete na WWW v souboru Kap04\03\Analyzer.java.

 Podíváte-li se nyní do adresáře, ve kterém máte přeložený program, zjistíte, že kromě souboru `Analyzer.class` obsahuje také soubor `Dvojice.class`. Každá třída v přeloženém programu je v samostatném souboru.

4.3 Počítání slov potřetí, tentokrát v JDK 5

5 Java 5 nabízí další možnosti, jak si usnadnit práci. Především jde o tzv. parametrisované třídy, které umožňují zachytit v době překladu některé chyby, které ve starších verzích jazyka zůstaly neodhaleny nebo se projevily až za běhu programu. Druhou novinkou, která se nám zde bude hodit, je knihovní třída `java.util.Scanner`, která nám usnadní formátované čtení.

Zjednodušujeme čtení

Dosud jsme pro formátované čtení používali statické metody nestandardní třídy `MojeIO`. Už ale víme, že v JDK 5 je k dispozici nová třída `java.util.Scanner`, která nabízí podobné služby. Než ji použijeme, řekneme si něco o jejích konstruktorech a metodách. Zatím si ukážeme jen nejjednodušší použití; v kapitole 13 se k ní vrátíme podrobněji.



Konstruktorem této třídy předáme instanci vstupního proudu `System.in`, představujícího standardní vstup (tedy přesměrovatelný vstup z klávesnice).

Metoda `next()` přečte ze vstupu další slovo. Metoda `hasNext()` vrátí hodnotu typu `boolean` určující, zda má smysl dále číst, tj. zda je ve vstupu ještě nějaké další slovo.

Vedle toho obsahuje tato třída metody `nextInt()`, `nextDouble()` apod., které umožňují přečíst následující číslo typu `int`, `double` (reálné číslo) apod. Metody `hasNextInt()`, resp. `hasNextDouble()` a další umožňují zjišťovat, zda ve vstupu následuje slovo představující znakovou reprezentaci celého, resp. reálného čísla apod.

Zbývá vyjasnit si, co to je „slovo“ (v dokumentaci najdeme termín `token`). Třída `java.util.Scanner` tak označuje posloupnost znaků, která neobsahuje bílý znak (mezeru, tabulátor nebo přechod na nový řádek).

Čtení

Podívejme se tedy, jak bychom mohli čerstvě získané znalosti využít při řešení našeho úkolu – spočítat různá slova v textovém souboru.

Metoda `next()` bude vracet jednotlivá slova, nikoli řádek jako celek, a proto odpadne metoda `analyzuj()`, jejímž úkolem byl právě rozklad řádku na slova.

Změní se i cyklus, v němž jsme v metodě `beh()` četli jednotlivé řádky. Nyní využijeme jako podmínu opakování hodnotu vrácenou metodou `hasNext()` třídy `java.util.Scanner`, a přečtená slova budeme hned ukládat.

Naše nová metoda `beh()` by mohla vypadat např. takto:

```
void beh()
{
    java.util.Scanner skan = new java.util.Scanner(System.in);
    String slovo;
    java.util.ArrayList sezn = new java.util.ArrayList();
    while(skan.hasNext())           // Dokud je co číst
    {
        slovo = skan.next();       // přečti nové slovo,
        pridej(slovo, sezn);      // vlož ho do seznamu
    }
}
```

```
        System.out.println(sezn); // a vypiš výsledek
```

Metody `analyzuj()` a `preskocOddelovace()` můžeme ze zdrojového textu odstranit.

Jestliže tento program přeložíme a spustíme, dočkáme se nepříjemného překvapení: Naše slova budou obsahovat tečky na konci vět, čárky oddělující slova ve větách, vykřičníky a otazníky, neboť tato interpunkční znaménka se připojují za slovo bez mezery, a proto je třída `java.util.Scanner` bude chápat jako součásti slov.

Nezbývá tedy, než je z nich odstranit. To svěříme nové metodě `odstraňOddělovače()`.²⁹ Tato metoda dostane jako parametr odkaz na znakový řetězec představující slovo získané pomocí metody `next()` a vrátí odkaz na řetězec, z něhož byly odstraněny oddělovače:

```
String odstraňOddělovače(String slo)
{
    StringBuffer s = new StringBuffer();
    int i = 0;
    while(i < slo.length() &&
          oddelovace.indexOf(slo.charAt(i)) == -1)
    {
        s.append(slo.charAt(i));
        i++;
    }
    return s.toString();
}
```

Zde si nejprve vytvoříme instanci `s` třídy `StringBuffer`. Pak v cyklu procházíme jednotlivé znaky slova, předaného jako parametr. Dokud nenarazíme na konec slova nebo dokud nenarazíme na oddělovač, přidáváme tyto znaky do `s`. Nakonec vrátíme vytvořený řetězec.

Zbývá ještě upravit příkaz v metodě `beh()`, kterým přidáváme přečtená slova do seznamu:

```
pridej(odstraňOddělovače(slovo), sezn);
```

Generické třídy

V našem programu používáme knihovní kontejnery `ArrayList`. Jako všechny kontejnery v Javě obsahuje odkazy na třídu `Object`.

²⁹ Pokud se vám nelibí háčky a čárky v identifikátoru, nemusíte je tam psát. Java to však dovoluje, a abychom si to předvedli, použiji je zde. Ve zbytku knihy se jim vyhýbám, aby byly zdrojové texty snadno čitelné i na jiných platformách než pod MS Windows.

To nám – díky polymorfismu – umožňuje uložit do něj odkaz na instanci jakékoli třídy v našich programech. V tom spočívá nejen síla, ale i velká slabost kontejnerů: Vše je `Object`. Uložíme-li do kontejneru omylem objekt nepatřičného typu, nejenže naši chybu nezjistí překladač, nezjistí ji ani běžící program při ukládání. Na chybu se zpravidla přijde až při vyjímání z kontejneru, kdy se získaný odkaz nepodaří přetypovat na odkaz potřebného typu a vznikne výjimka.

Na chybu se tedy přijde jindy a jinde, než kde vznikla.

Proto nabízí Java 5 tzv. parametrizované (generické) třídy, které umožňují zesílit typovou kontrolu. Používáme je tak, že za jméno třídy připíšeme do lomených závorek jméno typu, který bude sloužit jako typový parametr. U kontejnerů tento typový parametr označuje typ ukládané hodnoty.

V našem programu používáme třídu `java.util.ArrayList` a ta je, stejně jako všechny kontejnery v JDK 5, parametrizována. To znamená, že chceme-li omezit typ ukládaných hodnot na třídu `Dvojice`, napišeme

```
java.util.ArrayList<Dvojice> seznam =  
    new java.util.ArrayList<Dvojice>();
```

Metoda `add()` takto parametrizované třídy očekává jako parametr odkaz na instanci třídy `Dvojice`. Pokusíme-li se zavolat ji s parametrem jiného typu, např.

```
Object obj = new Dvojice("Ahoj", 1);  
seznam.add(obj);
```

ohlásí překladač chybu, i když obj ve skutečnosti obsahuje odkaz na třídu `Dvojice`, neboť deklarovaný typ této proměnné je `Object`. Díky tomu zachytí většinu chyb při ukládání do kontejnerů již překladač.

Také metoda `get()`, jež vrací prvek na zadané pozici v kontejneru, vrátí odkaz na instanci třídy `Dvojice`, nikoli odkaz na `Object` (jak tomu bylo ve starších verzích Javy).

Podobně se chovají i další metody parametrizovaných knihovních tříd.

Použití Nyní již víme dost, abychom mohli ve svém programu použít parametrizovanou třídu `java.util.ArrayList<>`. Změny v programu se dotknou vlastně jen metod `beh()` a `pridej()`. Ukážeme si však celou třídu `Analyzer` včetně změn kvůli použití třídy `java.util.Scanner`.

```
public class Analyzer  
{  
    // seznam znaků, které mohou oddělovat slova v textu  
    static String oddelovace = " .;!?";
```

```

String odstraňOddělovače(String slo)
{                               // Vrací slovo sč bez oddělovačů
    StringBuffer s = new StringBuffer();
    int i = 0;
    while(i < slo.length() &&
          oddelovace.indexOf(slo.charAt(i)) == -1)
    {
        s.append(slo.charAt(i));
        i++;
    }
    return s.toString();
}

// Přidání nového slova do seznamu
void pridej(String slovo, java.util.ArrayList<Dvojice>
sezn)
{
    // nebo zvýšení počtu výskytů
    Dvojice d = new Dvojice(slovo); // Vytvoř dvojici
    int i = sezn.indexOf(d);      // Zkus ji najít v seznamu
    if(i<0)                      // Když tam není
    {
        sezn.add(d);             // tak ji přidej
    }
    else                         // jinak
    {
        d = sezn.get(i);         // si ji vyndej a
        d.pridej();              // zvětší počet výskytů
    }
}

void beh()                     // Vlastní běh programu
{
    java.util.Scanner skan =
        new java.util.Scanner(System.in);
    String slovo;               // Přečtené slovo
    java.util.ArrayList<Dvojice> sezn =
        new java.util.ArrayList<Dvojice>();
    while(skan.hasNext())       // Dokud je co číst
    {
        slovo = skan.next();    // Odstraň okrajové mezery
        pridej(odstraňOddělovače(slovo), sezn);
    }
    System.out.println(sezn);   // Vypiš výsledek
}

public static void main(String[] s)
{
    Analyzer a = new Analyzer();
    a.beh();
}

```

 Zdrojový text tohoto programu najdete na WWW v souboru Kap04\04\Analyzer.java.

4.4 Počítání slov počtvrté

V tomto oddílu budeme řešit týž problém jako v předchozím, tedy napsat program, který přečeť soubor a spočítá výskytu jednotlivých slov. Tentokrát si ale naprogramujeme potřebný kontejner sami. Použijeme ten nejjednodušší, jednosměrně zřetězený seznam. Tento program bude použitelný i ve starších verzích JDK.

Jednosměrně zřetězený seznam

Nejprve si musíme ujasnit, o co jde. *Jednosměrně zřetězený seznam* (nebo prostě jen *seznam*) je datová struktura, která se skládá z prvků, z nichž každý obsahuje vedle užitečných dat ještě odkaz na další prvek. Prvky jsou pomocí těchto odkazů „zřetězeny“, takže známe-li první prvek, můžeme (s pomocí odkazu, který obsahuje) najít druhý prvek, s jeho pomocí pak třetí atd. Poslední prvek bude místo odkazu na následující obsahovat hodnotu `null` (odkaz „nikam“). Schéma seznamu ukazuje obr. 4.1.



Obr. 4.1 Schéma jednosměrně zřetězeného seznamu

Je jasné, že abychom mohli se seznamem pracovat, musíme znát odkaz na jeho první prvek.

Prvek seznamu

Prvky seznamu budou instance třídy `Prvek`, kterou naprogramujeme v tomtéž souboru jako třídu `Analyzer`. Bude obsahovat odkaz `slovo` na uložený řetězec, celé číslo `pocet` obsahující počet výskytů řetězce („počítadlo“) a odkaz `dalsi` na další prvek. Kromě toho bude obsahovat konstruktor, metody pro zjištění a zvětšení počtu, pro zjištění, zda prvek obsahuje daný řetězec a pro zjištění a nastavení odkazu na další prvek. Podívejme se na její výpis.

```
class Prvek
{
    String slovo;           // Uložený řetězec
    int pocet;              // Počítadlo počtu výskytů
    Prvek dalsi;            // Odkaz na další prvek seznamu
    Prvek(){}
    Prvek(String s) { slovo = s; pocet = 1; dalsi = null; }
```

```

int obsahuje(String s) // Obsahuje prvek daný řetězec?
{
    // Vrátí buď počet výskytů nebo 0
    if(s.equals(slovo)) return pocet;
    else return 0;
}
void setDalsi(Prvek q){ dalsi = q; }
Prvek getDalsi(){ return dalsi; }
String getSlovo(){ return slovo; }
void zvetsiPocet(){ pocet++; }
int getPocet(){ return pocet; }
}

```



Pro metody, které zjišťují, resp. nastavují hodnotu datové složky x , se zpravidla používají jména `getX()`, resp. `setX()`.

Implementace seznamu

Třída `Seznam` bude obsahovat jedinou datovou složku, odkaz na první prvek. Nazveme ho `hlava`.

Dále budeme potřebovat metodu `přidej()`, která projde seznam a zjistí, zda některý prvek neobsahuje daný řetězec; pokud ano, zvětší hodnotu `pocet`, jinak připojí nový prvek obsahující dané slovo na konec seznamu. Vedle toho budeme potřebovat ještě metodu `vypis()`, která vypíše jednotlivé prvky. To je vše.

```

class Seznam {
    Prvek hlava = null;           // Ukazatel na první prvek
    void přidej(String slovo)    // Přidá nový prvek na konec
    {
        if(hlava == null)         // Je-li seznam prázdný.
        {
            hlava = new Prvek(slovo); // přidej ho na začátek
        }
        else                      // jinak
        {
            Prvek p = hlava, q = null;
            while(p != null)       // Procházej prvky a
            {
                // hledej, zda některý
                if(p.obsahuje(slovo) > 0) // prvek neobsahuje
                {
                    p.zvetsiPocet(); // dané slovo.
                    return;           // Pokud ano, zvětší počet
                }
                else
                {
                    q = p;             // Pokud ne, přejdi
                    p = p.getDalsi(); // na další.
                }
            } // Konec if
        } // konec while // Najdeš-li konec, připoj
        q.setDalsi(new Prvek(slovo)); // nový prvek za
                                       // poslední.
    }
}

```

```

        }

void vypis()
{
    Prvek p = hlava;
    while(p != null)           // Dokud nenarazíš na konec
    {
        // vypiš data
        System.out.println(p.getSlovo() + " " + p.getPocet());
        p = p.getDalsi();       // a přejdi na další prvek.
    }
}

```

Podívejme se nejprve na metodu `vypis()`. V ní do pomocné proměnné `p` uložíme odkaz na první prvek a pak v cyklu – dokud nenarazíme na konec seznamu – probíráme jednotlivé prvky a vypisujeme z nich data. Příkazem

`p = p.getDalsi();`

pak přejdeme na další prvek. Protože poslední prvek obsahuje místo odkazu na další prvek `null`, metoda po vypsání posledního prvku skončí.

Metoda `pridej()` pro přidání nového prvku je složitější. Nejprve se podívá, zda je seznam prázdný. Pokud ano, tj. pokud pomocná `hlava` obsahuje `null`, vytvoří nový prvek a vloží ho do seznamu, tj. odkaz na tento prvek přiřadí proměnné `hlava`. O vložení dat do nového prvku se postará jeho konstruktor.

Pokud seznam není prázdný, projde jeho jednotlivé prvky. K tomu použije dvě pomocné proměnné, `p` a `q`. První z nich, `p`, vždy obsahuje odkaz na zkoumaný prvek, druhá, `q`, odkaz na předchozí prvek.

U každého prvku se pomocí metody `obsahuje()` zeptá, zda obsahuje řetězec `slovo`. Pokud ano, zavolá metodu `zvetsiPocet()`, tak zvýší hodnotu „počítadla“, a pak skončí.

Jestliže daný prvek řetězec neobsahuje, přejdeme na další prvek. Nejdříve si v `q` uložíme odkaz na aktuální prvek, pak příkazem

`p = p.getDalsi();`

uložíme do `p` odkaz na prvek následující.

Tento cyklus skončí, bude-li `p` obsahovat `null`. Přitom bude `q` obsahovat odkaz na poslední prvek seznamu. Pokud tedy dojdeme až na konec tohoto cyklu, znamená to, že seznam neobsahuje prvek s daným řetězcem, a proto vytvoříme nový prvek a vložíme ho na konec

`q.setDalsi(new Prvek(slovo));`



Třídy Analyzer se tato změna téměř nedotkne. Musíme pouze nahradit deklaraci instance třídy `ArrayList` deklarací instance třídy `Seznam` a změnit příkaz, kterým se řetězec ukládá do seznamu; to už jistě zvládnete sami. Ostatně hotový program najdete na WWW v souboru Kap04\05\Analyzer.java.



Se seznamem lze samozřejmě dělat řadu dalších operací: Lze ho vyprázdnit, lze přidávat prvky na začátek, lze přidávat prvky dovnitř seznamu, lze z něj prvky odebírat atd. Můžete si zkusit některé z těchto operací naprogramovat sami. My jsme v tomto oddílu implementovali pouze ty metody, které jsme potřebovali pro náš příklad.

Seznam můžeme také deklarovat jako generickou třídu. K tomu se však vrátíme později.

5 Začínáme naostro

V předchozích kapitolách jsme si na řadě příkladů ukázali některé možnosti, které Java nabízí. Nyní se pustíme do soustavného výkladu. Nenechte se zmást tím, že o některých věcech jsme se již zmínili v předchozích kapitolách; informace tam nebyly úplné.

Unicode Než se pustíme do podrobností, řekneme si, že Java používá kódování Unicode, ve kterém je každý znak zobrazen ve 2 bajtech. Toto kódování umožňuje používat znaky většiny národních abeced.

Jak budeme Javu popisovat Při popisu syntaxe jazyka Java, tedy pravidel, podle nichž sestavujeme deklarace, příkazy a jiné konstrukce, budeme postupovat takto:

- Před popisem uvedeme na samostatném řádku název popisované konstrukce ukončený dvojtečkou.
- Části, které lze do programu beze změny opsat (tzv. terminální symboly), zapíšeme **tučným písmem**.
- Části, které je třeba dále definovat (nebo které jsme už někde definovali – tzv. neterminální symboly), zapíšeme *kurzivou*.
- Popisované konstrukce mohou mít několik možností. Každá z nich bude začínat odrážkou „*“.

Občas nahradíme přesný syntaktický popis neformálním vysvětlením – v mnoha situacích to bude snáze srozumitelné. Někdy také použijeme zjednodušený popis a doplníme ho výkladem.

Příklad: Jako příklad si ukážeme část popisu komentáře.

komentář:

- */* text_vícerádkového_komentáře */*
- */* text_jednorádkového_komentáře do konce řádku*
- */** text_dokumentačního_komentáře */*

Tento popis ukazuje, že v Javě existují tři druhy komentářů. První z nich začíná znaky */** (v popisu jsou tučně, takže je můžeme přímo zapsat do svého programu). Pak následuje *text_vícerádkového_komentáře*. Tento symbol je zapsán kurzivou, takže budeme ještě muset upřesnit, o co jde: Buď musíme uvést další syntaktický popis, nebo ho nahradit výkladem. Tentokrát zvolíme neformální výklad: *text_vícerádkového_komentáře* je prostě jakýkoli text, který může zabírat i několik řádků. (Další podrobnosti najdete v následujícím oddílu.) ♦

5.1 Základní pojmy

Komentář

Komentář je součást zdrojového textu, kterou bude překladač ignorovat. Slouží programátorovi ke zpřehlednění zdrojového textu. Syntaktický popis jsme si ukázali na předchozí straně, takže už víme, že v Javě máme k dispozici tři druhy komentářů:

- Jednořádkový komentář začínající znaky // a končící přechodem na nový řádek.
- Komentář začínající znaky /* a končící */. Tento komentář může zabírat i několik řádků.
- Dokumentační komentář začínající /** a končící */. Slouží k automatickému generování dokumentace a podrobněji jsme o něm hovořili ve druhé kapitole, v oddílu 2.3.

Komentáře začínající /* nelze do sebe vnořovat. To znamená, že následující řádky překladač označí za chybné:

```
/*          CHYBA - VNORENÉ KOMENTÁŘE
while(i < 10)
{
    n += i++; /* výpočet postupného součtu */
}
```

S podobnými chybami se často setkáváme při ladění programů, kdy potřebujeme některou část zdrojového textu dočasně odstranit. Proto je vhodnější běžné komentáře začínat dvěma lomítky, neboť ty do komentáře začínajícího /* vnořit lze:

```
/*          OK - TO LZE
while(i < 10)
{
    n += i++; // výpočet postupného součtu
}
```

Identifikátor

Identifikátor je jméno, kterým v programu označujeme proměnné, třídy, metody, balíky atd. Identifikátor je tvořen písmeny a číslicemi, začínat musí písmenem. (Za písmena se v Javě pokládá i znak podtržení a znak \$.) Přitom se rozlišují velká a malá písmena. To znamená, že správné identifikátory jsou např. bubu, Bubu, \$bub123. Délka identifikátoru není omezena.

Identifikátor nesmí začinat číslicí a nesmí obsahovat mezeru, tabulátor ani jiný z tzv. bílých znaků. Nesmí také obsahovat tečku, čárku, závorku a jiné speciální znaky. To znamená, že např. `Ahoj lidi` není identifikátor, neboť obsahuje mezeru; také `Jdi-do-haje` není identifikátor, neboť obsahuje znak minus. Na druhé straně `Jdi_do_haje` je správně utvořený identifikátor, neboť obsahuje jen písmena a podtržitka.

Identifikátor se také nesmí shodovat se žádným z tzv. klíčových slov, tj. vyhrazených slov označujících deklarace, příkazy atd. To znamená, že např. `while` není možné použít jako identifikátor. Naproti tomu `While` je správný identifikátor, neboť se od klíčového slova `while` liší velikostí písmen.

Identifikátory mohou obsahovat písmena národních abeced (tedy např. znaky s háčky a čárkami), pokud je zapíšeme v kódu Unicode nebo pokud při překladu převedeme odpovídající kódování; k tomu se ještě vrátíme v oddílu o datových typech.

Obvyklý tvar identifikátorů

Žádné syntaktické pravidlo neurčuje, jak mají vypadat jména tříd, jak mají vypadat jména metod atd.; následující konvence se ovšem všeobecně dodržuje. Doporučuji zvyknout si na ni, neboť vám může výrazně usnadnit orientaci ve vlastních i cizích zdrojových programech, stejně jako v dokumentaci.

- Identifikátory tříd a rozhraní začínají vždy velkým písmenem, ostatní písmena jsou malá. Pokud se identifikátor skládá z více slov, jsou tato slova spojena dohromady a jejich první písmena jsou velká. Příklady jsme viděli v minulé kapitole – identifikátory tříd `Seznam` nebo `ArrayList` se touto konvencí řídí.
- Identifikátory metod a proměnných začínají vždy malým písmenem. Skládají-li se z jediného slova, obsahují jen malá písmena, jinak je – podobně jako u identifikátorů tříd – vždy první písmeno každého dalšího slova velké. Slova jsou opět spojena bez mezer. V dokumentaci a v mnoha knihách (i v naší) se metody odlišují od proměnných tím, že za jejich identifikátor píšeme kulaté závorky.
- Identifikátory balíků obsahují pouze malá písmena. O podrobnostech budeme hovořit později v této kapitole.
- Identifikátory konstant obsahují pouze velká písmena. Skládají-li se z více slov, oddělují se tato slova podtržítkem. Za příklad poslouží `PÍ` nebo `MAXIM_HOUNOTA`.

Klíčová slova

Takto se označují vyhrazená slova, která v Javě označují příkazy, datové typy atd. Klíčové slovo nelze použít jako identifikátor. Všechna klíčová slova jsou zapsána malými písmeny.

Pro úplnost zde uvedeme přehled klíčových slov jazyka Java. S některými z nich se ovšem v této knize nesetkáte, neboť výklad o nich přesahuje naše možnosti.

abstract	assert ⁽⁴⁾	boolean	break	byte
byvalue*	case	cast*	catch	char
c'ass	const*	cont(pue)	default	do
double	else	enum ⁽⁵⁾	extends	false
final	finally	float	for	future*
generic*	goto*	if	implements	import
inner*	instanceof	int	interface	long
native	new	null	operator*	outer*
package	private	protected	public	rest*
return	short	static	super	switch
synchronized	this	throw	throws	transient
true	try	var*	void	volatile
while				

Hvězdičkou jsou označena slova, která jsou sice vyhrazena, takže je nelze použít jako identifikátory, nejsou ale v současné verzi Javy k ničemu využita. Klíčové slovo assert je součástí Javy počínaje JDK 1.4 a klíčové slovo enum počínaje JDK 5.

Zápis programu

Pravidla pro zápis programu v Javě lze shrnout do následujících bodů:

- Identifikátory a klíčová slova nesmíme rozdělit bílým znakem (mczerou, přechodem na nový řádek, tabulátorem, komentářem). Nesmíme tedy napsat např. `ma in()`.
- Také některé operátory, které se skládají z několika znaků, nelze rozdělit. To se týká např. operátorů `++` (nesmíme napsat `++` s mezou mezi znaky `+`), `--`, `*=`, `==`, `<=`, `&&` a dalších.
- Je-li třeba zapsat v programu bezprostředně vedle sebe dva identifikátory, identifikátor a klíčové slovo nebo dvě klíčová slova, musíme je oddělit alespoň jedním bílým znakem.
- Na místě, kde může být jedna mezera, může být libovolný počet bílých znaků.

5.2 Organizace zdrojového kódu: balíky

V rozsáhlých programech občas zjistíme, že bychom chtěli pojmenovat dvě různé třídy stejně, nebo že bychom potřebovali pojmenovat svou třídu stejně jako některou knihovní třídu. Může se také stát, že chceme použít v jednom programu několik knihoven vytvořených různými lidmi, ale naneštěstí se třídy v těchto knihovnách se jmenují stejně. Co s tím?

Java používá řešení zvané *balíky*³⁰ (package). *Balík* je skupina souborů, které tvoří logický celek – typicky jednu knihovnu nebo jeden modul programu. Jeden balík je obvykle uložen v jednom adresáři; takový adresář může obsahovat podadresáře, které představují „podbalíky“. Může být také komprimován do jednoho archivu. Pak odpovídá balíku archiv – komprimovaný soubor – .jar nebo zip.

Balíky také hrají svou roli při specifikaci tzv. přístupových práv, tj. při určování, kdo smí které třídy, případně které složky používat. O tom budeme hovořit později.

Jména balíků

Jméno balíku se skládá z cesty k souborům se třídami, které v něm jsou; jména adresářů a podadresářů ovšem spojujeme tečkami, nikoli znaky používanými v běžných operačních systémech (lomítky či obrácenými lomítky).

V předchozích kapitolách jsme se už setkali balíkem `java.util`, který obsahoval mj. třídu `java.util.ArrayList`. Toto jméno napovídá, že soubor `ArrayList.class` je umístěn v podadresáři `java.util` nebo v odpovídajícím archivu. V mé instalaci JDK 5 je uložen v souboru `jdk1.5.0\jre\lib\rt.jar` v podadresáři `java.util`. Podobná byla i cesta k němu v předchozích verzích JDK. Jméno tohoto souboru a cesta k němu by proto měla být uvedena v systémové proměnné CLASSPATH, např.:

```
SFT CLASSPATH=.:C:\jdk1.3\jre\lib\rt.jar
```

Celosvětová konvence pojmenování balíků

Java je určena mj. k programování pro internet, a proto je potřeba, aby pojmenování balíků byla pokud možno celosvětově jednoznačná. Specifikace jazyka Java doporučuje používat označení, která vycházejí ze jmen internetových domén. Takto utvořené jméno balíku začíná „adresou pozpátku“, přičemž jednotlivé složky jsou odděleny tečkami.

C⁺

³⁰ Znáte-li C++, můžete se na ně divat jako na analogii prostorů jmen.

To znamená, že balík `seznam`, který je umístěn na adrese (v doméně) `alfabetagama.cz`, bude mít jméno `cz.alfabetagama.seznam`³¹.

Ovšem dodržování této konvence má smysl pouze v případě, že své třídy chcete dát k dispozici na WWW.

Deklarace a používání balíku

Příkaz package Chceme-li třídy v nějakém souboru zařadit do balíku, uvedeme jako první v daném souboru příkaz `package`. Jeho syntax je

příkaz package:

- `package jméno_balíku ;`

Příklad: Jako příklad vezmeme náš první program, který ale umístíme do balíku `pozdrav`. (Jeho zdrojový text najdete na WWW v souboru `Kap05\01\Ahoj.java`.)



```
/* Soubor Kap05\01\pozdrav\Ahoj.java
 * analogie prvního programu, ale v balíku pozdrav
 */
package pozdrav;

public class Ahoj {
    void text()
    {
        System.out.println(
            "To už tady bylo ... a už je to tu zas.");
    }

    public static void main(String[] s)
    {
        Ahoj a = new Ahoj();
        a.text();
    }
}
```

Po překladu umístíme soubor `Ahoj.class` do podadresáře `pozdrav` aktuálního adresáře. Program pak spustíme z aktuálního adresáře příkazem

```
java pozdrav.Ahoj
```

ve kterém uvedeme jméno třídy, před něž připojíme tečkou jméno balíku. Zdůrazněme, že přitom musí být soubor obsahující třídu `Ahoj`

³¹ V JDK 1.1 se pro první složku jména vyhovujícího této konvenci používala velká písmena, tedy např. `CZ.alfabetagama.seznam`. Od verze Java 2 se však používají malá písmena.

v podadresáři `pozdrav` některého z adresářů, na které ukazuje proměnná `CLASSPATH` – například aktuálního adresáře. ♦

Kvalifikace jménem balíku

Na třídy, které jsou součástí balíku, se odvoláváme konstrukcí *jméno_balíku.jméno_třídy*. (Jméno třídy „kvalifikujeme“ jménem balíku.) To znamená, že na třídu `Ahoj` bychom se měli odvolávat zápisem `pozdrav.Ahoj`. Podobně jsme se v minulé kapitole odvolávali na třídu `ArrayList` z balíku `java.util` zápisem `java.util.ArrayList`.

Ovšem ve třídách uvnitř balíku můžeme kvalifikaci vyněchat. Proto jsme v našem příkladu mohli napsat jen

```
Ahoj a = new Ahoj();
```

Příkaz import

Jména tříd s kvalifikací jménem balíku mohou být nepříjemně dlouhá. Proto máme v Javě příkaz `import`, který umožní jméno balíku vyněchat. Má tvar

příkaz import:

- `import jméno_balíku.jméno_třídy;`
- `import jméno_balíku.*;`

Za tímto příkazem už můžeme kvalifikaci jménem balíku vyněchat. Chceme-li takto zpřístupnit všechna jména v balíku, uvedeme místo jména třídy hvězdičku.

V minulé kapitole jsme při počítání slov používali třídu `java.util.ArrayList`. Použijeme-li na počátku souboru příkaz

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

budeme moci ve zbytku programu místo `java.util.ArrayList` psát pouze `ArrayList`. Zkrácený výpis upraveného programu `Kap04\03\Analyzer.java` bude vypadat takto:

```
/* Soubor Kap05\02\Analyzer.java
   Vznikl úpravou souboru Kap04\03\Analyzer.java
*/
import java.util.ArrayList;

class Dvojice {
    // Stejném jako v minulé kapitole
}

// Hlavní třída programu
public class Analyzer {
    static String oddelovace = " .;!?"';

    int preskočOddelovace(String rad, int od) { /*...*/ }
    void pridaj(String slovo, ArrayList seznam) { /*...*/ }
```

```
void analyzuj(String řádek, ArrayList seznam) { /*...*/ }
void beh() throws Exception
{
    ArrayList slovnik = new ArrayList();
    // a dále beze změny
}

public static void main(String[] s) throws Exception
{ /*...*/
} ◇
```

Balík java.lang

V minulých kapitolách jsme se také setkali s **třídami**, které jakoby nepatřily do žádného balíku (např. String, StringBuffer). Tyto třídy jsou však ve standardním balíku java.lang a překladač se chová tak, jak kdyby na počátku každého souboru stál příkaz

```
import java.lang.*;
```

Pozor: Import celého balíku neznamená také import všech jeho podbalíků! Jestliže si příkazem

```
import java.awt.*;
```

vyžádáme import všech tříd z balíku java.awt, neznamená to že budeme moci používat bez kvalifikace i třídy z balíku java.awt.event. Pro každý podbalík musíme použít samostatný příkaz import.

Když vynecháme příkaz package

Jestliže v nějakém souboru příkaz package neuvedeme, umístí překladač třídy z tohoto souboru do jakéhosi „implicitního balíku“. Do něj ovšem uloží všechny třídy ze všech souborů, v nichž není balík specifikován.

V malých programech, jako byly příklady v minulé kapitole, není třeba balík definovat. Ve skutečných aplikacích je to však zpravidla nezbytné.

Statický import

5

Java 5 zavedla ještě tzv. statický import. Jde o příkaz, který umožňuje vynechávat při používání statických složek kvalifikaci jménem třídy. Jeho tvar je

statický import:

- **import static jméno_třídy.jméno_složky;**
- **import static jméno_třídy.*;**

Jméno_třídy může obsahovat i jméno balíku. První možnost znamená import jediné statické složky, druhá možnost znamená import všech statických složek dané třídy.

Například běžné matematické funkce, jako sinus nebo kosinus, jsou v Javě deklarovány jako statické metody třídy `Math`. Chceme-li např. vypočítat sinus hodnotu uložené v proměnné `x` a uložit ho do proměnné `y`, napíšeme

```
y = Math.sin(x);
```

To není příliš přehledné, nejspíš se to dost ličí od vzorečků, které do programu opisujeme, aproto můžeme v Javě 5 použít statický import:

```
import static Math.*;
```

pak můžeme použít obvyklejší zápis

```
y = sin(x);
```

Ve velké většině případů však statický import program spíše zatemní než vyjasní, a proto doporučuji používat ho co nejméně.

Poznamenejme, že statický import – alespoň v současné verzi JDK – nefunguje pro třídy, které nejsou v žádném balíku (tedy jsou v implicitním balíku).

6 Datové typy, proměnné

Každá proměnná v Javě musí mít nějaký datový typ. Java rozlišuje dvě základní skupiny typů: Typy vyjadřující čísla, logické hodnoty a znaky patří mezi tzv. *primitivní datové typy*, ostatní shrneme pod společné označení *odkazy* (reference). Reference mohou odkazovat na pole, objekty a tzv. rozhraní, o nichž budeme mluvit v 11 kapitole.

6.1 Primitivní datové typy

Do této skupiny patří celočíselné typy, typy reálných čísel, typ `char` pro práci se znaky a typ `boolean` pro práci s logickými hodnotami. Pokud bychom potřebovali na místě primitivního typu objektový typ, můžeme použít některé z tzv. obalových tříd. To jsou třídy, jejichž instance mohou zapouzdřit hodnoty odpovídajících primitivních typů. V Javě lze hodnoty primitivních typů automaticky konvertovat na hodnoty obalových typů a naopak – o tom budeme hovořit v oddílu *Balení a vybalování* na konci této podkapitoly.

Celá čísla

V úvodních kapitolách jsme se seznámili s typem `int`. Java ovšem nabízí celkem 4 celočíselné typy (viz tabulka 6.1).

Typ	B	Rozsah
<code>byte</code>	1	-128 .. 127
<code>short</code>	2	-32 768 .. 32 767
<code>int</code>	4	-2 147 438 648 .. 2 147 438 647
<code>long</code>	8	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807

Tabulka 6.1 Celočíselné typy v Javě. Ve sloupci **B** je uveden počet bajtů, které proměnná tohoto typu zabírá v paměti

Tato tabulka říká, že např. proměnná typu `byte` bude v libovolné implementaci zabírat 1 bajt a lze do ní uložit jakékoli celé číslo v rozmezí od -128 do 127 (včetně těchto krajních hodnot).

Poznámky ■ Na rozdíl od mnoha jiných programovacích jazyků Java jednoznačně specifikuje počet bajtů, rozsah hodnot a způsob zobrazení a uložení celočíselných hodnot v paměti. To je nezbytné, aby bylo možno přenášet bajtový kód mezi různými počítači (a mezi různými implementacemi Javy).

C⁺ ■ Uvedené názvy typů jsou jediné možné. Nelze napsat např. `snort` jako v C/C++.
■ Java neobsahuje celočíselné typy bez znaménka, jako je `unsigned` v C/C++.

Literály Ze třetí kapitoly už víme, že celočíselnou konstantu (literál) můžeme zapsat podobně jako v běžném životě, nesmíme však použít žádné oddělovače tisíců. Můžeme tedy napsat `54321`, nikoli však `54 321`.

Vedle toho můžeme celočíselnou konstantu zapsat v šestnáctkové (hexadecimální) soustavě; přitom se jako číslice s významem `10, 11, ..., 15` používají písmena `a, b, ..., f` nebo `A, B, ..., F`. Konstanta v šestnáctkové soustavě začíná `0x` nebo `0X` (nula a malé nebo velké X). Například šestnáctková konstanta `0xF` znamená v desítkové soustavě 15, `0x1B` znamená 27, `0xFFFF` znamená 65535.

Třetí možností je použít k zápisu celočíselného literálu osmičkovou soustavu. Takováto konstanta smí obsahovat jen číslice `0 .. 7` a její zápis musí začínat nulou; např. `05` je osmičková konstanta znamenající v desítkové soustavě 5, `010` je osmičková konstanta znamenající 8, `0321` znamená 209.

Všechny celočíselné literály - v programu přímo zapsané konstanty - jsou typu `int`. Pokud chceme konstantu typu `long`, musíme k ní připojit koncovku `l` nebo `L`.

Přiřazování Java dovoluje přiřadit proměnné typu s větším rozsahem hodnotu typu s menším rozsahem. To znamená, že např. proměnné typu `int` můžeme přiřadit hodnotu typu `short`. Opačné přiřazení se považuje za chybné, neboť při něm může dojít ke ztrátě hodnoty. Platí-li deklarace

```
int i = 15;  
short s = 25;  
  
smíme napsat  
i = s;           // OK  
  
neboť všechny hodnoty typu short leží v rozsahu typu int. Opačné  
přiřazení,  
s = i;           // CHYBA
```

překladač označí za chybné, neboť při něm může dojít ke ztrátě informace – některé hodnoty typu `int` nelze do proměnné typu `short` uložit. Pokud něco takového potřebujeme, musíme použít přetypování:

```
s = (short)i; // OK
```

Leží-li přiřazovaná hodnota v rozsahu typu proměnné na levé straně, bude vše v pořádku. Pokud ne, výsledek nejspíš nebude mít smysl, neboť se přenese jen část bitů tvořících reprezentaci čísla v paměti.

Operace s celými čísly

Základní operace s celými čísly už známc. Operátory `+`, `-` a `*` slouží ke sčítání, odečítání a násobení. Pro dělení používáme operátor `/`. (To znamená, že dělence i dělíteli zapisujeme do stejné řádky, např. `a/b`. Chceme-li dělit složitější výrazy, uzavřeme je do závorek, například `(a+c)/(b-d)`.) Dělíme-li dvě celá čísla, je výsledkem opět celé číslo, tedy např. `13 / 5` je `2`.

Pomocí operátoru `%` zjistíme zbytek po dělení. Např. `13 % 5` jsou `3`.

Dále můžeme použít unární operátory plus a minus. Je-li `a` celé číslo, znamená `+a` stejnou hodnotu jako `a` a `-a` znamená číslo se stejnou absolutní hodnotou a s opačným znaménkem.

Celá čísla můžeme porovnávat pomocí operátorů `>`, `<`, `>=` a `<=`, které znamenají větší, menší, větší nebo rovno a menší nebo rovno. Zda jsou si dvě číslo rovna, resp. zda si nejsou rovna, zjišťujeme pomocí operátorů `==`, resp. `!=`. Tyto tzv. relační operátory vytvářejí hodnotu typu `boolean` (logickou hodnotu, tj. pravda nebo nepravda).

Na celočíselné proměnné (obecně na celočíselné l-hodnoty, tj. na jakékoli výrazy, které mohou stát na levé straně přiřazení) můžeme také použít operátory `++` a `--`. První z nich zvětší hodnotu proměnné o 1, druhý ji zmenší o 1. Podrobněji jsme o nich hovořili v kapitole 3.4.

Se všemi výše uvedenými operátory jsme se už setkali v předechozích kapitolách. Vedle nich můžeme pro celá čísla použít tzv. bitové operátory `&`, `|`, `^`, `<<`, `>>` a `>>>`. Jejich stručnou charakteristiku najdete v tabulce 7.1, výklad o nich ale přesahuje rámec naší knihy.

Příklad: převod do jiné soustavy

Napíšeme třídu `Konvertor`, jejíž metoda `konverze()` převede číslo typu `long` na znakový řetězec, který bude vyjádřením tohoto čísla v zadáné číselné soustavě. Základem této soustavy bude číslo v rozmezí od 2 do 36. (Za chvíli uvidíme proč.)

Nejprve si ujasněme, co budeme dělat. My jsme zvyklí na desítkovou soustavu (soustavu se základem 10). Např. zápis `123` znamená, že jde o číslo, které obsahuje jednu stovku, 2 desítky a 3 jednotky, tedy

$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 123$. Podobně se vyjadřují i čísla v jiných číselných soustavách. Například zápis čísla 1011 ve dvojkové soustavě znamená $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, tedy 11 v desítkové soustavě.

Z toho plyne, že hodnotu poslední číslice zjistíme, když vezmeme zbytek po dělení daného čísla základem číselné soustavy; např. $123 \% 10$ je rovno 3.

Při zjišťování předposlední číslice můžeme postupovat podobně; nejdříve se ale musíme zbavit poslední číslice. Toho dosáhneme např. vydelením základem. (Abychom zjistili předposlední číslici čísla 123, vezmeme číslo $123 / 10 = 12$ a zjistíme jeho poslední číslici.)

Ted' už můžeme navrhnout postup, který postupně zjistí jednotlivé číslice vyjadřující dané číslo v soustavě o základu z (ovšem v obráceném pořadí):

1. Nejprve zjistíme, zda je dané číslo n záporné. Pokud ano, zapamatujeme si to a dále budeme pracovat s číslem $-n$.
2. Výrazem $n \% z$ zjistíme hodnotu poslední číslice; tu si uložíme.
3. Číslo n nahradíme číslem n / z (celočíselné dělení základem).
4. Je-li $n != 0$, vrátíme se na krok 2.
5. Připojíme znaménko.

Třída Konvertor bude obsahovat datovou složku `zaklad`, v níž bude uložen základ číselné soustavy, do které budeme dané číslo převádět. Dále bude mít statickou (tj. pro všechny instance společnou) složku

```
static String cislice =
    "0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ";
```

obsahující všechny znaky, které použijeme jako číslice. (Je jich celkem 36, proto nejvyšší základ, který dovolíme, bude 36. V této soustavě bude znak Z představovat číslici s hodnotou 35.) Vlastní metoda `konverze()` bude jednoduchá:

```
public String konverze(long n)
{
    StringBuffer s = new StringBuffer(""); // Zde ho vytvoříme
    boolean zaporne = false; // Je záporné?
    if(n==0)
    {
        // Je-li to nula, vrát řetězec "0"
        return "0";
    }
    if (n < 0) // Je-li n záporné, zapamatuj si to
    {
        // a dále pracuj s -n
        zaporne = true;
        n = -n;
    }
    while(n != 0){ // Dokud nezpracuješ všechny číslice
        s.append(cislice.charAt((int)(n % zaklad)));
        n /= zaklad;
    }
}
```

```
|  
| if(zaporne)s.append('-');  
| return new String(s.reverse());  
|
```

Je-li `n` rovno nule, vrátíme řetězec "0" a skončíme. Jinak otestujeme, zda je `n < 0`, a pokud ano, zapamatujeme si to (uložíme do `zaporne` hodnotu `true`) a místo `n` budeme dále pracovat s `-n`. Do následujícího cyklu tedy přijdeme s hodnotou `n > 0`.

Znakovou reprezentaci vytváříme číslici po číslici v pomocné proměnné s typu `StringBuffer`. Hodnotu jednotlivých číslic zjišťujeme výrazem `n % zaklad`, potřebný znak výrazem

```
cislice.charAt((int)(n % zaklad)).
```

(Protože `n` je typu `long` a metoda `charAt()` očekává parametr typu `int`, musíme výsledek operace `n % zaklad` přetypovat.) Pak nahradíme `n` číslem „bez poslední číslice“, tj. podílem `n / zaklad`.

Cyklus skončí, jakmile zjistíme všechny číslice, tj. jakmile bude podíl `n / zaklad` roven 0. Nakonec připojíme případné znaménko minus.

Takto získáme číslice v opačném pořadí, než potřebujeme. K obrácení řetězce použijeme metodu `reverse()` třídy `StringBuffer`. Poslední příkaz vlastně znamená „Vytvoř novou instanci třídy `String` z instance `s`, ve které otočíš pořadí znaků, a tu vrat.“

Celou třídu Konvertor najdete (i s metodou `main()`), která obsahuje jednoduché testy funkčnosti) v souboru Kap06\01\Konvertor.java. ♦

Přetečení

Při aritmetických operacích se může stát, že výsledek bude ležet mimo rozsah daného typu. Tomu se říká *přetečení* a výsledek pak většinou nemá smysl, podobně jako při přetypování na číslo s menším rozsahem.

Příklad:

 Ve 3. kapitole jsme napsali program, který přečte z konzoly celé číslo `n` a vypíše jeho faktoriál (součin $1 \cdot 2 \cdot \dots \cdot n$). Na WWW v adresáři Kap06\02 najdete soubor Fakt.java, který obsahuje týž program, jen upravený tak, že čísla z konzoly čte a vypisuje jejich faktoriály v cyklu až do chvíle, kdy zadáme záporné číslo. Budeme-li postupně zadávat čísla 12, 13, 14, dostaneme

```
Zadej cele cislo: 12  
Jeho faktorial je 479001600  
Zadej cele cislo: 13  
Jeho faktorial je 1932053504  
Zadej cele cislo: 14  
Jeho faktorial je 1278945280
```

Faktoriál 14 nám vyšel menší než faktoriál 13, i když by měl být jeho čtrnáctinásobkem. Ovšem po troše počítání se přesvědčíme, že ani faktoriál 13 není třináctinásobkem faktoriálu 12, takže ani tento výsledek už není dobře. Důvodem je, že faktoriál 13 (a žádného většího celého čísla) se už nevejde do rozsahu typu int.

Zkusme tedy změnit definici metody faktorial() tak, aby vracela výsledek typu long:



```
// Soubor Kap06\02\Fakt.java
public static long faktorial(long n)
{
    long s = 1;      // Nutno změnit i typ proměnné s, která
    if(n < 0)        // bude obsahovat výsledek
    {
        System.out.println("Není definovan");
        System.exit(0);
    } else {
        while(n > 1)// Cyklus, ve kterém se faktoriál vypočte
        {
            s *= n--;
        }
    }
    return s;
}
```

Kromě návratového typu je nezbytné změnit i typ proměnné s, ve které se počítá výsledek. Typ parametru n jsme změnili především proto, abychom mohli metodu faktorial() přetížit – aby mohla třída Fakt obsahovat vedle sebe dvě metody se stejným jménem. (Připomeňme si, že typ návratové hodnoty k rozlišení přetížených metod nestačí.)

I zde se ale poměrně brzy dočkáme zklamání, neboť již hodnota 21! se nevejde ani do rozsahu typu long. Upravený program vypíše

```
Zadej cele cislo: 20
Jeho faktorial je 2432902008176640000
Zadej cele cislo: 21
Jeho faktorial je -4249290049419214848
```

Všimněte si, že celočíselné přetečení se nehlásí jako běhová chyba; musíme si být vědomi možnosti, že může nastat. ♦

Obalové třídy

Obalové třídy pro celočíselné typy se jmenují Byte, Short, Integer a Long. Novou instanci třídy Integer vytvoříme např. příkazem

```
Integer i = new Integer(98);
```

Konstruktory třídy Integer (a podobně obalových tříd ostatních celočíselných typů) mohou mít jako parametr odpovídající celé číslo nebo znakový řetězec, který takové číslo představuje.

Hodnotu uloženou v instanci nemůžeme měnit; můžeme ji ale zjistit pomocí metod `byteValue()`, `intValue()` atd.

Obalové třídy obsahují mimo jiné konstanty `MIN_VALUE` a `MAX_VALUE`, které představují minimální a maximální hodnoty jednotlivých typů. (Jde o statické složky, takže je můžeme používat i v případě, že neexistuje žádná instance dané třídy.) Chceme-li např. zjistit, zda se hodnota `x` typu `int` vejde do proměnné `b` typu `byte`, můžeme napsat

```
if((x > Byte.MIN_VALUE) && (x < Byte.MAX_VALUE)) {  
    b = x;  
} else i chyba();
```

Pro porovnávání instancí obalových tříd navzájem lze použít metodu `compareTo()`. Ze statických metod těchto tříd stojí za zmínku `parseByte()`, `parseInt()` a další, které umožňují převést znakový řetězec představující číslo na číselnou hodnotu primitivního typu.

Znaky

Pro práci s jednotlivými znaky nabízí Java typ `char`. Proměnná typu `char` zabírá dva bajty, neboť Java používá šestnáctibitové kódování Unicode. Znakové konstanty můžeme v programu zapsat několika způsoby:

- Znakem uzavřeným mezi apostrofy, např. `'a'`, `'ž'`.
- Řídicí znaky, jako je přechod na nový řádek, tabulátor ap., můžeme vyjádřit některou z řídicích posloupností (escape sequence) uvedených v tabulce 6.2 uzavřených mezi apostrofy, např. `'\n'`.

Znak	UNICODE	Význam
\n	\u000A	Přechod na nový řádek
\r	\u000D	Návrat na počátek řádku
\t	\u0009	Tabulátor
\b	\u0008	Návrat o jeden znak
\\"	\u005C	Obrácené lomítko, znak „\“
\'	\u002C	Apostrof
\"	\u0022	Uvozovky

Tab. 6.2 Řídicí posloupnosti v Javě

- Pomocí univerzálního jména znaku, tj. zápisem `'\uXXXX'`, kde `XXXX` jsou šestnáctkové číslice vyjadřující kód znaku v kódování

Unicode. (Musíme použít malé u a čtyři číslice.) Tento zápis se používá především pro znaky s diakritickými znaménky; v tabulce 6.3 najdete kódy těchto znaků z české a slovenské abecedy.

- Zápisem '\ooo', kde ooo jsou tři osmičkové číslice.

á	\u00E1	ó	\u00F3	Á	\u00C1	Ó	\u00D3
ä	\u00E4	ö	\u00F4	Ä	\u00C4	Ö	\u00D4
č	\u010D	ŕ	\u0155	Č	\u010C	Ŕ	\u0154
ď	\u010F	ř	\u0159	Ď	\u010E	Ř	\u0158
é	\u00E9	š	\u0161	É	\u00C9	Š	\u0160
ě	\u011B	ť	\u0165	Ě	\u011A	Ť	\u0164
í	\u00ED	ú	\u00FA	Í	\u00CD	Ú	\u00DA
ŕ	\u013E	ü	\u016F	Ľ	\u0130	Ü	\u016E
í	\u013A	ý	\u00FD	Ĺ	\u0139	Ý	\u00DD
ň	\u0148	ž	\u017F	Ň	\u0147	Ž	\u017D

Tab. 6.3 Univerzální jména českých a slovenských znaků s diakritickými znaménky

Řetězcové konstanty Všechny uvedené způsoby vyjádření znaků můžeme použít i v zápisu řetězcové konstanty. Například příkaz

```
System.out.print("ah\u00f3j\u0103j\nlidi");
```

vypíše

```
ahoj
lidi
```

Identifikátory Univerzální jména znaků lze použít i v identifikátorech. To znamená, že např. po\u010Det je správný zápis identifikátoru počet. Pokud chcete použít zápis obsahující písmeno s diakritickým znaménkem, např. již zmíněný počet, máte dvě možnosti:

- Napsat zdrojový text v kódování češtiny, které vaše instalace JDK implicitně používá; např. pod Windows to je kódová stránka 1250.
- Uvést v příkazové řádce, kterou spouštíte překladač javac.exe, přepínač -encoding, za který zapíšete způsob kódování. To může být např. Cp1250 (čeština pod Windows), Cp852 (kódování Latin 2), ISO8859_2 (čeština pro UNIX), nebo unicode. Příkaz, kterým spustíme překlad, může vypadat např. takto:

```
javac -encoding Cp1250 Znaky.java
```

Znaky jsou čísla

Typ `char` se chová jako číselný typ bez znaménka s hodnotou odpovídající kódu daného znaku. To znamená, že znaky – konstanty i proměnné – můžeme používat ve výrazech spolu s celými čísly. Můžeme na ně používat tytéž operátory jako na celá čísla.



Na WWW v souboru `Kap06\03\Znaky.java` najdete program, který vypíše univerzální jména znaků z tabulky 6.3.

Obalová třída

Obalová třída pro znaky se jmenuje `Character`. Podobně jako v případě obalových tříd pro celá čísla neumožňuje změnit uložený znak.

Reálná čísla

Zámito označením se v Javě skrývají „desetinná čísla“, tedy „čísla s plovoucí čárkou“. Java nabízí dva datové typy pro reprezentaci reálných čísel, a to `double` a `float`. Jejich vlastnosti shrnuje tabulka 6.2.

Typ	B	Rozsah	Platných cifer
<code>float</code>	4	$1,4 \times 10^{-45} \dots 3,4 \times 10^{38}$	7–8
<code>double</code>	8	$4,9 \times 10^{-324} \dots 1,7 \times 10^{308}$	15–16

Tab. 6.4 Datové typy vyjadřující reálná čísla v Javě. Ve sloupci **B** je uveden počet bajtů, které proměnná tohoto typu zabírá v paměti

Uvedený rozsah znamená, že např. v proměnné typu `float` můžeme uložit číslo x , pro které platí $1,4 \times 10^{-45} \leq x \leq 3,4 \times 10^{38}$ nebo je rovno nule nebo leží v rozmezí $3,4 \times 10^{-38} \leq x \leq 1,4 \times 10^{-45}$. Vedle toho mohou obsahovat „strojové nekonečno“ (hodnotu, která se chová jako $\pm\infty$) a `NaN`, hodnotu, která není číslem a která vznikne při některých výpočetních chybách.

Literály

Reálné konstanty (tedy desetinná čísla) zapisujeme v programu několika způsoby.

- Podobně jako v běžném životě, pouze desetinnou čárku nahradíme desetinnou tečkou a neděláme mezery mezi trojicemi číslic, např. číslo 1 234,654 321 zapíšeme 1234.654321; tomu říkáme „zápis s pevnou řádovou tečkou“.

- V tzv. semilogaritmickém³² tvaru. To je zápis odvozený od způsobu, jakým běžně zapisujeme velmi velká nebo velmi malá čísla, např. $1,4 \times 10^{-45}$. Pouze v něm desetinnou čárku nahradíme desetinnou tečkou, vypustíme symbol násobení, desítku zaměníme znakem e nebo E a exponent zapíšeme do stejné řádky jako zbytek čísla. Číslo $1,4 \times 10^{-45}$ tedy v Javě zapíšeme $1.4e-45$ nebo $1.4E-45$. Poznamenejme, že číslo před symbolem e nebo E nemusí obsahovat desetinnou část; můžeme také napsat např. $14E-46$.

Operace s reálnými čísla

Unární operátory +, -, ++ a -- a binární operátory +, - a * mají pro reálná čísla podobný význam jako pro celá čísla. Operátor / slouží k dělení v obvyklém smyslu. Operátor % vypočte opět zbytek po dělení. Při přiřazování můžeme kromě operátoru = použít také složené operátory +=, *= a další.

Pro porovnávání reálných čísel máme k dispozici operátory <, <=, >, >=, != a ==, podobně jako pro porovnávání celých čísel. Bitové operátory nelze pro reálná čísla použít.

Ve výrazech se mohou vedle sebe objevit celá a reálná čísla. Výsledky pak budou reálné; o tom si podrobněji povíme v následující kapitole.

Zaokrouhlovací chyby

Reálná čísla jsou v paměti uložena ve 4, resp. v 8 bajtech. To znamená, že nemohou obsahovat čísla zadána s libovolnou přesností, ale jen s několika platnými číslicemi. Jinak řečeno, v počítači nelze zobrazit jakákoli čísla, ale jen některá. Vyjde-li jako výsledek číslo, které nelze zobrazit, zaokrouhlí počítač výsledek na nejbližší zobrazitelnou hodnotu.

Obvykle s vystačíme s přesností, kterou nám typy float a double poskytuji. Nicméně jsou situace, kdy chyby vzniklé zaokrouhlováním mohou zcela pokazit výpočet.

Podívejme se na jednoduchý příklad. Následující úsek programu by měl vypsat čísla 0.1, 0.2, ..., 0.9. Ve skutečnosti ale představuje nekonečný cyklus:

```
double d = 0.0;  
while(d != 1.0)  
{  
    System.out.println(d);  
    d += 0.1;  
}
```

³² V návodech ke kalkulačkám se mu říká „vědecká notace“. Přiznám se, že se mi nepodařilo zjistit, co je na tomto způsobu zápisu vědeckého; pokud vím, učí se v 8. třídě základních škol.

Problém je v tom, že číslo 0,1 (jedna desetina v desítkové soustavě) má ve dvojkové soustavě tvar

0.000110011001100...

tj. je periodické – k jeho přesnému vyjádření bychom potřebovali nekončně mnoho bitů. Počítac mu ale může vyhradit jen 64 bitů, a proto si ho musí zaokrouhlit. Ovšem součet deseti takovýchto zaokrouhlených čísel nedá přesně 1.

Na druhé straně reálné číslo 1,0 lze v počítači vyjádřit přesně, a proto v podmínce příkazu `while` nenastane nikdy rovnost. Tento příklad ukazuje, že porovnávání reálných čísel pomocí operátorů `==` a `!=` může vést k chybám, a proto bychom se mu měli vyhýbat.



Ukázkový program obsahující právě uvedený cyklus najdete na WWW v souboru Kap06\04\Realna.java. Po spuštění ho budete muset zastavit klávesovou kombinací Ctrl+C. ♦

Konstanty

a funkce:

třída Math

Téměř všechny programovací jazyky poskytují také nástroje pro matematické výpočty; v Javě jsou soustředěny ve třídě `Math`.

Tato třída obsahuje pouze statické složky, takže není třeba vytvářet její instance. Jako datové složky v ní najdeme konstanty `e` (Eulerovo číslo e, základ přirozených logaritmů) a `PI` (π , Ludolfov číslo). Některé z metod ukazuje tabulka 6.5. Většina těchto metod očekává parametr typu `double` a vrácí hodnotu typu `double`.

Metoda	Význam	Metoda	Význam
<code>abs()</code>	absolutní hodnota	<code>asin()</code>	arkussinus
<code>atan()</code>	arkustangens	<code>atan2()</code>	arkustangens podílu
<code>ceil()</code>	nejbližší vyšší celé číslo	<code>cos()</code>	kosinus
<code>exp()</code>	E^x	<code>floor()</code>	celá část čísla
<code>log()</code>	přirozený logaritmus	<code>max()</code>	větší ze dvou čísel
<code>min()</code>	menší ze dvou čísel	<code>sin()</code>	sinus
<code>sqrt()</code>	druhá odmocnina	<code>tan()</code>	tangens
<code>rand()</code>	náhodné číslo	<code>round()</code>	zaokrouhlení

Tab. 6.5 Některé ze statických metod třídy `Math`

Poznamenejme, že tyto matematické funkce je třeba při volání kvalifikovat jménem třídy `Math`, např. `Math.sin(x)`.

Obalové třídy

Obalové třídy pro reálná čísla se jmenují `Double` a `Float`. Obsahují mj. konstanty `MIN_VALUE` a `MAX_VALUE`, které určují nejmenší a největší kladné hodnoty různé od 0, které lze do této typů uložit. (Jde o hodnoty z tab. 6.4.) Dále tu najdeme konstanty `POSITIVE_INFINITY` a `NEGATIVE_INFINITY`, které vyjadřují „strojové nekonečno“, metodu `isInfinite()` pro testování, zda jde o nekonečnou hodnotu, a mnohé další.

Logické hodnoty

Pro vyjadřování logických hodnot používáme v Javě typ `boolean`. Tento typ má pouze dvě hodnoty, vyjádřené klíčovými slovy `true` a `false`, které znamenají pravdu a nepravdu.³³ Logické hodnoty vznikají jako výsledky relačních operátorů `<`, `<=` a dalších.

Pro vytváření složitějších výrazů (např. z několika podmínek) můžeme použít operátor negace `!`, logického součinu (konjunkce) `&&` a logického součtu (disjunkce) `||`. Jejich význam shrnuje tabulka 6.6. Poznamenejme, že operátor konjunkce `&&` použijeme, jestliže chceme, aby dvě podmínky platily zároveň, kdežto operátor disjunkce `||` použijeme, chceme-li, aby platila alespoň jedna z podmínek.

A	B	<code>!A</code>	<code>A && B</code>	<code>A B</code>
<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	<code>true</code>

Tab. 6.6 Význam logických operátorů

Tato tabulka např. říká, že má-li proměnná A hodnotu `true` a proměnná B hodnotu `false`, má výraz `A && B` hodnotu `false`.

Vedle toho můžeme logické hodnoty porovnávat pomocí operátorů `==` a `!=`.

Neúplné
vyhodnocení

Z tabulky 6.4 je vidět, že má-li první operand hodnotu `true`, bude výsledek operace disjunkce `A || B` pravdivý, ať má druhý operand jakoukoli hodnotu. Proto je-li první operand pravdivý, druhý se již nevyhodnotí (nevypočte), protože ho není třeba k určení hodnoty výsledku.

C⁺

³³ Na rozdíl od jazyků C a C++ nepatří v Javě typ `boolean` mezi celočíselné typy. Číselné hodnoty se nekonvertují automaticky na typ `boolean`.

Podobně má-li první operand hodnotu `false`, bude výsledek operace konjunkce `A && B` nepravdivý, ať má druhý operand jakoukoli hodnotu. Proto je-li první operand nepravdivý, druhý se již nevyhodnotí.

Ve čtvrté kapitole jsme při rozkladu řetězce na jednotlivá slova požadovali, aby index `i` byl menší než délka řetězce `radek` a zároveň aby znak s indexem `i` neležel v řetězci `oddelovace`. Protože chceme, aby byly tyto podmínky splněny zároveň, použijeme operátor konjunkce:

```
(i < radek.length()) &&  
    (oddelovace.indexOf(radek.charAt(i)) == -1))
```

Přitom se nemusíme bát, že při `i >= radek.length()` překročíme meze řetězce a vznikne chyba: Není-li první podmínka splněna, druhá se již nebude vyhodnocovat.

- Úplné vyhodnocení** Pokud bychom z nějakých důvodů trvali na úplném vyhodnocení, použili bychom místo `&&` a `||` operátorů `&` a `|`. Tyto operátory dávají stejné výsledky jako `&&` a `||`, vždy se však vypočte hodnota obou operandů. To znamená, že jsou-li `F()` a `G()` metody vracející hodnotu typu `boolean`, způsobí výraz `F() | G()`, že se za všech okolností zavolají obě.

- Obalová třída** Obalová třída pro logické hodnoty se jmenuje `Boolean`.

„Prázdný“ typ `void`

- C⁺** Z formálních důvodů se mezi primitivní typy někdy také zařazuje `void`. Toto klíčové slovo ovšem můžeme použít pouze ve specifikaci typu návratové hodnoty metody, nikde jinde. Na rozdíl od C/C++ ho nelze použít v hlavičce metody k vyznačení, že metoda nemá parametry. Nelze ho také použít k přetypování (k „zahození“ hodnoty). To znamená, že následující konstrukce běžné v C/C++ jsou v Javě chybějící:

```
int f(void){           // Nelze  
    (void)g();          // Nelze
```

Balení a vybalování

- 5** Java 5 umožňuje automatický převod (konverzi) primitivních typů na jejich obalové typy a naopak. Automatický převod hodnoty primitivního typu na hodnotu odpovídajícího obalového typu označujeme jako *balení* nebo *zabalení*, opačný převod jako *vybalení*; v angličtině se používá termín *autoboxing*.

Zabalené hodnoty lze použít i v aritmetických výrazech.

Příklad: Podivejme se nejprve na následující příklad, který sice postrádá jakýkoli hlubší smysl, ale ukazuje, jak to funguje.



```
// Soubor Kap06\07\Bal.java
public class Bal
{
    public static void main(String[] s)
    {
        Integer n = 3;           // Proměnná n typu Integer je
        n++;                     // inicializována hodnotou
        System.out.printf("%d", n); // a vypsána jako int
    }
}
```

Balení nejčastěji použijeme ve spojitosti s parametrizovanými kontejnery. Do kontejnerů lze ukládat pouze odkazy na objekty, nikoli hodnoty referenčních typů. Pokud potřebujeme kontejner pro některý z primitivních typů, musíme místo něj použít kontejner pro obalový typ.

Příklad: V programu potřebujeme grafický objekt představující lomenou čáru. To je vlastně posloupnost bodů, které jsou spojeny úsečkami. Třídu LomenaCara bychom mohli deklarovat např. takto:



```
// Soubor Kap06\07\TestLom.java
class LomenaCara
{
    private ArrayList<Integer> X, Y;
    public LomenaCara(ArrayList<Integer> XX,
                       ArrayList<Integer> YY)
    { /* ... */ }
    public void nakresli()
    { /* ... */ }
    public void posunX(int d)
    { /* ... */ }
    public void posunY(int d)
    { /* ... */ }
    // a další složky a metody
}
```

Souřadnice *x* a *y* jednotlivých bodů uložíme do složek *X* a *Y* typu *ArrayList<Integer>*. Konstruktor je pro nás nezajímavý, takže si jen řekneme, že uloží hodnoty, předané v parametrech, do *X* a *Y*.

Protože kreslit v Javě zatím neumíme, použijeme v metodě *nakresli()* oblíbený trik – vypíšeme zprávu, že se kreslí lomená čára. Metody *posunX()* a *posunY()* posunou celou čáru o zadaný počet bodů ve směru osy *x*, resp. *y*. To znamená, že ke přičtu danou hodnotu kaž-

démou z čísel, uložených v X, resp. v Y. Podivejme se na metodu posunX(); metoda posunY() je podobná.

```
public void posunX(int d)
{
    int i = 0;
    while(i < X.size())
    {
        X.add(i, X.remove(i)+d); // Zde dojde ke změně
        i++;
    }
}
```

O změnu hodnoty, uložené v i-tém prvku pole X, se stará příkaz označený komentářem. Metoda remove() odstraní i-tý prvek z kontejneru a vrátí odkaz na něj (jako odkaz na Integer). Tato hodnota se automaticky vybalí a přičte se k ní požadovaný posun d. Výsledek – hodnota typu int – se automaticky zabalí a uloží na původní místo pomocí přetížené metody add().♦

6.2 Skupina proměnných: pole

Pole je, jak víme, skupina proměnných stejného typu, se kterou zacházíme jako s celkem. Jednotlivé prvky pole jsou očíslovány a přistupujeme k nim pomocí jejich čísel v poli, tzv. *indexů*.

V Javě má první prvek pole vždy index 0. To znamená, že v poli s N prvky má poslední z nich vždy index N-1.

S poli pracujeme v Javě pomocí odkazů (referenci), podobně jako s objekty. V deklaraci pole vytvoříme odkaz; pole samotné musíme vytvořit dodatečně.

Deklarace a vytvoření pole

Deklarace pole

Deklaraci pole (přesněji deklaraci odkazu na pole) můžeme zjednodušeně³⁴ popsat takto:

deklarace jednorozměrného pole bez inicializace:
• typ_prvku[] seznam_identifikátorů ;

Typ_prvku je typ prvků pole; prázdné lomené závorky za ním naznačují, že deklarujeme odkaz na pole, nikoli proměnnou primitivního typu. *Seznam_identifikátorů* jsou identifikátory deklarovaných odkazů na pole.

³⁴ Java připouští také zápis, ve kterém uvedeme závorky [] za identifikátorem deklarovaného pole. Upřednostňujeme ale uvedený způsob.

Všimněte si, že v deklaraci neuvádíme počet prvků pole. Ten specifikujeme později, při vytváření pole. Z toho plyně, že jedna proměnná může postupně obsahovat odkazy na pole s různým počtem prvků, pokud je typ prvků stejný.

Vytvoření pole

Pole vytvoříme pomocí operátoru `new`; hodnotu, kterou tento operátor vrátí, přiřadíme proměnné typu odkaz. Za operátorem `new` uvedeme typ prvků a v lomených závorkách i jejich počet. Podívejme se na příkazy

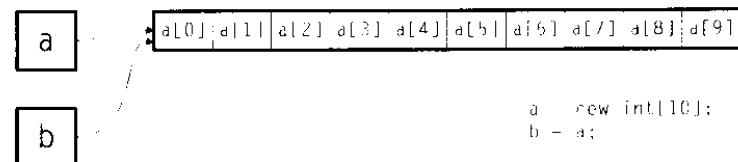
```
int[] a, b;  
a = new int[10];  
b = a;
```

V prvním z nich deklarujeme dva odkazy na pole s prvky typu `int`. Ve druhém vytvoříme pole 10 prvků typu `int` a odkaz na ně uložíme do proměnné `a`. Poslední příkaz okopíruje odkaz na toto pole také do proměnné `b`. Situaci ukazuje obr. 6.1

Pole zanikne, jestliže na něj nebude odkazovat žádná reference.

Použití prvků

Už jsme si řekli, že s prvky pole pracujeme pomocí jejich pořadových čísel – indexů. Ty zapisujeme do lomených závorek za identifikátor odkazu. Například prvek s indexem 3 v poli vytvořeném v předchozím příkladu můžeme použít zápisem `a[3]` nebo `b[3]`, neboť jak `a` tak i `b` odkazují na totéž pole.



Obr. 6.1 S poli pracujeme pomocí referencí



JVM za běhu programu důsledně hlídá, zda nepracujeme s neinicializovaným odkazem na pole nebo zda nepřekračujeme meze polí. Pokusíme-li se indexovat referenci, která obsahuje `null`, vznikne výjimka `NullPointerException`. Použijeme-li záporný index nebo index větší nebo rovný počtu prvků pole, vznikne výjimka `ArrayIndexOutOfBoundsException`.

Počet prvků pole

Jedna proměnná může postupně obsahovat odkazy na různě dlouhá pole. Abychom se vyhnuli nepříjemným chybám, které z toho mohou vzejít, obsahuje každé pole datovou složku `length`, ve které je uložen

počet prvků. Kdybychom např. chtěli vypsat všechny prvky pole `a`, mohli bychom použít např. příkazy

```
int i = 0;           // Počáteční prvek má index 0
while(i < a.length) // Poslední má index a.length-1
{
    System.out.println(a[i++]);
```

Typicky se ale pro zpracování všech prvků pole používá příkaz `for`, o němž budeme hovořit v kapitole 8.

Inicializace

Někdy známe prvky pole předem a chtěli bychom je uvést přímo v deklaraci. To jde. Jestliže za deklaraci odkazu na pole napišeme rovnitko a za ně do lomených závorek seznam hodnot oddělených čárkami, vytvoří se pole, tyto hodnoty se do něj uloží a odkaz na toto pole se předá deklarované proměnné. Počet prvků pole si překladač zjistí z počtu hodnot v závorkách. Například příkazem

```
double[] bod = { 1.1, 5.6, -3.14 };
```

vytvoříme pole o 3 prvcích a odkaz na ně uložíme do proměnné `bod`.

Příklad: Napišeme třídu `Kalendar`, která bude umět zjistit pořadové číslo dne v roce. Její metoda `cisloDne()` dostane jako parametry datum rozložené na den, měsíc a rok a vrátí celé číslo, které bude říkat, kolikátý den od počátku roku to je.

V této třídě si nejprve deklarujeme jako datovou složku pole `mesice`, do kterého uložíme počet dnů v jednotlivých měsících:

```
int[] mesice = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

V dalším poli, které nazveme `dny`, si uložíme počty dnů, které uplynuly od počátku roku vždy 1. dne daného měsíce. Prvek `dny[0]` bude obsahovat 0, neboť 1. ledna neuplynul ještě žádný den, prvek `dny[1]` bude obsahovat 31, neboť prvního února uplynulo 31 dnů (a běží dvaatřicátý) atd. Jinak řečeno, prvek `dny[i]` obsahuje součet prvků `mesice[0] + mesice[1] + ... + mesice[i-1]`. Toto pole vytvoříme a obsah jeho prvků vypočteme v konstruktoru třídy `Kalendar`.

```
Kalendar() {
    dny = new int[12]; // Vytvoření pole
    int i = 1;
    dny[0] = 0;
    while(i < dny.length)
        dny[i] = dny[i-1] + mesice[i-1]; // mesice
```

```
    i++;
}
}
```

Při výpočtu pořadového čísla dne musíme ještě vzít v úvahu možnost, že je přestupný rok, kdy má únor 29 dnů. V takovém případě musíme u dat následujících po 1. březnu přičíst 1.

Metoda `prestupny` bude vracet booleovskou hodnotu. Přestupný rok je rok, jehož číslo je dělitelné čtyřmi, pokud to není celé století. Pokud to je celé století, musí být dělitelné 400:

```
boolean prestupny(int rok)
{
    if(((rok % 4 == 0) && (rok % 100)!=0) || (rok % 400 == 0))
        return true;
    else return false;
}
```

Samotná metoda `cisloDne()` bude jednoduchá – sečte počet dnů, které uplynuly od počátku roku do konce předchozího měsíce, a den v měsíci.

```
int cisloDne(int den, int mesic, int rok)
{
    int d = dny[mesic-1]+den;
    if(prestupny(rok) && mesic > 2) d++;
    return d;
}
```



Třídu `Kalendar` spolu s programem, který ji používá, najdete v souboru `Kap06\05\Pokus.java`. Poznamenejme, že tento program není dokonalý – nekontroluje, zda datum, které dostal, je možné, zda uživatel nezadal např. 36. února. ♦

Kopírování polí

Chceme-li přenést obsah jednoho pole do jiného, nelze použít přiřazení, neboť bychom přenesli pouze odkaz na pole. Mohli bychom si samozřejmě naprogramovat kopírování jednoho prvku po druhém, např.

```
int a[] = {1,2,3,4,5};
int b[] = new int[5];
int i = 0;
while(i < a.length) { b[i] = a[i]; i++; }
```

Pole jsou ovšem v Javě zvláštní případ objektových typů, takže mají mj. metodu `clone()`, zděděnou od třídy `Object`. Tato metoda vytvoří kopii původního pole a vrátí odkaz na ni, ovšem jako odkaz na třídu `Object`, takže ho musíme přetypovat. Například takto:

```
int[] a = {1, 2, 3, 4, 5};           // Kopirované pole
int[] b = (int[]) a.clone();        // Vytvoření kopie
```

Pro kopírování části pole se hodí metoda `System.arraycopy`. Předchozí úsek programu bychom tedy mohli přepsat

```
int[] a = {1, 2, 3, 4, 5};
int[] b = new int[5];
System.arraycopy(a, 0, b, 0, a.length);
```

Pole `b` pak bude obsahovat stejné prvky jako pole `a`. První parametr této metody udává pole, které kopírujeme, druhý pozici (index prvku), od které kopírování začíná. Třetí parametr udává pole, do kterého kopírujeme, čtvrtý pozici, od které chceme okopirované hodnoty ukládat, a pátý určuje počet kopírovaných prvků.

Vícerozměrná pole

Pod tímto názvem se skrývají pole, jejichž prvky jsou zase pole. K jejich prvkům přistupujeme pomocí několika indexů.

Podívejme se na dvourozměrné pole, tj. na pole se dvěma indexy. To je pole, jehož prvky jsou jednorozměrná pole. V jejich deklaraci uvedeme dvakrát závorky `[]`. Např.

```
int[][] a;
```

je deklarace dvourozměrného pole. Vytvoříme-li ho příkazem

```
a = new int[2][3]
```

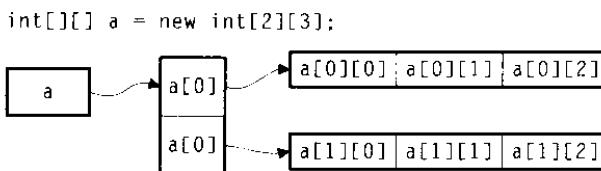
dostaneme pole s prvky `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]`.³⁵ Kdybychom si představili prvky dvourozměrného pole `a` vypsáne do tabulky (matice),

```
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]
```

bude první index určovat číslo řádku, druhý pak číslo sloupce. Proto o prvcích se stejným prvním indexem často hovoříme jako o „řádku pole“, „řádku matice“ nebo jen „řádku“. Podobně prvky se stejným druhým indexem označujeme jako „sloupec“. Pole `a` je tedy pole o dvou prvcích, a tyto dva prvky jsou odkazy pole o 3 prvcích typu `int`.

Prvky dvourozměrného pole jsou tedy odkazy na pole – např. `a[1]` je odkaz na řádek s indexem 1. (Viz též obr. 6.2.) To ale znamená, že při vytváření dvourozměrného pole `a` můžeme postupovat i jinak:

³⁵ Na rozdíl od Pascalu a některých jiných programovacích jazyků nesmíme napsat např. `a[1,2]`. Každý index musí být v samostatných závorkách.



Obr. 6.2 Dvourozměrného pole v Javě

Nejprve vytvoříme jednorozměrné pole odkazů na pole,

```
a = new int[5][]; // Druhý rozsah neuváděme
```

a pak vytvoříme jednotlivé řádky

```
for(int i = 0; i < a.length; i++) a[i] = new int[6];
```

Takto lze vytvořit i pole s různě dlouhými řádky. Podobně zacházíme také s vícerozměrnými poli.

Všimněte si, že `a.length` znamená počet polí, která tvoří `a`. Počet prvků v řádku `a[i]` najdeme v konstantě `a[i].length`.

Inicializace vícerozměrných polí

Při inicializaci vícerozměrných polí vyjdeme z toho, že to jsou vlastně jednorozměrná pole obsahující pole. Dvourozměrné pole tedy zapíšeme jako pole tvořené svými řádky:

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

Kopírování vícerozměrných polí

Pro vytvoření identické kopie vícerozměrného pole použijeme opět metodu `clone()`. Rozhodneme-li se použít metody `arraycopy()`, nemůžeme postupovat stejně jako v případě jednorozměrných polí, neboť bychom se dočkali nemilého překvapení:

```
int[][] a = {{1,2},{3,4,5}};
int[][] b = new int[5][5];
// Kopíruji se odkazy na řádky, nikoli řádky
System.arraycopy(a,0,b,0,a.length);
```

Tento úsek programu okopíruje pouze odkazy na jednotlivé řádky, takže kdybychom nyní napsali `a[1][1] = 546`, změnila by se i hodnota prvku `b[1][1]`. Proto musíme v cyklu okopírovat jednotlivé řádky:

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

```
int i = 0;
while(i < a.length)
{
    System.arraycopy(a[i],0,b[i],0,a[i].length);
    i++;
}
```

6.3 Objekty

S objekty pracujeme pomocí odkazů, podobně jako s polí. Je-li A třída, vytvoří deklarace

A a;

odkaz na instanci třídy A. Instanci pak vytvoříme pomocí konstruktoru a operátoru new:

A a = new A(); // Konstruktor může mít i parametry

Příkazem

A b = a; // Kopirujeme odkaz, nikoli instanci

uložíme do b odkaz na tutéž instanci, na niž odkazuje a; to je stejné jako u polí. Podrobněji budeme o objektových typech a objektech hovořit ve zbytku této knihy.

6.4 Výčtové typy

5 S výčtovými typy se setkáme až v Javě 5. Jde o typy, které představují malou, uzavřenou množinu předem pevně daných hodnot. Příkladem může být typ představující hlavní světové strany: Ten má pouhé čtyři hodnoty – sever, jih, východ a západ.

Deklaraci výčtového typu můžeme zjednodušeně popsat takto:

deklarace výčtového typu:

- *modifikátory* **enum** { *seznam_identifikátorů* ;*nes* }

Modifikátory vyjadřují přístupová práva (public, protected, private) – o tom budeme hovořit později, v kapitole věnované objektovým typům. Pak následuje klíčové slovo enum a za ním v lomených závorkách seznam identifikátorů vyjadřujících jednotlivé hodnoty („výčtové konstanty“). Jednotlivé identifikátory v tomto seznamu oddělujeme čárkami. Za seznamem může následovat středník.

Příklad: Deklarujeme výčtový typ představující světové strany.

enum Strany {SEVER, JIH, VYCHOD, ZAPAD;}

Deklarujeme-li nyní proměnnou

```
Strany s;  
můžeme jí přiřadit hodnotu např. příkazem  
s = Strany.SEVER;  
Proměnné s nelze přiřadit jinou hodnotu než Strany.SEVER, Stra-  
ny.JIH, Strany.VYCHOD nebo Strany.ZAPAD.36 Hodnoty výčtového  
typu lze porovnávat pomocí operátoru ==. ♦
```

Pohled pod pokličku

Výčtové typy jsou ve skutečnosti zvláštním případem objektových typů. Identifikátory výčtových hodnot jsou identifikátory konstantních odkazů na jediné instance tohoto typu, které existují.

Všechny výčtové typy obsahují metody `toString()` a `name()`, jež obě vracejí znakový řetězec představující jméno výčtové konstanty. To znamená, že příkaz

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

kde s je proměnná definovaná výše, vypíše SEVER.

Metoda `ordinal()` vráti pořadové číslo dané výčtové hodnoty, přičemž první deklarovaná hodnota má pořadové číslo 0, druhá 1 atd.; např. v typu `Strany` platí, že `Strany.SEVER.ordinal() == 0`.

Metoda `values()` vráti odkaz na pole výčtových hodnot daného typu (přesněji odkaz na pole odkazů na jednotlivé výčtové hodnoty). Prvek s indexem 0 takto získaného pole obsahuje odkaz na první hodnotu výčtového typu (hodnotu s pořadovým číslem 0) atd.

Příklad: Napišeme jednoduchý program, v němž deklarujeme výčtový typ představující světové strany a pak je vypíšeme. Za hodnotu JIH připojíme vykřičník.

```
  
C++ // Soubor Kap06\06\SvStr.java  
enum Strany {SEVER, JIH, VYCHOD, ZAPAD;}  
  
public class SvStr  
{  
    public static void main(String[] arg)  
    {  
        Strany[] s = Strany.values(); // Získáme pole hodnot  
        int i = 0;  
        while(i < s.length) // a to v cyklu vypíšeme  
        {
```

³⁶ V tom se pojetí výčtových typů v Java zásadně liší od pojetí výčtových typů v C nebo v C++.

```

        System.out.print(s[i].name());
        if(s[i] == Strany.JIH) System.out.println("!");
        else System.out.println();
        i++;
    }
}

```

Ve skutečném programu bychom k výpisu všech prvků pole použili příkaz `for`, s nímž se seznámíme v 8. kapitole. ♦

To je jen začátek

Deklarace výčtového typu může vedle výčtových konstant obsahovat i další datové složky a metody; výklad o nich ovšem přesahuje možnosti knihy pro zelenáče. Podrobnější informace lze najít např. v dodatku k českému vydání knihy [12].

6.5 Automatická správa paměti

Všechny instance objektových typů (objekty) jsou tzv. dynamické; to znamená, že je vytváříme pomocí operátora `new` a pracujeme s nimi pomocí odkazů. Podobně jsou dynamická i pole.

Např. novou instanci třídy `Seznam` vytvoříme zápisem `new Seznam()`, který se skládá z klíčového slova `new` a volání konstruktoru třídy `Seznam`. Vytvořený odkaz můžeme přiřadit proměnné vhodného typu, předat metodě jako parametr atd.

O rušení instancí objektových typů a polí se programátor v Javě nestará.³⁷ JVM u každého z objektů a polí sleduje, kolik existuje odkazů na ně, a pokud zjistí, že žádný, zruší ho (uvolní jeho paměť). O to se, jak už víme, stará mechanizmus zvaný `garbage collector` neboli automatická správa paměti.

To je na jedné straně velice pohodlné, neboť odpadá řada chyb v práci s pamětí, naprostě běžných v jazycích C, C++, Pascal aj., které automatickou správy paměti nemají. Na druhé straně to ovšem znamená větší nároky na počítač, pomalejší běh programu ap.

Jak mohou odkazy na pole nebo objekty zaniknout? Je několik možností:

- Proměnné, která představuje odkaz na objekt nebo pole, přiřadíme hodnotu `null` nebo odkaz na jiný objekt.

³⁷ Java neobsahuje žádný příkaz, který by umožňoval explicitně zrušit objekt nebo pole. Proto také v Javě neexistuje analogie destruktérů známých z C++ nebo z Pascalu.

- Proměnná, která představuje odkaz, zanikne sama. Takto mohou zaniknout lokální proměnné nebo formální parametry metod poté, co program opustí tělo metody. Může také zaniknout objekt, který proměnnou obsahuje (protože ho zrušil garbage collector).
- O spuštění garbage collectoru se stará JVM. Pokud bychom ale chtěli, můžeme ho zavolat sami pomocí metody `System.gc()`, která je bez parametrů.

Příklad: Ve 4. kapitole v oddílu 4.3 jsme implementovali jednosměrně zřetězený seznam. Třída `Seznam` ovšem obsahovala jen metody, které jsme nezbytně potřebovali pro zvládnutí problému, jímž jsme se v této kapitole zabývali (šlo o spočítání výskytů různých slov v daném textu). Seznam je ale univerzální kontejner, který se může hodit i při jiných příležitostech. Proto ho zkuste vylepšit a přidejme do něj metodu `vyprazdni()`, která ze seznamu odstraní všechny prvky. Výsledkem její činnosti tedy bude prázdný seznam.

Její implementace v Javě je velice jednoduchá:

```
class Seznam {
    Prvek hlava = null;           // Ukazatel na první prvek
    public void vyprazdni() {    // Vyprázdní seznam
        hlava = null;
    }
    // Ostatní metody jako v kap. 4.3
}
```

Vše, co musíme udělat, je přiřadit ukazateli `hlava` na první prvek seznamu hodnotu `null` a tím zrušit odkaz na první prvek. Pokud na něj neexistuje žádný další odkaz, tento prvek zanikne. Tím ovšem zanikne odkaz na druhý prvek, a pokud na něj neexistuje žádný další odkaz, také tento prvek zanikne atd. Pokud na některý z prvků existuje ještě jiný odkaz, tento prvek nezanikne, ale to je v pořádku, neboť prostřednictvím tohoto odkazu ho nejspíš někdo ještě bude chtít použít. Spolu s tímto prvkem zůstanou zachovány i prvky následující. ♦

6.6 Proměnné

Už víme, že proměnná v Javě je vlastně pojmenovaná oblast paměti. Abychom mohli proměnnou používat, musíme ji nejprve deklarovat.

Deklarace

Ukažme si zjednodušený popis syntaxe deklarace proměnné:

deklarace proměnné:

- **static**_{nep} **final**_{nep} **typ seznam_deklarátorů;**

Seznam_deklarátorů, to jsou jednotlivé *deklarátory* oddělené čárkami.
Deklarátor je *identifikátor*, za kterým může následovat *inicializace*:

Deklarátor:

- *identifikátor*
- *identifikátor = výraz*

Konstanty Klíčové slovo **final** označuje konstanty, tj. proměnné, jejichž hodnoty se v programu nemohou měnit. Klíčové slovo **static** lze použít pouze pro datové složky a budeme o něm hovořit v kapitole 9.3.

Typ, identifikátor *Typ* označuje typ deklarovaných proměnných. Může to být klíčové slovo označující některý z primitivních typů, konstrukce představující pole nebo identifikátor třídy (objektového typu). *Identifikátor* představuje jméno nově deklarované proměnné.

Neinicializované proměnné

Jestliže za identifikátor proměnné v deklaraci připíšeme rovnítko a za ně výraz, předepisujeme tím inicializaci proměnné: Hodnota tohoto výrazu se proměnné přidělí při vytvoření. Co se stane, jestliže proměnnou v deklaraci neinicializujeme? Odpověď závisí na okolnostech.

- Jestliže se jedná o datovou složku třídy, postará se o její inicializaci překladač. Proměnné číselných typů přidělí hodnotu 0, znakovým proměnným hodnotu '\u0000', logickým proměnným hodnotu `false` a odkazům na objekty nebo pole hodnotu `null`.
- Jestliže se jedná o lokální proměnnou deklarovanou v těle metody, bude její obsah náhodný. O uložení nějaké smysluplné hodnoty do ní se musíme postarat později sami. Ovšem pozor -- překladač rozpozná, že jsme danou proměnnou dosud neinicializovali, a nedovolí nám použít její hodnotu.

V následujícím příkladu ohláší překladač chybu, neboť používáme proměnnou `z`, kterou jsme opomněli inicializovat:

```
void f() {
    int z;
    System.out.println(z); // CHYBA
    // ...
```

7 Výrazy a operátory

Výraz předepisuje výpočet hodnoty (výsledku). Skládá se z *operandů* (daných hodnot, například konstant a proměnných) a *operátorů*, které říkají, jak se má výsledek vypočítat. Přesný syntaktický popis výrazu v Javě je poměrně komplikovaný a pro nás ve skutečnosti naprostě nezajímavý. Daleko důležitější je znát pravidla, která omezují používání jednotlivých operátorů, a o nich si v této kapitole povíme.



Souhrnný příklad k této kapitole najdete na WWW v souboru Kap07\Souhrn.java.

7.1 Vlastnosti operátorů

Priorita Z matematiky jsme zvyklí, že ve výrazu $a + b \times c$ se nejprve vypočte součin $b \times c$ a teprve pak se přičte a , neboť násobení má přednost před sčítáním. (Říkáme, že násobení má *vyšší prioritu* než sčítání.) Pokud se nám toto pořadí výpočtu nelibí, můžeme předepsat jiné pomocí závorek – např. $(a + b) \times c$ znamená, že chceme nejprve vypočítat součet $(a + b)$ a ten pak vynásobit c .

Podobná pravidla platí i v Javě: Ve výrazu $a + b * c$ se také nejprve vypočte součin $b * c$ a k výsledku se pak přičte a . Chceme-li toto pořadí změnit, můžeme použít závorky: $(a + b) * c$ opět znamená, že chceme nejprve sčítat a teprve pak násobit. V Javě smíme použít pouze kulaté závorky, neboť hranaté [] a složené {} jsou vyhrazeny pro jiné účely.

Operátory v Javě jsou rozděleny do skupin podle priorit (viz tabulka 7.1). Ve skutečnosti ale nemá smysl si priority operátorů pamatovat; stačí vědět, že násobení a dělení má přednost před sčítáním a odečítáním, aritmetické operace před relacemi (porovnávání ap.) a všechny tyto operace mají přednost před přiřazováním. (To lze považovat za „přirozená“ pravidla, protože víceméně odpovídají běžným zvyklostem.) Pokud si nebudeme jisti, použijeme závorky; výraz tak bude i srozumitelnější.

Asociativita Někdy může být důležité i pořadí, ve kterém se vyhodnocují operátory se stejnou prioritou. Například výraz $a + b + c$ se vyhodnotí, jako kdyby byl uzávorkován $(a + b) + c$. (Říkáme, že operátor + je asocia-

ativní zleva doprava, neboť nejdříve se vyhodnotí sčítání zapsané nejvíce vlevo.) Podobně se chová většina operátorů v Javě; existují ovšem i operátory, které jsou asociativní zprava doleva. Mezi ně patří např. přiřazovací operátor `=`. Výraz `a = b = c` se v Javě vyhodnotí jako `a = (b = c)`.

7.2 Aritmetické výrazy

To jsou výrazy, které znamenají výpočet číselné hodnoty. Jako operandy v nich mohou vystupovat hodnoty číselných a znakových typů. Jako operátory můžeme použít unární `+` a `-`, binární operátory `+`, `-`, `*`, `/` a `%` (sčítání, odečítání, násobení, dělení, zbytek po dělení) a bitové operátory `^`, `&`, `|`, `<<`, `>>` a `>>>`. Pro výpočty můžeme použít také operátory inkrementace a dekrementace `++` a `--` a složené přiřazovací operátory `+=`, `*=` atd. O většině z nich jsme hovořili v předchozí kapitole.

Smíšené výrazy

V aritmetických výrazech mohou vedle sebe vystupovat operandy různých typů. Jakého typu pak bude výsledek? To není jen akademická otázka: Bude-li výsledek např. typu `long`, nedovolí nám Java přiřadit ho proměnnému typu `int` (budeme ho muset přetypovat).

Pravidla pro typ výsledku při aritmetických výpočtech lze shrnout do následujících bodů:

1. Nejprve se operandy typu `byte`, `short` a `char` převedou (rozšíří) na `int`.
2. Pak se v případě binárních operátorů porovnají typy obou operandů. Jsou-li stejné, bude stejněho typu i výsledek. Pokud ne, převeďte se operand typu s menším rozsahem na typ druhého operandu (s větším rozsahem). Tohoto typu pak bude i výsledek.
3. Číselné typy jsou v Javě seřazeny podle rozsahu takto: `int` → `long` → `float` → `double`. Šipky směřují od typů s menším rozsahem k typům s větším rozsahem.

Tato pravidla říkají, že výsledek nemůže být typu `byte`, `char` nebo `short`. To znamená, že pokud napišeme např.

```
short x = 11;  
x += x;
```

ohlásí překladač chybu (možnou ztrátu přesnosti), neboť hodnota `x` na pravé straně přiřazení je typu `int`, zatímco proměnná `x` na levé

straně je typu `short`. Chceme-li něco podobného napsat, musíme použít přetypování:

```
x = (short)-x;
```

Podívejme se jiný příklad:

```
int a = 11;
long b = 25;
long c = b/a;
```

Nejprve se oba operandy převedou podle druhého pravidla na typ `long`. To znamená, že se i výsledek bude typu `long` (použije se tedy celočíselné dělení). Jeho hodnota bude 2 a ta se uloží do `c`.

Změníme-li typ jednoho z operandů, např.

```
int a = 11;
double b = 25;
double c = b/a;
```

převede se nejprve `a` na typ `double`. Tohoto typu bude i výsledek. To znamená, že se použije dělení reálných čísel a výsledek bude mít hodnotu 2,27272727...

Složitější výrazy se vyhodnocují po jednotlivých podvýrazech. Podívejme se na příklad:

```
int i = 11, j = 22;
long k, l = 2;
k = i * j + l;
```

Nejprve se bude vyhodnocovat výraz `i * j`. Protože jsou oba operandy typu `int`, bude i výsledek typu `int`. Pak se k výsledku přičte `l`. Tato proměnná je typu `long`, a proto se i předchozí výsledek nejprve převede na `long`.



Pokud vám to připadá nedůležité, pak se podívejte na trochu pozeměný příklad:

```
int i = 2000000000, j = 2000000000;
long l = 1;
System.out.println(i + j + l);
```

Poslední příkaz vypíše -29496/295. To by nám mělo připadat podivné: Součet `i + j + l` je typu `long`, a do rozsahu typu `long` se správný výsledek, hodnota 4000000001, přece vejde.

Problém je ale v tom, že se nejprve vypočte součet `i + j`, a protože oba operandy jsou typu `int`, bude i výsledek typu `int` a do rozsahu typu `int` se hodnota součtu 4000000000 již nevejde, takže dojde k přetečení a výsledek nebude mít smysl. Jestliže ovšem předepišeme pomocí závorek jiné pořadí sčítání,

```
System.out.println(i + (j + 1));
```

dostaneme správný výsledek. Zde se totiž nejprve sčítá j typu `int` a l typu `long`. Proto se j převede na `long` a výsledek je typu `long`. K tomu se pak přičte i, ale to se také převede na `long`. Všechny operace tedy proběhnou s proměnnými typu `long`, a proto nedojde k přetečení.

7.3 Relační výrazy

Relační výrazy představují hodnoty typu `boolean`. Vyjadřují relace tedy zkoumají, zda platí vztahy „je menší“, „je rovno“, „je menší nebo rovno“ atd. Mohou obsahovat relační operátory `<`, `<=`, `>`, `>=`, `==` a `!=`.

Mezi relační operátory patří také `instanceof`, který umožňuje testovat příslušnost instance ke třídě.

 Často se setkáme s podmínkou tvaru $0 < x \leq 1$. Poměrně běžnou chybou začínajících programátorů je, že ji do programu opíší doslova:

```
if(0 < x <= 1) ... // NEJZT
```

Ve skutečnosti jde o spojení dvou podmínek, $0 < x$ a $x \leq 1$, které mají platit zároveň (konjunkce). Proto musíme v programu napsat

```
if((0 < x) && (x <= 1)) ... // OK
```

7.4 Logické výrazy

Logické výrazy mají hodnotu `true` nebo `false` typu `boolean`. Mohou obsahovat proměnné a konstanty typu `boolean`, volání funkcí, které vracejí hodnotu typu `boolean`, a relační výrazy.

K vytváření logických výrazů můžeme používat operátory `&&` a `||`, které znamenají konjunkci, resp. disjunkci s neúplným vyhodnocením, `&` a `|`, které znamenají konjunkci, resp. disjunkci s úplným vyhodnocením, a operátor `!`, který znamená negaci. (Podrobněji jsme o nich hovořili v kapitole 6.1.)

7.5 Přehled operátorů

V tabulce 7.1 najdete přehled operátorů, které máme v Javě k dispozici. Vyšší priorita znamená přednost při vyhodnocování. O velké většině z nich jsme již hovořili; o bitových operátorech hovořit nebudeme. Zbývá zastavit se podrobněji u operátorů `?:`, přetypování, přiřazení a čárka.

Operátor	P	A	Význam
. (tečka)	1	→	Přístup ke složce instance
[]	1	→	Indexování (přístup ke složkám pole)
(typ)	1	←	Přetypování
!	2	←	Negace
++ --	2	←	Inkrementace o 1, dekrementace o 1
+ -	2	←	Unární plus, unární minus (otočení znaménka)
~	2	←	Bitový doplněk
* / %	3	→	Násobení, dělení, modulo (zbytek po dělení)
+ -	4	→	Sčítání, odečítání; + také spojování řetězců
<< >> >>	5	→	Bitové posuny
< > <- >-	6	→	Test nerovnosti (větší, větší nebo rovno atd.)
== !=	7	→	Test rovnosti (rovna se, resp. nerovna se)
&	8	→	Logický součin (konjunkce) s úplným vyhodnocením, logický součin po bitech
^	9	→	Nonekvivalence po bitech
	10	→	Logický součet (disjunkce) s úplným vyhodnocením, logický součet po bitech
&&	11	→	Logický součin (konjunkce) s neúplným vyhodnocením
	12	→	Logický součet (disjunkce) s neúplným vyhodnocením
? :	13	→	Podmíněný výraz
:=	14	←	Prosté přiřazení
+ - * / % ^ = & << >> >>=	14	←	Složené přiřazení: Je-li @ některý z operátorů uvedených před ~ v tomto výčtu, znamená výraz a @= b totéž co a ~ a @= b.
, (čárka)	15	→	Postupné vyhodnocení (jen ve for)

Tab. 7.1 Přehled operátorů. Ve sloupci **P** je uvedena priorita, ve sloupci **A** asociativita. Nižší číslo znamená vyšší prioritu

Podmíněný výraz

Často se setkáme s výpočtem, kde výsledek závisí na splnění nějaké podmínky. Chceme např. napsat metodu `max()`, která bude vracet větší ze dvou čísel. Řešení založené na příkazu `if` je jednoduché:

```
int max(int a, int b) {  
    if(a > b) return a;  
    else return b;  
}
```

Nevýhodou příkazu `if` je, že ho nemůžeme použít jako součást výrazu. Proto nabízí Java operátor `? :`, který používáme takto:

Syntax použití operátoru podmíněného výrazu:
• `podminka ? výraz_1 : výraz_2`

Nejprve se vyhodnotí podmínka. Je-li splněna, je výsledkem `výraz_1`, jinak je výsledkem `výraz_2`. (Oba výrazy musí být stejného typu nebo musí být možné převést je na stejný typ.)

Metoda `max()` by s použitím operátoru podmíněného výrazu mohla vypadat takto:

```
int max(int a, int b) {  
    return a < b ? b : a;  
}
```

Podmíněný výraz můžeme použít tam, kde se příkaz `if` nehodí. Chceme-li předat větší ze dvou čísel jako parametr metodě `f()` a nechceme-li k tomu deklarovat zvláštní metodu (nebo používat standardní metodu `Math.max()`), můžeme napsat `f(a < b ? b : a)`.

Konverze a přetypování

V mnoha situacích potřebujeme hodnotu jednoho typu převést – konvertovat – na hodnotu jiného typu. Některé konverze mohou v Javě proběhnout automaticky. To jsou ty, při nichž se neztrácí informace („rozšiřující“ konverze). Pro číselné typy je můžeme vyjádřit následujícím schématem:

`byte → short → int → long → float → double` (1)

Vedle toho může automaticky proběhnout konverze typu `char` na `int`. Z pravidel objektového programování, o nichž jsme hovořili v první kapitole, také plyne, že by měla být (a je) automatická konverze odkažu na instanci odvozené třídy na odkaz na instanci bázové třídy. Protože v Javě mají všechny třídy společného předka, třídu `Object`, znamená to, že proměnné `obj`, deklarované

`Object obj;`

můžeme přiřadit odkaz na jakoukoli instanci jakékoli třídy.

Vedle toho jsou konverze, které Java sice umí, ale nedělá je automaticky. To mohou být „zužující“ konverze (proti směru šipek ve schématu (1), např. typu `int` na `byte`), konverze typu `int` na `char` nebo konverze odkazu na předka na odkaz na potomka.

Syntax Tyto konverze musíme explicitně předepsat pomocí operátoru přetypování, který má tvar `(typ) objekt` (jméno cílového typu uzavřené v závorkách) a zapisuje se před konvertovaný výraz.

Pokud přetypování není možné, protože ho pravidla Javy zakazují, ohlásí překladač chybu.

Existují ovšem situace, kdy překladač nemůže poznat, zda je přetypování možné. Budeme-li např. mít odkaz `obj` na typ `Object` a pokusíme-li se přetypovat ho na odkaz na instanci třídy `Pokus`, nemůže překladač vědět, zda bude `obj` opravdu obsahovat odkaz na instanci této třídy. Proto se správnost takového přetypování kontroluje až za běhu programu. Pokud přetypování není možné, vznikne výjimka typu `ClassCastException`.

Operátor čárka

Tento operátor lze použít pouze v inicializační a v aktualizační části příkazu `for`, o němž budeme hovořit v příští kapitole. Jeho syntax je

Syntax použití operátoru čárka:
• `výraz_1, výraz_2`

Nejprve se vyhodnotí `výraz_1`, pak `výraz_2`, a ten je výsledkem. V příkazu `for` umožňuje tento operátor zadat několik inicializací a několik aktualizací. Například takto:

```
int i, j;  
for(i = 0, j = n-1; i < n; i++, j--) f(i,j);
```

Přiřazovací operátor

Prostý přiřazovací operátor = používáme v přiřazovacích výrazech

Syntax použití prostého přiřazovacího operátoru:
• `l-hodnota = výraz`

Takovýto výraz znamená „vypočti `výraz` na pravé straně operátoru = a jeho hodnotu ulož do místa, které představuje `l-hodnota` na levé straně.“

ně“. Hodnotou (výsledkem) výrazu $a = b$ je to, co se uloží do proměnné a .

Za označením *l-hodnota* se skrývá proměnná (nekonstantní), formální parametr, prvek pole a jiné výrazy, které určují místo v paměti, jehož obsah lze měnit. Hodnota výrazu musí být stejného typu jako *l-hodnota* nebo ji na typ této *l-hodnoty* musí být možné převést automatickými konverzemi. (Proměnné typu `int` můžeme přiřadit hodnotu typu `byte`, ale proměnné typu `byte` nelze přiřadit hodnotu typu `int` – o tom jsme již hovořili.)

Přiřazovací příkaz představuje sám o sobě hodnotu. Z toho plyne, že přiřazení můžeme zřetězit – můžeme napsat

$a = b = c$



Přiřazení lze ale použít všude, kde se očekává nějaká hodnota. Ve čtvrté kapitole jsme napsali program, který analyzoval slova v textovém souboru (viz soubor `Kap04\01\Analyzer.java` na WWW). Metoda `analyzuje()` obsahovala cyklus

```
i = preskocOddelovace(radek, i);
while(i >= 0)
{
    // nějaké zpracování
    i = preskocOddelovace(radek, i);
}
```

ve kterém jsme před vstupem do cyklu zjistili hodnotu i , pak jsme testovali, zda je podmínka opakování splněna, zpracovali data v cyklu a nakonec jsme zjistili novou hodnotu i . To je správné, lze to ale zapsat jednodušeji:

```
while((i = preskocOddelovace(radek, i)) >= 0) //Znamená totéž
{
    // nějaké zpracování
}
```

Hodnotu vrácenou metodou `preskocOddelovace()` uložíme do i , a pokud je tato hodnota nezáporná, zpracujeme ji. Poznamenejme, že výraz $i = preskocOddelovace(radek, i)$ jsme museli uzavřít do závorek, neboť operátor $=$ má nižší prioritu než $>=$.

Složené přiřazení

Význam složených přiřazovacích operátorů $+=$, $-=$ a dalších už známe: výraz $a @= b$ znamená téměř přesně totéž co $a = a @ b$. Přitom $@$ může být některý z operátorů $+, -, *, /, %, <<, >>, >>>, \&, |, ^$.

Kromě toho, že znamenají méně psaní, mohou tyto operátory usnadnit překladači optimalizaci.

8 Příkazy

Příkazy řídí chod programu, tj. vyjadřují kroky, které má program udělat. Můžeme se na ně divat jako na elementární kroky, které slouží k zápisu algoritmu v Javě. Obvykle se provádějí v pořadí, ve kterém jsou v programu zapsány; některé ovšem mohou způsobit, že dojde k „přenosu řízení“, tj. že program přejde na příkaz, který právě není na řadě.

Středník Součástí některých příkazů v Javě je středník. To znamená, že ho v těchto příkazech prostě uvádět musíme bez ohledu na to, kde je zapsán. (To je rozdíl např. oproti Pascalu, kde středníky slouží jako oddělovače a lze je v určitých situacích vynechat.)

C⁺ Příkazy v Javě jsou velmi podobné příkazům v C++. Jediné závažnější rozdíly lze shrnout do následujících bodů:

- V Javě chybí příkazy `asm` a `goto`.
- Je rozšířen význam příkazů `break` a `continue`.
- Deklarace není příkaz.
- Počínaje JDK 1.4 je k dispozici příkaz `assert`.
- V Javě 5 je k dispozici varianta příkazu `for` pro postupné zpracování všech prvků kontejneru.

8.1 Blok (složený příkaz)

Často zjistíme, že nám syntaktická pravidla Javy dovolují zapsat na určité místo jen jeden příkaz. Pokud jich tam potřebujeme více, použijeme blok neboli složený příkaz. To je skupina příkazů a deklarací uzavřená mezi složené závorky `{` a `}`, která se z hlediska jazyka Java chová jako jediný příkaz. Syntaktický popis bloku vypadá takto:

blok:

- { *příkazy_nebo_deklarace_{nep}* }

Index `nep` naznačuje, že blok může být i prázdný, tj. nemusí obsahovat žádný příkaz ani deklaraci. Pak představuje tzv. prázdný příkaz, tj. příkaz, který nedělá nic. (I taková věc se v programech občas hodí.) Za ukončující závorku bloku nepíšeme středník.

V úvodních kapitolách této knihy jsme se seznámili se zjednodušenou podobou příkazu `while`, kde jsme jako tělo cyklu používali vždy blok.

Vnořené bloky

Součástí bloku může být další blok; pak hovoříme o *vnitřním* nebo *vnořeném bloku* a o bloku *obklopujícím*. Proměnné, které deklarujeme ve vnořeném bloku, nelze mimo tento blok používat. Kdybychom např. napsali

```
// Závorka otevírající blok
int a = 55;      // Deklarace uvnitř bloku
f(a);
// Závorka ukončující blok
a++;            // Chyba - neznámá proměnná
```

ohlásil by překladač na posledním řádku chybu, protože proměnnou *a* je lokální v předcházejícím bloku.



Ve vnořeném bloku nesmíme deklarovat proměnnou se stejným identifikátorem jako v bloku, který ho obklopuje. Například následující úsek programu označí překladač za chybný:

```
int x = 11;
double x = 2.81; // Chyba - opakování deklarace
```

8.2 Výrazový příkaz

Připojíme-li k výrazu středník, stane se z něj příkaz. Musí ovšem jít o výraz, který má nějaký vedlejší efekt – který změní hodnotu některé proměnné, zavolá metodu ap. Například přiřazení *i = 11* představuje výraz s hodnotou 11 a s vedlejším efektem, kterým je změna hodnoty proměnné *i*. Připojíme-li za tento výraz středník, dostaneme příkaz:

```
i = 11;
```

Jako výrazový příkaz lze také použít např. volání metody nebo aplikaci operátorů *++* nebo *--*. S příklady výrazových příkazů jsme se již mnohokrát setkali.

Prázdný příkaz

Prázdný příkaz je příkaz, který nedělá nic. Hodí se v situacích, kdy syntaktická pravidla požadují, abychom někde zapsali příkaz, a my přitom chceme, aby tam program nic nedělal.

V Javě ho můžeme vyjádřit buď samotným středníkem nebo prázdným blokem. (Samotný středník je vlastně „prázdný“ výraz, za který jsme připojili středník.)

prázdný příkaz:

- ;
- {}

I když to vypadá podivně, přece jen se prázdné příkazy používají poměrně často. Podívejme se na příklad: Máme řetězec s (typ `String`), ve kterém chceme najít první výskyt znaku jiného než 'a'. Jedním z možných řešení je cyklus

```
int i = -1;  
while(s.charAt(++i) == 'a')  
; // Prázdný příkaz tvořící tělo cyklu
```

jehož tělo je tvořeno prázdným příkazem.³⁸ Procházení jednotlivých znaků obstará operátor `++` v podmínce, takže po skončení cyklu bude `i` obsahovat hodnotu indexu prvního znaku jiného než 'a'. (Pokud řetězec s takový znak neobsahuje, skončí tento úsek programu výjimkou `StringIndexOutOfBoundsException`, neboť překročíme meze řetězce.)

8.3 Rozhodování

Tyto příkazy se často označují jako „podmíněné“. Umožňují vyjádřit větvení algoritmu, tj. předepsat, která část programu se má provést, v závislosti na hodnotě nějakého výrazu. Java obsahuje dva takové příkazy, `if` a `switch`.

Příkaz if

Tento příkaz už známe, alespoň částečně. Má dva tvary, které označujeme jako *neúplné if* a *úplné if*. Podívejme se na syntaktický popis tohoto příkazu:

příkaz if:

- `if(podminka) příkaz_1` // Neúplné if
- `if(podminka) příkaz_1 else příkaz_2` // Úplné if

Podminka je výraz s hodnotou typu `boolean`.³⁹ Závorky okolo ní jsou nezbytné.

Nejdříve se vypočte hodnota *podmínky*; je-li splněna (má-li hodnotu `true`) provede se *příkaz_1*. Není-li splněna (má-li hodnotu `false`),

³⁸ Je dobrým zvykem zapisovat v takovýchto případech prázdný příkaz na samostatný řádek. Program je pak přehlednější.

³⁹ Na rozdíl od jazyků C a C++ nelze v Javě použít jako podmínu výraz s číselnou hodnotou.

C⁺

neprovede se v případě neúplného `if` nic, v případě úplného `if` se provede `příkaz_2`.

O `příkazu_1`, resp. o `příkazu_2` občas hovoříme jako o „větvích“ `příkazu if`. (Větev za podmínkou, resp. větev za `else`.) Jestliže potřebujeme na místě `příkazu_1` nebo `příkazu_2` zapsat skupinu příkazů, použijeme blok.

Podívejme se na několik příkladů:

```
if(x == 0)
    System.out.println("Pomoc!");
else
    System.out.println("Vse je OK.");
```



Zde jsme v obou větvích úplného `if` použili jednoduché příkazy (tedy nikoli blok). Všimněte si středníku před `else`; pokud jste zvyklí na Pascal, bude vám to připadat podivné, v Javě je ale nezbytný – je totiž součástí výrazového příkazu.



Jako `příkaz_1` v neúplném `if` můžeme použít i další příkaz `if`. Tak může vzniknout např. konstrukce

```
if(x>0) if(y<0) x = 11; else x = 22;
```

Zde není na pohled jasné, ke kterému `if` patří uvedené `else`. V takovémto případě ho překladač sdruží s nejbližším předchozím nespárováným `if`. Z toho plyne, že předchozí příklad znamená totéž co

```
if(x>0) {if(y<0) x = 11; else x = 22;}
```

Pokud bychom chtěli, aby toto `else` patřilo k prvnímu `if`, museli bychom použít složené závorky, tj. uzavřít vnořené `if` do bloku:

```
if(x>0) {if(y<0) x = 11;} else x = 22;
```

Příkaz `switch` (přepínač)

Tento příkaz oceníme, jestliže potřebujeme rozvětvit program na několik částí. Jeho formální syntaktický popis je poměrně komplikovaný a nepřehledný, proto si ukážeme jen zjednodušenou podobu a tu doplníme vyčerpávajícím výkladem:

příkaz switch:

- `návěstidlo:nep switch(výraz) {`
 `case konstanta: příkazy`
 `case konstanta: příkazy`
 `...`
 `default:nep příkazy`
 `}`

Před příkazem `switch` můžeme zapsat návěstí (identifikátor) oddělené od příkazu dvojčekou. Tím uvedený příkaz pojmenujeme, což se nám bude hodit v souvislosti s příkazem `break`, o kterém si povíme dále.

Příkaz `switch` začíná klíčovým slovem `switch`, za kterým v závorkách následuje celočíselný výraz (typu `int`); v Javě 5 to může být i hodnota výčtového typu. Pak následuje blok („tělo příkazu `switch`“), ve kterém jsou některé příkazy označeny jedním nebo několika návěstími tvaru `case konstanta:`. Jeden z příkazů také může mít návěstí `default:`. Konstanty musí být celočíselné (typu `int`) nebo znakové; v Javě 5 to mohou být i výčtové konstanty.

Při zpracovávání příkazu `switch` se nejprve vyhodnotí výraz. Pak se vyhledá návěstí `case`, v němž je uvedena konstanta se stejnou hodnotou jako má výraz, program přejde na příkaz, který za ním následuje, a provede všechny další příkazy do konce těla příkazu `switch`.

Pokud program odpovídající návěstí nenajde, přejde na příkaz označený návěstí `default` (je-li uvedeno), provede jej a všechny příkazy, které za ním následují, až do konce těla příkazu `switch`.

Jestliže program odpovídající návěstí nenajde a tělo příkazu `switch` neobsahuje návěstí `default`, nestane se nic – tělo příkazu `switch` se prostě přeskočí.

V jednom návěstí `case` smíme uvést jen jednu konstantu, ale před jedním příkazem může být více návěstí. Žádná konstanta se v návěstích v jednom příkazu `switch` nesmí opakovat. Návěstí `default` nemusí být poslední, i když se jako poslední obvykle zapisuje.

Skupiny příkazů, které začínají jednotlivými návěstími, označujeme občas jako „alternativy“.

Příklad: Podívejme se na jednoduchý příklad, který ukáže, jak příkaz `switch` funguje. Napíšeme program, který si vyžádá malé celé číslo. Zadáme-li 1 nebo 5, vypíše Ahoj, zadáme-li 2 nebo 6, vypíše ře pic, pokud zadáme 3 nebo 7, vypíše Cau. V ostatních případech vypíše upozornění, že nerozumí.



Náš první pokus nesplní očekávání:

```
System.out.print("Zadej male cele cislo: ");
int i = MojelO.nextInt();
switch(i)
{
    case 1:
    case 5:
        System.out.println("Ahoj");
    case 2:
    case 3:
    case 4:
    case 6:
    case 7:
        System.out.println("Cau");
    default:
        System.out.println("Nerozumi");
```

```

        case 6:
            System.out.println("Te pic");
        case 3:
        case 7:
            System.out.println("Cau");
        default:
            System.out.println("Nerozumim");
    }

```

Jestliže po výzvě Zadej male cele cislo napišeme např. ?, vypíše program

```

Te pic
Cau
Nerozumim

```



Program nejprve vyhledal příkaz s návěstím `case ?:`, provedl příkaz za ním a pak pokračoval příkazy, které za ním následují. Vyspal tedy i všechny další zprávy – ale to jsme nechtěli.⁴⁰ (Úplný program v této podobě najdete na WWW v souboru Kap08\01\Switch1.java.) ♦

Příkaz break

Předchozí příklad ukazuje, že většinou budeme potřebovat předčasně ukončit provádění těla příkazu `switch`. K tomu slouží v Javě příkaz `break`, jehož syntaxe je velice jednoduchá:

příkaz break:

- `break ;`
- `break návěští ;`

Středník je součástí tohoto příkazu.

Příkaz `break` v těle příkazu `switch` způsobí, že program ukončí provádění těla příkazu `switch` a přejde na první příkaz za ním. Varianta s návěstím se hodí v případě příkazu `switch`, který je „vnořen“ do jiného příkazu `switch` (je součástí jeho těla). Příkaz `break návěští`; pak znamená, že chceme opustit příkaz `switch`, který je označen tímto návěstím.

Příklad: Upravme tedy předchozí příklad tak, aby vypisoval vždy jen jediný pozdrav:

```

System.out.print("Zadej male cele cislo: ");
int i = MojeIO.readInt();
switch(i)
{
    case 1:

```

⁴⁰ Chová se tedy jinak než např. příkaz `case` v Pascalu.

```

        case 5:
            System.out.println("Ahoj");
            break;
        case 2:
        case 6:
            System.out.println("Te pic");
            break;
        case 3:
        case 7:
            System.out.println("Cau");
            break;
        default:
            System.out.println("Nerozumim");
            break;
    }
}

```

Jestliže po nyní výzvě Zadej male cele cislo napíšeme opět 2, odpoví nám program

```

le pic
jak jsme si původně představovali.
```

Všimněte si, že jsme příkaz `break` zapsali i za příkazy za návštěním `default`, i když je tam zbytečný. Nezpůsobí žádnou škodu, a pokud později připíšeme do příkazu `switch` ještě nějaké další alternativy, ušetří nám možná pracné hledání chyby, kdybychom na něj zapomněli.



(Takto upravený program najdete na WWW v souboru Kap08\01\Switch2.java.) ◊

8.4 Cykly

Další důležitou skupinu tvoří příkazy, které umožňují předepsat opakování jednoho příkazu. Označujeme je jako *příkazy cyklu, smyčky* nebo *iterační příkazy*.

Příkaz, který se má opakovat, označujeme jako *tělo cyklu*. Opakování těla cyklu závisí na hodnotě logického výrazu, který nazýváme *podmínka opakování*. Jednotlivé průchody tělem cyklu označujeme také jako *iterace*.

Jazyk Java nabízí tři příkazy cyklu, a to `while`, `do-while` a `for`.

Návštěti Před kterýkoli z příkazů cyklu můžeme umístit návštětí, tj. identifikátor následovaný dvojtečkou. Např. takto:

Vnejsí: `while(x != 0)/* ...něco dělej ... */`

Tím cyklus pojmenujeme. Význam tohoto návštětí poznáme v souvislosti s příkazy `break` a `continue`.

Příkaz while

Tento příkaz vlastně už známe. Jeho syntax je

příkaz while:

- **while(*podminka*) příkaz**

Syntaktická pravidla Javy předepisují, že tělo cyklu `while` tvoří jediný příkaz; chceme-li jich tam zapsat více, použijeme blok. *Podminka* je logický (booleovský) výraz a závorky kolem ní jsou nezbytné.

Při provádění tohoto příkazu se nejprve vyhodnotí *podminka*. Není-li splněna (má-li hodnotu `false`), tělo cyklu se přeskočí a provádění příkazu `while` tím končí. Je-li splněna (má-li hodnotu `true`), provede se *příkaz* a znova se vyhodnotí podmínka. Není-li splněna, tělo cyklu se přeskočí a provádění příkazu `while` tím končí. Je-li splněna, provede se tělo cyklu a znova se vyhodnotí *podminka* atd.

Tělo příkazu `while` se nemusí provést ani jednou, neboť podmínka opakování se vyhodnocuje *před* průchodem tělem cyklu.

Příkaz do-while

Tento příkaz je velice podobný příkazu `while`, liší se jen tím, že se podmínka opakování vyhodnocuje až po průchodu tělem cyklu, takže se tělo provede vždy alespoň jednou. Jeho syntax je

příkaz do-while:

- **do příkaz while(*podminka*);**

Pro *podminku* opakování platí totéž co pro podmínu u příkazu `while`. Středník za tímto příkazem je nezbytný. Význam tohoto příkazu je:

Nejprve se provede *příkaz* představující tělo cyklu. Pak se vyhodnotí *podminka*. Není-li splněna, provádění příkazu `do-while` končí a program pokračuje prvním příkazem následujícím za ním. Je-li *podminka* splněna, provede se znova tělo cyklu, opět se vyhodnotí *podminka* atd.



Cyklus `do-while` skončí, jestliže podmínka *není* splněna. Tím se liší od jinak velice podobného cyklu *repeat-until* známého z jazyka Pascal. Tato podobnost je bohužel častým zdrojem chyb.

Příklad:

Jednou z univerzálních matematických konstant je Eulerovo číslo e, známé především jako základ tzv. přirozených logaritmů; ve skutečnosti na ně lze v matematické analýze narazit skoro všude. Napíšeme program, který ho vypočte se zadanou přesností.

Švýcarský matematik Leonhard Euler již před více než čtvrt tisíciletím zjistil, že toto číslo lze vypočítat jako nekonečný součet

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!} + \dots \quad (1)$$

kde $n! = 1 \cdot 2 \cdot \dots \cdot n$ je faktoriál čísla n . Je jasné, že nemůžeme sečítat nekonečně mnoho sčítanců. Z matematiky je ale známo, že sečteme-li pouze prvních k sčítanců, bude chyba, které se tím zde dopustíme, v tomto případě úměrná poslednímu sčítanci, tj. $\frac{1}{k!}$.

Napišeme metodu `double pocitej(double eps)`, která dostane jako parametr `eps` přesnost, s níž chceme e vypočítat, a vrátí hodnotu, která se od e nebude lišit o více než `eps`. Bude v cyklu sčítat členy řady (1), dokud nepříčte sčítanec, který bude menší než `eps`. Pak skončí. To znamená, že chceme nejprve přičíst sčítanec a pak teprve testovat, zda máme skončit – použijeme tedy cyklus `do-while`.

Na první pohled by se mohlo zdát, že pro výpočet sčítanců je rozumné použít metodu `faktorial()`, kterou jsme naprogramovali ve 3. kapitole. Není to však pravda: Počítali bychom naprostě zbytečně stále stejně součiny. Jestliže totiž ve třetím sčítanci vypočteme $\frac{1}{1 \cdot 2 \cdot 3}$, pak ve čtvrtém budeme počítat $\frac{1}{1 \cdot 2 \cdot 3 \cdot 4}$, v pátém $\frac{1}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5}$ atd. Snadno zjistíme, že n -tý sčítanec v (1) můžeme vypočítat, když vezmeme předchozí sčítanec a vydělíme ho n .

Následující výpis ukazuje, jak lze metodu `pocitej()` naprogramovat.

```
double pocitej(double eps)
{
    double s = 2, d = 1;      // s je součet, d je sčítanec
    int i = 2;                // Počítá, kolikátý sčítanec
    do {
        d /= i++;            // Vypočti další sčítanec
        s += d;               // a přičti ho k součtu
    } while(d > eps);       // Je-li menší než eps, hotovo
    return s;
}
```

V proměnné `s` budeme počítat součet (1); proměnná `d` bude postupně obsahovat jednotlivé sčítance. První dva sčítance jsou rovny 1; ty sečteme sami, takže jako počáteční hodnotu do `s` uložíme 2 a do `d` hodnotu 1. Proměnná `i` bude sloužit jako „počítadlo“, kolikátý sčítanec právě počítáme.

V cyklu nejprve vypočteme novou hodnotu `d` (při prvním průchodu cyklem to bude $\frac{1}{2}$) a pak zvýšíme hodnotu `i` o jedničku. Pak sčítanec přičteme k celkovému součtu. Nakonec otestujeme, zda nebyl už menší než `eps`, a pokud ano, skončíme.



Zavoláme-li tuto metodu s parametrem `1e-5`, vrátí `2.7182815...`. Necháme-li si vypsat hodnotu konstanty `Math.E`, dozvím se, že správný výsledek s přesností na 15 desetinných míst je `2.718281828459045`. Program, který tuto metodu obsahuje, najdeme na WWW v souboru `Kap08\02\Euler.java`. ♦

Cyklus for

Poslední z cyklů, které Java nabízí, je příkaz `for`. Jeho syntax je

příkaz `for`:

- `for(inicializacenep ; podminkanep ; aktualizacenep)` příkaz

kde `index` _{nep} naznačuje, že *inicializaci*, *podmítku* i *aktualizaci* lze vynechat (středníky ale musíme uvést). Příkaz `for` znamená totéž co posloupnost příkazů

```
inicializace;
while(podminka) {
    příkaz           // Tělo cyklu
    aktualizace;
}
```

Pro *podminku* platí totéž co v případě cyklu `while`. *Inicializace* je buď výrazový příkaz nebo deklarace lokální proměnné s inicializací (příp. deklarace několika lokálních proměnných stejného typu). Zpravidla definuje hodnoty proměnných, které budou řídit (v *podmince*) opakování cyklu. *Aktualizace* je opět výrazový příkaz, který typicky upravuje hodnoty proměnných, které řídí počet opakování cyklu.

Význam tohoto příkazu: Nejprve se provede *inicializace*. Pak se otestuje, zda je splněna *podminka* opakování. Pokud ne, provádění příkazu `for` skončí. Pokud *podminka* je splněna, provede se tělo cyklu a *aktualizace* a znova se vyhodnotí *podminka* opakování atd.

V *inicializaci* a v *aktualizaci* můžeme zapsat i skupinu výrazů oddělených čárkami.

Vynecháme-li *podminku*, dosadí si překladač `true`. To znamená, že příkaz

```
for(;;) { /* ... */ }
```

představuje nekonečný cyklus. Jeho opakování musíme ukončit jinak – např. pomocí příkazu `break` nebo `return` v jeho těle.

Příklad: Chceme-li vypsat všechny prvky pole `A`, použijeme příkaz `for`:

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

Podobně použijeme příkaz `for` i při jakémkoli jiném zpracování všech prvků jednorozměrného pole. Při zpracování vícerozměrných polí použijeme zpravidla několik cyklů `for` vnořených do sebe.



Na uvedený zápis se můžeme dívat jako na idiom – vžitou frázi .. starších verzí jazyka Java pro průchod celým polem. V Javě 5 používáme poněkud jiný tvar příkazu `for`, o kterém si povíme dále.

Příklad: Ve 3. kapitole jsme metodu, která počítala faktoriál přirozeného čísla, napsali pomocí cyklu `while`. Pomocí cyklu `for` bude zápis o něco přehlednější:

```
public static int faktorial(int n)
{
    int s = 1;      // Promenná s bude obsahovat výsledek
    for(int i = n; i > 0; i--)
        s *= i;
    return s;
}
```

Proměnná `s` obsahuje stejně jako předtím průběžně počítaný součin.



Podívejme se, jak bude tento cyklus probíhat. Nejprve se (jako součást inicializace) vytvoří proměnná `i`, která zde bude sloužit jako *parametr cyklu*. Tato proměnná dostane počáteční hodnotu `n`. Pak se ověří, zda je splněna podmínka opakování, tj. zda je `i > 0`. Pokud ano, proběhne tělo cyklu, tj. příkaz `s *= i;`, který způsobí, že se hodnota, uložená v `s`, nahradí hodnotou `s*i`. Pak proběhne aktualizace, v našem případě příkaz `i--`, který zmenší hodnotu uloženou v `i` o 1. Dále se ověří, zda je splněna podmínka opakování atd. Úplný text tohoto příkladu najdete na WWW v souboru Kap08\03\Fakt.java. ♦



Proměnné, deklarované v inicializační části cyklu `for`, jsou k dispozici jen v tomto příkazu (v podmínce, aktualizaci a v těle). To znamená, že jsme nemohli napsat

```
public static int faktorial(int n)
{
    for(int i = n, s = 1; i > 0; i--)
        s *= i;
    return s;           // CHYBA - neznámý identifikátor s
}
```

neboť v příkazu `return` by proměnná `s`, deklarovaná v inicializační části příkazu `for`, už nebyla k dispozici (tentotéž příkaz už není součástí těla cyklu).

Příkaz for v Javě 5

5

Chceme-li zpracovat všechny prvky pole nebo některého ze standardních kontejnerů, můžeme v Javě 5 použít také novou podobu příkazu *for*, kterou mnozí programátoři označují jako příkaz *for-each*.⁴¹

příkaz *for-each*:

- **for(deklarace : kontejner) příkaz**

Kontejner je výraz, který představuje odkaz na instanci některé ze tříd kolekcí deklarovaných v balíku `java.util` nebo pole. *Deklarace* představuje deklaraci proměnné, do níž se postupně uloží jednotlivé hodnoty prvků z *kontejneru*. Tato deklarace nesmí obsahovat inicializaci a typ deklarované proměnné musí odpovídat typu prvků uložených v *kontejneru*.

Příklad: Napíšeme metodu, která vypíše všechny prvky pole typu `int`, které ji předáme jako parametr.

```
public static void vypisPole(int[] a)
{
    for(int x: a)
        System.out.println(x);
}
```

V příkazu *for* zde deklarujeme proměnnou *x*, do níž se postupně uloží jednotlivé prvky pole *a*; ty se v těle cyklu vypíší.

Podobně můžeme napsat metodu, která vypíše všechny prvky kolekce `ArrayList`:

```
public static void vypisKontejneru(ArrayList a)
{
    for(Object x: a)
        System.out.println(x);
}
```

I když jsme v Javě 5, použili jsme neparametrizovanou kolekci `ArrayList`; to lze, překladač ovšem upozorní, že bychom raději měli použít parametrizovanou verzi, neboť je to bezpečnější. O tom ale budeme hovořit později.



Neparametrizované kolekce v Javě obsahují pouze odkazu na `Object`. Proto jsme museli použít proměnnou *x* typu `Object`.

Zdrojový text třídy, implementující obě uvedené metody, najdete na WWW v souboru `Kap08\05\Foreach.java`. ♦

⁴¹ *For each* znamená anglicky „pro každý“. Podobný příkaz v některých jiných programovacích jazycích bývá vyjádřen klíčovým slovem *foreach*; tvůrci Javy 5 však nechtěli zavádět nové klíčové slovo, a proto použili *for*.

Poznámka Jako kontejner lze ve skutečnosti použít instanci jakékoli třídy, která implementuje rozhraní `java.lang.Iterable<T>`. Co to znamená, to si povíme později, v kapitole o rozhraních.

Příkaz break

Občas potřebujeme „vyskočit“ z cyklu, tedy předčasně ukončit jeho provádění. V tom případě můžeme použít příkaz `break`, se kterým jsme se setkali už v oddílu o příkazu `switch`. I zde ho můžeme použít ve dvou variantách:

příkaz break:

- `break ;`
- `break návěští ;`

První varianta, příkaz `break bez návěští`, způsobí okamžité ukončení cyklu, v jehož těle se nachází. (V případě do sebe vnořených cyklů způsobí ukončení nejvnitřejšího cyklu, v jehož těle je.)

Druhá varianta, příkaz `break obsahující návěští`, způsobí ukončení cyklu označeného tímto *návěstím*.

Příklad: Máme pole řetězců `s` a chceme zjistit, zda se v něm vyskytuje znak `c`. Budeme předpokládat, že v instanci jsou dvě proměnné `i` a `j` typu `int`, které použijeme k vyhledávání. Metoda `najdi()` může vypadat takto:

```
public boolean najdi(char c, String[] s)
{
    Ven:
        for(i = 0; i < s.length; i++)
            for(j = 0; j < s[i].length(); j++)
            {
                if(s[i].charAt(j) == c) break Ven;
            }
        if(i == s.length) return false;
        else return true;
}
```

Tělo metody začíná dvěma do sebe vnořenými cykly `for`, z nichž vnější (označený návěstí `Ven`) prochází jednotlivé prvky pole řetězců `s`, tedy jednotlivé řetězce, a vnitřní prochází jednotlivé znaky v řetězci `s[i]`. Jakmile najdeme hledaný znak, vyskočíme z obou cyklů zároveň pomocí příkazu `break Ven`: a přejdeme na příkaz `if`. Ten podle hodnoty proměnné `i` určí, zda jsme hledaný znak našli. Pokud hledaný znak v žádném z řetězců není, bude `i` obsahovat `s.length`. Jestliže tam hledaný znak je, bude `i` obsahovat index řetězce, ve kterém jsme ho našli, a `j` bude obsahovat index hledaného znaku v tomto řetězci.

Budeme-li např. hledat znak 'u' pole řetězců `s`,

```
String[] s = {"ahoj", "lidi", "to", "je", "hruza"};
uloží tato metoda do proměnných i a j hodnoty 4 a 2, neboť pole i
řetězce jsou indexovány od 0.
```

Tuto metodu najdete na WWW v souboru Kap08\04\Break.java. ♦



Příkaz continue

Tento příkaz umožnuje přeskočit zbytek těla cyklu a přejít k další iteraci. Existuje ve dvou variantách, s návěštím a bez něj.

příkaz continue:

- **continue;**
- **continue návěští;**

Středník je povinnou součástí příkazu `continue`. První varianta, příkaz `continue` bez návěští, způsobí přeskočení zbytku těla cyklu, v němž se nachází. (V případě do sebe vnořených cyklů způsobí přeskočení zbytku těla nejvnitřnějšího cyklu, v němž je.)

Druhá varianta, příkaz `continue` obsahující návěští, způsobí přeskočení zbytku těla cyklu označeného tímto návěštím.

Příklad: Vezmeme opět pole znakových řetězců. Chceme vypsat všechny řetězce, které toto pole obsahuje, s výjimkou řetězců "necenzurovaný". Metoda `cenzura()`, která se o to postará, může vypadat např. takto:

```
public void cenzura()
{
    for(int i = 0; i < st.length; i++)
    {
        if(st[i].equals("necenzurovaný")) continue;
        System.out.print(st[i] + " ");
    }
}
```

K porovnávání jednotlivých řetězců používáme metodu `equals()`. Najdeme-li zakázaný řetězec, přeskočíme zbytek těla cyklu a začneme zpracovávat další. Bude-li tato metoda zpracovávat pole

```
String[] st= {"Toto", "je", "necenzurovaný", "řetězec"};
```

vypíše

```
Toto je řetězec
```

Kdybychom místo příkazu `continue` použili `break`, dostali bychom jen

```
Toto je
```



Program s touto metodou najdete na WWW v souboru Kap08\04\Continue.java. ♦

8.5 Přenos řízení

Tímto poněkud tajemně znějícím názvem se označují příkazy, které způsobí, že počítač přeruší přirozenou posloupnost příkazů, přestane je plnit v pořadí, v jakém jsou v programu zapsány, a přejde na jiné místo programu. Do této skupiny příkazů tradičně patří příkaz skoku, který ale Java neobsahuje (i když `goto` je v seznamu rezervovaných slov). Dále sem patří `break` a `continue`, které jsme probrali v oddilech věnovaných příkazům cyklu a příkazu `switch`, a příkaz `return`.

Poznamenejme, že přenos řízení může také být způsoben vznikem výjimky. O tom budeme hovořit později v kapitole 10.

Příkaz `return`

Syntax tohoto příkazu je

příkaz return:

- `return ;`
- `return výraz ;`

Středník je povinnou součástí tohoto příkazu. První varianta (bez *výrazu*) se smí objevit pouze v metodě typu `void`. Znamená „zde končí provádění těla metody, vrat' se na místo, odkud byla zavolána“. Přejde-li program přes složenou závorku i uzavírající tělo metody, znamená to totéž co provedení příkazu `return ;`.

Druhá varianta se smí objevit pouze v metodě jiného typu než `void`. Znamená „zde končí provádění těla procedury, vrat' se na místo, odkud byla zavolána, a jako výsledek tam přenes hodnotu, kterou představuje *výraz*“. Typ *výrazu* musí být stejný jako typ vracený metodou nebo ho musí být možné na tento typ převést rozšiřující konverzí. (To např. znamená, že v metodě s deklarovaným typem `int` musí být *výraz* typu `int` nebo typu `byte`, `short` nebo `char`. Nemůže být např. typu `long`, nebo třeba `String`.)

8.6 Aserce (příkaz `assert`)

4, 5

Při ladění programu často potřebujeme ověřit, zda jsou v určitých místech programu splněny nějaké podmínky. Představte si např., že píšeme program, ve kterém používáme metodu `faktorial()`, jež po-

čítá faktoriál; poprvé jsme se s ní setkali v kapitole 3.4. Jak víme, faktoriál je definován pouze pro nezáporná celá čísla.

Jsme si zcela jisti, že pokud je nás program správně, nemůže se stát, že bychom této metodě předali záporný parametr. Jenže: můžeme si být jisti, že je nás program správně? Ze nemůže zavolat metodu fakt() se záporným parametrem?

Dokud svůj program neodladíme, jistí si být nemůžeme, a proto je při ladění třeba v metodě faktorial() testovat, zda je daná *vstupní podminka* splněna. K tomu účelu nabízí Java počinaje JDK 1.4 příkaz assert (aserci). Jeho tvar je

příkaz assert:

- assert výraz_1;
- assert výraz_1 : výraz_2 ;

Výraz_1 představuje testovanou podmínu. Je-li splněna, nic se nestane a program pokračuje následujícím příkazem. Není-li splněna, vznikne výjimka typu java.lang.AssertionError. Výjimky tohoto typu není třeba deklarovat v hlavičkách metod a nemusíme je nijak ošetřovat; vznikne-li takováto výjimka, ukončí JVM program a vypíše chybové hlášení, z něhož určíme, kde k chybě došlo.

Za výrazem_1 může následovat dvojtečka a za ní výraz_2. Výraz_2 je jakýkoli výraz, který lze převést na znakový řetězec. Použijeme-li tuto možnost a nebude-li splněna podmínka představovaná výrazem_1, vypíše se kromě chybového hlášení i řetězec představovaný výrazem_2.

Příklad:

Napišeme ještě jednou metodu faktorial() a ověříme v ní, zda je splněna podmínka kladená na skutečný parametr. Nejprve si ale definujeme pomocnou třídu Pomoc, která bude obsahovat příznak, zda program ladíme nebo zda už je v „ostrém“ provozu:

```
public class Pomoc
{
    public static final boolean LADIM = true;
}
```

Vlastní definice metody faktorial() bude jednoduchá:

```
public static int faktorial(int n)
{
    if (Pomoc.LADIM) // Určuje, zda se použije aserce
        assert n >= 0 : "Zaporný parametr faktorialu: " + n;
    int s = 1;          // Vlastní výpočet se nezměnil
    for(int i = n; i > 0; i--) s *= i;
    return s;
}
```

Jestliže tuto metodu zavoláme příkazem

```
int x = faktorial(-1);
```

program skončí chybovým hlášením

```
Exception in thread "main" java.lang.AssertionError:  
Zaporný parametr faktoriału: -1  
at Assert.faktorial(Assert.java:19)  
at Assert.main(Assert.java:30)
```

Změníme-li později v deklaraci třídy pomoc hodnotu konstanty LADIM na false, odstraníme tím uvedenou aserci z programu. (To má samozřejmě význam především v případě, že náš program obsahuje větší počet asercí.) ♦

Poznámky

Invarianty

Příkaz assert používáme k testování vstupních podmínek metod – jako např. že číslo, jehož faktoriál chceme počítat, je nezáporné. (Tím ověřujeme, zda je metoda správně použita.)

Podobně můžeme testovat i hodnotu, kterou metoda vraci. Například víme, že faktoriál celého nezáporného čísla nemůže být záporný nebo nula. Před příkaz return v metodě faktorial() bychom proto mohli umístit příkaz assert, který bude tuto podmínu ověřovat. Ukáže-li se, že tato podmínka není pro nějaké hodnoty vstupních parametrů splněna, znamená to, že jsme na něco zapomněli – třeba na to, že pro větší hodnoty parametrů může dojít k celočíselnému přetečení a výsledek nebude mít smysl.

Příkaz assert můžeme také použít k testování tzv. invariantů programu – můžeme např. ověřovat, zda trojúhelník má tři strany nebo zda součet úhlů v něm je roven 180° .

I když to vypadá podivně, je testování invariantů součástí běžné programátorské praxe. Jejich porušení je totiž vždy příznakem nějaké chyby v programu – a čím dříve chybu zjistíme, tím snáze ji opravíme.

Překlad v JDK 1.4

Pracujeme-li s JDK 1.4, musíte k překladu použít příkaz

```
javac -source 1.4 Assert.java
```

Nepoužijeme-li volbu -source 1.4, bude překladač s klíčovým slovem assert zacházet jako s identifikátorem a ohlási podivné chyby.

V JDK 5 je assert automaticky chápáno jako klíčové slovo, takže uvedenou volbu lze vynechat.

Běh Při spuštění musíme jak v JDK 1.4, tak i v JDK 5 použít aserci povolit, a to pomocí volby `-enableassertions` nebo `-ea`. To znamená, že program spusťte příkazem

```
java -enableassertions Assert
```

Podrobnější informace lze najít v návodě.

Optimalizace Konstanta `POMOC.LADIM`, kterou používáme v podmínce příkazu `if` určujícího, zda se má použít aserce, je deklarována s modifikátory `static` a `final`. Má-li hodnotu `false`, nemůže být příkaz, který tvoří tělo příkazu `if`, nikdy proveden, a proto ho překladač bude ignorovat. Proto se v bajtovém kódu, který bude výsledkem překladu třídy `Assert`, tento příkaz (a ani příkaz `assert`) vůbec neobjeví.



Spustitelný příklad obsahující třídu `Pomoc` a metodu `faktorial()` z tohoto oddílu najdete na WWW v souborech `Kap08\06\Assert.java` a `Kap08\06\Pomoc.java`.

9 Třídy a objekty

V předchozích kapitolách jsme poznali výrazy, příkazy, datové typy a podobné konstrukce, které představují základní stavební kameny programování ve většině objektových i neobjektových jazyků. Nyní se konečně dostáváme ke třídám, které jsou základem objektově orientovaného programování. O významu potřebných pojmu jsme obecně hovořili v kapitole 1.5 a s některými možnostmi jejich použití v Javě jsme se seznámili v následujících kapitolách; zde si tyto vědomosti doplníme a upřesníme.

9.1 Deklarace třídy

Deklaraci třídy lze popsát takto⁴²:

deklarace třídy:

- *modifikátory_{nep} class identifikátor předek_{nep} rozhraní_{nep}*
 {
 tělo_třídy
 }

Klíčové slovo `class` říká, že jde o deklaraci třídy, a *identifikátor* bude její jméno.⁴³

Za identifikátorem může následovat specifikace předka, v syntaktickém popisu vyjádřená slovem *předek*. Ta nám umožňuje definovat naší třídu jako potomka jiné třídy. Pak může následovat specifikace rozhraní, které tato třída implementuje; v popisu je vyjádřena slovem *rozhraní*.

Dále následuje ve složených závorkách *tělo třídy*, tj. deklarace datových složek a metod.⁴⁴

Před klíčovým slovem `class` mohou stát modifikátory `public` a `abstract` nebo `final`. Uvedeme-li jich více, musí být první `public` a za ním `abstract` nebo `final`.

⁴² Také tento popis je zjednodušený.

⁴³ V jistých situacích můžeme identifikátor spolu s klíčovým slovem `class` vynechat; tak vznikne tzv. anonymní třída. Výklad o nich však přesahuje rámec naší knihy.

⁴⁴ Počínaje JDK 1.1 může tělo třídy obsahovat také deklaraci jiné třídy. Vnořenými třídami se však také nebudeme zabývat.

Modifikátory v deklaraci třídy

`public`

Zapíšeme-li před klíčové slovo `class` modifikátor `public`, říkáme tím, že jde o veřejně přístupnou třídu. To znamená, že ji může používat deklarovat její instance, volat její metody atd. – kdokoli ve kterémkoliv části programu.⁴⁵ Jestliže klíčové slovo `public` neuvedeme, budeme tuto třídu moci používat pouze v rámci aktuálního balíku.⁴⁶



V každém souboru smí být jen jedna třída se specifikací `public`. Soubor, ve kterém ji deklarujieme, se musí jmenovat stejně jako tato třída (a musí mít příponu `.java`).



Třídy bez specifikace `public` jsou většinou pomocné. Jejich služeb využívají veřejně přístupné třídy z aktuálního balíku, ale nikdo jiný k nim nemá přístup. Je to jeden ze způsobů ukrývání implementace: Protože je nemůže používat nikdo kromě tříd v balíku, ve kterém jsou deklarovány, nebude nás jejich změna nutit nijak měnit ostatní části programu (možná až na aktuální balík). V malých programech to příliš neoceníme, ale v rozsáhlých projektech nám rozlišování veřejně přístupných a nepřístupných tříd může opravdu usnadnit život.

`abstract`

Druhým z modifikátorů, které se mohou objevit před klíčovým slovem `class`, je `abstract`. Označuje abstraktní třídu, tj. třídu, od níž nelze vytvářet instance. Abstraktní třída typicky shrnuje společné vlastnosti svých potomků.

`final`

Klíčové slovo `final` je v jistém smyslu opakem klíčového slova `abstract`. Znamená „konečnou“ třídu, tedy třídu, od níž nelze odvozovat potomky. Abstraktní třída nemůže být konečná, z těchto dvou klíčových slov můžeme uvést vždy jen jedno.

**Specifikace
předka**

Specifikace předka má tvar

předek:

- **extends jméno_bázové_třídy**

kde `jméno_bázové_třídy` je identifikátor třídy, která bude předkem třídy právě deklarované. Je-li to nutné, můžeme (a musíme) k identifikátoru bázové třídy připojit i jméno balíku. Pokud tuto část vynechá-

⁴⁵ Pokud to dovolí přístupová práva ke složkám, které chceme použít; o tom budeme hovořit dále.

⁴⁶ Připomeňme si, že třídy, u kterých neuvedeme balík, leží v tzv. implicitním balíku (všechny v jednom). To znamená, že třídy bez specifikace `public` a bez specifikace balíku jsou dostupné ze všech souborů, které neobsahují specifikaci balíku – jsou tedy téměř veřejně přístupné.

me, doplní si překladač jako předka třídu `Object`, která je společným předkem všech jazických tříd.



Java připouští pouze jednoduchou dědičnost. To znamená, že každá třída může mít pouze jednoho přímého předka. Jako předka (bázovou třídu) nemůžeme použít právě deklarovanou třídu ani žádného z potomků právě deklarované třídy. (Třída nemůže být svým vlastním předkem, a to ani nepřímým.) Pokusíme-li se o to, ohlási překladač chybu „cyclic inheritance“, tedy cyklická dědičnost.

Implementace rozhraní

Specifikace rozhraní, které třída implementuje, má tvar *rozhraní*:

- **implements *seznam_rozhrani***

kde *seznam_rozhrani* jsou jména implementovaných rozhraní. Je-li jich více, oddělíme je čárkou. Rozhraními a jejich implementací se budeme zabývat v kapitole 11. (Nevedeme-li tuto část, znamená to, že naše třída neimplementuje žádné rozhraní. Takové byly všechny naše dosavadní třídy.)

Příklad:



V první kapitole, v oddílu 1.5, jsme hovořili o grafických objektech a navrhli jsme mimojiné třídu `Bod`, která měla za úkol reprezentovat bod na obrazovce. Nečíslovaný obrázek po straně ukazuje její ikonu v UML. Deklarace této třídy v Javě by mohla vypadat takto:

```
public class Bod {  
    // Tělo třídy Bod  
}
```

Třída `Bod` je veřejně přístupná. To znamená, že musí být v souboru `Bod.java` a v tomto souboru už žádná další veřejně přístupná třída být nesmí. Tato třída není abstraktní (v její deklaraci není modifikátor `abstract`) a je přímým potomkem třídy `Object`, neboť jsme neuvedli žádného předka. Neimplementuje také žádné rozhraní. ♦

9.2 Tělo třídy

Tělo třídy je uzavřeno mezi složené závorky. Obsahuje deklarace datových složek, metod atd. Pro každou ze složek třídy, tj. pro datové složky i pro metody, musíme určit tzv. přístupová práva. Tím předepisujeme, kdo bude smět tyto složky používat.

Přístupová práva

Java rozlišuje 4 stupně přístupových práv ke složkám třídy. Tři z nich jsou vymezeny klíčovými slovy (modifikátory) `public`, `protected`

`a private`. Čtvrtý zadáme tím, že nepoužijeme žádný z uvedených modifikátorů. Podívejme se na význam jednotlivých možností.

<code>public</code>	Složky s tímto modifikátorem označujeme jako <i>veřejné</i> nebo <i>veřejně přístupné</i> . Toto klicové slovo nastavuje vůbec nejširší možná přístupová práva: veřejně přístupnou složku můžeme použít kdekoli – ve kterékoli metodě ve kterékoli části programu. Jako veřejné obvykle deklarujeme některé metody a některé konstanty, jen zřídka proměnné. <i>Veřejně přístupné složky tvoří uživatelské rozhraní třídy, tj. nástroje, které dává tato třída k dispozici uživatelům – programátorům, kteří ji budou ve svých programech používat.</i>
<code>protected</code>	Složky s tímto modifikátorem obvykle označujeme jako <i>chráněné</i> . Můžeme je používat v metodách třídy, do které patří, v metodách jejich potomků a ve všech třídách téhož balíku. (Odvozené třídy – potomci – nemusí ležet v témže balíku jako bázová třída.) <i>Chráněné složky představují spolu s veřejnými složkami rozhraní pro odvozování potomků</i> . Občas totiž potřebujeme, aby potomek mohl manipulovat se složkami, které jsou ostatním částem programu nedostupné.
<code>neuvedeno</code>	Přístupová práva pro složky, u nichž není žádný modifikátor přístupových práv, označujeme obvykle jako <i>přátelská</i> (friendly). Složky s přátelským přístupem smíme používat jen v témže balíku. <i>Přátelské složky představují rozhraní pro podpůrné třídy z téhož balíku.</i>
<code>private</code>	Tyto složky označujeme jako soukromé. Smíme je používat jen v metodách třídy, k níž patří. Jako soukromé zpravidla deklarujeme datové složky, především proměnné. <i>Soukromé složky představují „implementační detaily“.</i>

Stupeň	Táž třída	Týž balík	Potomci	Celý svět
<code>public</code>	ano	ano	ano	ano
<code>protected</code>	ano	ano	ano	ne
<code>přátelský</code>	ano	ano	jen v témže balíku	ne
<code>private</code>	ano	ne	ne	ne

*Tab. 9.1 Souhrn specifikací přístupových práv pro datové složky; **ano**, resp. **ne** znamená, zda je složka s odpovídajícím modifikátorem dostupná v téže třídě, témže balíku atd.*

Souhrn významu jednotlivých stupňů přístupových práv poskytuje tabulka 9.1. Koncepce přístupových práv je v Javě sice velice podobná jako v jazyce C++, liší se však v několika podstatných bodech:

- C +**
- „Jednotkou ochrany“ není jen třída, ale i balík.
 - Složky s modifikátorem `protected` lze používat i v rámci balíku.
 - Složky bez modifikátoru přístupových práv mají zvláštní stupeň ochrany.

9.3 Datové složky

Java rozlišuje dvě základní skupiny datových složek (atributů, proměnných) – statické a nestatické. Obojí mohou být dále deklarovány jako konstantní (`final`).

Nestatické datové zložky

To jsou „obyčejné“ datové složky, s jakými jsme se dosud setkávali. Jejich deklaraci lze zjednodušeně popsat takto:

deklarace nestatické datové složky:

- `přístupnep finalnep typ identifikátor inicializacenep;`

Typ je typ datové složky. Obsahuje-li deklarace specifikátor `final`, jde o konstantu (hodnota složky se nesmí měnit). *Identifikátor* je její identifikátor, *inicializace* má tvar

inicializace:

- `= výraz`

a předepisuje počáteční hodnotu datové složky. *Výraz* může předepisovat také inicializaci pole (viz 6.2).

Každá instance bude obsahovat své vlastní nestatické datové složky. Tyto složky typicky slouží k uchovávání údajů o stavu实例.

Inicializace

Jestliže v deklaraci datové složky uvedeme *inicializaci*, bude tato složka mít uvedenou hodnotu bezprostředně po vytvoření instance (ještě před voláním konstruktora). Pokud ji neuvedeme, bude jejich hodnota ihned po vytvoření instance záviset na typu.

Neinicializované atributy

Neinicializované datové složky číselných typů budou obsahovat hodnotu 0, znakové složky budou mít hodnotu '\u0000', logické datové složky `false` a odkazy na pole či objekty hodnotu `null`. Inicializace nestatických datových složek proběhne ještě před tělem konstruktora.

Příklad: třída Bod

Vraťme se ke třídě `Bod` a doplňme její deklaraci o datové složky. Jak souřadnice, tak i barva jsou údaje, které budou u různých bodů různé. Popíšeme je tedy pomocí nestatických datových složek typu `int`. (Poloha bodů na obrazovce monitoru a jejich barva se obvykle vyjádřuje pomocí celých čísel.) Tyto složky deklarujeme jako soukromé,

neboť jde o „implementační detail“: Zatím jsme se rozhodli ukládat „obrazovkové“ souřadnice (jednotkou je jeden pixel, tedy bod na obrazovce, a proto je vyjadřujeme pomocí proměnných typu `int`). Co když se ale později rozhodneme vyjadřovat souřadnice v centimetrech? Aby případná změna co nejméně zasáhla zbytek programu, je nutné způsob reprezentace bodu co nejvíce utajit před uživateli této třídy (tj. před programátory, kteří ji budou užívat - když na to přijde, i před námi samými). Podobně deklarujeme jako soukromou i složku reprezentující barvu bodu.

```
public class Bod {  
    private int x, y;  
    private int barva = 0xFFFFFFFF;  
    // Metody zatím ponecháme stranou  
}
```

Jestliže nyní vytvoříme dvě instance třídy `Bod`,

```
Bod a = new Bod();  
Bod b = new Bod();
```

bude mít každá instance své datové složky `x`, `y` a `barva`. U složek `x` a `y` jsme počáteční hodnotu nepředepsali, takže budou nulové; složka `barva` bude mít hodnotu `0xFFFFFFFF`.

Protože jsou tyto složky soukromé, mohou měnit jejich hodnotu pouze metody třídy `Bod`. Kdybychom např. v metodě `main()` třídy `Pokus` napsali

 `a.barva = 0; // V cizí metodě CHYBA - soukromá složka`
ohlásil by překladač chybu. ♦

Statické datové složky

O statických atributech (datových složkách) se občas hovoří jako o „datových složkách třídy“, neboť jde o data společná všem instancem – tedy *data, která jsou charakteristická pro třídu jako celek*. Toto označení je ale poměrně těžkopádné a může vést k nedorozuměním, a proto jim budeme říkat *statické* podle toho, že jejich deklarace obsahuje klíčové slovo `static`. Syntax deklarace statické datové složky je

deklarace statické datové složky:

- *přístup static final_{nep} typ identifikátor inicializace_{nep};*

Ve srovnání s deklarací nestatické složky je tu navíc klíčové slovo `static`; význam ostatních částí deklarace je stejný.

Už jsme si řekli, že statické datové složky jsou společné pro všechny instance. To znamená, že existují nezávisle na tom, kolik instance

dané třídy jsme v programu vytvořili; existují – a můžeme je tedy používat – i v případě, že jsme nevytvořili ani jednu instanci.

Použití V metodách třídy, v níž je statická složka deklarována, ji používáme bez kvalifikace. Budeme-li ji chtít použít mimo její třídu (pokud to dovolí přístupová práva), musíme ji kvalifikovat jménem třídy.

Příklad: π Ve třídě `Math` je definována mj. veřejně přístupná konstanta `PI` představující Ludolfovou číslo:

```
public static final double PI = 3.141592653589793;
```

Na tuto konstantu se odvoláváme zápisem `Math.PI`, ve kterém jméno konstanty kvalifikujeme jménem třídy (připojíme k němu tečkou jméno třídy). Tuto konstantu můžeme používat, i když nevytvoříme ani jednu instanci třídy `Math`. ♦

Inicializace statických atributů Pro inicializaci statických atributů platí podobná pravidla jako pro inicializaci nestatických atributů. Tato inicializace proběhne v době zavádění třídy do paměti, tj. zpravidla při spuštění programu nebo těsně před jejím prvním použitím.

Existují ovšem situace, kdy se nám obvyklý způsob inicializace statických atributů nehodí, např. proto, že ke zjištění jejich hodnot potřebujeme složitější výpočet. Pak můžeme použít tzv. *statický inicializátor*. Jeho syntax můžeme popsat

Statický inicializátor:
• `static{ příkazy }`

a zapíšeme ho v těle funkce za deklarací statické složky. Například takto:

```
static int[] k;           // Statické pole
static {                  // a statický inicializátor
    k = new int[10];
    for(int i = 0; i < 10; i++) k[i] = i*i;
}
```

9.4 Metody

Metody představují zprávy poslané instancím nebo třídám jako celku. Volání metody způsobí, že s instancí nebo s třídou jako celkem proběhnou nějaké operace. (To vlastně znamená, že metoda představuje naprogramovaný algoritmus této operace.)

Přetěžování Metody lze přetěžovat. To znamená, že jedna třída může obsahovat několik metod se stejným identifikátorem, pokud se liší počtem nebo typem parametrů (nebo obojím). K rozlišení přetížených metod nasta-

čí, když se liší typem vrácené hodnoty; stačí ale, když se liší pořadím parametrů s různým typem, neboť pak se vlastně liší typem parametrů na odpovídajících místech.

Deklarace metody

Základem deklarace metody je tzv. hlavička:

hlavička metody:

- *typ identifikátor(parametry)*

v níž *identifikátor* je identifikátor – jméno – deklarované metody. Pomocí tohoto jména ji budeme volat. *Typ* je typ jejího výsledku a *parametry* jsou formální parametry. Pokud metoda nemá žádné parametry, musíme zde zapsat prázdné závorky; o tom všem jsme již hovořili v úvodních kapitolách.

Typ Typ výsledku, tedy typ vrácené hodnoty, není nijak omezen. Metoda může vracet hodnoty primitivních typů, referenci na objekt nebo referenci na pole. Zápis (volání) metody pak představuje hodnotu tohoto typu, kterou můžeme používat ve výrazech.
U metod, které nic nevracejí, které voláme jen kvůli jejich „vedlejšímu efektu“, uvedeme typ `void`.

Modifikátory Před určením *typu* může být specifikátor přístupových práv. Jeho význam je podobný jako u datových složek, tj. určuje, kdo smí danou metodu volat. Za specifikátorem přístupu může následovat modifikátor `synchronized`, který oceníme v souvislosti s vícevláknovými programy; zatím ho ponecháme stranou. Pak může následovat jeden z modifikátorů `static`, `final` nebo `abstract`; o jejich významu si povíme dále.

Za závorkou uzavírající seznam parametrů může následovat specifikace výjimek, které se mohou z dané metody rozšířit. Pak následuje blok, tj. příkazy tvořící tělo metody uzavřené ve složených závorkách.

Předchozí povídání můžeme shrnout do následujícího popisu:

deklarace metody, která není abstraktní:

- *přístup_{nep} synchronized_{nep} modifikátor_{nep} typ_{rep} identifikátor(parametry_{nep}) výjimky_{nep} {tělo_metody}*

Typ smíme vynechat jedině u konstruktoru (tam je to naopak nezbytné). Pro úplnost připojíme ještě popis deklarace abstraktní metody:

deklarace abstraktní metody:

- $\text{přístup}_{\text{nep}} \text{ abstract}$
 $\text{typ identifikátor}(\text{parametry}_{\text{nep}}) \text{ výjimky}_{\text{nep}};$

Deklarace abstraktní metody neobsahuje tělo a končí středníkem.

Parametry metod

Formální a skutečné parametry	Při povídání o metodách se setkáváme se dvěma podobnými pojmy: <i>formální parametry</i> a <i>skutečné parametry</i> . Formální parametry se objevují v deklaraci metody, skutečné parametry jsou hodnoty, které zapišeme na místě formálních parametrů při volání.
Formální parametry	Specifikace formálních parametrů se podobá deklaraci proměnných. Skládá se ze zápisu tvaru <i>typ identifikátor</i> , kde <i>typ</i> je typ parametru a <i>identifikátor</i> je jeho jméno. Formální parametr vlastně představuje proměnnou, která vznikne v těle metody v okamžiku, kdy ji zavoláme, a bude obsahovat hodnotu, kterou jsme při volání předali jako skutečný parametr.
Parametry a argumenty	Poznamenejme, že v literatuře se často setkáme i s jinou terminologií: Formální parametry se označují jako <i>parametry</i> a skutečné parametry jako <i>argumenty</i> .
Předávání parametrů	Hodnoty skutečných parametrů se předávají formálním parametrům po řadě: Prvnímu formálnímu parametru se předá hodnota prvního skutečného parametru, druhému formálnímu parametru se předá hodnota druhého skutečného parametru atd.
Představme si, že máme metodu	
	<code>void f(int x, double y): /* ... */</code>
a zavoláme ji příkazem	
	<code>f(5, 6);</code>
	V těle metody <code>f()</code> pak bude mít formální parametr <code>x</code> počáteční hodnotu 5 a formální parametr <code>y</code> hodnotu 6.0. (Hodnota skutečného parametru 6 typu <code>int</code> se konvertuje na typ formálního parametru <code>double</code> , takže <code>y</code> bude 6.0.)
Předávání hodnotou	Parametry se v Javě předávají hodnotou. ⁴⁷ To znamená, že formální parametr je <i>kopii</i> skutečného parametru; změny formálního parametru nezpůsobí změny skutečného parametru.

⁴⁷ V programování se rozlišuje několik způsobů předávání parametrů metodám (obecně podprogramů). O *předávání hodnotou* jsme již hovořili. Při *předávání odkazem* se můžeme na formální parametr dívat jako na jiné jméno

Příklad: Ve 3. kapitole jsme napsali několik verzí výpočtu faktoriálu čísla n , tedy součinu všech přirozených čísel od 1 do n . Připomeňme si, jak může vypadat metoda, která $n!$ počítá:

```
static int f(int n)
{
    int s; // Výsledek
    for(s = 1; n > 0; n--) s *= n; // Výpočet součinu
    return s;
}
```

Formální parametr n používáme v těle této metody jako proměnnou a měníme jeho hodnotu. Jestliže však tuto metodu zavoláme příkazy

```
int x = 5, y = f(x);
System.out.println("faktorial " + x + " je " + y);
```

vypíše

```
faktorial 5 je 120
```



To znamená, že hodnota proměnné x – skutečného parametru – se nezměnila, i když jsme v těle $f()$ měnili hodnotu formálního parametru. (Zdrojový text tohoto příkladu najdete na WWW v souboru Kap09\01\Pokus.java.) ☺

Předávání odkazem

Už víme, že s objekty a poli zacházíme v Javě prostřednictvím odkazů. To platí i pro předávání parametrů. Předáním pole nebo objektu metodě se tedy vytvoří kopie odkazu, nikoli kopie objektu či pole.

Proto změny, které provedeme s formálním parametrem (polem nebo objektem), způsobí změny skutečného parametru. Pole a objekty se tedy vlastně předávají odkazem.

Příklad:

Deklarujeme objektový typ `MujInt`, který zapouzdří typ `int`, ale na rozdíl od standardního typu `Integer` bude umožňovat měnit uloženou hodnotu, a definujeme pro něj metody, které nahradí operátory `--`, `>` a další, potřebné pro výpočet faktoriálu:

```
class MujInt {
    private int i; // Uložené celé číslo
    MujInt(int x){ i = x; } // Konstruktor
    int getInt(){ return i; } // Vrátí uložené celé číslo
    int minusMinus(){ return --i; } // Operátor --
    boolean vetsiNez(int x){ return i > x; } // Operátor >
    void setInt(int x){ i = x; } // Změna uložené hodnoty
    void kratRovnaSe(MujInt n){ i *= n.getInt(); } // Operátor *=
}
```

pro skutečný parametr a změny formálního parametru způsobí okamžité změny skutečného parametru. Existují i jiné způsoby předávání parametrů, tyto dva jsou ale v dnešních programovacích jazycích nejčastější.

Nyní můžeme naprogramovat výpočet faktoriálu i pro tyto třídy:

```
static MujInt f(MujInt n)
{
    MujInt s;
    for(s = new MujInt(1); n.vetsiNez(0); n.minusMinus())
        s.kratRovnaSe(n);
    return s;
}
```

Postup je naprostě stejný jako při výpočtu pro typ `int`, jen operátory jsme nahradili metodami. Jestliže tuto metodu spustíme příkazy

```
MujInt xx = new MujInt(5), yy = f(xx);
System.out.println("faktorial " + xx.getInt() +
    " je " + yy.getInt());
```

dostaneme

```
faktorial 0 je 120
```



Změny formálního parametru zde způsobily, že se změnil skutečný parametr. ♦

Zdrojový text tohoto příkladu najdete na WWW v souboru Kap09\01\Pokus.java.



Všimněte si jmen „přístupových metod“ `getInt()` a `setInt()`. Metody, které vracejí hodnotu datové složky uložené v instanci, se typicky pojmenovávají `get...`, metody, které hodnotu datové složky mění, se typicky pojmenovávají `set...`. Doporučuji, abyste se této konvenci přizpůsobili, neboť např. při programování komponent JavaBeans je v určitých situacích závazná.

Nestatické metody

Zpráva instanci

Na volání nestatické metody se můžeme dívat jako na zprávu posланou instanci. Nestatickou metodu voláme vždy pro konkrétní instanci a tato metoda může pracovat s jejími daty. Při volání kvalifikujeme jméno nestatické metody jménem instance, pro kterou ji voláme. Voláme-li v těle nestatické metody další nestatickou metodu pro tutéž instanci, kvalifikaci neuvádíme. Také datové složky instance, se kterou metoda pracuje, používáme v těle této metody bez kvalifikace.

Podívejme se na třídu `MujInt` z předešlého oddílu. Vytvoříme-li dvě instance,

```
MujInt a = new MujInt(5), b = new MujInt(6);
```

vrátí volání `a.getInt()` hodnotu 5, neboť příkaz `return i;` v těle metody `getInt()` vezme složku `i` z instance `a`. Podobně volání `b.getInt()` vrátí hodnotu 6, neboť vezme složku `i` z instance `b`. ♦

Nic nám samozřejmě nebrání pracovat v těle metody i s jinými instancemi též nebo jiné třídy. Při volání metod pro tyto instance musíme ovšem použít kvalifikaci.

Připomeňme si metodu `kratRovnaSe()` z třídy `MujInt`:

```
void kratRovnaSe(MujInt n) { i *= n.getInt(); }
```

Složka `i` je zde užita bez kvalifikace, a to znamená, že se použije složka aktuální instance. Na druhé straně metodu `getInt()` voláme pro jinou instanci, pro parametr `n`, a proto ji musíme kvalifikovat – musíme napsat `n.getInt()`. Kdybychom kvalifikaci vynechali a napsali

```
void kratRovnaSe(MujInt n) { i *= getInt(); } // CHYBA
```

použila by se metoda `getInt()` pro aktuální instanci a vypočítalo by se `i * i`, což nechceme. ♦

Statické metody

Zpráva tříd

Na volání statické metody se můžeme dívat jako na zprávu poslanou celé třídě. I když to vypadá podivně, je to potřebné. Objektový program se, jak víme, skládá z objektů, které si posílají zprávy. Jenže když chceme např. vytvořit nový objekt, komu máme poslat odpovídající zprávu? Instanci, která ještě neexistuje? To nedává smysl. Musíme ji tedy poslat třídě, protože ta ví, jak mají její instance vypadat.⁴⁸ Podobně chceme-li se zeptat, kolik instancí dané třídy existuje: I takový dotaz je třeba adresovat třídě jako celku, nikoli jednotlivé instanci – už proto, že podobný dotaz má smysl i v situaci, kdy ještě (nebo už) žádná instance neexistuje.

Z předechozího povídání plyně, že statické metody nevoláme pro žádoucí určitou instanci. Při volání z těla jiné metody též třídy ji nemusíme kvalifikovat vůbec, při volání z metody jiné třídy ji musíme kvalifikovat jménem třídy.

Příklad: Ve třídě `Math` jsou definovány statické metody pro výpočet běžných matematických funkcí, jako je sinus ap. Chceme-li tedy spočítat sinus hodnoty `x`, napišeme `Math.sin(x)`.

⁴⁸ To ale znamená, že i třída je objekt, tedy instance jiné třídy (tzv. metatřídy). To v Javě – a v mnoha dalších objektově orientovaných jazycích opravdu platí, ale výklad na toto téma přesahuje možnosti učebnice pro zelenáče.

Jiným příkladem může být čtení celých čísel z konzoly, pro které zatím používáme zápis `MojeIO.inInt()`. Nejde o nic jiného než o volání statické metody `inInt()` třídy `MojeIO`. ♦

Omezení

Protože statické metody nepracují s žádnou konkrétní instancí, nemohou volat nestatické metody a používat nestatické atributy. To jest: Ve statické metodě samozřejmě můžeme vytvořit instanci a s ní pracovat - pro ni můžeme volat nestatické metody atd. Nemůžeme ale např. zavolat ve statické metodě nestatickou metodu bez kvalifikace.

Ve statických metodách samozřejmě smíme používat statické datové složky a volat statické metody.

Podívejme se na jednoduchý příklad.

```
class Test {
    int a;          // Nestatický atribut
    static int b;   // Statický atribut
    int f() { return a + b; }

    public static void main(String[] s) {
        System.out.println(b); // OK - b je statický atribut
        System.out.println(a); // CHYBA - nestatický atribut
        f();                 // CHYBA - nelze volat
                             // nestatickou metodu
        Test t = new Test(); // Vytvoříme instanci
        t.f();              // OK - pro konkrétní instanci lze
                           // volat nestatickou metodu
    }
}
```

Metoda `main()` je statická. Proto v ní můžeme používat bez omezení statický atribut `b`, nikoli však nestatický atribut `a`. Podobně nesmíme zavolat (bez kvalifikace) nestatickou metodu `f()`. Můžeme si ovšem vytvořit instanci třídy `Test` a zavolat `f()` pro ni. ♦

Lokální proměnné

V metodách (statických i nestatických) si můžeme definovat tzv. lokální proměnné. Deklarujeme je podobně jako datové složky tříd, nesmíme u nich však uvádět přístupová práva. Nelze je také deklarovat jako statické.

Tyto proměnné vznikají v okamžiku volání metody a po návratu z ní zanikají. *Mezi jednotlivými voláními se jejich hodnota nezachovává.*

Na rozdíl od statických i nestatických datových složek instancí nejsou automaticky inicializovány a po vytvoření mají náhodnou hodnotu. Překladač ovšem kontroluje, zda nepoužíváme neinitializovanou proměnnou, a pokud ano, ohláší chybu.

Třída Bod

Vraťme se ke třídě Bod, kterou jsme navrhli v první kapitole a kterou jsme si připomněli na počátku této kapitoly. Tato třída bude obsahovat dva něstatické atributy vyjadřující souřadnice a jeden něstatický atribut vyjadřující barvu. (Každý bod může mít jiné souřadnice a jinou barvu.) Navíc nás bude zajímat celkový počet bodů, které byly od spuštění programu vytvořeny; k tomu použijeme statický atribut počet. (Jde o údaj společný pro celou třídu.)

Metody, které budou zjišťovat a nastavovat souřadnice a barvu jednotlivých bodů, budou samozřejmě něstatické, protože budou pracovat s něstatickými atributy. Na druhé straně metoda, která bude zjišťovat počet bodů, musí být statická, neboť ji budeme chtít volat i v situaci, kdy nebude existovat žádná instance.

O vytváření instancí se stará konstruktor. Je tedy přirozené, že se bude starat také o zvyšování hodnoty statického atributu pocet.



Upravenou ikonu této třídy v UML ukazuje nečíslovaný obrázek po straně.⁴⁹ Při implementaci ovšem trochu přejmenujeme metody – přidržíme se konvencí a místo „zjistit“, resp. „nastav“ budeme psát „get“, resp. „set“. Implementace třídy Bod může vypadat např. takto:

```
class Bod {
    private int x, y;           // Soukromé datové složky
    private int barva;
    private static int pocet;   // Počet instancí
    public Bod(){ pocet++; }   // Konstruktor

    public void setX(int _x){ x = _x; }
    public int getX(){ return x; }
    public void setY(int _y){ y = _y; }
    public int getY(){ return y; }
    public void setBarva(int _barva){ barva = _barva; }
    public int getBarva(){ return barva; }
    void nakresli(){
        System.out.println("kreslim bod (" + x + ", " +
                           y + ")");
    }
    public static int getPočet(){ return pocet; }
}
```

Protože zatím nevíme, jak se v Javě kreslí, zvolili jsme v metodě nakresli() zástupnou možnost – vypíšeme informaci o tom, jaký bod se kreslí.

⁴⁹ Znak plus, resp. minus před položkou označuje, že jde o veřejně přístupnou, resp. o soukromou složku. Chráněné složky se v UML označují znakem #.

Nyní můžeme napsat jednoduchý testovací program, který bude obsahovat třídu `Test`. Její metoda `main()` bude mít tvar

```
public static void main(String[] args) {
    System.out.println(Bod.getPocet()); // 1
    final int N = 10;
    Bod[] body = new Bod[N];
    for(int i = 0; i < N; i++) body[i] = new Bod(); // 2
    System.out.println(Bod.getPocet()); // 3
    System.out.println(body[5].getBarva());
    for(int i = 0; i < N; i++) body[i].nakresli();
    // 4
}
```

Zde v řádku označeném v komentáři číslem 1 zjišťujeme počet bodů, i když žádný neexistuje. V řádku označeném 2 vytvoříme pole 10 instancí; v řádku označeném 3 pak znova zjistíme, kolik instance bylo vytvořeno. V řádku označeném 4 body „nakreslíme“; metoda `main()` je sice statická, ale protože ji voláme pro instance, které jsme v ní vytvořili, je vše v pořádku.



Uvedená podoba třídy `Bod` samozřejmě ještě zdaleka není ideální. Tento příklad najdete na WWW v souboru Kap09\02\Test.java. ♦

Rekurze

Metody mohou volat nejen jiné metody, ale i samy sebe, přímo nebo prostřednictvím jiných metod. Tomu se v programování říká rekurze a v Javě ji lze bez problémů používat.

Příklad: V učebnicích se jako jednoduchý příklad často uvádí rekurzivní výpočet $n!$, tedy faktoriálu nezáporného celého čísla n . Jak víme, jde o součin všech čísel od 1 do n . Snadno se přesvědčíme, že ho můžeme definovat také následujícím způsobem:⁵⁰

- pro $n = 0$ platí $0! = 1$,
- pro $n > 0$ platí $n! = n \times (n - 1)!$.

Uvedené vztahy můžeme přímo opsat do programu:

```
static int f(int n)
{
    if(n == 0) return 1;
    else return n*f(n-1);
}
```

⁵⁰ V těchto dvou vztazích znamená znak = rovnost, nikoli přiřazení.

Metodu jsme deklarovali jako statickou, protože nepracuje se žádnými daty instance a tedy není důvod, proč bychom ji měli pro nějakou instanci volat.

Jestliže předáme této metodě hodnotu 0, vrátí 1. Jinak zavolá sama sebe s parametrem $n-1$ a vrátí výsledek vynásobený n . To znamená, že při volání $f(3)$ vrátí $3*f(2)$. To ale způsobí nové volání, $f(2)$, které vrátí $?*f(1)$ atd.



Zdrojový text i s jednoduchým použitím najdete na WWW v souboru Kap09\03\RekFakt.java. ♦

Je jasné, že řetěz postupných volání musí někde skončit, tj. že pro nějakou hodnotu musíme umět naši úlohu vyřešit bez rekurze.

Příklad s faktoriálem je trochu umělý – přece jen počítat součin pomocí rekurze není obvykle to, co nás napadne první. Existují ovšem úlohy, kdy rekurzivní formulace přirozeně vyplýne z rozkladu úlohy na jednodušší kroky, když zjistíme, že po rozdělení jsme dostali úlohu téhož typu, jen s jinými daty.

Metoda main()

Už víme, že spustíme-li javský program příkazem

`java jméno_třídy`

vyhledá JVM v dané třídě metodu `main()` a začne od jejího prvního příkazu. Ovšem aby ji JVM našla, musí mít tato metoda hlavičku

`public static void main(String[] s)`

ve které lze změnit pouze jméno formálního parametru.

Na druhé straně metodu `main()` může obsahovat i více tříd v programu. JVM použije tu, kterou najde v udané „startovní“ třídě, ostatní bude ignorovat.

Parametr metody main() Jediným parametrem metody `main()` je pole řetězců, které obsahuje parametry příkazové řádky programu. Ty můžeme ignorovat nebo zpracovat, jak je nám libo.

Příklad:

Napíšeme program, kterému zadáme v příkazové řádce číslo v desítkové soustavě a základ soustavy, do kterého ho chceme převést. Program vypíše číslo v cílové soustavě. Přitom využijeme služeb třídy Konvertor, kterou jsme vytvořili v 6. kapitole a kterou najdete v souboru Kap06\01\Konvertor.java. Stačí, když tento soubor (nebo odpovídající soubor .class) překopírujete do aktuálního adresáře nebo do některého z adresářů, na které ukazuje proměnná CLASSPATH.

My napíšeme pouze třídu `Prevod`, jejíž metoda `main()` zpracuje parametry příkazového řádku:

```
public class Prevod {  
    public static void main(String[] args) {  
        if(args.length != 2)  
        {  
            System.out.println("Pouziti:\n"+  
                "java Prevod cislo ciłova_soustava\n"+  
                "ciłova soustava musi byt v rozmezí 2 - 36");  
            System.exit(1);  
        }  
        int cislo = Integer.parseInt(args[0]);  
        int zaklad = Integer.parseInt(args[1]);  
        System.out.println(  
            new Konvertor(zaklad).konverze(cislo));  
    }  
}
```

Program nejprve zkontroluje, zda dostal v příkazové řídce dva parametry. To zjistí pomocí proměnné `length`, která je v parametru `args` k dispozici (jako v každém poli). Pokud jich dostal více nebo méně, vypíše informaci o tom, jak je ho třeba spouštět, a skončí. (Tak se chovají všechny slušně vychované programy spouštěné z příkazové řádky.)

Pokud má program správný počet parametrů, převede je z řetězcové podoby na čísla. K tomu využije statickou metodu `parseInt()` třídy `Integer`. Nakonec vytvoří instanci třídy `Konvertor` (konstruktorem předá jako parametr základ cílové soustavy), zavolá její metodu `konverze()` s parametrem rovným danému číslu a výsledek vypíše.

Spusťme-li tento program příkazem

```
java Prevod 8 2
```

vypíše 1000, neboť to je zápis čísla 8 ve dvojkové soustavě.



Zdrojový text tohoto programu najdete na WWW v souboru Kap09\04\Prevod.java. ♦

this

Zamyslime-li se nad voláním nestatických metod, zjistíme, že v jejím těle musí program znát instanci, pro kterou je volána. Jinak by nedokázal určit, kde vzít složky, se kterými má pracovat. Občas bychom se ovšem na tuto instanci potřebovali explicitně odvolat také my – programátoři. To jde; v těle nestatické metody představuje odkaz na aktuální instanci klíčové slovo `this`.

Příklad: Připomeňme si třídu Bod, o níž jsme hovořili o 3 stránky zpět, a její metodu

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

Jestliže tuto metodu zavoláme příkazem `a.getX()`, bude v jejím těle `this` znamenat odkaz na instanci `a`. Překladač si k identifikátoru proměnné `x` doplní kvalifikaci `this.x`, a proto volání `a.getX()` správně vrátí složku `x` instance `a`. ♦

Ve statických metodách není `this` k dispozici.

Konstruktory

Konstruktor je zvláštní metoda, která má na starosti vytvoření instance. Měl by se postarat o inicializaci všech datových složek, které nejsou inicializovány přímo v deklaraci.

Konstruktor se musí jmenovat stejně jako třída, ke které patří. Chceme-li, aby třída měla více konstruktorů, musí se lišit počtem nebo typem parametrů. V deklaraci konstruktoru nesmíme uvést typ vrácené hodnoty (ani `void`). Nesmíme také použít modifikátory `final`, `static` nebo `abstract`.

Pokud v nějaké třídě nedefinujeme žádný konstruktor, vytvoří si ho překladač sám a my ho pak můžeme používat. Takovýto konstruktor má prázdné tělo a je veřejně přístupný. Jestliže však deklarujeme alespoň jeden konstruktor, překladač už žádné konstruktory vytvářet nebude.

Konstruktor voláme při vytváření nové instance pomocí `new`. Syntax tohoto volání je

volání konstruktoru:

- `new identifikátor_třídy(parametry_nep)`

Tento výraz představuje odkaz na nově vytvořenou instanci.

Příklad: Deklarujeme třídu A bez konstruktoru:

```
class A {}
```

Její instanci můžeme vytvořit zápisem

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

neboť překladač si do deklarace této třídy doplnil konstruktor s prázdným tělem:

```
class A { A(){}}}
```

Kdybychom však do deklarace třídy A doplnili konstruktor s parametry,

```
class A { A(int i){} }
```

způsobilo by volání new A() chybu, neboť jsme v této třídě již jeden konstruktor definovali a překladač si žádný další nevytvořil, takže konstruktor bez parametrů ve třídě A chybí. ♦

Metody s proměnným počtem parametrů

Chceme-li v Javě napsat metodu, o níž předem nevíme, s kolika parametry ji budeme volat, použijeme jako formální parametr pole. Představme si např., že chceme napsat metodu, která najde největší ze zadaných celých čísel, ovšem jejich počet předem neznáme. Napíšeme tedy metodu, která bude mít jako parametr pole:

```
public static int max(int[] x)
{
    // res: budoucí výsledek
    int res = Integer.MIN_VALUE; // Nejmenší možná hodnota
    if(x != null)               // Je-li parametr null, konec
    {
        // Projdi pole a najdi největší prvek
        for(int i = 0; i < x.length; i++)
            if(x[i] > res) res = x[i];
    }
    return res;                  // a vrát ho
}
```

Tato metoda nejprve uloží do pomocné proměnné nejmenší možnou hodnotu typu int, která je k dispozici jako statická konstantní složka MIN_VALUE ve třídě Integer. Pak projde v cyklu předané pole a najde největší prvek. Pokud předáme této metodě jako parametr null nebo prázdné pole (pole o délce 0 prvků), vrátí Integer.MIN_VALUE.

Chceme-li zjistit největší hodnotu z proměnných a, b a c typu int, vytvoříme z těchto hodnot pole a to předáme metodě max():



```
int z = max(new int[]{a, b, c});
```

Zdrojový text třídy obsahující tuto metodu a její použití najdete na WWW v souboru Kap09\06\Max.java.

Výpustka

Řešení, které jsme si ukázali, můžeme použít ve všech verzích JDK. V Javě 5 máme navíc ještě jednu možnost: Specifikovat jako parametr tzv. výpustku (anglicky se nazývá ellipsis). I když jde jen o syntaktickou zkratku – tedy zjednodušený zápis – pro specifikaci pole, je to v mnoha situacích velice příjemné a pohodlné.

5

Specifikace výpustky:

- typ ... identifikátor

Výpustka musí být v seznamu parametrů metody poslední. Skládá se z typu proměnných, které chceme této metodě předávat, tří teček bezprostředně za sebou a z identifikátoru formálního parametru.

Identifikátor představuje těle metody jednorozměrné pole daného *typu*. Při volání však můžeme na místě výpustky zapsat seznam výrazů oddělených čárkou; tyto výrazy samozřejmě musí být daného *typu* nebo je musí překladač mocí na tento *typ* převést.

Podívejme se na znovu na hledání největšího ze zadaných čísel. V JDK 5 můžeme metodu `max()` napsat takto:

```
public static int max(int ... x)
{
    int res = Integer.MIN_VALUE; // Nejmenší možná hodnota
    if(x != null)              // Je-li parametr null, konec
    {
        for(int s: x)          // Projdi pole
            if(s > res) res = s;
    }
    return res;                // a vrát ho
}
```

Tuto metodu pak zavoláme např. zápisem

```
int z = max(a, b, c);
```



Zdrojový text třídy obsahující tuto metodu a její použití najdete na WWW v souboru Kap09\06\Max5.java.

S metodami s proměnným počtem parametrů se v knihovně Javy 5 setkáme na více místech. Příkladem může být např. metoda `printf()`, kterou lze použít k formátovanému výstupu.

9.5 Dědění

O významu dědění v OOP jsme si povídali v první kapitole. Teď se podíváme, jak se programuje v Javě.

Jak víme, pokud neurčíme v deklaraci třídy žádného předka, bude třída odvozena přímo od třídy Object, která je společným předkem všech jazykových tříd. Chceme-li, aby třída měla jiného předka, musíme za identifikátor v deklaraci třídy zapsat klíčové slovo `extends` a jméno předka (v případě potřeby včetně balíku).

Připomeňme si, že Java zná jen jednoduchou dědičnost, tj. každá třída může mít jen jednoho bezprostředního předka (zapsaného v klauzuli `extends`.) Potomek (odvozená třída) zdědí všechny datové složky a všechny metody předka. My k nim můžeme v odvozené třídě přidat

nové datové složky a nové metody. Můžeme také některé zděděné metody předefinovat.

Jak víme, můžeme instanci potomka použít všude tam, kde program očekává instanci předka.

Předefinované metody

Jak víme, odvozená třída je zpravidla specializací předka. Některé operace proto musíme v potomkově implementovat odlišně – musíme předefinovat⁵¹ odpovídající metody.

Musí mít stejnou hlavičku

Přitom musí nová definice metody mít stejnou hlavičku, tj. musí mít stejný identifikátor a parametry stejných typů ve stejném pořadí a musí vracet hodnotu téhož typu.⁵² Pokud toto pravidlo nedodržíme, metodu nepředefinujeme, ale přetížíme – v potomkově budeme mít dvě různé metody vedle sebe a program se bude chovat jinak, než chceme. Přístupová práva v potomkově nesmí být užší než v předkovi – nelze např. metodu, která je v předkovi veřejná, deklarovat v potomkově jako soukromou.

5 Java 5 nabízí elegantní možnost, jak zajistit, aby překladač zkontoval, zda určitá metoda předefinovává nějakou metodu předka. Slouží k tomu tzv. *anotace @Override*, kterou připojíme jako modifikátor před deklarací metody v potomkově. Například takto:

```
@Override public void blíkej(int interval) { // ... }
```

super

Občas se stane, že předefinovaná metoda dělá téměř totéž co původní metoda v předkovi, a k tomu přidává cosi navíc.

Kdybychom např. od třídy *Bod*, se kterou se v této kapitole čas od času setkáváme, odvodili třídu *ZakrouzkovanyBod*, potřebovali bychom při kreslení nejprve nakreslit bod a pak okolo něj kroužek. Bylo by samozřejmě nesmyslné programovat kreslení bodu znova, když jsme to už jednou udělali v předkovi, ve třídě *Bod*.

⁵¹ V angličtině se používá termin *override*, což bývá také překládáno slovem *překrýt* nebo poněkud nevhodně *potlačit*.

⁵² V JDK 5 bylo toto pravidlo poněkud uvolněno: Vrací-li metoda v předkovi odkaz na třídu *T*, může překrývající metoda v potomkově vracet odkaz na třídu *U*, která je potomkem *T*.

`super` Java umožňuje volat v potomkovi metodu předka. K tomu postačí, když identifikátor metody kvalifikujeme klíčovým slovem `super`.⁵³

Příklad: Metoda `nakresli()` ve třídě `ZakrouzkovanýBod` by tedy mohla mít tvar

```
public void nakresli() {
    super.nakresli(); // Voláme metodu předka - nakreslíme
                      // "obyčejný" bod.
    NakresliKrouzek(); // Zde nakreslíme kroužek okolo bodu.
}
```

Pro jednoduchost předpokládáme, že existuje metoda `NakresliKrouzek()`, která se postará o nakreslení kroužku okolo daného bodu. ♦

Konstruktor potomka

Konstruktory se nedědí. To znamená, že když v předkovi deklarujeme např. konstruktor s parametrem typu `int`, neznamená to, že stejný konstruktor můžeme automaticky používat v potomkovi. Potomek bude mít jen ty konstruktory, které v něm explicitně deklarujeme (nebo konstruktor bez parametrů vytvořený překladačem). Místo toho používá Java rafinovanější mechanizmus:

`super` Konstruktor potomka vždy nejprve volá konstruktor předka. Pokud nám vyhovuje, že zavolá konstruktor bez parametrů (a pokud ho předek má), je vše v pořádku. Jinak musíme konstruktor předka zavolat sami. K tomu použijeme zápis

```
super(parametry_konstruktoru_předka);
```

Volání konstruktoru předka musí být prvním příkazem v těle konstruktoru potomka.

Příklad: grafické objekty

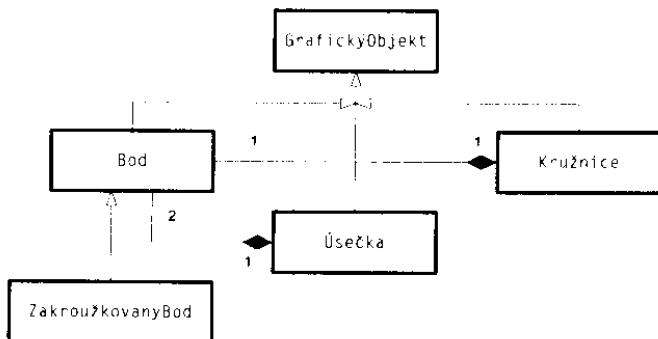
V kapitole 1.5 jsme navrhli hierarchii grafických objektů se společným předkem, třídou `GrafickyObjekt`. Od té doby jsme několikrát použili samostatnou třídu `Bod`; nyní se pokusíme implementovat všechny třídy z této hierarchie, do které navíc přidáme i třídu `ZakrouzkovanýBod`. Diagram těchto tříd v UML ukazuje obr. 9.1.

Každý grafický objekt bude mít barvu vyjádřenou jedním číslem typu `int`, které uložíme do soukromého atributu `barva`, konstruktor a me-

⁵³ Každý objekt odvozené třídy se skládá z objektu bázové třídy a ze součástí přidaných při odvozování. Klíčové slovo `super` představuje odkaz na „zděděný podobjekt“ bázové třídy.

todu `nakresli()`.⁵⁴ Vedle toho bude mít metodu pro změnu a pro zjištění barvy.

Vzhledem k tomu, že naše grafické objekty budou tvořit samostatnou knihovnu, uložíme je do balíku `grafika`. (To znamená, že zdrojové soubory musíme uložit do podadresáře `grafika` adresáře, ze kterého budeme spouštět překladač i JVM.)



Obr. 9.1 Diagram tříd; vyjadřuje jak dědičnost tak i skutečnost, že každá úsečka obsahuje dva body a každá kružnice jeden

GO

Při implementaci vynecháme ve jménech tříd háčky a čárky – i když to není nezbytné – a jméno `GrafickyObjekt` zkrátíme na `GO`. Implementace třídy `GO` může vypadat takto (zdrojové texty najdete na WWW v adresáři Kap09\05\grafika v souborech s odpovídajícími jmény):

```
/* Soubor Kap09\05\grafika\GO.java */
package grafika;

public class GO {
    private int barva; // Soukromá složka obsahující barvu

    public int getBarva() // Metody, které nastavují
        { return barva; } // a zjišťují barvu
    public void setBarva(int _barva){ barva = _barva; }
    public GO() {} // Konstruktory
    public GO(int _barva){ setBarva(_barva); }
    public void nakresli() // Nakreslí GO
    {
```

⁵⁴ Protože dosud neumíme kreslit, použijeme známý trik – místo kreslení vypíšeme zprávu. Až se naučíme kreslit, prostě přepíšeme u jednotlivých tříd metodu `nakresli()`. (Kreslení v Javě však přesahuje rámec této knihy.)

```
        System.out.println("Kreslim GO, barva " + barva);
    }
}
```

Třída GO má dva konstruktory. Jeden je bez parametrů a nedělá nic, druhý má jeden parametr typu int a jeho hodnotu uloží do atributu barva. Všimněte si, že využívá metodu setBarva(). To sice může vypadat jako zbytečné, ale ve skutečnosti to znamená, že s hodnotu atributu barva mění jedině metoda setBarva(). V případě, že se později rozhodneme změnit způsob reprezentace barvy v grafických objektech, nám to může ušetřit práci.

Bod Třída Bod bude potomkem třídy GO; to znamená, že zdědí datovou složku barva a metody pro přístup k ní. Navíc bude mít soukromé datové složky obsahující souřadnice a metody, které souřadnice zjišťují a mění. Metodu pro nakreslení bodu musíme pochopitelně předefinovat, protože bod se kreslí jinak než obecný grafický objekt. Implementace třídy Bod může být

```
/* Soubor Kap09\05\grafika\Bod.java */
package grafika;

public class Bod extends GO {
    private int x, y; // Souřadnice bodu

    public int getX(){ return x; } // Přístupové metody
    public void setX(int _x){ x = _x; }
    public int getY(){ return y; }
    public void setY(int _y){ y = _y; }

    public void nakresli(){ // "Kreslení" bodu
        System.out.println("Kreslim Bod (" + x + ", " + y +
                           "), barva " + getBarva());
    }

    public Bod(int _x, int _y, int _barva) {
        super(_barva);
        setX(_x);
        setY(_y);
    }
}
```

Metody pro práci se souřadnicemi ani metodu pro kreslení není třeba komentovat; podívejme se na konstruktor. Ten má tři parametry – dvě souřadnice a barvu bodu. Barvy je třeba uložit do atributu barva; to jsme ale už naprogramovali v konstruktoru předka, a proto ho zavoláme příkazem

```
super(_barva);
```

K uložení hodnot souřadnic do datových složek `x` a `y` použijeme metody `setX()` a `setY()`, a to ze stejných důvodů, jako jsme použili metodu `setBarva()` k uložení barvy v konstruktoru třídy `GO`.

Zakrouzkovaný Bod

Třída `ZakrouzkovanýBod` je potomkem třídy `Bod`. Musíme v ní deklarovat konstruktor a předefinovat metodu `nakresli()`; ostatní metody i atributy lze bez zámeny převzít od předka. Navíc v ní deklarujeme soukromou⁵⁵ metodu `nakresliKrouzek()`, která se postará o nakreslení kroužku kolem bodu. Implementace této třídy bude

```
/* Soubor Kap09\05\grafika\ZakrouzkovanýBod.java */
package grafika;
public class ZakrouzkovanýBod extends Bod {
    public ZakrouzkovanýBod(int _x, int _y, int _barva) {
        super(_x, _y, _barva);
    }
    private void nakresliKrouzek() {
        System.out.println("Kreslim krouzek");
    }
    public void nakresli() { // Pře definovaná metoda
        super.nakresli(); // Volá metodu předka
        nakresliKrouzek();
    }
}
```



Podívejme se na konstruktor této třídy. I když má stejné parametry a dělá přesně totéž co konstruktor předka, třídy `Bod`, musíme ho deklarovat, neboť konstruktory se nedědí. Kdybychom na to zapomněli, ohlásil by překladač chybu, neboť třída `Bod` nemá konstruktor bez parametrů. Jediným příkazem v těle konstruktoru je

```
super(_x, _y, _barva);
```

který prostě předá parametry konstruktoru předka a ten se postará o vše potřebné.

Zajímavá je i metoda `nakresli()`. Chceme-li nakreslit zakroužkovaný bod, musíme nejprve nakreslit samotný bod a pak kroužek. Kreslení bodu jsme ovšem již naprogramovali v předkově, a proto nejprve zavoláme pomocí klíčového slova `super` metodu předka. Pak nakreslíme kroužek.

Úsečka

Úsečka je určena svými dvěma krajními body. Ty si bude uchovávat v datových složkách `a` a `b` typu `Bod`. Kromě přístupových metod a konstruktoru v ní musíme opět předefinovat metodu `nakresli()`. Implementace této třídy je:

⁵⁵ Je to implementační detail, který nechceme dát k dispozici samostatně.

```

/* Soubor Kap09\05\grafika\Usecka.java */
package grafika;

public class Usecka extends GO {
    private Bod a, b; // Krajní body
    public Bod getA(){ return a;} // Přístupové metody
    public void setA(Bod _a){ a = _a;}
    public Bod getB(){ return b;}
    public void setB(Bod _b){ b = _b;}

    public void nakresli(){ // "Kreslení" úsečky
        System.out.println("Kreslim Usecku, barva " +
                           getBarva() + ", krajní body:");
        a.nakresli();
        b.nakresli();
    }

    public Usecka(int _x1, int _y1, // Konstruktor
                  int _x2, int _y2, int _barva) {
        super(_barva); // Konstruktor předka
        a = new Bod(_x1, _y1, _barva); // Krajní body
        b = new Bod(_x2, _y2, _barva);
    }
}

```

Konstruktor třídy Usecka zavolá konstruktor předka, třídy GO, a svěří mu inicializaci atributu barva. Pak vytvoří oba krajní body.



První test

Třídu Kruznice si můžete zkoušet implementovat sami. (Její zdrojový text najdete na WWW v souboru Kap09\05\grafika\Kruznice.java.)

Nyní si naše grafické objekty vyzkoušíme. Napišeme jednoduchý program, který vytvoří postupně instance jednotlivých tříd a pro každou z nich zavolá metodu `nakresli()`. (Proč bychom jinak vytvářeli grafické objekty, kdybychom je nechtěli kreslit?) Odkazy na vytvořené instance budeme ukládat do proměnné typu GO. To jde, neboť proměnná odkazující na GO může obsahovat odkaz na jakoukoli třídu od GO odvozenou – připomeňme si, že to je jedno ze základních pravidel OOP.

```

import grafika.*;

public class Test {
    public static void main(String[] args) {
        GO g = new Bod(5,6,11);
        g.nakresli();
        g = new Usecka(1,2,3,4,5);
        g.nakresli();
        g = new Kruznice(8, 9, 10, 98);
        g.nakresli();
        g = new ZakrouzkovanýBod(6,9,888);
        g.nakresli();
    }
}

```

	<p>Uvedeme alespoň začátek výstupu tohoto programu:</p> <pre>Kreslim Bod (5, 6), barva 11 Kreslim Usecku, barva 5, krajní body: Kreslim Bod (1, 2), barva 5 Kreslim Bod (3, 4), barva 5 ...</pre>
Polymorfizmus	Je zřejmé, že i když se všemi instancemi pracujeme pomocí odkazu na společného předka, třídu GO, zavolá se vždy metoda <code>nakresli()</code> odpovídající skutečnému typu instance. Připomeňme si, že se tomu říká <i>polymorfizmus</i> a je to jedna z nejúčinnějších zbraní OOP.
Objekty v kontejneru	Při skutečném použití (např. v grafickém editoru) budeme chtít vytvořené objekty zachovat, a proto si je budeme ukládat do vhodného kontejneru. V grafickém editoru ovšem nebude mít předem vědět, kolik objektů uživatel našeho programu vytvoří, a proto nepoužijeme pole, ale kontejner <code>ArrayList</code> . Objekty do něj budeme vkládat pomocí metody <code>add()</code> .
Iterátor	Když budeme chtít nakreslit celý obrázek, tedy všechny uložené grafické objekty, budeme muset projít celý obsah kontejneru prvek po prvku. K tomu lze s výhodou využít tzv. iterátorů, datových struktur, které umožňují zacházet s kontejnerem podobně jako s polem. (Iterátor lze přirovnat k indexu pole, se kterým ale musíme zacházet pomocí metod.) Standardní balík <code>java.util</code> obsahuje třídu ⁵⁶ <code>Iterator</code> ; instanci iterátoru pro náš kontejner získáme pomocí metody <code>iterator()</code> .

Podívejme se, jak by mohla vypadat metoda `nakresliObrazek()`, která vezme instanci `obrazek` a nakreslí všechny grafické objekty v něm uložené:

```
public void nakresliObrazek() // Nakreslí grafické objekty
{
    // uložené v kontejneru
    Iterator i = obrazek.iterator(); // Iterátor poslouží jako
                                    // parametr cyklu
    while( i.hasNext() ) {
        GO g = (GO)i.next(); // Vezmi grafický objekt
        g.nakresli();         // z kontejneru a nakresli ho
    }
}
```

Zde nejprve vytvoříme instanci `i` iterátoru; ta nám poslouží jako parametr cyklu, ve kterém projdeme všechny prvky kontejneru. Iterátor po vytvoření „ukazuje“ na první prvek kontejneru.

⁵⁶ Iterátor je ve skutečnosti rozhraní, ale zatím se na ně můžeme dívat jako na třídu.

Podmínka opakování je `i.hasNext()`. Metoda `hasNext()` vrací `true`, pokud iterátor ukazuje na nějaký prvek, pokud už není na konci kontejneru. Prvek, na který iterátor ukazuje, vrátí metoda `next()`, která zároveň iterátor „přesune“ na následující prvek kontejneru.

Problém ale je, že standardní kontejnery mohou obsahovat jakékoli objekty, tedy instance jakékoli třídy odvozené od třídy `Object`. Proto metoda `next()` vrací odkaz na `Object`. Abychom tento odkaz mohli přiřadit proměnné typu `G0`, musíme ho přetypovat, neboť přetypování na potomka není v Javě automatické. Správnost tohoto přetypování nemůže kontrolovat překladač, kontroluje se až za běhu programu. Pokud bychom do kontejneru uložili objekt typu, který není potomkem třídy `G0`, vznikla by při přetypování za běhu programu výjimka.



Na WWW v souboru Kap09\05\Test.java najdete třídu `Test`, která naplní kontejner několika náhodně vytvořenými grafickými objekty a pak je nakreslí. Obsahuje mimo jiné i právě uvedenou metodu `nakresliObrazek()`.

Abstraktní metody, abstraktní třídy

Už z první kapitoly víme, že `G0` je typickým příkladem abstraktní třídy. Nebudeme vytvářet její instance – budeme potřebovat body, úsečky, ale nikdy obecné grafické objekty. Obecný grafický objekt nelze nakreslit – nám se to podařilo jen proto, že jsme kreslení nahradili výpisem zprávy. Mohlo by se tedy zdát, že metoda `nakresli` ve třídě `G0` nemá smysl. Pokud ale tuto metodu z deklarace třídy `G0` odstraníte, ohláší překladač v příkazu

`g.nakresli(); // g je typu G0`

chybu – volání neznámé metody. To znamená, že tuto metodu deklarovat musíme, i když nemá žádný smysl.

Abstraktní metody

Takovou metodu deklarujeme jako abstraktní pomocí klíčového slova `abstract`. Abstraktní metoda nemá tělo, pouze „drží místo“ pro implementaci v odvozených třídách. Její deklarace končí středníkem.

Třídu, která obsahuje alespoň jednu abstraktní metodu, musíme také deklarovat jako abstraktní, to znamená, že v její deklaraci musíme před klíčovým slovem `class` uvést modifikátor `abstract`.

Příklad: Deklaraci třídy `G0` tedy upravíme následujícím způsobem:

```
public abstract class G0 {  
    private int barva;  
    public int getBarva() { return barva; }  
    public void setBarva(int _barva) { barva = _barva; }
```

```
public GO() {}  
public GO(int _barva) { setBarva(_barva); }  
public abstract void nakresli(); // Abstraktní metoda  
} ◇
```

Překladač nám nedovolí vytvářet instance abstraktních tříd.

Finální třídy, finální metody

S klíčovým slovem `final` jsme se zatím setkávali jen v deklaracích datových složek nebo lokálních proměných; tam znamenalo konstantu („proměnnou, jejíž hodnotu nelze měnit“). Můžeme je však použít i ve deklaraci třídy nebo metody.)

Metoda s modifikátorem `final` je „konečná“, tj. nelze ji v odvozených třídách předefinovat. (Jako konečné lze deklarovat jen nestatické metody.)

Třídu s modifikátorem `final` označujeme také jako „konečnou“, neboť od ní nelze odvozovat potomky. Všechny její metody jsou automaticky konečné.

10 Výjimky

Co dělat, když v programu dojde k chybě? Zatím jsme se tvářili, že se nás to netýká, že se to našim programům nemůže stát. Jenže to je možné ve školních programcích, ale ne ve vážně miněných aplikacích. Počítače dnes už řídí kděco a programy musí počítat s tím, že něco bude jinak, než by mělo. Zkuste si např. představit, že uprostřed přistávacího manévrů odmítne palubní počítač v letadle vysunout podvozek, neboť v programu, který se o to stará, došlo k chybě, protože pilot zmáčkl špatné tlačítko.

Nebo si zkuste představit, že máte otevřený soubor, upravujete v něm data, a uprostřed úprav dojde k chybě. Pokud by program v tom okamžiku skončil, mohlo by dojít k poškození dat v něm a tím i k obrovským škodám. (A nemusí jít jen o soubor s bankovními účty.)

Program by měl počítat nejen s chybami svého okolí, ale i se svými vlastními chybami. Výzkumy ukazují, že software špičkové kvality obsahuje jednu chybu přibližně v každých 10 000–20 000 řádcích zdrojového kódu [7].

Dnes se stává samozřejmým požadavkem odolnost softwaru proti chybám (fault tolerance). Přitom „chybou“ může být nesprávně zadáný vstup, chybějící nebo poškozený soubor, přerušení síťového spojení, ale také třeba dělení nulou. Chybu zjistíme v nějaké metodě, kterou volala jiná metoda ... a v metodě, kde chybu zjistíme, ji nedokážeme opravit, protože k tomu nemáme potřebné informace.

Příklad: Zkuste znova spustit program `Prevod`, který jsme vytvořili v kapitole 9.4 v oddílu věnovaném metodě `main()`, příkazem

```
java Prevod co ty na to
```

Tento program očekává v příkazové řádce dvě čísla v desítkové soustavě, a pokud je tam nenajde, vypíše obsáhlé chybové hlášení a skončí. To způsobí metoda `Integer.parseInt()`, která jako vstup očekává znakový řetězec představující celé číslo v desítkové soustavě. Pokud dostane jiný řetězec, je to chyba, kterou ale nelze v této metodě opravit – to musí dát do pořádku ten, kdo ji se špatným parametrem zavolal. ♦

Takže ještě jednou: Co dělat, když v programu dojde k chybě, kterou nedokážeme v daném místě opravit? V principu máme několik možností:

- Můžeme vypsat zprávu a ukončit program. To je sice dobré pro ladění, ale dodat takovýto program zákazníkovi, to prostě nelze. I když je chyba na straně uživatele. Náš program by měl alespoň popsat chybu, ke které došlo, způsobem srozumitelným uživateli, a doporučit mu, jak s programem zacházet.
- Můžeme ukončit metodu a vrátit hodnotu, která bude signalizovat chybu. To bylo oblíbené řešení ve starších programovacích jazycích – mj. v C a v prvních verzích C++. (Např. faktoriál je definován pouze pro nezáporná čísla a jeho hodnoty jsou vždy kladné. Metoda `f()`, počítající faktoriál, by proto v případě záporné hodnoty parametru mohla vracet 0.) Programátor ale nikdo nenutí vrácenou hodnotu kontrolovat, a tak se může stát, že program bude počítat s nesmyslným výsledkem. Navíc to nelze použít vždy; např. výsledkem metody `parseInt()` může být jakékoli celé číslo, takže nelze žádnou hodnotu vyhradit jako příznak chyby.
- Můžeme do každé instance přidat proměnnou, která bude sloužit jako příznak chyby. Jestliže metoda volaná pro tuto instanci zjistí chybu, uloží do ní kód této chyby. Programátor pak musí po ukončení metody kontrolovat, zda proběhla úspěšně. (Tady je opět slabé místo: Kdo k tomu programátor donutí? Když na to zapomene, může se stát, že program poběží dál a bude dělat nesmysly.)
- Můžeme použít mechanizmus výjimek. Ten umožnuje bezpečný přenos řízení z místa, kde jsme zjistili chybu, na místo, kde ji můžeme osetřit. O něm bude tato kapitola.

10.1 Výjimka v Javě

Když v Javě vznikne výjimka, vytvoří se v programu objekt, který poneše informace o vzniklému problému do místa, kde tento problém bude možné vyřešit. Nositelem informace o druhu problémů je především typ tohoto objektu; navíc do něj lze uložit ještě znakový řetězec s dalšími informacemi.



Aby nebyly věci tak snadno srozumitelné, říká se těmto objektům také „výjimky“. Takže slovo výjimka může znamenat buď chybu v programu nebo objekt, který o ni nese informace. Podle typu tohoto objektu se pak hovoří o „výjimce typu `ArrayIndexOutOfBoundsException`“ ap.

Třídy pro přenos informací o výjimkách

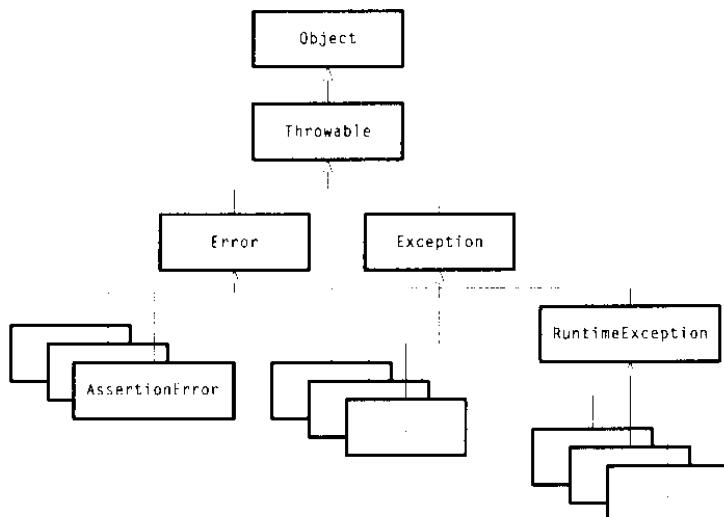
Mechanismus výjimek v Javě používá k přenosu informací o výjimkách tříd odvozených od standardní třídy `Throwable`, která je bezpro-

středním potomkem třídy `Object`. Tato třída má dva bezprostřední potomky, třídy `Error` a `Exception`. V programech se ovšem využívají především potomci těchto dvou tříd. Většina z nich je v balíku `java.lang`.

Error Výjimky odvozené od třídy `Error` se používají jen při chybách běhového systému JVM – např. při vyčerpání paměti. Výjimku tohoto typu může také vyvolat příkaz `assert`. Běžné programy je neošetrují a my se jimi nebudeme zabývat.

Exception Tato třída je předkem řady tříd používaných k přenosu informací o běžných chybách v běžných aplikačních programech. Mezi jejími potomky má zvláštní postavení třída `RuntimeException`, od níž jsou odvozeny třídy indikující nejběžnější chyby, jako je použití odkazu obsahujícího `null` k volání metody (třída `NullPointerException`), celočíselné dělení nulou (třída `ArithmaticException`) atd. Základ hierarchie těchto tříd ukazuje obr. 10.1.

Všechny tyto třídy mají vedle konstruktoru bez parametrů také konstruktor, kterému lze předat znakový řetězec s informacemi o chybě.



Obr. 10.1 Hierarchie objektových typů pro práci s výjimkami v Javě

Vznik výjimky

K vyvolání výjimky slouží klíčové slovo `throw`, za nímž uvedeme odkaz na objekt, který ponese informace o chybě. Formálně lze popsat syntax vyvolání výjimky takto:

vyvolání výjimky:

- **throw odkaz_na_objekt ;**

Můžeme použít odkaz na existující objekt nebo můžeme vytvořit nový pomocí `new`. Po vzniku výjimky se přeruší přirozené pořadí provádění příkazů a program začne hledat handler, část kódu, která vzniklou situaci ošetří.

Příklad: V kapitole 3.4 jsme napsali metodu, která počítala faktoriál celého čísla. V jedné z pokročilejších verzí jsme testovali, zda je skutečný parametr této metody kladný, a pokud nebyl, ukončili jsme program. To je, jak víme, velice nevhodné řešení.

Později jsme správnost parametrů testovali pomocí příkazu `assert`. To má smysl, pokud víme, že ve správně odladěném programu nemůže naše metoda dostat záporný parametr; nelze to ale použít např. u knihovní metody, protože nevíme, jak bude používána.

Nejrozumnějším řešením bude vyvolat výjimku vhodného typu, např. `ArithmetricException`. Tím dáme programátorovi – uživateli naší metody – možnost chybu nějak opravit.

Nová podoba metody `fakt()` může být

```
public static long fakt(int n){  
    if((n < 0) || (n > 20)) throw new ArithmetricException();  
    int s = 1;  
    while(n > 1) s *= n--;  
    return s;  
}
```

Zde nejprve otestujeme, zda je parametr v předepsaném rozmezí, a pokud není, vyvoláme výjimku. (O tom, co se při tom stane, si povíme podrobněji dál.) Jinak se výpočet neliší od většiny předchozích verzí.

Jestliže tuto metodu zavoláme v metodě `main()`,

```
public static void main(String[] args) {  
    System.out.print(fakt(-5));  
}
```

skončí program chybovým hlášením. To sice na první pohled vypadá, že jsme neučinili žádný pokrok, neboť program opět končí, ale jak uvidíme dále, máme nyní možnost s chybou něco dělat. Za prvé se brzy naučíme výjimku zachytit a ošetřit a za druhé z chybového hlášení můžeme při ladění leccos vyčist. Například při spuštění programu, který najdete na WWW v souboru Kap10\01\Fakt10.java a který obsahuje uvedenou verzi metod `fakt()` a `main()`, vypíše počítac

```
Exception in thread "main" java.lang.ArithmetricException
```



```
at Fakt10.fakt(fakt10.java:14)
at Fakt10.main(fakt10.java:20)
```

Z této zprávy se dozvím, že v programu došlo k výjimce typu `java.lang.ArithmetricException`, a to ve třídě `Fakt10` v metodě `fakt()`, kterou najdeme v souboru `Fakt10.java` na 14. řádku. Tato výjimka se rozšířila do metody `main()` v téžem souboru na 20. řádku. ♦

10.2 Ošetřování výjimek

Syntax Práce s výjimkami je v Javě založena na tzv. hlídaném bloku⁵⁷. Operace, které se nemusí podařit, tj. při nichž může vzniknout výjimka, uzavřeme do bloku, před který zapíšeme klíčové slovo `try`. Za něj pak připojíme jeden nebo několik handlerů nebo koncovku (případně obojí). Ukažme si syntaktický popis hlídaného bloku:

hlídaný blok:

- `try { příkazy } handlery nebo koncovka nebo`

Hlídaný blok musí obsahovat alespoň jeden *handler* nebo *koncovku*. Handlerů může být více, koncovka může být nejvýše jedna. *Koncovky* zatím ponecháme stranou, vrátíme se k nim později.

Handler

Handler je část programu, která může ošetřit výjimku. Jeho syntax je

handler:

- `catch(typ identifikátor){ tělo_handleru }`

Začíná klíčovým slovem `catch`⁵⁸, za kterým následuje v závorkách *typ* výjimky, kterou může ošetřit. Musí jít o třídu odvozenou od `Throwable`. *Identifikátor*, který následuje, bude v těle handluera představovat odkaz na objekt, který nese informace o výjimce. Budeme mu také říkat „parametr handluera“. *Tělo handluera* je tvořeno příkazy, které s tím „něco udělají“, tedy které řeší vzniklou situaci.

C +

⁵⁷ Celý mechanizmus práce s výjimkami je velmi podobný jako v C++. Navíc zde najdeme koncovku bloku (`finally`), která je zjevně inspirována strukturovanými výjimkami (běžné rozšíření jazyka C na PC pod Windows).

⁵⁸ *Try* znamená zkus, pokus se – tj. zkus provést následující operace. O bloku s prefixem `try` budeme občas hovořit jako o „pokusném bloku“. *Throw* znamená hod, vrhni – „vyhod“ výjimku“ česky moc pěkně nezna, a proto budu raději říkat, že se výjimky vyvolávají. *Catch* znamená chyt – handler výjimku zachytí a ošetří. U koncovek se ještě setkáme se slovem `finally`, které znamená „na konec“ – tyto operace se provedou na konci pokusného bloku, ať v něm vznikla výjimka nebo ne.

Když vznikne výjimka

Už víme, že při vzniku výjimky se přeruší přirozená posloupnost plnění příkazů a začne se hledat vhodný handler. To znamená, že v bloku, ve kterém vznikla výjimka, se již následující příkazy neprovedou, program tento blok opustí. Je-li to pokusný blok a je-li k němu připojen vhodný handler, přejde do něj. Jinak přejde do nadřízeného bloku (pokud v dané metodě nějaký je) a bude hledat vhodný handler u něj atd. Nenajde-li program vhodný handler v dané metodě, přejde do metody, která ji zavolala, a bude se chovat, jako kdyby výjimka vznikla v příkazu, kterým jsme danou metodu volali. To znamená, že opět přeskočí všechny příkazů do konce bloku a bude hledat vhodný handler. O tomto procesu hovoříme jako o *šíření výjimky*. Nalezením handleru a vstupem do něj se výjimka pokládá za ošetřenou.

Šíření výjimky

Pokud program v handleru neukončíme např. voláním metody `System.exit()`, bude po ukončení handleru pokračovat za hlídaným blokem. To znamená, že přeskočí případně další handlery připojené k témuž pokusnému bloku. (Provede ale příkazy z koncovky, pokud ji hlídaný blok obsahuje.)

Nenajde-li program vhodný handler ani v metodě `main()`, převezme výjimku JVM, ukončí program a vypíše informace o typu, místu vzniku a šíření výjimky, jak jsme to viděli v předchozím příkladu.

Jestliže v pokusném bloku nevznikne výjimka, proběhnou všechny příkazy v jeho těle. Program pak přeskočí handlery, které za ním následují, a pokračuje příkazy za nimi.

Příklad: Ponecháme zatím stranou koncovky a podíváme se, jak bychom mohli v předchozím příkladu využít, co jsme se právě dozvěděli. Metodu `fakt()` ponecháme beze změny; špatná hodnota parametru je problém toho, kdo ji volá, tato metoda to řešit nemůže. Můžeme ale upravit metodu `main()` – využít pokusného bloku.

```
/* Soubor Kap10\01\Fakt10a.java */
public class Fakt10a {
    public static long fakt(int n)           // Stejně jako prve
        if((n < 0) || (n > 20))
            throw new ArithmeticException();      // !
        int s = 1;
        while(n > 1) s *= n--;
        return s;
}

public static void main(String[] args) {
    try {                                // Pokusný blok
        System.out.print("Zadej cele cislo: ");
        int i = MojeIO.readInt();
        System.out.println("Faktorial je " + fakt(i)); // !!
        System.out.print("Vypocet probehl bez problemu");
    }
}
```

```

        }
    catch(ArithmeticException e){ // Handler
        System.out.println(
            "Cislo musí lezeti v rozmezí od 0 do 20");
        /* Konec handleru */
        System.out.println(
            " Ferda Mravenec, prace vseho druhu ");
    } /* Konec metody main() a celé třídy */
}

```

Všimněte si, že jsme do pokusného bloku uzavřeli všechny operace, které se týkají normálního běhu programu, tj. i ty, které výjimku nemohou vyvolat; jde např. výpis zprávy `Vypocet` proběhl bez problémů. Blok `try` totiž obsahuje kód, který se týká normálního běhu programu, a handlery se starají o ošetřování chyb. Aparát pro ošetřování výjimek tak přispívá i ke zpřehlednění programu.

Podívejme se nejprve na běh programu za „normálních“ okolností. Program vypíše žádost o zadání celého čísla. Uživatel napiše např. 8. Volání metody `fakt()` proběhne, aniž nastane výjimka, program vypíše výsledek a zprávu, že vše proběhlo bez problémů. Pak přeskočí handler, vypíše závěrečný reklamní slogan a skončí.

Jestliže však uživatel zadá např. číslo -7, vznikne výjimka ve funkci `fakt()` v příkazu označeném vykřičníkem. To znamená, že se nejprve vytvoří objekt typu `ArithmeticException`. Pak se přeskočí všechny následující příkazy až do konce bloku, tj. do konce těla metody. Protože v této metodě není žádný handler, skončí tím běh metody `fakt()` a program se vrátí na místo, odkud byla volána, k příkazu označenému dvěma vykřičníky. Tento příkaz se už nedokončí a přeskočí se všechny příkazy až do konce bloku. To znamená, že se nevytisknou slova `Faktorial je, ani hlášení Vypocet` proběhl bez problémů.

K tomuto bloku je ovšem připojen handler, který může ošetřit výjimku typu `ArithmeticException`, a proto program do přejde tohoto handleru a vypíše upozornění `Cislo musí lezeti v rozmezí od 0 do 20`. Po ukončení handleru pokračuje prvním příkazem za hlídaným blokem. To znamená, že – stejně jako v případě, že nedošlo k chybě – vypíše reklamní slogan.



Zdrojový text takto upraveného program najdete na WWW v souboru `Kap10\01\Fakt10a.java`. ♦

Výpis informací

Ošetření výjimky je jistě skvělé, při ladění ale potřebujeme často informace o místě výjimky. Můžeme samozřejmě výjimku „propustit“, jako jsme to dělali dosud; rozumnější je ale vytvořit handler a v něm použít metodu `printStackTrace()` objektu, který přenáší informace o výjimce. Například takto:

```
catch(ArithmeticException e){
```

```
        e.printStackTrace();
    }
```

Tato metoda vypíše také případný řetězec předaný konstruktoru výjimkového objektu.

Vyhledávání handleru

Program vyhledává vhodný handler podle typu výjimky. Přitom se uplatňuje známé pravidlo, že potomek může vždy zastoupit předka. To znamená, že handler pro výjimky např. typu `ArithmeticeException` zachytí nejen výjimky tohoto typu, ale i výjimky všech typů, které budou od třídy `ArithmeticeException` odvozeny. Přitom program použije první handler, na který při šíření výjimky narazí a který může výjimku osetřit. To nám umožnuje výjimky třídit. Můžeme se např. rozhodnout, že budeme zvlášť ošetřovat výjimky typu `ArithmeticeException`, ale všechny ostatní „hodíme do jednoho pytle“.

Příklad: Upravme metodu `main()` v předchozím příkladu takto:

```
public static void main(String[] args) {
    try {
        // jako předtím
    }

    catch(ArithmeticeException e) {
        System.out.println("Cislo musí byt od 0 do 20");
    }

    catch(Exception e) {
        System.out.println("Nejaká divna chyba");
        e.printStackTrace();
    }

    System.out.println("Ferda Mravenec, prace vseho druhu");
}
```

Zde jsme k pokusnému bloku připojili dva handlery. První zachycuje už známý případ - špatný parametr. Druhý zachytí všechny ostatní výjimky (zatím by žádné neměly být). Jestliže dojde k výjimce typu `ArithmeticeException`, zachytí ji první handler, protože na něj program narazí dříve. Dojde-li k jakékoli jiné výjimce, zachytí ji až druhý handler (ten zachytí všechny výjimky, které k němu projdou.) ♦

Příklad:

V kapitole 9.4 jsme napsali program pro převod čísel do libovolné soustavy, který využíval třídu `Konvertor` z oddílu 6.1. Teď už vidíme, že jak třídu `Prevod`, tak třídu `Konvertor` je možné podstatně vylepšit. Ani jedna totiž není zabezpečena proti hloupým chybám uživatele – a mechanizmus výjimek nám k tomu dává skvělou možnost.

Jednou z chyb, kterých se může uživatel programu dopustit a která v předchozí verzi nebyla ošetřena, je zadání základu číselné soustavy mimo rozmezí 2–36. Protože tuto chybu budeme chtít později odlišit

od ostatních, definujeme si pro ni vlastní třídu odvozenou od `RuntimeException`:

```
/* Soubor Kap10\02\konvertor\WrongBaseException.java */
package konvertor;

public class WrongBaseException extends RuntimeException {
    public WrongBaseException() {}
    public WrongBaseException(String s) { super(s); }
}
```

Tato třída obsahuje pouze dva konstruktory. Jeden z nich má jako parametr znakový řetězec, do kterého uložíme informaci o skutečně zadané soustavě. Tento konstruktor prostě předá svůj parametr konstruktoru předka. Druhý konstruktor je bez parametrů a nedělá nic; deklarujeme ho jen proto, že by si ho překladač sám nevytvořil.

Dále upravíme konstruktor třídy `Konvertor`, jejíž instance se starají o konverzi. Tento konstruktor zkontroluje, zda požadovaný základ soustavy leží v předepsaném rozmezí, a pokud ne, vyvolá výjimku:

```
package konvertor;
public class Konvertor {
    private long zaklad; // Základ číselné soustavy
    static String cislice =
        "0123456789ABCDEFHIJKLMNOPQRSUVWXYZ";
    public String konverze(long n){
        // Stejná jako v kap. 6.1
    }

    public Konvertor(int baze) { // Konstruktor nyní může
        if(baze < 2 || baze > 36) // vyvolat výjimku
            throw new WrongBaseException(""+baze);
        else zaklad = baze;
    }
}
```

Konstruktoru výjimky předáme řetězec představující desítkový zápis špatného základu.

Třída `Prevod`, jejíž metoda `main()` zpracovává parametry příkazové řádky a předává je instanci třídy `Konvertor`, může vypadat takto:

```
public class Prevod {
    public static void main(String[] args) {
        try {
            int cislo = Integer.parseInt(args[0]);
            int zaklad = Integer.parseInt(args[1]);
            System.out.println(
                new konvertor.Konvertor(zaklad).konverze(cislo));
        } catch(RuntimeException e)
```

```

    {
        System.out.println("Pouziti:\n"+
            "java Prevod cislo cilova_soustava\n"+
            "cilova soustava musi byt v rozmezi 2 - 36");
        e.printStackTrace();
    }
}

```

Všimněte si, že použití výjimek opět od sebe oddělilo „normální“ běh programu a ošetřování chyb. Odpadl také test počtu parametrů. Pokud uživatel zadá nějaké navíc, nic se neděje, pokud jich zadá málo, dojde při přístupu k nim k výjimce typu `ArrayIndexOutOfBoundsException`. Pokud uživatel zadá jako parametr řetězec, který nepředstavuje celé číslo, vyvolá metoda `Integer.parseInt()` výjimku `NumberFormatException`. Konečně pokud sice zadá celá čísla, ale základ cílové soustavy nebude ležet v předepsaném rozmezí, vyvolá konstruktor třídy `Konvertor` výjimku `WrongBaseException`. Všechny tyto třídy jsou potomky typu `RuntimeException`, a proto je může zachytit jeden společný handler. Ten vypíše upozornění, jak se má program používat, a ladící hlášení.

Ukončení programu voláním `System.exit()` je nyní zbytečné, protože handler je na konci metody `main()`.



Tento program najdete na WWW v adresáři Kap10\02 a v jeho podadresáři `konvertor`, odpovídajícím balíku, ve kterém leží třídy `Konvertor` a `WrongBaseException`. ♦



V handleru můžeme vyvolat novou výjimku stejného nebo jiného typu. Tu už nezachytí žádný z následujících handlerů u téhož bloku, ale až handler nadřízeného bloku. Jestliže v příkazu `throw` uvnitř handleru uvedeme parametr handleru, vyvoláme tím výjimku stejného typu a se stejnou instancí, jako byla aktuální výjimka.

Koncovka

Když vznikne výjimka, postará se program o správné zrušení lokálních proměnných v metodách, které při hledání handleru opustí. Instance polí a objektových typů zaniknou, když zaniknou odkazy na ně; o to se postará automatická správa paměti.

Může se ale stát, že si v metodě otevřeme soubor a při jeho zpracování vznikne výjimka. Před opuštěním metody bychom ho potřebovali zavřít. Může se stát, že napišeme na obrazovku nápis, který tvrdí, že zpracování probíhá úspěšně, a tento nápis bychom při výjimce potřebovali odstranit.

Může se zdát, že by se o to měl postarat handler; jenž jak uzavření souboru, tak odstranění nápisu musíme udělat nejen v případě, že se akce podaří, ale i v případě, že v průběhu zpracování dojde k výjimce. Abychom je nemuseli programovat dvakrát, použijeme koncovku hlídaného bloku. Operace v ní se provedou, ať skončí pokusný blok normálně nebo výjimkou. Její syntax je:

koncovka hlídaného bloku:

- **finally{ příkazy }**

Použití si ukážeme nejprve jen schématicky:

```
try {
    OtevriSoubor(F);      // Zkus to
    Zpracuj(F);
}
catch(Exception e){
    NecoSTimUdelej();    // a když se to nepovede...
}
finally{
    ZavriSoubor(F);     // Tohle udělej v každém případě
}
```



Poznamenejme, že koncovka se provede i v případě, že pokusný blok opustíme příkazem `return`, `break` nebo `continue`. Neproběhne v případě, že v handleru ukončíme program voláním `System.exit()`.

Příklad:

Zatím neumíme pracovat se soubory, a proto bude následující příklad trochu umělý. Vrátme se ještě jednou ke třídě `Prevod` z předchozího příkladu a upravíme ho tak, aby při různých chybách vypisoval různé zprávy. Ve všech případech ale chceme, aby program vypsal poučení, jak se má spouštět programu. Proto upravíme metodu `main()` třídy `Prevod` následujícím způsobem:

```
public static void main(String[] args) {
    try{
        int cislo = Integer.parseInt(args[0]);
        int zaklad = Integer.parseInt(args[1]);
        System.out.println(
            new konvertor.Konvertor(zaklad).konverze(cislo));
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(
            "Pri volani je treba zadat dva parametry");
    }
    catch(NumberFormatException e){
        System.out.println("Jeden z parametru není číslo " +
            e.getMessage());
    }
}
```

```

        catch(Konvertor.WrongBaseException e){
            System.out.println("Zaklad " + e.getMessage() +
                "lezi mimo interval 2--36");
        }
    finally{
        System.out.println("Pouziti:\n"+
            "java Prevod cislo_cilova_soustava\n"+
            "cilova soustava musi byt v rozmezi 2--36\n");
    }
}

```

Třídy, které jsme použili k přenosu informací o výjimce, mají sice společného předka, ale navzájem nejsou předky ani potomky, a proto na pořadí handlerů nezáleží. Při výjimkách způsobených špatným základem soustavy nebo řetězcem, který nelze převést na číslo, se do instance uloží řetězec, který obsahuje hodnotu, jež způsobila chybu. Tento řetězec získáme v handleru pomocí metody getMessage() a vypíšeme ho. (Tento program najdete na WWW v adresáři Kap10\03.)



Spustíme-li takto upravený program příkazem

```

java Prevod 8 99
vypíše
Zaklad 99 lezi mimo interval 2--36
Pouziti:
java Prevod cislo_cilova_soustava
cilova soustava musi byt v rozmezi 2--36 ◊

```

10.3 Výjimky a metody

V úvodních kapitolách jsme si řekli, že syntaktická pravidla Javy požadují, abychom v deklaraci každé metody uvedli výjimky, které se z ní mohou rozšířit. K tomu slouží klíčové slovo throws, za kterým následuje seznam typů výjimek. To znamená, že např. konstruktor třídy Konvertor z předchozího příkladu by měl mít tvar

```
public Konvertor(int base) throws WrongBaseException
```

Pokud jste si ale předchozí příklady přeoložili, zjistili jste, že v některých případech na tom překladač netrvá. Přesné pravidlo zní: *V hlavičce metod musíme uvádět výjimky, které nejsou odvozeny od třídy RuntimeException.* Neuvádíme tam také výjimky odvozené od třídy Error.

Výjimky odvozené od třídy RuntimeException se totiž vyskytují tak často, že jejich důsledné uvádění by znepřehlednilo program.

C⁺ Poznamenejme, že i když se specifikace výjimek v deklaraci metody v Javě velice podobá specifikaci výjimek v deklaraci funkce v C++, její význam je poněkud odlišný. Zatímco v C++ tato specifikace způsobí *běhovou* kontrolu, v Javě umožňuje kontrolu v době překladu. V Javě se nemůže stát, že by se z metody rozšířila výjimka nedeklarovaného typu.

11 Rozhraní

Tuto kapitolu začneme trochu ze široka: Připomeňme si třídu `ZakrouzkovanyBod`, kterou jsme navrhli v 9. kapitole. (Odpovídající hierarchii ukazuje obr. 9.1 v oddílu 9.5.) Odkaz na instanci této třídy, vytvořenou např. příkazem

`ZakrouzkovanyBod zb = new ZakrouzkovanyBod(1, 55, 14789);`

můžeme přiřadit proměnné typu odkaz na `Bod` nebo odkaz na `GO`. Důvod již dávno známe – tyto dvě třídy jsou předky třídy `ZakrouzkovanyBod`. Zatím jsme si říkali, že jde prostě o jedno ze základních pravidel OOP a víc jsme o tom neuvažovali. Zkusme se na to nyní podívat z jiné strany: *Přiřazení*

`Bod b = zb;`

je správné, protože instance třídy `ZakrouzkovanyBod` je zároveň také instance třídy `Bod`. Připomíná vám to, že odvozená třída je speciálním případem svého předka, jak jsme si říkali v 1. kapitole? Jistě, je to totéž, jen jsme to vyjádřili jinými slovy.

Instance může mít několik typů

Dědičnost tedy umožňuje mít instance, které naleží k několika vzájemně blízkým typům. Občas se ale stane, že potřebujeme, aby instance naležela k několika typům, které spolu logicky příliš nesouvisejí; to umožňují rozhraní (interface).⁵⁹

Seznam metod

Rozhraní je vlastně seznam metod, které se třída zavazuje implementovat. Můžeme se na ně dívat jako na abstraktní třídu, která nesmí obsahovat nic jiného než abstraktní metody.

Příklad:

Představte si (jako už po kolikáté), že píšeme grafický editor a používáme v něm mj. třídy odvozené od společného předka `GO`. V programu máme instanci `g` a potřebovali bychom napsat

```
zobraz(g); // Naše metoda, očekává instanci třídy GO  
SaveToFile(g); // Knihovní metoda, očekává Serializable
```

Dvě různé metody, které chceme použít ke zpracování jedné instance, očekávají parametry dvou různých tříd. Řešení založené na dědičnosti

⁵⁹ Rozhraní jsou v objektovém programování poměrně novou věcí. Java byla jedním z prvních široce používaných jazyků, které je programátorům nabídry; po vzoru Javy se nyní objevují i v jiných jazycích – najdeme je např. v několika posledních verzích Object Pascalu v Delphi.

by vyžadovalo odvodit třídu `G0` od třídy `Serializable`⁶⁰. To ale není nejšikovnější, protože do souborů můžeme chtít ukládat také záznamy o událostech, které v programu nastaly, záznamy o počasí atd. Měly by to všechno být třídy se společným předkem `Serializable`?

Položme si tuto otázku jinak: Co by měly společného? Samozřejmě schopnost nechat se zapsat do souboru a později přečíst, tedy metody, které nazveme třeba `uloz()` a `nacti()`. Jenže tyto metody budou implementovány v každé třídě jinak, neboť grafický objekt se bude ukládat do souboru jinak než záznam o počasí. To znamená, že tyto třídy budou mít společný pouze určitý *seznam metod* (někdy se také říká *protokol*). Odtud plyně, že stačí, když budou implementovat stejně rozhraní. ♦

11.1 Deklarace a implementace rozhraní

Deklarace rozhraní

Deklarace rozhraní se podobá deklaraci třídy, ve které ale zapisujeme pouze hlavičky metod. Jeho syntax je

deklarace rozhraní:

- `publicnep interface identifikátor { seznam_metodnep }`

Význam specifikátoru `public` je stejný jako v deklaraci třídy – označuje veřejně přístupné rozhraní, tedy rozhraní, které smí používat kterákoli část kteréhokoli programu. Podobně jako veřejně přístupná třída musí být i veřejně přístupné rozhraní uloženo v souboru se stejným jménem jako je jméno tohoto rozhraní. Každý soubor může obsahovat nejvýše jedno veřejné rozhraní nebo veřejnou třídu.

Identifikátor je jméno rozhraní. *Seznam metod* obsahuje hlavičky metod ve tvaru

metoda v seznamu metod:

- `přístupnep typ identifikátor(parametrynep) výjimkynep ;`

Význam jednotlivých součástí tohoto popisu je stejný jako v případě metod deklarovaných v těle třídy; v rozhraní však nelze použít modifikátory `static` a `final`. Modifikátor `abstract` nemá význam, neboť všechny metody rozhraní jsou automaticky abstraktní.

Seznam metod může být i prázdný. Takovéto rozhraní slouží vlastně jen k „označení“ třídy, k přidělení dalšího typu.

⁶⁰ `Serializable` je ve skutečnosti jedno ze standardních knihovních rozhraní.

Implementace rozhraní

Rozhraní se nedědí, rozhraní se implementuje. To je sice na první pohled pouze slovní rozdíl, nicméně v Javě je vyjádřen použitím jiného klíčového slova než dědění – `implements`. Syntax jeho použití je

implementace rozhraní:

- **implements seznam_rozhrani**

Seznam_rozhrani, to jsou identifikátory jednotlivých rozhraní, oddělené čárkami. Ze syntaktického popisu deklarace třídy víme, že informace o implementovaných rozhraních se zapisuje za informaci o předkoví (pokud je v deklaraci uvedena) a před tělo třídy.

Jestliže nějaká třída implementuje dané rozhraní, musí být abstraktní nebo musí implementovat všechny jeho metody. Implementaci rozhraní pak dědí odvozené třídy.

11.2 Použití rozhraní

Na rozhraní v Javě se můžeme dívat jako na „lehkou“ náhradu vícenásobné dědičnosti, známé z C++. Třída sice nemůže mít více předků, může ale implementovat několik rozhraní. Tím dosáhneme toho, že její instance jsou několika různými typů.

Ukazuje se, že používání rozhraní patří k oblíbeným trikům jazyka Java (aniž bychom slibovali, že se k nim v této knize vrátíme).

- Iterátory na standardních kontejnerech jsou objekty, které implementují rozhraní `Iterator`. Mohou patřit do různých tříd, lze s nimi ale zacházet jednotným způsobem.
- Chceme-li implementovat možnost ukládání objektů do souborů a jejich následného obnovení (tzv. persistenci), využijeme rozhraní `java.io.Serializable`.
- Chceme-li implementovat možnost vytváření identických kopií („klonování“) objektů pomocí metody `clone()` zděděné od třídy `Object`, musíme implementovat rozhraní `java.lang.Cloneable`.
- Jednou z možností, jak vytvořit třídu, která bude představovat samostatné vlákno (paralelní tok výpočtu) v programu, je implementovat rozhraní `java.lang.Runnable`.
- Rozhraní `java.lang.Comparable` obsahuje metodu `compareTo()`, jež umožňuje navzájem porovnávat dvě instance – zjišťovat, zda je jedna z nich menší nebo větší než druhá nebo zda jsou si rovny.

Příklad: Kontrola konzistence objektů

Podivejme se na rozsáhlejší příklad. Píšeme program, ve kterém používáme jak naše oblíbené grafické objekty, které jsme navrhli v kapitole 9, tak i znakové řetězce. Všechny tyto objekty se mohou dostat do nekonzistentního stavu, tj. jejich vnitřní data mohou být nějak poškozena.

Před použitím je proto nutné jejich stav zkontovalovat. V hlavní třídě programu k tomu napíšeme metodu `kontrolaVsech()`, jejímž parametrem bude kontejner obsahující objekty, jejichž stav chceme prověřit.

Každý z objektů bude mít svoji metodu `zkontroluj()`, která prověří konzistenci jeho vnitřních dat. Abychom mohli se vsemi objekty zácházet jednotným způsobem, potřebujeme, aby byly stejného typu. Kdybychom chtěli použít dědičnost, museli bychom nejprve definovat společného předka, abstraktní třídu `Kontrolovatelná`, která by obsahovala jedinou metodu `zkontroluj()`. Od této třídy bychom pak odvodili jak třídu `Bod`, tak i třídu `Retezec`. Tím by vznikla poměrně rozsáhlá hierarchie, která by obsahovala grafické objekty vedle řetězců a bůhví čeho ještě.

Použijeme proto řešení založené na rozhraní. Nejprve deklarujeme veřejně přístupné rozhraní `Kontrola`, které umístíme do balíku `kontrola`.

```
/* Soubor Kap11\01\kontrola\Kontrola.java */
package kontrola;

public interface Kontrola {
    public boolean zkонтrolуй();
}
```

Pak upravíme deklaraci tříd z balíku `grafika`. Ve třídě `GO` deklarujeme implementaci rozhraní `Kontrola` a metodu `zkontroluj()`, která zkontovaluje, zda je barva vyjádřena nezáporným číslem:

```
/* Soubor Kap11\01\grafika\GO.java */
package grafika;

import kontrola.*;

public abstract class GO implements Kontrola {
    private int barva;

    public int getBarva() { return barva; }
    public void setBarva(int _barva) { barva = _barva; }
    public GO() {}
    public GO(int ..barva) { setBarva(_barva); }
    public boolean zkонтrolуй() { return barva >= 0; }
    public abstract void nakresli();
}
```

Ve třídách odvozených od třídy GO není třeba deklarovat, že implementují rozhraní Kontrola; stačí, když ho implementuje společný předek. Ve většině z nich však budeme muset předefinovat metodu zkонтroluj(), neboť kontrola správnosti údajů v nich bude složitější než v obecném grafickém objektu. V instanci třídy Bod budeme např. navíc kontrolovat, zda jsou obě souřadnice kladné:

```
public class Bod extends GO {  
    // Ostatní složky a metody jsou stejné jako v kap. 9  
    public boolean zkонтroluj() {  
        return super.zkонтroluj() && (x > 0) && (y > 0);  
    }  
}
```

Ve třídě ZakrouzkovanýBod budeme kontrolovat pouze konzistenci bodu, neboť tato třída má navíc jen jednu metodu. Postačí tedy volání metody zkонтroluj() předka, takže metodu třídy Bod zde nebude definovávat. V kružnici budeme kontrolovat, zda je v pořádku střed, který je instancí třídy Bod, a zda je poloměr kladný:

```
public class Kružnice extends GO {  
    // Ostatní složky a metody jsou stejné jako v kap. 9  
    public boolean zkонтroluj() {  
        return super.zkонтroluj() &&  
            stred.zkонтroluj() && (r > 0);  
    }  
}
```

nakonec ve třídě Usecka budeme kontrolovat, zda jsou v pořádku oba krajní body.

```
public class Usecka extends GO {  
    // Ostatní složky a metody jsou stejné jako v kap. 9  
    public boolean zkонтroluj() {  
        return super.zkонтroluj() &&  
            a.zkонтroluj() && b.zkонтroluj();  
    }  
}
```

Pak definujeme třídu Retezec. Nám ve skutečnosti nejde o nic jiného, než přidat k existující implementaci řetězců v Javě nějaké své metody, ale protože jak String tak StringBuffer jsou finální třídy, nezbývá, než použít skládání a deklarovat třídu, která využije služeb třídy StringBuffer, ale nebude jejím potomkem. Také tato třída implementuje rozhraní Kontrola.Kontrola. Ukažeme si alespoň část implementace:

```
/* Soubor Kap11\01\retezec\Retezec.java */  
package retezec;
```

```

import kontrola.*;
public class Retezec implements Kontrola {
    StringBuffer str = new StringBuffer("");
    public Retezec() { }
    public boolean zkонтroluj() { // Kontrola
        return str != null;
    }
    String getString() { return str.toString(); }
    public void pridej(String s) {
        str.append(s);
    }
}

```

Kontrola zde spočívá ve zjištění, zda instance obsahuje platný odkaz na StringBuffer.

Nyní se podíváme na využití v metodě kontrolaVsech, která je součástí třídy Test a která očekává kontejner typu ArrayList, obsahující instance implementující rozhraní Kontrola.

```

import grafika.*;
import kontrola.*;
import retezec.*;
import java.util.*;

public class Test {
    ArrayList seznam = new ArrayList();
    // Zkontroluj obsah kontejneru ...
    public boolean kontrolaVsech(ArrayList s){
        try{// ... pomocí metody kontrola() z rozhraní Kontrola
            for(Iterator i = s.iterator(); i.hasNext();)
            {
                Kontrola k = (Kontrola)i.next();
                System.out.println(k);
                if(!k.zkонтroluj()) return false;
            }
            return true;
        } /* Konec bloku try */
        catch(Exception e){
            System.out.println("Kontejner obsahuje " +
                               "nekontrolovatelný objekt")
            e.printStackTrace();
            return false;
        } /* Konec handleru */
        /* Konec metody kontrolaVsech() */
        // ... a další metody
    }
}

```

V metodě kontrolaVsech() projdeme všechny objekty v kontejneru pomocí iterátoru; s tím jsme se setkali už v příkladu v kapitole 9.5. S

objekty z kontejneru zacházíme prostřednictvím odkazu na rozhraní `Kontrola`. Protože metoda `i.next()` vrací odkaz na `Object`, musíme ho přetypovat na odkaz na `Kontrola`. Správnost tohoto přetypování nemůže překladač zkontovalovat v době překladu, a proto se kontroluje až za běhu; pokud není možné, tj. pokud objekt vrácený iterátorem neimplementuje požadované rozhraní, vznikne výjimka. Odpovídající handler vypíše upozornění a ukončí metodu tím, že vrátí `false`.

Pokud je přetypování možné, zavolá se pro daný objekt metoda `zkontroluj()`. Metoda `kontrolaVsech()` skončí, když projde všechny prvky kontejneru nebo když narazí na nekontrolovatelný objekt nebo na první nekonzistentní objekt (objekt, u kterého metoda `zkontroluj()` vrátí `false`).

- JDK 5 Poznamenejme, že v JDK 5 lze kontejner `ArrayList` (nebo kterýkoli jiný) parametrizovat typem `Kontrola`. Pak odpadne možnost, že do kontejneru omylem uložíme nekontrolovatelný objekt, a tím i možnost výjimky.



Úplný zdrojový text tohoto příkladu najdete na WWW v adresáři `Kap11\01` a v jeho podadresářích odpovídajících uvedeným balíkům.

Klonování objektů⁶¹

Čas od času potřebujeme vytvořit nový objekt, který bude kopii objektu již existujícího. V Javě neexistuje analogie kopírovacího konstruktora z C++; můžeme však použít metodu `clone()`, kterou najdeme již ve třídě `Object` a kterou proto dědí všechny třídy.

Aby věci ale nebyly tak jednoduché, je tato metoda ve třídě `Object` chráněná (`protected`) a abychom ji mohli použít v některé z odvozených tříd, musí tato třída implementovat rozhraní `java.lang.Cloneable`. Toto rozhraní je prázdne, slouží pouze k označení tříd, v nichž je klonování dovoleno.⁶²

Metoda `clone()` třídy `Object` má hlavičku

```
protected Object clone()
```

a vytvoří binární kopii instance *odvozené* třídy (tj. vyhradí paměť pro novou instanci a okopíruje do ní bit po bitu data z kopírované instance). Občas je to přesně to, co potřebujeme. Problémy ale mohou nastat např. při klonování objektů, které jako složky obsahují odkazy na jiné

⁶¹ Na rozdíl od klonování živých organismů nezpůsobuje klonování objektů v programech žádné morální ani politické problémy.

⁶² Zavoláme-li metodu `clone()` ve třídě, která toto rozhraní neimplementuje, vznikne výjimka `CloneNotSupportedException`.

objekty – metoda `clone()` třídy `Object` totiž okopíruje odkazy, nebude ale klonovat objekty, na něž tyto složky odkazují.⁶³

Rozhodneme-li se klonovat instance určité třídy, musíme v ní tedy

- implementovat rozhraní `Cloneable()`,
- předefinovat metodu `clone()` jako veřejnou a zavolat v ní zděděnou verzi, tj. `super.clone()`,
- v případě potřeby se postarat o klonování objektů, na něž odkazují složky právě klonované instance.

Příklad: Ještě jednou se vrátíme k hierarchii grafických objektů a ke třídě `Re-
tezec`, které jsme používali v předchozím oddílu. Grafické objekty budeme chtít klonovat. To znamená, že jejich společný předek, třída `G0`, musí implementovat rozhraní `Cloneable` a předefinovat metodu `clone()` jako veřejně přístupnou, abychom ji mohli volat prostřednictvím odkazu na `G0`:

```
public abstract class G0 implements Kontrola, Cloneable {  
    // Ostatní metody a složky jako v Kap11\01  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

V odvozených třídách `Bod` a `ZakrouzkovanyBod` nemusíme metodu `clone()` předefinovávat, neboť plně vyhovuje zděděná verze (stačí nám binární kopie vzoru). Ovšem ve třídách `Kruznice` a `Usecka` budeme muset tuto metodu změnit, neboť chceme, aby klon kružnice obsahoval svůj vlastní střed a aby klon úsečky obsahoval své vlastní instance krajních bodů. Musíme proto metodu `clone()` definovat tak, aby se o to postarala (viz obr. 11.1 a 11.2):

```
public class Kruznice extends G0 {  
    // Ostatní metody a složky jako v Kap11\01  
    public Object clone() throws CloneNotSupportedException {  
        Kruznice k = (Kruznice) super.clone();  
        k.stred = (Bod)stred.clone(); // Vytvoř klon středu  
        return k;  
    }  
}  
  
public class Usecka extends G0 {  
    // Ostatní metody a složky jako v Kap11\01  
    public Object clone() throws CloneNotSupportedException {  
        Usecka u = (Usecka) super.clone();  
        u.a = (Bod)a.clone(); // Vytvoř klony krajních
```

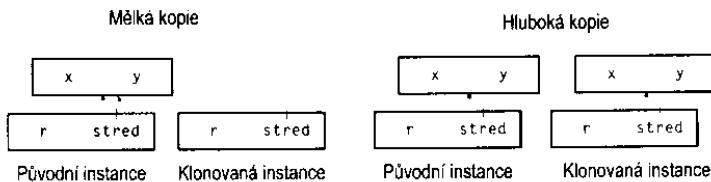
⁶³ Říkáme, že vytvoří *mělkou* kopii. Chceme-li vytvořit *hlubokou* kopii, tj. klonovat i objekty, na něž odkazují složky klonované instance, musíme předefinovat metodu `clone()`.

```

        u.b = (Bod)b.clone(); // bodů
        return u;
    }
}

```

Obě tyto metody nejprve pomocí metody předka (tedy vlastně metody třídy `Object`) vytvoří mělkou kopii klonované instance. Pak vytvoří klony středu, resp. krajních bodů, a takto upravený klon původní instance vrátí.



Obr. 11.1 Metoda `clone()`, zděděná po třídě `Object`, vytvoří mělkou kopii kružnice (vlevo); my ale potřebujeme hlubokou kopii (vpravo)



Metoda `clone()` je poprvé deklarována ve třídě `Object`, a proto musí vracet odkaz na `Object`. To znamená, že vrácenou hodnotu musíme vždy přetypovat na skutečný typ instance.⁶⁴



Na WWW v adresáři Kapitola02 najdete zdrojové texty všech tříd z tohoto příkladu. Najdete tam také třídu `Test`, která v metodě `beh()` naplní instanci seznam třídy `ArrayList` několika grafickými objekty a několika instancemi třídy `Retezec`, jež neimplementuje rozhraní `Cloneable`.

Tento kontejner se pokusíme okopírovat. Nepoužijeme metodu `clone()` třídy `ArrayList`, neboť ta nevytváří hlubokou kopii – neklonuje uložené objekty. (Do tohoto kontejneru lze ukládat i instance neklonovatelných tříd, jako je např. `Retezec`.) Proto si kopírování naprogramujeme sami. V cyklu pomocí iterátoru projdeme seznam, zkusíme vytvořit kopii každého z uložených objektů, a pokud se to podaří, vytvořený klon uložíme do nového seznamu.

```

ArrayList seznamml = new ArrayList(); // Nový seznam
for(Iterator i = seznam.iterator(); i.hasNext();)
{
    try {
        g = (GO)i.next();           // Vezmi další objekt
        g = (GO)g.clone();          // Klonuj ho
        g.setBarva(123);            // Změň barvu kopie
        seznamml.add(g);            // Ulož ji do nového seznamu
    }                                // Když se něco nepodaří,
}

```

⁶⁴ V JDK 5 může vracet hodnotu aktuálního typu.

```
        catch(Exception e) {}           // nic se neděje
    }
```

Zde se prostě pokusíme objekt vyňatý z kontejneru přetypovat na `GO` a zavolat pro něj metodu `clone()`. Pokud se některá z těchto operací nepodaří, vznikne výjimka a řízení přejde do handleru. Ten je prázdny, neboť jediné, co potřebujeme, je jít dál – zpracovat další prvek. Výsledkem bude, že kontejner `seznam` bude obsahovat pouze klonovatelné grafické objekty. ♦



Všimněte si triku založeného na rozhraní `Cloneable`. Toto rozhraní slouží jako příznak, který říká „instance této třídy a jejích potomků lze klonovat“. Metodu `clone()` totiž dědí všechny javské třídy; na druhé straně určitě se v řadě programů vyskytnou třídy, u nichž nebudeme chtít klonování z nejrůznějších důvodů povolit. Proto musí každá třída, kterou chceme klonovat, implementovat rozhraní `Cloneable`. Už víme, že pokud ho **neimplementuje**, způsobí volání metody `clone()` výjimku `CloneNotSupportedException`; tak lze klonování zabránit.

Anonymní třída implementující rozhraní

Občas se stane, že v programu potřebujeme jedinou instanci třídy, která implementuje určité rozhraní; např. v kapitole 13 se uvidíme, že k přenosu informace o události potřebujeme instance třídy, která implementuje rozhraní

```
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

Mohli bychom samozřejmě deklarovat třídu, která toto rozhraní implementuje, a použít její instanci. Protože s ní však budeme zacházet pouze pomocí rozhraní, používá se instance anonymní třídy, vytvořená podle následující syntaxe:

vytvoření instance anonymní třídy implementující rozhraní:

- `new jméno_rozhraní() { tělo_třídy }`

Jméno rozhraní použijeme jako konstruktor bez parametrů a za něj vypíšeme do složených závorek tělo třídy – tedy deklarace metod. Instanci třídy implementující rozhraní `ActionListener` tedy můžeme vytvořit takto:

```
ActionListener a = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Něco dělej...
    }
}
```

12 Parametrizované třídy a metody

5

Parametrizované – v literatuře o jazyku Java se také říká *generické* – třídy a metody najdeme až v JDK 5. O jejich významu jsme hovořili už ve 4. kapitole; nyní se na ně podíváme podrobněji. Jde ovšem o značně rozsáhlé téma, takže si řekneme jen nejnutnější informace. Podrobnosti najdete v dokumentaci k JDK, v [11] nebo v [12].

12.1 Parametrizované třídy

Deklarace parametrizované třídy se od deklarace neparametrizované třídy liší jen v několika drobnostech – především obsahuje navíc specifikaci typových parametrů:

deklarace parametrizované třídy:

- *modifikátory_{nep} class <typové_parametr> identifikátor
předek_{nep} rozhrani_{nep}
{
tělo_třídy
}*

Význam *modifikátorů*, *identifikátoru*, *předka* a *rozhrani* je stejný jako u neparametrizované třídy. Za jménem třídy následují v lomených závorkách *typové parametry*. V nejjednodušším případě to je jeden nebo několik identifikátorů oddělených čárkami. O těchto identifikátořech hovoříme jako o *formálních typech*.

Formální typy můžeme použít v *těle třídy* při specifikaci typu datových složek, návratových typů metod atd.

Příklad: Ve 4. kapitole jsme pro účely počítání slov deklarovali třídu *Seznam*, jejíž prvky obsahovaly odkazy na typ *String* a informace o počtu výskytů. Takovýto seznam jde použít opravdu jen pro práci s řetězci, s ničím jiným. Přitom většina metod třídy prvků – ale i třídy seznamu – na typu uložené hodnoty nijak nezávisí.

Nabízí se samozřejmě možnost deklarovat prvky seznamu tak, že budou obsahovat odkazy na typ *Object*, ale pak zase přijdeme o typovou kontrolu a může se stát, že do seznamu uložíme omylem něco jiného, než chceme. Využijeme tedy možnosti parametrizace:

```

class Prvek<T>
{
    T data = null;           // Data uložená v prvku
    int pocet = 0;           // Počet výskytů
    Prvek<T> dalsi = null;  // Odkaz na další prvek seznamu
    Prvek(){}
    Prvek(T s) { data = s; pocet = 1; }
    int obsahuje(T s) { return s.equals(data) ? pocet : 0; }
    void setDalsi(Prvek<T> q){ dalsi = q; }
    Prvek<T> getDalsi(){ return dalsi; }
    T getData(){ return data; }
    void zvetsiPocet(){ pocet++; }
    int getPocet(){ return pocet; }
}

```

Identifikátor `T` představuje formální typ. V těle pak deklarujeme datovou složku typu `T`, metody s parametrem typu `T`, metody vracející `T` apod.

Podobně parametrizujeme i samotnou třídu seznam.

```

class Seznam<T>
{
    Prvek<T> hlava = null;
    void pridej(T data)
    {
        if(hlava == null)
        {
            hlava = new Prvek<T>(data);
        }
        else
        {
            Prvek<T> p = hlava, q = null;
            while(p != null)
            {
                if(p.obsahuje(data) > 0)
                {
                    p.zvetsiPocet();
                    return;
                }
                else
                {
                    q = p;
                    p = p.getDalsi();
                } // konec if
            } // konec while
            q.setDalsi(new Prvek<T>(data));
        }
    }
    void vypis()
    {
        Prvek<T> p = hlava;
        while(p != null)

```

```

        {
            System.out.println(p.getData() + " " + p.getPocet());
            p = p.getDalsi();
        }
    }

    void vyprazdni()
    {
        hlava = null;
    }
}

```

Nyní můžeme v metodě `beh()` třídy `Analyzer` deklarovat instanci

```
Seznam<String> slovnik = new Seznam<String>();
```

Do tohoto seznamu lze ukládat pouze instance typu `String`. Mohli bychom tam ukládat i instance typů odvozených od třídy `String`, kdyby ovšem třída `String` nebyla konečná. Metoda `pridej()` má parametr typu `T`, a protože v případě instance `slovnik` typový parametr `T` znamená `String`, bude překladač důsledně kontrolovat, zda tuto metodu voláme s parametrem správného typu.

Třída `Seznam<T>` využívá parametrizovanou třídu `Prvek<T>`. To znamená, že instance `slovnik`, která je typu `Seznam<String>`, pracuje s instancemi typu `Prvek<String>`. Metody, u nichž jsme deklarovali návratový typ `T`, budou vracet hodnoty typu `String`.

Úplný zdrojový text tohoto příkladu včetně třídy `Analyzer` najdete na WWW v souboru Kap12\01\Analyzer.java. ♦



12.2 Parametrizovaná rozhraní

Podobně jako třídy můžeme parametrizovat i rozhraní. Typové parametry lze použít v deklarácích formálních parametrů a návratových typů metod, které toto rozhraní obsahuje.

Například rozhraní `java.lang.Comparable` je v knihovně Javy 5 deklarováno takto:

```
public interface Comparable<T>
{
    int compareTo(T obj);
}
```

12.3 Parametrizované metody

Java 5 umožňuje parametrizovat nejen celé třídy, ale i jednotlivé metody tříd, které samy o sobě nejsou parametrizovány. Syntax deklarace parametrizované metody lze popsat takto:

deklarace parametrizované metody:

- *modifikátory_{nep} <typové_parametry> typ jméno tělo*

I zde je ve srovnání s deklarací neparametrizované metody navíc pouze specifikace typových parametrů v lomených závorkách. Má stejný tvar a stejný význam jako v případě deklarace parametrizované třídy a zapisujeme ji *před* specifikaci návratového typu. Formální typy zde uvedené lze použít ke specifikaci typu parametrů a návratových typů metod a k deklaraci lokálních proměnných v *těle* metody.

Tělo je buď blok obsahující operace, které metoda provádí, nebo středník (to v případě abstraktních metod).

Příklad: Deklarujeme si třídu Pomoc, jež bude obsahovat pomocné metody pro práci s poli. První z těchto metod bude vracet prvek ležící uprostřed pole. V případě, že má pole sudý počet prvků, a tedy „uprostřed“ leží dva prvky, vrátíme ten z nich, který má menší index. Přitom předpokládáme, že skutečným parametrem bude pole objektového typu ne-nulové délky.

```
public class Pomoc
{
    // Najde prvek ležící uprostřed pole typu T
    public static <T> T median(T[] x)
    {
        if(x == null || x.length == 0)
            throw new IllegalArgumentException();
        return x[x.length / 2];
    }
    // ... a další pomocné metody ...
}
```

V těle této metody nejprve ověříme, že skutečným parametrem není null ani pole nulové délky; není-li tato podmínka splněna, vyvoláme výjimku. Pak vrátíme prostřední prvek.

Nyní můžeme tuto metodu použít:

```
Integer[] A = {5, 4, 3, 2, 1}; // Automatické zabalení
System.out.println(Pomoc.median(A));
```

Poslední příkaz vypíše 3. Poznamenejme, že při inicializaci pole A jsme využili automatické zabalení.



Všimněte si, že při volání metody Pomoc.median() jsme nikde neuváděli skutečnou hodnotu typového parametru T. Překladač si ho totiž téměř vždy dokáže odvodit z typu skutečného parametru. Nic nám ale nebrání tento parametr uvést v lomených závorkách před jménem metody. To znamená, že předchozí příkaz může vypadat také takto:

```
System.out.println(Pomoc.<Integer>median(A)); ♦
```

12.4 Meze typových parametrů

Zkusme nyní napsat metodu, která seřadí prvky v předaném poli podle velikosti (tedy seřídí dané pole).

Abychom to mohli udělat, musí platit, že libovolné dva prvky je možno porovnat. Protože jde o prvky objektového typu, nikoli o prvky základních typů, nelze k jejich porovnávání použít operátory < a >, musíme užít vhodnou metodu.⁶⁵ Jednou z možností je, že typ T implementuje standardní rozhraní `java.lang.Comparable`, jež obsahuje jedinou metodu `compareTo()`. Výraz `a.compareTo(b)` má hodnotu zápornou, 0 nebo kladnou podle toho, zda je a menší než b, rovno b nebo větší než b.

Pro seřazení prvků pole použijeme algoritmus třídění výběrem, s nímž jsme se seznámili v první kapitole, a metodu nazveme v duchu nejlepších programátorských tradic `sort()`. Náš první pokus ovšem překládač odmítne:



```
public static <T extends Comparable> void sort(T[] x)
{
    for(int i = 0; i < x.length; i++)
    {
        int min = i;
        // Najdi index nejmenšího prvku v úseku od i+1 do konce
        for(int j = i+1; j < x.length; j++)
            if(x[j].compareTo(x[min]) < 0) min = j;
        // a prohoď ho s prvním v daném úseku
        if(min != i)
        {
            T a = x[i];
            x[i] = x[j];
            x[j] = a;
        }
    }
}
```



Z chybového hlášení se dozvímme, že typ T neobsahuje metodu `compareTo()`. Problém je v tom, že překladač s typovým parametrem T zachází jako s typem `Object`, neboť o něm nic bližšího neví, – a typ `Object` opravdu metodu `compareTo()` neobsahuje.

Java 5 nabízí řešení v podobě tzv. *mezí* (omezení) typových parametrů, které připojíme za identifikátor typového parametru v deklaraci parametrizované třídy nebo metody.

⁶⁵ Přitom na chvíli zapomeneme, že ve třídě `java.util.Arrays` je k dispozici několik verzí statické metody `sort()`.

deklarace mezi typového parametru:

- **extends jméno_1**
- **extends jméno_1 \$ jméno_2**
- ...

Jméno_1, stejně jako *jméno_2* zde představuje jméno třídy nebo rozhraní.

První možnost – za jméno typového parametru připíšeme klíčové slovo `extends` a za ně jméno třídy nebo rozhraní – znamená, že jako skutečný parametr smíme použít jen uvedenou třídu nebo třídu od ní odvozenou, nebo třídu implementující uvedené rozhraní nebo rozhraní od něj odvozené.

Druhá možnost v syntaktickém popisu říká, že chceme-li zde uvést několik rozhraní, oddělíme je znakem `$`.



Všimněte si, že zde používáme klíčové slovo `extends` jak pro třídy, tak i pro rozhraní.

Vratme se nyní k metodě `sort()`. Vše, co musíme udělat, aby byl překladač spokojen a naši metodu přeložil, je zadat správnou mez v deklaraci:

```
public static <T extends Comparable> void sort(T[] x)
{
    // Tělo metody zůstane stejné
}
```

Je-li `A` pole typu `Integer` z předešlého oddílu, můžeme nyní napsat

```
Pomoc.sort(A);
for(Integer n: A)System.out.print(n+" ");
```

a program vypíše

1 2 3 4 5



Zdrojový text třídy `Pomoc` obsahující mj. metodu `sort` najdete na WWW v souboru `Kap12\02\Pomoc.java`. Soubor `Program.java` v tomtéž adresáři ukazuje její volání. ♦

12.5 Pohled pod pokličku

Pro typové parametry platí řada omezení. Abychom jim snáze porozuměli, musíme si povědět, jak jsou implementovány.

Především je třeba vědět, že v bajtovém kódu, který vznikne překladem parametrisované třídy nebo metody, již žádné parametrisované třídy ani metody nejsou.



- Překladač z deklarace třídy odstraní deklaraci typových parametrů, tedy lomené závorky a identifikátory typových parametrů s případnými mezemi.
- Všechna použití typových parametrů bez mezi nahradí odpovídajícím použitím typu `Object`.
- Všechna použití typových parametrů s mezemi nahradí odpovídajícím použitím prvního omezujícího typu.

Tomuto procesu se říká *vymazání typů* (type erasure); vznikne tak tzv. *surový typ*.

Omezení

Odtud plynou například následující omezení:



- Jako skutečné typové parametry lze použít pouze objektové typy. Místo primativních typů musíme použít odpovídající obalové typy. To znamená, že nelze použít `Prvek<int>`, místo toho musíme použít `Prvek<Integer>`.
- Parametrizovaná třída nesmí být odvozena od třídy `Throwable`. A tedy ji nelze uvést ani v příkazu `throw`, ani v příkazu `catch`.
- Nelze deklarovat pole parametrizovaných typů. Místo pole používáme – pokud to jde – instanci třídy `java.util.ArrayList`.
- Nelze volat konstruktor formálního typu, tj. nelze vytvořit instanci formálního typu.
- Typy zavedené typovými parametry nelze použít v deklaraci statických datových složek nebo statických metod.

Podívejme se na příklad, který shrne většinu uvedených omezení.

```
class Pokus<T> extends Throwable // Nelze
{
    T t;
    private static T pocet() /* ... */ // Nelze
    public Pokus() {
        t = new T(); // Nelze
    }
    public metoda()
    {
        Pokus<String>[] pok; // Nelze
    }
}
```

Parametrizované typu lze používat i bez parametrů. To znamená, že i v Javě 5 můžeme napsat

```
ArrayList al = new ArrayList(); // Lze i v JDK 5
```

Dědění a parametrizované typy

Podívejme se opět na naše oblíbené třídy grafických objektů. Nic nám samozřejmě nebrání deklarovat si v programu kontejner na grafické objekty, např. typu `ArrayList<GO>`. Stejně dobře však můžeme deklarovat kontejner `ArrayList<Bod>`, kde třída `Bod` je potomkem třídy `GO`.

Nabízí se otázka: Jaký je vztah mezi třídami `ArrayList<GO>` a `ArrayList<Bod>`?



Odpověď vás možná překvapí: *Žádný*.

Z toho, že třída `Bod` je potomkem `GO`, neplyne, že by třída `ArrayList<Bod>` byla potomkem `ArrayList<GO>`. Obecně ze vztahu mezi parametry neplyne žádný vztah mezi parametrizovanými třídami. To znamená, že nesmíme např. napsat

```
ArrayList<GO> alg = new ArrayList<Bod>(); // Nelze
```

Na druhé straně ale platí, že parametrizovaný typ je podtypem typu bez parametrů (surového typu). To znamená, že můžeme napsat

```
ArrayList al = new ArrayList<Bod>(); // To lze
```

nebo dokonce

```
ArrayList<Bod>[] al = new ArrayList[100]; // To také
```

Zříkáme se tím ovšem typové kontroly.

12.6 Zástupný typ (žolík)

Často budeme potřebovat metodu, jejímž parametrem bude instance parametrizovaného typu. Přitom se ovšem dostaneme do potíží, neboť – jak už víme – dvě parametrizace téhož surového typu se v JDK 5 chovají jako dva různé typy.

Představte si, že chceme napsat metodu pro výpis obsahu kolekce. Řešení založené na neparametrizovaných typech (a použitelné i ve starších verzích JDK) je jednoduché využijeme skutečnosti, že všechny třídy kolekcí implementují rozhraní `java.util.Collection`, jež obsahuje metodu vracející iterátor:

```
public static void tisk(java.util.Collection c)
{
    java.util.Iterator it = c.iterator(); // Získáme iterátor
    while(it.hasNext()) // a projdeme kolekcí
        System.out.println(it.next()); // prvek po prvku
}
```

Rozhraní `Collection` je v JDK 5 také parametrizováno. Rádi bychom toho využili, neboť pak za nás bude překladač kontrolovat, zda do

nich vkládáme hodnoty správného typu. Přitom se ale dostaneme do potíží: Jak vyjádřit, že tato metoda má jako skutečný parametr akceptovat jakýkoli kontejner s jakýmkoli parametrem?

Nejjednodušší možnost, která nás napadne, totiž využít typ `Object` a napsat

```
public static void tisk(Collection<Object> c) // Nelze
    /* totéž co předtím */ i
```

nebude fungovat – o tom jsme hovořili v předchozím oddílu.

Proto nabízí JDK 5 možnost použít na místě formálního typu *žolík* (wildcard, též „zástupný typ“) – znak ?. Metoda `tisk()` s použitím žolíku bude vypadat takto:

```
public static void tisk(Collection<?> c)
{
    for(Object x: c)
        System.out.println(x);
}
```

Poznamenejme, že i pro žolík můžeme specifikovat meze. To znamená, že můžeme napsat např.

```
public static void tisk(Collection<? extends G0> c)
{ ... }
```

Tím říkáme, že skutečným parametrem metody `tisk()` smí výt instancce jakékoli třídy, která je instancí třídy `G0` nebo jejím potomkem. Užijeme-li místo klíčového slova `extends` slovo `super`, např.

```
public static void tisk(Collection<? super Bod> c)
{ ... }
```

předepíšeme tím, že skutečným parametrem musí být instance třídy `Bod` nebo jakéhokoli jejího předka.

 Pozor: Žolík, tedy znak ?, nelze použít jako jméno typu. To znamená, že ho např. nelze použít v těle metody k deklaraci proměnné.

13 Soubory, vstupy a výstupy

Soubor a proud Java rozlišuje *soubory* (file) a *proudy* (stream).⁶⁶ *Soubor* je množina dat, která existuje na disku nebo v jiném paměťovém médiu víceméně nezávisle na běhu programu a se kterou zacházíme jako s logickým celkem. Na druhé straně *proud* si vytváří program jako nástroj pro zápis do souboru nebo pro čtení z něj. Můžeme se na něj dívat jako na cestu, po které proudí data mezi souborem a programem.

java.io Nástroje pro práci se soubory a vstupními a výstupními proudy jsou v Javě součástí knihovny, balíku `java.io`, kde najdeme více než 40 tříd. To může být pro začátečníka nejen nepřehledné, ale i značně znechucující. Upřímně řečeno, neznám nikoho, kdo by se o řešení vstupů a výstupů v Javě vyjadřoval s nadšením, jak ale časem zjistíte, není to tak zlé, jak to na první pohled vypadá. Ve skutečnosti jde o mocný nástroj, který umožňuje nejen práci se soubory, ale přenosu dat rourami (pipe), výstup do paměti atd.

V této kapitole si povíme jen základní informace; podrobný výklad najdete např. v [2] nebo [3].

java.nio Vedle toho nabízí Java počínaje JDK 1.4 i nové prostředky pro vstupu a výstupy založené na tzv. kanálech. Najdeme je v balíku `java.nio`. Jsou rychlejší než prostředky z balíku `java.io`, práce s nimi je však o něco složitější a méně přehledná. V této knize s jimi nebudeme zabývat; informace o nich lze najít v dokumentaci nebo např. v knize [12].

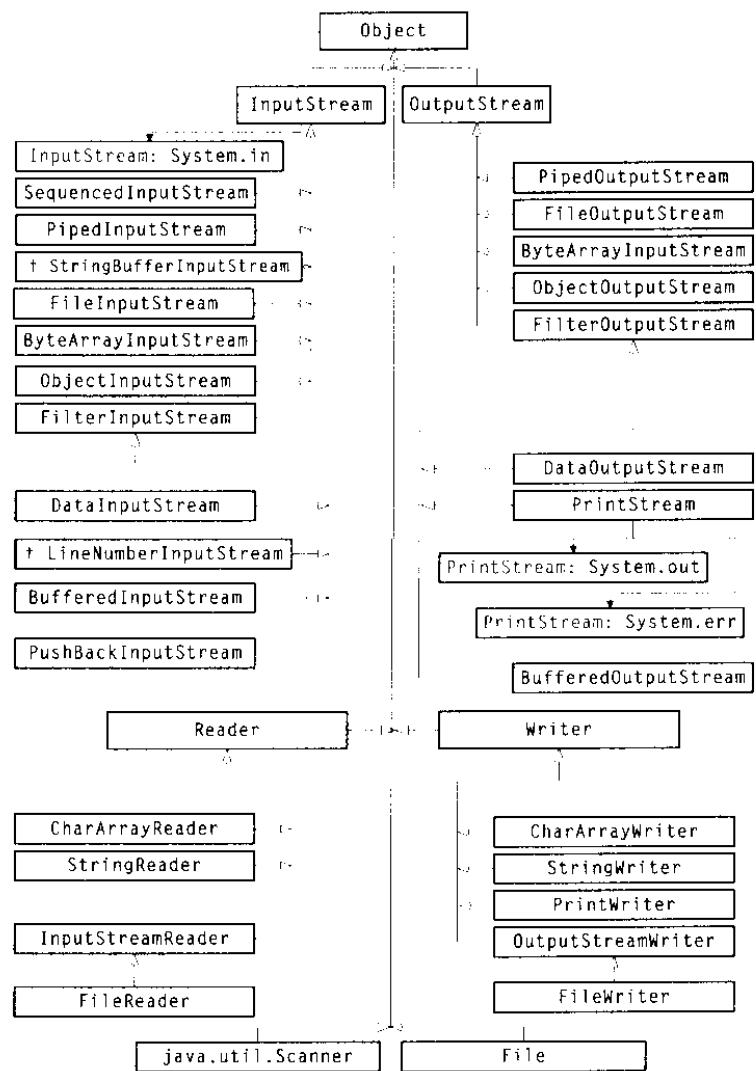
Obrázek 13.1 ukazuje hierarchii základních tříd a jejich standardní instance. Podíváme-li se na něj pozorně, zjistíme, že obsahuje několik skupin tříd.

První z nich tvoří samotná třída `File`, která obsahuje nástroje pro manipulace se soubory a adresáři – umožňuje zjišťovat, zda soubor nebo adresář existuje, vytvořit ho, přejmenovat ho, zrušit ho atd.

Dále tu najdeme abstraktní třídy `InputStream` a `OutputStream`, které jsou společnými předky tříd pro vstupní a výstupní operace. (Nenechte se zmást skutečností, že diagram na obr. 13.1 naznačuje, že `System.in` je instancí třídy `InputStream`; to neznamená nic jiného, než že třída `System` obsahuje odkaz na instanci třídy `InputStream`.



⁶⁶ Podobně jako např. jazyky C a C++.



Obr. 13.1 Základní třídy pro vstupní a výstupní operace a standardní instance. Třídy, u kterých je znak †, jsou podle dokumentace „nevzhodné“ (deprecated), neboť některé z jejich metod nefungují v určitých situacích korektně. Třída Scanner je počínaje JDK 5

Ve skutečnosti ale pracuje s instancí některého z potomků; dokumentace ovšem neříká, kterého.) Tyto třídy jsou základem pro bajtově orientované proudy (tj. proudy, které pracují s jednotlivými bajty).

Instance odvozených tříd nám budou sloužit především pro práci s daty v binární podobě.

Třídy Reader a Writer jsou základem pro znakově orientované proudy, tj. proudy, které pracují se znaky v kódování UNICODE. Instance odvozených tříd nám budou sloužit především pro textové vstupy a výstupy.

Třídy odvozené od `InputStream` obsahují tři přetížené veřejně přístupné verze metody `read()`, která se stará o čtení jednotlivých bajtů:

Metody všech proudu

- `int read() – přečte následující bajt;`
- `int read(byte[] b) – přečte skupinu bajtů a uloží je od počátku do pole b;`
- `int read(byte[] b, int off, int len) – přečte ze vstupu len bajtů a uloží je po pole b počínaje indexem off.`

Tyto metody vracejí hodnotu typu `int`, která říká, kolik bajtů se podařilo přečíst. Pokud se nepřečte ani jeden bajt, protože jsme na konci souboru, vrátí `-1`.

Podobně všechny třídy odvozené od `OutputStream` obsahují tři přetížené verze metody `write()`, která se stará o výstup jednotlivých bajtů:

- `void write(int b) – zapíše do daného proudu nejnižší bajt čísla b;`
- `void write(byte[] b) – zapíše do daného proudu všechny bajty z pole b;`
- `void write(byte[] b, int off, int len) – zapíše do daného proudu len bajtů z pole b, počínaje indexem off.`

Podobné metody najdeme i ve třídách odvozených od Reader a Writer, místo typu `byte` se v nich však používá typ `char`.

Práce s jednotlivými bajty je značně nepohodlná, a proto se bajtově orientované proudy používají především ve spolupráci s tzv. filtry – proudy odvozenými od třídy `FilterInputStream`, resp. `FilterOutputStream`. Můžeme si představovat, že tyto filtry převedou data z podoby, v níž je máme v programu, na posloupnost bajtů, a tuto posloupnost pak předají „podkladovému“ bajtově orientovanému proudu. Podobné filtry se používají i pro znakově orientované proudy.

13.1 Soubor a jeho vlastnosti

Už víme, že třída `File` obsahuje nástroje pro práci se souborem. Její konstruktor očekává znakový řetězec představující jméno souboru (příp. včetně cesty).

Tato třída obsahuje mj. statické složky `separatorChar` a `separator`, které obsahují znak oddělující v zápisu cesty v daném operačním systému jednotlivé podadresáře; ve Windows je to obrácené lomítko (`\`), v Unixu je to obyčejné lomítko (`/`). První z nich je typu `char`, druhá je typu `String`. Třída `File` nabízí metodu `exists()`, která umožňuje zjistit, zda existuje soubor se jménem udaným v konstruktoru. Chceme-li soubor s tímto jménem vytvořit, použijeme metodu `createNewFile()`; chceme-li vytvořit nový adresář, použijeme metodu `mkdir()`.

Další metody umožňují zjišťovat délku souboru, datum poslední změny, soubor přejmenovat nebo smazat a mnohé jiné operace.

Pomoci této třídy můžeme pracovat i s adresáři. Poznamenejme, že jméno aktuálního adresáře můžeme zjistit voláním `System.getProperty("user.dir")`.

Příklad:

Napíšeme metodu, která zjistí, zda existuje soubor daného jména, a pokud ne, vytvoří ho; pokud existuje, smaže jeho obsah (smaže existující soubor a vytvoří nový prázdný). Jednoduché řešení může vypadat takto:

```
public static File vytvorNovyPrazdnySoubor(String jmeno)
throws java.io.IOException
{
    File f = new File("Data.dta"); // Instance třídy File
    if(f.exists()) f.delete();    // Pokud existuje, smaž ho
    f.createNewFile();            // Vytvoř nový soubor
    return f;                    // Vráť odkaz na instanci f
}
```

Zdrojový text třídy `Test`, která obsahuje tuto metodu, najdete na WWW v souboru Kap13\01\test.java. ♦



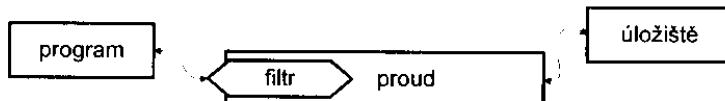
Instance třídy `File` můžeme použít jako parametr konstruktoru některých dalších tříd z hierarchie na obr. 13.1.

13.2 Čtení a zápis binárních dat

Jako „binární“ označujeme data ve tvaru, v jakém s nimi pracuje počítač. Ze 6. kapitoly např. víme, že jedno číslo typu `int` zabere v paměti 4 bajty a obsahuje hodnotu vyjádřenou ve dvojkové soustavě. Jestliže tyto 4 bajty okopírujeme do souboru, dostaneme soubor, který nebude čitelný pro člověka, ale počítač s ním bude pracovat velice snadno.

Pro čtení binárních dat ze souboru, resp. zápis těchto dat do souboru používáme třídy `FileInputStream`, resp. `FileOutputStream`. Paramet-

rem konstruktoru může být mj. řetězec představující jméno souboru nebo odkaz na instanci třídy `File`. Proud je po vytvoření automaticky otevřen; po skončení práce bychom ho měli uzavřít pomocí metody `close()`.⁶⁷ Pro čtení, resp. zápis můžeme použít metody `read()`, resp. `write()`. Ve skutečnosti ale nejsou samy o sobě mnoho platné, protože očekávají parametry typu `byte` a s ničím jiným pracovat neumějí. Proto je často používáme prostřednictvím filtrů; symbolicky to naznačuje obr. 13.2.



Obr. 13.2 Typické použití proudů v Javě. Filtr upravuje data, proud spojuje program a úložiště (soubor, rouru, síť...)



Třída `FileOutputStream` má ještě další konstruktory. Konstruktor `FileOutputStream(String jméno)` očekává jako parametr řetězec představující jméno souboru (případně včetně cesty). Pokud soubor obsahuje nějaká data, budou při otevření smazána. Konstruktor `FileOutputStream(String jméno, boolean pripojit)` má druhý parametr typu `boolean`, který určuje, zda se má při otevření obsah souboru smazat. Má-li tento parametr hodnotu `true`, obsah zůstane zachován a nová data se budou připisovat na konec.

Čtení a zápis primitivních typů

Pro práci s primitivními typy použijeme filtry `DataInputStream`, resp. `DataOutputStream`. Tyto proudy nabízejí metody jako `readInt()`, `readFloat()`, resp. `writeInt()`, `writeFloat()` a další, které umožňují číst, resp. zapisovat hodnoty primitivních typů. Parametrem konstruktoru třídy `InputStream` musí být instance třídy `InputStream` (tedy „podkladový“ vstupní proud), parametrem konstruktoru třídy `OutputStream` musí být instance třídy `OutputStream` (tedy „podkladový“ výstupní proud).

Příklad: Napíšeme metodu, jež bude mit jako parametry jméno souboru a pole typu `int` a která toto pole zapiše do daného souboru. Dále napišeme metodu, která přečte celá čísla ze souboru a uloží je do daného pole. Její zdrojový kód může vypadat takto:

```
public static void zapisPoleDoSouboru(int[] a, String jmeno)  
throws java.io.IOException  
{
```

⁶⁷ Zapomeneme-li na to, může se stát, že se část dat ztratí.

```

File f = new File(jmeno);
FileOutputStream fos = new FileOutputStream(f);
DataOutputStream vystup = new DataOutputStream(fos);
for(int i = 0; i < a.length; i++)
    vystup.writeInt(a[i]);      // V cyklu vypiš data
vystup.close();
}

```

Zde nejprve vytvoříme instanci třídy `File` se zadáným jménem. Pak vytvoříme podkladový proud. Protože chceme zapisovat binární data do souboru, použijeme `FileOutputStream`. K němu vytvoříme filtr, a protože nám jde o zápis primitivních typů, použijeme `DataOutputStream`; jeho konstruktoru předáme jako parametr odkaz na podkladový proud.

Nakonec data v cyklu vypíšeme pomocí metody `writeInt()` a pak proud uzavřeme. Poznamenejme, že pokud není soubor prázdný, jeho obsah se při otevření (při vytvoření proudu) přepíše.

Metoda pro čtení dat do pole může vypadat takto:

```

public static void nactiPozeSouboru(int[] a, String jmeno)
throws java.io.IOException
{
    final int SIZE_OF_INT = 4;
    File f = new File(jmeno);
    if(!f.exists()) throw new java.io.IOException();
    int delka = (int)f.length()/SIZE_OF_INT;
    if(delka > a.length) delka = a.length;

    FileInputStream fis = new FileInputStream(f);
    DataInputStream vstup = new DataInputStream(fis);
    for(int i = 0; i < delka; i++)
        a[i] = vstup.readInt();
}

```

Nejprve si do pojmenované konstanty uložíme velikost typu `int`; budeme ji potřebovat, až budeme zjišťovat, kolik čísel v souboru je. Pak vytvoříme instanci třídy `File` a s její pomocí zjistíme, zda požadovaný soubor existuje; pokud ne, vyvoláme výjimku. Jestliže soubor existuje, zjistíme pomocí metody `length()` jeho délku v bajtech a z ní pak počet celých čísel, které v souboru jsou. Tu porovnáme s délkou pole `a`, do kterého chceme hodnoty ukládat, a zapamatujeme si menší z nich.

Pak vytvoříme instanci podkladového proudu `FileInputStream`, napojeného na daný soubor, a filtru `DataInputStream`, napojeného na podkladový proud. Nakonec v cyklu přečteme potřebná data a uložíme je do pole `a`. ♦

Přenositelnost a nepřenositelnost binárních souborů



Binární soubory, vytvořené programem v Javě, může číst jakýkoli jazykový program běžící na jakémkoli počítači. *Na druhé straně jazykový program nemusí dokázat přečíst binární soubor vytvořený na témže počítači programem napsaným v jiném programovacím jazyce, a podobně program napsaný v jiném programovacím jazyce nemusí umět správně přečíst data vytvořená programem v Javě.*



Jestliže např. program v C++ zapíše na PC do binárního souboru řadu celých čísel (typu `int`), pak program v Javě tato čísla nepřečte správně. Na podobné problémy narazíme i při přenosu souborů obsahujících reálná čísla.

Důvod je jednoduchý: Java přesně určuje způsob uložení dat – mimo jiné i typu `int` – v paměti počítače, a toto uložení je jiné než je „přirozené“ uložení těchto typů na PC, používané v jiných programovacích jazycích.

Příklad:



Na WWW v adresáři Kap13\02a najdete program `test.cpp`, napsaný v C++, který vytvoří binární soubor `data.dta` a zapíše do něj čísla od nuly do 9. Dále tam najdete tento program přeložený překladačem pro 32bitová Windows (soubor `test.exe`); spusťte ho z příkazové řádky příkazem

```
test
```

Program `test.java`, který najdete v téžem adresáři, se tato čísla pokusí přečíst a vypsat. Na obrazovce ale uvidíme čísla 0, 16777216, 33554432, 50331648, 67108864, 83886080, 100663296, 117440512, 134217728, 150994944.

Na podobné problémy narazíme i při práci s reálnými čísly (typu `double` a `float`). ◊

Ukládání objektů (serializace)

Ukládání hodnot primitivních typů je sice skvělé, ale při programování v Javě dříve či později narazíte na nutnost ukládat do souborů či jinam celé objekty tak, abyste je mohli později načíst a obnovit v původním stavu, včetně vztahů k jiným objektům. Tomu se říká *perzistence* a v Javě se to řeší pomocí tzv. *serializace*. Problematika perzistence je poměrně komplikovaná, ovšem její implementace založená na serializaci je pro uživatele velice jednoduchá.

Proudy
pro objekty

K ukládání objektů do souborů a k jejich následnému čtení slouží proudy `ObjectOutputStream`, resp. `ObjectInputStream`. Jsou to proudy, které používáme podobně jako filtry, tj. ve spojení s některým

z podkladových proudů. Proud `ObjectOutputStream` obsahuje metodu `writeObject()`, která očekává parametr typu `Object` odkazující na ukládaný objekt; proud `ObjectInputStream` obsahuje metodu `readObject()`, která vrací odkaz typu `Object` na přečtený objekt.



Třída, jejíž instance, chceme ukládat a číst pomocí mechanizmu serializace, musí implementovat rozhraní `java.io.Serializable` a musí obsahovat hodnoty primitivních typů, odkazy na objekty, které samy implementují toto rozhraní, nebo pole primitivních typů nebo objektů, které implementují toto rozhraní, jinak vznikne při pokusu o čtení nebo zápis výjimka typu `java.io.NotSerializableException`. Většina knihovních tříd tuto podmínu splňuje (mj. i třída `ArrayList` a další kontejnery).

Při čtení musí být k dispozici soubory `.class` obsahující přeložené třídy objektů, které se načítají, jinak vznikne výjimka typu `ClassNotFoundException`.

Poznamenejme, že rozhraní `Serializable` neobsahuje žádné metody, slouží pouze k „označení“ tříd, které dovolují serializaci svých instancí.

Příklad: Vratme se zase jednou k našim oblíbeným grafickým objektům. Mají-li být opravdu použitelné v grafickém editoru, musíme vyřešit i jejich ukládání do souboru – a zde se serializace nabízí jako nejjednodušší cesta.⁶⁸ Zásahy do programu s grafickými objekty budou minimální:

- Třída `GO`, která je společným předkem grafických objektů, implementuje rozhraní `Serializable`.
- Ve třídě `Test` přibudou metody pro uložení kontejneru obsahujícího grafické objekty do souboru a pro jeho opětovné načtení.

To je vše. Podívejme se na odpovídající části zdrojového kódu. Nejprve třída `GO`, ve které přibyla implementace dalšího rozhraní:

```
/* Soubor Kap12\03\grafika\GO.java */
package grafika;
import kontrola.*;
import java.io.*;

public abstract class GO
implements Kontrola, Cloneable, Serializable {
    // Datové složky a metody stejně jako předtím
}
```

⁶⁸ Nikoli ovšem nejúspornější. Při serializaci se kromě obsahu objektů ukládají i další data, která umožní následné obnovení objektů a vztahů mezi nimi. Pokud by nám tedy šlo o velikost souboru s uloženým obrázkem, bylo by rozumnější nespoléhat na serializaci a navrhnout vlastní vektorový formát.

V metodě `beh()` třídy `Test` vytvoříme instanci kontejneru `ArrayList` a uložíme do něj nějaké grafické objekty. Pak zavoláme metodu `uloz()`, která se postará o uložení celého kontejneru.

Nyní se podívejme na metodu pro uložení seznamu objektů do souboru.

```
public void uloz(String soubor, Object co) throws IOException
{
    File f = new File(soubor);
    FileOutputStream fos = new FileOutputStream(f);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(co);
    oos.close();
}
```

První parametr obsahuje jméno souboru, druhý ukládaný objekt. Začneme tím, že vytvoříme instanci třídy `File`, a s její pomocí zjistíme, zda existuje potřebný soubor; pokud ne, vytvoříme ho.

Pak vytvoříme podkladový proud. Protože nám jde o ukládání do binárního souboru, použijeme `FileOutputStream`. Ten pak předáme jako parametr konstruktoru filtru typu `ObjectOutputStream`. Poslední, co je třeba udělat, je zapsat objekt `co` do souboru pomocí metody `writeObject()` a uzavřít proud.

Metoda pro čtení objektu ze souboru bude podobně jednoduchá:

```
public Object nacti(String soubor)
throws IOException, ClassNotFoundException
{
    File f = new File(soubor);
    if(!f.exists()) throw new IOException(
        "soubor " + soubor + " neexistuje");

    FileInputStream fis = new FileInputStream(f);
    ObjectInputStream ois = new ObjectInputStream(fis);
    return ois.readObject();
}
```

Pokud udaný soubor neexistuje, vyvoláme výjimku. Jinak vytvoříme instanci podkladového proudu typu `FileInputStream`, kterou předáme konstruktoru filtru `ObjectInputStream`. O vlastní přečtení se postará metoda `readObject()`, která vrátí odkaz na typ `Object`.

V metodě `beh()` třídy `Test` uložíme kontejner plný grafických objektů do souboru příkazem

```
uloz("data.dta", seznam);
```

Při příštém běhu programu nahradíme tento příkaz příkazy

```
seznam = (ArrayList) nacti("data.dta");
System.out.println(seznam);
```

které přečtou uložené objekty a pro kontrolu je vypíší. Kontrolní výpis ukáže, že seznam obsahuje tytéž objekty, jaké jsme v předchozím běhu uložili.

Povšimněte si přetypování hodnoty vrácené metodou `nacti()`. Ta vrací odkaz na nejobecnější typ `Object` (tak, jak ho vráti metoda `readObject()`). Abychom ho mohli používat, musíme ho přetypovat na `ArrayList`.



Úplný zdrojový kód tohoto příkladu najdete na WWW v adresáři `Kap13\03` v podadresářích odpovídajících jednotlivým balíkům. ♦

13.3 Čtení a zápis znakových dat

Za poněkud neobvyklým označením „znaková data“ se skrývají tzv. textové soubory, tedy soubory, které obsahují znaky – znakovou reprezentaci nejrůznějších dat. Data v textových souborech jsou navíc rozdělena na řádky.

Textové soubory (a znaková data vůbec) slouží převážně pro komunikaci mezi počítačem a člověkem. Práce s nimi je pro počítač obtížnější, neboť chceme-li do textového souboru zapsat nějakou hodnotu, musí ji počítač nejprve převést na znakový řetězec a ten pak vyspat.

Pro čtení znakových dat používáme proudy odvozené od třídy `Reader` a pro zápis proudy odvozené od třídy `Writer`.

Už jsme si řekli, že tyto třídy pracují se znaky. Mohli bychom tedy předpokládat, že data čtou a ukládají v kódování Unicode. Ovšem vzhledem k tomu, že většina současných operačních systémů používá kódování založené na kódu ASCII (jednobajtovém), transformují se data při ukládání z Unicode do kódování používaného v daném operačním systému a při čtení z kódování používaného v daném operačním systému do Unicode.

Zápis do textového souboru

Pro výstup do textového souboru používáme zpravidla proud `FileWriter`. Parametrem konstruktoru může být instance třídy `File` nebo řetězec obsahující jméno souboru. Konstruktor proudu `FileWriter` může mít vedle znakového řetězce ještě druhý parametr typu `boolean`, který určuje, zda se mají data v souboru při otevření smazat nebo zachovat (podobně jako v případě proudu `FileOutputStream`).

Proud `FileWriter` obsahuje metodu `write()`; hovořili jsme o ní podrobněji v úvodu této kapitoly. Proud `FileWriter` se však často používá jako podkladový pro proud `PrintWriter`, který obsahuje nám už dobře známé metody `print()` a `println()`.

**Metoda
`printf()`**

Počínaje JDK 5 obsahuje proud `PrintWriter` také metodu `printf()`, jež umožňuje formátovaný výstup několika hodnot zároveň. Na rozdíl od metod `print()` a `println()` přitom využívá lokálního nastavení, takže např. v českém prostředí vypíše reálné číslo s desetinnou čárkou, nikoli tečkou. Má proměnný počet parametrů a její deklarace má tvar

```
public PrintWriter printf(String format, Object ... param);
```

5

První parametr představuje znakový řetězec, který bude překopirován do výstupu. Tento řetězec může obsahovat tzv. specifikace konverze, tj. zápisy začínající znakem % a končící písmenem určujícím typ parametru, který vystupuje. Tak například písmena d, o, h znamenají celé číslo po řadě v desítkové, osmičkové a šestnáctkové soustavě, e, f a g znamenají reálné číslo, s znamená znakový řetězec, % znamená znak procent atd. Každá ze specifikací konverzí se vztahuje k jednomu z následujících parametrů – první specifikace k prvnímu parametru za řetězcem `format`, druhá specifikace k druhému parametru atd. Platí-li např. deklarace

```
double x = 3.14159;  
long l = Long.MIN_VALUE;
```

způsobí příkaz

```
System.out.printf("Zadaná čísla jsou %f\n a %d%%\n", x, l);
```

že počítač vypíše

```
Zadaná čísla jsou 3,141590  
a -9223372036854775808%
```

Mezi znak %, jímž specifikace začíná, a znak specifikující typ konverze lze ještě zapsat celé číslo určující šířku (tj. minimální počet znaků, které má výstup této hodnoty zabírat) a přesnost (počet míst za desetinnou čárkou, které mají vystoupit). Například příkaz

```
System.out.printf("%7.2f\n", x);
```

předepisuje, že výstup má zabírat 7 míst, z toho 2 za desetinnou čárkou. Tento příkaz způsobi výstup

3.14



se třemi mezerami na počátku. Viz též soubor `Kap13\06\Pokus.java`.

Podrobnější informace o této metodě najdete v dokumentaci k JDK 5.

**Příklad:
zápis
do souboru** Ukážeme si výstup čísel a textu do textového souboru. Vypíšeme tam nejprve celá čísla od 1 do 10, pak čísla π a e jako příklady reálných čísel a nakonec tam zapíšeme známou větu o žluťoučkém koni, na které se zkouší čeština.

```
import java.io.*;
public class Test1250 {
    public static void main(String[] args) throws Exception {
        File f = new File("data.dta");
        FileWriter fw = new FileWriter(f);
        PrintWriter pw = new PrintWriter(fw);
        for(int i = 1; i <= 10; i++) pw.print(i+ " ");
        pw.println();
        pw.println(Math.PI);      // Ludolfovovo číslo
        pw.println(Math.E);       // Eulerovo číslo
        pw.println("Žluťoučký kůň píšerně úpěl dábešské ódy");
        pw.close();               // Nezapomenout uzavřít!
    }
}
```



Pracujete-li pod Windows a otevřete-li vytvořený soubor data.dta např. v poznámkovém bloku, zjistíte, že se české znaky vypsaly správně. Zdrojový text tohoto příkladu najdete na WWW v souboru Kap13\04\Test.java. ♦

Čtení z textového souboru

5 Ke čtení z textového souboru slouží v JDK 5 třída `java.util.Scanner`. Skener proud, je to univerzální nástroj, který lze připojit k souboru, proudu, kanálu nebo znakovému řetězci a který z tohoto zdroje čte jednotlivé položky („slova“) oddělené bílými znaky.

- O připojení skeneru ke zdroji dat se postará konstruktor. Jeho parametrem může být instance třídy `File` představující čtený soubor, instance třídy `InputStream` představující vstupní proud, instance třídy `java.nio.ReadableByteChannel` představující vstupní kanál nebo instance třídy `String`. Jako parametr konstruktoru lze samozřejmě použít také instanci `System.in` představující standardní vstup.
- Každý z konstruktorů může mít jako druhý parametr znakový řetězec vyjadřující kódování, které se má při vstupu použít. Pro nás mohou být zajímavé hodnoty „Cp852“ nebo „Cp1250“. Neuvedeme-li ho, použije se kódování, které je v daném prostředí implicitní (pod Windows to v českém prostředí bude Cp1250). Poznámejme, že pokud uvedeme nesprávný řetězec určující kódovou

stránku, vyvolá tento konstruktor výjimku `UnsupportedEncodingException`.

- V programu můžeme mít několik instancí této třídy, které budou číst z různých zdrojů. Proto jsou všechny následující metody nestatické.
- Metoda `hasNext()` vrátí hodnotu typu `boolean` určující, zda má smysl dále číst, tj. zda je ve vstupu ještě nějaké další slovo. Metody `hasNextInt()`, resp. `hasNextDouble()` a další umožňují zjišťovat, zda ve vstupu následuje slovo představující znakovou reprezentaci celého, resp. reálného čísla apod. Metoda `hasNextLine()` umožňuje zjistit, zda vstup obsahuje další řádek.
- Metoda `next()` přečte ze vstupu další slovo. Metody `nextInt()`, `nextDouble()` apod. umožňují přečíst následující číslo typu `int`, `double` apod. Metoda `nextLine()` vrátí zbytek aktuálního řádku (bez oddělovače řádků) a přejde na nový řádek.
- Metoda `close()` uzavře daný skener.

Příklad: Vyzbrojeni těmito vědomostmi si položíme následující úkol: V poznámkovém bloku či v jiném textovém editoru, který používáme k přípravě zdrojových textů našich programů, vytvoříme soubor `data.txt` obsahující několik řádků celých čísel oddělených mezerami, např.

```
99 68  
88  
78 98 55  
123 5555
```

a napišeme program, který v tomto souboru vyhledá největší číslo a vytiskne ho. S třídou `Scanner` je to opravdu jednoduché:

```
public int maxVSouboru(String jmeno) throws Exception  
{  
    File f = new File(jmeno);  
    if(!f.exists()) throw new IOException("soubor neexistuje");  
    int max = Integer.MIN_VALUE; // Dočasné maximum  
    Scanner skan = new Scanner(f); // Připojíme skener k souboru  
    while (skan.hasNextInt()) // čti, dokud je co  
    {  
        int n = skan.nextInt();  
        if(n > max) max = n;  
    }  
    skan.close();  
    return max;  
}
```

Nejprve ověříme, zda soubor existuje, a pak na něj napojíme skener. V podmínce příkazu `while` testujeme, zda lze ze souboru číst ještě

nějaké další číslo. Pokud ano, přečteme ho a zjistíme, zda není větší než největší dosud přečtená hodnota.

Přitom předpokládáme, že někde v záhlaví zdrojového souboru jsou příkazy

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



Zdrojový text tohoto příkladu najdete na WWW v souboru Kap13\05\TestSc.java. ♦

Nyní se podíváme, jak je to se čtením z textových souborů ve starších verzích Javy. I když se může zdát, že zabývat se tím nemá smysl, doporučují vám tuto část nepřeskakovat; zaprvé se seznámíte s několika užitečnými třídami a metodami a zadruhé je to užitečné programátorské cvičení.

Starší verze JDK

Ve starších verzích JDK je situace poněkud problematičejší. Java až po verzi JDK 1.4 včetně totiž neobsahuje nástroje, které by umožňovaly číst přímo z textového souboru číselné nebo logické hodnoty. Pokud něco takového potřebujeme, musíme si to naprogramovat.

Pro čtení z textových souborů zpravidla používáme proud `FileReader`. Jeho konstruktor může mít jako parametr instanci třídy `File` nebo znakový řetězec obsahující jméno souboru, v případě potřeby včetně cesty. Tato třída obsahuje přetíženou metodu `read()`, o níž jsme se zmiňovali v úvodu této kapitoly.

Často ale třídu `FileReader` používáme jako podkladovou třídu pro třídu `BufferedReader`, která obsahuje navíc metodu `readline()`. Tato metoda vrátí znakový řetězec obsahující načtený řádek (bez oddělovače řádků) nebo `null`, pokud jsme již došli na konec souboru.

Pokud načtený text obsahuje čísla, která chceme převést ze znakové reprezentace na numerické hodnoty, musíme se o to postarat také sami, Java nám k tomu ovšem poskytuje docela dobré nástroje:

- Řetězec, obsahující několik položek oddělených mezerami, můžeme rozložit na jednotlivé položky (slova) pomocí metod třídy `java.util.StringTokenizer`.
- Řetězec, obsahující znakovou reprezentaci celého čísla, převedeme na odpovídající celé číslo např. pomocí statické metody `Integer.parseInt()`.
- Řetězec, obsahující znakovou reprezentaci reálného čísla, převedeme na odpovídající reálné číslo např. pomocí statické metody `Double.parseDouble()`.

Ukážeme si to na příkladu.

Příklad: Položíme si stejný úkol jako prve: Máme textový soubor obsahující zápisy celých čísel oddělené bílými znaky a chceme napísat program, který v tomto souboru vyhledá největší číslo a vytiskne ho.

```
public static int maxVSouboru(String jmeno) throws Exception
{
    File f = new File(jmeno);
    if(!f.exists()) throw new IOException("soubor neexistuje");
    int max = Integer.MIN_VALUE; // Dočasné maximum

    FileReader fr = new FileReader(f);
    BufferedReader bw = new BufferedReader(fr);
    String cisla;

    while ((cisla = bw.readLine()) != null)
    {
        // analyza řádku
        StringTokenizer st = new StringTokenizer(cisla);
        while(st.hasMoreTokens())
        {
            String s = st.nextToken();
            int i = Integer.parseInt(s);
            if(i > max) max = i;
        }
    }
    bw.close();
    return max;
}
```

Jako obvykle při vstupních operacích nejprve zkontrolujeme, zda existuje požadovaný soubor, a pokud ne, vyvoláme výjimku.

Pak vytvoříme nový proud typu `FileReader` a napojíme ho na proud `BufferedReader`, který umožňuje čtení po řádcích s použitím vyrovnávací paměti.

Proměnná `cisla` typu `String` bude obsahovat jednotlivé řádky přečtené ze vstupního souboru. Proměnná `max` poslouží k výpočtu maxima z přečtených hodnot. Proto do ní na počátku uložíme nejmenší hodnotu, jakou může typ `int` obsahovat.

V následujícím cyklu čteme jednotlivé řádky ze souboru a zpracováváme je. Čtení skončí, když narazíme na konec souboru a metoda `readLine()` vrátí `null`.

V tomto cyklu nejprve vytvoříme instanci třídy `StringTokenizer`; konstruktorem předáme jako parametr načtený řetězec. Pomocí metody `hasMoreTokens()` pak v dalším cyklu testujeme, zda řetěz obsahuje další skupinu znaků oddělenou mezerou; pokud ano, vyžádáme si ji

pomocí metody `nextToken()` a její hodnotu zjistíme pomocí statické metody `Integer.parseInt()`.

Podobně jako v předchozím příkladu i zde předpokládáme, že v záhlaví zdrojového souboru jsou příkazy

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

Tento program najdete na WWW v souboru `Kap13\05\Test.java`. ♦



Čeština v Javě pod Windows

V oddílu, věnovaném výstupu do souborů, jsme viděli, že řetězce se nezapisují v kódování Unicode, ale v kódování, které je v daném operačním systému běžné (implicitní). Pod Windows to je kódová stránka 1250. Při výstupu do konzolového okna se ovšem ve Windows používá kódová stránka 852 (tzv. kódování Latin 2, pozůstatek z dob operačního systému DOS). Proto je výstup obsahující české znaky do proudu `System.out` prakticky nečitelný.

Výstup českých znaků

Java počítá s tím, že budeme chtít vytvořit soubor v jiném než implicitním kódování. (Nemusí jít jen o problémy s češtinou pod Windows; můžeme např. chtít vytvořit soubor v jiném jazyce.) Konstruktor proudu `OutputStreamWriter` může mít jako druhý parametr znakový řetězec, který určuje požadované kódování. Pro nás mohou být opět zajímavé hodnoty "Cp852" nebo "Cp1250"; některé další můžete najít v návodě k této třídě. Uvedeme-li nesprávný řetězec určující kódovou stránku, vyvolá tento konstruktor výjimku `UnsupportedEncodingException`.

Tato třída ale pracuje s podkladovým proudem odvozeným od třídy `OutputStream`, nikoli od `Reader`. Proto ji jako parametr můžeme zadat mj. např. `System.out`. Proud `OutputStreamWriter` ovšem nepoužíváme přímo, ale prostřednictvím některého z dalších proudu – vhodný je např. `PrintWriter`.

Příklad: Žlutý kůň



Ve 3. kapitole jsme se neúspěšně snažili chtěli vypsat na obrazovku větu o žlutoučkém koni. Podivejme se, jak na to:

```
/* soubor Kap03\02\ZlutyKun2.java*/
import java.io.*;

public class ZlutyKun2 {
    public static void main(String[] arg) throws Exception
    {
        OutputStreamWriter osw =
            new OutputStreamWriter(System.out, "Cp852");
        PrintWriter p = new PrintWriter(osw);
```

```
    p.println("Žluťoučký kůň příšerně úpěl dábejské ódy.");
    p.close();
}
}
```

Pokud bychom chtěli vytvořit textový soubor v kódování Latin 2, můžeme. Protože neustálé opakování jedné věty je úmorné a mohlo by někomu připadat jako týrání zvířat, napišeme pro změnu větu, na kterou asi hned tak nezapomene nikdo, kdo se o vánocích 2000 díval na televizi. Také na ní si prověříme správnost kódování českých znaků.

```
import java.io.*;
public class Hlod {
    public static void main(String[] args) throws Exception {
        File f = new File("data.dta");
        if(!f.exists()) f.createNewFile();

        FileOutputStream fos = new FileOutputStream(f);
        OutputStreamWriter osw =
            new OutputStreamWriter(fos, "Cp852");
        PrintWriter pw = new PrintWriter(osw);

        pw.println("Generální ředitel ČT Jiří Hodač se obrátil\n"
            + "na Radu České republiky pro rozhlasové \nna televizní "
            + "vysílání se žádostí o rozhodnutí, \nkterý program ČT "
            + "je legálním a autorizovaným \n"
            + "programem v souladu se zákonem o České \n"
            + "televizi a který nikoli. Do rozhodnutí Rady \n"
            + "vysílá ČT jako svůj program toto sdělení.");
        pw.close();
    }
}
```

Nejprve vytvoříme instanci podkladového proudu; protože nám jde o zápis do souboru, použijeme `FileOutputStream`. Tento proud pak předáme konstruktoru třídy `OutputStreamWriter` spolu s požadavkem na kódovou stránku 852. Vytvořený proud budeme používat prostřednictvím instance třídy `PrintWriter`.



Vstup českých znaků

Zdrojový text tohoto programu najdete na WWW v souboru Kap13\05\Hlod.java.

Potřebujeme-li správně načíst znaky české (nebo slovenské) národní abecedy, postupujeme podobně jako při jejich vypisování, pouze místo třídy `OutputStreamWriter` použijeme třídu `InputStreamReader` a místo výstupního proudu typu `OutputStream` použijeme vstupní proud typu `InputStream`.

Příklad: Napišeme program, který přečte ze souboru v kódování Latin 2 text, který tam uložil náš předchozí program, a vypíše ho na obrazovku:

```

import java.io.*;
public class CtiHlod {
    public static void main(String[] args) throws Exception {
        File f = new File("data.dta");
        if(!f.exists())
            throw new IOException("soubor neexistuje");
        FileInputStream fis = new FileInputStream(f);
        InputStreamReader isr =
            new InputStreamReader(fis, "Cp852");
        BufferedReader br = new BufferedReader(isr);
        OutputStreamWriter osw =
            new OutputStreamWriter(System.out, "Cp852");
        PrintWriter pw = new PrintWriter(osw);

        String text;
        while((text = br.readLine()) != null)
            pw.println(text);
        pw.close();
    }
}

```

Nejprve vytvoříme podkladový proud typu `FileInputStream`, ten pak předáme konstruktoru proudu `InputStreamReader` spolu s požadavkem na kódovou stránku. Pro vlastní čtení použijeme jako obvykle `BufferedReader` a jeho metodu `readLine()`.

5 V JDK 5 místo vstupních proudů použijeme už známou třídu `Scanner` a její metody `hasNextLine()` a `nextLine()`. Zde si ukážeme jen ty části, které se změní.

```

public static void main(String[] args) throws Exception {
    // Ověření existence souboru f a deklarace výstupního proudu
    // pw je stejná jako předtím
    Scanner skan = new Scanner(f, "Cp852");

    while(skan.hasNextLine())           // Čteme a vypisujeme
        pw.println(skan.nextLine());    // řádek po řádku
    pw.close();
    skan.close();
}

```



Zdrojové texty tohoto příkladu najdete na WWW v souborech `Kap13\05\CtiHlod.java` a `Kap13\05\CtiHlod5.java` ♦

14 Grafické uživatelské rozhraní

Nástroje pro vytváření grafického uživatelského rozhraní programů, tedy „oken“, jsou součástí Javy již od počátku. Jejich první verzi najdeme v balíku `java.awt` (AWT je zkratka slov Abstract Windowing Toolkit, tedy „abstraktní okenní nástroje“). Ideálem tvůrců bylo, aby aplikace, vytvořené pomocí AWT, vypadaly stejně dobře ve všech prostředích, v nichž Java poběží.

Tato verze se ovšem příliš nepodařila. Zlý jazykové dokonce prohlašovali, že aplikace vytvořené pomocí AWT vypadají ve všech prostředích stejně bídňě. Proto se časem objevila knihovna vizuálních a nevizuálních komponent⁶⁹, které AWT nahradily. Tato knihovna dostala označení JFC, což je zkratka anglických slov Java Foundation Classes⁷⁰, a její část určená pro tvorbu grafického uživatelského rozhraní aplikací se nazývá Swing. Třídy z této knihovny najdeme v balíku `javax.swing`.

My se v této kapitole seznámíme se základy práce s knihovnou Swing. Použijeme přitom podobný postup jako v úvodních kapitolách, tj. na příkladech si ukážeme některé z možností, které tato knihovna poskytuje. Podrobný výklad o této knihovně by vydal na samostatnou knihu, a nijak tenkou, a proto mějte na paměti, že jde opravdu jen o neúplný úvod.

14.1 První okno

Začneme programem, který zobrazí okno na obrazovce monitoru.

Správné ukončení programu

Prázdné okno na obrazovce představuje třída `JFrame`⁷¹. My ale zpravidla budeme potřebovat okna obsahující další komponenty – tlačítka,

⁶⁹ Tyto komponenty lze snadno využít i k vizuálnímu programování v prostředích, jako je JBuilder nebo NetBeans. Poznamenejme, že komponenty, tedy předdefinované třídy vyhovující některým dalším požadavkům, se v Javě označují *JavaBeans*, tedy „kávové boby“.

⁷⁰ Označení *Foundation Classes* se užívá pro knihovnu tříd, které umožňují samy o sobě vytvořit ucelenou aplikaci.

⁷¹ Třídy v knihovně AWT se jmenovaly „logicky“ Frame (okno), Button (tlačítko), Applet (aplet) atd. Třídy v novější knihovně JFC/Swing se stejným

textová pole atd. – a proto si od této třídy odvodíme vlastního potomka, třídu, kterou vtipně nazveme *Vokno*; do ní budeme později přidávat různé datové složky a metody. Umístíme ji do balíku *vokno*:

```
package vokno;
import javax.swing.*;

public class Vokno extends JFrame
{
    Vokno(){} // Volá implicitní konstruktor předka, nic víc
}
```

Příkazem `new Vokno()` vytvoříme novou instanci třídy *Vokno*, tedy datovou reprezentaci okna v paměti. Abychom okno zobrazili na obrazovce monitoru, musíme zavolat metodu `setVisible()` s parametrem `true`.

 Pokud si nové okno ihned vyzkoušíme, nedopadne to nejlépe:

```
public class Prvni
{
    public static void main(String[] a)
    {
        vokno.Vokno okno = new vokno.Vokno(); // Vytvoř okno
        okno.setVisible(true); // a zobraz ho
    }
}
```

 Ikona mouchy po levé straně upozorňuje, že s tímto programem není něco v pořádku. Jestliže ho přeložíte a spustíte, budete okno nejspíš chvíli hledat, a teprve pak najdete někde v rohu miniovéko nejmenší možné velikosti. (Výsledek ve Windows ukazuje nečíslovaný obrázek po straně). Zvětšíme-li si ho myší, zjistíme, že jde o normální prázdné okno se sedým pozadím a bez nápisu v záhlaví (titulkové liště).

Když se ho pokusíme uzavřít klepnutím na ikonu s křížkem v pravém rohu, okno zmizí, ale program poběží dál a bude blokovat konzolové okno operačního systému. Budete ho muset ukončit stisknutím klávesové kombinace `Ctrl+C`.

Aby se nás program choval korektně, musí reagovat na uzavření svého hlavního okna tím, že skončí. To můžeme zařídit několika způsoby. V JDK 1.3 a pozdějších stačí, když pro instance hlavního okna naší aplikace zavoláme metodu

```
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // JDK1.3+
```

účelem mají jména začínající písmenem *J*, tedy *JFrame*, *JButton*, *JApplet* atd.

např. v konstruktoru třídy `Prvni`, ihned po vytvoření instance. Uzávření okna pak způsobí ukončení programu voláním metody `System.exit(0)`.⁷²

V JDK 1.2 (a samozřejmě i v pozdějších verzích Javy) můžeme místo toho použít následující kód:

```
okno.addWindowListener(new WindowAdapter() { // JDK 1.2+
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
```

Zatím nehloubejte nad tím, co tento kód znamená; brzy mu porozumíte.

Poznamenejme, že sem – před volání metody `System.exit()` – můžeme zařadit ještě další operace, např. dotaz, zda si uživatel opravdu přeje skončit.

Třetí možnost, se kterou jsme se mohli setkat už v JDK 1.0, je ze všech nejsložitější, a vůbec bychom o ní nehovořili, kdyby ji dodnes nepoužívaly některé vývojové nástroje (např. JBuilder). Zde musíme do konstruktoru okna přidat příkaz

```
enableEvents(AWTEvent.WINDOW_EVENT_MASK); // JDK 1.0+
```

kterým povolujeme vznik událostí v okně. (Co to znamená, to si povíme později). Kromě toho musíme ve třídě okna předefinovat zděděnou metodu `processWindowEvent()`, která událost uzavření okna zpracuje:

```
protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e); // JDK 1.0
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        System.exit(0);
    }
}
```

Jejím parametrem je instance `e` třídy `java.awt.event.WindowEvent`, která obsahuje data určující, o jakou událost jde. Tuto událost nejprve předáme metodě předka. Pak zjistíme, zda jde o uzavření okna, a pokud ano, ukončíme program.

⁷² Konstanta `EXIT_ON_CLOSE` je definována ve třídě `JFrame`. Další konstanty, které lze této funkci předat jako parametry, jsou `DISPOSE_ON_CLOSE` (zruš okno při uzavření), `DO NOTHING ON CLOSE` (nedělej nic), `HIDE ON CLOSE` (skryj okno při uzavření). Jsou definovány v rozhraní `javax.swing.WindowConstants`.

Poznamenejme, že třídu `AWTEvent` najdeme v balíku `java.awt` a třídu `WindowEvent` v balíku `java.awt.event`. Abychom nemuseli jména tříd kvalifikovat jménem balíků, zařadíme na počátek zdrojových souborů odpovídající příkazy import.



V JDK 1.3 a novějších můžeme použít kteroukoli z uvedených možností. Takto upravený program se již chová standardně – po uzavření okna skončí. Zdrojový text obsahující možnost pro JDK 1.0 najdete na WWW v adresáři `Kap14\01-1_0`, zdrojový text programu pro JDK 1.2 a 1.3 v adresáři `Kap14\01-1_2`. ♦

Všimněte si jedné zvláštnosti: Metoda `main()` by měla po provedení svých příkazů skončit a s ní by mělo zaniknout také okno programu. Ve skutečnosti však aplikace, která si vytvoří okno, po provedení všech příkazů metody `main()` neskončí – okno „žije dál“ a zpracovává události, ke kterým v něm dojde.



Programy, které používají grafické uživatelské rozhraní, musí zpravidla končit voláním metody `System.exit()`.

Vlastnosti komponent

Když jsme chtěli zobrazit okno, zavolali jsme metodu `setVisible()`. Kdybychom chtěli zjistit, zda je okno viditelné, použili bychom metodu `isVisible()`, která vrací hodnotu typu `boolean`.

Budeme-li chtít nastavit velikost okna, použijeme jeho metody `setHeight()`, resp. `setWidth()`, a budeme-li chtít jeho rozměry zjistit, použijeme metody `getHeight()`, resp. `getWidth()`, které vracejí `int`. Podobně budeme postupovat při určení nápisu v záhlaví – použijeme metodu `setTitle()`, resp. `Title()`.

Vlastnost je dána metodami
get... a set...

Říkáme, že okno – nebo obecně komponenta JavaBean – má *vlastnost* (*property*) `Visible`, `Width`, `Height`, `Title` atd. Identifikátory metod, které nastavují hodnotu vlastnosti, se skládají ze slova `set` a jména vlastnosti, identifikátory metod, které zjišťují hodnotu vlastnosti, se skládají ze slova `get` (jde-li o vlastnost typu `boolean`, pak `is`) a jména vlastnosti.

Poznamenejme, že některé vlastnosti komponent JavaBeans lze pouze nastavit (tzn. je k dispozici pouze metoda `set...`, některé jiné vlastnosti mohou být pouze „ke čtení“ (je pro ně k dispozici pouze metoda `get...`, resp. `is...`).

Velikost a umístění okna

Nyní vylepšíme nás program a nastavíme rozměry a umístění vytvořeného okna. Před tím, než ho zobrazíme, nastavíme jeho rozměry na polovinu rozměru obrazovky.⁷³

Rozměry okna

Mohli bychom využít vlastnosti `Height` a `Width`, o nichž jsme hovořili v předchozím odstavci. Protože nám ale jde o nastavení obou rozměrů zároveň, použijeme vlastnost `Size`, která je shrnuje obě uvedené vlastnosti. Pro vyjádření dvojice rozměrů (tj. dvojice čísel typu `int`) se v Javě používá knihovná třída `java.awt.Dimension`, která má složky `height` a `width` typu `int`.

Rozměry obrazovky

Ke zjištění rozměrů obrazovky monitoru použijeme metodu `java.awt.Toolkit.getDefaultToolkit().getScreenSize()`, která vrátí instanci třídy `Dimension`.

Nyní tedy upravíme konstruktor našeho okna následujícím způsobem:

```
public Vokno()
{
    Dimension obrazovka = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension rozm = new Dimension();
    rozm.height = obrazovka.height/2;
    rozm.width = obrazovka.width/2;
    setSize(rozm);
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });
}
```

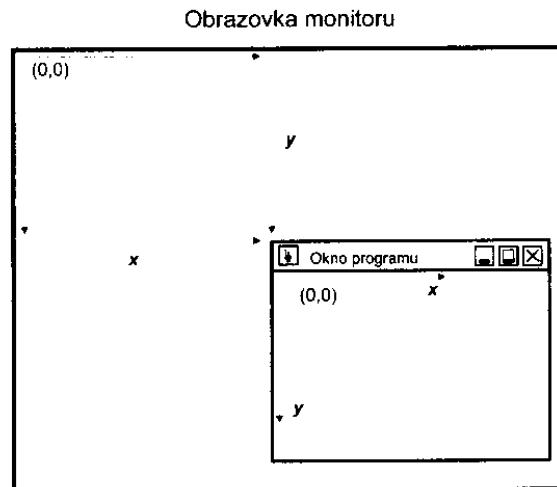
Po překladu a spuštění se objeví okno, které již nebudeme muset hledat; bude v levém horním rohu obrazovky. To je ale poněkud neobvyklé – byli bychom raději, kdyby se zobrazilo uprostřed. K tomu nám poslouží vlastnost `Location`, která vyjadřuje polohu pravého horního rohu okna. Nastavíme ji pomocí metody `setLocation()`, jejíž první parametr znamená vzdálenost od levého horního rohu okna od levého horního rohu obrazovky ve vodorovném směru a druhý parametr znamená vzdálenost od téhož rohu ve svislému směru. (Způsob udávání polohy na obrazovce monitoru a v okně programu ukazuje obr. 14.1.)

Má-li okno délku, resp. šířku, rovnou polovině délky, resp. šířky obrazovky a má-li být uprostřed, musí být vzdálenost jeho levého horní-

⁷³ Rozměry oken a jiných komponent jsou celočíselné a vyjadřují se v pixelech (bodech, ze kterých se skládá obraz v grafickém režimu).

ho rohu od levého horního rohu obrazovky rovna čtvrtině šířky, resp. čtvrtině výšky obrazovky. Do konstruktoru okna tedy přidáme příkaz

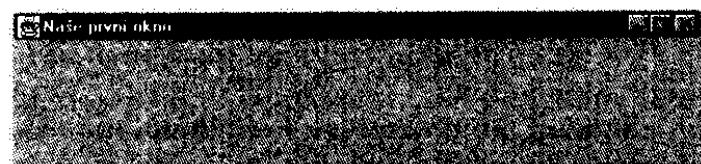
```
setLocation(rozm.width/2, rozm.height/2);
```



Obr. 14.1 Souřadnicový systém na obrazovce, resp. v okně programu. Počátek soustavy souřadnic, tedy bod, od kterého se měří, je v levém horním rohu obrazovky, resp. okna. Souřadnice x roste zleva doprava, souřadnice y shora dolů

Titulek v záhlaví okna

Nakonec bychom chtěli do záhlaví okna umístit nápis, který bude informovat o smyslu okna nebo programu, který ho zobrazil. Tento nápis je dán vlastností `Title`; metoda `setTitle()` má jediný parametr, znakový řetězec, který chceme zobrazit. Výsledek (na výšku změněný, neboť prázdné okno je poměrně nezajímavé) ukazuje obr. 14.2.



Obr. 14.2 Okno našeho programu

Všimněte si, že JDK 1.3 a pozdější se bez obtíží vyrovná s češtinou, a to i pod Windows. Bohužel, v některých starších verzích JDK to nebylo tak jednoduché.⁷⁴



Tento program najdete na WWW v adresáři Kap14\02.

Nastavování počátečních rozměrů okna, titulku v záhlaví a dalších vlastností, to jsou inicializace, které je třeba udělat ještě před zobrazením okna na obrazovce. Zdá se tedy jasné, že patří do konstruktoru.

Na druhé straně není důvodu, proč by naše aplikace nemohla obsahovat několik oken též třídy s různými rozměry, s různým nápisem v záhlaví atd. To ale znamená, že bude rozumnější ponechat tyto operace na třídě, která si instanci okna vytvoří – v našem případě na třídě Prvni. Přesuneme tedy tyto operace do jejího konstruktoru.

Podobně je rozumné nastavovat ve třídě Prvni i chování okna při uzavření. Lze si snadno představit, že v programu budeme mít několik oken též třídy, ale jen jedno z nich bude „hlavní“ a jeho uzavření bude znamenat ukončení programu.

Nová podoba třídy Vokno tedy může opět být



```
/* Soubor Kap14\03\vokno\Vokno.java */
package vokno;
import javax.swing.*;

public class Vokno extends JFrame {
    public Vokno() {
    }
}
```

Třída Prvni bude mít tvar

```
/* Soubor Kap14\03\Prvni.java */
import javax.swing.UIManager;
import java.awt.*;
import java.awt.event.*;
import vokno.*;

public class Prvni {
    public Prvni() { // Konstruktor aplikace
        Vokno vokno = new Vokno();
        Dimension obrazovka =
            Toolkit.getDefaultToolkit().getScreenSize();
        Dimension ro = new Dimension();
        ro.height = obrazovka.height/2;
        ro.width = obrazovka.width/2;
        vokno.setSize(ro);
        vokno.setLocation(ro.width/2, ro.height/2);
        vokno.setTitle("Naše první okno");
```

⁷⁴ Viz též www.java.cz.

```

vokno.setVisible(true);
vokno.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
        | System.exit(0); i
    );
}
public static void main(String[] args) {
    new Prvni();
}
}

```

Metoda `main()` pouze vytvoří instanci třídy `Prvni`, nic víc. V konstruktoru této třídy se vytvoří instance okna, třídy `Vokno`. Na první pohled to vypadá podivně – okno by mělo po opuštění konstruktoru zase zaniknout, protože zanikne jediný odkaz na ně. Nicméně okno bude existovat dál; o tom jsme ale už hovořili.

Poznamenejme, že část inicializaci později přesuneme do metody `jblInit()`, která se bude starat o inicializaci komponent vložených do okna.

Vzhled aplikace Okno našeho programu vypadá podobně jako jiná okna v prostředí, ve kterém pracujeme. To znamená, že např. pod Windows se podobá oknům ostatních programů pod Windows.

Vzhled oken v programu má na starosti třída `UIManager`; požadovaný vzhled můžeme předepsat pomocí její statické metody `UIManager.setLookAndFeel()`, které jako parametr předáme znakový řetězec se jménem třídy určující vzhled. Můžeme použít např. třídu

- `javax.swing.plaf.windows.WindowsLookAndFeel`, nastavující pro aplikaci vzhled obvyklý ve Windows,
- `javax.swing.plaf.motif.MotifLookAndFeel`, která pro aplikaci nastavuje vzhled odpovídající spíše zvyklostem Unixu.

Obvykle však chceme, aby se aplikace přizpůsobila prostředí, pod kterým ji spustíme. V tom případě zjistíme implicitní vzhled („Look and Feel“) pomocí statické metody `UIManager.getSystemLookAndFeelClassName()`.



Vzhled nastavujeme v metodě `main()` hlavní třídy aplikace ihned po spuštění programu. Upravíme ji proto následujícím způsobem:

```

public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(
            UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

```

```
    |
    new Prvni();
}
```

Metoda `setLockAndFeel()` může vyvolat výjimky několika typů; jestliže se to stane, vypíšeme o tom zprávu a pokračujeme, neboť aplikace může zpravidla běžet, i když se tato operace nepovede.

14.2 Používání komponent

Naším dalším cílem bude přidat do okna tlačítko, které způsobí ukončení programu. Než se k tomu ale dostaneme, budeme se muset seznámit s uspořádáním komponent v okně a s ošetřováním událostí v Javě.

Obsah okna	Okno se chová jako kontejner, který obsahuje další komponenty. Tento kontejner je typu <code>JPanel</code> a je dostupný pomocí metody <code>getContentPane()</code> . Jednotlivé komponenty pak do okna přidáváme pomocí metody <code>add()</code> tohoto panelu.
-------------------	--

Vkládáme komponenty do okna

Jedním z problémů, který museli tvůrci Javy vyřešit, byla skutečnost, že aplikace vytvořené v tomto jazyce poběží v mnoha různých prostředích na mnoha různých počítačích. Proto nám poskytli možnost, jak se vyhnout napevnou stanovenému rozvržení komponent v okně.

Správce uspořádání	Tuto možnost představuje tzv. správce uspořádání, Layout Manager. (Přesněji, jde o komponentu, která implementuje rozhraní <code>LayoutManager</code> .) Java obsahuje několik předdefinovaných tříd, které lze jako správce rozložení použít. Použití vybraného správce předepišeme pomocí metody <code>setLayout()</code> . Podívejme se na některé z nich.
---------------------------	---

BorderLayout	Tento správce uspořádání umísťuje komponenty ke stranám okna. Podívejme se nejprve, jak vložíme do okna jednu komponentu typu <code>JButton</code> (tlačítko):
---------------------	--

- Do této třídy okna přidáme datovou složku, která bude obsahovat odkaz na instanci správce rozložení (např. typu `BorderLayout`).
- Dále do této třídy přidáme datovou složku typu `JButton`, která bude obsahovat odkaz na přidávané tlačítko.
- Deklarujeme soukromou metodu `jbInit()`, ve které k oknu (k panelu, představujícímu jeho obsah) připojíme pomocí metody `setLayout()` správce rozložení.

- Pomocí metody `add()` vložíme do okna tlačítko nebo jinou komponentu.

Do metody `jbInit()` přesuneme také některé z inicializací okna. Deklarace třídy `Vokno` bude mít nyní tvar

```
/* Soubor Kap14\04\vokno\Vokno.java */
package vokno;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Vokno extends JFrame {
    BorderLayout bl = new BorderLayout(); // Správce
    JButton tlacitkol = new JButton(); // Tlačítko

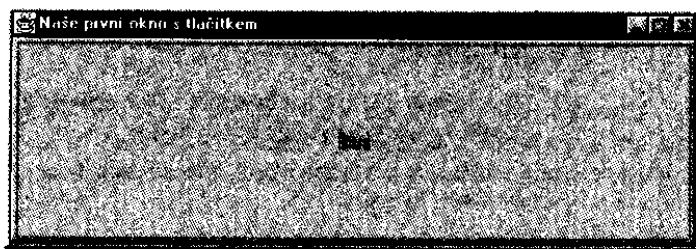
    public Vokno() {
        try {
            jbInit(); // Inicializace
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    private void jbInit() throws Exception // Metoda pro
    { // inicializaci komponent
        Dimension obrazovka = Toolkit.getDefaultToolkit().getScreenSize(); // To už známe
        Dimension ro = new Dimension();
        ro.height = obrazovka.height/2;
        ro.width = obrazovka.width/2;
        setSize(ro);
        setLocation(ro.width/2, ro.height/2);
        setTitle("Naše první okno s tlačítkem");
        tlacitkol.setText("Šlus");

        getContentPane().setLayout(bl); // Určí správce
        getContentPane().add(tlacitkol, BorderLayout.CENTER);
    }
}
```

Většinu z operací zde už známe. V předposledním řádku metody `jbInit()` získáme pomocí metody `getContentPane()`, zděděné po třídě `JFrame`, panel představující povrch okna a připojíme k němu správce uspořádání. V posledním řádku do něj vložíme komponentu `JButton` a předepíšeme, že ji chceme umístit do středu (`CENTER`). Výsledek – opět poněkud zmenšený, pokud jde o výšku okna – ukazuje obr. 14.3.

Další znakové řetězce, definované ve třídě `BorderLayout`, které předepisují umístění komponent, mají identifikátory `NORTH` (sever, tj. horní část komponenty), `SOUTH` (jih, tj. spodní část komponenty), `WEST` a `EAST` (západ a východ, tj. levá a pravá část komponenty).



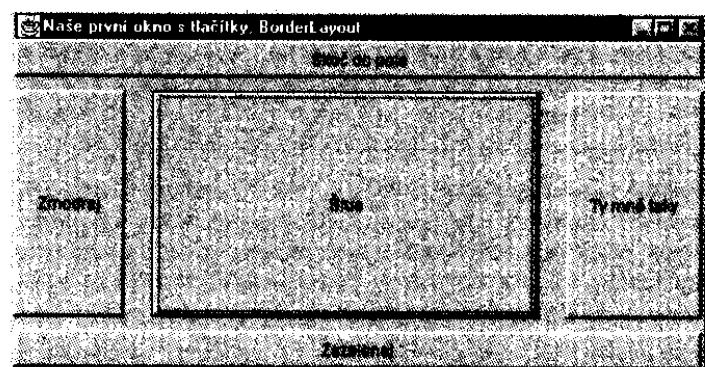
Obr. 14.3 BorderLayout: Jediná komponenta vyplňuje celé okno

Mezery mezi komponentami

Chceme-li, aby byly mezi komponentami mezery, zadáme jejich velikost jako parametry konstruktoru. Např. `BorderLayout(20, 10)` předpisuje, že mezi komponentami má být vzdálenost 20 pixelů vodorovně a 10 svisle. Jak to dopadne, vložíme-li do okna pět různých tlačitek s různými umístěními a s takovýmto mezerami, ukazuje obr. 14.4. (Zadáme-li několika komponentám stejné umístění, položí se přes sebe.)



Program v této podobě najdete na WWW v adresáři Kap14\04.



Obr. 14.4 BorderLayout a pět tlačítek v jednom okně

Poznamenejme, že v JDK 5 není třeba získávat odkaz na kontejner pomocí `getContentPane()`; komponenty lze vkládat metodou `add()` přímo do okna a to se již samo postará o vše potřebné. To znamená, že poslední dva příkazy metody `jbInit()` v předchozím příkladu můžeme v JDK 5 upravit takto:

```
private void jbInit() throws Exception
{
    // Ostatní příkazy se nemění
    setLayout(bl);                                // Určí správce
```

```
        add(tlacitko1, BorderLayout.CENTER);  
    }
```



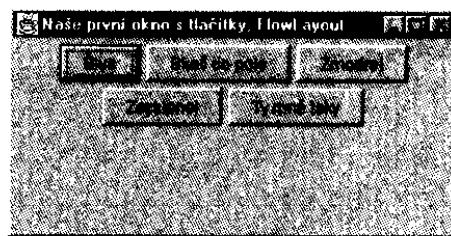
Předchozí program s takto upravenou metodou `jbInit()` najdete na WWW v adresáři Kap14\04a.

Podívejme se nyní na některá další uspořádání.

FlowLayout

Tento správce uspořádání ukládá komponenty na plochu okna zleva doprava a shora dolů v pořadí, v jakém je přidáváme. Komponenty budou mít implicitní velikost, což je v případě tlačítek nejmenší rozumná velikost vzhledem k nápisu na nich. Když se komponenty nevejdou do jedné řady, začne se další pod ní. V každé řadě jsou komponenty vycentrovány.

Stejných pět tlačítek vložených do stejného okna jako v předchozím případě, ale s umístěním `FlowLayout`, ukazuje obr. 14.5. Poznamenejme, že při vkládání do tohoto správce umístění zadáváme jako druhý parametr metody `add()` hodnotu `null`.



Obr. 14.5 FlowLayout uspořádá komponenty do řady za sebe

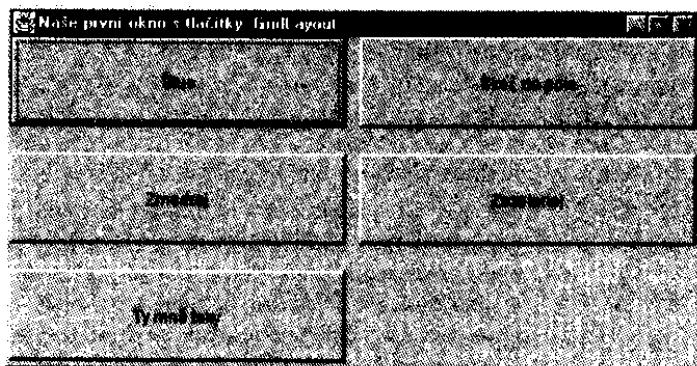
GridLayout

Tento správce uspořádání ukládá komponenty do mřížky, tedy do stejně velkých polí v okně. Pokud použijeme konstruktor bez parametrů, vytvoří se tolik polí, kolik bude třeba, ve vodorovné řadě za sebou. Jako parametry konstruktoru můžeme zadat počet řádků a sloupců mřížky a popřípadě vzdálenosti mezi komponentami ve vodorovném a svislém směru (v pixelech). Také při vkládání komponent do tohoto správce uspořádání zadáváme jako druhý parametr metody `add()` hodnotu `null`.

Obrázek 14.6 ukazuje, jak to dopadne, jestliže našich pět tlačítek vložíme do tohoto správce uspořádání vytvořeného příkazem

```
GridLayout bl = new GridLayout(3, 3, 10, 20);
```

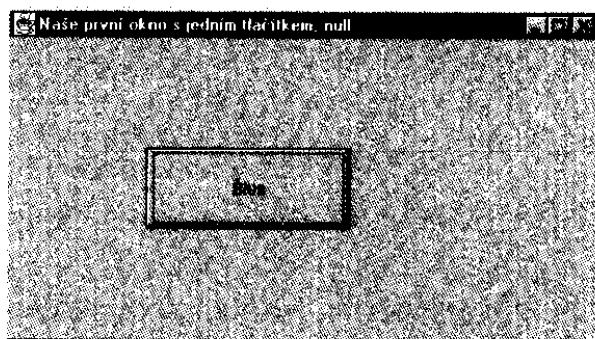
který předepisuje 3 řádky a tři sloupce a vzdálenost mezi komponentami 10 pixelů vodorovně a 20 pixelů svisle.



Obr. 14.6 Rozložení GridLayout

Sloupce „mřížky“ se zaplňují postupně zleva doprava; úplně prázdné sloupce se nezobrazí.

Pokud bychom chtěli přece jen přesně předepisovat umístění a rozložení jednotlivých komponent, zadáme místo správce uspořádání null.



Obr. 14.7 Jedno tlačítko, uspořádání null

Potom musíme u každé komponenty zadat oblast okna, ve které má být umístěna. K tomu poslouží metoda setBounds(), jejímž parametrem je obdélník, tj. instance třídy java.awt.Rectangle(). Například příkazem

```
tacitko.setBounds(new Rectangle(100, 80, 150, 60));
```

předepisujeme, že tlačítko (instance třídy JButton, kterou jsme používali v předchozích příkladech) má být v okně v obdélníku, jehož levý horní roh má souřadnice 100 a 80, délku 150 a výšku 60 pixelů. Velikost ani tvar této komponenty se nebude měnit, změníme-li za běhu programu velikost nebo tvar okna. Výsledek ukazuje obr. 14.7.

Učíme tlačítko fungovat

Zkusíte-li za běhu programu stisknout tlačítko vložené do okna, zjistíte, že se nic nestane. Musíme ho tedy naučit fungovat, tj. musíme se seznámit s obsluhou událostí v Javě.

- | | |
|-----------|--|
| Událost | Když stiskneme myší tlačítko v okně – a při mnoha jiných příležitostech – nastane v programu <i>událost</i> (anglicky event). Tuto událost vyvolá („odpálí“, fire) komponenta, ve které k ní došlo; jestliže tedy stiskneme myší tlačítko, vyvolá toto tlačítko událost typu <code>java.awt.event.ActionEvent</code> . |
| Posluchač | Každá komponenta, která může vyvolat událost, si udržuje seznam instancí, které může tato událost zajímat („posluchačů“, příjemců událostí, anglicky listener). O técto příjemcích se ovšem musí nějak dozvědět; to znamená, že instance, která chce přijímat zprávy o událostech, se musí jako posluchač <i>zaregistrovat</i> . K tomu slouží metody s obecným jménem <code>addXxxListener()</code> , kde Xxx určuje druh události. Pro nás bude nyní zajímavá metoda <code>addActionListener()</code> . |
| Handler | Obvykle chceme, aby výsledkem stisknutí tlačítka (nebo výsledkem jiné události) bylo volání nějaké metody, která se postará o „obsluhu“ této události – o reakci programu na ni. Z nedostatku lepšího označujeme tuto metodu anglickým slovem <i>handler</i> . |
| | Handler je typicky metodou okna (nebo panelu, appletu..., který obsahuje komponentu, jež událost vyvolala). Ovšem okna se zpravidla jako příjemci událostí neregistrují; běžně se používá řešení, které je sice na první pohled složitější, program je pak ale pružnější: Jako příjemce se registruje instance pomocné třídy, která implementuje rozhraní <code>java.awt.event.ActionListener</code> a která obsahuje metodu s předepsaným jménem <code>actionPerformed(ActionEvent)</code> . Tato metoda teprve zavolá handler. (To umožňuje mj. použít týž handler k obsluze událostí od pocházejících od několika různých komponent.) |

Podívejme se, jak to bude vypadat v našem případě. Do okna jsme přidali jedno tlačítko s nápisem „Slus“ a chceme, aby po jeho stisknutí program skončil. Do části kódu, která se stará o inicializaci okna, nejlépe do metody `jbInit()`, přidáme registraci posluchače:

```
tlačitko.addActionListener(  
    new java.awt.event.ActionListener()  
    {  
        public void actionPerformed(ActionEvent e)  
        { tlačitkoAkce(e); }  
    } );
```

Základem této konstrukce je volání `tlacitko.addActionListener()`. Jako skutečný parametr jí předáváme instanci anonymní třídy, která implementuje rozhraní `ActionListener`:

```
new java.awt.event.ActionListener() { tělo třídy }
```

Tělo třídy obsahuje jedinou metodu s povinným jménem `actionPerformed()`. Tato metoda zavolá handler – metodu okna, kterou jsme pojmenovali `tlacitkoAkce()` – a předá jí jako parametr objekt popisující vzniklou událost.

Handler – metoda třídy `Vokno` – prostě ukončí program:

```
void tlacitkoAkce(ActionEvent e) {
    System.exit(1);
}
```

Jestliže uživatel našeho programu stiskne tlačítko, vznikne v něm událost typu `ActionEvent` a toto tlačítko o tom rozešle zprávu všem zaregistrovaným posluchačům. To znamená, že zavolá metodu `actionPerformed()` oné nepojmenované instance nepojmenované pomocné třídy. Ta zavolá handler, metodu `tlacitkoAkce()`.

To je vše. Celý program najdete na WWW v adresáři `Kap14\05`.



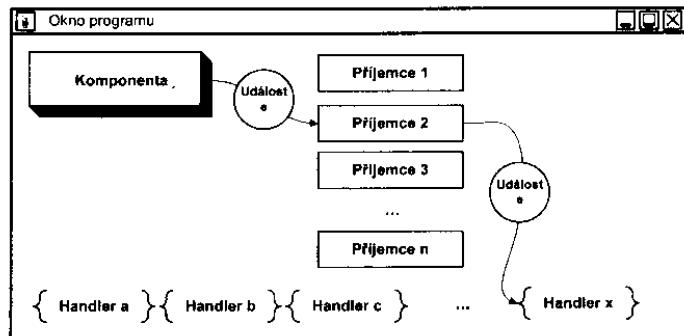
Události trochu podrobněji

V tomto odstavci shrneme, co jsme si dosud řekli, a přidáme další informace. Výklad ani potom nebude úplný – to bychom museli věnovat několik stran vnitřním třídám, anonymním potomkům a dalším věcem, které se do naší knihy nevešly –, ale povíme si vše, co je pro rutinní práci s událostmi nezbytné.

Používání komponent v programech s grafickým uživatelským rozhraním je založeno na událostech. Schéma práce s nimi v Javě naznačuje obr. 14.8.

Objekt události

Při vzniku události se vytvoří objekt typu `XxxEvent` (tedy `WindowEvent`, `ActionEvent` atd.,) který přenáší k posluchačům – a dále do handleru – podrobnější informace o ní. Všechny tyto třídy jsou odvozeny od třídy `java.awt.Event` a pomocí jejich metod můžeme zjistit např. instanci, která událost „odpálila“.



Obr. 14.8 Komponenta si udržuje seznam příjemců. Nastane-li událost, vytvoří objekt s informacemi u ní a předá ho příjemcům. Příjemce zavolá handler a událost mu opět předá

Každá komponenta, která může vyvolávat události, si vede seznam příjemců. K tomu jí slouží metody `addXXXListener()`, která přidá nového posluchače, a `removeXXXListener()`, která posluchače ze seznamu odstraní. Přitom `Xxx` je stále označení druhu události.

Příjemce je zpravidla instancí anonymní vnitřní třídy, která obsahuje jedinou metodu. Tato metoda zavolá handler (metodu, která událost ošetří). Handler je typicky metodou okna, v němž je uložena komponenta, která událost vyvolala, nikoli samotného příjemce.

Rozhraní a adaptér Instance, která chce být příjemcem události typu `xxxEvent`, musí implementovat rozhraní `XxxListener`. Toto rozhraní zpravidla obsahuje jedinou metodu, která je při události volána. Některá z těchto rozhraní ovšem obsahují více metod; abychom je nemuseli implementovat všechny, nabízí knihovna Swing tzv. adaptéry, předdefinované třídy, které lze použít místo těchto rozhraní a které definují implicitní verze všech metod tohoto rozhraní; nám pak stačí v odvozené třídě předefinovat jen metodu, kterou opravdu potřebujeme. Jejich použití při registraci handleru události se nicméně neliší od použití rozhraní.

Některé události Podívejme se na přehled některých událostí a komponent, které je vyvolávají. Všechny jsou deklarovány v balíku `java.awt.event`. Další informace i nich lze najít např. v dokumentaci k JDK.

ActionEvent Události `ActionEvent` odpovídá rozhraní `ActionListener`, které obsahuje metodu `actionPerformed(ActionEvent)`. K registraci posluchačů slouží metoda `addActionListener`, k jejich uvolnění metoda `removeActionListener()`. Tuto událost mohou vyvolávat komponenty `JButton`, `JTextField`, `JList`, `JMenuItem` a mnohé další. Odpovídá stisknutí tlačítka, vybrání položky nabídky atd.

Adjustment Event	Odpovídající rozhraní se jmenuje <code>AdjustmentListener</code> a obsahuje metodu <code>adjustmentValueChanged(AdjustmentEvent)</code> . Tuto událost může vyvolat komponenta <code>JScrollBar</code> a některé další. Odpovídá změně polohy táhla.
KeyEvent	Tuto událost mohou vyvolávat všechny třídy odvozené od společného předka, třídy <code>Component</code> . Jako posluchači se používají instance implementující rozhraní <code>KeyListener</code> nebo instance odvozené od adaptéra <code>KeyAdapter</code> , která toto rozhraní implementuje. Tato událost odpovídá stisknutí nebo uvolnění klávesy, pokud má daná komponenta fokus (je na ni směrován vstup z klávesnice). Rozhraní <code>KeyListener</code> obsahuje metody <code>keyPressed(KeyEvent)</code> , <code>keyReleased(KeyEvent)</code> a <code>keyTyped(KeyEvent)</code> .
MouseEvent	Také tuto událost mohou vyvolávat všechny třídy odvozené od třídy <code>Component</code> ; jako posluchače lze použít třídy implementující rozhraní <code>MouseListener</code> nebo odvozené od adaptéra <code>MouseAdapter</code> . Rozhraní <code>MouseListener</code> obsahuje metodu <code>mouseClicked(MouseEvent)</code> , která je volána při klepnutí myši, <code>mouseEntered(MouseEvent)</code> a <code>mouseExited(MouseEvent)</code> , které se volají, když kurzor myši vstoupí, resp. opustí plochu komponenty, a <code>mousePressed(MouseEvent)</code> a <code>mouseReleased(MouseEvent)</code> , které se volají při stisknutí, resp. při uvolnění tlačítka myši. Vedle toho lze pro práci s myší použít rozhraní <code>MouseMotionListener</code> nebo adaptér <code>MouseMotionAdapter</code> , které obsahují metody <code>mouseDragged(MouseEvent)</code> , resp. <code>mouseMoved(MouseEvent)</code> . Volají se při pohybu myši přes plochu komponenty se stisknutým tlačítkem, resp. se všemi tlačítky volnými.
WindowEvent	Událost <code>WindowEvent</code> může vyvolat instance třídy odvozené od <code>Window</code> – mj. <code>JFrame</code> nebo <code>JDialog</code> . Této události odpovídá jednak rozhraní <code>WindowListener</code> , jednak adaptér <code>WindowAdapter</code> . Událost <code>WindowEvent</code> vyvolá okno při uzavírání, otevírání, zmenšení do ikony, zvětšení z ikony atd. Rozhraní <code>WindowListener</code> obsahuje metody <code>windowOpened(WindowEvent)</code> , <code>windowClosing(WindowEvent)</code> , <code>windowClosed(WindowEvent)</code> , <code>windowActivated(WindowEvent)</code> , <code>windowDeactivated(WindowEvent)</code> , <code>windowIconified(WindowEvent)</code> a <code>windowDeiconified(WindowEvent)</code> .

14.3 Piškorky, verze 1.0

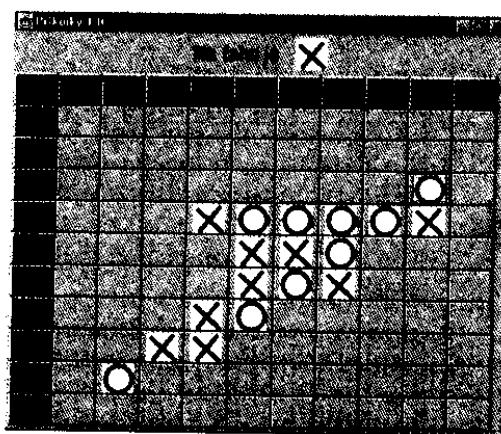
Nyní už víme dost, abychom se mohli pokusit o složitější projekt. Napíšeme program, který bude sloužit jako hrací plocha pro piškor-

ky⁷⁵. Určitě tu hru znáte; hraje se na čtverečkovaném papíru, jeden z hráčů dělá do prázdných čtverečků křížky, druhý kolečka a vyhrává ten, kdo udělá *piškorku* – pět svých symbolů vedle sebe v řadě, sloupci nebo na diagonále. Samozřejmě se v tazích střídají, každý vždy udělá jeden symbol.

Program, který by opravdu hrál piškorky a měl naději vás porazit, přesahuje svou složitostí rámec naší knihy; pokud byste ho přesto chtěli zkusit, můžete hledat inspiraci v [8], kde je mu věnována celá kapitola. My napíšeme pouze program, který bude představovat hrací plochu. Jeho okno bude obsahovat řadu políček, na která budou hráči střídavě klepat myší a program bude tato polička vyplňovat kroužky nebo křížky.

Požadavky

Než se pustíme do programování, rozmyslíme si, co vlastně chceme a jak to budeme dělat. Jinými slovy, ujasníme si požadavky, které budou na náš program kladeny, a postup a prostředky, které chceme použít.



Obr. 14.9 Hrajeme piškorky

- Náš program na počátku zobrazí okno obsahující $n \times n$ prázdných políček. Pro snazší orientaci zobrazí po levé straně čísla a nahore písmena (jako na šachovnici).

⁷⁵ Pokud tuto hru znáte pod názvem *piškvorky*, omlouvám se, ale já jsem hrával *piškorky*.

- Před každým tahem bude napovídat, který z hráčů je na řadě – zda ten, který používá křížek, nebo ten, který používá kolečko.
- Po kliknutí na políčko se v něm měl zobrazit příslušný symbol.

Později uvidíme, že tyto požadavky jsou nedostatečné, že program, který je splní, nebude ještě nijak dobrý. To je ale víceméně obvyklé až v průběhu vývoje zjišťujeme další požadavky a omezení.

Okno, které vytvoříme, ukazuje obr. 14.9.

Komponenty

Dvě různá umístění

Z uvedeného plyne, že potřebujeme rozdělit okno na dvě části; v horním, poměrně úzkém pruhu chceme zobrazovat nápis „Na tahu je“ a symbol, ve spodní potřebujeme $(n + 1) \times (n + 1)$ políček, která budou napodobovat čtverečkovaný papír. Přitom bychom chtěli, aby se velikost políček měnila při změně velikosti okna.

To znamená, že v horní části okna by nám vyhovoval správce umístění `FlowLayout`, ve spodní pak `GridLayout`. Pomůžeme si tak, že do okna vložíme dvě komponenty `JPanel` a teprve na ně budeme vkládat další komponenty. Na každém z panelů použijeme jiného správce umístění.

Ikony

Symboly hráčů (křížek a kolečko) si vytvoříme jako samostatné grafické soubory. Java podporuje internetové formáty GIF, JPG a PNG; pokud nemáte k dispozici editor, který by je uměl, můžete si je vytvořit v některém jiném formátu a pak je konvertovat např. pomocí sharewareového programu Graphic Workshop. Můžete také použít už hotové soubory `kruh.gif` a `kriz.gif`, které najdete na WWW v adresáři Kap14\06. Poznamenejme, že stačí obrázky velikosti např. 32×32 pixelů.



Abychom s nimi mohli v programu pracovat, musíme je „zabalit“ do instance třídy, která implementuje rozhraní `Icon`; použijeme třídu `javax.swing.ImageIcon`, jejímuž konstruktoru lze předat řetězec obsahující jméno souboru s obrázkem.

JButton a JLabel

Ikonu lze zobrazit na řadě komponent. Jednou z nich je `JLabel`, komponenta, která slouží ke zobrazení nápisu nebo obrázku (nebo obojího). Jinou možnost představuje `JButton`, který také může obsahovat jak nápis tak obrázek. My použijeme v horní části okna `JLabel`, ve spodní části, tj. na hrací ploše, `JButton`; jednotlivá políčka tedy budou představována tlačítka.

Program

Hlavní program V našem programu zatím definujeme dvě třídy – Piskorky (hlavní třídu aplikace) a Vokno (okno programu). Hlavní třída Piskorky se bude držet osvědčeného schématu:

```
/* Soubor Kap14\06\Piskorky.java */
import javax.swing.UIManager;
import java.awt.*;
import vokno.Vokno;

public class Piskorky
{
    public Piskorky()
    {
        Vokno okno = new Vokno();

        Dimension obr =
            Toolkit.getDefaultToolkit().getScreenSize();
        okno.setSize(
            new Dimension(obr.width/2, obr.height/2+50));
        okno.setLocation(obr.width/4, obr.height/4);

        okno.setVisible(true);
        okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] a)
    {
        try {
            UIManager.setLookAndFeel(
                UIManager.getSystemLookAndFeelClassName());
        }
        catch(Exception e) {
            e.printStackTrace();
        }

        Piskorky p = new Piskorky();
    }
}
```

Metoda `main()` nastaví vzhled aplikace a vytvoří instanci třídy `Piskorky`. Konstruktor této třídy vytvoří okno, nastaví jeho rozměry a polohu, zobrazí ho a nastaví jeho chování při uzavření pomocí systémové nabídky. Vše ostatní se bude odehrávat ve třídě `Vokno`.

Okno Třídu `Vokno` uložíme do balíku `vokno`. Podívejme se nejprve na její datové složky:

```
/* Soubor Kap14\06\vokno\Vokno.java */
package vokno;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```

public class Vokno extends JFrame
{
    final String titulek = "Piškorky 1.0";
    boolean tahneKruh = false;           // Kdo je na tahu
    int n = 10;                         // Rozměr hrací plochy
    Icon kruz = new ImageIcon("S_kriz.gif");
    Icon kruh = new ImageIcon("S_kruh.gif");
    JPanel okno;                      // Plocha okna
    JPanel panel1 = new JPanel();       // Vložené panely
    JPanel panel2 = new JPanel(new GridLayout(n+1,n+1));
    JLabel textKdoTahne = new JLabel("Na tahu je ");
    JLabel kdoTahne = new JLabel(tahneKruh ? kruh : kruz);
    JButton[][] policka = new JButton[n+1][n+1];
    BorderLayout bl = new BorderLayout(1,1);

```

Složka **Titulek** obsahuje text, který bude v záhlaví okna. Složka **kdoTahne** slouží k zapamatování, který z hráčů je na tahu; podle toho se bude zobrazovat v horní části okna symbol napovídající, kdo má táhnout, a podle toho se také bude určovat, kam jaký symbol se vykreslí na poličku, na které hráč klepne myši. Přidělime mu počáteční hodnotu `false`, která bude znamenat, že tähne hráč užívající křížek.

Proměnná **n** určuje rozměry hrací plochy ($n \times n$ poliček).

Další dvě datové složky, **kruz** a **kruh**, jsou typu **Icon** (jak víme, i jméno rozhraní lze použít jako jméno typu). Inicializujeme je nově vytvořenými instancemi třídy **ImageIcon**. Parametry konstruktu jsou jména souborů, které leží v adresáři s hlavním programem. (Nikoli v podadresáři **vokno**!)

Složka **okno** bude představovat **JPanel** s obsahem hlavního okna; následující dvě složky jsou panely, které do něj vložíme. Oba hned inicializujeme. Použijeme-li konstruktor bez parametrů, bude užívat správce umístění **FlowLayout**; pokud chceme jiného správce, zadáme ho jako parametr konstruktoru.

Složka **textKdoTahne** bude komponenta **JLabel**, která bude obsahovat nápis "Na tahu je", a **kdoTahne** je opět **JLabel**, který bude zobrazovat symbol hráče, jenž je na tahu. Už jsme se řekli, že komponenta **JLabel** může zobrazovat text i obrázek zároveň; v tom případě je ale obrázek nalevo od textu. My ovšem potřebujeme nejprve text a za ním obrázek, a proto použijeme dvě komponenty za sebou.

Předposlední datovou složkou je dvourozměrné pole tlačítek (komponent typu **JButton**). To budou tlačítka představující hrací plochu. Horní řada a levý sloupec navíc představují označení řádků a sloupců. Poslední složkou je správce umístění **BorderLayout**; použijeme ho v hlavním okně.

Konstruktor Konstruktor třídy Vokno pouze zavolá metodu jbInit().

```
public Vokno()
{
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Metoda jbInit() Metoda jbInit() se – jako vždy – stará o inicializaci komponent:

```
private void jbInit() throws Exception
{
    setTitle(titulek);
    okno = (JPanel)getContentPane();
    okno.setLayout(bl);

    okno.add(panel1, BorderLayout.NORTH);
    okno.add(panel2, BorderLayout.CENTER);

    panel1.add(textKdoTahne, null);
    textKdoTahne.setFont(
        new java.awt.Font("Dialog", Font.PLAIN, 20));
    panel1.add(kdoTahne, null);
    panel1.setBackground(new Color(200, 200, 200));
    hraciTlacitka();
}
```

První tři příkazy nastaví titulek (záhlaví) okna, získají odkaz na panel představující obsah okna a nastaví v něm správce umístění BorderLayout. (Vložíme-li do tohoto okna pouze dvě komponenty, jednu s umístěním NORTH a druhou s umístěním CENTER, bude první komponenta zabírat úzký pruh při horním okraji a druhá celý zbytek okna – a to je přesně to, co potřebujeme. Nahoře bude jen nápověda, ve zbytku hraci plocha.)

Pak do tohoto okna vložíme připravené dva panely. Připomeňme si, že první (horní) bude mít správce umístění FlowLayout, druhý bude mít správce umístění GridLayout.

Do horního panelu pak vložíme připravené komponenty textKdoTahne a kdoTahne. Implicitní velikost nápisu nám ovšem nevyhovuje, a proto bychom chtěli trochu zvětšit písmo. K tomu použijeme metodu setFont(), jejímž parametrem je instance třídy Font. Prvním parametrem konstruktoru třídy fontu je řetězec představující jeho jméno, druhým číslo vyjadřující, zda jde o normální písmo (Font.PLAIN), tučné (Font.BOLD) nebo kurzivu (Font.ITALIC) a třetím je jeho velikost v bodech.

Nakonec ještě změníme barvu pozadí horního panelu. K tomu použijeme metodu `setBackground()`, jejímž parametrem je instance třídy `java.awt.Color`. Parametry konstruktoru jsou celá čísla v rozmezí od 0 do 255 a vyjadřují podíl červené, zelené a modré ve výsledné barvě.⁷⁶

Posledním příkazem je volání metody `hraciTlacitka()`, která se postará o vytvoření hrací plochy.

Logická jména fontů

Vratíme se ještě k písmu. V Javě pracujeme pouze s tzv. logickými jmény fontů (druhů písma). Má-li být vytvořený program přenositelný do libovolného jiného prostředí, nemůže spoléhat na to, že v něm bude mít k dispozici určitý přesně předepsaný font. Proto zpravidla specifikujeme pouze obecnou kategorii (DialogInput – neproporcionalní písmo používané v dialogových oknech, SansSerif – bezpatkové písmo, Serif – patkové písmo, a několik dalších). Přiřazení logických jmen fontů skutečným písmům je věcí konkrétní instalace Javy.

Příprava tlačítek

Metoda `hraciTlacitka()` má za úkol vytvořit $(n + 1) \times (n + 1)$ tlačítek, umístit je do spodního panelu okna, na tlačítka na levém a horním okraji umístit nárazy označující řádky a sloupce a pro ostatní tlačítka definovat handlery, které se postarají o vykreslení patřičného obrázku při stisknutí.

```
private void hraciTlacitka()
{
    Color barvaOkraje = new Color(0xA0, 0xA0, 0xA0);
    for(int i = 0; i < n+1; i++)
        for(int j = 0; j < n+1; j++)
    {
        JButton b = new JButton();
        policka[i][j] = b;
        panel2.add(b, null);
        if(i != 0 && j != 0)b.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    tlacitkoNandejIconu();
                }
            });
    }
}
```

⁷⁶ Jak známo, složíme-li červené, zelené a modré světlo o stejně intenzitě, dostaneme světlo bílé. Na tom je založeno vytváření barev na monitoru počítače. `Color(0,0,0)` vyjadřuje černou barvu, `Color(255,255,255)` bílou, `Color(255,0,0)` jasně červenou, `Color(0,255,0)` jasně zelenou a `Color(0,0,255)` jasně modrou. Zadáme-li tři stejné parametry menší než 255, dostaneme různé stupně šedé, tři různá čísla povedou k různým jiným barvám. Vedle toho můžeme pro označení barev použít předdefinované hodnoty `Color.red` (červená), `Color.black` (černá) a další.

```

}
for(int i = 1; i < ntl; i++)
{
    policka[i][0].setBackground(barvaOkraje);
    policka[i][0].setText(""+i);
    policka[0][i].setBackground(barvaOkraje);
    policka[0][i].setText(""+(char)('A'+i-1));
}
policka[0][0].setBackground(barvaOkraje);
}

```

Okrrajová políčka by se měla na první pohled lišit od ostatních, neboť na nich nelze hrát. Obarvíme je tedy trochu jinak než ostatní. V prvním řádku metody `hracillacitka()` si proto připravíme barvu.

Pak ve dvou do sebe vnořených cyklech vytvoříme potřebný počet tlačítek (komponent `JButton`) a přidáme je do spodního panelu. Nejde-li o tlačítko na levém nebo horním okraji (je-li `i` nebo `j` nenulové), zaregistrujeme pro něj posluchače (příjemce událostí).

Nakonec v dalším cyklu projdeme okrajová tlačítka, změníme pomocí metody `setBackground()` jejich barvu a pomocí metody `setText()` na ně vložíme odpovídající nápis. U tlačítka po levé straně to bude číslo, u horní řady to bude písmeno.

Metoda `setText()` očekává jako parametr znakový řetězec; číslo `i` parametr cyklu – převedeme na řetězec pomocí už známé konstrukce `""+i`. V případě znaků využijeme toho, že písmena anglické abecedy tvoří v Unicode souvislou řadu, tj. za `'A'` následuje `'B'` atd. Protože znaky lze používat jako čísla s hodnotou odpovídající kódu znaku, představuje výraz s hodnotou `'A'+i-1` kód i-tého velkého písmene (`'A'` je první). Přetypováním na `char` z tohoto čísla dostaneme odpovídající znak a sečtením s prázdným řetězcem ji převedeme na `String`.

Poslední cyklus vynechává políčko v levém horním rohu, neboť do něj nechceme zapsat ani písmeno, ani číslici. Proto jeho barvu změníme samostatným příkazem.

Uzavření okna Aby nás program korektně skončil při uzavření okna, musíme zavolat metodu `setDefaultCloseOperation(EXIT_ON_CLOSE)` nebo použít jinou z možností popsaných v 14.1; to ale už známe.

Stisknutí tlačítka Nakonec musíme ještě napsat handler, který se postará o zobrazení ikony na stisknutém tlačítku. Je samozřejmě nemyslitelné, abychom psali zvláštní handler pro každé tlačítko: Bude-li mít hrací plocha rozměr 10×10 políček, museli bychom psát 100 handlerů, které budou všechny dělat totéž.

Parametr handleru, objekt e typu ActionEvent, v sobě ale naštěstí nese informaci o tom, kde tato událost vznikla. Získáme ji pomocí metody getSource(), která vrací odkaz na původce (v našem případě na tlačítko, které uživatel programu stiskl). Tento odkaz je ovšem typu Object, takže ho musíme přetypovat.

Handler tedy bude mít tvar

```
protected void tlacitkoNandejIkonu(ActionEvent e)
{
    ((JButton)e.getSource()).setIcon(tahneKruh ? kruh : kriz);
    tahneKruh = ! tahneKruh;
    kdoTahne.setIcon(tahneKruh ? kruh : kriz);
}
```

V prvním příkazu získáme původce události a nastavíme na něm ikonu; který obrázek to bude, to určíme podle hodnoty proměnné tahneKruh. Pak změníme hodnotu proměnné tahneKruh na opačnou a následně změníme obrázek v náhovědě v horní části okna.



Tím jsme hotovi; program v této podobě najdete na WWW v adresáři Kap14\06. Výsledek ukazuje obr. 14.9.

14.4 Piškorky, verze 2.0

Předchozí program lze opravdu využít jako hrací plochu pro piškorky, má ale mnoho vad. Řekněme si alespoň o některých.

- Chceme-li začít novou hru, musíme program ukončit a znova spustit. Nás program by měl umožňovat vymazat současný stav a začít znova poněkud pohodlnějším způsobem.
- Program umožňuje hrát i na obsazeném poli. To je špatné, měl by zabránit v přepsání obsahu jednou použitého políčka.
- Program by měl umožnit používat jiné obrázky, změnit mezi hrami velikost hrací plochy a vrátit chybný tah.
- Běžné programy umožňují vyvolat dialogové okno s informacemi o programu.

Vývoj požadavků

Jde o vlastně o typický příklad vývoje programu. Původní požadavky se po implementaci ukázaly jako nedostatečné, po prvním použití zákazník teprve začíná zjišťovat, co vlastně chtěl. Při úvodní specifikaci totiž zákazníkovi (ale často i vývojářům) připadá jako samozřejmé spousta věcí, které ve skutečnosti vůbec samozřejmě nejsou; nutnost blokovat políčko, na kterém už je značka jednoho z hráčů, je poměrně typickým příkladem.

Možnost vrácení posledního chybného tahu, ale i ostatní požadavky ukazují, že k programu budeme muset připojit buď nástrojový panel s tlačítky, nebo nabídku (menu).

Nástrojový panel bychom si mohli vytvořit z komponent JButton a JPanel, jako jsme to udělali s upozorněním, kdo je na tahu. Je to ale zbytečné, neboť knihovna Swing nabízí hotovou komponentu JToolBar.

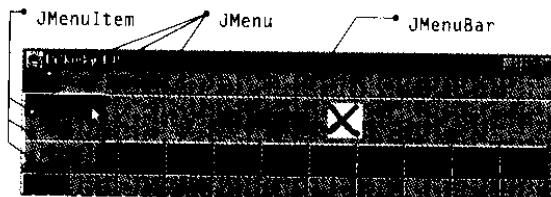
My ovšem použijeme tradiční nabídku (menu), abychom si ukázali, jak se s ní zachází.

Přidáváme nabídku

Nabídku vytvoříme pomocí několika komponent (viz též obr. 14.10):

- JMenuBar je komponenta, která vytváří nabídkový pruh okna. Slouží jako kontejner na jednotlivé nabídky (komponenty JMenu). Připojíme ji k oknu pomocí metody setMenuBar().
- JMenu je komponenta, která vytváří hlavní nabídku (nápis v nabídkovém pruhu okna). Vložíme ji do nabídkového pruhu pomocí metody add().
- Jednotlivé položky nabídky, které se rozvinou po vybrání nabídky v nabídkovém pruhu, představují komponenty JMenuItem. Do hlavní nabídky je vložíme pomocí její metody add().

Text nabídky zadáme jako parametr konstruktoru nebo pomocí metody setText().



Obr.14.10 Komponenty, které vytvářejí nabídky

Do našeho programu přidáme tři hlavní nabídky – Hra, Volby a Nápo- věda. První z nich bude mít položky Nová, Zpět a Konec. Druhá bude mít jedinou položku, Nastavení..., a třetí O programu...⁷⁷ To zna-

⁷⁷ Tři tečky za položkou nabídky obvykle naznačují, že po vybrání této nabídky se objeví dialogové okno.

mená, že do deklarace třídy `Vokno` přidáme odpovídající datové složky:

```
JMenuBar pruhNabidky = new JMenuBar();           // Nabídkový pruh
JMenu nabidkaHra = new JMenu("Hra");             // Nabídka Hra a
JMenuItem hraNova = new JMenuItem("Nová");        // její položky
JMenuItem hraZpet = new JMenuItem("Zpět");
JMenuItem hraKonec = new JMenuItem("Konec");

JMenu nabidkaVolby = new JMenu("Volby");
JMenuItem nastaveni = new JMenuItem("Nastavení...");

JMenu nabidkaNapoveda = new JMenu("Nápověda");
JMenuItem oProg = new JMenuItem("O programu...");
```

Jejich připojení k oknu uložíme do zvláštní metody, kterou zavoláme z metody `jbInit()`.

```
private void nabidky()
{
    setJMenuBar(pruhNabidky); // Připoj nabídkový pruh k
oknu

    pruhNabidky.add(nabidkaHra); // Vlož do něj hlavní nabídky
    pruhNabidky.add(nabidkaVolby);
    pruhNabidky.add(nabidkaNapoveda);

    nabidkaHra.add(hraNova);    // Vlož do jednotlivých
    nabidkaHra.add(hraZpet);    // nabídek jejich položky
    nabidkaHra.addSeparator();
    nabidkaHra.add(hraKonec);

    nabidkaVolby.add(nastaveni);
    nabidkaNapoveda.add(oProg);
}
```

Mezi položku *Zpět* a *Konec* vložíme vodorovnou čáru – separátor, který bude naznačovat, že následující příkaz je významově poněkud odlišný. K tomu nám poslouží metoda `JMenu.addSeparator()`.

Jestliže nyní program přeložíme a spustíme, uvidíme něco podobného jako na obr. 14.10. Vybrání položky nabídky ovšem zatím nebude mít žádný účinek, příslušné akce musíme ještě naprogramovat. Postup bude podobný jako v případě stisknutí tlačítka: Komponenta `JMenuItem` po vybrání způsobí událost `ActionEvent`, takže zaregistroujeme okno jako posluchače a napíšeme odpovídající handler.

Ukončení programu

Začneme tím, co už vlastně známe – ukončením programu. Nejprve musíme zaregistrovat příjemce události, která nastane při vybrání této položky. Na konec metody `nabidky` připojíme tento příkaz:

```
hraKonec.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent e) {
    nabidkaKonec_Akce(e);
}
```

Pak napišeme handler, metodu třídy Vokno, kterou jsme nazvali nabidkaKonec_Akce():

```
protected void nabidkaKonec_Akce(ActionEvent e)
{
    System.exit(0);
}
```

Překladem a spuštěním se přesvědčíme, že tento příkaz již funguje. Nicméně okamžité ukončení programu není to nejlepší, co můžeme udělat – je to nevratná akce, a vybereme-li ji omylem, můžeme zkazit pěkně rozehranou partii. Rozumný program se nejprve zeptá, zda chceme opravdu skončit.

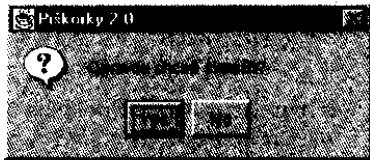
K tomu bychom si mohli vytvořit zvláštní dialogové okno. Knihovna Swing nám však nabízí řadu předdefinovaných okenek, které lze vyvolat pomocí statických metod třídy JOptionPane; nám se bude hodit showConfirmDialog(), neboť chceme, aby uživatel potvrdil svůj úmysl ukončit program.

```
protected void nabidkaKonec_Akce(ActionEvent e)
{
    int i = JOptionPane.showConfirmDialog(this,
        "Opravdu chceš skončit?", titulek,
        JOptionPane.YES_NO_OPTION);
    if(i == JOptionPane.YES_OPTION) System.exit(0);
}
```

První parametr představuje odkaz na komponentu, která toto okno vyvolala, druhý, resp. třetí je nápis v okně, resp. v záhlaví, a poslední určuje, jaká tlačítka se mají v okně objevit. Máme na vybranou mj. hodnoty YES_NO_OPTION (tlačítka Ano a Ne), YES_NO_CANCEL_OPTION (tlačítka Ano, Ne a Storno) a OK_CANCEL_OPTION (tlačítka OK a Storno). Metoda vrátí hodnotu odpovídající stisknutému tlačítku, tj. např. YES_OPTION, pokud uživatel stiskne tlačítko Ano (Yes).

Jestliže takto upravený program přeložíme a spustíme, objeví se po zvolení nabídky Konec okno, které vidíte na obr. 14.11.

Podobně bychom mohli ošetřit i uzavření okna pomocí systémové nabidky (stisknutím tlačítka s křížkem v pravém horním rohu). Museli bychom však změnit chování okna při uzavření pomocí metody setDefaultCloseOperation().



Obr. 14.11 Smíme skončit? Knihovní dialogová okna mají na tlačítkách anglické nápisy

Zpět

Nyní se podívejme, co je třeba udělat, aby mohl hráč vrátit poslední tah: Po každém tahu si musíme zapamatovat políčko, na které hráč umístil svůj symbol. Proto do třídy Vokno přidáme datovou složku

```
 JButton posledni = null;
```

```
a do metody tlacitkoNandejikonu() příkaz
```

```
posledni = (JButton)e.getSource();
```

kterým si zapamatujeme tlačítko, s nímž se naposledy hrálo. Dále musíme samozřejmě zaregistrovat příjemce zprávy a vytvořit handler. To znamená, že na konec metody nabidky() přidáme příkaz

```
hraZpet.addActionListener(new ActionListener() { // Zpět
    public void actionPerformed(ActionEvent e) {
        nabidkaZpet_Akce(e);
    }
});
```

a ve třídě Vokno definujeme metodu nabidkaZpet_Akce(), která se postará o vrácení tahu. Přitom nastavený obrázek na tlačítku zrušíme tím, že mu jako odkaz na novou ikonu zadáme null. Zároveň se postaráme, aby si program pamatoval, že už jeden krok vrátil: do proměnné posledni uložíme null. Na začátku vždy zkонтrolujeme, zda posledni neobsahuje null, a pokud ano, neuděláme nic.

Poslední, o co se musí tato metoda postarat, je změna hodnoty booleovské proměnné kdoTahne a odpovídající změna ikony, zobrazované v návodě v horní části okna.

```
protected void nabidkaZpet_Akce(ActionEvent e)
{
    if(posledni != null) {
        posledni.setIcon(null);
        posledni = null;
        tahneKruh = !tahneKruh;
        kdoTahne.setIcon(tahneKruh ? kruh : kriz);
    }
}
```

Nová hra

Chceme-li začít novou hru, musíme ze všech políček vymazat značky, které tam hráči při předchozí hře nakladli. K tomu stačí zavolat pro každé z políček (tlačítek) metodu `setIcon()` s parametrem `null`. Dále musíme vrátit původní hodnotu proměnné `tahneKruh`, která řídí nápo-vědu, a podle ní upravit zobrazení nápo-vědy, kdo je na tahu. Také proměnné poslední **přiřadíme null**, neboť ihned po začátku nemá vrácení tahu smysl.

Jako obvykle musíme registrovat příjemce události,

```
hraNova.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        nabidkaNova_Akce(e);  
    };
```

a pak napsat handler:

```
protected void nabidkaNova_Akce(ActionEvent e)  
{  
    for(int i = 0; i < n; i++) // Odstraň ikony z tlačítek  
    for(int j = 0; j < n; j++)  
        policka[i+1][j+1].setIcon(null);  
    tahneKruh = false; // Začíná křížek  
    kdoTahne.setIcon(tahneKruh ? kruh : kriz);  
    posledni = null; // Zpět nyní nemá smysl  
}
```

Okno O programu

Toto okno většinou obsahuje název programu, ikonu programu nebo logo výrobce a informace o držiteli autorských práv. Také zde můžeme s výhodou využít předdefinovaného dialogového okna ze třídy `JOptionPane`. Vyvoláme ho pomocí metody `showMessageDialog()`, které předáme jako parametry odkaz na komponentu, která ho vyvolala, řetězec představující nápis v okně, řetězec představující nápis v titulku okna, příznak druhu okna a ikonu, kterou chceme v okně zobrazit. (Můžete použít vlastní, nebo třeba soubor `strom.gif`, který najdete na WWW v adresáři Kap14\07.)



Úpravy programu tedy budou velmi jednoduché: Do metody `nabidky()` přidáme registraci handlu:

```
oProg.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        nabidkaOProg_Akce(e);  
    };
```

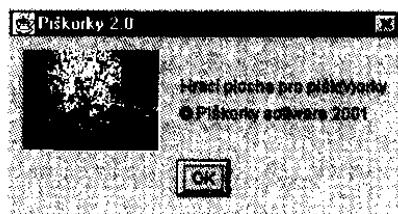
do třídy `Vokno` přidáme handler, tj. metodu

```

protected void nabidka0Prog_Akce(ActionEvent e)
{
    JOptionPane.showMessageDialog(this,
        "Hrací plocha pro pišk(v)orky"+
        "\n© Piškorky software 2001", "Piškorky 2.0",
        JOptionPane.PLAIN_MESSAGE, new ImageIcon("strom.gif"));
}

```

Poznamenejme, že nápis v tomto okně může (na rozdíl od nápisu v komponentě `JLabel`) obsahovat i přechod na nový řádek. Výsledek ukazuje obr. 14.12.



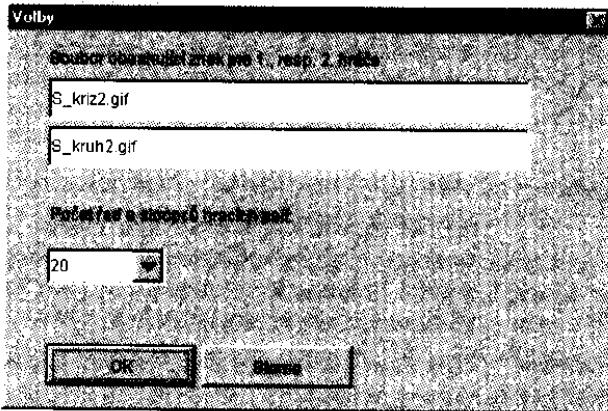
Obr. 14.12 Také okno O programu lze vytvořit velice jednoduše, pokud se spokojíme s běžným písmem

Volby

Nyní nás čeká poněkud těžší úkol – vytvořit dialogové okno, ve kterém budeme moci zadávat různá nastavení. Chtěli bychom měnit počet políček v okně a symboly, které hráči používají. I tentokrát bychom mohli využít předdefinované okno, které lze vyvolat pomocí statické metody `JOptionPane.showOptionDialog`; my si však potřebné okno vytvoříme sami. Protože mnohé z věcí už známe, budeme postupovat rychleji.

Datové složky

Třídu tohoto okna pojmenujeme `OknoVoleb` a deklarujeme ji jako potomka předdefinované třídy `javax.swing.JDialog`. Nejprve se podívejte, jak bude toto okno vypadat (obr. 14.13); to nám usnadní další výklad. Okno voleb bude obsahovat dvě textová pole (komponenty `JTextField`, které nazveme `obrazek1` a `obrazek2`), dvě tlačítka (komponenty `JButton`, které pojmenujeme `ok` a `storno`), jeden kombinovaný seznam (komponenta `JComboBox`, kterou pojmenujeme `kombo`) a dva nápisy (komponenty `JLabel`, které pojmenujeme `cancel` a `canc?`).



Obr. 14.13 Dialogové okno s volbami pro piškorky

Vedle uvedených komponent deklarujeme ve třídě okna veřejně přístupné statické konstanty DOLNI_MEZ, HORNÍ_MEZ a MINIMUM s hodnotami 10, 20 a 5. Nejmenší počet řad a sloupů hracích políček bude 5 (MINIMUM), neboť na menší pole se nevejdou žádná piškorka. Největší počet bude dán hodnotou HORNÍ_MEZ. Konstanta DOLNI_MEZ bude určovat rozmezí, které bude nabízet kombinovaný seznam. Třída Vokno si tyto hodnoty bude brát odtud.

Datová složka stiskOK typu boolean bude obsahovat informaci o tom, zda uživatel uzavřel toto okno stisknutím tlačítka OK nebo Storno.

Kromě toho budeme potřebovat dvě pole typu Object[], která pojmenujeme volby a rozmery. Pole volby bude obsahovat aktuální nastavení voleb vyjádřené jako pole objektů, pole rozmery bude obsahovat hodnoty, které chceme zobrazit v kombinovaném seznamu.

Konstruktor Nyní se podíváme na konstruktor této třídy. Už jsme řekli, že jako jeden z parametrů budeme předávat odkaz na pole obsahující aktuální nastavení voleb. První dvě složky tohoto pole budou řetězce představující jména grafických souborů se symboly jednotlivých hráčů, třetí bude na instanci typu Integer obsahující počet řádků a sloupců v hracím poli.

Dalším parametrem bude odkaz na „vlastník“, okno, které toto dialogové okno vytvořilo.

Modální okno V grafických uživatelských rozhraních se rozlišují dva druhy dialogových oken: *modální* a *nemodální*. Nemodální okno se chová podobně jako běžné okno programu – můžeme ho ponechat otevřené a přitom přejít do jiného okna, pracovat v něm a pak se zase vrátit. Modální

okno obvykle představuje žádost programu o informace, bez kterých nemůže rozumně pokračovat, a proto nedovoluje přepnutí do jiného okna. Okna s volbami jsou zpravidla modální, a ani naše okno nebude výjimkou. Jeho „modalitu“ předepíšeme v konstruktoru jeho třídy při volání konstruktoru předka:

```
super(vlastník, "Volby", true);
```

Prvním parametrem konstruktoru předka je odkaz na komponentu, která toto okno vytvořila, druhým je nápis v titulkové liště a třetím je logická hodnota, která určuje, zda bude toto okno modální. (Jinou možnost představuje volání metody `setModal(true)`.)

Uzavření systémovým menu

Dále musí konstruktor určit, jak se bude okno chovat při uzavření. K tomu poslouží metoda `setDefaultCloseOperation()`, které zadáme parametr `JDialog.HIDE_ON_CLOSE`. To znamená, že při uzavření pomocí systémové nabídky se okno nezruší, ale jen skryje.⁷⁸

Do okna voleb budeme potřebovat přenést aktuální nastavení a z něj zpět nové hodnoty, nastavené uživatelem. Asi nejjednodušší způsob, jak to zařídit, je deklarovat ve třídě `Vokno` vhodné pole typu `Object[]`, do něj jednotlivá nastavení uložit a toto pole předat jako parametr konstruktoru.⁷⁹

Konstruktor

Konstruktor třídy `OknoVoleb` bude tedy vypadat takto:

```
public OknoVoleb(Frame vlastník, Object[] vol)
{
    super(vlastník, "Volby", true);
    volby = vol;
    setDefaultCloseOperation(JDialog.HIDE_ON_CLOSE);
    try {
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Nejdřív v něm voláme konstruktor předka; to je naprosto nezbytné. O parametrech tohoto volání jsme již hovořili. Pak do proměnné `volby` uložíme odkaz na pole obsahující aktuální nastavení a předepíšeme, jak se má toto dialogové okno chovat, jestliže ho někdo uzavře pomocí systémové nabídky. Nakonec zavoláme metodu `jbInit()`, která se postará o inicializaci komponent.

⁷⁸ To je ve skutečnosti implicitní chování.

⁷⁹ Vzhledem k tomu, že se bude předávat pouze odkaz na toto pole, budou změny, které v něm provedou metody okna voleb, viditelné i ve třídě `Vokno`.

Metoda jbInit()

Metoda `jbInit()` se postará o nastavení počátečních rozměrů okna, o nastavení správce umístění, o vložení komponent a o zaregistrování handlerů; to už známe, takže se tím nebudeme podrobněji zabývat. Jediné, co je pro nás nové, je používání komponent `JTextField` a `JComboBox` (kombinovaného seznamu).

Textové pole

Text v textovém poli je dán vlastností `Text`, takže ho zjistíme pomocí metody `getText()` a vložíme ho pomocí metody `setText()`.

Kombinovaný seznam

Konstruktoru kombinovaného seznamu zadáváme pole instancí třídy `Object[]` obsahující položky, které bude tento seznam po stisknutí tlačítka se šípkou dolů nabízet. Pokud nám žádná nebude vyhovovat, můžeme požadovanou hodnotu zapsat do textového pole seznamu.

Toto pole typu `Object[]` jsme již deklarovali a nazvali jsme ho rozemy. Protože potřebujeme, aby obsahovalo celá čísla v rozmezí daném veřejně přístupnými konstantami `DOLNI_MEZ` a `HORNI_MEZ`, uložíme do něj odpovídající instance typu `Integer`. K tomu poslouží tyto příkazy:

```
rozmery = new Object[HORNI_MEZ-DOLNI_MEZ+1];
for(int i = DOLNI_MEZ; i <= HORNI_MEZ; i++){
    rozmery[i-DOLNI_MEZ] = new Integer(i);
}
```

Pak zavoláme konstruktor, určíme rozměry a polohu komponenty v okně a předepíšeme, že položky lze editovat – tj. že do textového pole můžeme zapsat i jinou hodnotu, než je v seznamu.

```
kombo = new JComboBox(rozmery);
kombo.setBounds(new Rectangle(30, 160, 87, 26));
kombo.setEditable(true);
```

Nakonec určíme, že tato komponenta bude nabízet jako vybranou hodnotu nejvyšší prvek seznamu:

```
kombo.setSelectedIndex(rozmery.length-1);
```

Tlačítka

Nakonec musíme napsat handlery, které se postarájí o reakci programu na stisknutí tlačítka OK, resp. Storno.

Při stisknutí tlačítka Storno vrátíme do komponent v dialogovém okně pokud možno původní hodnoty, nastavíme příznak `stiskOK` na false a nakonec okno skryje příkazem `setVisible(false)`:

```
void stornoActionPerformed(ActionEvent e) {
    stiskOK = false;
    obrazek1.setText((String)volby[0]);
    obrazek2.setText((String)volby[1]);
    kombo.setSelectedIndex(rozmery.length-1);
    this.setVisible(false);
}
```

V případě, že uživatel stiskne tlačítko OK, přeneseme hodnoty z komponent do pole volby, uložíme do stiskOK hodnotu true a také skryjeme okno:

```
void ok_actionPerformed(ActionEvent e) {
    volby[0] = obrazek1.getText();
    volby[1] = obrazek2.getText();
    int n = Integer.parseInt(
        kombo.getSelectedItem().toString());
    n = Math.max(MINIMJM, n);
    volby[2] = new Integer(Math.min(HORNI_MEZ, n));
    stiskOK = true;
    this.setVisible(false);
}
```

Při přenosu hodnot z kombinovaného seznamu kontrolujeme, zda uživatel nezadal hodnotu mimo povolené rozmezí, a pokud ano, nahradíme ji nejbližší dovolenou hodnotou.

Změny ve třídě Vokno

Ve třídě Vokno si na počátku vytvoříme pole aktualniNastaveni, které bude obsahovat počáteční nastavení voleb:

```
int n = OknoVoleb.HORNI_MEZ;
Object[] aktualniNastaveni = {obr1, obr2, new Integer(n)};
```

Jak pracovat s dialogem

Při práci s dialogovým oknem (ale i s jinými okny) máme dvě možnosti: Můžeme si ho vytvořit jednou provždy při spuštění programu a pak ho jen zobrazovat a skrývat pomocí vlastnosti `Visible`, nebo ho můžeme pokaždé znova vytvářet a rušit. My použijeme první možnost, neboť tak z něj budeme moci kdykoli získávat informace, které v něm budou uloženy.

To znamená, že si okno voleb se ve třídě Vokno vytvoříme na hned počátku, v průběhu inicializací, a jeho konstruktorem předáme aktualniNastavení:

```
vokno.OknoVoleb vol = new vokno.OknoVoleb
    (this, aktualniNastaveni);
```

V handleru⁸⁰, který bude reagovat příkaz Nastavení z nabídky, prostě toto okno budeme zobrazovat a skrývat pomocí metody `setVisible()`.

```
protected void nabidkaVolby_Akce(ActionEvent e)
{
    vol.setVisible(true);
    if(vol.getStiskOK())
```

⁸⁰ Parametr `e` je v následující implementaci handleru zbytečný, nikde ho nepoužijeme. Přesto ho zde uvádíme, neboť při pozdějších úpravách bychom ho mohli potřebovat – obsahuje údaje o původci události ap.

```

        r = ((Integer)aktualniNastaveni[2]).intValue();
        obr1 = (String)aktualniNastaveni[0];
        obr2 = (String)aktualniNastaveni[1];
        kruz = new ImageIcon(obr1);
        kruh = new ImageIcon(obr2);
        pocatecniNapoveda();
        panel12.removeAll();
        this.repaint();
        hraciTlacitka();
    }
}

```

Okno nejprve zobrazíme. Protože je modální, bude tato metoda počkat, až když ho uzavřeme; proto se hned v zápisu můžeme zeptat, zda uživatel uzavřel okno stiskem tlačítka OK. Pak si vyzvedneme z pole aktualniNastaveni nové hodnoty a uložíme je do odpovídajících proměnných; protože toto pole obsahuje odkazy na typ `Object`, musíme je přetypovat. Vytvoříme objekty typu `ImageIcon`, pomocí metody `removeAll()` odstraníme všechny komponenty z panelu s hracími tlačítky, pomocí metody `repaint()` si vyžádáme překreslení obsahu okna a pomocí metody `hraciTlacitka()` vytvoříme novou hrací plochu.



Zdrojový text tohoto programu najdete na WWW v adresáři Kapi4\0/ a v jeho podadresáři vokno.

14.5 Piškorky, verze 2.1

I když jsme původní program citelně vylepšili, přece jen má stále řadu vad. Asi nejprotivnější z nich je, že při zadávání nového souboru v okně voleb musíme vypisovat jméno, popřípadě i cestou. To je nejen nepříjemné, ale i může to i snadno vést k chybám. Slušné programy umožňují vyhledat potřebný soubor pomocí dostatečně chytřeho dialogového okna.

Takové okno je naštěstí v knihovně Swing již připraveno. Výsledek bude sice smíšený česko-anglický, ale i tak bude stát zato.

Komponenta JFileChooser

Komponenta představující dialogové okno pro výběr souboru se jmenuje `JFileChooser` a její základní obsluha je velice jednoduchá:

- Vytvoříme instanci pomocí bezparametrického konstruktoru.
- Pomocí metody `SetCurrentDirectory()` nastavíme adresář, ve kterém chceme začít. Parametrem této metody je objekt třídy `File`.

- Podle hodnoty vrácené metodou `showOpenDialog()` zjistíme, zda uživatel stiskl tlačítko Open (otevřít) nebo Cancel (storno).
- Metoda `getSelectedFile()` vrátí jméno souboru jako instanci třídy `File`.

Třída `JFileChooser` obsahuje také další metody, které umožňují na stavit nápis na některých tlačítkách, v záhlaví okna atd.

Nás další postup tedy bude jednoduchý: Do okna voleb přidáme dvě tlačítka (`JButton`), která pojmenujeme `volic1` a `volic2`. Umístíme je za textová pole a jako nápis na ně dáme tři tečky. Pro obě tato tlačítka zaregistrujeme společný handler, který se postará o reakci na stisknutí.

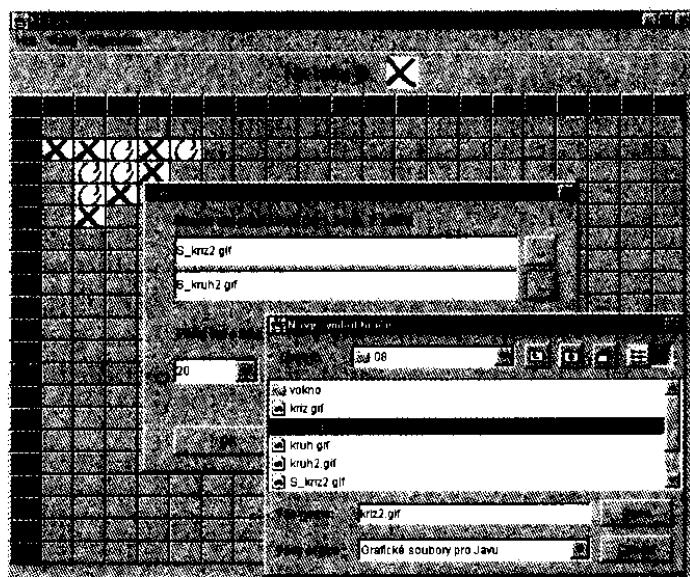
```
void volba_Akce(ActionEvent e)
{
    JTextField pole = // Pro které textové pole?
        e.getSource().equals(volic1) ? obrazek1 : obrazek2;
    File soubor = new File(".");
    JFileChooser vyberSoubor = new JFileChooser();
    vyberSoubor.setCurrentDirectory(soubor);
    vyberSoubor.setDialogTitle("Nový symbol hráče");
    int vysledek = vyberSoubor.showOpenDialog(this);
    if(vysledek == JFileChooser.APPROVE_OPTION)
    {
        pole.setText(vyberSoubor.getSelectedFile().getPath());
    }
}
```

V prvním řádku si zjistíme, které tlačítko uživatel stiskl, a do proměnné `pole` si uložíme odkaz na odpovídající textové pole. Pak vytvoříme instanci třídy `File` představující aktuální adresář. (Jak v Unixu, tak i ve Windows se označuje tečkou.)

V následujících dvou příkazech vytvoříme instanci třídy `JFileChooser` a nastavíme ji na aktuální adresář. Poté změníme titulek v záhlavi okna, okno zobrazíme a hodnotu, vrácenou metodou `showOpenDialog()` uložíme do pomocné proměnné `vysledek`.

Je-li tato hodnota rovna konstantě `JFileChooser.APPROVE_OPTION`, znamená to, že uživatel stiskl tlačítko Open. V tom případě pomocí metody `getSelectedFile()` získáme instanci třídy `File` představující zvolený soubor a pomocí metody `getPath()` třídy `File` pak řetězec představující jméno souboru včetně cesty. To je vše. Výsledek ukazuje obr. 14.14.

 Program v této podobě najdete na WWW v adresáři Kap14\08.



Obr. 14.14 Volba souboru pomocí předdefinovaného okna

A co dál

Program lze samozřejmě vylepšovat dál a dál. Zde jsou některé možnosti – zkuste to sami:

- K programu lze doplnit nástrojový panel, který bude obsahovat tlačítka pro rychlou volbu nejčastěji používaných operací.
- Program by měl umožňovat uložit do souboru stav rozehrané partie a znova ho načíst.
- Program by měl kontrolovat, zda zadané soubory existují, a pokud ne, upozornit uživatele.
- Program by měl umožňovat vrátit větší počet kroků.
- Dialogové okno pro volbu souborů by mělo zobrazovat jen grafické soubory v podporovaných formátech (.gif, .jpg, .png).

Většinu potřebných věcí znáte; některé – např. jak zajistit, aby se v okně pro volbu souborů zobrazovaly jen soubory s určitými příponami – si budete muset zjistit z dokumentace. (Řešení, které najdete na WWW v souboru Kap14\08, je založeno na třídě `Filtr`, odvozené od třídy `javax.swing.filechooser.FileFilter`.)

15 Aplet

Aplet je miniaplikace napsaná v Javě, umístěná do HTML stránky a spouštěná ve WWW prohlížeči. Svého času patřily k nejčastějším programům vytvářeným v Javě; sloužily k oživení webových stránek, k upoutání pozornosti, ale i k vytváření seriózních aplikací založených na WWW stránkách.

Jejich doba, jak se zdá, už pominula, přesto si o nich alespoň velmi stručně povíme.

15.1 Struktura apletu

Aplet je podobně jako aplikace tvořen jednou nebo několika třídami. Základem je třída odvozená od knihovní třídy `javax.swing.JApplet`, která představuje prázdné okno apletu. Vytváření apletu je podobné jako programování jakékoli jiné aplikace s grafickým uživatelským rozhraním, až na to, že

- na rozdíl od „normální“ aplikace nemusí mít aplet metodu `main()`.
- nesmí pracovat se soubory ani jinak zasahovat do systému na počítači, na němž běží.

Aplet tedy nemusí (a zpravidla nemá) metodu `main()`. Zato by měl mít metody `init()`, `destroy()`, `start()`, `stop()` a některé další.⁸¹

Metody apletu

Aplet dostane při spuštění vyhrazenou jistou plochu WWW stránky, a tu musí zaplnit. Běží jako součást webového prohlížeče, nikoli jako normální aplikace; prohlížeč volá podle potřeby jeho metody. Podívejme se na ty, které pro nás mohou být nějak zajímavé. Pokud výslově neřekneme něco jiného, budou všechny následující metody veřejně přístupné (`public`).

Konstruktor Volá ho prohlížeč při zavedení apletu do paměti (po natažení WWW stránky, která aplet obsahuje). Obvykle inicializuje nestatické pro-

⁸¹ V JDK 1.x se jako společný předek pro aplety používala třída `java.applet.Applet`. Odvozená třída také musela obsahovat metodu `paint()`.

měnné apletu. Na rozdíl od „normálních“ aplikací nevolá metodu `jbInit()`.

`void init()` Tuto metodu zavolá prohlížeč ihned po konstruktoru. Jejím úkolem je postarat se o inicializaci apletu. V klasickém pojetí tak trochu dublovala funkci konstruktoru. Typicky načte parametry apletu z HTML stránky a pak zavolá metodu `jbInit()`, jež se postará o inicializaci komponent.

`void destroy()` Metoda `destroy()` představuje v jistém smyslu opak metody `init()` – připravuje aplet k zániku. Prohlížeč ji volá těsně před uvolněním apletu z paměti. V jistém smyslu nahrazuje destruktor známý např. z C++ nebo z Object Pascalu. Nemusí obsahovat žádné akce.

`void start()` Spustí činnost apletu. Tuto metodu zavolá prohlížeč na počátku po metodě `init()` a pak ji volá vždy, když se aplet vrátí na plochu okna prohlížeče. Pokud např. aplet předvádí nějakou animaci, tato metoda ji spustí. Jejím opakem je metoda `stop()`.

`void stop()` Tato metoda zastaví činnost apletu. Prohlížeč ji zavolá vždy, když aplet opustí plochu jeho okna – např. proto, že se uživatel přesune na stránce o kus dále. Pokud aplet předvádí nějakou animaci, metoda `stop()` ji zastaví, protože je zbytečné, aby tato animace zatěžovala systém, když ji uživatel stejně nevidí.

Když se později uživatel vrátí zpět a plocha apletu se znova objeví na obrazovce, zavolá prohlížeč metodu `start()` a tím obnoví činnost apletu. Jestliže uživatel opustí HTML dokument, který tento aplet obsahoval, zavolá prohlížeč nejprve metodu `stop()`, která aplet zastaví, a teprve pak zavolá `destroy()`.

`String getAppletInfo` Tato metoda vrací znakový řetězec obsahující informace o apletu, které bychom mohli použít např. v dialogovém okně O programu, pokud bychom aplet program spustili jako samostatnou aplikaci. Aplety obvykle nic podobného nenabízejí – používání dialogových oken se v apletech nedoporučuje.

`String getParameter` Hlavice této metody má tvar `String getParameter(String jmeno)` a má za úkol přečíst hodnotu parametru uvedeného v příkazu PARAM v HTML stránce. Poznamenejme, že v řetězci `jmeno` se nerozlišují malá a velká písmena.

`void jbInit()` Podobně jako v běžných aplikacích se tato metoda i v apletech stará o inicializaci komponent. Obvykle ji deklarujeme jako soukromou (`private`) a mohou se z ní rozšířit výjimky několika typů.

```
void paint()
```

Tato metoda byla velice důležitá u apletů v JDK 1.x (odvozených od třídy `Applet`). Volala se vždy, když bylo potřeba překreslit obsah okna – např. proto, že uživatel část plochy apletu zakryl jiným oknem nebo když si to program vyžadal voláním metody `repaint()`. V apletech vytvořených v JDK 1.2 se již zdaleka tak často nepoužívá, neboť komponenty JavaBeans se o překreslování své plochy starají samy. Nic nám samozřejmě nebrání definovat ji také, pokud to potřebujeme (např. chceme-li v našem apletu kreslit myší).

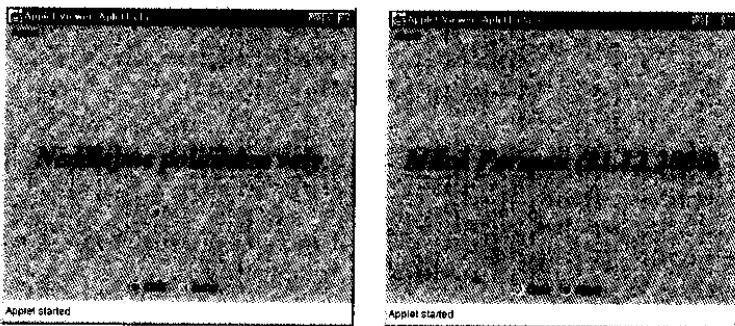
Hlavička této metody má tvar `void paint(java.awt.Graphics g)`, kde `Graphics` je třída představující obecnou „kreslicí plochu“.⁸² Její metody umožňují kreslit běžné geometrické tvary, vkládat text atd. Podrobnější informace o kreslení (a příklady) najdete např. v [3], [4] nebo [12].

Příklad

Jako příklad napišeme jednoduchý aplet, který bude zobrazovat nějaký vhodný nebo nevhodný citát a jeho autora. Jako implicitní nápis (nápis, který použijeme, pokud se parametry nepodaří přečíst) i jako hodnotu parametrů apletu použijeme citáty z knihy Murphyho zákon [9].

Co budeme potřebovat

Naše představa je jednoduchá: V horní části plochy apletu bude nápis, u dolního okraje dva přepínače a stisknutím jednoho z nich budeme volit, zda se má zobrazovat citát nebo jeho autor. (Viz též obr. 15.1.)



Obr. 15.1 Dva různé stavy našeho apletu v appletvieweru

Nápis zobrazíme pomocí komponenty `JLabel`, kterou už známe. K přepínání použijeme komponenty `JRadioButton`. U nich je obvyklé,

⁸² Pokud znáte základy programování pro Windows, pak stačí říci, že jde o analogii kontextu zařízení (device context).

že fungují ve skupinách a když jeden vybereme, ostatní „vyskočí“ – přestanou být vybrány. Tyto přepínače nám pomůže sdružit do skupiny nevizuální komponenta JButtonGroup, do které přidáme jednotlivé přepínače pomocí metody add().

Umístění komponent na ploše apletu vyřešíme podobně jako v případě piškorek: Na ploše apletu použijeme správce uložení BorderLayout a nápis vložíme doprostřed (CENTER). Do spodní části (SOUTH) vložíme pomocný panel se správcem FlowLayout a do něj vložíme oba přepínače. (Komponenta JButtonGroup se na plochu nepřidává.)

Datové složky apletu

Začátek deklarace třídy apletu tedy může být

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class Aplet1 extends JApplet {
    String citat = "Chaos vládne i bez ministrů.";
    String autor = "Bobbyho přesvědčení";
    JLabel napis = new JLabel(); // Nápis (citat nebo autor)
    JPanel pomocny = new JPanel(); // Pomocný panel
    ButtonGroup bg = new ButtonGroup(); // Skupina přepínačů
    JRadioButton zobrazCitat = new JRadioButton();
    JRadioButton zobrazAutora = new JRadioButton();
```

Proměnné citat, resp. autor obsahují text citátu a jeho autora. Na počátku jím přidělíme hodnotu, kterou použijeme, pokud se nepodaří přečíst parametry apletu v HTML stránce.

Metody

Konstruktor třídy Aplet1 ponecháme prázdný, proto ho zde ani nebude uvádět. Podívejme se rovnou na metodu init():

```
public void init() { // Inicializace
    try {
        citat = getParameter("VYROK", citat);
        autor = getParameter("AUTOR", autor);
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Zde nejprve přečteme parametry a pak zavoláme metodu jbInit(). K inicializaci nápisu se použije zde přečtená hodnota proměnné citat. Pak zavoláme jbInit().

Metoda jbInit()

```
private void jbInit() throws Exception {
    JPanel panel = (JPanel)getContentPane();
    this.setSize(new Dimension(400,300));
```

```

napis.setText(citat);
napis.setFont(new java.awt.Font // (1)
("Serif", Font.ITALIC | Font.BOLD, 20));
napis.setForeground(Color.black); // (2)
napis.setHorizontalAlignment(SwingConstants.CENTER);
panel.add(napis, BorderLayout.CENTER);

panel.add(pomocny, BorderLayout.SOUTH);

bg.add(zobrazCitat); // Přidej přepínače do skupiny
bg.add(zobrazAutora);
pomocny.add(zobrazCitat); // a vlož je na spodní panel
pomocny.add(zobrazAutora);
zobrazCitat.setText("Citát");
zobrazCitat.setSelected(true);
zobrazAutora.setText("Autor");

// Instance příjemce události // (3)
ActionListener prijemce = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        stiskPrepinace(e);
    }
};

zobrazCitat.addActionListener(prijemce);
zobrazAutora.addActionListener(prijemce);
!
```

První dva příkazy nastaví velikost plochy okna apletu na 300×400 bodů a získají panel představující obsah tohoto okna. Pak následuje skupina příkazů, které připraví nápis a vloží ho do okna. V příkazu označeném číslem (1) v komentáři si všimněte výrazu

`Font.ITALIC | Font.BOLD`

který znamená, že chceme písmo, které bude jak tučné tak kurziva. V příkazu označeném (2) voláme metodu `setForeground()`, která nastaví barvu písma.

Pak do okna vložíme panel pomocny; v jeho konstruktoru jsme neuvedli správce uložení, takže bude mít FlowLayout. Následují příkazy, kterými přidáme přepínače zobrazCitat a zobrazAutora do skupiny bg. Vzápětí je také vložíme na panel pomocny. Pomocí metody `setText()` také určíme popisky těchto přepínačů (nápis „Citát“ a „Autor“). Volání metody `setSelected()` přepínače zobrazCitat určí, že na počátku bude vybrán přepínač zobrazCitat.

V příkazu, označeném (3), vytvoříme instanci anonymní třídy implementujícím rozhraní `ActionListener`, a v následujících dvou příkazech ji zaregistrujeme jako příjemce událostí od obou přepínačů.

Nakonec definujeme handler, který se postará o reakci na vybrání jednoho z přepinačů. Protože je pro oba přepínače společný, musí nejprve určit, který přepínač událost vyvolal, a pak nastaví odpovídající text:

```
protected void stiskPrepinace(ActionEvent e) {  
    if(e.getSource().equals(zobrazCitat))  
        napis.setText(citat);  
    else  
        napis.setText(autor);
```

Nejspíš jste si ale všimli, že metoda `getParameter()`, kterou voláme v metodě `jbInit()`, má dva parametry, zatímco standardní metoda popsaná v úvodu má jeden parametr. Ve třídě apletu totiž deklarujeme svou vlastní přetíženou verzi, která se pokusí přečíst parametr, a pokud neuspěje, vrátí implicitní hodnotu, uloženou ve druhém parametru:

```
public String getParameter(String key, String def)  
{  
    return  
        (getParameter(key) != null ? getParameter(key) : def);  
}
```

To je vše. Všimněte si, že jsme nepotřebovali metody `start()` a `stop()`, neboť aplet neobsahuje animaci ani jiné procesy, které by zatěžovaly systém, pokud není zobrazen. Nepotřebovali jsme také metodu `destroy()`, neboť aplet nepoužívá žádné prostředky, které by musel před ukončením vracet.

Hotový program najdete na WWW v souboru `Kap15\01\Aplet1.java`.



Aplet spuštěný jako aplikace

Třída apletu může obsahovat metodu `main()`. Takový aplet můžeme spustit i jako obyčejnou aplikaci.

V takovém případě musíme v metodě `main()` vytvořit instanci apletu a zavolat metody `init()` a `start()`. (Běží-li nás program jako aplet, postará se o to prohlížeč.)

Při programování takovýchto „kočkopsů“ ale musíme mít na paměti, že aplet podléhá omezením, která se normálních aplikací netýkají – nesmí např. pracovat se soubory na cílovém počítači.

15.2 Spuštění apletu

Aplet vkládáme do HTML stránek pomocí značky <APPLET>. Nejednodušší stránka, ze které spustíme náš první aplet, může vypadat např. takto:⁸³

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;
                                              charset=windows-1250">
<TITLE>
Testovací stránka pro první aplet
</TITLE>
</HEAD>
<BODY>
Můj oblibený výrok a jeho autor.<BR>
<APPLET>
  CODEBASE = "."
  CODE    = "Aplet1.class"
  NAME   = "TestAplet"
  WIDTH  = 400
  HEIGHT = 300
  HSPACE = 0
  VSPACE = 0
  ALIGN   = top
>
<PARAM NAME="VYROK" VALUE="Směj se. Zítra bude hůř.">
<PARAM NAME="AUTOR" VALUE="Murphyho filozofie">
</APPLET>
</BODY>
</HTML>
```

Vše za značkou <APPLET> až po ukončení, tedy po značku </APPLET>, je zadání apletu. CODEBASE zde určuje adresář, ve kterém má prohlížeč třídy apletu hledat. Tečka znamená aktuální adresář, tedy adresář, ve kterém je daná HTML stránka. NAME je jméno, kterým bude tento aplet označován v hlášeních některých prohlížečů. HSPACE, resp. VSPACE, znamená velikost mezery, kterou prohlížeč vynechá okolo plochy apletu, a ALIGN znamená zarovnání apletu ve stránce; hodnota middle říká, že chceme umístit aplet doprostřed stránky, top znamená umístění nahoru na stránku.

Specifikace parametrů začíná slovem PARAM. Pak následuje klíčové slovo NAME určující jméno parametru, rovnítko a jméno v uvozovkách. Pod tímto jménem bude aplet hledat tento parametr pomocí funkce getParameter(). Pak následuje klíčové slovo VALUE a za rovníkem hodnota parametru.



Tento soubor najdete na WWW v adresáři Kap15\Aplet1.html.

⁸³Podrobnější informace o HTML, určené pro začátečníky najdete např. v [10].

Problémy

Nebuděte ale překvapeni, jestliže běžné prohlížeče tento aplet nespustí; nepodporují totiž novější verze JDK. Aby v nich váš aplet běžel, je třeba použít doplňky k nim; podrobnější informace lze najít na WWW na adrese java.sun.com nebo v knize [6].

AppletViewer

Součástí JDK je také program `appletviewer.exe`, který najdete ve stejném adresáři jako překladač nebo interpret Javy. Slouží k ladění aplétů. Pokud systémová proměnná `PATH` na vašem počítači obsahuje cestu do tohoto adresáře, spustíme ho příkazem

```
appletviewer url
```

kde `url` je internetová adresa souboru HTML obsahujícího aplet. Pokud je na stejném počítači a v aktuálním adresáři, stačí jméno souboru, např.

```
appletviewer Aplet1.html
```

Literatura

- [1] J. Gosling, B. Joy, G. Steel: *The Java Language Specification*. Addison-Wesley 1996. ISBN 0-201-63451-1.
- [2] B. Eckel: *Thinking in Java*. Prentice Hall 2000.
ISBN 0-13-027363-5. (Český překlad *Myslime v jazyce Java*, Grada Publishing 2000, ISBN 80-247-9010-6.)
- [3] P. Herout: *Učebnice jazyka Java*. Kopp, České Budějovice 2000.
ISBN 80-7232-115-3;
P. Herout: *Java – grafické uživatelské prostředí a čeština*. Kopp, České Budějovice 2000. ISBN 80-7232-150-1;
P. Herout: *Java – bohatství knihoven*. Kopp, České Budějovice 2000. ISBN 80-7232-209-5.
- [4] D. Štrupl, M. Virius: JBuilder verze 3. Grada Publishing 1999.
ISBN 80-7169-890-3.
- [5] D. E. Knuth: *The Art of the Computer Programming*. Vol. 1–3. Addison-Wesley. (Tato kniha existuje v řadě vydání od 70. let po současnost. K dispozici je také ruský překlad *Iskusstvo programirovaniya dlja EVM*.)
- [6] H. M. Deitel, P. Deitel: *Java – How to program*. Prentice Hall, 1999. ISBN 0-13-012507-5
- [7] *Software Fault Tolerance*. . Editor M. R. Lyu. John Wiley & Sons, New York 1995. ISBN 0-471-95068-8
- [8] R. Pecinovský, M. Virius: *Práce s daty* 2. Grada Publishing 1997. ISBN 80-7169-470-3.
- [9] Bloch, A.: *Murphyho zákon*. Svoboda-Libertas, Praha 1993.
ISBN 80-205-0322-3.
- [10] Kosek, J.: *HTML – tvorba dokonalých WWW stránek*. Grada Publishing, Praha 1998. ISBN 80-7169-608-0.
- [11] Horstmann, C. S., Cornell, G.: *Core Java* 2. Volume 1 – Fundamentals. 7th edition. Prencice Hall 2004. ISBN 0-13-148202-5.
- [12] Horton, I.: *Beginning Java* 2. SDK 1.4. edition. Wrox, 2002.
ISBN 1-861005-69-5.

Rejstřík

@

@Override, 158

A

abstract, 165
adaptér, 233
adresa, 8
algoritmus, 11
anotace
 @Override, 158
aplet, 256
argument, 146
ArrayList, 164
aserce, 135
assert, 135; 169; 170
atribut, 15
autoboxing, *viz* balení
automatická správa paměti, 54;
 108
AWT, 218

B

bajt, 8
balení, 98
balík, 81
 celosvětová konvence, 81
 implicitní, 84
 java.lang, 84
 jméno, 81
 kvalifikace, 83
barva, 240
bit, 8
blok, 120
 hlídaný, 171
 obklopující, 121
 vnitřní, 121
 vnořený, 120

boolean, 97
byte, 86

C

catch, 171
CLASSPATH, 28; 81
Cloneable, 186
cyklus, 41; 126
 podminka opakování, 126
tělo, 126

Č

čeština, 35
číslo
 celé, 86
 magické, 49
 reálné, 94

D

datové složky
 initializace, 142
 nestatické, 142
 statické, 143
dědění, 17
dědičnost, 20; 157. *viz* dědění
deklarace
 pole, 100
 proměnné, 110
 třídy, 30; 138
dokumentace, 32
 vytvoření, 32
double, 94

E

enum, 106

F

faktoriál, 40; 90
false, 42; 97
filtr, 202
final, 49; 110; 166
font, 239; 240
for
 v JDK 5, 131
funkce
 matematické, 96

G

garbage collector, 54; 108

H

handler, 171

C

char, 92
chyba
 běhová, 58
 syntaktická, 26
zaokrouhlovací, 95

I

identifikátor, 9; 78
 a čeština, 93
 konvence, 79
if, 43; 122
import, 83
 statický, 84
index, 100
instance, 15
 odkaz, 52
 vytvoření, 52
int, 86

iterátor, 164; 198

J

Java, 10; 21; 27
internetové zdroje, 21
tíedy, 23
vizuální vývojová prostředí,
 22
vztah k C++, 23
javac
 a čeština, 93
javadoc, 32
javský virtuální stroj, 10
jazyk programovací, 9
 interpretace, 10
 Java, 10
 překlad, 10
jazyk UML, 16
JDK, 21
JVM, 10

K

kanál, 200
klíčové slovo, 80
klonování, 186
knihovna, 14
kód
 strojový, 9
 zdrojový, 10; 25
komentář, 29; 78
 dokumentační, 29
 vnořování, 78
kompilátor, 10
komponenta
 vlastnost, 221
konstanta, 49; 110
 celočíselná, 87
 výčtová, 107
konstruktor, 50
 volání konstruktoru předka,
 159
kontejner, 59
konverze
 specifikace, 37
kvalifikace, 53

L

Layout Manager, 226
l-hodnota, 119
long, 86
Look and Feel, 225

M

main(), 30; 153
Math, 96
menu. viz nabídka
metoda, 15; 144
 abstraktní, 20; 165
 addXxxListener(), 231; 233
 clone(), 104
 deklarace, 145
 equals(), 60; 65
 generická. viz
 parametrisovaná
 hlavička, 46
 jbInit(), 224
 konečná, 166
 main(), 30
 parametrisovaná, 194
 printf(), 37; 210
 předdefinovaná, 158
 přetěžování, 51
 removeXxxListener(), 233
 s proměnným počtem
 parametrů, 156
 setDefaultCloseOperation(),
 220
 statická a nestatická, 52
 System.arraycopy(), 103
 System.exit(), 219
 System.gc(), 109
 System.out.print(), 37
 System.out.println(), 37
 toString(), 68
 typ, 145
 volání, 48
 volání metody předka, 158
metoda shora dolů, 12
metoda zdola nahoru, 14
mezí typového parametru, 196
MojeIO, 38

N

nabídka, 243
 separátor, 244
návěští, 124; 125
neúplné vyhodnocení, 97
new, 53
nový řádek, 35
null, 55

O

objekt, 15
 skládání, 16
 ukládání, 206
obrázky
 použití v programu, 236
obrazovka
 rozměry, 222
odkaz, 52
 přiřazení, 54
okno, 222
 modální, 250
 souřadnice, 223
operační systém, 11
operátor
 !, 97
 &, 98
 &&, 62; 97
 |, 98
 ||, 97
 ++ a --, 45
 =, 36; 118
 konjunkce, 62
 neúplné vyhodnocení, 62
 neúplné vyhodnocení, 97; 98
 new, 53
přetypování, 66
přiřazení, 36
relační, 41
složené přiřazení, 44

P

package, 82
paměť
 automatická správa, 54

operační, 7; 8
vnější, 7
parametr, 146
předávání hodnotou a
odkazem, 146
typový, 191
meze, 196
piškorky, 235
počítač, 7
podmínka, 41; 122; 126
spojování, 62
vstupní, 135
pole, 100
index, 100
kopírování, 103
třídění, 12
polymorfismus, 19
printf(), 37
procesor, 7
program, 9
běh, 27
java, 27
javac, 26
javadoc, 32
pravidla pro zápis, 80
překlad, 26
proměnná, 9
CLASSPATH, 28; 81
deklarace, 36; 110
 inicializace, 36
lokální, 150
proud, 200
překladač, 10
přetečení
celočíselní, 90
přetěžování, 51
přetykování, 66
příkaz
assert, 135; 169; 170
blok, 120
cyklu, 126
for-each, 131
if, 43; 122
import, 83
package, 82
podmíněný, 122
prázdný, 120; 121
return, 47
složený, 120

výrazový, 121
while, 41
přiřazení, 36; 118

R

RAM, 7
real, 94
relační operátory, 41
return, 47
rozhodování, 43
rozhraní, 180
 ActionListener, 233
 AdjustmentListener, 234
 Cloneable, 182
 Collection, 198
 Comparable, 183; 193; 195
 Icon, 236
 Iterator, 182
 KeyListener, 234
 MouseListener, 234
 MouseMotionListener, 234
 parametrizované, 193
 Runnable, 182
 Serializable, 182; 207
 WindowListener, 234

Ř

řetězec, 31
formátovací, 37
spojování, 35
řídící posloupnost, 92

S

separátor, 244
Serializable, 207
serializace, 206
setDefaultCloseOperation(), 220
seznam, 73
short, 86
skener, 211
soubor, 200; 202
vytvoření, 203
způsob otevření, 204
specifikace konverze, 37; 210

správce uspořádání, 226
static, 143; 144
statický inicializátor, 144
strojový kód, 9
středník, 120
super, 158; 159
synchronized, 145

T

test is a - has a, 21
throw, 169
throws, 58
true, 42; 97
try, 171
třída, 14
abstraktní, 20
ActionEvent, 233
AdjustmentEvent, 234
anonymní, 138
ArrayList, 59
Byte, 91
Color, 240
datová složka, 15
dědičká hierarchie, 19
dědičnost, 18
deklarace, 30
Error, 169
File, 202
FilterInputStream, 202
FilterOutputStream, 202
Font, 240
generická viz
parametrizovaná
ImageIcon, 236
InputStream, 200
Integer, 91
JFrame, 219
JMenu, 243
JMenuBar, 243
JMenuItem, 243
JOptionPane, 245
KeyAdapter, 234
KeyEvent, 234
konečná, 166
Long, 91
Math, 96
metoda, 15

MouseListener, 234
MouseEvent, 234
MouseMotionAdapter, 234
NullPointerException, 169
ObjectInputStream, 206
 ObjectOutputStream, 206
OutputStream, 200
parametrizovaná, 68; 191
polymorfismus, 19
pro výjimky
hierarchie, 169
Reader, 202
RuntimeException, 169
Scanner, 68; 211; 217
Short, 91
skládání, 16; 20
String, 50; 55; 60
StringBuffer, 50; 61
Throwable, 168
UIManager, 225
WindowAdapter, 234
WindowEvent, 234
Writer, 202
znázornění, 16
třídění, 12
typ
boolean, 42; 97
byte, 86
datový, 9
double, 94
formální, 191
char, 92
int, 36; 86
long, 86
objektový, viz třída
primitivní
ukládání, 204
real, 94
short, 86
surový, 197
void, 98
výčtový, 106
vymazání, 197
zástupný, 198

U

událost, 231; 232

handler, 231
posluchač, 231
příjemce. viz posluchač
typ, 232
ukončení grafického programu,
219
UML, 16
UNICODE, 93
univerzální jméno znaku, 92
úplné vyhodnocení, 98
uspořádání
 FlowLayout, 229
 GridLayout, 229
null, 230

V

vlastnost, 221
void, 98; 145
vstup
 MojelO, 38
vstup a výstup, 7
vstupní podmínka, 135
vybalení. viz balení
výjimka, 57; 168
 CloneNotSupportedException,
 186

ošetření, 171
šíření, 172
vyvolání, 169
vymazání typů, 197
výraz, 112; 121
výstup
 standardní, 31
vzhled aplikace, 225

W

while, 41

Z

zapouzdření, 14; 15; 51
znak, 92
 univerzální jméno, 92
zápis, 92
zpráva, 144

Ž

žolík, 198

-- V nakladatelství NEOCORTEX dále výšlo --



Shelly Brisbin **WI-FI** 251 stran, cena 199,- Kč

- ❖ seznámení se základy bezdrátové sítové technologie
- ❖ jak vybrat správné wi-fi komponenty podle potřeb a finančních možností
- ❖ použití wi-fi sítě: sdílení souborů a tiskáren, přístup na internet, hraní her

David M. Beazley **PYTHON** 445 stran, cena 99,- Kč

- ❖ charakteristické vlastnosti programovacího jazyku Python
- ❖ detaily o typech, operátorech, třídách a běhu programu
- ❖ přehled o rozhraních operačních systémů, vláken a modulů sítového programování

Jim Mock **FREE BSD** 694 stran, cena 199,- Kč

- ❖ podrobný průvodce operačním systémem FreeBSD pro vývojáře s instalacním CDROM
- ❖ spolehlivá stabilita, snadná instalace a kompletní vývojové prostředí
- ❖ nejlepší internetový operační systém, vysoký výkon a tisíce aplikací

Marty Hall **JAVA SERVLETY A JSP** 608 stran, cena 199,- Kč

- ❖ praktický návod na použití platformy pro webové aplikace
- ❖ stavky plně přenositelných, podrobně zdokumentovaných příkladů
- ❖ návod na použití technologií, jako jsou JDBC, sledování sezení (sessions), apletů....

Pavel Satrapa **IPV6** 240 stran, cena 99,- Kč

- ❖ podrobný popis vlastností a schopností protokolu Ipv6
- ❖ implementace protokolu Ipv6 na platformách MS Windows, Linux, Cisco Systems, BSD
- ❖ formát datagramu a hlaviček, DNS pro Ipv6, bezpečnostní prvky (IPsec),...

Jakub Stainer **GIMP** 224 stran, cena 299,- Kč

- ❖ grafický editor GIMP procházející z prostředí OS Linux
- ❖ praktické ukázky tohoto mladého, ale mocného nástroje

Martin Hynar **JAVA - nástroje** 326 stran, cena 199,- Kč

- ❖ jak na dokumentaci s programem Javadoc
- ❖ ladění aplikací s Jdb, respektive s JSwat
- ❖ sestavovací schéma aplikace Ant
- ❖ JUnit a programování s podporou testů
- ❖ statická kontrola zdrojových kódů programem Pmd
- ❖ testování výkonu serverových aplikací nástrojem JMETER
- ❖ Maven, čili komplexní správa projektů

Clive Barker **MYSTERIUM** 414 stran, cena 99,- Kč.

V knize se otevírá svět lidského podvědomí, fantazie, úzkosti a dávných traumat. Současně je román temným hororem a nakonec ústí do čistokrevné sci-fi. Autor je ve Spojených státech považován za největšího konkurenta Stephena Kinga.

Tom Clancy **NÁMOŘNÍ PĚCHOTA** 384 stran,

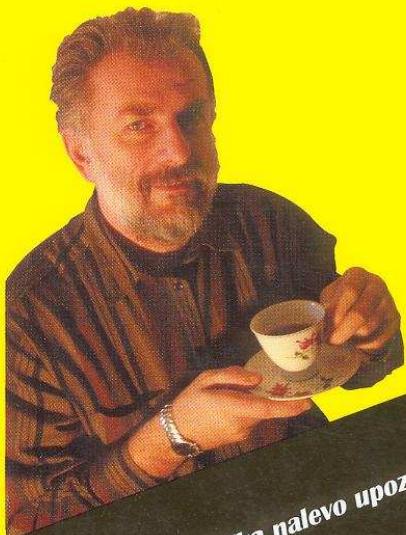
cena 189,- Kč. V této knize je velmi čitivě, jak lze od světové „jedničky“ očekávat, popsán skutečný příběh mužů a žen, kteří slouží u jedné z nejslavnějších větví armádních sil USA. Jsou zde obsaženy skutečné i smyšlené válečné scénáře s detailním popisem jednotek a techniky.

Neocortex, s. r. o.
Na Rovnosti 3, 130 00 Praha
tel.: 284 860 682,
fax: 284 860 117
e-mail: info@neo.cz,
internet: <http://www.neo.cz>

Vydané knihy je možno objednat telefonicky, nebo přes internet na dopravku s 15% slevou. Jinak se ptejte u dobrého knihkupce.

--- Java pro zelenáče srozumitelnou formou popisuje: ---

- jaké jsou principy objektově orientovaného programování
- jak snadno vytvořit dobrý program v Javě
- jaké datové typy a příkazy se v Javě používají
- jak se pracuje s poli a parametrizovanými i neparametrizovanými datovými typy
- k čemu jsou rozhraní, jak využít mechanizmus výjimek a asercí k vytvoření bezpečného programu
- jak programovat grafické uživatelské rozhraní



Jestliže vás programování zajímá,
pak vám tato kniha pomůže dostat se
z úrovně naprostého začátečníka
až na úroveň středně
pokročilého programátora.

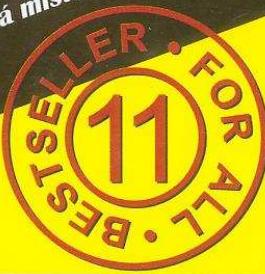
Ručička nalevo upozorňuje na místa, která si zaslouží zvýšenou pozornost.

Pavouk upozorňuje na informace, které lze najít na internetu.
Zpravidla jde o zdrojové soubory příkladů z této knihy.

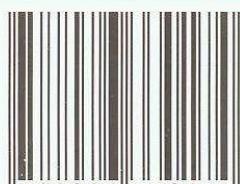
5 Tato ikona upozorňuje na vlastnosti, které jsou specifické pro Javu 5.

Moucha upozorňuje na možné programátorské chyby a na obtížná a problematická místa.

CHIP tip



<http://www.neo.cz>



9 788086 330174

BEŽNÁ CENA 386,00 Sk

Partner Technic, spol.s.r.o. 6776
251024-6/15 090701 97 30174
2205106883 2/2 50902
JAVA pro zelenáče 2.vyd.

UOL/informatika,výpočet.tech/vývojový SW

Ing.
Miroslav Virius, CSc.

přednáší programovací jazyky a programovací techniky na Fakultě jaderné a fyzikálně inženýrské ČVUT. Je autorem více než dvaceti knih a vysokoškolských skript, věnovaných převážně programování. Některé z nich získaly různá ocenění (Tip Chip, Cena nakladatelství Grada aj.).