

UČEBNICE JAZYKA

JAVA™

Pavel Herout



Učebnice jazyka Java

Pavel Herout

Učebnice je určena pro využití v počítačových učebnících na středních školách, v počítačových učebnících pro dospělé a také pro využití v soukromém studiu. Obsahuje všechny potřebné informace o jazyce Java, který je v současnosti nejvíce rozšířeným programovacím jazykem světa. Je možné ho použít k vývoji webových aplikací, mobilních aplikací, aplikací pro operační systémy nebo k vývoji herních aplikací. Jazyk Java je využíván v mnoha různých oblastech, včetně vývoje softwaru, výroby elektronických zařízení, vývoje hraček a dalších.



nakladatelství

České Budějovice, 2001

Obsah

Úvod	15
Typografické a syntaktické konvence	19
2 Základní pojmy	20
2.1 Trocha historie nikoho nezabije	20
2.2 Způsob zpracování programu v Javě	20
2.3 Dva různé typy programů	21
2.4 Vývojové nástroje a jejich verze	21
2.4.1 Vývoj a verze JDK	22
2.5 Používané pojmy	23
2.5.1 Java Core API	23
2.5.2 Java Platforma	24
2.5.3 JIT kompilátor	24
2.6 Co byste měli vědět, než skutečně začnete programovat	25
2.6.1 Jak přeložit a spustit program	25
2.6.2 Běžné problémy	26
3 Základní dovednosti v Javě	31
3.1 Komentáře	31
3.2 Způsob zápisu identifikátorů	32
3.3 Hlavní program	34
3.4 Základní datové typy	35
3.4.1 Celočíselné typy a jejich konstanty	36
3.4.2 Znakový typ a jeho konstanty	37
3.4.3 Řetězcové konstanty (literály)	39
3.4.4 Logický typ a jeho konstanty	40

3.4.5 Reálné datové typy a jejich konstanty	40
3.5 Deklarace proměnných	42
3.5.1 Deklarace proměnných s konstantní hodnotou	43
3.6 Operátor přiřazení	45
3.7 Operátor přetypování	46
3.7.1 Rozšiřující konverze	47
3.7.2 Zužující konverze	49
3.7.3 Rozšiřující konverze se ztrátou přesnosti	49
3.8 Aritmetické výrazy	50
3.8.1 Unární operátory	51
3.8.2 Binární operátory	52
3.8.3 Přiřazovací operátory	53
3.9 Relační operátory	55
3.9.1 Zkrácené vyhodnocování logických součtu a součinu	55
3.9.2 Úplné vyhodnocování logických součtu a součinu	57
3.10 Bitové operace	58
3.10.1 Bitový součin	59
3.10.2 Bitový součet	60
3.10.3 Negace bit po bitu	61
3.10.4 Bitový exkluzivní součet	62
3.10.5 Operace bitového posunu doleva	62
3.10.6 Operace bitového posunu doprava znaménkově	62
3.10.7 Operace bitového posunu doprava neznaménkově	63
3.11 Priority vyhodnocování operátorů	64
4 Terminálový vstup a výstup	67
4.1 Balík <code>java.io</code>	67
4.2 Formátovaný výstup pomocí <code>System.out.print()</code>	67
4.3 Formátovaný vstup	70
4.4 Neformátovaný vstup jednoho znaku	72

5 Řídicí struktury	75
5.1 Příkaz if a příkaz if-else	75
5.1.1 Neúplná podmínka	75
5.1.2 Úplná podmínka	76
5.2 Podmíněný výraz – ternární operátor	77
5.3 Návštětí	79
5.4 Příkazy break a continue	80
5.5 Iterační příkazy – cykly	80
5.5.1 Příkaz while	80
5.5.2 Příkaz do-while	82
5.5.3 Příkaz for	83
5.6 Příkaz switch	89
5.7 Příkaz return	94
6 Metody	97
6.1 Deklarace metody	98
6.2 Metoda bez parametrů	99
6.3 Metoda bez návratového typu – procedura	100
6.3.1 Procedura bez parametrů	100
6.4 Metoda s více parametry různých typů	101
6.5 Rekurzivní metody	102
6.6 Konverze skutečných parametrů a návratové hodnoty metod	103
6.7 Způsoby předávání skutečných parametrů metod	104
6.8 Přetížené metody	104
6.9 Místa deklarací metod	106
6.10 Proměnné z pohledu přístupnosti z metod	107
6.10.1 Nelokální proměnné – “globální” proměnné	107
6.10.2 Proměnné metod – lokální proměnné	108
6.10.3 Zastínění nelokálních proměnných lokálními	109
7 Pole	111
7.1 Pojem referenční datový typ	111
7.2 Deklarace pole	112

7.3	Délka pole	113
7.4	Inicializované pole	114
7.5	Dvouzměrná pole	114
7.5.1	Inicializace dvouzměrného pole	116
7.6	Trojrozměrná pole	116
7.7	Více rozměrů v jednorozměrném poli	116
8	Třídy a objekty – základní dovednosti	119
8.1	Deklarace třídy	120
8.2	Vytvoření objektu	121
8.3	Přímý přístup k datům objektu	122
8.4	Práce s metodami	122
8.5	Konstruktory	124
8.5.1	Implicitní konstruktor	127
8.6	Využití <code>this</code> pro přístup k proměnným	127
8.7	Přetížení metod a konstruktorů	128
8.8	Využití <code>this</code> pro přístup ke konstruktoru	130
8.9	Volání metod jinými metodami téže třídy nebo konstruktorem	131
8.10	Použití proměnné třídy v objektech	131
8.11	Použití statických metod v objektech	134
8.11.1	Použití statické metody ze třídy z Java Core API . .	134
8.11.2	Použití statické metody z téže třídy	136
8.12	Inicializace proměnných třídy	138
8.13	Rušení objektů	140
8.14	Ukončení práce s objekty	141
9	Řetězce a znaky	145
9.1	Vytvoření řetězce	146
9.1.1	Inicializované pole řetězců	147
9.2	Práce s celými řetězci	147
9.2.1	Porovnávání	147
9.2.2	Převody na malá či velká písmena	148
9.2.3	Spojení řetězců	149

11 Dědičnost	179
11.1 Úvodní poznámky	179
11.2 Realizace dědičnosti	180
11.3 Problémy s neimplicitními konstruktory rodičovské třídy .	183
11.4 Nechceme, aby bylo možné metodu překrýt – finální metody	184
11.5 Chceme, aby bylo nutné metodu překrýt – abstraktní me-	
tody a třídy	185
11.6 Nechceme, aby bylo možné třídu zdědit – finální třídy . .	187
11.7 Překrytí proměnné	188
11.8 Základem je Object	189
11.8.1 Metoda <code>clone()</code>	190
11.8.2 Metoda <code>equals()</code>	196
11.8.3 Metoda <code>hashCode()</code>	197
11.8.4 Metoda <code>getClass()</code>	198
12 Balíky	201
12.1 Import balíků	202
12.2 Vytváření balíků	203
12.2.1 Celosvětově platná konvence pro pojmenování . .	205
12.3 Přístupová práva	206
12.3.1 Specifikátor <code>private</code>	206
12.3.2 Specifikátor <code>protected</code>	208
12.3.3 Specifikátor <code>public</code>	210
12.3.4 Specifikátor neuveden	210
12.4 Při dědění nelze zeslabit přístupová práva	210
13 Rozhraní (interface)	212
13.1 Konstrukce rozhraní	213
13.2 Použití jednoho rozhraní	214
13.3 Použití rozhraní jako typu referenční proměnné	215
13.4 Implementace více rozhraní jednou třídou	216
13.5 Instance rozhraní může využívat jen metody rozhraní .	217
13.6 Implementované rozhraní se dědí beze změny	218

13.7 Dědění třídy a současná implementace rozhraní	219
13.8 Dědění rozhraní a konstanty rozhraní	221
13.9 Využití operátoru <code>instanceof</code>	222
14 Polymorfismus	224
14.1 Využití abstraktní třídy	225
14.2 Použití neabstraktních tříd	228
14.3 Použití rozhraní	230
15 Vnořené třídy	232
15.1 Vnitřní třídy	233
15.1.1 Implementace rozhraní pomocí metody využívající vnitřní třídu	234
15.1.2 Implementace rozhraní pomocí metody využívající anonymní vnitřní třídu	235
15.1.3 Proměnná typu rozhraní využívající anonymní vnitřní třídu	237
15.1.4 Vnitřní třída je vytvořena děděním	238
15.1.5 Vnitřní anonymní třída vznikne děděním	239
16 Výjimky	241
16.1 Možné druhy výjimek	242
16.1.1 Třída <code>Error</code>	243
16.1.2 Třída <code>RuntimeException</code>	243
16.1.3 Třída <code>Exception</code>	244
16.2 Způsoby ošetření výjimky	245
16.2.1 Předání výjimky výše – deklarace výjimky	246
16.2.2 Kompletní ošetření výjimky	247
16.2.3 Ošetření výjimky a následné předání výše	249
16.2.4 Naprosto nejhorší reakce na výjimku	250
16.2.5 Rozumná reakce na výjimku	251
16.3 Seskupování výjimek	252
16.3.1 Postupná selekce výjimek	253
16.4 Vyvolání výjimky	253

16.5 Vytvoření a použití vlastní výjimky	255
16.6 Konstrukce try–catch–finally	257
16.6.1 Konstrukce try–finally	259
17 Adresáře a soubory	261
17.1 Zajištění nezávislosti na operačním systému	261
17.2 Vytvoření instance třídy File	262
17.3 Vytvoření souboru nebo adresáře	263
17.4 Práce se souborem nebo adresářem	264
17.5 Výpis adresáře	265
17.5.1 Selektivní výpis adresáře	266
18 Čtení ze vstupů a zápis na výstupy	269
18.1 Proudy znaků a proudy bajtů	270
18.2 Dva různé typy tříd zděděných od základních tříd	271
18.2.1 Třídy pro práci se zařízeními	272
18.2.2 Třídy vlastností (filtrů)	272
18.3 Čtení ze souboru a zápis do souboru	273
18.3.1 Vstupy a výstupy znaků	273
18.3.2 Vstupy a výstupy bajtů	275
18.3.3 Další dovednosti se soubory	276
18.4 Třídy vlastností	277
18.4.1 Vlastnost: bufferování	278
18.4.2 Vlastnost: čtení po řádcích	278
18.4.3 Vlastnost: výběrové čtení po řádcích	279
18.4.4 Vlastnost: vrácení přečteného znaku	279
18.4.5 Vlastnost: formátovaní výstupu	280
18.4.6 Vlastnost: formátování výstupu s řádkovým bufferováním	281
18.4.7 Formátovaný vstup	281
18.4.8 Vlastnost: neformátovaný vstup a výstup základních datových typů	283
18.4.9 Vlastnost: serializace objektů	285
18.4.10 Seskupování vlastností	287

18.5 Vstup a výstup do paměti	288
18.6 Vstup a výstup do roury	289
18.7 Soubory s náhodným přístupem	291
19 Systémové akce	297
19.1 Parametry příkazové řádky	297
19.2 Systémové atributy a zdroje	299
19.2.1 Standardní vstupní a výstupní proudy	300
19.2.2 Systémové vlastnosti	302
19.3 Užitečné metody ze třídy System	305
19.3.1 Informace o čase	305
19.3.2 Spuštění garbage collectoru	306
19.3.3 Zjištění velikosti dostupné paměti	307
19.3.4 Spuštění finalizeru	308
19.3.5 Násilné ukončení programu	308
20 Vlákna	310
20.1 Třída Thread	311
20.2 Spolupráce dvou vláken	313
20.2.1 Vlákna se pravidelně střídají	316
20.2.2 Vlákna se střídají nepravidelně	317
20.3 Stavy vlákna a plánovací algoritmus	318
20.3.1 Stavy vlákna	318
20.3.2 Priorita vlákna	319
20.3.3 Sdílení času	320
20.3.4 Praktické ověření plánování a priorit	320
20.4 Rozhraní Runnable	321
20.5 Čekání na vstup či výstup	323
20.6 Synchronizace vláken	325
20.6.1 Vlákno čeká trpělivě na konec jiného vlákna	325
20.6.2 Vlákno čeká netrpělivě na konec jiného vlákna	326
20.6.3 Vlákno ukončí předčasně jiné vlákno	327
20.6.4 Vlákno je násilně probuzeno	329

20.6.5 Kritické sekce – synchronizované metody	330
20.6.6 Kritické sekce – synchronizované bloky	332
20.6.7 Synchronizace časové posloupnosti vláken	333
20.7 Další informace o vláknech	339
20.7.1 Problematika hladovění a uváznutí	339
20.7.2 Skupiny vláken	340
20.7.3 Vlákna typu démon	340
Literatura	343
Rejstřík	344

Úvod

Java je v dnešní době fenomén, který je skloňován ve všech pádech. Za pět let od svého vzniku prošla neuvěřitelným rozvojem a stejně tak neuvěřitelně začíná ovlivňovat i dění v počítačovém světě. Pravděpodobně ovlivnila i vás, protože čtete tento úvod.

V současné době můžete studovat Javu z minimálně několika desítek knih, z nichž některé – a to i ty nejkvalitnější od nerenomovanějších autorů – jsou k dispozici v plném znění na WWW. Java je v nich popisována ze všech možných úhlů pohledu, takže si každý může vybrat ten, který mu nejvíce vyhovuje. Naprostá většina těchto knih je ovšem v angličtině, ale i u nás si můžete (v březnu 2000) opatřit přinejmenším osm českých nebo do češtiny přeložených knih, které se Javou zabývají. Tato kniha je tedy jen další knihou z řady jiných knih o Javě.

Co můžete od knížky očekávat

Příznivé ohlasy čtenářů na moji předchozí knihu **Učebnice jazyka C** mne přivedly k tomu, že jsem ji napsal v podobném duchu – jako učebnici pro začátečníky a mírně pokročilé.

Výklad se zabývá teorií v naprosto minimální míře a je založen na konkrétních příkladech.¹ Tento postup jsem převzal z již zmíněné **Učebnice jazyka C**, ale hlavně mě v jeho správnosti utvrdily dvě skvělé knihy. **The Java Tutorial** [Cam1], ze které jsem se sám Javu učil, a **Thinking in Java** [Eck1], která se mi bohužel dostala do rukou až v závěru práce na této knížce. Obě zmíněné publikace používají stejný postup, tj. minimum teorie a hodně příkladů, ač jsou jinak naprostě odlišné.

V této učebnici ale nehledejte velkou podobnost s žádnou jinou knihou o Javě. Vznikla tak, že jsem vzpomíнал, co vše bylo pro mě při učení se Javy důležité a to jsem se pak podle svého nejlepšího přesvědčení pokusil logicky srovnat do kapitol.

¹Těch je v knize více než 198, jak si můžete sami lehce ověřit, protože jsou číslovány.

Při psaní jsem se snažil o co největší jednoduchost. Vycházel jsem z přesvědčení, že čtenář musí „na první pohled“ pochopit algoritmus, aby se mohl věnovat tomu podstatnému, tj. probírané jazykové konstrukci.

Kniha se nesnaží o detailní popis Javy, snaží se naučit používat Javu. V každém případě jsem se pokusil vybrat z nejrůznějších podrobností ty, které se opravdu používají nebo na které pravděpodobně narazíte. Snažil jsem se nezabývat tím, co je sice teoreticky zajímavé, ale prakticky zřídka používané.

Uspořádání knihy

Výklad je veden tak, aby jednotlivé kapitoly na sebe navazovaly a v dalších kapitolách byly používány věci již dříve vysvětlené. To se samozřejmě nemůže podařit zcela, ale v takových místech je vždy uveden odkaz na část knihy, ve které je příslušná pasáž vysvětlena.

V knize je množství *Poznámek*, *Upozornění*, *Dobrých rad*² atd., které vznikly na základě mých (většinou negativních) zkušeností s učením se Javy. Věřím, že vám tyto poznatky pomohou vyhnout se popisovaným problémům a ušetřit tak množství času.

Pokud již umíte programovat v jazyce C či C++, pak máte určitý náskok před ostatními, protože Java z této jazyků zjevně vychází. Zejména v začátcích vám bude množství jazykových konstrukcí jasné pochopitelných. Pro vás jsou pak určeny *Poznámky pro programátora* v C či C++, které vás mají varovat před odlišnostmi obou jazyků a nenechat vás ukolet počátečním pocitem, „že je to vlastně C“.

V textu – a zejména v poznámkách pod čarou – naleznete porůznu drobné „žertíky“, přičemž na některé budete upozorněni známým symbolem ; -) a na ty zbývající přijdete pravděpodobně sami. Ovšem pokud naleznete „žertík“ někde ve výpisu programu nebo bránící pochopení, pak to není žertík, ale moje chyba ; -), na kterou mne, prosím upozorněte.

Naprosto většinu příkladů jsem sám vytvářel od samého začátku. Ty zbylé jsem převzal z literatury, která je uvedena na konci knihy. Každý příklad jsem ověřil³ a upravil tak, aby svým stylem zapadal do celkové koncepce knihy. Příklady jsou tak jednoduché a krátké, jak jen

²Dobré rady na konci kapitoly platí pro celou kapitolu.

³Všechny příklady, byly ověřovány pomocí JDK od firmy JavaSoft, neboť tento přístup zaručuje stoprocentní kompatibilitu s jazykem Java.

bylo možné, což občas vede k tomu, že jsou zcela primitivní. Tuto primitivnost neberte, prosím, jako podceňování vašich schopností ani jako úroveň mého myšlení ; -)

Možná, že se vám při čtení budou zdát nejen příklady, ale i některé pasáže výkladu nepochopitelně triviální. Skutečně zde některé takové jsou a to proto, že z výuky vím, jaké problémy činí nezanedbatelné části studentů. Nehledejte tedy v takovýchto pasážích žádný skrytý smysl – skutečně jsou tak triviální, jak se vám zdají, a klidně je při čtení přeskočte.

Jazykový koutek

Pokud je to možné, pak používám českou terminologii. Při prvním použití termínu je téměř vždy uváděn v závorkách příslušný anglický ekvivalent, se kterým se setkáte, budete-li číst literaturu nebo náповědu v angličtině. V žádném případě nepoužívám české termíny za „každou cenu“, protože prvotní účel této knihy má být podle mého názoru **srozumitelnost** jazyka Java, nikoliv čistota použitého jazyka českého. Proto v knize naleznete anglicismy typu „bufferování“, což je dle mého mínění všeobecně známý pojem pro „využití vyrovnávacích pamětí“, případně přímo anglické výrazy jako *garbage collector*, který se sice dá přeložit, ale nikdo pak nebude vědět, o čem je řeč. Dopředu se omlouvám všem, kteří by uvedený přístup cítili jako „prznění“ našeho mateřského jazyka, a chci je ujistit, že to tak nebylo míněno.

Co v knížce nenaleznete

Pravděpodobně vás překvapí, že se zde nedočtete nic o programování apletů ani o tvorbě grafických uživatelských rozhraní. To není opomenutí, ale záměr. Na téměř 350 stranách knihy byste se měli naučit používat jazyk a základní knihovní třídy, tedy dovednosti, které použijete v každém ze svých programů. Tento jazyk je v současné době natolik stabilní, že se pravděpodobně nebude v dohledné době měnit. Na rozdíl od např. prostředků pro „tvorbu okének“, kde ještě v roce 1998 suverénně kralovalo AWT, které je od začátku roku 1999 postupně vytlačováno JFC Swing.

Protože však grafická uživatelská rozhraní jsou nesmírně důležitá, budou popisována v již připravovaném pokračování této knihy.

Při psaní knihy jsem neustále přidával další a další pasáže, které „jsou nezbytně nutné“ pro pochopení, takže z původně plánovaných 250 stran jsem se „přehoupl“ na 300 a pak až na 350, kdy již nakladatel řekl dost. Proto některé věci, které by do této knížky ještě logicky patřily, naleznete až v dalším připravovaném dílu, který již zde označuji jako [UJJ2].

Dále v knize najdete podrobný a souvislý výklad metodiky objektově orientovaného programování – v případě zájmu jej můžete nalézt v [Rac].

Poděkování

Mojí milou povinností je poděkovat doc. Ing. Stanislavu Rackovi, CSc. a Ing. Miroslavu Viriusovi, CSc. za pečlivou korekturu, která významně přispěla ke zkvalitnění obsahu knihy.

Kde hledat nejnovější informace

Učebnice jazyka Java má vytvořeny vlastní WWW stránky

www.kiv.zcu.cz/~herout/java/ujj1
na kterých se snažím udržovat aktuální informace. Zde je možné dozvědět se o všech novinkách, můžete mi odtud poslat vzkaz nebo dotaz, můžete si přečíst odpovědi na nejčastěji pokládané dotazy atd. Také se zde lze dozvědět o již nalezených chybách. V této souvislosti mám na Vás prosbu – pokud se stane, že v knize naleznete jakoukoliv chybu, buděte prosím tak laskaví a dejte mi o této chybě vědět pomocí připraveného formuláře. Pomůžete tím nejenom mě, ale i ostatním čtenářům. Děkuji.

Pavel Herout

Poznámky k druhému dotisku

Narozdíl od prvního dotisku bylo nutné, kromě oprav překlepů a drobných chyb, provést v knize menší úpravy v několika programech. Obsah diskety je ale **stejný jako u prvního vydání** – to aby bylo zamezeno zmatkům s různými verzemi disket.

Všechn pět opravených programů si můžete stáhnout z již zmíněného WWW.

Typografické a syntaktické konvence

V knize jsou různými typy písma odlišeny ty části textu, které si odlišení zaslouží. Jedná se o:

if	klíčové (rezervované) slovo jazyka Java v textu
if (a == b)	úsek programu
if (podmínka)	<i>podmínka</i> je příklad obecného syntaktického objektu ve vysvětlení konstrukce jazyka
Prvni.java	jméno souboru
.class	skupina souborů s danou příponou
znak x byl	znak x zmiňovaný v textu
dědičnost	výraz, který je použit poprvé a bude dále vysvětlen
garbage collector	anglický výraz
nepřeloží se	zvýraznění významu slova
Poznámka:	začátek poznámky, varování, ...
□	konec poznámky, upozornění, varování, ...
[9.6/153]	tato problematika je rovněž zmiňována v části 9.6 na straně 153
; -)	v tomto místě jsem se pousmál ⁵

Výpis některých programů nezačíná od prvního sloupce. Má to signalizovat, že se jedná pouze o **výsek** programu, který je důležitý pro pochopení právě probírané jazykové konstrukce. V případě zájmu si pak můžete celý program prohlédnout v souboru na disketě.

⁵Vy ale nemusíte ; -)

2 Základní pojmy

2.1 Trocha historie nikoho nezabije

Od roku 1991 vyvíjela firma Sun Microsystems programovací jazyk na principech C a C++ pro „vestavěné systémy“, což je odborný termín pro běžná elektronická zařízení¹ ovládaná zabudovaným mikroprocesorem. Jazyk měl původně název *Oak* (dub), podle dubu, který stál před oknem pana Goslinga, vedoucího týmu.

Posléze se zjistilo, že již programovací jazyk s tímto názvem existuje, takže vývojová skupina hledala název nový a po návštěvě firemního bufetu padl návrh Java, což znamená v americkém slangu „kafe“. Projektu se příliš nedařilo, ale v roce 1993 si firma Sun uvědomila vznikající důležitost WWW a možnosti využít Javu pro programování aplikací pro WWW. V květnu roku 1995 byla Java firmou Sun oficiálně představena na konferenci. Již během této přednášky začalo být mnohým účastníkům zřejmé, že se jedná o programovací jazyk, který bude hrát významnou úlohu při programování webových aplikací.

Mnozí si však jasně uvědomili, jaké možnosti skýtá Java i jako „běžný“ programovací jazyk. Java ovšem vzbudila okamžitě zájem průmyslu zejména díky právě počínající vlně zájmu o obchodní využití Internetu a WWW.

2.2 Způsob zpracování programu v Javě

Standardní postup je ten, že program v Javě prochází pěti fázemi – editováním, překladem (kompilací), zavedením (*load*), ověřováním (verifikací) a prováděním. Čtyři z těchto fází jsou běžné i v ostatních programovacích jazycích. Fáze ověřování je něco nového, ale pro Javu (a zejména programování na WWW) velmi důležitého – umožňuje totiž dosáhnout

¹Typu pračky, mikrovlnné trouby atd.

velmi vysoké bezpečnosti spuštěného programu, čímž je míněna hlavně ochrana toho, kdo program spouští.

Další zvláštností Javy je, že překlad neprobíhá do jazyka relativních adres (srozumitelně – do .OBJ), který je v podstatě totéž, co strojový jazyk počítače, ale do pseudojazyka nazývaného *byte-code* (česky **bajtkód**). Tento jazyk je nezávislý na cílovém počítači, což prakticky znamená, že programátora nemusí vůbec zajímat, na jakém počítači jeho program poběží. Přeložený program – bajtkód – je uložen v souboru s vyhrazenou příponou .class. Tento soubor je pak z disku zaváděn do paměti počítače a současně probíhá ověření bajtkódu, což je možné provést jednotně díky nezávislosti bajtkódu na platformě. Po ověření je program spouštěn pomocí interpreteru – Java je tudiž interpretovaný jazyk, jako byl (a je) BASIC.

2.3 Dva různé typy programů

Programy v Javě se podle cílového použití dělí na dvě velké skupiny – první jsou *aplikace*, které si můžeme představit jako běžné programy známé z jiných programovacích jazyků – s těmi se budeme v této knize setkávat. Druhou skupinou jsou *aplety* (*applets*), které se používají na WWW stránkách. Pro obě skupiny platí stejná pravidla, co se týče fází editování a překladu. Fáze zavedení se liší, protože aplety na WWW stránce jsou zaváděny do paměti počítače pomocí standardních prohlížečů typu Netscape nebo MSIE. Fáze ověření se opět trochu liší, protože pro aplety platí přísnější pravidla než pro aplikace. Provádění programu je u obou skupin stejné – program je interpretován.

2.4 Vývojové nástroje a jejich verze

Všechny potřebné programy pro plnohodnotnou práci s Javou můžete získat zdarma na www.javasoft.com

Hledejte v *Products & APIs* a hledejte zkratku JDK, která tradičně označuje *Java Development Kit*. Prakticky ale nejdříve hledejte značku J2SDK (*Java™ 2 SDK, Standard Edition, v.1.2*), jak byla z licenčních důvodů JDK přejmenována. Zkratka SDK znamená *Standard Development Kit*.

Poznámka:

Toto přejmenování je na:

<http://www.javasoft.com/products/jdk/1.2/java2.html>

detailně vysvětleno. Abychom se neztratili v různých označeních, bude se v knize se dále používat označení JDK 1.2, protože toto označení je v programátorské veřejnosti všeobecně známé a používané. □

2.4.1 Vývoj a verze JDK

Jazyk Java se bouřlivě vyvíjí a s ním se mění i JDK. Tento vývoj je zachycen i v číslování jednotlivých verzí JDK. Každé číslo se skládá ze tří čísel oddělených tečkami.

Poznámka:

Z licenčních důvodů se od JDK 1.2 jazyk Java správně označuje jako JavaTM 2. Toto označení se **nemění** s verzemi JDK. Hovoříme-li tedy o Javě, je tím míněna – viz verze dále – stále JavaTM 2. □

První číslo (zatím vždy rovno 1 – nehledě na označení JavaTM 2 ; –) znamená hlavní verzi (*major version*). Toto číslo se bude měnit pouze při velmi závažných změnách Javy.

Druhé číslo (zatím 0, 1, 2 a připravuje se 3) znamená méně významnou verzi (*minor version*) a mění se při změnách jazyka a/nebo API. Změna představuje poměrně významnou událost ve světě Javy.

Třetí číslo představuje číslo verze s opravami chyb (*bugfix version*). Toto číslo se mění relativně často a v dokumentaci se běžně neuvádí.

Současné verze (stav v lednu 2000):

JDK 1.0.2 – Původní verze Javy, dnes už téměř nepoužívaná.

JDK 1.1.8 – Dnes ještě běžně používaná verze, od které se ale pomalu ustupuje. Oproti JDK 1.0 došlo ke změnám jazyka a k podstatným změnám (zejména rozšíření) API.

JDK 1.2.2 – Současná platná verze, označovaná jako JavaTM 2 SDK, Standard Edition, v.1.2. Oproti JDK 1.1. se prakticky vůbec nezměnil jazyk. Ke změnám došlo v API. Podstatnou změnou bylo zahrnutí některých nových knihoven (např. JFC Swing) jako součásti JDK.

JDK 1.3.0 – Připravovaná verze, označovaná jako JavaTM 2 SDK, Standard Edition, v.1.3. Nejvýznamnější změnou proti JDK 1.2. je zrychlení části API. Změny v jazyce nenastaly žádné, změny v API jsou pouze rozšíření a vylepšení.

Poznámka:

V této knize bude popisován zejména jazyk Java, který je stabilní od verze JDK 1.1. Použitá verze JDK pro ověřování všech příkladů je JDK 1.2.2. □

2.5 Používané pojmy

V Javě i v této knize se setkáte s množstvím zkratek a pojmu. Pro začátek je třeba, abyste znali význam alespoň těch dále uvedených.

2.5.1 Java Core API

S touto zkratkou se budete setkávat často a znamená *Application Programming Interface* (aplikační programové rozhraní).² Pod touto zkratkou se skrývá značné množství knihovních tříd (odhadem hodně přes tisíc), které jsou považovány za standardní, čili **musí** se vyskytovat v každém prostředí, kde se Java používá.

To znamená, že když náš program využívá metody z API, není jejich kód součástí programu, protože je součástí API. Prakticky to znamená, že naše programy obsahují pouze kód, který jsme napsali my, a soubory, ve kterých jsou naše přeložené programy uloženy, mají proto poměrně malou velikost.

Všechny třídy, jejich metody a proměnné jsou velmi dobře zdokumentovány a dokumentace je přístupná pomocí WWW prohlížečů. Je nutné pouze najít na disku počáteční soubor index.html, který se často nachází v adresáři:

C:\Program Files\jdk1.2.2\docs\api\index.html

Poznámka:

Je téměř nezbytné naučit se tuto dokumentaci používat, protože díky tomu, že je generována automaticky z komentářů ve zdrojových kódech knihovních tříd, je vždy aktuální. Java se neustále vyvíjí a je velmi nepříjemné, když si někde v tištěné dokumentaci přečtete použití nějaké knihovní metody, ale program vám přesto nefunguje. V tomto případě je jednou z prvních věcí, které je vhodné učinit, nalezení popisu této metody v dokumentaci o API a její kontrola. □

²Core znamená jádro.

2.5.2 Java Platforma

Jedním z nejvíce oceňovaných přínosů Javy je plná přenositelnost programů na libovolnou platformu (rozuměj počítač s operačním systémem) bez nutnosti jejich překladu na této platformě. Jak již bylo zmíněno dříve, dosahuje se této přenositelnosti pomocí bajtkódu, jehož interpretace je pak úkolem speciálních programů, nazývaných souhrnně **Java platforma**, které jsou ovšem předem pro tuto platformu připraveny.

Poznámka:

Plná přenositelnost tedy neznamená, že program v Javě půjde spustit i na historickém ZX Spectru, ale vztahuje se jen na počítače, pro které je někým připravena Java platforma. □

Java platforma se skládá ze dvou hlavních částí. První část tvoří tzv. **virtuální stroj** (Java Virtual Machine – JVM), který se skládá z části zajišťující vazbu na hardware a z části interpretující bajtkód. Tento interpreter může být nahrazen JIT komplilátorem. Druhou část Java platformy tvoří již zmíněné **Java Core API**.

2.5.3 JIT komplátor

Problémem interpretovaných jazyků je jejich pomalost ve srovnání s kompliovanými jazyky. Java tento problém částečně řeší použitím tzv. **JIT komplátorů** (Just In Time), které v době zavádění programu z disku do paměti počítače (po ověření správnosti bajtkódu) jej přeloží *on-line* do strojového jazyka konkrétního počítače, címž z něj prakticky vyrobí v paměti .EXE program. Ten pak běží stejnou rychlostí, jako kterýkoliv jiný kompliovaný program napsaný třeba v C.

Poznámka:

Výraz „stejnou rychlosti“ je nutné chápat ve smyslu „stejnou rychlosti, jako když program není interpretován“. Znamená to, že málokdy (i při použití JIT) dosáhnete stejné rychlosti programu v Javě jako v C. Je to z toho důvodu, že Java má odlišnou vnitřní filosofii, kdy se např. vše vytváří dynamicky a uvolňování probíhá automaticky. To je z hlediska programátora značně pohodlné, ale z hlediska počítače značně časově náročné. Z tohoto pohledu bude dobré napsaný program v C vždy rychlejší než tentýž program v Javě.

Rychlosť programu ovšem dnes není to, co by nás nejvíce „bolelo“. S neuvěřitelným vývojem v oblasti procesorů se počítače zrychlují a zlevňují, takže pomalost Javy je vyrovnaná rychlosťí procesoru. To,

v čem Java vyniká a proč se používá, je snadnost použití, rychlosť vývoje programu, přenositelnost a robustnost programu. A to jsou oblasti, ve kterých (alespoň podle mého názoru :-)) Java nad C jasně vede. □

2.6 Co byste měli vědět, než skutečně začnete programovat

2.6.1 Jak přeložit a spustit program

Při učení se jakéhokoliv programovacího jazyka je vhodné provádět pokusy. Na těch se totiž naučíme zdaleka nejvíce.

Zkuste si na svém počítači v textovém editoru přesně opsat následující nejjednodušší program v Javě. Je velmi důležité, aby se soubor jmenoval `Prvni.java`, nikoliv `prvni.java` nebo `PRVNI.JAVA` či dokonce `pokus.java`.

Příklad 1:

```
public class Prvni {  
    public static void main(String[] args) {  
        System.out.println("Ahoj");  
    }  
}
```

Nad významem jednotlivých příkazů nebudeme zatím hloubat, ovšem pokud znáte programovací jazyk C či C++, pak je vám zřejmě většina „slov“ určitě alespoň povědomých.

Máme-li soubor, pokusíme se jej přeložit. Jste-li v MS Windows, vyvolajte *Command Prompt* (příkazovou řádku DOSu) pomocí Start/Programs/Command Prompt. Pak se přepněte do adresáře, ve kterém je uložen soubor `Prvni.java`, a spusťte překladač `javac.exe`. Příkaz je:

```
javac Prvni.java
```

Po úspěšném překladu vznikne soubor `Prvni.class`, který spusťte pomocí interpreteru `java.exe`. Příkaz je:

```
java Prvni
```

Úspěšný průběh překladu i spuštění vidíte na následujícím snímku obrazovky.

```
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.
```

```
D:\JAVA\ujj\prvni>dir
Volume in drive D has no label.
Volume Serial Number is 5C3C-F999

Directory of D:\JAVA\ujj\prvni

02.04.99 13:58      <DIR>
02.04.99 13:58      <DIR>
02.04.99 12:17      107 Prvni.java
                           107 bytes
                           3 File(s)   1 014 626 304 bytes free
```

```
D:\JAVA\ujj\prvni>javac Prvni.java
```

```
D:\JAVA\ujj\prvni>dir
Volume in drive D has no label.
Volume Serial Number is 5C3C-F999
```

```
Directory of D:\JAVA\ujj\prvni

02.04.99 13:59      <DIR>
02.04.99 13:59      <DIR>
02.04.99 13:59      408 Prvni.class
02.04.99 12:17      107 Prvni.java
                           515 bytes
                           4 File(s)   1 014 626 304 bytes free
```

```
D:\JAVA\ujj\prvni>java Prvni
Ahoj
```

```
D:\JAVA\ujj\prvni>
```

2.6.2 Běžné problémy

2.6.2.1 Překladač javac.exe nelze spustit

Je třeba přidat do cesty (PATH) adresář, ve kterém se javac.exe nachází, např.: \jdk1.2\bin . Například ve Windows NT pomocí:

Start/ControlPanel/System/Environment

nebo použít úplné cesty, např.:

```
c:\jdk1.2\bin\javac Prvni.java
```

2.6.2.2 Pojmenovali jsme soubor jinak

Pro výskyt této chyby stačí i rozdíl v malých či velkých písmenech. Soubor se musí jmenovat naprosto stejně jako identifikátor za public class ve zdrojovém programu.

2.6.2.3 Při překladu jsme zadali neúplné jméno souboru

Napsali jsme jen:

javac Prvni

místo správného:

javac Prvni.java

Ukázky chybových výpisů si můžete prohlédnout na snímku obrazovky.

```
D:\JAVA\ujj\prvni>dir
Volume in drive D has no label.
Volume Serial Number is 5C3C-F999

Directory of D:\JAVA\ujj\prvni

02.04.99  14:04      <DIR>
02.04.99  14:04      <DIR>
02.04.99  12:17            107  prvni.java
                           3 File(s)        107 bytes
                           1 013 985 280 bytes free
```

```
D:\JAVA\ujj\prvni>javac prvni.java
prvni.java:1: Public class Prvni must be defined in
a file called "Prvni.java".
public class ^Prvni {
1 error
```

```
D:\JAVA\ujj\prvni>ren prvni.java Prvni.java
```

```
D:\JAVA\ujj\prvni>javac Prvni
Prvni is an invalid option or argument.
Usage: javac <options> <source files>
```

where <options> includes:

-g	Generate all debugging info
-g:none	Generate no debugging info
-g:{lines,vars,source}	Generate only some debugging
-O	Optimize; may hinder debugging
-nowarn	Generate no warnings
-verbose	Output messages about what the compiler does
-deprecation	Output source locations where deprecated code is used
-classpath <path>	Specify where to find user class files
-sourcepath <path>	Specify where to find input source files
-bootclasspath <path>	Override location of bootstrap class files
-extdirs <dirs>	Override location of installed extension packages
-d <directory>	Specify where to place generated class files
-encoding <encoding>	Specify character encoding used for source files
-target <release>	Generate class files for specified release

```
D:\JAVA\ujj\prvni>
```

2.6.2.4 Při spuštění jsme zadali celé jméno souboru

Napsali jsme:

```
java Prvni.class
```

místo správného:

```
java Prvni
```

Ukázku chybového výpisu si můžete prohlédnout na obrázku.

```
D:\JAVA\ujj\prvni>dir
Volume in drive D has no label.
Volume Serial Number is 5C3C-F999

Directory of D:\JAVA\ujj\prvni

02.04.99  14:11      <DIR>
02.04.99  14:11      <DIR>
02.04.99  14:11          408  Prvni.class
02.04.99  12:17          107  Prvni.java
                           515 bytes
                           1 012 969 472 bytes free

D:\JAVA\ujj\prvni>java Prvni.class
Exception in thread "main"
java.lang.NoClassDefFoundError: Prvni/class

D:\JAVA\ujj\prvni>
```

Pokud jste příznivci UNIXu, můžete si na další straně prohlédnout, že i pod tímto operačním systémem fungují programy javac a java stejně, včetně podobných či identických chybových výpisů.

```

eryx1.zcu.cz> ls -l
total 1
-rw-r--r-- 1 herout users          102 Apr  2 14:39 Prvni.java
eryx1.zcu.cz> javac Prvni.java
eryx1.zcu.cz> ls -l
total 2
-rw----- 1 herout users          454 Apr  2 14:48 Prvni.class
-rw-r--r-- 1 herout users          102 Apr  2 14:39 Prvni.java
eryx1.zcu.cz> java Prvni
Ahoj
eryx1.zcu.cz> java Prvni.class
Can't find class Prvni.class
eryx1.zcu.cz> javac Prvni
javac: invalid argument: Prvni
use: javac [-g][-O][-debug][-depend][-nowarn][-verbose][-classpath pre
[-deprecation][-d dir][-J<runtime flag>] file.java...
eryx1.zcu.cz> mv Prvni.java prvni.java
eryx1.zcu.cz> ls -l
total 2
-rw----- 1 herout users          454 Apr  2 14:48 Prvni.class
-rw-r--r-- 1 herout users          102 Apr  2 14:39 prvni.java
eryx1.zcu.cz> javac prvni.java
prvni.java:1: Public class Prvni must be defined in
a file called "Prvni.java".
public class Prvni {
                           ^
1 error
eryx1.zcu.cz> ■

```

Poznámka:

Dokud se vám nepodaří program `Prvni.java` přeložit a spustit, nepo-
kračujte v dalším čtení. □

Poznámka pro programátora v C či C++:

Uvedený příklad je skutečně minimální. Pokud se pokusíte neuvést parametr funkce `main()`, dojde k chybovému hlášení bud' při překladu – to když ve zdrojovém kódu bylo `main(void)`, nebo při spuštění – to v případě, že bylo napsáno jen `main()`. □

Dobré rady:

- Pokud jste úplným začátečníkem, nepoužívejte žádné vývojové prostředí. Jejich složitost a nutnost mnoha administrativních úkonů, např. vytvoření projektu (jistěže pro pokročilého samé užitečné věci) vás bude stát spoustu času.

Kromě toho se ztratíte v záplavě předgenerovaného kódu

- Vývojová prostředí svádí k tomu, že lze snadno vytvořit „zadarmo“ okénka. To samozřejmě časem velmi oceníme, ale v začátcích je nutné rozumět jazyku.
- Používání JDK zbaví program tajemnosti – budete vždy vědět, co se děje, protože ve zdrojovém souboru bude pouze váš kód.
- Udržujte ve svých pokusech pořádek – každý program umístěte do nového adresáře. To vám může připadat jako hodně zbytečný luxus, ale důvod, proč mít oddělené adresáře, se jmenuje „přístup do implicitního balíku“ a dočtete se o něm v [12.3/206].

3 Základní dovednosti v Javě

V této kapitole se naučíme používat naprosto nejzákladnější věci, které budeme potřebovat ve všech dalších programech.

3.1 Komentáře

Jako každý programovací jazyk má i Java možnost **komentářů**.

Poznámka:

Všechno se doporučuje komentáře používat. Je velmi vhodné zvyknout si na komentování úseků kódu ihned při jejich vytváření, ne „až na to někdy budu mít víc času.“ Komentář by se měl vždycky objevit v místě, kde je použit neobvyklý programátorský obrat, a pak na místě, kde jste něco (zpočátku neúspěšně) zkoušeli. Pokud to neuděláte, dá se téměř s jistotou očekávat, že až se k programu za nějaký čas vrátíte, strávíte přemýšlením nad těmito neokomentovanými částmi spoustu času. □

Java má možnost tří typů komentářů:

1. **jednořádkový komentář** – začíná `//` a vše, co je za nimi až do konce řádky je komentář:

```
utrata = pocetPiv * 15; // typický jednořádkový komentář
```

2. **komentářový blok** – začíná znaky `/*` a až do výskytu znaků `*/` je vše komentář; je asi zřejmé, že takto lze komentovat v rámci jedné řádky i několika řádek:

```
/* metoda vypočte obsah kruhu
parametrem je poloměr kružnice
-- typický komentář přes více řádek */
double obsahKruhu(double r) { ... }
```

Lze komentovat i uprostřed řádky, ale tento způsob není příliš přehledný:

```
utrata = pocetPiv * /* 15 */ 20; /* 15 je pro desítku */
```

3. dokumentační komentář – začíná znaky `/**` a končí znaky `*/`. Používá se pro automatické generování dokumentace¹ programem `javadoc.exe`.

V tomto druhu komentáře se používají upřesňující slova (značky), která začínají znakem `@`. Protože program `javadoc.exe` generuje dokumentaci ve formátu HTML, je možné používat i většinu značek z tohoto formátu.

```
/*
 * @author Pavel <b>Herout</b>
 */
public class Prvni {
/*
 * @param args - parametry <b>vstupni</b> radky
 */
    public static void main(String[] args) { ... }
```

Poznámky:

- Doplníte-li si uvedeným způsobem již známý program `Prvni.java` a pak zadáte příkaz:
`javadoc -author Prvni.java`
 vygeneruje se 9 souborů `.HTML` (a jeden `.css`), které si můžete prohlédnout běžnými prohlížeči typu MSIE či Netscape.²
- V dokumentačních komentářích se velmi často uvádí jako první znak na řádce znak `*`, ale není tam nezbytně nutný.

Pozor:

Není možné použít **vnořených komentářů** (*nested comments*) typu:

```
/* začátek komentáře /* jeho vnoření */ konec komentáře */
```

Překladač hlásí chybu `Illegal start of expression.` □

3.2 Způsob zápisu identifikátorů

Možná vám bude připadat zvláštní, že je způsobu zápisu identifikátorů věnována celá podkapitola, ale postupem času zjistíte, že to není náhoda,

¹Tak např. vzniká celá dokumentace k API.

²Bez přepínače `-author` se vygeneruje totéž, ale jméno autora pak budete v `.HTML` souborech hledat marně. ; -)

ale nutnost. Je „životně“ důležité dodržovat následující konvence a tím si na ně zvyknout. Tyto konvence jsou totiž důsledně dodržovány v celém Java Core API a jejich znalost vám umožní (mimo jiné) vyznat se v něm.

Java rozlišuje malá a velká písmena a poměrně často se setkáte s tím, že dva identifikátory jsou rozlišeny pouze velikostí písmen.

Namátkový konkrétní příklad z Java Core API: `accessibleContext` je proměnná a `AccessibleContext` je třída. Nebo jiný příklad, jehož smysl je vysvětlen v [3.4/35] – `double` je základní datový typ³ a `Double` je třída.

Striktně dodržované konvence pro zápisu identifikátorů představující:

- **Třídy a rozhraní** – identifikátor začíná vždy velkým písmenem a ostatní písmena jsou malá. V případě, že je identifikátor tvořen více slovy, začíná každé slovo velkým písmenem. Například pro třídy `String` a `StringBuffer` a pro rozhraní `Cloneable`.
- **Metody a proměnné** – identifikátor začíná (a pokračuje) malým písmenem. Stejně jako u tříd se při použití více slov označuje začátek dalšího slova velkým písmenem. Například pro proměnné `pocet`, `pocetPrvku` a pro metody `start()`, `getSize()`.
- **Balíky** – identifikátor se skládá pouze z malých písmen. Ve složených jménech je oddělovačem tečka. Například `java.lang`
- **Konstanty** – používají se pouze velká písmena. Ve víceslových identifikátorech je oddělovačem podtržítko, např. `PI`, `MAX_VALUE`.

Poznámka:

V dokumentaci se proměnné od metod odliší tím, že metody mají za jménem uvedeny prázdné kulaté závorky, a to i v případě, že metoda má ve skutečnosti (tj. v programu) formální parametry. □

Délka všech identifikátorů není omezena. Každý identifikátor musí začínat písmenem nebo podtržítkem. Jako další znaky se mohou vyskytnout i číslice.

³A současně klíčové slovo.

3.3 Hlavní program

Hlavní program, přesněji řečeno metoda, která je vyvolána po spuštění programu jako první, se musí jmenovat `main` a musí být v programu vždy uvedena. Tato metoda musí být „zapouzdřena“ (tj. musí být uvnitř) v nějaké třídě. Na jméně třídy nám zatím nezáleží, ale jak již bylo uvedeno v [2.6.1/25], musí se tato třída bezpodmínečně jmenovat jako soubor, ve kterém je uložena, včetně dodržení stejných malých a velkých písmen v názvu.

Pro metodu `main()` platí další pravidla:

- Měla by být uvedena ve třídě, která je označena jako `public`.
- Musí mít přesně tuto podobu:

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

Poznámka:

Je možný i tvar:

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

ale doporučuje se používat předchozí způsob, protože konvence Javy při deklaraci polí je trochu jiná než v jazyce C. □

- Nesplňuje-li `main()` zmíněné požadavky, program nebude možné přeložit.

Poznámka:

Sestává-li program z více tříd (modulů), smí být `main()` kupodivu ve všech třídách, a to i ve třídách `public`. Z prvního pohledu je to nesmysl, ale při podrobnější úvaze zjistíme, že lze tuto vlastnost s výhodou využít. Jedná se o použití tzv. *self-tested* modulů (viz též [UJC2]), tedy modulů, které je možné samostatně otestovat.

Každý z těchto modulů pak bude obsahovat metodu `main()`, která volá ostatní metody této třídy a testuje je. Po otestování se zavolá interpret `java.exe` se jménem té třídy, ve které je „skutečná“ metoda `main()`. Jinak řečeno, ve výsledném programu, kde je více `main()`, poběží ta metoda `main()`, která leží ve třídě, jejíž jméno je parametrem použitým při spuštění programu `java.exe`. □

Tím, že je metoda `main()` označena jako `static`, je možné ji volat i bez existující instance její „mateřské“ třídy. Podrobně viz [6/97].

Tělo metody `main()` je uzavřeno mezi závorky `()`. Tyto závorky lze použít i pro složený příkaz.

Příklad 2:

```
public class Druhy {
    public static void main(String[] args) {
        int i = 5;
        System.out.println("i = " + i);

        boolean b = false;
        System.out.println("b = " + b);
    }
}
```

Poznámka:

Přibližný význam všech příkazů by vám měl být zřejmý. V každém případě však budou podrobně vysvětleny v dalších částech. □

3.4 Základní datové typy

Základní datové typy (též **primitivní datové typy** – *primitive data types*) jsou v Java několika druhů – celočíselné, znakové, logické, reálné a prázdný datový typ `void`, který se používá jen u metod (tak se nazývají v objektově orientovaných jazycích funkce).

Java jednoznačně definuje u každého typu jeho velikost, tj. počet bajtů, které zabírá v paměti. Tím je dosaženo mnohem většího stupně přenositelnosti než v C.

Pozor:

Protože základní datové typy nejsou z pohledu Javy objekty, dává Java k dispozici ke každému základnímu datovému typu i jeho *wrapper class* (**obalující třídu**), pomocí níž se základní datový typ může stát objektem. O těchto třídách bude zmínka v [9.6/153].

Na tomto místě je třeba varovat před tím, abyste náhodou nepoužili jméno třídy místo jména základního datového typu. V naprosté většině případů se totiž liší jen prvním velkým písmenem.

Jedná se o:⁴

boolean – Boolean, **char** – Char, **byte** – Byte, **short** – Short,
int – Integer, **long** – Long, **float** – Float, **double** – Double.

□

3.4.1 Celočíselné typy a jejich konstanty

Jsou pouze znaménkové, ve dvojkovém doplňku, jak je v počítačích běžným zvykem. Není možné použít neznaménková celá čísla a neexistuje modifikátor `unsigned`.

Celočíselné typy se od sebe liší pouze svojí velikostí a tím i rozsahem zobrazitelných čísel.

název	bitů	rozsah	řád
byte	8	-128 až +127	10^2
short	16	-32 768 až +32 767	10^4
int	32	-2 147 483 648 až +2 147 483 647	10^9
long	64	-9 223 372 036 854 775 808 až +9 223 372 036 854 775 807	10^{18}

Poznámka:

To, že neexistuje neznaménkové celé číslo, je značnou nevýhodou jen u typu **byte**, kdy hodnotu velikosti jeden bajt často (např. v grafických aplikacích) potřebujeme mít v rozsahu 0 až 255. V tomto případě pak občas musíme použít převod na typ **int** tím, že přičteme konstantu 256, např. tímto způsobem:

```
int i; byte b;
b = -1; // -1 = 255 neznaménkově
i = (b < 0) ? b + 256 : b;
i = 128;
b = (byte) ((i > 127) ? i - 256 : i);
```

□

Poznámka pro programátora v C či C++:

Stejně jako nelze použít konstrukci `unsigned int`, nelze použít ani `short int` nebo `long int`.

□

Celočíselné konstanty mohou být zapsány ve třech číselných soustavách:

- desítkové (dekadické) – posloupnost číslic 0 až 9, z nichž první nesmí být 0
- osmičkové (oktalové) – číslice 0 (nula) následovaná posloupností osmičkových číslic 0 až 7

⁴První je uváděn název základního datového typu a druhý je název třídy.

- šestnáctkové (hexadecimální) – číslice 0 (nula) následovaná znakem x (nebo X) a posloupností šestnáctkových číslic 0 až 9, a až f nebo A až F

Příklad 3:

- desítkové: 86, 15, 0, 1
- osmičkové⁵: 0126, 015, 0, 01
- šestnáctkové: 0x56, 0X56, 0x3A, 0XCD, 0xCD, 0xCd, 0x0, 0x1

Dobrá rada:

U šestnáctkových konstant je vhodné vybrat si jeden druh zápisu a toho se důsledně držet. Doporučujeme způsob 0x3A, tj. malé x a velká písmena pro jednotlivé šestnáctkové číslice. Způsob 0xcd je chaotický a 0xcd nebo 0XCD méně přehledný. □

Pozor:

Pozor na skutečnost, že všechny konstanty jsou implicitně typu `int`. Potřebujeme-li inicializovat typ `long` konstantou, použijeme na jejím konci znak L, např. `k = 1234567890123L`; Pokud to neuděláme, hlásí překladač chybu `Integer literal out of range`. □

Poznámka:

Při pokusu inicializovat proměnnou typu `byte` konstantou např. 189 (tj. větší, než je rozsah typu `byte`, který je max. +127), hlásí překladač chybu `Incompatible type`. □

3.4.2 Znakový typ a jeho konstanty

Je pouze jeden – `char` a má velikost 16 bitů (dva bajty). Je to z toho důvodu, že Java vnitřně pracuje se znaky v kódování Unicode, což mimo jiné jednoznačně řeší problémy s různými kódovánimi češtiny, které jsou velmi nepříjemné v jiných programovacích jazycích.

Poznámka:

Podrobnosti o Unicode viz v [UJJ2]. □

Znakové konstanty jsou vždy uzavřeny do apostrofů a mohou být představovány:

1. **jedním znakem** – v případě neakcentovaných znaků běžně dostupných na klávesnici, např.: 'A', '1', '%'

⁵Osmičkovou konstantu 00 nedoporučujeme z estetických důvodů používat ;-)

2. posloupností '\uxxxx', kde xxxx jsou šestnáctkové číslice. Tento zápis se často používá v případě akcentovaných znaků, ale lze tak zapsat libovolný znak. Například '\u00E1' představuje 'á', '\u011B' pak 'ě' a '\u0041' znak 'A'

Z následující stručné tabulky Unicode získáte tato „nesmyslná“ čísla pro všechny běžně používané akcentované znaky.

00C1	Á	00C2	Â	00C4	Ä	00E1	á	00E2	â	00E4	ä
00DF	ß	010C	Č	010D	č	010E	Ď	010F	ď	00C9	É
00CB	Ě	011A	Ě	00E9	é	00EB	ë	011B	ě	00CD	Í
00CE	Î	00ED	í	00EE	î	013D	Ľ	013E	í	0147	Ň
0148	ň	00D3	Ó	00D4	Ô	00D6	Õ	00F3	ô	00F4	ô
00F6	ö	0154	Ŕ	0158	Ř	0155	ŕ	0159	ř	0160	Š
0161	š	0164	Ť	0165	ť	016E	Ů	00DA	ú	00DC	Ů
016F	ú	00FA	ú	00FC	ü	00DD	Ý	00FD	ý	017D	Ž
017E	ž										

3. „escape“ sekvencí

sekvence	hodnota	význam
'\n'	\u000A	nová řádka (newline, linefeed - LF)
'\r'	\u000D	návrat na začátek řádky (carriage return - CR)
'\t'	\u0009	tabulátor (tab - HT)
'\b'	\u0008	posun doleva (backspace - BS)
'\\'	\u005C	zpětné lomítko (backslash)
'\''	\u002C	apostrof (single quote)
'\"'	\u0022	uvozovky (quote)

Poznámka pro programátora v C či C++:

Další „escape“ sekvence známé z jazyka C:

- '\a' \u0007 písknutí (alert - BELL) – není podporováno
- '\f' \u000C nová stránka (formfeed - FF)
– je možné použít, ale nemá praktický význam

Pozor:

Mezi „escape“ sekvencemi a zápisem '\uxxxx' (unicode znak) je významný rozdíl. Unicode znaky mohou být kdekoli v programu, tj. například i ve jménech identifikátorů, kdežto „escape“ sekvence pouze na místě znakových nebo řetězcových konstant. Například:

```
int po\u010Det; // správný zápis deklarace identifikátoru počet
int pocet\n;    // chybný zápis identifikátoru s významem
                // pocetNovychRadek
```

□

- 4. osmičkovým zápisem** '\ooo' – vždy jsou nutné všechny tři osmičkové číslice (tj. i nevýznamové nuly jsou zde významové), např.: '\007'

Poznámka pro programátora v C či C++:

Zápis pomocí šestnáctkového čísla typu '\x07' používaný v jazyce C není možný.

□

3.4.3 Řetězcové konstanty (literály)

Stejnými způsoby, jako se tvoří znakové konstanty, se tvoří i řetězcové konstanty, pouze jsou uzavřeny do uvozovek. V jednom řetězci lze výše uvedené způsoby zápisu znaku libovolně kombinovat⁶, např.:

"Program kon\u010D\u00ED! \n\007"

Tisk tohoto řetězce způsobí vypsání Program končí! odřádkování a písknutí.

Poznámka:

Na některých počítačích je údajně problém s písknutím, pokud počítač není vybaven zvukovou kartou. Nepodařilo se mi to však ověřit.

□

Java umožňuje automatické zřetězování dlouhých literálů oddělených mezerami, tabelátory nebo novými řádkami, před kterými ale musí být znak + .

Tato vlastnost podstatně zpřehledňuje zápis např. při příkazech tisku. Následující tři literály jsou zcela ekvivalentní:

1. "Takhle vypada velmi dlouhy retezec"
2. "Takhle" + " vypada " + "velmi dlouhy retezec"
3. "Takhle vypada " + "velmi " +
 "dlouhy retezec"

Poznámka pro programátora v C či C++:

V jazyce C se pro spojování řetězců znak + nepoužívá.

□

⁶Ale není to příliš čitelný znění.

3.4.4 Logický typ a jeho konstanty

Používá se typ `boolean` o udávané velikosti jeden bit.⁷

Typ `boolean` může nabývat pouze dvou hodnot, představovaných logickými konstantami `true` (= logická 1) a `false` (= logická 0).

Pozor:

Logický typ je nepřevoditelný na celočíselné typy a naopak. Potřebujeme-li tuto operaci (v duchu jazyka C), je vhodné použít např.:

```
b = (i != 0);      // převod hodnoty int na boolean
i = (b) ? 1 : 0;  // převod hodnoty boolean na int    □
```

3.4.5 Reálné datové typy a jejich konstanty

Java rozeznává typy `float` a `double`. Oba vyhovují mezinárodnímu standardu IEEE 754. To znamená, že v Javě mají stejné zobrazení jako v jiných programovacích jazycích.⁸

<i>název</i>	<i>bitů</i>	<i>rozsah</i>		
<code>float</code>	32	$\pm 1.4E-45$	az	$\pm 3.4E+38$
<code>double</code>	64	$\pm 4.9E-324$	az	$\pm 1.7E+308$

Poznámka:

Skutečně jsou intervaly exponentů takto výrazně nesymetrické. □

Reálné konstanty se tvoří podle běžných zvyklostí, mohou začínat a končit desetinou tečkou, obsahovat znaménka, exponenty atd., např.:
`15.` , `56.8` , `.84` , `3.14` , `5e6` , `7E23` , `-7E+23` , `+7E-23`

Poznámka:

Reálná konstanta je automaticky typu `double`. Potřebujeme-li konstantu typu `float`, musíme na konec přidat příponu `F` (nebo `f`), např.:
`float flt = 3.14F;` □

Maximální a minimální hodnoty celočíselných i reálných typů lze získat pomocí konstant `MIN_VALUE` a `MAX_VALUE`, před které se dá ještě jméno příslušné třídy – `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`.⁹

⁷Tento údaj je nutno brát s rezervou, pravděpodobně zabírá celý bajt paměti.

⁸To může být velmi důležité v případě, když v Javě zpracováváme binární soubor obsahující reálná čísla připravený v jiném programovacím jazyce.

⁹Podrobnosti o těchto třídách viz v [9.6/153].

Například minimální hodnotu typů `int` a `float` získáme:

```
int i = Integer.MIN_VALUE;
float f = Float.MIN_VALUE;
```

Při operacích s reálnými čísly může výsledek operace nabýt několika „nenormálních“ hodnot, které ale nejsou chybové. Jedná se o hodnoty kladné a záporné nekonečno (příslušné konstanty ze tříd `Float` nebo `Double` jsou `POSITIVE_INFINITY` a `NEGATIVE_INFINITY`). Tyto hodnoty „získáme“, pokud např. dělíme kladné či záporné číslo nulou.

Další speciální hodnotou je *NaN* (*Not a Number*), kterou se podaří získat operací typu dělení nuly nulou.

Všechny tyto tři hodnoty lze otestovat pomocí metod třídy `isInfinite()` a `isNaN()`. Naší snahou je, pokud možno, se těmto hodnotám vyhýbat.

Příklad 4:

Hodnotu nekonečna můžeme získat mimo jiné i tak, že nám přeteče rozsah čísla – to je ukázáno na příkladě `Float.MAX_VALUE`

```
double nula = 0.0;
double vysledek = +5.0 / nula;
System.out.println(vysledek);
if (Double.isInfinite(vysledek) == true)
    System.out.println("nekonecno");
vysledek = -5.0 / nula;
System.out.println(vysledek);
if (Double.isInfinite(vysledek) == true)
    System.out.println("nekonecno");
System.out.println("MAX = " + Float.MAX_VALUE +
                   ", 2 * MAX = " + (2 * Float.MAX_VALUE));
vysledek = nula / nula;
System.out.println(vysledek);
if (Double.NaN(vysledek) == true)
    System.out.println("neni cislo");
```

Vypíše:

```
Infinity
nekonecno
-Infinity
nekonecno
MAX = 3.4028235E38, 2 * MAX = Infinity
NaN
neni cislo
```

Poznámka pro programátora v C či C++:

Neexistuje operátor `sizeof()`, takže se v programu nedají zjistit velikosti jakýchkoliv datových typů. U základních datových typů to ale není nutné, protože jejich velikosti jsou jednoznačně stanoveny v definici jazyka. □

3.5 Deklarace proměnných

Pod pojmem deklarace se mímí příkaz, který přidělí proměnné určitého typu jméno, paměťový prostor a počáteční hodnotu. Přidělení hodnoty znamená, že je každá proměnná při deklarování explicitně inicializovaná nějakou konstantou. Druhou možností je, že inicializační hodnota není uvedena a pak proběhne implicitní inicializace, která nastaví celočíselné typy na 0, reálné typy na 0.0 a typ `boolean` na hodnotu `false`. Typ `char` je implicitně inicializován na hodnotu '\u0000'.

Pozor:

Implicitní nastavení proměnných platí ale pouze pro členské proměnné třídy¹⁰, tj. proměnné, které nejsou deklarovány uvnitř žádné metody (nejsou lokální). Pro neinicializované lokální proměnné platí, že je jejich hodnota náhodná.

Naštěstí zde opět pomáhá striktní kontrola překladače `javac.exe`, který případné použití neinicializované proměnné označí jako chybu překladu. To znamená, že nedovolí neinicializovanou proměnnou použít jinak než na levé straně přiřazovacího příkazu.¹¹ □

Poznámka pro programátora v C či C++:

Pojmy deklarace a definice nemá smysl v Javě rozlišovat, protože příkaz, který by proměnné přiděloval pouze jméno a typ (jak to provádí deklarace v C) v Javě neexistuje. Deklarace v Javě tedy znamená to, co definice v C či C++.

Příklad 5:

Deklarace proměnných základních datových typů.

```
int i;
char c, ch;
float f, g;
```

¹⁰ Podrobně v [6.10.1/107].

¹¹ Překladače C tuto situaci označily maximálně varovným hlášením.

Poznámka:

Neexistují globální proměnné, tedy proměnné, které „patří“ celému programu a jsou tak odkudkoliv přístupné. V Javě každá proměnná „někomu patří“ – bud’ třídě nebo instanci nebo metodě (funkci). □

Dobré rady:

- Každá proměnná by měla být deklarována na samostatné řádce a okomentována¹², např.:

```
int cPlat;           /* celkovy plat */
```

- Každá explicitně inicializovaná proměnná by měla být deklarována samostatně, např.:

```
int i;
int j = 1;
```

- Mezi deklaracemi a příkazy je vhodné mít prázdnou řádku.

Poznámka:

Upřímně řečeno, poslední pravidlo se často porušuje, protože Java dovoluje deklarovat proměnné kdekoliv v kódu. Dochází tedy ke zmírnění pravidla na: „Máme-li všechny deklarace na jednom místě na začátku metody, pak je oddělujeme od příkazů prázdnou řádkou.“ □

Poznámka pro programátora v C či C++:

Deklarace mohou být v Javě kdekoliv uvnitř třídy, tedy ne jenom bezprostředně za otevírací závorkou { jako tomu bylo v jazyce C. Z důvodů přehlednosti se však snažíme mít všechny potřebné deklarace na začátku třídy i na začátku metody, stejně jako definice v C či C++. □

3.5.1 Deklarace proměnných s konstantní hodnotou

Tyto proměnné se běžně označují jako **konstanty** a deklarují se jako „normální“ proměnné, ale navíc se použije klíčové slovo **final**. Tak lze deklarovat konstantu dvěma způsoby:

1. `final int MAX = 10;`

Konstantě MAX nelze znova přiřadit jinou (ani stejnou) hodnotu. Překladač ohláší chybu `variable is declared final; cannot be assigned`

¹²Využímkou tvoří pomocné proměnné, které není třeba zvlášť komentovat.

```
2. final int MAX;
   ...
MAX = 10;
```

Konstanta `MAX` je deklarována jako **prázdná konstanta** (*blank final*) a pak je jedním jediným přiřazovacím příkazem inicializována. Další změny její hodnoty nejsou možné.

Dobrá rada:

Tento způsob je méně přehledný než první způsob a měl by se používat jen ve zcela výjimečných a odůvodněných případech. Jeden z těchto případů je použití ve **statickém inicializačním bloku** [8.12/138]. □

Poznámka:

Skutečné konstanty, které se dají používat i mimo třídu, musí být deklarovány vně jakékoli metody takto:

```
public class TridaSKonstantou {
    public static final int MAX = 10;
    . . .
}
```

Pak může být kdykoliv použita ve své třídě pod jménem `MAX` a mimo třídu pod jménem `TridaSKonstantou.MAX`. □

Poznámka pro programátora v C či C++:

Tím je zabráněno konfliktu jmen konstant, což se v jazyce C často stávalo při použití konstant vzniklých pomocí `#define`. Další výhodou oproti konstantám v C je, že konstanty v Javě mohou být pouze určitého typu, což využívá kompilátor pro silnou typovou kontrolu. □

Poznámky:

- V Javě existuje množství předdefinovaných konstant, např. již zmíněné `MIN_VALUE` a `MAX_VALUE` či `java.lang.Math.PI`¹³, tj. Ludolfovovo číslo.
- Jak již bylo uvedeno v [3.2/33], pro jména konstant se používají zásadně velká písmena; případným oddělovačem je podtržítko, např.:


```
public static final int MAX = 10;
      public static final int MAX_HODNOTA = 20;
```

¹³Toto je její naprostě jednoznačné pojmenování, obvykle se ale používá jen `Math.PI`

3.6 Operátor přiřazení

Přiřazení je nejčastějším příkazem ve většině programovacích jazyků.

Často se pracuje s pojmem ***l-hodnota*** (*I-value*). Pod tímto pojmem si můžeme představit něco, co má adresu v paměti, tedy nejčastěji proměnnou. Například proměnná `x` je l-hodnotou, ale konstanta `123` nebo výraz `(x + 3)` l-hodnotami nejsou.

Stručně lze říci, že l-hodnota je to, co může být na levé straně přiřazení.

Pozor na následující terminologii:

česky	anglicky	symbolicky	prakticky
výraz	expression	výraz	<code>i * 2 + 3</code>
přiřazení	assignment	<code>l-hodnota = výraz</code>	<code>j = i * 2 + 3</code>
příkaz	statement	<code>l-hodnota = výraz;</code>	<code>j = i * 2 + 3;</code>

Tedy:

- výraz má vždy hodnotu (číselnou nebo booleovskou),
- přiřazení je výraz a jeho hodnotou je hodnota přiřazená levé straně,
- přiřazení se stává příkazem, je-li ukončeno středníkem.

Příklad 6:

Různé typy přiřazovacích příkazů:

```
j = 5;
d = 'z';
f = f + 3.14 * i;
```

Skutečnosti, že přiřazení je výraz, se s výhodou často využívá v cyklech nebo podmínkách – viz [5.5.1/81].

Příklad 7:

Přiřazení je výraz a smí být tedy použit např. v podmínce. Asi vás ne-překvapí, že příkaz: `System.out.println(i);` vypíše na obrazovku hodnotu proměnné `i`.

```
int j, i = 5;
if (i == (j = 5))
    System.out.println(i);
```

Zde je vidět, že přiřazení (`j = 5`) je součástí podmínky `if()` a má hodnotu 5, která je dále porovnávána na rovnost s proměnnou `i`.

Pozor:

Jak je vidět v předchozím příkladu, porovnání je `==` ne pouze `=`. Naštěstí je Java velmi striktní při typové kontrole a záměnu `==` za `=` (např. `if (j = 5)`) hlásí jako chybu programu (*incompatible types*), protože na tomto místě očekává výraz typu `boolean` a dostává hodnotu typu `int`.

Z tohoto pravidla existuje jedna výjimka a to, když přiřazujeme typ `boolean`, např.:

```
boolean j = false;
if (j = true)
```

Zde nedochází k nekompatibilitě typů a Java považuje `if ()` podmínu-ku za vždy splněnou. □

Protože přiřazení je výraz, je možné několikanásobné přiřazení:

```
k = j = i = 2;
```

které se vyhodnocuje zprava doleva tedy:

```
k = (j = (i = 2));
```

3.7 Operátor přetypování

Přetypování neboli konverze datových typů se používá tehdy, máme-li k dispozici jeden datový typ (konstantu, proměnou, ...) a potřebujeme z něj vyrobit jiný datový typ (nejčastěji proměnnou nebo skutečný parametr metody).

Operátor se zapisuje ve formě kulatých závorek, uvnitř kterých je jméno datového typu, na který chceme přetypovat, např.:

<code>int i = 5;</code>	<code>char c = 'A';</code>
<code>double d;</code>	<code>int i = (int) c;</code>
<code>d = (double) i;</code>	<code>char d = (char) i;</code>

Pozor:

Přetypování má nejvyšší prioritu (viz [3.11/64]). Pokud přetypováváme jakýkoliv výraz, je nutné jej uzavřít do závorek, jinak bude přety-pován pouze první člen výrazu, např.:

<code>(double) i + j</code>	<code>// přetypuje se jen i</code>
<code>(double) (i + j)</code>	<code>// dobré</code>



Rozšiřující konverze pro základní datové typy jsou tyto:

`byte → short → int → long → float → double`

Schema znamená, že např. typ `short` lze kdykoliv převést na typy `int`, `long`, `float` a `double` bez operátoru přetypování, ale na typ `byte` jen s přetypováním.

Příklad 8:

Použití rozšiřujících konverzí:

```
short s = 10;
byte b;
int i;
float f;
i = s;
f = s;
// b = s; // zužující konverze
// chyba-possible loss of precision: short, required: byte
b = (byte) s;
System.out.println("s = " + s + ", b = " + b);
```

Vypíše:

`s = 10, b = 10`

Pozor:

Explicitní konverze na typ `byte` (`b = (byte) s;`) je zužující a má svá rizika – viz dále. □

Poznámka:

Typ `boolean` není číselný typ, proto jej na ně nelze převést.

```
int i = 5;
boolean b = true;
i = b; // chyba při překladu
b = i; // chyba při překladu
```

Řešení viz [3.4.4/40]. □

I pro referenční proměnné (viz [7.1/111]) lze použít rozšiřující konverze. Dvě nejčastější¹⁴ jsou:

- potomek na rodiče,
- třída implementující rozhraní na toto rozhraní.

Výsledkem těchto rozšiřujících konverzí jsou opět reference, tj. nevznikají žádné nové objekty. Příklady viz v [13.7/219].

¹⁴Podrobně viz např. [Tom].

3.7.2 Zužující konverze

Při provádění zužující konverze může dojít ke změně nebo ztrátě původní hodnoty. Z tohoto důvodu od nás komplilátor vyžaduje zapsání operátoru přetypování, čímž nás nutí, abychom si uvědomili případné důsledky. **Zužující konverze** pro základní datové typy jsou tyto:

`double → float → long → int → short → byte`

Příklad 9:

„Nevydařená“ zužující konverze:

```
short s = 300;
byte b;
b = (byte) s;
System.out.println("s = " + s + ", b = " + b);
b = (byte) 255;
System.out.println("b = " + b);
```

Vypíše:

```
s = 300, b = 44
b = -1
```

Zde došlo v prvním případě ke **ztrátě původní hodnoty**, protože typ `byte` je osmibitový a `short` šestnáctibitový. Oříznutí vyššího bajtu typu `short` prakticky znamená, že se číslo vydělí modulo 256 a zbytek po dělení (zde 44), který zůstává v nižším bajtu, se uloží do proměnné typu `byte`.

V druhém případě se jedná o **změnu hodnoty** (ke ztrátě informace nedošlo). Tento případ je zálužnější než první, protože máme většinou „zafixováno“, že bajt má rozsah 256 čísel. Je to pravda, ale nesmíme zapomenout na to, že polovina těchto čísel je záporná.

Opět existuje možnost zužujících konverzí pro referenční typy. Těchto konverzí je devět typů,¹⁵ pro nás bude nejdůležitější konverze rodiče na potomka, kdy potomek je specializovanější třída, tedy s užším zaměřením. Příklad bude uveden v [14.2/228].

3.7.3 Rozšiřující konverze se ztrátou přesnosti

U rozšiřujících konverzí bylo uvedeno, že je možné je provádět beze strachu, protože nedochází ke ztrátě informace. Toto tvrzení má jednu drobnou výjimku, která nás ale dokáže v nepříznivém případě nepřijemně potrápit. Proto jí bude věnována tato část.

¹⁵ Podrobně viz např. [Tom].

Při konverzi **int** na **float** nebo **long** na **float** či **double** nedochází ke ztrátě hodnoty, ale v určitých případech ke ztrátě přesnosti. Tato ztráta je v porovnání s původní hodnotou čísla zanedbatelná, takže numerickou správnost výpočtu pravděpodobně neohrozí.

Ovšem při zpětné konverzi tedy např.: **int → float → int** dostaneme jiné číslo. To pak může způsobit značné problémy např. při porovnávání.

Příklad 10:

Ztráta přesnosti u rozšiřující konverze.

```
int j, i = 1234567890;
float f;
double promile;
f = i;
j = (int) f;
promile = ((double) (j - i)) / ((double) i) * 1000.0;
System.out.println("i = " + i + "\nj = " + j);
System.out.println("promile = " + promile);
```

Vypíše:

```
i = 1234567890
j = 1234567936
promile = 3.7260000339066E-5
```

Je vidět, že chyba je sice nejspíše zanedbatelná, ale je.

3.8 Aritmetické výrazy

V aritmetických výrazech je nutné si připomenout, že výraz ukončený středníkem se stává příkazem, např.:

```
i = 2           // výraz s přiřazením
i = 2;          // příkaz
```

Poznámka:

Pouhý středník představuje často používaný **prázdný příkaz** (*null statement*), který se používá např. v cyklu **while** nebo **for** (např. [5.5.1/81]).

Aritmetické výrazy jsou tvořené pomocí operátorů, které se dají dělit do tří základních skupin:

1. **unární**, kde je pouze jeden operand (např. proměnná či konstanta),

2. **binární**, kde je operátor zleva i zprava obklopen operandy,
3. **ternární**, který by měl mít okolo sebe tři operandy, ale protože by to šlo těžko, jedná se o operátor složený ze dvou znaků, mezi nimiž je prostřední operand – podrobně viz [5.2/77].

3.8.1 Unární operátory

Běžně známé jsou unární minus - a unární plus +. Oba se používají stejně, jako jsme zvyklí v matematice. Oba se píší před operand (*prefix*), např.:

```
int j;
int i = -3;
j = -i;
```

3.8.1.1 Speciální unární operátory

Jsou to unární operátory složené ze dvou stejných znaků a mají význam: inkrement ++ dekrement --

Oba operátory se dají použít jako předpony (*prefix* – před operandem), tak i jako přípony (*suffix* – za operandem) s tímto odlišným významem:¹⁶

1. *++l-hodnota*
 - inkrementování před použitím,
 - *promenná* je nejprve zvětšena o jedničku a pak je tato nová hodnota vrácena jako hodnota výrazu.
2. *l-hodnota++*
 - inkrementování po použití,
 - je vrácena původní hodnota *proměnné* a ta je pak zvětšena o jedničku.

Pozor:

Časté chybné použití je:

45++ nebo --(i + j)



¹⁶ Pro operátor -- je význam analogický.

Příklad 11:**Různá použití operátorů ++ a --**

```
int i = 5, j = 1, k;
i++;           // i bude 6
j = ++i;       // j bude 7, i bude 7
j = i++;       // j bude 7, i bude 8
k = --j + 2;   // k bude 8, j bude 6, i bude 8
```

3.8.2 Binární operátory

Rozeznáváme operátory pro:

- + scítání
- odečítání
- * násobení
- / reálné dělení
- / celočíselné dělení
- % dělení modulo (tj. zbytek po celočíselném dělení)

Poznámka:

To, zda dělení bude celočíselné nebo reálné, závisí na typu operandů
 – dochází totiž k implicitní konverzi (viz [3.7/46]):¹⁷

int / int	– celočíselné
int / float	– reálné
float / int	– reálné
float / float	– reálné

□

Příklad 12:**Dělení a dělení modulo.**

```
int i = 5, j = 13;
j = j / 4;           // celočíselné dělení, j bude 3
j = i % 3;           // dělení modulo, j bude 2
```

Při aritmetických operacích může dojít k přetečení či podtečení, kdy dostaneme matematicky nesprávný výsledek, přičemž Java žádnou chybu neohlásí. Častý případ je sečtení dvou kladných čísel, kdy výsledek přesáhne hodnotu největšího kladného čísla.

¹⁷ Pro **double** platí totéž co pro **float**.

Příklad 13:

```
byte b = 126;  
System.out.println("b = " + b);  
b = b + 3;  
System.out.println("b = " + b);  
b = -126;  
b = b - 5;  
System.out.println("b = " + b);
```

Vypíše:

b = 126
b = -127
b = 125

Proti tomuto druhu chyb je velmi těžké se bránit, pokud potřebujeme zachovat typy proměnných. Pokud ne, pak je obrana jednoduchá – stačí použít jiný datový typ s větším rozsahem, např. `int` místo `byte`.

Poznámka:

Jediný případ, kdy program při výpočtu aritmetických výrazů ohláší chybu, je dělení nulou, ale jen u celočíselných typů – viz [3.4.5/41].

V tomto případě je vyvolána výjimka `ArithmetricException`. □

3.8.3 Přiřazovací operátory

Základní operátor přiřazení je `=`.

Java dává navíc k dispozici ještě celou škálu rozšířených přiřazovacích operátorů.

Místo přiřazení:

`l-hodnota = l-hodnota operátor výraz`
se velmi často používá zkrácený zápis:

`l-hodnota operátor= výraz`

Dají se použít následující přiřazení:

použití	význam
<code>l-hodnota += výraz</code>	<code>l-hodnota = l-hodnota + výraz</code>
<code>l-hodnota -= výraz</code>	<code>l-hodnota = l-hodnota - výraz</code>
<code>l-hodnota *= výraz</code>	<code>l-hodnota = l-hodnota * výraz</code>
<code>l-hodnota /= výraz</code>	<code>l-hodnota = l-hodnota / výraz</code>
<code>l-hodnota %= výraz</code>	<code>l-hodnota = l-hodnota % výraz</code>
<code>l-hodnota &= výraz</code>	<code>l-hodnota = l-hodnota & výraz</code>
<code>l-hodnota ^= výraz</code>	<code>l-hodnota = l-hodnota ^ výraz</code>
<code>l-hodnota = výraz</code>	<code>l-hodnota = l-hodnota výraz</code>
<code>l-hodnota >>= výraz</code>	<code>l-hodnota = l-hodnota >> výraz</code>
<code>l-hodnota <<= výraz</code>	<code>l-hodnota = l-hodnota << výraz</code>
<code>l-hodnota >>>= výraz</code>	<code>l-hodnota = l-hodnota >>> výraz</code>

Poznámka:

Operátory `>>` `<<` `>>>` budou vysvětleny v [3.10.5/62] a operátory `&`, `^` a `|` v [3.10.1/59]. □

Pozor:

Použití mezery pro oddělení operátoru a rovnítka je chyba, např.:
`j + = 5;` // chyba □

Příklad 14:

Různá použití přiřazovacích operátorů.

```
int i = 4, j = 3;
j += i;           //      j bude 7
j /= --i;         //      j bude 2, i bude 3
j *= i - 2;       //      j = j * (i - 2) = 2
                  // ne j = j * i - 2 = 4
```

Poznámka:

Zvětšit proměnnou `citac` o jednu je tedy možné čtyřmi rovnocenými¹⁸ způsoby. Programátor v Javě ale použije poslední z nich.

```
citac = citac + 1;
citac += 1;
++citac;
citac++;
```

¹⁸ Pokud nesouhlasíte se slovem rovnocenými, pak uvažujete správné, ale nezapomeňte, že kompilátor optimalizuje, takže uvedené příkazy se pravděpodobně převedou na stejný počet strojových instrukcí. □

3.9 Relační operátory

Pomocí relačních operátorů se vytvářejí booleovské výrazy nezbytné v jazykových konstrukcích řídících běh programu, jako jsou např. **for**, **if** atd.

Booleovské výrazy se zapisují pomocí těchto relačních operátorů:

- `==` rovnost,
- `!=` nerovnost,
- `&&` logický součin se zkráceným vyhodnocováním,
- `||` logický součet se zkráceným vyhodnocováním,
- `&` logický součin s úplným vyhodnocováním,
- `|` logický součet s úplným vyhodnocováním,
- `!` negace,
- `<` menší,
- `<=` menší nebo rovno,
- `>` větší,
- `>=` větší nebo rovno.

Pozor:

Operátory `&` a `|` je možné použít též jako bitové operátory – viz [3.10.1/59].

Operátory `<=` a `>=` a `!=` nelze „obrátit“. Zápis `=>` nebo `=<` nebo `=!` jsou chybné. □

- Je důležité si uvědomit rozdíl mezi porovnáním `==` a přiřazením `=`.
- `i = 5` je celočíselný výraz s hodnotou 5, kterou také přiřazuje do proměnné `i` – změní její původní hodnotu.
 - `i == 5` je booleovský výraz poskytující `true`, jestliže má proměnná `i` hodnotu 5, nebo poskytující `false`, má-li `i` hodnotu jinou, než 5; hodnota proměnné `i` se ani v jednom případě nezmění.

3.9.1 Zkrácené vyhodnocování logických součtů a součinů

Logický výraz obsahující logické součty a/nebo součiny lze vyhodnocovat buď zcela (tak je to používáno v Pascalu) nebo **zkráceně** (*short circuit*) (jako v jazyce C) nebo oběma způsoby – a to je možné v Javě.

Zkrácené vyhodnocování znamená, že operandy výrazu jsou vyhodnocovány zleva doprava a jakmile je možno určit konečný výsledek, vyhodnocování okamžitě končí.

To má význam zejména u složených logických výrazů, kdy při **logických**

- **součinech** první hodnota **false** způsobí ukončení vyhodnocování,
- **součtech** první hodnota **true** způsobí ukončení vyhodnocování.

Význam je jasný – urychlení výpočtu, které však někdy může být zaplacenou ne zcela zřejmým chováním programu.

Příklad 15:

Pro logický součin.

```
i = 1; j = 2; k = 3;
if (i == 2 && ++j == 3)
    k = 4;
System.out.println("i = " + i + ", j = " + j + ", k = " + k);
```

Vypíše:

i = 1, j = 2, k = 3

protože i == 2 je **false**, nedochází k vyhodnocení výrazu ++j a hodnota j tedy zůstává 2.

Příklad 16:

Pro logický součet.

```
i = 1; j = 2; k = 3;
if (i == 1 || ++j == 2)
    k = 4;
System.out.println("i = " + i + ", j = " + j + ", k = " + k);
```

Vypíše:

i = 1, j = 2, k = 4

protože i == 1 je **true**, opět nedochází k vyhodnocení výrazu ++j a hodnota j tedy zůstává 2.

Toto zkrácené vyhodnocování je většinou vhodné používat. Typický příklad použití je např.:

```
if (y != 0 && x / y < z)
```

kdy si nemusíme dělat starosti s případným dělením nulou, protože k němu v tomto případě nikdy nedojde.

První část logického výrazu: $y != 0$ totiž ukončí vyhodnocování celého výrazu před potenciálním dělením nulou.

Dobrá rada:

Pro urychlení programu je vhodné dávat v případě složeného logického výrazu na první místo (nejvíce doleva) ten výraz, jehož hodnota je:

- $u \&&$ nejpravděpodobnější nepravda (**false**),
- $u ||$ nejpravděpodobnější pravda (**true**). □

Příklad vhodného uspořádání:

```
if (maDite == true && vekDitete < 18 && plat < 15000)
    pridavyNaDeti = 1000;
```

Příklad nevhodného uspořádání:

```
if (plat < 15000 && vekDitete < 18 && maDite == true)
    pridavyNaDeti = 1000;
```

V druhém případě se zcela zbytečně zkoumá výše platu zaměstnance, pak věk jeho dítěte¹⁹ a teprve nakonec to hlavní, zda vůbec má děti.

Příklad pro logický součet nebudeme uvádět v zápisu Javy, ale v analógii ze života, na kterém je vidět, že zkrácené vyhodnocování není jen výmysl akademiků. Přijdete-li do restaurace, číšník se vás většinou zeptá: „Pivečko?“, neboť je to logická proměnná, která má u většiny zákazníků hodnotu **true** v logickém součtu nabídky nápojů. Málokdy se vám stane, že se jako první ozve: „Čajíček?“

3.9.2 Úplné vyhodnocování logických součtů a součinů

Někdy ovšem požadujeme, aby proběhlo vyhodnocení celého logického výrazu, bez ohledu na to, že je hned zpočátku zřejmé, že celý složený výraz je nepravda (u součinu) nebo pravda (u součtu). Pak je v Javě možné použít logické součty a součiny s **úplným vyhodnocováním**, kdy místo operátoru **&&** použijeme jen operátor **&** a místo **||** jen **|**.

¹⁹V případě, že děti ve skutečnosti nemá, je to potenciální chyba, protože proměnná **vekDitete** nemusí být správně inicializována.

Příklad 17:

Úplné vyhodnocování logického součinu.

```
i = 1; j = 2; k = 3;
if (i == 2 & ++j == 3)
    k = 4;
System.out.println("i = " + i + ", j = " + j + ", k = " + k);
```

Vypíše:

i = 1, j = 3, k = 3

přestože `i == 2` je **false**, dojde k vyhodnocení výrazu `++j` a hodnota `j` se tedy změní na 3 (ovšem platnost celého složeného výrazu to neovlivní).

Příklad 18:

Úplné vyhodnocování logického součtu:

```
i = 1; j = 2; k = 3;
if (i == 1 | ++j == 2)
    k = 4;
System.out.println("i = " + i + ", j = " + j + ", k = " + k);
```

Vypíše:

i = 1, j = 3, k = 4

přestože `i == 1` je **true**, je výraz `++j` opět vyhodnocen.

3.10 Bitové operace

Java disponuje také sedmi operátory, které se dají použít pro bitové operace – manipulují s jednotlivými byty čísla. Tyto operace nejsou sice příliš často využívané, ale přesto se s nimi pravděpodobně setkáte v počítačové grafice, při práci se zvuky a případně v aplikacích, které něco řídí²⁰.

Poznámka:

To, že je na začátku učebnice poměrně podrobně vykládána část jazyka, kterou dlouho vůbec nemusíte použít, je z toho důvodu, že operátory bitových operací jsou podobné (nebo i stejné) jako relační operátory. Proto je vhodné mít informace ucelené na jednom místě, aby v případě nejasnosti bylo možné rychle najít vysvětlení. Zdá-li se vám, že tuto část nepoužijete, klidně ji ve čtení přeskočte. Pro pochopení dalšího výkladu není podstatná. □

Bitové operace se týkají nižší úrovně jazyka – vykazují rysy jazyka assembleru.

²⁰Pro tyto aplikace vlastně Java vznikala.

Využitelné operátory jsou tyto:

- & bitový součin – AND
- | bitový součet – OR
- \sim bitový exklusivní součet – XOR (nonekvivalence)¹⁹
- << posun doleva
- >> posun doprava
- >>> neznaménkový posun doprava
- jedničkový doplněk²⁰ – negace bit po bitu – **unární** operátor

Poznámky:

- Argumenty bitových operací nesmějí být proměnné typů **float** a **double**.
- Pozor na skutečnost, že pro logické operace součinu a součtu s plným vyhodnocováním se používají naprostě stejné operátory (**&** a **|**). Z charakteru výrazu ale překladač pozná, o jaký typ operace se jedná – viz dále příklady.

3.10.1 Bitový součin

i-tý bit výsledku bitového součinu: $x \& y$
bude 1, pokud i-tý bit x a i-tý bit y budou 1, jinak bude 0

Čili každý bit výsledku bude záležet jen na odpovídajících bitech obou operandů a nebude ovlivněn žádnými jinými byty. To znamená, že se při bitovém součinu např. **int** číslo „rozpadne“ na 32 bitů, které jsou pro tuto operaci na sobě nezávislé.

`byte i = 7 & 9; // i bude 1, protože:`

$$\begin{array}{r} 0000\ 0111\ (= 7) \\ \& 0000\ 1001\ (= 9) \\ \hline 0000\ 0001\ (= 1) \end{array}$$

Příklad 19:

Test, zda je číslo liché²¹ či sudé, lze provést klasicky pomocí dělení modulo a s využitím bitových operací pomocí AND.

```
if (i % 2 == 0) // klasicky
    System.out.println(i + " je sude");
```

¹⁹ Znak „stříška“ nebo „šipka nahoru“.

²⁰ Znak „tilda“ neboli „vlnka“.

²¹ Lichá čísla mají nultý bit nastaven na 1 – byty jsou číslovány od nuly zprava doleva.

```
if ((i & 1) == 0)           // bitově
    System.out.println(i + " je sude");
```

Na příkladu je vidět, že s bitovým součinem jsme si příliš příkaz nezkrátili. Častější použití bitového součinu je pro „vymaskování“ (tj. nastavení na nulu) určitých bitů. Chceme-li například v nějaké grafické operaci vynulovat každý sudý bit bajtové proměnné a liché bity ponechat beze změny, použijeme např.:

b = b & 0xAA; // 0xAA je 1010 1010 binárně
nebo zkráceně:

```
b &= 0xAA;
```

Poznámka pro programátora v C či C++:

Java díky své silné typové kontrole nedává možnost splést si logický a bitový součin.²²:

```
int i = 1, j = 2, m;
boolean k;
k = i && j;           // nelze - chyba překladu
m = i & j;            // lze - bitový součin
```

3.10.2 Bitový součet

i-tý bit výsledku bitového součtu: $x \mid y$

bude 1, pokud i-tý bit x nebo i-tý bit y bude 1, budou-li oba nulové, bude výsledek 0, např.:

byte i = 7 | 9; // i bude 15, protože:

$$\begin{array}{r} 0000\ 0111 \ (\text{=}7) \\ \mid 0000\ 1001 \ (\text{=}9) \\ \hline 0000\ 1111 \ (\text{=}15) \end{array}$$

Bitový součet se často používá pro nastavení určitých bitů na 1, přičemž ostatní bity nechá nedotčeny. Například chceme každý lichý bit nastavit na jedničku a sudé bity ponechat beze změny.

b = b | 0xAA; // 0xAA je 1010 1010 binárně
nebo zkráceně:

```
b |= 0xAA;
```

²²Podobně to platí i pro bitový a logický součet.

V aplikacích logického řízení nebo v grafických operacích se poměrně často používá „stavové slovo“, což je celočíselná proměnná (nejčastěji typu byte), jejíž každý bit uchovává nezávislou informaci. Pro manipulaci se stavovým slovem se pak s výhodou využívají konstanty²³ a bitové operace.

Příklad 20:

Manipulace s grafickým objektem:

```
final byte VIDITELNY = 1;
final byte PREMISTITELNY = 2;
final byte MENITELNY = 4;
final byte SMAZATELNY = 8;
```

```
byte stav = 0;           // s tímto objektem nelze provádět nic
stav |= VIDITELNY;     // od této chvíle je viditelný
if ((stav & VIDITELNY) == VIDITELNY)
    System.out.println("je viditelný");
```

Je zřejmé, že další vlastnosti se dají přidávat stejným způsobem, např.:

```
stav |= MENITELNY;
```

Na problém ale narazíme, chceme-li nějakou vlastnost objektu zrušit. Pomocí bitového AND to nejde tak snadno. Řešení je popsáno v další části.

3.10.3 Negace bit po bitu

Pro tuto akci se často používá také název **jedničkový doplněk** (*one's complement*). Pozor na skutečnost, že je to **unární** operátor.

Příkaz: `~x`

převrátí nulové bity na jedničkové a jedničkové na nulové.

Tento operátor se používá např. v situaci, která byla popsána v předchozím příkladu. Chceme-li zrušit nějakou vlastnost, použijeme bitového AND, ale operandem bude jedničkový doplněk příslušné konstanty, např.:

```
stav &= ~VIDITELNY;           // od této chvíle je neviditelný
if ((stav & VIDITELNY) == VIDITELNY)
    System.out.println("viditelný");
```

Nastavení všech příznaků na 0 je:²⁴

```
stav &= ~(VIDITELNY | PREMISTITELNY | MENITELNY | SMAZATELNY);
```

²³ Jejich hodnoty musí být mocniny dvou.

²⁴ Jednodušší je ale přiřadit proměnné stav rovnou nulu :-)

3.10.4 Bitový exkluzivní součet

i-tý bit výsledku bitového XOR: $x \wedge y$

bude 1, pokud i-tý bit x se nerovná i-tému bitu y

budou-li oba nulové, nebo oba jedničkové, bude výsledek 0

Tato operace se často používá v grafice pro rychlé nalezení rozdílu v jednotlivých bitech dvou bajtů. Například:

`byte i = 7 ^ 5; // i bude 2, protože:`

$$\begin{array}{r} 0000\ 0111 \quad (=7) \\ \wedge\ 0000\ 0101 \quad (=5) \\ \hline 0000\ 0010 \quad (=2) \end{array}$$

3.10.5 Operace bitového posunu doleva

Příkaz: $x << n$

posune bity v x doleva o n pozic

Při tomto posunu se zleva bity ztrácí – jsou vytlačovány – a zprava jsou doplnovány 0.

Bitový posun doleva se občas používá pro rychlé násobení dvěma, respektive mocninou dvou, např. příkaz:

$x = x << 1;$ nebo totéž: $x <<= 1;$

vynásobí x dvěma

nebo příkaz: $x <<= 3;$ násobí x osmi ($8 = 2^3$)

3.10.6 Operace bitového posunu doprava znaménkově

Příkaz: $x >> n$

posune bity v x doprava o n pozic

Při tomto posunu se zprava bity ztrácí – jsou vytlačovány. Zajímavá situace ale nastává v případě nejlevějšího bitu, kdy si nejvíce významový (znaménkový) bit ponechává svoji hodnotu a jeho obsah se kopíruje do bitu vpravo. Důvodem je to, že jsou všechny celočíselné typy v Javě znaménkové.

Bitový posun doprava má opačný význam než posun doleva, tedy celočíselné dělení dvěma, respektive mocninou dvou, např. příkaz:

$x = x >> 1;$ nebo $x >>= 1;$

dělí x dvěma

nebo příkaz: $x >>= 4;$ dělí x šestnácti ($16 = 2^4$)

Příklad 21:

Význam „držení znaménka“ si ukážeme na dvou příkladech:

```
byte x = 16;
```

```
x >>= 2;
```

představuje dělení čtyřmi (2^2) a bitově je znázorněno:

```
0001 0000 = +16 před posunem
```

```
0000 0100 = +4 po posunu o dva bity doprava
```

```
byte x = -16;
```

```
x >>= 2;
```

představuje dělení čtyřmi (2^2) a bitově je znázorněno:²⁵

```
1111 0000 = -16 před posunem
```

```
1111 1100 = -4 po posunu o dva bity doprava
```

což je správný výsledek, protože při doplňování zleva nulou bychom dostali $0011\ 1100 = +60$, což určitě není $-16 / 4$

Poznámka:

Tyto operace „násobení“ a „dělení“ dvěma pomocí posuvů jsou rychlejší než skutečné násobení a dělení. Při výpočtech s celými čísly, kde jde o rychlosť, je dobré zamyslet se nad tím, zda by se nedaly využít. □

Příklad 22:

Než násobit 80, je často²⁶ lepší „násobit“ 64 a 16 a sečít výsledek:

```
i = j * 80; // pomalejsi
```

```
i = (j << 6) + (j << 4); // rychlejsi
```

Pozor:

Je třeba si uvědomit, že priority operátorů $>>$ a $<<$ jsou velmi nízké, takže je nutno téměř vždy závorkovat. □

3.10.7 Operace bitového posunu doprava neznaménkově

Příkaz: $x >>> n$

posune bity v x doprava o n pozic.

Při tomto posunu se zprava bity ztrácí – jsou vytlačovány a zleva jsou doplňovány nulami. Tato operace se nedá pokládat za korektní dělení

²⁵ Nezapomeňte, že čísla jsou interně zobrazena v doplňkovém kódu.

²⁶ Závisí to na efektivitě implementace operace násobení v instrukčním souboru příslušného procesoru.

mocninou dvou (pro kladná čísla funguje, pro záporná ne – viz dříve). Tento operátor je použit proto, že v bitových operacích je poměrně častý požadavek skutečně bity posunout tak, jako by se jednalo o neznaménkové číslo. Pak je operátor `>>>` vhodný. Pozor na skutečnost, že funguje správně jen pro typy `int` a `long`.

Příklad 23:

```
int x = 16;
x >>>= 2;
```

představuje posun o dva bity doprava – bitově je znázorněno:

0000 0000 0000 0000 0000 0000 0001 0000 = 16	před
0000 0000 0000 0000 0000 0000 0000 0100 = 4	po

Pokus se záporným číslem:

```
int x = -16;
x >>>= 2;
1111 1111 1111 1111 1111 1111 1111 0000 = -16      před
0011 1111 1111 1111 1111 1111 1111 1100 = +1073741820 po
```

Příklad 24:

Operátor `>>>` se také často používá pro získání hodnoty konkrétního bitu. Používá se jednoduchý trik, kdy se posun opakuje tak dlouho, až je požadovaný bit na nejnižší pozici.

Metoda `getBit()` vrátí hodnotu i-tého bitu svého parametru.

```
int getBit(int x, int i) {
    if (i >= 32) // int je čtyřbajtový, tj. max. 31
        return (-1);
    else
        return ((x >>> i) & 1);
}
```

Protože příkaz: `x >>> i`

posune i-tý bit na poslední pozici doprava a příkaz: `& 1`
nastaví na nulu všechny vyšší bity.

3.11 Priority vyhodnocování operátorů

Dále bude uvedena tabulka priorit vyhodnocování všech dosud popsaných operátorů. Její význam je v tom, že v případě výrazu složeného z více operátorů je jednoznačně určeno, který má přednost (tj. bude vyhodnocován dříve). V tabulce jsou uvedeny prioritně vyšší operátory dříve.

<i>operátor</i>	<i>význam</i>
- [] (typ)	reference, indexování, přetypování
! ++ -- - +	unární operátory
* / %	násobení a dělení
+	sčítání a odečítání
<< >>> >>	posuny bitů
< <= >= <	relační operátory
== !=	rovnost a nerovnost
&	logický i bitový součin (AND)
^	logický i bitový XOR
	logický i bitový součet (OR)
&&	logický zkrácený součin
	logický zkrácený součet
? :	ternární operátor
= += -= *= atd.	přiřazovací operátory
,	operátor čárka ve for

Dobrá rada:

Nejsme-li si jisti s prioritami, platí místo zdlouhavého hledání či pokusů zlaté pravidlo:

„Máš-li pochyby, závorkuj!“

**Poznámka:**

Podobné tabulky se většinou uvádějí i včetně směru vyhodnocování (tj. zleva doprava nebo zprava doleva). To má význam, jsou-li ve výrazu dva operátory se stejnou prioritou. Praktické zkušenosti ale říkají, že téměř nikdo tyto informace nepoužije. Komplikované výrazy je vhodné rozepsat do více výrazů nebo použít závorek. Je to rychlejší (z hlediska psaní programu) a přehlednější způsob, než bádat nad směry vyhodnocení.

**Dobré rady:**

- Každý program by měl mít na začátku komentář, kde je stručně napsáno, co program dělá.
- Oddělujte deklarace proměnných prázdnou řádkou od příkazů.
- Používejte odsazení vložených příkazů a snažte se vždy jen o jeden příkaz na jedné řádce.

- Nepoužívejte „magická čísla“, místo nich použijte proměnnou `final`.
- Java se stále vyvíjí – při využívání metod z API zkонтrolujte, zda používáte stejnou verzi překladače i dokumentace.
- Pokud po příkazu `System.out.print()` neuvidíte na obrazovce to, co jste chtěli vytisknout, je to možná způsobeno bufferováním výstupu. Zkuste použít příkazy:

```
System.out.flush();
System.out.print("Ahoj");
System.out.flush();
```

- Oddělujte prázdnou řádkou každý celistvý úsek kódu a tento úsek okomentujte na jeho začátku stručným komentářem.

Běžné chyby:

- Pokus o dělení modulo reálným číslem: `i = 5 % 3.14;`
- Mezery mezi částmi operátorů `= = ! = > = < =`
- Přehození znaků v operátorech `=! => ==`
- Záměna přiřazovacího operátoru `=` za porovnání `==`
(na rozdíl od C to způsobí chybu při překladu)

Cvičení:

1. Napište program, který pro všechny celočíselné základní datové typy vypíše jejich rozsah. Ověřte výsledky s tabulkou na str. 36.
2. Napište program, který totéž vypíše pro reálné datové typy. Výsledek porovnejte s tabulkou na str. 40.
3. Napište program, který vydělí dvě reálná čísla. Pokud bude výsledek `NaN` nebo nekonečno, program vypíše: `Delitel je příliš male cislo!`. V ostatních případech vypíše podíl.
4. Napište program, který krátce pípne.
5. Napište program, který vypíše jen celou část proměnné deklarované jako: `double d = 3.14;`
6. Napište program, který využije zkráceného i plného vyhodnocení logického součinu.
7. Napište podobný program pro logický součet.

4 Terminálový vstup a výstup

Používání terminálového vstupu a výstupu (V/V) by mělo být omezeno pouze na **cvičné** či **zkušební** příklady.¹ Důvody jsou dva:

- Java umožňuje používat grafická uživatelská rozhraní (GUI – *Graphical User Interface*), která jsou pro tyto účely ve skutečných programech mnohem vhodnější².
- Použitím terminálového V/V vytvoříme program, který není 100% čistá Java (tzn. 100% přenositelný na všechny platformy), protože některé operační systémy – jako např. MAC OS – vůbec terminálový V/V nemají.

4.1 Balík `java.io`

Aby bylo možno správně používat všechny metody pro vstup a výstup, je nutné na začátku programu připojit **balík**, v němž jsou popsány. To se provede pomocí příkazu:

```
import java.io.*;           // středník je nutný!
```

Poznámka:

Používáme-li pouze výstup pomocí metody `System.out.print()`, není, jak jsme již viděli, uvedený příkaz pro připojení balíku nutný. □

4.2 Formátovaný výstup pomocí `System.out.print()`

Jedná se o způsob tisku, kdy při se při výstupu číselné proměnné automaticky konvertuje její hodnota na odpovídající posloupnost číslic. Co to prakticky znamená?

¹Tato učebnice je celá cvičná ;–)

²Rehušel ale pracovník

Mějme proměnnou deklarovanou jako: `int i;`

Tato proměnná zabírá v paměti vždy čtyři bajty. Pokud bychom ji (respektive obsah této paměti) tiskli neformátovaně, vytiskly by se vždy dva znaky (Java má dvoubajtové znaky), které by odpovídaly příslušným znakům v kódové tabulce Unicode. Například po přiřazení:

`i = 4849714; (tj. šestnáctkově 0x004A0032)`

by se při neformátovaném tisku vypsaly na obrazovku znaky `J2`, protože `\u004A` představuje znak `J` a `\u0032` znak `2`.

My bychom však spíše chtěli vytisknout sedm znaků `4849714`, které představují skutečnou hodnotu proměnné `i`. Pokud by ovšem v proměnné `i` byla hodnota `3` (`i = 3;`), pak požadujeme vytisknout pouze jeden znak, tj. `3`.

Takže při formátovaném tisku proměnné `i`, která zabírá v paměti vždy čtyři bajty, může kolísat počet vytištěných znaků od 1 do 11, např. od 0 do `-2147438648`, kdy je nutné uvažovat i znaménko.

Tuto konverzi čísla na řetězec znaků můžeme provést několika způsoby (např. `String.valueOf(i);`), ale v případě tisku je nejsnažší použít metodu `System.out.print()`.

Metoda `System.out.print()` převede proměnnou, která je jejím parametrem, na řetězec představující dekadickou hodnotu proměnné a ten vytiskne.

Například: `i = 5; System.out.print(i);` vytiskne řetězec³ `5`

Poznámka pro programátora v C či C++:

Pokud se ted' začínáte shánět po informacích, jak změnit formát výpisu, např. vynutit nevýznamové nuly, tisknout na určitý počet míst, převádět do jiné než desítkové soustavy, pak vězte, že to pomocí `System.out.print()` jednoduše nejde.

To, co bylo v C možné v široké škále specifikovat pomocí formátovačiho řetězce, zde nelze. Např. `printf("%02X", i);` což, jak jistě víte, znamenalo „vytiskni proměnnou `i` šestnáctkově vždy na dvě místa, tiskni i nevýznamové nuly“. Ovšem nezoufejte, je toho možné dosáhnout použitím metod z balíku `java.text`. □

Chceme-li vytisknout více proměnných, případně i doplněných textem, je to možné. Například:

`System.out.print("Toto je hodnota i: " + i + " a toto j: " + j);`

³Obsahující jen jeden znak.

Je možné samozřejmě odřádkovat výpisem znaku '\n' nebo řetězce "\n", přičemž řetězec je lepší než znak, protože odpadá konverze znaku na řetězec:

```
System.out.print("Toto je hodnota promenne i: " + i + "\n");
ale pohodlněji dosáhneme téhož použitím metody println(), tedy:
System.out.println("Toto je hodnota promenne i: " + i);
```

Ve výpisu lze odřádkovat i několikrát:

```
System.out.println("Toto je promenna i: "+i+"\nna toto j: "+j);
```

Příklad 25:

Různé způsoby tisku.

```
i = 4; j = 7;
```

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

vypíše: Soucet je 47

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

vypíše: Soucet je 11

```
System.out.println("Pracovali na 100%");
```

vypíše: Pracovali na 100%

Poznámka pro programátora v C či C++:

Znak % není při výpisu nutné nijak zdvojovovat. □

```
System.out.println("Soucet je "+(i+j)+"\tSoucin je "+(i*j));
```

vypíše: Soucet je 11 Soucin je 28 a odřádkuje

```
System.out.println("\007Chyba, pokus o deleni nulou.");
```

pískne a vypíše: Chyba, pokus o deleni nulou.

```
System.out.println("Toto je \"backslash\": '\\\"');
```

vypíše:⁴ Toto je "backslash" : '\'

```
System.out.println("Toto je 'backslash' : '\\\"');
```

vypíše: Toto je 'backslash' : '\'

Příklad 26:

Při tisku znaků (uzavřených do apostrofů, např. '\n') dejte pozor na jednu nepříjemnou věc:

⁴Proto se při tisku používají často pouze apostrofy místo uvozovek, neboť s apostrofy nejsou žádné problémy.

```

char c = '\n';
int i = 5;
System.out.println(i + c);
System.out.println(i + '\n');
System.out.println("i " + i + '\n');
System.out.println(i + '\n' + " i");

```

Vypíše:

15
15
i 5

15 i

Zdánlivě nepochopitelné číslo 15 vzniklo tak, že se vzala hodnota proměnné `i` (tj. 5) a sečetla se (operátor `+` zde funguje jako operátor součtu dvou čísel, nikoliv jako operátor pro spojení řetězců) s hodnotou 10, což je hodnota znaku `'\n'`, konvertovaná implicitní konverzí na číslo typu `int`. Výsledná hodnota (15) je převedena na řetězec `"15"` a ten je vytisknuto.

Že se jedná o záladnou vlastnost (není to chyba!), je vidět z předposledního výpisu, kde je již všechno v pořádku. Zde zafungovalo vyhodnocování operátorů zleva doprava, takže se nejdříve připravil řetězec `"i 5"` a pak se k němu přidal řetězec `"\n"` vzniklý ze znaku `'\n'`. Poslední výpis je opět „chybný“.

Příklad 27:

Chceme-li při tisku změnit typ proměnné, např. chceme získat ASCII hodnotu nějakého znaku, můžeme použít přetykování.

```

char c = 'A';
System.out.println("Znak "+c+" ma ASCII hodnotu: "+(int) c);

```

4.3 Formátovaný vstup

Použití formátovaného vstupu má stejnou filosofii jako použití formátovaného výstupu. Potřebovali bychom metodu, která bude fungovat takto: Zadáme-li z klávesnice znaky (respektive bajty), nače je a pak je dohromady převede na číslo.

Tak např. zadáme-li z klávesnice číslo 123, přečteme ve skutečnosti tři na sobě nezávislé bajty s hodnotami 49, 50, 51 (znaky '1', '2' a '3')

ukončené navíc jedním znakem odřádkování (případně více znaky, což záleží na operačním systému⁵). Tyto bajty by se načetly jako pole bajtů, převedly by se na řetězec, ořízly by se ukončovací znaky a řetězec by se převedl na čtyřbajtové číslo typu `int`. Tato operace ale není přímo podporována žádnou knihovní metodou inverzní k `print()`.

Z tohoto důvodu je vhodné připravit si vlastní metodu pro vstup čísla typu `int`, kterou pak budeme dále používat.

Příklad 28:

Metoda, která načítá celé číslo z klávesnice.

```
import java.io.*;
public class VstupInt {
    public static int ctiInt() {
        byte[] pole = new byte[20];
        String nacteno;
        int i;

        try {
            System.in.read(pole);
            nacteno = new String(pole).trim();
            i = Integer.valueOf(nacteno).intValue();
            return i;
        }
        catch (IOException e) {
            System.out.println("Chybne nactene cislo!");
            return 0;
        }
    }

    public static void main(String[] args) {
        System.out.print("Zadej cele cislo: ");
        int i = ctiInt();
        System.out.println("i = " + i);
    }
}
```

V příkladu se objevují dosud neznámé konstrukce, které budou podrobně vysvětleny v dalších kapitolách. Nyní berte metodu `ctiInt()` jako danou.⁶

⁵V Unixu je to `\n`, v MS Windows `\r\n`.

⁶Pokud vám to pomůže, můžete se na ní dívat jako na knihovní ; -)

Poznámka:

Potřebujete-li načíst číslo jiného typu, dejme tomu **double**, pak se změní hlavička metody na:

```
public static double ctiDouble()
a řádka konverze řetězce na:
d = Double.valueOf(nacteno).doubleValue();
kde d je typu double.
```

Pro ostatní základní datové typy je to analogické. □

Poznámka pro programátora v C či C++:

Opět zde není ekvivalent formátovacího řetězce, který se vyskytuje ve **scanf()**. □

4.4 Neformátovaný vstup jednoho znaku

V tomto případě chceme z klávesnice přečíst jeden znak, tak jak byl zadán.

Zde je situace jednodušší než u formátovaného vstupu čísla, protože existuje knihovní metoda **System.in.read()**, která dokáže znak přečíst. Problém je v tom, že tato metoda může vyvolat výjimku⁷ a bez toho, že tuto výjimku programově ošetříme, nelze program přeložit.

Příklad 29:

Načtení znaku z klávesnice.

```
import java.io.*;
public class VstupZnaku {
    public static void main(String[] args) throws IOException {
        char c;
        c = (char) System.in.read();
        System.out.println("c = " + c);
    }
}
```

Příklad 30:

Pro ty, kteří budou používat vstup z klávesnice častěji, je lepší napsat si samostatnou funkci, která se o ošetření výjimky transparentně postará. Uvedenou metodu **ctiZnak()** budeme používat v dalších kapitolech.

⁷Výjimku mohla vyvolat i metoda **ctiInt()** z předchozí části. Podrobnosti o výjimkách viz v [16/241].

tolách, vysvětlení dosud neznámých konstrukcí **try** a **catch** naleznete v [16.2.2/247].

```
import java.io.*;
public class VstupZnaku {
    public static char ctiZnak() {
        try {
            return (char) System.in.read();
        }
        catch (IOException e) {
            return '\uFFFF';
        }
    }

    public static void main(String[] args) {
        char c;
        c = ctiZnak();
        System.out.println("c = " + c);
    }
}
```

Poznámka:

Jako platné znaky se načtou i znaky odřádkování, konkrétně ve Windows NT znaky CR ('\r') a LF ('\n'). To, zda bude konec řádky '\r' a '\n' nebo pouze '\n', jako je tomu v UNIXu, se snadno určí podle hodnoty systémové vlastnosti `line.separator`, jejíž použití je např.:
`String odradkovani = System.getProperty("line.separator");`

Ve Windows NT bude mít řetězec odřádkování hodnotu "\r\n" a v UNIXu "\n". □

Poznámka pro programátora v C či C++:

Pozor na skutečnost, že případný znak '\r' není nijak systémem filtrován („požírána“) a je s ním třeba počítat. To znamená, že konec řádky je nutné testovat na shodu s řetězcem `odradkovani` – viz předchozí příklad, nikoliv pouze na shodu se znakem '\n'. □

Pozor:

Metoda `System.in.read()` pracuje spolehlivě jen pro znaky z dolní poloviny ASCII tabulky (znaky s kódy 0 až 127). To znamená, že české (obecně akcentované znaky) není možné vždy korektně načíst. Vysvětlení viz v [UJJ2]. □

Běžné chyby:

- Míchání operátorů + ve významu sečtení a ve významu spojení řetězců:

`x = 3; System.out.println("x + 1 = " + x + 1);`
vytiskne: x + 1 = 31, kdy 31 je složeno z číslic 3 a 1.

Cvičení:

1. Napište program, který přečte znak a vypíše znak s hodnotou o jednu vyšší. Použijte explicitní konverzi.

vstup: A

výstup: B

2. Napište program, který přečte znak a vypíše jeho Unicode hodnotu v desítkové soustavě.

vstup: C

výstup: C = 67

3. Napište program, který připočítává 25% daň, např.:

vstup: Zadejte cenu bez dane: 100

výstup: Prodejní cena s daní (25%): 125

4. Napište program, který přečte reálné číslo a vypíše jeho celou část.

5. Napište program, který přečte tři malá písmena a vypíše je jako tři velká písmena. (Od hodnoty malého písmene odečtěte 32 a dostanete velké písmeno – pozor ale na nutnost přetypování).

6. Napište program, který načte tři celá čísla a vypíše jejich součet, součin, průměr a největší a nejmenší číslo.

5 Řídicí struktury

Tato kapitola vysvětluje konstrukce pro popis programem realizovaného algoritmu. Uvedené konstrukce je vhodné pečlivě prostudovat včetně příkladů, protože se bez nich při jakémkoliv dalším programování určitě neobejdeme.

Poznámka:

V následujících příkladech nebude pouze z důvodu šetření místem uváděna řádku se jménem třídy a jí odpovídající uzavírací složená závorka } . Jméno třídy totiž zatím není pro pochopení zde vysvětlovaného problému nutné. Správně by mělo vždy být:

```
public class NejakeJmenoTridy {
    public static void main(String[] args) { ... }
```

5.1 Příkaz if a příkaz if-else

Příkaz **if** je jeden z nejužívanějších příkazů a Java jej umožňuje použít jak v neúplné podmínce – pouze **if**, tak i v uplné podmínce **if-else**.

5.1.1 Neúplná podmínka

Pokud je podmínka splněna, provede se **příkaz**, jinak se **příkaz** přeskočí. Syntaxe je:

```
if (booleovský výraz)      // závorky jsou nezbytné
    příkaz;
```

Například:

```
if (i > 3)
    j = 5;
```

Poznámka pro programátora v C či C++:

V závorkách za **if** musí být booleovský výraz, jehož hodnotou je buď **true** nebo **false**. Konstrukce typu **if (i)** či **if (pocetZnaku())** jsou komplátorem považovány za chvbné (**není-li i tvůrce boolean**).

Je nutné použít booleovský výraz,¹ např. `if (i != 0)` či `if (početznaku() > 0)`

Z tohoto důvodu není možné napsat častou céckovou chybu se zapomenutými závorkami kolem přiřazení. Kompilátor Javy očekává typ `boolean` a ohlásí chybu překladu.

C

```
if (c = getchar() != 'A')    if (c = ctiznak() != 'A')
překlad bez chyb
```

Java

```
if (c = ctiznak() != 'A')
překlad s chybou
```

□

Za `if` (nebo za `else`, viz dále) může být jen jeden příkaz. Potřebujeme-li jich uvést více, je třeba použít složený příkaz, kdy se příkazy uzavřou do bloku pomocí závorek `{ a }`, a tento blok nesmí být ukončen středníkem! Například:

```
if (i > 3) {
    j = 5;
    i = 7;
}                                // žádný středník!
```

Dobrá rada:

Z hlediska čitelnosti programu a jeho dalších úprav je vhodné dávat do složených závorek (do bloku) i jeden příkaz.

□

Poznámka:

Pokud je to možné, uvádějte testy a podmínky v aserci (kladné znění tvrzení) a ne v negaci. To má význam zvláště u složených podmínek, např. chceme provést příkaz, když se proměnná `c` nebude rovnat znaku '`a`' ani '`b`' ani '`c`'

nevhodné: `if (!(c == 'a' || c == 'b' || c == 'c'))`

lepší: `if (c != 'a' && c != 'b' && c != 'c')`

□

5.1.2 Úplná podmínka

Příkaz `if` je možné rozšířit i o část `else`, která se provede, když podmínka za `if` není splněna. Tím dostaneme tzv. úplnou podmínku.

¹Pokud jste ale psali programy v C dle doporučení, jste na tento zápis již určitě zvyklí

```
if (booleovský výraz)    // závorky jsou nutné
    příkaz1;                // středník je nutný
else
    příkaz2;
```

Například:

```
if (i > 3)
    j = 5;
else
    j = 1;
```

Poznámka:

Pokud je do sebe zanořeno více příkazů if, pak platí pravidlo, že se else vztahuje vždy k nejbližšímu nespárovanému if. Grafická struktura zápisu toto pravidlo nezmění! Chceme-li zajistit „spárování“ else s jiným if, je nutné použít složených závorek, např.:

chybně

```
if (dalSiPivo == true)
    if (desitku == true)
        pocetDesitek++;
    else
        pocetLimonad++;
```

správně

```
if (dalSiPivo == true) {
    if (desitku == true)
        pocetDesitek++;
}
else
    pocetLimonad++;
```

□

5.2 Podmíněný výraz – ternární operátor

Ternární („trojity“) operátor by měl mít tři operandy. Protože by ale nebylo možné napsat všechny tři do jedné řádky, skládá se ternární operátor ze dvou znaků ? a :, které jsou obklopeny třemi operandy.

Zjednodušeně řečeno, ternární operátor funguje jako **if-else**. Proč se tedy používá? Protože **if-else** není výraz, ale příkaz, takže nejde použít tam, kde je potřebný výraz. V mnoha případech je to jedno, ale existují místa, kde je použití podmíněného výrazu velmi vhodné. To znamená, že se dá sice obejít, ale za cenu „těžkopádnější“ jazykové konstrukce.

Syntaxe podmíněného výrazu je:

```
booleovský.výraz ? výraz1 : výraz2
```

a má podobný význam jako:

```
if (booleovský.výraz)
    výraz1;
else
    výraz2;
```

čili výsledná hodnota podmíněného výrazu je bud' `výraz1`, pokud `booleovský.výraz` bude `true`, v opačném případě bude `výraz2`. Ternární operátor se od `if-else` liší tím, že je to výraz – nikoli příkaz – a tudíž předává hodnotu.

Příklad 31:

```
int i, k, j = 2;
i = (j == 2) ? 1 : 3;    // i bude 1
k = (i > j) ? i : j;    // k bude maximum z i a j, tedy 2
```

Obecný omyl je, že ternární operátor může být použit jen v přiřazení, tak jako v předchozích případech. Není tomu tak, např. v dalším příkazu žádné přiřazení nenaleznete a přesto má smysl.

```
(i == 1) ? i++ : j++; // inkrementuje se buď i nebo j
```

Poznámka:

Toto byly školní příklady, ze kterých není užitečnost ternárního operátoru zřejmá. Ve všech třech příkladech byste pravděpodobně použili konstrukci `if-else` s naprosto stejným výsledkem, např. pro první příklad:

```
if (j == 2)
    i = 1;
else
    i = 3;
```

Příklad 32:

Výhoda ternárního operátoru je více patrná např. v případě, kdy chceme vypisovat čísla a pravidelně po deseti číslech odřádkovávat. Bez použití ternárního operátoru to s pomocí `if-else` není problém:

```
for (int i = 1; i <= 100; i++) {
    System.out.print(i);
    if (i % 10 == 0)
        System.out.print("\n");
    else
        System.out.print(" ");
}
```

Při použití ternárního operátoru bude ovšem program kratší.

```
for (int i = 1; i <= 100; i++)
    System.out.print(i + ((i % 10 == 0) ? "\n" : " "));
```

Každé desáté číslo bude následováno řetězcem "\n", všechna ostatní mezerou.

Dobrá rada:

Závorky kolem podmínky ($i \% 10 == 0$) nejsou nutné, ale uvádějí se pro zvýšení čitelnosti. □

5.3 Návěští

Návěští není samostatný příkaz, což znamená, že program nic neprovádí – není generován žádný kód. Návěští slouží pro označení místa, kam se lze přesunout pomocí příkazů **break** nebo **continue**.

Návěští je jakýkoliv identifikátor, který je zakončen dvojtečkou, např.:
sem_se_vrat:

Pozor:

Za návěstím může následovat pouze příkaz cyklu nebo blok začínající levou složenou závorkou. Chybou je tedy:

```
sem_se_vrat:
    i = 5;
```

Pokud toto návěští jen takto zapíšete a nebudete na něj skákat pomocí **break** nebo **continue**, překladač neohlásí chybu. Návěští totiž negeneruje žádný výkonný kód. Chyba při překladu nastane až při pokusu takto definované návěští využít. □

Poznámka pro programátora v C či C++:

Java nezná příkaz **goto**.² Příkazy **break** a **continue**, společně s aparátem pro zachycení výjimek, jej dostatečně a hlavně bezpečněji nahrazují. □

²I když je **goto** rezervované slovo.

5.4 Příkazy `break` a `continue`

Oba tyto příkazy lze použít ve všech typech cyklů a oba nějakým způsobem mění „normální“ průběh cyklu.

break ukončuje nejvnitřnější neuzavřenou smyčku – opouští ihned cyklus

continue skáče na konec nejvnitřnější neuzavřené smyčky (tj. přeskočí zbytek těla cyklu) a tím vynutí další iteraci smyčky – cyklus neopouští

Oba příkazy mají verzi s návěstím, které se zapisuje za ně. Pak oba příkazy nefungují jen v cyklu, ve kterém byly uvedeny, ale jejich účinek se přesouvá na cyklus, na jehož **začátku** se návěstí vyskytuje.

break ukončuje cyklus s návěstím

continue vynucuje další obrátku cyklu s návěstím

Poznámky:

- Příkaz `break` (na rozdíl od `continue`) ukončuje také právě prováděný blok, ve kterém nemusí být žádný cyklus, nebo příkaz `switch`.
- Příklady budou uvedeny u cyklů.

5.5 Iterační příkazy – cykly

Java umožňuje použít tři příkazy pro iteraci `while`, `for`, `do-while`. S těmito cykly jsou spojeny již uvedené příkazy `break` a `continue`, které mohou jejich provádění ovlivnit.

5.5.1 Příkaz `while`

Tento iterační příkaz testuje podmínu cyklu před průchodem cyklem – cyklus tedy nemusí proběhnout ani jednou.

Příkaz `while` používáme, závisí-li ukončovací podmínka na nějakém příkazu v těle cyklu³.

```
while (logickýVýraz)      // závorky jsou nutné
    příkaz;
```

³Předem tedy nedokážeme např. říci, proběhne-li cyklus jednou nebo stokrát.

Například:

```
while (x < 10)
    x++;
```

Poznámka:

Občas je výhodné použít nekonečnou smyčku: while (true)
která ale musí mít někde ve svém těle příkaz break nebo return. □

Poznámka pro programátora v C či C++:

Nekonečná smyčka pomocí while (1) není možná, protože while očekává boolean, nikoliv int. □

Příklad 33:

Program čte znaky z klávesnice, malá písmena opisuje na obrazovku, ostatních znaků si nevšímá⁴ a zastaví se po přečtení znaku z. Ke čtení znaku využívá metodu známou z [4.4/72].

```
public static void main(String[] args) {
    char c;
    while ((c = ctiznak()) != 'z') {
        if (c >= 'a')
            System.out.print(c);
    }
    System.out.println("\nCteni znaku bylo ukonceno.");
}
```

Příklad 34:

Tentýž příklad s ukázkou nekonečného cyklu while a s použitím příkazů break a continue.

```
public static void main(String[] args) {
    char c;
    while (true) {           // nekonecna smycka
        if ((c = ctiznak()) < 'a')
            continue;          // zahozeni velkych pismen atd.
        if (c == 'z')
            break;             // zastaveni po nacteni znaku 'z'
        System.out.print(c);   // tisk znaku
    }
    System.out.println("\nCteni znaku bylo ukonceno.");
}
```

⁴Je zde použito výrazné zdůvodnění – ve skutečnosti si program nevšímá pouze znaků, jejichž kód je menší než kód a.

Poznámka:

Tělo příkazu `while` může být prázdné, např. lze využít toto vynechávání mezer na vstupu:⁵

```
while (ctiZnak() == ' ')
```

□

Dobrá rada:

Je-li příkaz tvořící tělo cyklu prázdný, je středník ; odsazen vždy na nové řádce.

□

5.5.2 Příkaz do-while

V tomto cyklu se podmínka testuje až po průchodu cyklem – cyklus tedy proběhne nejméně jednou. Je to opět vhodný typ cyklu v případě, že předem neznáme, kolikrát má cyklus proběhnout.

```
do {
    příkazy;
} while (logickýVýraz);
```

Například:

```
do {
    i--;
} while (i > 0);
```

Pozor:

Stejně jako v C se cyklus opouští při nesplněné podmínce, což je podstatný rozdíl od Pascalu. Jinak řečeno – cyklus `do-while` pracuje tak dlouho, dokud má podmínka hodnotu `true`.

□

Dobrá rada:

Pište uzavírací složenou závorku na stejný řádek jako klíčové slovo `while`. Napíšete-li totiž `while` na novou řádku, budete se pak při prvním pohledu do programu divit, jaký smysl ten prázdný cyklus `while` má.

□

Příklad 35:

Program je podobný jako u cyklu `while`. Opět opisuje všechna malá písma zadaná z klávesnice a zastaví se po stisku znaku z . Rozdíl je v tom, že vypíše i tento ukončovací znak.

⁵ Existuje lepší způsob, který má podporu v knihovnách funkcí – viz [9.3.3/150].

```

public static void main(String[] args) {
    char c;
    do {
        if ((c = ctiZnak()) >= 'a')
            System.out.print(c);
    } while (c != 'z');

    System.out.println("\nCtení znaku bylo ukončeno.");
}

```

5.5.3 Příkaz for

Je to typický příkaz cyklu, který použijeme v případě, že předem známe omezující kriteria, tedy počáteční nastavení, ukončující podmítku a způsob ovlivnění řídící proměnné.⁶ V tomto případě není vhodný cyklus **while** – viz [5.5.1/80], protože ten nemá uvedené tři prvky tak přehledně na jednom místě jako **for**.

Příkaz **for** má následující syntaxi:

```

for (výraz_start; výraz_stop; výraz_iter)
    příkaz;

```

Například typický cyklus **for**:

```

for (int i = 1; i <= 10; i++)
    System.out.println(i);

```

Cyklus **for** probíhá tak, že se na počátku vyhodnotí **výraz_start**, otěstuje se, zda je **výraz_stop** pravdivý (**true**), provede se příkaz a nakonec se provede **výraz_iter**. Pak začíná další obrátka cyklu nyní již rovnou testováním **výraz_stop**.

Poznámka pro programátora v C či C++:

V Javě se velmi často deklaruje řídící proměnná cyklu přímo v hlavičce cyklu, tak jak je to uvedeno v předchozím případě. Je to dobrý zvyk („deklaruj data tam, kde je potřebuješ“), ale není to bezpodmínečně nutné. Naprosto stejně bude pracovat i tento úsek programu:

```

int i;
for (i = 1; i <= 10; i++)
    System.out.println(i);

```

⁶Například chceme, aby cyklus běžel od jedné do dvaceti.

Předchozí příklad ukazoval typické použití **for**. Můžeme se však setkat s mnohem komplikovanějším zápisem tohoto příkazu – viz např. příklady dále. Jsme-li pak na pochybách, co má příkaz **for** vlastně dělat, je dobré si uvědomit, že se dá většinou⁷ přepsat pomocí cyklu **while** takto:

```
výraz_start;
while (výraz_stop) {
    příkaz;
    výraz_iter;
}
```

a podle tohoto schématu také ve skutečnosti cyklus **for** pracuje.

Pozor:

- Výrazy: `výraz_start` `výraz_stop` `výraz_iter` spolu nemusí vůbec souviset, a dokonce ani nemusí být uvedeny. Ovšem použití takových zápisů není to nejšťastnější využití cyklu **for**.
- Pokud není některý z výrazů: `výraz_start`, `výraz_stop`, `výraz_iter` uveden, je nutné vždy uvést středník ;, kterým je tento chybějící výraz oddělen od ostatních – viz dále příklady.

Příklad 36:

Všechny příklady vždy tisknou čísla 0 až 9.

1. „Klasické“ (a doporučované) užití **for**.

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

2. Využití inicializace i při deklaraci – nevhodné řešení, protože není vše pohromadě v hlavičce **for**.

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

3. Řídící proměnná je měněna v těle cyklu – opět nevhodné řešení.

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

⁷Výjimkou je použití příkazu **continue** v těle cyklu.

4. V iterační části nemusí být žádný iterační příkaz. To je ale krajně nepřehledné řešení, které bez důvodu prohazuje výkonný příkaz cyklu (tj. tisk) s iteračním příkazem.

```
int i = 0;
for (; i < 10; System.out.println(i - 1))
    i++;
```

5. Tělo cyklu může být prázdné – opět nepřehledné řešení.

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

    ;
```

6. Využití nekonečného cyklu pomocí **for** a příkazu **break**. Opět zcela nevhodné řešení, na kterém je ale vidět nutnost použití složeného příkazu uzavřeného do (), protože cyklus **for** obsahuje více než jeden příkaz.

```
int i = 0;
for ( ; ; ) {
    if (i >= 10)
        break;
    System.out.println(i++);
}
```

Poznámka pro programátora v C či C++:

Java nezná **operátor čárka**, který v C umožňuje spojení více výrazů do výrazu jediného. Jedno z mála rozumných použití operátoru čárka v C bylo ale právě v cyklu **for**. Tato možnost naštěstí zůstala v Javě zachována – viz dále. □

V inicializační části a v iterační části cyklu **for** může být více výrazů oddělených čárkami. V testovací části nikoliv.

Příklad 37:

Příklady ukáží nevhodné i vhodné využití více výrazů oddělených čárkami. Opět se tisknou čísla od 0 do 9.

- Nevhodné přesunutí výkonného příkazu do iterační části.

```
for (int i = 0; i < 10; System.out.println(i), i++)
    ;
```

2. Vhodné využití vícenásobné inicializace. Pozor na skutečnost, že proměnné `i` a `sum` musí být deklarovány **vně cyklu `for`**. Pokud by byly obě deklarovány uvnitř, nastal by případ 5. – viz dále.

```
int i, sum;
for (i = 1, sum = 0; i <= 10; i++)
    sum += i;
```

3. Chybný pokus o vícenásobnou inicializaci. Proměnná `sum` je deklarována vně `for` a uvnitř `for` je pokus o její redeklaraci.

```
int sum;
// chyba při překladu
for (int i = 1, sum = 0; i <= 10; i++)
    sum += i;
```

4. Nevhodné přesunutí výkonného příkazu do iterační části.

```
int i, sum;
for (i = 1, sum = 0; i <= 10; sum += i, i++)
    ;
```

5. Možný, ale nevhodný pokus o deklaraci proměnné `sum` v těle `for`. Proměnná `sum` má pak oblast platnosti pouze v cyklu `for`, takže po opuštění cyklu s ní už nelze pracovat, tj. výpočet v cyklu byl vlastně zbytečný.

```
for (int i = 1, sum = 0; i <= 10; i++)
    sum += i;
// chyba při překladu, sum není deklarována
System.out.println(sum);
```

Poznámka:

V obecném podvědomí je mylná myšlenka, že v iterační části může být skutečně jen inkrementace (tj. zvětšení o jedničku). Není to pravda, jak už to ostatně bylo patrné z některých předchozích příkladů. □

Příklad 38:

Následující úsek programu ukazuje, jak lze při výpočtu součinu lichých čísel značně zrychlit činnost programu. Je to také ukázka, kdy klasický zápis⁸ cyklu `for` nemusí být optimální.

⁸Míněn zápis od nuly do nějakého maxima s přičítáním jedné.

Klasicky, ale nevhodně:

```
int i, soucin;
for (i = 1, soucin = 1; i <= 9; i++)
    if (i % 2 == 1) // dělení modulo pro zjištění lichosti
        soucin *= i;
```

Nestandardně, ale rychle.⁹

```
int i, soucin;
for (i = 3, soucin = 1; i <= 9; i += 2)
    soucin *= i;
```

Tak, jako pro cykly `while` a `do-while`, se i pro ovládání cyklu `for` mohou využít příkazy `break` a `continue`. Základní použití příkazu `break` (bez návěští) již bylo uvedeno výše.

Příklad 39:

Následující příklad ukáže, jak příkaz `break` s návěstím dokáže ukončit i blok.

```
public static void main(String[] args) {
    System.out.print("Zacatek ");
    odskok:
    {
        for (int i = 1; i < 10; i++) {
            if (i == 5)
                break odskok;
            // break;
            System.out.print(i + " ");
        }

        System.out.print("Stred ");
    }

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

Pro `break odskok;` vypíše:

Zacatek 1 2 3 4 Konec

Pro `break;` vypíše:

Zacatek 1 2 3 4 Stred Konec

⁹Při pokusu byl více než třikrát rychlejší.

Příklad 40:

Využití příkazu **continue** ve verzi s návěstím ve vnořených cyklech **for**.

navesti:

```
for (int n = 0; n < 4; n++) {
    for (int m = 0; m < 2; m++) {
        if (n == 2 && m == 1)
            continue navesti;
        System.out.print(n + "-" + m + " ");
    }
}
```

Vypíše:

0-0 0-1 1-0 1-1 2-0 3-0 3-1

Příklad 41:

Typické použití příkazu **break** s návěstím – výskok z vnořených (zahnízděných) cyklů.¹⁰

chyba:

```
{
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            for (int k = 0; k < 10; k++) {
                if (x[k] == 0)
                    break chyba;
                a[i] = a[i] + b[j] / x[k];
            }
        }
    }
    System.out.println("Dobre");
    // a např. tisk výsledků a return
}
System.out.println("Nulový delitel");
```

Dobrá rada:

Pro cyklus **for** platí několik obecných pravidel, které je vhodné dodržovat:

- Snažíme se mít vždy pouze jednu řídicí proměnnou.
- Řídicí proměnná cyklu má být ovlivňována pouze v řídicí části cyklu a ne v jeho těle.
- Všechny potřebné inicializace mají být v inicializační části.

¹⁰Podobného výsledku ale můžete dosáhnout i s využitím výjimky – viz řešení

Další dvě pravidla platí i pro ostatní cykly.

- Pokud má cyklus **for** nebo **while** prázdné tělo, má být středník ukončující tento příkaz odsazen na nové řádce.
- Je-li to možné, preferujte cykly **while** a **for** před cyklem **do-while**, protože jsou přehlednější.

□

5.6 Příkaz switch

Je to přepínač neboli příkaz pro mnohonásobné větvení programu.

Syntaxe příkazu **switch** je následující:

```
switch (výraz) {  
    case hodnota_1 : příkaz_1;  
        break;  
    . . .  
    case hodnota_n : příkaz_n;  
        break;  
    default : příkaz_def;  
        break;  
}
```

Vlastnosti příkazu **switch**:

- výraz, podle kterého se rozhoduje, musí být pouze typu **char**, **byte**, **short** nebo **int**; nesmí být typu **long**, **float** nebo **double**,
- počet větví (větev začíná klíčovým slovem **case**) není omezen,
- v každé větvi může být více příkazů, které není nutno uzavírat do složených závorek,
- lze použít větev **default**, která se provádí, když žádná z větví **case** nevyhovuje,
- nelze „jednoduše“ (tj. jako např. v Pascalu 'A'... 'Z') napsat prostý výčet několika hodnot pro jeden příkaz.

Poznámka:

Výčet několika hodnot, pro které má být proveden tentýž příkaz, musí být rozepsán pro **všechny** hodnoty, což jej v případě rozsáhlějšího výčtu činí nepoužitelným. Forma zápisu je:

```
case hodnota_1.1 :
case hodnota_1.2 :
    ...
case hodnota_1.m : příkaz;
break;
```

□

Pozor:

Jak je již možná jasné z předchozí poznámky, má **switch** tu vlastnost, že není-li větev přepínače (jeden nebo více příkazů) ukončena pomocí příkazu **break** (nebo **return**), program neopouští **switch**, ale zpracovává následující větev v pořadí! Nezáleží na tom, zda jsou za klíčovým slovem **case** uvedeny nějaké příkazy či nikoliv. V přechodu do dalších větví pokračuje do dosažení nejbližšího příkazu **break** (nebo **return**) nebo do ukončení příkazu **switch**.

□

Příklad 42:

Tento úsek programu vypíše znaky 123 po stisku klávesy a nebo b nebo c . Po stisku d vypíše 23 a po stisku jiné klávesy vypíše 3 .

```
switch (ctiZnak()) {
    case 'a' :
    case 'b' :
    case 'c' : System.out.print("1");
    case 'd' : System.out.print("2");
    default   : System.out.print("3");
}
```

Příklad 43:

Pro „normální“ (očekávanou) funkci přepínače je nutné program pooprat takto:

```
switch (ctiZnak()) {
    case 'a' :
    case 'b' :
    case 'c' :
        System.out.print("1");
        break;
```

```
case 'd' :  
    System.out.print("2");  
    break;  
  
default :  
    System.out.print("3");  
    break;  
}
```

Dobré rády:

- Související větve se neoddělují prázdnou řádkou, např.:

```
switch (ctiZnak()) {  
    case 'a' :  
    case 'b' :  
        System.out.print("1");  
        break;
```

- Příkazy větve jsou na nové řádce a odsazeny o obvyklý počet dvou mezer.
- Větev **default** se většinou píše, i když je prázdná, tedy:

```
default :  
    break;
```
- Příkaz **break** za posledním příkazem poslední větve není nutný, ale z konvence se píše.
- Větev **default** nemusí být uvedena jako poslední větev přepínače, ale z konvence je vždy jako poslední. Není-li však jako poslední, je její ukončující **break** nutný.

Poznámky:

- Příkaz **break** bez návěstí ruší vždy nejvnitřnější smyčku cyklu nebo ukončuje příkaz **switch**, ve kterém je vnořen. Je tedy nutné dávat zvýšený pozor, obsahuje-li **switch** nějaký cyklus nebo naopak nějaký cyklus obsahuje **switch**. Pokud tomu tak je, je výhodné použít příkaz **break** s návěstí, což způsobí odskok přesně tam, kam potřebujeme.
- Příkaz **continue** nemá s konstrukcí přepínače nic společného!

Příklad 44:

Příkazy za **default** se provádějí vždy až tehdy, když není nalezena žádná vyhovující větev, ať již je větev **default** uvedena v přepínači kdekoliv.

```
switch (ctiZnak()) {  
    default :  
        System.out.println("Nestiskl jsi ani '1' ani '2'");  
        break;  
  
    case '1' :  
        System.out.println("Stiskl jsi '1'");  
        break;  
  
    case '2' :  
        System.out.println("Stiskl jsi '2'");  
        break;  
}
```

□

Příklad 45:

Úsek programu čte znaky z klávesnice a opisuje je na obrazovku. Bílé znaky – mezeru a tabulátor – nahradí znakem # . Jakmile přečte znak *, skončí okamžitě svoji činnost.

Toto řešení je uvedeno jako odstrašující příklad, jak nemá program vypadat. Míchají se totiž do sebe věci, které nemají nic společného!

```
char c = 'a';  
while (c != '*') {  
    switch (c = ctiZnak()) {  
        case ' ' :  
        case '\t' :  
            System.out.print("#");  
            continue;  
  
        case '*' :  
            break;  
        default :  
            System.out.print(c);  
            break;  
    } // konec switch  
} // konec while
```

Program pracuje takto:

- Proměnná **c** byla inicializována znakem **a**, takže první průchod přes **while** určitě proběhne.
- Načtená mezera nebo tabulátor způsobí tisk znaku **#** a pak příkaz **continue**¹¹ způsobí přímý odskok na konec smyčky **while**, odkud je řízení předáno na její začátek a testována podmínka (**c != '*'**)
- Načtený znak ***** způsobí odskok na konec **switch**, kde nastává další obrátku cyklu **while**, která ale neproběhne, protože podmínka (**c != '*'**) bude mít hodnotu **false**.
- Každý jiný načtený znak je vypsán beze změny.

Příklad 46:

Mnohem lepší řešení předchozího příkladu – test ukončovacího znaku ***** se provede bez průtahů hned na začátku.

```
while ((c = ctiZnak()) != '*') {
    switch (c) {
        case ' ' :
        case '\t' :
            System.out.print('#');
            break;

        default:
            System.out.print(c);
            break;
    }
}
```

Příklad 47:

Kratší, ale méně přehledná úprava téhož programu. Je z ní vidět, že když ušetříme tři řádky, můžeme program dobře „zamlžit“.

¹¹ Který, jak víme, nemá s příkazem **switch** nic společného!

```
while ((c = ctiznak()) != '*') {
    switch (c) {
        case ' ' :
        case '\t' :
            c = '#';
        default:
            System.out.print(c);
            break;
    }
}
```

5.7 Příkaz return

Dojde-li provádění programu na příkaz `return`, ukončí se provádění metody (funkce), která tento příkaz obsahuje. V metodě `main()` ukončí příkaz `return` celý program.

Často se pomocí **return** vrací nějaká hodnota, jejíž typ záleží na typu metody – podrobné vysvětlení a příklady viz [6.1/98].

Poznámka:

Občas se můžeme setkat s použitím metody `System.exit()`¹². Ta má podobný význam jako příkaz `return`. Rozdíl je v tom, že je-li vyvolána z kterékoliv metody, ukončí bezprostředně program, bez návratu do funkce volající.

Běžné chyby:

- if ($x == 1$) then then není klíčové slovo
 - if $x == 1$ chybí závorky
 - if ($x == 1$)
 $y = x$ chybí středník
else
 $x++;$
 - if ($x = 1$) přiřazení (=) místo porovnání (==)
 - while ($x == 1$) do za while není do

¹²Je popsána v balíku `java.lang.System` a v [19.3.5/308]

- `for (int i = 0; i < 10; i++)`; zde nemá být středník `x += i;`
- Používání reálných čísel jako řídicích proměnných (zejména `for`) cyklů apod, tj. místo celých čísel. Reálná čísla jsou většinou v počítači reprezentována více či méně přesnou approximací, takže cyklus pak nemusí fungovat správně.
- Stejně tak neporovnávejte dvě reálná čísla pomocí `==` nebo `!=` ale použijte `>=` nebo `<=`

Cvičení:

1. Napište program, který přečte dvě celá čísla a pomocí ternárního operátoru vytiskne např.:
Mensi cislo: 5
Vetsi cislo: 8

2. Napište program, který přečte dva znaky v rozsahu 0 – 9 a A – F. Tyto dva znaky pak považujte za jedno hexadecimální číslo a jeho hodnotu vypište dekadicky.

3. Přečtete znak, a není-li to ani písmeno, ani číslice, pak vytiskněte nápis:
Interpunkci znak

V ostatních případech vytiskněte nápis:
Alfanumericky znak

Použijte jen jednoho příkazu `if-else` a nezapomeňte na malá písmena.

4. Napište program, který načte délku a šířku obdélníka a podle těchto údajů tento obdélník vykreslí, např.:

* * * * *

5. Napište program, který čte celá čísla tak dlouho, dokud nezadáte nulu. Z přečtených čísel vypíše to s největší hodnotou. Použijte cyklu `while`.

6 Metody

Program v Javě obsahuje jednu nebo více deklarací **metod** (tak se v OOP nazývají podprogramy neboli – ne úplně správně – funkce), z nichž jedna se musí vždy jmenovat `main()`.

Metody v Javě ale mají tu zvláštnost¹, že se dělí na **metody třídy** (též označované jako **statické metody**) a **metody instance**. Podrobně budou tyto podskupiny probrány v [8.11/134] a [8.4/122]. V této kapitole se budeme věnovat pouze společným znakům obou podskupin metod, abychom se při výkladu objektů nemuseli zabývat věcmi, jako jsou počty a typy parametrů, návratové hodnoty apod.

Poznámky:

- Tyto společné znaky budou vysvětlovány na **statických** metodách (mají na svém začátku klíčové slovo **static**), aby bylo možné ověřit popisované skutečnosti bez nutnosti použití objektů.
- Abychom odlišili jméno proměnné od jména metody, budeme stále důsledně používat již dříve zmíněnou konvenci, že za jménem metody budou následovat kulaté závorky, tedy `ahoj()` označuje metodu a `ahoj` označuje proměnnou.

Zpracování programu začíná voláním metody `main()` a končí opuštěním této metody.

Na rozdíl od jazyků typu Pascal nemohou být metody vhnízděné – jedna metoda nemůže obsahovat ve svém těle deklaraci druhé metody. Z toho vyplývá, že formální parametry a lokální proměnné jsou tedy přístupné pouze v metodě, v níž byly deklarovány, a jsou skryté z vnějšku této metody.

Dobrá rada:

Ačkoliv překladač nečiní žádné omezení co do názvu metody ani co do

¹ Z hlediska OOP to žádná zvláštnost není.

délky jejího kódu, je vhodné psát takové metody, které provádějí jednu a pouze jednu jasně specifikovanou činnost. Délka těla metody by neměla přesáhnout cca 20 rádek. Jméno metody by mělo být krátké, výstižné a v celém programu tvořené podle jednotného schématu, např. `otevřiSoubor()`, `zavřiSoubor()`, `otevřiOkno()`, `zavřiOkno()`, atd. Pokud nejsme schopni přiřadit metodě „rozumné“ krátké jméno, je to pravděpodobně tím, že se snažíme, aby prováděla více než jednu činnost. To svědčí o špatné dekompozici problému. □

6.1 Deklarace metody

Deklarace metody zahrnuje jak hlavičku metody, tj. jméno metody, typ návratové hodnoty a případně i typy a jména jejích **formálních parametrů**, tak i její tělo, ve kterém je uložen výkonný kód metody.

Příklad 48:

Příklad metody, která vrátí větší ze svých dvou parametrů:

```
static int max(int a, int b) {
    if (a > b)
        return (a);
    else
        return (b);
}
```

Dobrá rada:

Hlavička metody není ukončena středníkem a konvence říká, že levá složená závorka je na téže řádce a je oddělena jednou mezerou. □

Poznámka pro programátora v C či C++:

Typ metody (její návratová hodnota) nemůže být vynechán. V tom případě ohlásí kompilátor chybu:

```
Invalid method declaration: return type required
```

V Javě nenastávají problémy s nutností uvádět funkční prototypy, aby překladač věděl, jakého návratového typu je metoda. V Javě je lhostejně, zda deklarace metody leží před nebo za místem volání metody. To bylo sice možné i v C, ale vyžadovalo to dodatečné úsilí v podobě funkčních prototypů. □

Tělo metody (program) je uzavřeno do složených závorek { a }, naprosto stejně jako u metody `main()`, a může obsahovat jak příkazy, tak i deklarace proměnných.

Výstupní hodnota metody se předává příkazem:

`return (výraz);` nebo `return výraz;`

který vypočte hodnotu výrazu `výraz`, přiřadí ji jako návratovou hodnotu metody a tuto metodu ukončí.

Statické metody uvnitř jedné třídy² jsou volány použitím běžné konvence, např.:

```
x = max(10 * i, j - 15);
```

6.2 Metoda bez parametrů

Metoda, která nemá žádné parametry, musí být deklarována i volána včetně obou kulatých závorek. Například metoda, která přečte dvě celá čísla z klávesnice³ a vrátí jejich součet:

```
static int secti() {
    int a, b;
    a = ctiInt();
    b = ctiInt();
    return (a + b);
}
```

je volána: `j = secti();`

Poznámka pro programátora v C či C++:

Zapomenete-li uvést prázdné kulaté závorky, např.: `j = secti;` vypíše překladač chybu: variable secti not found in class ... což je velmi příjemná pomoc, protože tato chyba patřila v C k značně základním.

Není možné uvést místo neexistujících **formálních parametrů** klíčové slovo **void**, např.: `static int secti(void) {`

Překladač hlásí chybu: <identifier> expected

□

²Více tříd jsme zatím neprobírali.

³Použitá metoda `ctiInt()` byla popsána v [4.3/71].

6.3 Metoda bez návratového typu – procedura

Striktně řečeno, metodu bez návratového typu (což se v jiných jazycích označuje jako procedura) v Javě nelze napsat, protože každá metoda musí mít návratový typ uveden. Pro tyto případy je však možné použít návratový typ **void** (tj. prázdný). Metoda se pak deklaruje, jako metoda vracející typ **void**, např.:

```
static void tiskPenez(int koruny) {
    System.out.println("Cena: " + koruny + " Kc");
}
```

volání je pak: `tiskPenez(a + b);`

nebo: `tiskPenez(10);`

nebo: `tiskPenez(a);`

Poznámky:

- Příkaz **return** u procedur není nutný. Pokud není uveden, nahrazuje ho uzavírací závorka `)` metody.
- Příkaz **return** se pak používá pouze pro nucené ukončení metody před dosažením jejího konce, např. po nějaké podmínce.

Jako proceduru lze použít i metodu, která vrací jiný návratový typ než je typ **void**. Volaná metoda návratovou hodnotu sice vrací, ale volající vrácenou hodnotu nevyužije. Tato možnost se ale příliš nepoužívá.

Poznámka pro programátora v C či C++:

Explicitní přetypování na typ **void** v tomto případě nelze provést. U příkazu:

`(void) secti();`

překladač ohláší hned dvě chyby:

- 1) not an expression statement
- 2) 'void' type not allowed here



6.3.1 Procedura bez parametrů

Je nejjednodušším typem metody a používá se typicky zejména pro nejrůznější tisky. Vypadá takto:

```
static void tisk() {
    System.out.println("ahoj");
}
```

a volá se: `tisk();`

Poznámka pro programátora v C či C++:

V objektově orientovaném programovacím jazyce se setkáváme s metodami bez parametrů mnohem častěji než v procedurálním jazyce – viz též [6.4/101]. □

6.4 Metoda s více parametry různých typů

Potřebuje-li metoda větší množství formálních parametrů, vypisují se jednotlivě včetně svých typů a oddělují se čárkami. V deklaraci metody si můžeme zvolit pořadí parametrů libovolně⁴, při volání pak toto pořadí musíme dodržet. Například:

```
static double secti(int a, double b) {
    return a + b;
}
```

je totéž⁴, co:

```
static double secti(double b, int a) {
    return a + b;
}
```

Pozor:

Tento způsob je intuitivní u různého typu parametrů. Pokud jsou ale parametry stejného typu, snadno se napíše tato chyba:

```
static int secti(int a, b) { // chyba
správně je:
```

```
static int secti(int a, int b) {
```

□

Dobrá rada:

Ačkoliv počet parametrů není nijak omezen, je vhodné nepřesáhnout počet pěti parametrů. Při větším počtu je metoda nepřehledná a svědčí to o špatné dekompozici problému – pravděpodobně se snažíte v metodě udělat více, než jen jeden úkol.

V objektově orientovaném jazyce metody nemusejí mít (obecně) moc formálních parametrů. Je to způsobeno jednak tím, že metody obvykle pracují nad proměnnými deklarovanými ve stejné třídě, které pak není nutné do téhoto metod předávat pomocí parametrů. Druhým důvodem je, že lze jako formální parametr uvést odkaz na jiný objekt. Pak předáváme skutečný parametr jako referenci na třídu – viz [10.5.2/173]. □

⁴Ne tak úplně – viz přetižení [6.8/104].

6.5 Rekurzivní metody

Java umožňuje vytvářet rekurzivní metody. Hlavička těchto metod se nijak neliší od hlavičky metody nerekurzivní. Rozdíl je pouze v těle metody, kde rekurzivní metoda volá sama sebe.

Příklad 49:

Školní příklad je metoda pro výpočet faktoriálu.

```
public class Faktor {
    public static void main(String[] args) {
        System.out.println("20! = " + fakt(20));
    }

    public static long fakt(long n) {
        if (n > 1)
            return n * fakt(n - 1);
        else
            return 1;
    }
}
```

Poznámka:

Rekurzivní metody používáme jen tehdy, vede-li na ně algoritmus, který nelze snadno obejít. Například algoritmus faktoriálu je sice rekurzivní, ale lze jej bez problémů obejít pomocí jednoduchého cyklu **for**. Ten je přehlednější a také rychlejší. Naopak prohledávání adresářů do hloubky je vhodným kandidátem na rekurzivní funkci, protože pracuje nad rekurzivně uspořádanými daty. □

Příklad 50:

Ještě více odstrašujícím příkladem, než je výpočet faktoriálu rekurzí, je rekurzivní výpočet Fibonacciho řady (0, 1, 1, 2, 3, 5, 8, ...), kdy každý další člen je pouze součtem dvou členů předchozích. Použijete-li rekurzi, získáte neuvěřitelně pomalý program, protože pro každý člen posloupnosti je nutné volat dvakrát celý rekurzní výpočet.

```
public static long fib(long n) {
    if (n == 0 || n == 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Při pokusech byl 17. člen řady vypočítán za 10 msec, 20. za 40 msec, 25. za 480 msec a 30. již za 5 197 msec. Pro výpočet 17. člena se bude metoda `fib()` volat 5 167krát a přitom se bude 987krát „počítat“ hodnota `fib(0)` a 1597krát hodnota `fib(1)`.

6.6 Konverze skutečných parametrů a návratové hodnoty metody

Pokud není typ návratové hodnoty shodný s návratovým typem metody a neprovádí se implicitně rozšiřující typová konverze, je nutné provést explicitní zužující typovou konverzi.⁵

Totéž platí i pro ***skutečné parametry*** metody – to jsou ty, s nimiž je metoda volána, pokud jejich typy nesouhlasí s ***formálními parametry*** – s těmito byla metoda deklarována.

Příklad 51:

Použití explicitních typových konverzí.

```
static int konv1(double d) {
    return (int) d;
}

static double konv2(int d) {
    return d;
}
public static void main(String[] args) {
    int k = konv1(4);
    double j = konv2((int) 4.5);
}
```

V příkladu jsou všechny explicitní konverze nutné, jinak dojde k chybě při překladu.

Poznámka:

Java provádí důslednou typovou kontrolu skutečných parametrů a současně kontroluje i počet parametrů. □

⁵Vysvětlení obou typů konverzí je v [3.7/46].

6.7 Způsoby předávání skutečných parametrů metod

Java umožňuje pouze jeden způsob **předávání parametrů** a to **hodnotou** (*call-by-value*). To znamená, že skutečné parametry mohou být v metodě pouze čteny. Lze je sice pomocí přiřazovacího příkazu i změnit, ale pouze dočasně, protože se ve skutečnosti mění jen jejich lokální kopie. Jinak řečeno – jakákoli změna parametrů uvnitř metody je sice možná, ale je pouze dočasná, a po opuštění metody se ztrácí.

Příklad 52:

```
static int zmena(int i) {
    i++;
    return i;
}

public static void main(String[] args) {
    int j, k = 4;
    j = zmena(k);
    System.out.println("k = " + k + ", j = " + j);
}
```

Vypíše:

k = 4, j = 5

Poznámka pro programátora v C či C++:

Možnost **volání odkazem** Java neřeší pomocí ukazatelů (*pointers*) jako je tomu u jazyka C. Problém je řešen na jiné „filosofické úrovni“ a to využitím prostředků objektově orientovaného programování – viz v [10.5/171].

Stejně tak Java neumožňuje využívat metod s proměnným počtem parametrů. Tento handicap se řeší pomocí přetížení metod – viz dále.



6.8 Přetížené metody

Přetížené metody (*overloaded*) jsou metody, které mají stejná jména, ale různé hlavičky. To, že je metoda přetížená, znamená, že se její formální parametry **musí lišit počtem** nebo **typem** nebo **pořadím**, případně i kombinacemi těchto způsobů.

Pozor:

Metodu nelze přetížit pouhou změnou typu návratové hodnoty – je nutné to zařídit minimálně jedním ze tří výše uvedených rozlišení. V opačném případě hlásí překladač chybu:

duplicate definition □

Kdykoliv je přetížená metoda volána, komplilátor vybere tu z metod, která přesně vyhovuje počtu, typům a pořadí skutečných parametrů.

K čemu se takové metody používají? Většinou se jedná o několik funkčně příbuzných metod, které se liší parametry. Tím umožňují provádět stejnou základní operaci, ale s různými datovými typy. Uživatel metody se pak nemusí obtěžovat např. přetypováním skutečných parametrů na typ formálních parametrů nebo zadáváním pomocných skutečných parametrů.

Přetížených metod se často využívá, např. známá metoda `print()` je v Java Core API přetížená celkem devětkrát – pro všechny základní datové typy a navíc pro typ `Object`.

Příklad 53:

Ukázka trojnásobně přetížené metody `ctverec()`, kdy se rozlišuje podle typu formálního parametru. Na jeho jménu – zde vždy `i` – naprosto nezáleží.

```
static int ctverec(int i) { return i * i; }

static double ctverec(double i) { return i * i; }

// static long ctverec(int i) { // chyba
static long ctverec(long i) {
    return i * i;
}

public static void main(String[] args) {
    int j = ctverec(5);
    double d = ctverec(5.5);
    long l = ctverec(12345L);

    System.out.println("j = " + j + ", d = " + d + ", l = " + l);
}
```

Vypíše:

j = 25, d = 30.25, l = 152399025

Příklad 54:

Jiné časté použití přetížených metod je přidání dalšího (upřesňujícího) formálního parametru.

```
static void tiskPenez(int koruny) {
    System.out.println("Cena: " + koruny + ",-- Kč");
}

static void tiskPenez(int koruny, int halere) {
    System.out.println("Cena: " + koruny + "," + halere + " Kč");
}

public static void main(String[] args) {
    tiskPenez(12);
    tiskPenez(12, 50);
}
```

Vypíše:

Cena: 12,-- Kč

Cena: 12,50 Kč

Poznámka:

Metody `sekti()` uvedené v [6.4/101] jsou tedy přetížené. □

6.9 Místa deklarací metod

Překladač Javy je velice benevolentní v umístění deklarací metod. Je jedno, zda je deklarace volané metody před místem volání nebo za ním. Jediné omezení je v tom, že každá metoda musí patřit do nějaké třídy (podrobně viz [8/119]).

Poznámka:

Jak již víme z [2.6.1/25], podle jména této třídy se musí jmenovat i soubor, ve kterém je uložena. □

6.10 Proměnné z pohledu místa deklarace

6.10.1 Nelokální proměnné – „globální“ proměnné

Striktně řečeno, v Javě žádné *globální proměnné* neexistují, protože každá proměnná musí někomu (tj. třídě) patřit. Pokud je ale proměnná deklarována vně jakékoliv metody, je přístupná všem metodám, které jsou deklarovány v téže třídě. V procedurálním programovacím jazyce bychom se na tyto proměnné dívali jako na globální. V Javě to však není správná terminologie!

Poznámka:

Lze dokonce nejdříve deklarovat metody a teprve potom deklarovat proměnné, které budou přesto viditelné v celé třídě, protože třída představuje jeden *prostor jmen* (též označovaný jako *oblast viditelnosti*), ve kterém na pořadí deklarací metod a proměnných nezáleží. Není to ovšem moc rozumný, přehledný a používaný způsob zápisu.

□

Poznámka pro programátora v C či C++:

V jazyce C je proměnná viditelná od místa své deklarace/definice, což je významný rozdíl.

□

Příklad 55:

Ukázka, že proměnná i metoda může být ve třídě deklarována na libovolném místě.

```
public class Metody {  
    static void tiskni1() {  
        System.out.println(i);  
    }  
  
    public static void main(String[] args) {  
        tiskni1();  
        tiskni2();  
    }  
    static int i = 5;  
  
    static void tiskni2() {  
        System.out.println(i);  
    }  
}
```

Pozor:

Existuje (a často bude dále používán) pojem **proměnná třídy** (též **statická proměnná**) jako protiklad **proměnná instance**. Obojí jsou nelokální proměnné a liší se od sebe jen tím, že proměnná třídy (stejně jako metoda třídy) je uvozena klíčovým slovem **static**, kdežto proměnná instance ne. Všechny zde uváděné proměnné jsou z tohoto pohledu proměnné třídy. Podrobné vysvětlení dalších rozdílů bude v [8.10/131] a [8.2/121].

Ve statické metodě je nutné použít výhradně jen statické proměnné. Statické metody totiž nemohou přímo využívat proměnné instance. □

Proměnné třídy (i instance) jsou implicitně inicializovány na nulu, tj. proměnné typu **int** mají hodnotu **0**, **float** hodnotu **0.0**, **char** hodnotu **'\u0000'**, **boolean** hodnotu **false**, atd. Je však dobrým zvykem nespoléhat se na tuto službu a u všech proměnných, které mají být inicializovány, tuto inicializaci výslovně uvést.

6.10.2 Proměnné metod – lokální proměnné

Jsou to všechny proměnné deklarované kdekoli uvnitř metody. Tyto proměnné jsou viditelné pouze v metodě, v níž jsou deklarovány. Pokud jsou deklarovány v bloku, pak se jejich viditelnost omezuje pouze na tento blok. To platí i pro proměnné deklarované ve funkci **main()**, které jsou také lokální pouze pro tu funkci – nejsou viditelné nikde jinde.

Rozsah platnosti lokálních proměnných je **od místa deklarace do konce metody** (případně do konce bloku), ve které jsou deklarovány.

Poznámka:

Na rozdíl od termínu „**globální proměnná**“, který je v Javě chybný, je termín **lokální proměnná** správný. Proměnné deklarované ve třídě tedy dělíme na proměnné třídy, proměnné instance a lokální proměnné. □

Příklad 56:

V bloku nelze deklarovat proměnnou stejného jména (na typu nezáleží) jako je již deklarovaná lokální proměnná, např.:

```
static void tiskni() {
```

```

int i = 6;
System.out.println(i);
{
//    int i = 7;           // chyba - dvojnásobná deklarace
//    long i = 7;          // chyba - dvojnásobná deklarace
    int j = 8;
    System.out.println(j);
}
//    System.out.println(j); // chyba - j není viditelná
}

```

Příklad 57:

Pravidlo uvedené v předchozím příkladu občas nechtěně snadno porušíme, např.:

```

static void tiskni2() {
    int i = 6;
    System.out.println(i);
//    for (int i = 1; i < 5; i++) // chyba
    System.out.println(i);
}

```

kdy deklarace v cyklu **for** je považována za deklaraci v témže bloku a překlad opět skončí chybou.

Poznámka:

Lokální proměnné nejsou automaticky inicializovány. Při pokusu o použití neinicializované lokální proměnné však překladač hlásí:
variable i might not have been initialized

Pozor:

U lokálních proměnných nelze použít klíčové slovo **static** ani **public**. Lokální proměnou lze označit jako **final** – viz [3.5.1/43] – ale tento způsob se příliš nepoužívá.

6.10.3 Zastínění nelokálních proměnných lokálními

Má-li proměnná třídy (či instance – [8.6/128]) stejné jméno (na typu nezáleží) jako lokální proměnná, pak ji lokální proměnná ve své metodě zastiňuje.

Existuje ale možnost, jak k této proměnné třídy přistoupit, a to pomocí **plně kvalifikovaného jména** (*fully qualified name*), které se skládá ze jména třídy a jména proměnné oddělených tečkou.

Příklad 58:

Využití plně kvalifikovaného jména.

```
public class Metody {
    static int i = 5;

    static void tiskni() {
        int i = 6;
        System.out.println(i);           // tiskne 6
        System.out.println(Metody.i);    // tiskne 5
    }

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

Dobré rady:

- Vyhněte se používání rekurzí tam, kde to není nezbytně nutné.

Běžné chyby:

- Má-li metoda více formálních parametrů stejného typu, musí být každý deklarován zvlášť.
chybně: int secti(int a, b)
správně: int secti(int a, int b)

Cvičení:

1. Napište statickou metodu void power(double x, int n), která vypíše tabulku mocnin x od 1 do n.
2. Napište statickou metodu boolean jePrvocislo(int i), která zjistí, zda je její parametr prvočíslo.
3. Napište rekurzivní statickou metodu double mocnina(double x, int n), která vypočítá x^n pomocí vztahu $x^n = x \times x^{n-1}$
4. Modifikujte program z kapitoly 5, který počítal e. Pro výpočet e vytvořte dvě přetížené metody. První bude double vypoctiE() a bude počítat e s přesností 10^{-7} . Druhá bude double vypoctiE(double presnost) a bude e počítat se zadanou přesností.

7 Pole

7.1 Pojem referenční datový typ

Java obsahuje dva neprimitivní datové typy a to pole a objekty. Proměnné těchto typů jsou označovány jako **referenční**.

Poznámka:

V literatuře se občas pod pojmem **referenční proměnná** míni pouze proměnná deklarovaná s datovým typem třídy a referenční proměnná typu pole se označuje jako **pole** případně **jméno pole** – tato konvence bude používána i v této knize. Uvedená dvojznačnost však nepůsobí prakticky žádné potíže. □

Referenční proměnné i jména polí se využívají podobně jako ukazatele v jiných programovacích jazycích. Není však možné považovat hodnotu referenční proměnné za adresu do paměti¹, což znamená, že není možné provádět jakékoli operace založené na přístupu do paměti pomocí známé adresy (tj. ukazatele).

Referenční proměnná ani jméno pole po své deklaraci nereprezentují žádná data, tj. objekt či prvky pole. Jak pole, tak i objekty vznikají dynamicky pomocí speciálního příkazu a zanikají, jakmile na ně neexistuje odkaz, tzn. že žádná referenční proměnná v sobě neuchovává odkaz na zanikající objekt. Podrobně viz [8.13/140].

Poznámky:

- Pod pojmem **odkaz** (*reference*) je míněna konkrétní hodnota referenční proměnné.
- Hodnota neplatného (neexistujícího) odkazu je hodnota konstanty **null**.²

¹Její hodnotu v „rozumné“ číselné podobě ani neumíme získat.

²**null** je klíčové slovo, proto je malými písmeny, i když konstanty se obecně píší velkými písmeny.

- Objekty a jejich referenční proměnné jsou vysvětleny v [8.2/121]. Pole objektů viz [10.4/169]. Pro pole znaků používá Java objekt typu **String** – viz [9/145].
- Způsoby předávání polí jako skutečných parametrů metod budou popsány v [10.5.3/175].

7.2 Deklarace pole

Skládá se ze dvou částí – z typu pole a z jeho jména. Kupodivu se při deklaraci neudává velikost pole. Důvod je jednoduchý – všechna pole jsou alokována dynamicky, takže velikost pole se určuje až při žádosti o jeho vytvoření.

Příklad deklarace referenční proměnné pole typu **int**:

```
int[] poleInt;
```

Poznámka pro programátora v C či C++:

Je možná deklarace i ve stylu jazyka C (tj. závorky za identifikátorem), tedy: `int poleInt[];` ale nedoporučuje se používat, protože po: `int[] pole1, pole2;` jsou `pole1` i `pole2` správné referenční proměnné pro pole typu **int**. □

Protože deklarace nepřiděluje paměť pro pole, je nutné před prvním použitím pole tuto paměť přidělit pomocí operátoru **new**, např.:

```
poleInt = new int[20];
```

Přidělení paměti pomocí **new** se často provádí³ již na řádku deklarace referenční proměnné jako její inicializace, např.:

```
int[] poleInt = new int[20];
```

Od této chvíle je pole `poleInt` možné používat. Kompilátor zaručuje, že jeho prvky budou mít nulové hodnoty (pokud je to pole typu **boolean**, pak hodnoty **false**).

Poznámka:

Pokud se pokusíme pracovat s polem, které nemá pomocí operátoru **new** přidělenu paměť, dostaneme chybové hlášení:

```
Variable poleInt might not have been initialized
```

□

³Ale není to podmínkou, jak bylo vidět v předchozím příkazu.

7.3 Délka pole

Délku (tj. velikost či počet prvků) pole zadanou při jeho vytvoření si nemusíme pamatovat, protože ji můžeme kdykoliv zjistit pomocí **členské proměnné** (viz [8/119]) se jménem `length`. Tuto konstantu vlastní automaticky jakékoli pole.

Například:

```
System.out.println("Pocet prvku pole: " + poleInt.length);
```

Vypíše:

```
Pocet prvku pole: 20
```

Pozor:

Proměnná (nebo lépe řečeno konstanta) `length` je přístupná pouze pro čtení a jakékoli pokusy o její změnu končí chybovým hlášením: `Variable length is declared final; cannot be assigned` □

K prvkům pole se přistupuje pomocí běžné konvence, kdy je index uveden v hranatých závorkách. Počáteční prvek pole má vždy index `[0]` a poslední prvek má index `[jmenoPole.length - 1]`.

Je asi zřejmé, že indexem pole může být jakýkoliv výraz, tedy i prvek jiného pole, případně metoda vracející hodnotu typu `int`, např.:

```
a[i + 1] = 5;
a[b[5]]++;
a[ctiInt()] = 8;
```

Poznámka pro programátora v C či C++:

Na rozdíl od jazyka C je za běhu programu striktně kontrolováno překročení obou mezí pole. Program pak reaguje vygenerováním výjimky `ArrayIndexOutOfBoundsException` (podrobně viz [16/241]). □

Příklad 59:

Typická práce s polem:

```
int[] poleInt = new int[20];
for (int i = 0; i < poleInt.length; i++) {
    poleInt[i] = i + 1;
    System.out.print(poleInt[i] + " ");
}
```

Dobrá rada:

Při průchodu polem v cyklu používejte jako hornímez konstantu `konkretniPole.length`. To vám zaručí, že nikdy nepřekročíte meze tohoto konkrétního pole, ani při jeho případné dynamické změně. Tomuto způsobu dávejte přednost před použitím konstanty. Příklad ukazuje toto nevhodné použití.

```
int[] poleInt = new int[MAX];           // MAX vhodné
for (int i = 0; i < MAX; i++) {        // MAX nevhodné
    poleInt[i] = i + 1;
    System.out.print(poleInt[i] + " ");
}
```

7.4 Inicializované pole

Pole nemusíme vytvářet jen pomocí `new`, byť je to asi nejčastější způsob. Pokud potřebujeme mít pole inicializované určitými hodnotami, jistě oceňme možnost vytvořit pole pomocí statického inicializátoru, např.:

```
int[] prvocisla = {1, 2, 3, 5, 7, 11};
```

Pole `prvocisla` bude opět dynamicky vytvořené, opět bude mít k dispozici konstantu `length`, která bude v tomto případě mít hodnotu 6. Práce s tímto polem je naprostě stejná jako s polem vytvořeným pomocí `new`, to např. znamená, že lze libovolně měnit prvky pole. Častý omyl totiž je, že u pole vzniklého pomocí statického inicializátoru jsou prvky pole konstantní.

Příklad 60:

Prvky inicializovaného pole nejsou konstanty.

```
for (int i = 0; i < prvocisla.length; i++) {
    System.out.print(prvocisla[i] + " -> ");
    prvocisla[i] = i + 1;
    System.out.print(prvocisla[i] + " ");
}
```

Vypíše:

```
1 -> 1 2 -> 2 3 -> 3 5 -> 4 7 -> 5 11 -> 6
```

7.5 Dvourozměrná pole

Jsou to pole polí a pracuje se s nimi obdobně jako s jednorozměrnými polí, jen je nutné při deklaraci zadat všechny rozměry.

Příklad 61:

Práce s dvouozměrným polem.

```
int[][] a = new int[5][4];
System.out.println("Pocet radek pole: " + a.length);
System.out.println("Pocet sloupca pole: " + a[0].length);
for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a[i].length; j++) {
        a[i][j] = i * 10 + j;
        System.out.print(a[i][j] + " ");
    }
    System.out.println();
}
```

Poznámka:

Všimněte si, jak se získá hodnota počtu sloupců – `a[i].length`. To znamená, že nemusí být pro všechny řádky stejná – viz další příklad.⁴

□

Protože jsou vícerozměrná pole organizována jako pole polí, není nutné alokovat celou paměť najednou a dokonce ani pole nemusí mít pro všechny řádky stejné počty sloupců.

Příklad 62:

Dvouozměrné pole s proměnnou délkou řádek.

```
int[][] a = new int[4][];
for (int i = 0; i < a.length; i++) {
    a[i] = new int[i + 1];
    for (int j = 0; j < a[i].length; j++) {
        a[i][j] = i * 10 + j;
        System.out.print(a[i][j] + " ");
    }
    System.out.println();
}
```

Vypíše:

```
0
10 11
20 21 22
30 31 32 33
```

⁴Takže to vlastně nejsou dvouozměrná pole tak, jak je známe z Pascalu nebo z C.

7.5.1 Inicializace dvourozměrného pole

I vícerozměrné pole lze vytvořit pomocí statického inicializátoru. Nejčastěji se vytváří „obdélníkové“ pole, ale není problém vytvořit i pole s různým počtem sloupců (tj. s proměnnou délkou řádek).

Příklad 63:

Inicializace obdélníkového pole a pole s proměnnou délkou řádek.

```
int[][] b = {{ 1, 2, 3},
              {11, 12, 13},
              {21, 22, 23}};
```



```
int[][] c = {{ 1, 2, 3},
              {11, 12},
              {21}};
```

7.6 Trojrozměrná pole

Trojrozměrná a vícerozměrná pole se vytvářejí analogicky. V případě, že se pole vytváří po částech, je nezbytné pamatovat si, že nelze „přeskakovat rozměry“, např.:

```
int[][][] d = new int[5][5][5]; // klasicky
```



```
int[][][] e = new int[5][5][];
e[0][1] = new int[8];           // správně
```



```
int[][][] f = new int[5][][5]; // chyba
f[0][][1] = new int[8];       // chyba
```

Poznámka:

„Přeskakování rozměrů“ znamená, že není možné vytvořit pomocí operátora `new` sloupce (tj. třetí index), když ještě nemáme vytvořeny řádky (druhý index). □

7.7 Více rozměrů v jednorozměrném poli

Tento požadavek je častý v grafických aplikacích, kdy potřebujeme do jednorozměrného pole (jednoho spojitého úseku paměti) přistupovat přes indexy řádků a sloupců. V tomto případě není vhodné dvourozměrné pole,

protože není zaručeno, že bude alokována spojité část paměti, případně že nebude obsahovat něco navíc kromě dat (např. konstanty `length` řádků).

Příklad 64:

Dvourozměrné pole v jedné oblasti paměti.

```
final int RADKY = 24;
final int SLOUPCE = 80;
byte[] obrazovka = new byte[RADKY * SLOUPCE];
for (int i = 0; i < RADKY; i++) {
    for (int j = 0; j < SLOUPCE; j++) {
        obrazovka[i * SLOUPCE + j] = 0;
    }
}
```

Poznámka:

V příkazu `obrazovka[i * SLOUPCE + j] = 0;` je použita skutečně konstanta `SLOUPCE`, nikoliv `RADKY` !!! Nevěříte-li, zkuste si to nakreslit na příkladu pole 3×2 . □

Dobré rady:

- Rozlišujte pojmy „třetí prvek“ pole a „prvek s indexem tří“. Třetí prvek pole má index 2, protože pole začínají vždy od indexu nula.
- Překročení mezí polí je kontrolováno a vyvolá výjimku: `ArrayIndexOutOfBoundsException`.

Běžné chyby:

- Indexy pole mohou být pouze konstanty, proměnné nebo výrazy typu `int`.


```
char c;
pole[c] = 1; // chyba
```
- Pole jsou indexována od `[0]` do `[jmenoPole.length - 1]`.


```
pole[-1] = 5; // chyba - o 1 méně
pole[pole.length] = 5; // chyba - o 1 více
```

Cvičení:

1. Napište program, který metodou Eratosthenova síta vypíše všechna prvočísla od 1 do N, kdy N zadáme z klávesnice.

2. Napište program, který přečte celé číslo typu `long` a vypíše jeho jednotlivé číslice oddělené dvěma mezerami. Jedná se o zobecněný příklad ze cvičení v kapitole 5.
3. Napište program, který přečte celé číslo typu `int` a vypíše jeho binární hodnotu.
4. Napište program, který přečte celé číslo typu `long` složené jen z nul a jedniček, které bude chápáno jako binární číslo. Vypište dekadickou hodnotu tohoto čísla.

8 Třídy a objekty – základní dovednosti

V objektově orientovaném jazyce je základním stavebním kamenem **třída** (*class*). Třída představuje soubor proměnných (nebo lépe dat – mohou tam být i konstanty) a podprogramů. Proměnným se říká **členské proměnné** (*member variables*) nebo též **datové složky** nebo **atributy** a je v nich uložen **stav** objektu (viz dále). Podprogramům se říká **metody** (*methods*) a manipuluje s členskými proměnnými, čímž mění stav objektu. Metody tedy popisují **schopnosti** („dovednosti“, vlastnosti, ...) objektu.

Třída je ale jen jakási šablona (**objektový typ**) a sama o sobě nemá přidělenou žádnou paměť, tedy nedá se pracovat ani s proměnnými ani s metodami¹. Můžeme si ji představit ve velkém zjednodušení např. jako datový typ `int` – dokud nedeklarujeme proměnnou typu `int`, není nám tento datový typ prakticky k ničemu.

Objekt (*object*) je datový prvek, který je vytvořen podle vzoru třídy. Často se říká, že objekt je **instance třídy** (*instance*) a často se termíny **objekt** a **instance** vzájemně volně zaměňují.

Podle jednoho vzoru třídy lze vytvořit libovolné množství objektů (**instancí**), které mají stejné „dovednosti“ – tj. metody, a mohou mít navzájem různé nebo i stejné stavy, tj. obsahy členských proměnných.

Poznámka:

Když bude v programu použita jen jedna jediná třída (s metodou `main()`) a tato třída bude mít alespoň jednu metodu `instance` (bez **static**), musí se objekt této třídy v `main()` vytvořit. To je jeden z velkých problémů², které musí programátor, zvyklý dosud na neobjektový jazyk, ve svém chápání Javy překonat. □

Pozor:

Proměnné, ať již třídy (se **static**) nebo instance (bez **static**), jsou v dokumentaci Java Core API označovány jako **field** (pole, oblast). Nechte se tím zmást, nemusí to být (a většinou nejsou) pole (**array**). □

¹ Pokud před nimi nebylo klíčové slovo **static** – viz [6/97].

² Třída vlastně vytváří instanci sama sebe.

8.1 Deklarace třídy

Program v Javě obsahuje vždy alespoň jednu třídu. Její deklarace začíná klíčovým slovem **class** a jsou v ní deklarovány proměnné i metody. Tělo třídy začíná a končí stejně jako tělo metody, tj. složenými závorkami **{ }** .

Příklad 65:

Budeme-li chtít pracovat s obdélníky, je vhodné vytvořit takovou třídu, která bude v sobě uchovávat všechny informace (tj. datové prvky) potřebné pro obdélník, a všechny metody, které se nám v danou chvíli jeví pro práci s obdélníkem užitečné.

```
public class Obdelnik {
    public int sirka;
    public int vyska;

    public int obvod() {
        int pom;
        pom = 2 * (sirka + vyska);
        return pom;
    }

    public int obsah() {
        return (sirka * vyska);
    }
}
```

Poznámky:

- Metody **obvod()** a **obsah()** jsou záměrně napsány různým způsobem, aby bylo jasné, že v nich lze využít všech možností, které byly již vysvětleny v [6/97].
- Všimněte si, že nikde není použito klíčové slovo **static**.
- Klíčové slovo **public** znamená, že takto označené metody a proměnné jsou „dobře“ přístupné i z vnějšku třídy. Pokud toto klíčové slovo neuvedete, nic se v našich jednoduchých programech nestane. Podrobně se lze o významu **public** dočíst v [12.3.3/210].
- Protože je třída **Obdelnik** označena jako **public**, musí být uložena v samostatném souboru pojmenovaném **Obdelnik.java** .

- Metoda `main()` nemusí být (a často nebývá) součástí každé třídy. Pokud ji tam ovšem uvedete, nic se nestane, protože `main()` může být ve všech třídách – viz [3.3/34].
- Způsob zápisu lokálních proměnných je naprosto stejný, jako u statických metod popisovaných v [6.10.2/108].

Třída `Obdelnik` se od této chvíle stává univerzálním vzorem, pomocí něhož a klíčového slova `new` můžeme vytvořit objekty různých obdélníků. Dokud tyto objekty nevytvoríme, není nám třída `Obdelnik` prakticky k ničemu.

8.2 Vytvoření objektu

Pro vytvoření objektu potřebujeme provést dvě akce:

- Deklarovat referenční proměnnou typu `Obdelnik`, která bude v sobě uchovávat referenci na skutečný objekt (instanci) obdélníka – bude na něj odkazovat.
- Pomocí `new` nechat vytvořit v paměti objekt obdélníka a získanou referenci na něj přiřadit do připravené proměnné.

Například:

```
Obdelnik obd;
```

V této chvíli máme deklarovanou referenční proměnnou `obd`, ale dosud žádný objekt. To znamená, že stále ještě nemůžeme použít metody `obvod()` a `obsah()` ani proměnné `sirka` a `vyska`.

```
obd = new Obdelnik();
```

Nyní se dynamicky³ vytvořila instance (objekt) třídy `Obdelnik` a odkaz na tuto instanci se přiřadil do referenční proměnné `obd`. Od této chvíle máme (přes referenční proměnnou) k dispozici obě proměnné (nastavené implicitně na nulu) i obě metody.

Protože se u deklarace proměnných `sirka` a `vyska` nevyskytlo klíčové slovo `static`, jedná se o **proměnné instance**. Tyto dvě proměnné vznikají vždy po použití `new`, to znamená, že každá instance třídy `Obdelnik` má své vlastní proměnné `sirka` a `vyska`. Protože patří konkrétní instanci, nazývají se také **instanční proměnné**.

³Všechny objekty jsou vytvářeny dynamicky.

Totéž platí i pro metody `obvod()` a `obsah()`, které se nedají použít samostatně (jak to bylo možné u statických metod), ale jejich použití je vázáno na konkrétní instanci. Podrobně viz [8.4/122].

8.3 Přímý přístup k datům objektu

Dosud vytvořený obdélník má nulovou výšku i šířku⁴. My ovšem můžeme hodnoty proměnných této konkrétní instance změnit. Pro tuto akci použijeme jméno referenční proměnné `obd` a jméno proměnné instance (např. `vyska`) oddělené tečkovou notací, např.:

```
obd.vyska = 5;
obd.sirka = 3;
```

Poznámka pro programátora v C či C++:

Je to jako kdybychom v C přistupovali k prvku struktury. □

Poznámka:

Možná cítíte, že toto není z hlediska bezpečnosti nejšťastnější způsob manipulace s daty. Je to tak – postupně si ukážeme lepší. □

8.4 Práce s metodami

Máme-li inicializovány proměnné instance, je možné použít smysluplně i její metody. Bez explicitní inicializace proměnných jdou ovšem použít také, otázkou je ale, nakolik smysluplné budou výsledky.

Pro přístup k metodám se používá naprostě stejný zápis jako pro přístup k datům, tj. opět operátor tečka za pomoci referenční proměnné.

```
i = obd.obvod();
```

Toto je výrazný rozdíl od procedurálního jazyka C, kde by bylo nutné zapsat něco jako: `i = obvod(obd); // toto není Java !`

Tedy předat funkci `obvod()` odkaz na objekt obdélníka.

Možná si řeknete, že to zase není takový rozdíl, jestli je `obd` před nebo za `obvod()`. Na první pohled ne, ale při pozornějším pohledu zjistíme, že se v zápisu Javy⁵ skrývá významný prvek bezpečnosti. V Javě totiž můžete volat metodu, která počítá obvod, jen pro objekty typu `Obdelnik` a chybným voláním zabrání komplíátor.

⁴Z části [3.5/42] víme, že nelokální proměnné mají nulovou počáteční hodnotu.

⁵Obecně v OOP přístupu.

Poznámka:

Uvědomit si, že u objektového jazyka se klade důraz nejprve na celé objekty a pak teprve na jejich části (ať již jsou to data nebo metody), je další z velmi důležitých věcí při učení se objektově orientovanému jazyku jako je Java. □

Při volání: `i = obd.obvod();`

nepředáváme metodě `obvod()` žádný parametr a přesto pracuje nad správnými daty. Je to tím, že metoda pracuje implicitně nad daty své instance. To, která je „jeho“ instance, se dozví z referenční proměnné `obd`.

Skutečnost je však ještě trochu jiná. Každá metoda má jeden skrytý implicitní parametr, který se nikde neobjevuje. Ten se jmenuje `this` a odkazuje na „tuto instanci“.

Implicitní parametr `this` nikde nedeklarujeme. Pokud totiž metoda přistupuje k datům své třídy, pak není potřeba. Přesto jej občas používáme – příklad viz [8.6/127].

Otázkou je, kam vlastně příkazy pro vytvoření nového objektu zapíšeme. Určitě je nelze zapsat do těla metod `obvod()` či `obsah()` – viz též [10.2/164].

Řešení je jednoduché, zapíšeme je např. do metody `main()`, která je součástí třídy `Obdelnik`. Stejně dobré je ale možné vytvořit nový objekt třídy `Obdelnik` i v kterékoliv jiné třídě – je to častější postup, který bude ukázán v [10.1.1/163]. V následujícím příkladu použijeme pro jednoduchost první způsob.

Příklad 66:

Pokusy se třídou `Obdelnik`.

```
public class Obdelnik {  
    public int sirka;  
    public int vyska;  
  
    public int obsah() {  
        return (sirka * vyska);  
    }  
}
```

```

public int obvod() {
    int pom;
    pom = 2 * (sirka + vyska);
    return pom;
}

public static void main(String[] args) {
    Obdelnik obd = new Obdelnik();
    obd.vyska = 5;
    obd.sirka = 3;
    System.out.println("Obvod je: " + obd.obvod());
}
}

```

Vypíše:

Obvod je: 16

Poznámka:

Všimněte si, že nikde není (kromě `main()`) uvedeno klíčové slůvko **static**, což je významný rozdíl od metod popisovaných v [6/97]. Jinak ale pro metody instance (to jsou metody bez **static**) platí vše, co bylo v [6/97] popisováno. Příklady budou postupně uvedeny dále. □

Pozor:

V terminologii OOP se často používá pojem **zasílání zpráv**, který znamená právě vyvolání metody nad nějakým objektem. □

8.5 Konstruktory

Při vytváření nového objektu třídy `Ondelnik` byla použita deklarace s inicializací, což je velmi častý způsob zápisu. V ní se ale objevil zvláštní zápis:

```
Obdelnik obd = new Obdelnik();
```

Je zřejmé, že musíme použít jméno třídy `Ondelnik` při deklaraci referenční proměnné `obd`, protože je to jméno typu. Proč je však nutné jméno třídy opakovat, a ještě k tomu se závorkami? Přece by úplně stačilo klíčové slůvko `new`, které říká, že se má vytvořit nová instance. Z typu referenční proměnné `obd` by si přece kompilátor mohl odvodit, že je nutné vytvořit instanci právě třídy `Ondelnik` a ne `Ctverec`, `Kruh` či jiné třídy.

To je naprostá pravda a pokud bychom se smířili s tím, že po vytvoření nové instance musíme vždy explicitně inicializovat datové prvky instance (zde `obd.vyska = 5;` a `obd.sirka = 3;`), pak bylo vše v pořádku.

Sami ale asi cítíte, že tento způsob není ideální. Stačí si představit třídu, která má dvacet datových prvků, a navíc provázaných složitými vazbami. Pak by vytvoření a inicializace každé nové instance byla otravná práce, náchylná k chybám.

Samozřejmě by nás napadlo u takové třídy napsat metodu `inic()`, ve které by se všechny datové prvky správně nastavily. Pak by stačilo pouze napsat:

```
Obdelnik obd = new;      // POZOR! toto není Java
obd.inic();
```

a objekt by byl správně vytvořen a inicializován.

Když se ale podíváme na:

```
Obdelnik obd = new Obdelnik();
zjistíme, že tato myšlenka už tvůrce Javy (a nejen jí) napadla.
```

Při vytváření objektu je totiž vždy dána možnost inicializace datových prvků objektu (případně dalších akcí) voláním specializované metody nazývané **konstruktor** (*constructor*). Konstruktor má vždy stejné jméno, jako je jméno třídy, čímž se předejde všem nejednoznačnostem.

Konstruktor je metoda, která nemá žádnou návratovou hodnotu, ale může mít libovolný počet parametrů i různých typů.

Příklad 67:

Vhodnější deklarace třídy `Obdelnik` s využitím konstruktoru.

```
public class Obdelnik {
    public int sirka;
    public int vyska;

    public Obdelnik(int parSirka, int parVyska) {
        sirka = parSirka;
        vyska = parVyska;
    }

    public int obvod() { ... } // nemění se
    public int obsah() { ... } // nemění se
}
```

Poznámka:

V konstruktoru se nesmí uvést jako návratový typ **void**, ani se nesmí vracet žádná hodnota pomocí **return**. Pokud potřebujeme předčasně ukončit práci konstruktoru, použijeme **return** bez parametru, např.:

```
if (parSirka == 0)
    return;
```



Nyní se použití objektu typu **Obdelnik** zjednoduší, protože se inicializace stala součástí vytvoření objektu.

```
public static void main(String[] args) {
    Obdelnik obd = new Obdelnik(5, 3);
    System.out.println("Obvod je: " + obd.obvod());
}
```

Příklad 68:

Vytvoření konstruktoru inicializujícího data objektu nás ale nezbavuje možnosti přistupovat k těmto datům. Je možné (ale ne moc chytré) napsat:

```
public static void main(String[] args) {
    Obdelnik obd = new Obdelnik(1, 2);
    obd.vyska = 5;
    obd.sirka = 3;
    System.out.println("Obvod je: " + obd.obvod());
}
```

Příklad 69:

Rozumnější použití přímého přístupu k proměnným instance je např.:

```
public static void main(String[] args) {
    Obdelnik obd = new Obdelnik(5, 3);
    System.out.println("Obednik vysky " + obd.vyska +
        " a sirky " + obd.sirka +
        " ma obvod: " + obd.obvod());
}
```

Vypíše:

Obednik vysky 5 a sirky 3 ma obvod: 16

8.5.1 Implicitní konstruktor

Jak je možné, že fungoval příklad uvedený v [8.4/123], kdy jsme žádný konstruktor nenapsali? Je to jednoduché – konstruktor není povinná složka třídy. Pokud sami žádný nenapíšeme, překladač vytvoří **implicitní konstruktor** (*default constructor*), jehož úkolem je vyřešit „vztahy s předky“ – podrobně viz [11.3/183].

Nechceme-li vytvářet konstruktor, ale chceme-li mít nastaveny proměnné instance na nějakou počáteční hodnotu, je možné tyto proměnné deklarovat s inicializací.⁶ Inicializace proměnných je provedena před voláním implicitního i explicitního konstruktora, např.:

```
public class Obdelnik {
    public int sirka = 5;
    public int vyska = 3;
```

Poznámka:

Není nutné inicializovat všechny proměnné. Ty explicitně neinicializované budou implicitně inicializovány na nulu. □

8.6 Využití **this** pro přístup k proměnným

Konstruktor uvedený v [8.5/125] lze napsat i takto:

```
public Obdelnik(int sirka, int parVyska) {
    this.sirka = sirka;
    vyska = parVyska;
}
```

Všimněte si použití **this**. To se dá s výhodou využít k přístupu k proměnným instance, mají-li stejný název, jako formální parametry metody. Zápis **this.sirka** pak znamená odkaz na proměnnou instance **sirka**. Toto je praktická vlastnost, která nás nenutí vymýšlet si různé „zkomoleniny“ jmen jen proto, abychom odlišili formální parametry metody od členských proměnných (negativní příklad je **vyska = parVyska;**)

Dobrá rada:

V žádném případě ale nemíchejte tyto dva způsoby (zde je to uvedeno pouze jako ukázka). Vyberte si jeden způsob a toho se držte. □

⁶Zcela stejně jako statické proměnné třídy.

Totéž, co platí pro formální parametry, platí i pro lokální proměnné, které by zastiňovaly díky svému stejném jménu členské proměnné. Zpřístupníme je opět pomocí `this`. Například pokud bychom (nešťastně) pojmenovali pomocnou proměnnou `sirka`, je možné přesto správně vypočítat obvod.

```
public int obvod() {
    int sirka;
    sirka = 2 * (this.sirka + vyska);
    return sirka;
}
```

8.7 Přetížení metod a konstruktorů

Pro přetížení metod instance platí naprosto stejná pravidla, jaká byla uvedena v [6.8/104] pro přetížení metod třídy – proto je zde nebudeme opakovat.

Je ale zajímavé, že je možné přetížit i konstruktory. K čemu je to dobré? Většinou k tomu, že chceme mít možnost inicializovat vznikající objekt různými způsoby. Pak je nutné mít více konstruktorů rozlišených typy, počtem nebo pořadím parametrů.

Příklad 70:

Třída s přetíženými konstruktory.

```
public class Obdelnik {
    public int sirka;
    public int vyska;

    public Obdelnik(int sirka, int vyska) {
        this.sirka = sirka;
        this.vyska = vyska;
    }

    public Obdelnik(Obdelnik o) {
        sirka = o.sirka;           // this není nutné
        this.vyska = o.vyska;      // ale zlepší čitelnost
    }

    public Obdelnik() {
        sirka = 1;
        vyska = 1;
    }
}
```

```
public int obvod() { ... } // nemění se  
public int obsah() { ... } // nemění se  
}
```

Zde máme tři různé konstruktory, první již známe, druhý inicializuje nový objekt hodnotami jiného objektu a třetí konstruktor je bez parametrů a vytvoří „jednotkový“ obdélník.

Poznámka:

Tento třetí konstruktor nemá na rozdíl od prvních dvou přílišný praktický význam. Zde je uveden proto, aby se předešlo oblíbenému omylu, že konstruktor musí mít parametry, protože bez parametrů může být pouze implicitní konstruktor. Je vidět, že to není pravda.

Konstruktor bez parametrů se ale často vytváří – důvody viz [11.3/183].

□

Použití je například toto:

```
public static void main(String[] args) {  
    Obdelnik obd = new Obdelnik(5, 3);  
    Obdelnik jiny = new Obdelnik(obd);  
    Obdelnik jedn = new Obdelnik();  
  
    System.out.println("Obvod je: " + obd.obvod());  
    System.out.println("Obvod je: " + jiny.obvod());  
    System.out.println("Obvod je: " + jedn.obvod());  
}
```

Vypíše:

```
Obvod je: 16  
Obvod je: 16  
Obvod je: 4
```

Pozor:

Jakmile jednou vytvoříme libovolný konstruktor s parametry, pak překladač nevytvoří implicitní konstruktor. Budeme-li tedy současně potřebovat i konstruktor bez parametrů, musíme si jej napsat sami. Důvod viz [11.3/183].

□

8.8 Využití `this` pro přístup ke konstruktoru

Klíčové slovo `this` má ještě speciální využití v přetížených konstruktorech. Pomocí něj (bez tečky, ale se závorkami) může konstruktor vyvolat jiný konstruktor stejné třídy. Volání `this()` se vztahuje na ten konstruktor, jehož parametry vyhovují co do počtu a pořadí typů.

Pozor:

Volání jiného konstruktoru pomocí `this()` musí být první příkaz ve volajícím konstruktoru. □

Příklad 71:

Je možné přepsat konstruktory z [8.7/128] jako volání prvního z nich.

```
public Obdelnik(int sirka, int vyska) {
    this.sirka = sirka;
    this.vyska = vyska;
}

public Obdelnik(Obdelnik o) {
    this(o.sirka, o.vyska);
}

public Obdelnik() {
    this(1, 1);
}
```

Poznámka:

V uvedeném příkladě zřejmě není příliš patrná výhoda takového kroku, ale když si představíme, že by konstruktor nastavoval dvacet proměnných (a toto nastavení by se ve všech konstruktorech nutně téměř beze změny opakovalo), pak je jasné, že se vyplatí si tuto možnost minimálně pamatovat. □

Pozor:

Tento způsob volání konstruktoru má ale jedno omezení. Volání pomocí `this()` musí být prvním příkazem v novém konstruktoru.

Konstruktor, který `this()` používá, může pak obsahovat libovolné množství jiných příkazů či volání metod (viz [8.9/131]), ale `this()` musí být jako první. Důvodem je, že v konstruktoru se automaticky volá konstruktor rodičovské třídy – podrobně viz [11.3/183]. □

8.9 Volání metod jinými metodami téže třídy nebo konstruktorem

Metody instance mohou libovolně volat jiné metody instancí z dané třídy. To znamená, že metody v jedné třídě jsou si rovny a neexistuje mezi nimi žádný „hierarchický vztah“. Konstruktor třídy může volat všechny metody téže třídy a může volat i jiné konstruktory téže třídy. Metoda smí konstruktor zavolat pouze prostřednictvím operátora `new`.

Příklad 72:

Volání metody téže třídy konstruktorem.

```
public void nastavSirku(int sirka) {  
    this.sirka = sirka;  
}  
  
public Obdelnik(int sirka, int vyska) {  
    nastavSirku(sirka);  
    this.vyska = vyska;  
}
```

Kde konstruktor volá metodu `nastavSirku()`.

8.10 Použití proměnné třídy v objektech

V deklaraci třídy `Obdelnik` byly dvě proměnné (`sirka` a `vyska`), které se vždy vytvoří znova pro každou instanci (po každém příkazu `new`). Je to pochopitelné, protože každý obdélník vytvořený podle tohoto vzoru má svoji šířku a výšku nezávislou na rozdílu mezi obdélníky.

Ale jak bylo již popsáno v [6.10.1/107], je možné deklarovat ve třídě proměnnou uvozenou klíčovým slovem `static`. Ta se nebude duplikovat v instancích této třídy, ale bude patřit jen této třídě. To znamená, že existuje v programu pouze v jediné kopii – je to téměř „globální“ proměnná známá z jiných programovacích jazyků, ale ukrytá ve třídě.

Tato proměnná, jak již víme, se nazývá **proměnná třídy** (na rozdíl od **proměnné instance**) a lze ji využívat i tehdy, pokud třída nemá žádné instance, tj. nebyly dosud vytvořeny žádné objekty pomocí operátoru `new`.

K čemu je ale proměnná třídy dobrá, když pomineme příklady z [6/97], které ukazovaly, jak lze v Javě programovat bez využití objektů?

Příklad 73:

Jako školní příklad se uvádí situace, kdy si chceme pamatovat, kolik objektů (instancí) jsme již od dané třídy vytvořili. U příkladu třídy `Obednik` to nemá valný význam, ale u příkladu třídy `Zakaznik` to svůj význam nepochybně má.

```
public class Zakaznik {  
    public static int pocetZakazniku = 0; // promenna tridy  
    public int utratil; // promenna instance  
  
    public Zakaznik() {  
        Zakaznik.pocetZakazniku++;  
        this.utratil = 0;  
    }  
  
    public void platil(int cena) {  
        this.utratil += cena;  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Pocet zakazniku: " + pocetZakazniku);  
        Zakaznik zak1 = new Zakaznik();  
        System.out.println("Pocet zakazniku: " + pocetZakazniku);  
        zak1.platil(15);  
        Zakaznik zak2 = new Zakaznik();  
        System.out.println("Pocet zakazniku: " + pocetZakazniku);  
        zak1.platil(30);  
        zak2.platil(20);  
        System.out.println("Utratili: " + zak1.utratil + " + "  
                           + zak2.utratil);  
        pocetZakazniku++; // nesmysl, ale prekladači nevadí  
    }  
}
```

Vypíše:

```
Pocet zakazniku: 0  
Pocet zakazniku: 1  
Pocet zakazniku: 2  
Utratili: 45 + 20
```

Všimněme si, že v konstruktoru je vidět rozdílný přístup k proměnné třídy a instance. K proměnné třídy přistupujeme pomocí jména třídy

a jména proměnné (`zakaznik.pocetZakazniku`), kdežto k proměnné instance pomocí `this` a jména proměnné (`this.utratil`).

V tomto jednoduchém případě, kdy se pohybujeme v rámci jedné třídy, to bylo zbytečné, protože konstruktor lze napsat i jako:

```
public Zakaznik() {
    pocetZakazniku++;
    utratil = 0;
}
```

Pokud bychom však chtěli k proměnné `pocetZakazniku` přistupovat z metod jiné třídy (např. `Obchod`), pak je nezbytný způsob zápisu:

`zakaznik.pocetZakazniku`

To je mimo jiné také výhoda, protože nikdy nemůže dojít ke kolizi jmen proměnných tříd z různých tříd – třídy se musí jmenovat různě a na jménu proměnných pak nezáleží.

Z prvního příkazu metody `main()` je vidět, že je možné využít proměnnou třídy `i` za neexistence instance této třídy (počet zákazníků je nulový).

K takto deklarované proměnné třídy je možný přímý přístup umožňující i změnu, což v tomto případě není příliš výhodné – viz závěrečný logicky nesmyslný příkaz: `pocetZakazniku++;`

Řešením je omezit přístupová práva k této proměnné – viz [10.3/166].

Poznámka:

Není zde řešeno, jak zajistit, aby se zákazníci také správně „ubírali“. Způsob bude popsán v [8.14/141]. □

Klíčové slovo `this` se dá použít kupodivu i pro přístup k proměnné třídy. Vhodnější a čitelnější zápis je ale přístup pomocí jména třídy.

Příklad 74:

Matoucí přístup k proměnné třídy pomocí `this`.

```
public class Pokus {
    public int promInstance;
    public static int promTridy;
    public Pokus(int promInstance, int promTridy) {
        this.promInstance = promInstance;
        this.promTridy = promTridy; // možný způsob
        Pokus.promTridy = promTridy; // čitelnější způsob
    }
}
```

```

public static void main(String[] args) {
    Pokus p = new Pokus(1, 2);

    System.out.println(p.promInstance + " " + promTridy);
}
}

```

Pozor:

Statické metody – tedy ani `main()` – nemohou používat proměnné instance bez přístupu přes referenční proměnnou. Je to z toho důvodu, že statická metoda není svázána s žádnou instancí a lze ji proto použít, i když žádná instance ještě neexistuje, a tudiž neexistuje ani proměnná instance. Častá chyba v předchozím případě vypadá takto:

```
System.out.println(promInstance + " " + promTridy);
```

Zde se snažíme k `promInstance` přistoupit bez referenční proměnné `p`. Překladač to ovšem nedovolí.

To prakticky znamená, že používáme metody a proměnné instance ze stejné třídy pouze ve statické metodě `main()`. Pokud jsou používány v jiné statické metodě, pak se jedná s nejvyšší pravděpodobností o metodu jiné třídy. V opačném případě by statická metoda třídy musela vytvářet instanci vlastní třídy. To je sice možné (např. ve spojovém seznamu), ale níkterak časté. □

8.11 Použití statických metod v objektech

Metody tříd (nebo též statické metody), které jsou uvozené klíčovým slovem `static`, byly popsány již v [6/97]. Otázkou zde je, zda má smysl jich využívat v objektově orientovaném přístupu. Odpověď je jednoduchá – smysl mají a používají se poměrně často.

Uvedeme si dva příklady – první, který bude využívat již připravenou metodu třídy, a druhý, který bude tuto metodu i sám vytvářet.

8.11.1 Použití statické metody ze třídy z Java Core API

Java Core API dává k dispozici poměrně dost statických metod, které samozřejmě můžeme využívat.

Příklad 75:

Ve třídě **Obdelník** by se nám mohla hodit metoda instance **delkaUhlopříký()**, která může být napsána např. takto:

```
public class Obdelnik {
    // proměnné, konstruktory a metody obvod() a obsah()
    // se nemění

    public double delkaUhlopříký() {
        double pom;
        pom = (sirka * sirka) + (vyska * vyska);
        pom = Math.sqrt(pom);
        return pom;
    }

    public static void main(String[] args) {
        Obdelnik obd = new Obdelnik(6, 8);
        System.out.println("Uhlopříčka je: " +
                           obd.delkaUhlopříký());
    }
}
```

Vypíše:

Uhlopříčka je: 10.0

Všimněte si volání: `pom = Math.sqrt(pom);`

To vypadá jako volání druhé odmocniny nad instancí **Math**, ovšem žádná instance **Math** nebyla v **main()** vytvořena a ani tam nebyla deklarována žádná referenční proměnná s tímto jménem.⁷

Dále je ve volání **sqrt()** podezřelý skutečný argument **pom**. Ve skutečnosti je **sqrt()** **metodou třídy Math**, která vypočítá druhou odmocninu svého parametru. Protože je to metoda třídy, potřebuje skutečný parametr, ze kterého vypočítá odmocninu, protože třída **Math** nemá žádnou členskou proměnnou, se kterou by metoda pro výpočet odmocniny implicitně pracovala.

Tento způsob je velmi výhodný, protože bez existence statických metod bychom museli pro výpočet odmocniny nejprve vytvořit objekt třídy, která má jako jednu z instančních metod metodu **sqrt()**.

⁷ Navíc by se podle konvencí musela jmenovat **math** – s malým **m** na začátku.

8.11.2 Použití statické metody z téže třídy

Ve třídě `Zakazník` nám vadila možnost modifikace počtu zákazníků mimo konstruktory, tj. použití **neautorizovaného přístupu** (viz [10.3/166]), protože tak se dala např. snadno „falšovat“ data. Řešením je znepřístupnit pomocí `private` statickou proměnnou `PocetZakazniku` z vnějšku a dát k dispozici metodu, která vrací aktuální počet zákazníků. Měla by to být metoda třídy, protože:

- Informace o počtu zákazníků se vztahuje ke třídě `Zakazník`, nikoliv k objektům jednotlivých zákazníků.
- Počet zákazníků potřebujeme zjistit kdykoliv, tj. i tehdy, když momentálně není žádný objekt typu `Zakazník` vytvořen, např. pro zjištění, zda již (nebo ještě) v obchodě někdo je.

Příklad 76:

Vytvoření statické metody pro přístup ke statické proměnné.

```
public class Zakaznik {  
    private static int pocetZakazniku = 0; // promenna tridy  
    public int utratil; // promenna instance  
  
    // konstruktor zustava nezmnen  
  
    public static int kolikZakazniku() {  
        return pocetZakazniku;  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Pocet zakazniku: " +  
                           kolikZakazniku());  
        Zakaznik zak1 = new Zakaznik();  
        System.out.println("Pocet zakazniku: " +  
                           Zakaznik.kolikZakazniku());  
        System.out.println("Pocet zakazniku: " +  
                           zak1.pocetZakazniku);  
        System.out.println("Pocet zakazniku: " +  
                           zak1.kolikZakazniku());  
    }  
}
```

Vypíše:

```
Pocet zakazniku: 0  
Pocet zakazniku: 1  
Pocet zakazniku: 1  
Pocet zakazniku: 1
```

Zde byla oproti dřívějšímu případu změněna deklarace:

```
public static int pocetZakazniku = 0;
```

na deklaraci:

```
private static int pocetZakazniku = 0;
```

proto, aby proměnná pocetZakazniku nebyla přístupná vně třídy. Od této doby jakékoliv pokusy vně třídy o Zakaznik.pocetZakazniku odhalí překladač jako chybu – podrobně viz v [10.3/166].

Poznámky:

- Všimněte si, že uvnitř třídy Zakaznik je možné volat metodu kolikZakazniku() i jako Zakaznik.kolikZakazniku(). Vně této třídy je pak možný jen způsob Zakaznik.kolikZakazniku().
- Věnujte pozornost také posledním dvěma kuriózním řádkám, kdy je vidět, že jak k proměnné třídy (zak1.pocetZakazniku), tak i k metodě třídy (zak1.kolikZakazniku()) lze přistoupit pomocí referenční proměnné. Ale tento způsob, stejně jako způsob přístupu pomocí **this** uvedený v [8.10/133], určitě nezvýší čitelnost programu.

Metody třídy se významně liší od metod instance tím, že se jim nepředává implicitní ukazatel **this**. To v důsledku znamená, jak již bylo dříve uvedeno, že metody třídy nemohou používat proměnné instance ani nemohou volat metody instance (leda že dostanou jako parametr referenci na již existující instanci). Metody instance ale mohou volat metody třídy.

Příklad 77:

Vztah mezi metodami třídy a metodami instance.

```
public class Zakaznik {  
    static int pocetZakazniku = 0; // promenna tridy  
    public int utratil; // promenna instance  
  
    public void platil(int cena) {  
        this.utratil += cena;  
        pocetZakazniku++;  
        int i = kolikZakazniku();  
    }  
}
```

```
public static int kolikZakazniku() {
    int i = utratil;      // chyba
    platil(10);          // chyba
    return pocetZakazniku;
}
```

Poznámka:

Z předchozího výkladu by možná mohla vzniknout domněnka, že metody třídy jsou dobré pouze k „obejítí“ objektově orientovaného přístupu a jinak mají jen samé nevýhody. Není to pravda – na příkladu výpočtu odmocnin bylo vidět, že dokonce i v knihovnách Javy existují třídy, které mají některé (nebo dokonce všechny) metody statické. Je to např. z toho důvodu, že operují nad reálnými čísly, která jsou základním datovým typem, a bylo by tudíž krajně nepraktické vytvářet komplikované metody instancí a následně i instance pro tyto jednoduché operace.

Další třídou, která stojí v této souvislosti za zmínku, je třída z API `System`, ve které jsou deklarovány různé užitečné metody, jako např. `currentTimeMillis()`, která se používá pro zjišťování časových poměrů v programu – viz též [19.3.1/305].

V této souvislosti lze také vysvětlit dosud záhadné:

```
System.out.println()
```

kde `System` je třída, `out` je její statická proměnná (proměnná třídy), která je referenční. Odkazuje na objekt třídy `PrintStream`, jejíž jedna metoda instance je `println()`. □

8.12 Inicializace proměnných třídy

Jak již víme z [8.5/125], lze proměnné instance inicializovat pomocí konstruktora. Provádět totéž pro proměnné třídy sice lze, ale není to příliš rozumné, protože při každém vytvoření nového objektu by se proměnné třídy znova a znova inicializovaly.⁸

Otzážka je, proč se s tím zabývat, když u proměnných třídy lze bez problémů uvést při deklaraci jejich inicializační hodnoty – viz právě třídu `Zakaznik`. Možné to je a také se to tak v naprosté většině případů provádí. Potíž však nastává v případech, kdy je nutné např. inicializovat pole konstant, a to netriviálním způsobem. Pak je možné využít způsob, který se nazývá **statický inicializační blok** (*static initialization block*).

⁸To, že jsme v konstruktoru třídy `Zakaznik` měnili proměnnou `pocetZakazniku`, je věc jiná. To byla změna, nikoliv počáteční inicializace.

Příklad 78:

Představme si třídu, která pracuje s prvočísly, a aby je nemusela neustále počítat, vypočte si je jednou na začátku do pole konstant.⁹

```
public class Prvocisla {
    public static final int MAX = 1000;
    public static final int cisla[] = new int[MAX];

    static {                      // statický inicializační blok - začátek
        int pocet = 2;
        cisla[0] = 1;
        cisla[1] = 2;

        dalsi:
        for (int i = 3; pocet < MAX; i += 2) {
            for (int j = 2; j < pocet; j++) {
                if (i % cisla[j] == 0) {
                    continue dalsi;
                }
            }
            cisla[pocet] = i;
            pocet++;
        }
    }                                // statický inicializační blok - konec

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

Všimněte si, že konstanty nemusí mít přiřazenou hodnotu při deklaraci. Je možné jim přiřadit tuto hodnotu kdekoli, ale pouze jednou.

Poznámka:

Statický inicializační blok se provede na samém začátku programu, přesněji okamžitě po natažení třídy do paměti a její verifikaci. □

⁹Inicializovat pole výčtem hodnot by bylo pro 1000 prvočísel poněkud nudné, rozsáhlé a silně náchylné k chybám.

8.13 Rušení objektů

V [8.2/121] bylo ukázáno, jak pomocí `new` vytvořit nový objekt. Logická otázka zní, jak odstranit tento objekt v případě, že jej již nepotřebujeme. Dosud jsme se tímto problémem nepotýkali, protože uvedené programy byly jednoduché a po jejich skončení jsme mlčky předpokládali, že „něco“ se o zaniklé objekty za nás postará.

Tento přístup, tak nebezpečný v jazyce C, je zcela bezpečný v Javě. Zde skutečně existuje „něco“, co odstraní nepotřebné objekty z paměti bez zásahu programátora. Toto „něco“ se jmenuje *garbage collector* (čistič paměti ; -) a průběžně se stará o uvolňování nepotřebných objektů.¹⁰

Jak se pozná nepotřebný objekt? Snadno – není na něj žádný odkaz z existujících referenčních proměnných.¹¹ Velice jednoduše můžeme zajistit zneplatnění odkazu a to dvěma způsoby, např.:

```
Obdelnik obd = new Obdelnik(5, 3);
obd = new Obdelnik(10, 4);
```

nebo:

```
Obdelnik obd = new Obdelnik(5, 3);
obd = null;
```

V prvním případě se do referenční proměnné přiřadí odkaz na jiný objekt (na obdélník o velikosti 10×4). V druhém případě se referenční proměnná rovnou „vynuluje“.

Pozor:

V žádném případě se ale zcela neztratí odkaz na původní obdélník 5×3 . Ten se uchovává v systému tak dlouho, dokud se nespustí *garbage collector*, který objekt zruší a odkaz skutečně definitivně vymaže. □

Druhý způsob se vyplatí používat v tom případě, že dočasně pracujeme s velkými objemy paměti, např.:

```
public void slozityVypocet() {
    double[] pomPole = new double[100000];
    for (int i = 0; i < 100000; i++) {
        // nějaký výpočet používající pomPole
    }
    pomPole = null;
    // pokračování výpočtu
}
```

¹⁰Tato technika není nicméně novým, poprvé byla použita před 30 lety v programovacím jazyce Simula.

¹¹To se lehce řekne, ale hůř zajistí, důležité však je, že se o to opravdu nemusíme starat.

V tomto případě se ihned po opuštění cyklu `for` označí `pomPole` za nepoužívané a *garbage collector* jej při nejbližší příležitosti z paměti smaže. Podstatné je ale, aby výpočet v této metodě dále pokračoval, protože jinak by příkaz: `pomPole = null;`
byl prakticky zbytečný. Jedná se totiž o lokální proměnnou, která zaniká při ukončení metody. Tím, že zanikne, se také uvolní paměť, kterou alokovala.

Garbage collector běží na pozadí programu s nízkou prioritou, takže jeho činnost běžící program zpomaluje jen málo. Jsou dvě možnosti, kdy se *garbage collector* spustí s vysokou prioritou, čímž výrazněji omezí nás program:

- Dojde k nedostatku paměti a *garbage collector* se snaží uvolnit nepotřebné části.
- Přímo *garbage collector* vyvoláme příkazem: `System.gc();` což je ovšem velmi řídký případ.

8.14 Ukončení práce s objekty

Mohlo by se zdát, že je tato část zbytečná, protože v minulé části bylo napsáno, že když přestaneme s objektem pracovat, *garbage collector* jej sám odstraní z paměti. To je samozřejmě pravda, ale stejně, jako jsme potřebovali konstruktor pro správnou inicializaci objektu při vzniku, potřebujeme občas něco, co před skutečným zánikem objektu provede nezbytné ukončovací akce. Toto „něco“ se jmeneuje *finalizer* („ukončovač“ : -)).

Finalizer je metoda instance (nesmí být `static`), která vrací `void`, má vyhrazené jméno `finalize()` a nemá žádné parametry. *Finalizer* může být – na rozdíl od konstruktorů – pouze jeden, protože jej nelze přetížit. Přesně řečeno, jeho hlavička musí vypadat jen takto:

```
protected void finalize() throws Throwable {
```

Jako poslední příkaz této metody by mělo být uvedeno:

```
super.finalize();
```

což znamená volání metody `finalize()` z nadřazené třídy – podrobně viz dále.

Kdy je potřeba *finalizer* naprogramovat? Většinou tehdy, když má objekt ještě jiné zdroje, než jen přidělenou paměť. Typicky jsou to otevřené soubory.

Java (přesněji JVM) spouští *finalizery* uvolněných objektů tehdy, když se jí to hodí. Potřebujeme-li využít spuštění *finalizerů*, je nutné použít metodu `System.runFinalization();`

Příklad 79:

Činnost finalizeru ukážeme na třídě `Zakaznik`, kdy bude vhodné po zrušení objektu snížit počet zákazníků.

```
public class Zakaznik {  
    // proměnné a metody jsou nezměněny  
  
    protected void finalize() throws Throwable {  
        pocetZakazniku--;  
        System.out.println("Konec zakaznika");  
        super.finalize();  
    }  
  
    public class Hlavní {  
        public static void main(String[] args) {  
            System.out.println("Pocet: " + Zakaznik.kolikZakazniku());  
            Zakaznik zakl = new Zakaznik();  
            System.out.println("Pocet: " + Zakaznik.kolikZakazniku());  
            Zakaznik zak2 = new Zakaznik();  
            System.out.println("Pocet: " + Zakaznik.kolikZakazniku());  
            zakl = null;  
            System.runFinalization();  
            System.gc();  
            System.out.println("Pocet: " + Zakaznik.kolikZakazniku());  
        }  
    }  
}
```

Vypíše:

```
Počet: 0  
Počet: 1  
Počet: 2  
Konec zakaznika  
Počet: 1
```

Poznámka:

I přesto, že lze volání `finalizerů` vynutit, je zřejmé, že zde je to řešení „přes ruku“, protože po každém uvolnění objektu musíme pamatovat na volání `runFinalization()` a `gc()`. Tato volání nelze vložit do metody `finalize()` třídy `Zakaznik`, protože právě tu chceme pomocí nich vyvolat. □

Dobré rady:

- Inicializujte proměnné instance v konstruktoru, nikoliv při deklaraci.
- Při deklaraci třídy uveděte vždy nejdříve všechny proměnné a konstanty a pak konstruktory a nakonec metody.

Běžné chyby:

- Přetížení metody pouhou změnou návratové hodnoty.

```
int secti(int a, int b)      a    long secti(int a, int b)
```
- Lokální proměnná není před prvním použitím inicializována.
- V deklaraci konstruktoru se vrací nějaká hodnota nebo **void**

```
class Pokus {  
    int Pokus() {
```
- Konstruktor má **return**, který vrací nějakou hodnotu; samotný **return;** je povolen.
- Jakmile ve třídě existuje nějaký konstruktor s parametry, nelze při vytváření instance této třídy použít implicitní konstruktor bez parametrů, viz též [8.7/129].

```
class Pokus {  
    int i;  
    int Pokus(int k) { i = k; }  
}  
* * *  
Pokus p = new Pokus();
```
- Použití **this** ve statické metodě.
- Použití proměnných instance ve statické metodě bez referenční proměnné. Tím je méněno, že tato statická metoda nemá jako svoji lokální proměnnou nebo formální parametr žádnou referenční proměnnou, pomocí níž by se pomocí operátoru tečka dalo přistoupit k proměnným instance.
- Volání metod instance ve statické metodě bez referenční proměnné.

Cvičení:

1. Napište třídu `Trojuhelnik`, která bude poskytovat metody `double obvod()` a `boolean jePravouhly()`. Členské proměnné budou celá čísla `stranaA`, `stranaB` a `stranaC`.
2. Modifikujte třídu `Trojuhelnik` tak, aby konstruktor ošetřil případy, kdy zadávané strany netvoří trojúhelník (tj. součet dvou stran je menší, než třetí strana). V tomto případě konstruktor nastaví všechny strany na nulu.
3. Přetěžte konstruktor `Trojuhelnik(int jednaStrana)`, který bude vytvářet rovnostranné trojúhelníky.
4. Přetěžte konstruktor `Trojuhelnik(int jednaStrana, int druháStrana)`, který bude vytvářet rovnoramenné trojúhelníky.
5. Vytvořte třídu `PravouhlyTrojuhelnik`, kdy se jako parametry konstruktoru budou zadávat pouze odvěsný a přepona se dopočítá. Třída bude poskytovat metodu `double delkaPrepony()`.
6. Přetěžte konstruktor `PravouhlyTrojuhelnik()`, který vytvoří rovnoramenný pravoúhlý trojúhelník s odvěsnami délky 1. Třída bude poskytovat metodu `double obsah()`.

9 Řetězce a znaky

Protože v dalších kapitolách budou ve stále větší míře používány řetězce, je vhodné, aby byly podrobněji vysvětleny. To je možné učinit až zde, protože z předchozí kapitoly již víme, jak se vytvářejí a používají objekty.

Řetězec je v Javě samostatný objekt (instance třídy `String`, přesněji řečeno `java.lang.String`). To znamená, že s řetězcem se pracuje jako s celým objektem, nikoliv jako s polem prvků typu `char`.

Poznámka pro programátora v C či C++:

Žádný ukončující znak jako je '\0' známý z jazyka C se v Javě nepoužívá. □

Pozor na skutečnost, že řetězce jsou konstantní, což znamená, že jednou vytvořený řetězec již nelze měnit. Potřebujeme-li řetězec průběžně měnit, je nutné vytvořit nový objekt typu `StringBuffer`, což je vlastně „měnitelný“ řetězec. Převody z jednoho typu řetězců na druhý jsou možné několika způsoby, které budou uvedeny dále.

Instance typu `String` se používá všude tam, kde se pracuje s konstantními řetězci, což je kupodivu většina případů. Tento typ je rychlejší a paměťově úspornější než typ `StringBuffer`. Navíc je vyloučena jeho nechtěná modifikace, což znamená, že program je bezpečnější.

Poznámka pro programátora v C či C++:

Java důsledně hlídá překročení mezí obou typů řetězců a odhalené problémy hlásí pomocí výjimky `StringIndexOutOfBoundsException`. □

Poznámka:

Java s řetězci zachází někdy tak, jako by to byly základní datové typy, např. nalezně-li kompilátor ve zdrojovém textu text v uvozovkách, okamžitě pro něj vytvoří objekt typu `String`. □

Pozor:

Jako u každého pole, i u řetězců má první prvek index `[0]` . □

9.1 Vytvoření řetězce

Nejjednodušejí a nejefektivnější lze vytvořit objekt řetězce přiřazením textu v uvozovkách do příslušné proměnné, např.:

```
String s = "ahoj";
```

Kromě toho lze použít pro vytvoření řetězce množství konstruktorů, podle následujícího příkladu.

Příklad 80:

```
String s1, s2, s3, s4, s5, s6, s7;
byte[] bajty = {(byte)'E', (byte)'v', (byte)'a'};
char[] znaky = {'M', 'a', 'r', 't', 'i', 'n', 'a'};
StringBuffer buf = new StringBuffer("dobry den");

s1 = new String("cao");
s2 = new String(s1);
s3 = new String(bajty);
s4 = new String(bajty, 1, 2);
s5 = new String(znaky);
s6 = new String(znaky, 3, 4);
s7 = new String(buf);
```

Vytiskneme-li tyto řetězce pomocí např.:

```
System.out.println("s1:" + s1);
```

dostaneme:

```
s1:cao
s2:cao
s3:Eva
s4:va
s5:Martina
s6:tina
s7:dobry den
```

Zajímavý je řetězec `s4`, kde se z pole bajtu vezmou dva bajty počínaje bajtem s indexem 1 (bajty jsou číslovány od nuly, jak je v polích zvykem). Totéž platí pro řetězec `s6`, jen se berou čtyři znaky od znaku s indexem 3 počínaje.

Poznámka:

Při vytváření řetězce z pole bajtu je zajímavé, že se bajty považují za znaky v osmibitovém kódování a to, jak se přivedou na 16bitové

znaky, záleží na přednastaveném kódování znaků. To je velmi důležité pro různá kódování češtiny – podrobně viz v [UJJ2].

V tomto případě, kdy byly použity jen neakcentované znaky, na přednastaveném kódování nezáleží. □

Pro zjištění aktuální délky řetězce lze použít jeho metodu `length()`, která vrací počet znaků. Jiným způsobem nelze délku řetězce zjistit.

```
System.out.println("delka s7 = " + s7.length());
```

Vypíše:

```
delka s7 = 9
```

Pozor:

Běžnou chybou je považovat řetězec za pole, které má svoji délku uloženou v **proměnné** `length`. U řetězce je to **metoda** `length()`. □

9.1.1 Inicializované pole řetězců

Při programování často využijeme možnost vytvořit inicializované pole řetězců. Jednotlivé řetězce se uvedou do složených závorek a navzájem se oddělí čárkami.

Příklad 81:

```
String[] pole = {"Dana", "Eva", "Martina"};
for (int i = 0; i < pole.length; i++)
    System.out.println(pole[i]);
```

9.2 Práce s celými řetězci

9.2.1 Porovnávání

Java umožňuje řetězce porovnávat několika způsoby pomocí metod:

- `compareTo()` – porovná lexikograficky dva řetězce
 - `compareToIgnoreCase()` – porovná lexikograficky dva řetězce, přičemž nerozlišíuje velká a malá písmena
- Obě metody vrací `int` číslo <0, pokud je řetězec v parametru metody větší, 0 v případě shody obou řetězců a `int` >0, je-li menší.
- `equals()` – zjistí, zda jsou řetězce shodné

- `equalsIgnoreCase()` – zjistí, zda jsou řetězce shodné, přičemž ne-rozlišuje velká a malá písmena

Obě metody vrací `true` v případě shody obou řetězců a `false` v případě neshody.

Různým porovnáním, např.:

```
System.out.println("s1.compareTo(s2): " + s1.compareTo(s2));
následujících řetězců:
```

```
String s1, s2, s3;
s1 = new String("ahoj");
s2 = new String("ahoi");
s3 = new String("AHOJ");
```

dostaneme:

```
s1.compareTo(s2): 1
s2.compareTo(s1): -1
s1.compareToIgnoreCase(s3): 0
s1.equals(s3): false
s1.equalsIgnoreCase(s3): true
```

Pozor:

Porovnávat řetězce pomocí operátoru `==` nelze, byť by nás k tomu svá-děla analogie s operátorem `+`. Operátor `==` pro řetězce pracuje jinak, než pro základní datové typy. U řetězců zjišťuje, zda obě referenční proměnné ukazují na tentýž objekt v paměti, tj. zda se jedná o tutéž instanci řetězce. To znamená, že pro dva různé řetězce se stejným obsahem vrátí operátor `==` hodnotu `false`. □

9.2.2 Převody na malá či velká písmena

Celý řetězec lze převést na malá nebo velká písmena pomocí metod `toLowerCase()` nebo `toUpperCase()`.

Příklad 82:

```
String s = "mala a VELKA";
System.out.println(s.toLowerCase()); // mala a velka
System.out.println(s.toUpperCase()); // MALA A VELKA
```

Poznámka:

Na malá nebo velká písmena je možný převod řetězců obsahujících i akcentované znaky – podrobně viz [UJJ2]. □

9.2.3 Spojení řetězců

Pro tuto operaci můžeme použít buď operátor + nebo se stejným výsledkem metodu concat(). V obou dvou případech se ze dvou spojovaných řetězců vytvoří řetězec třetí a oba dva spojované řetězce se nemění.

Příklad 83:

```
String s1 = "mala a";
String s2 = " VELKA";
String s3, s4;

s3 = s1 + s2;
s4 = s1.concat(s2);
System.out.println(s1); // "mala a"
System.out.println(s2); // " VELKA"
System.out.println(s3); // "mala a VELKA"
System.out.println(s4); // "mala a VELKA"
```

9.2.4 Náhrada všech znaků v řetězci

Pro tuto operaci použijeme metodu replace(), která vytvoří nový řetězec s již nahrazenými znaky.

Příklad 84:

```
String s2, s1 = "cacao";
s2 = s1.replace('c', 'k');
System.out.println(s1); // "cacao"
System.out.println(s2); // "kakao"
```

9.3 Práce s částí řetězce

9.3.1 Získání části řetězce

Potřebujeme-li získat z řetězce podřetězec, použijeme metodu substring().

Příklad 85:

```
String s2, s3, s1 = "mala a VELKA";
s2 = s1.substring(5);
s3 = s1.substring(5, 9);
System.out.println(s2); // "a VELKA"
System.out.println(s3); // "a VE"
```

kdy v prvním případě se bere řetězec od znaku s indexem 5 (šestý v pořadí) až do konce řetězce. Ve druhém případě se začíná od stejněho znaku, ale nový podřetězec bude končit znakem **před** znakem s indexem 9 (tj. končí devátým znakem v pořadí včetně).

Poznámka:

Všimněte si, že obě metody vrací typ `String`, takže pro řetězce `s2` a `s3` jsme nepotřebovali konstruktor – do referenční proměnné `s2` jsme přiřadili odkaz na nově vytvořený řetězec. □

Příklad 86:

Podobným způsobem můžeme umístit podřetězec do pole znaků.

```
String s = "mala a VELKA";
char znaky[] = new char[10];
s.getChars(2, 9, znaky, 0);
System.out.println(znaky); // "la a VE"
```

kde parametry znamenají: „kopíruj podřetězec od indexu 2 do indexu 8 včetně, do pole nazvaného `znaky` od jeho nultého indexu“.

9.3.2 Práce se začátkem a koncem řetězce

Velmi snadno lze také otestovat, zda řetězec začíná a končí určitým podřetězcem.

Příklad 87:

```
String s = "mala a VELKA";
if (s.startsWith("mala") == true)
    System.out.println("Zacina na \"mala\"");
if (s.endsWith("mala") == false)
    System.out.println("Nekonci na \"mala\"");
```

Vypíše:

Zacina na "mala"

Nekonci na "mala"

9.3.3 Oříznutí bílých znaků na okrajích

Tuto operaci potřebujeme provádět poměrně často v případě práce s řetězcem načteným ze vstupu. U takového řetězce jsou na jeho konci znaky odrádkování, kterých se potřebujeme zbavit. Používaná metoda `trim()` ubere všechny mezery, tabulátory a nové řádky jak na konci řetězce, tak i na jeho začátku.

Příklad 88:

```
String s2, s1 = "\r\n\t ahoj\t \r\n";
s2 = s1.trim();
System.out.println("Zacatek:" + s1 + ":konec");
System.out.println("Zacatek:" + s2 + ":konec");
```

Vypíše:

Zacatek:

ahoj

:konec

Zacatek: ahoj :konec

9.4 Práce s jednotlivými znaky řetězce

9.4.1 Získání znaku

Potřebujeme-li získat jednotlivé znaky z řetězce, použijeme metodu `charAt()`.

Příklad 89:

```
String s = "mala a VELKA";
System.out.println(s.charAt(7)); // vytiskne znak V
```

9.4.2 Hledání znaku

Znak můžeme vyhledávat od začátku směrem ke konci řetězce, a to buď od prvního znaku nebo od libovolného znaku v pořadí. Stejně tak lze hledat znak od konce řetězce směrem k jeho začátku. Není-li znak nalezen, vrací všechny metody hodnotu -1, je-li nalezen, je vrácen jeho index.

```
String s = "mala a VELKA";
int i;
i = s.indexOf('a');
System.out.println("Prvni a je na " + i + ". pozici");
i = s.indexOf('a', i + 1);
System.out.println("Dalsi a je na " + i + ". pozici");
i = s.lastIndexOf('a');
System.out.println("Posledni a je na " + i + ". pozici");
i = s.lastIndexOf('a', i - 1);
System.out.println("Predposledni a je na " + i + ". pozici");
```

Vypíše:

Prvni a je na 1. pozici
 Dalsi a je na 3. pozici
 Posledni a je na 5. pozici
 Predposledni a je na 3. pozici

Poznámka:

Všechny čtyři uvedené metody lze použít i pro hledání podřetězce v řetězci, např.:

```
i = s1.lastIndexOf("VEL");
System.out.println("Posledni VEL je na "+ i +". pozici");
```

□

9.5 Konverze základních datových typů na řetězec

Jakýkoliv základní datový typ (od `boolean` až po `double`) lze zkonzervovat na řetězec voláním statické metody `valueOf()`. Tuto akci potřebujeme provést nejčastěji před tiskem hodnoty příslušného datového typu.

Příklad 90:

```
boolean b = true;
int i = 1234567;
double d = Math.PI;
String s;

s = String.valueOf(b);
System.out.println("b: " + s);
s = String.valueOf(i);
System.out.println("i: " + s);
s = String.valueOf(d);
System.out.println("d: " + s);
```

Pozor na skutečnost, že metoda `valueOf()` je metodou třídy a je tedy nutné před ní uvést jméno třídy `String`.

Poznámka:

Pokud tiskneme pouze pomocí `System.out.println()`, není třeba metodu `String.valueOf()` volat, protože se při vyhodnocení skutečných

parametrů metody `println()` vyvolá automaticky. Pokud však chceme výsledky před tiskem ještě upravovat, např. oříznout počet míst za desetinnou tečkou na pět, je třeba metodu `String.valueOf()` použít.

```
System.out.println(Math.PI);      // 3.141592653589793
s = String.valueOf(Math.PI);
i = s.indexOf('.');
s = s.substring(0, i + 6);
System.out.println(s);           // 3.14159
```

□

Celá čísla lze konvertovat i do jiné číselné soustavy pomocí statických metod tříd `Integer` a `Long`. Jedná se o metody `toBinaryString()`, `toOctalString()` a `toHexString()`.

Příklad 91:

```
int i = 254;
System.out.println(Integer.toBinaryString(i));    // 11111110
System.out.println(Integer.toOctalString(i));      // 376
System.out.println(Integer.toHexString(i));        // fe
```

9.6 Konverze řetězce na základní datové typy

Po vstupu nějaké hodnoty z klávesnice dostáváme tuto hodnotu nejčastěji v poli znaků nebo bajtů. Toto pole sice lehce zkonzervujeme na řetězec, ale pak je nutné zkonzervovat řetězec ještě na odpovídající datový typ. Ukázku těto akce jsme již viděli v [4.3/71] v metodě `ctiInt()`.

Pro převod se používají metody tříd `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` a `Double` z balíku `java.lang`. Všechny tyto třídy obsahují statickou metodu `valueOf()`, která vráci řetězec zkonzervovaný na objekt třídy příslušného datového typu. Tuto třídu je pak nutné převést na odpovídající základní datový typ některou z metod `xxxValue()`.

```
double d1 = Double.valueOf("3.14").doubleValue();
double d2 = new Double("3.14").doubleValue();
boolean b = Boolean.valueOf("true").booleanValue();
int i = Integer.valueOf("123").intValue();
```

Poznámka:

Jak je vidět z příkladu proměnné `d2`, lze instanci třídy příslušného typu (zde `Double`) vytvořit standardní cestou pomocí konstruktoru. □

Celočíselné typy reprezentované třídami Byte, Short, Integer a Long mají statickou metodu valueOf() přetíženou, takže jako druhý parametr lze zadat základ číselné soustavy, ze které se má převádět, např.:

```
int j = Integer.valueOf("1A2B", 16).intValue();
```

Uvedené celočíselné typy nabízejí ještě jeden způsob konverze, a to pomocí metod třídy parseXXX(). Tyto metody jsou opět přetíženy, takže lze převádět i z jiných soustav než jen z desítkové.

```
long l1 = Long.parseLong("12345");
long l2 = Long.parseLong("1A2B", 16);
```

9.7 Vyvolání více metod jedním příkazem

Množství metod třídy String vrací opět objekt typu String, což prakticky znamená, že vytvoří nový řetězec. Odkaz na tento řetězec není nutné ukládat do pomocné referenční proměnné a pak s ní dále pracovat, ale můžeme volání téhoto metod spojit (zřetězit) za sebe.

Příklad 92:

```
String s1 = "\r\n\t cacao\t \r\n";
int i;
i = s1.trim().toUpperCase().substring(2).indexOf('O');
System.out.println("O je " + (i + 1) + ". znak");
```

Zde se postupně vytvoří řetězce:

```
"cacac"      // po trim()
"CACAO"      // po toUpperCase()
"CAO"        // po substring(2)
```

a v řetězci "CAO" se teprve hledá index písmene O. Všechny zmíněné řetězce jsou po použití zrušeny pomocí garbage collectoru.

Poznámka:

Podobně lze jakoukoliv z výše zmíněných metod zavolat jako metodu konstantního řetězce, např.:

```
String s = "obj".concat(String.valueOf(i)).concat(".jpg");
což je užitečný příkaz na generování jmen souborů v nějakém cyklu. □
```

9.8 Metoda `toString()`

Metoda `toString()` má zvláštní postavení mezi metodami pracujícími s řetězci. Obsahuje ji totiž každý objekt, bez ohledu na to, zda má jinak s řetězci něco společného. Je to tím, že `toString()` je deklarována už v kořenové třídě `Object`¹.

Pokud ji ve své třídě nepřekryjete² a přesto použijete, dostanete řetězec, který je složen ze jména vaší třídy, oddělovacího znaku @ a povídného čísla, které ve skutečnosti představuje jednoznačnou identifikaci objektu. Například: `MujString@7243ef93`

Účelem metody `toString()` je poskytnout znakovou reprezentaci objektu, kterou lze přímo (tj. bez dalšího úsilí) tisknout, např. pomocí `System.out.println()`, nebo zapisovat do logovacích souborů či jakkoli jinak využít.

Výpis pomocí implicitní metody `toString()` je sice kdykoliv možný, otázkou však je, zda poskytuje dostatečné množství informace. Z tohoto důvodu je důrazně doporučováno ve své třídě metodu `toString()` překrýt. Programátor třídy totiž nejlépe ze všech ví, jak by informace o třídě měla vypadat, aby byla co nejúplnejší. Je třeba zdůraznit, že neexistují žádná nařízení, co by měl řetězec vracený metodou `toString()` obsahovat.

Příklad 93:

Následující příklad ukáže, jak je možné překrýt metodu `toString()`. Upravená metoda vypisuje jméno třídy (získané `getClass().getName()` – viz [11.8.4/198]) a po dvojtečce i obsah proměnné hodnota, což je jediný rozumný údaj, který třída `MujString` obsahuje.

```
public class MujString {  
    int hodnota;  
  
    MujString(int h) { hodnota = h; }  
  
    public String toString() { // public je nutny  
        String jmenoTridy = new String(getClass().getName());  
        return (jmenoTridy + ": " + hodnota);  
    }  
}
```

¹Viz [11.8/189].

²Viz [11.2/183].

```

void puvodniToString() {
    System.out.println(super.toString());
}

public static void main(String[] args) {
    MujString s1 = new MujString(5);
    s1.puvodniToString();
    System.out.println(s1.toString());
}
}

```

Vypíše:

MujString@7243ef93
MujString: 5

Poznámky:

- Specifikátor **public** před metodou **toString()** je nezbytně nutný.
Důvod, proč tomu tak je, naleznete v [12.4/210].
- Příkaz **super.toString()** je ukázkou možnosti zavolání původní metody **toString()** ze třídy **Object**. Podrobnosti viz [11.2/183].

9.9 Třída StringBuffer

Tato třída poskytuje typ „měnitelný řetězec“, který využijeme tehdy, když potřebujeme libovolně měnit jednotlivé znaky řetězce, případně zvětšovat či zmenšovat délku téhož řetězce. Všechny znaky řetězce – s výjimkou nastavených či inicializovaných – mají hodnotu `\u0000`.

9.9.1 Vytvoření instance

K dispozici jsou tři konstruktory: `StringBuffer b1, b2, b3;`

`b1 = new StringBuffer();`

vytvoří neinicializovaný řetězec o kapacitě 16 znaků,

`b2 = new StringBuffer(100);`

vytvoří neinicializovaný řetězec o kapacitě 100 znaků,

`b3 = new StringBuffer("Ahoj");`

vytvoří inicializovaný řetězec o kapacitě 20 znaků – čtyři znaky pro Ahoj a 16 znaků jako rezervu.

9.9.2 Délka řetězce

Zde je třeba rozeznávat aktuální délku, která se zjistí pomocí metody `length()`, a maximální možnou délkou (tj. kapacitu), kterou vrátí metoda `capacity()`.

Kapacitu řetězce je možné změnit dvěma metodami:

- `ensureCapacity(int k)`, která pro:
 - k větší, než je současná kapacita, zajistí, že řetězec bude mít kapacitu maxima z velikosti k a z dvojnásobku současně kapacity plus dva znaky navíc,
 - k menší, než je současná kapacita, ponechá řetězec nezměněn.
- `setLength(int k)`, která pro:
 - k větší než současná kapacita zvětší aktuální délku řetězce na k a kapacitu na dvojnásobek současně kapacity plus dva znaky,
 - k menší než současná kapacita, ponechá kapacitu nezměněnu a aktuální délku nastaví na k – může řetězec prodloužit i oříznout.

Příklad 94:

Pro proměnné `b1`, `b2` a `b3` platí nastavení z předchozího příkladu.

```
System.out.println(b1.length() + " " + b1.capacity()); // 0 16
System.out.println(b2.length() + " " + b2.capacity()); // 0 100
System.out.println(b3.length() + " " + b3.capacity()); // 4 20
b1.setLength(18);
b2.ensureCapacity(110);
b3.setLength(3);
System.out.println(b1.length() + " " + b1.capacity()); // 18 34
System.out.println(b2.length() + " " + b2.capacity()); // 0 202
System.out.println(b3.length() + " " + b3.capacity()); // 3 20
```

9.9.3 Změny celého řetězce

Metodou `reverse()` lze celý řetězec obrátit, tzn. poslední znak bude první atd.

Příklad 95:

```
b = new StringBuffer("Ahoj");
```

```
System.out.println(b);      // Ahoj
b.reverse();
System.out.println(b);      // johA
```

9.9.4 Změny části řetězce

Těchto změn je několik a budou postupně uvedeny včetně příkladů:

- Ke stávajícímu řetězci lze přidat na konec libovolný základní datový typ pomocí metody `append(typ t)`

```
b = new StringBuffer("Ahoj ");
b.append(true);
System.out.println(b);    // Ahoj true
b.append(7);
System.out.println(b);    // Ahoj true7
```

- Z řetězce lze libovolnou část vyříznout pomocí `delete(int poč_index, int kon_index)`

```
b.delete(5, 9);
System.out.println(b);    // Ahoj 7
```

- Po jednotlivých znacích lze ubírat pomocí `deleteCharAt(int index)`

```
b.deleteCharAt(0);
System.out.println(b);    // hoj 7
```

- Do řetězce lze kamkoliv vkládat libovolný datový typ `insert(int index, typ t)`.

```
b.insert(0, 3.14);
b.insert(1, "HOJ");
System.out.println(b);    // 3HOJ.14hoj 7
```

- Chceme-li nahradit jeden podřetězec jiným, použijeme `replace(int poč_ind, int kon_ind, String nový_podřez)`

```
b.replace(0, 5, "3,");
System.out.println(b);    // 3,14hoj 7
```

- Aniž bychom řetězec ovlivnili, získáme jednotlivé znaky pomocí `charAt(int index)`

```
char c = b.charAt(0);
System.out.println(c); // 3
```

- Změna jednotlivých znaků za jiné je možná díky

```
setCharAt(int index, char ch)
b.setCharAt(1, '!');
System.out.println(b); // 3!14hoj 7
```

Pozor:

Metody `delete()`, `deleteCharAt()` a `replace()` jsou k dispozici až od JDK 1.2.
Koncový index u metod `delete()` a `replace()` ukazuje na první ne-
□ použitý znak, nikoliv na poslední použitý.

9.9.5 Konverze na String

Častým požadavkem je konverze typu `StringBuffer` na `String`, což je možné pomocí dvou metod. Metoda `toString()` převede celý řetězec, metoda `substring()` jen jeho část od zadaného indexu včetně.

Příklad 96:

```
b = new StringBuffer("Ahoj");
String s1, s2;
s1 = b.toString();
s2 = b.substring(1);
System.out.println(s1); // Ahoj
System.out.println(s2); // hoj
```

9.10 Třída Character - práce s jednotlivými znaky

Ve třídě `java.lang.Character` je mimo jiné několik metod třídy užitečných jak pro rozpoznávání druhu znaku, tak i pro změnu jednotlivých znaků.

Poznámka pro programátora v C či C++:

Tyto metody jsou to zhruba ekvivalenty maker z `<ctype.h>`

9.10.1 Rozpoznávání druhu znaků

Máme k dispozici následující metody třídy – `isDigit()`, `isLetter()`, `isLetterOrDigit()`, `isLowerCase()`, `isUpperCase()` a `isWhitespace()`. Jejich jména jsou dostatečně významová, všechny mají jako parametr `char` a všechny vrací typ `boolean`.

Pozor:

Vyplatí se je používat, protože znaky v Javě jsou v Unicode, kde např. znaky s hodnotami '\u0BE7' až '\u0BEF' jsou platné tamilské číslice³, tedy `isDigit()` pro ně vrací `true`. Podobné „zvláštnosti“ platí i pro písmena atd. □

Příklad 97:

```
System.out.println(Character.isDigit('1'));           // true
System.out.println(Character.isDigit('\u0BE7'));        // true
System.out.println(Character.isLetter('A'));           // true
System.out.println(Character.isLetterOrDigit('?'));    // false
System.out.println(Character.isLowerCase('b'));         // true
System.out.println(Character.isUpperCase('B'));         // true
System.out.println(Character.isWhitespace('\n'));       // true
```

Poznámka:

Jak je to s českými akcentovanými znaky se dozvíte v [UJJ2]. □

9.10.2 Změna velikosti písmene

Můžeme použít metody třídy `toLowerCase()` a `toUpperCase()`. Obě mají jako parametr `char` (ten zůstává nezměněn) a obě vrací převedený znak.

Příklad 98:

```
char c, d = 'A';
c = Character.toLowerCase(d);
System.out.println("c = " + c + " d = " + d); // c = a d = A
d = Character.toUpperCase('\u00FD');          // '\u00FD' je 'ý'
System.out.println("d = " + d);               // d = Ý
```

Poznámka:

Pokud ve výpisu na obrazovce nevidíte ý, ale jiný znak, je to pouze problém fontů. Podrobnosti viz v [UJJ2]. □

³Sám nevím, jak vypadají ;-)

9.10.3 Převod jednotlivých znaků na čísla

Většinou převádíme na číslo najednou všechny znaky ve `String` pomocí již známých postupů (viz [9.6/153]). Občas ale můžeme potřebovat převod jen jednotlivých znaků, např. pro průběžnou kontrolu vstupu. Pak nám pomůže metoda: `int digit(char znak, int základ)`

Příklad 99:

```
int i = Character.digit('5', 10);
int j = Character.digit('F', 16);
System.out.println("i = " + i + " j = " + j); // i = 5 j = 15
i = Character.digit('\u0BE7', 10);           // tamilská jednička
System.out.println("i = " + i);             // i = 1
```

Poznámka:

Chceme-li získat ze znaku jeho Unicode číslo, můžeme použít přety-pování na `int`, např.: `int i = (int) 'A';` □

Dobré rady:

- Délku řetězce vrátí metoda `length()`, nikoliv proměnná `length`.
- Třídy `String` a `StringBuffer` představují kvalitativně odlišné objekty a mají tudíž i některé metody rozdílné.

Cvičení:

1. Existují slova, která se čtou stejně zleva doprava i zprava doleva (např. **radar**). Napište program, který bude tato slova generovat z jejich první poloviny zadané z klávesnice.
2. Napište program, který načítá řetězce složené z číslic. Pokud bude první číslicí 0, je zbytek řetězce považovaný za osmičkové číslo. Jsou-li na začátku znaky 0x (nebo 0X), pak se jedná o šestnáctkové číslo. Všechna ostatní čísla jsou desítková čísla. Toto načtené číslo vypište binárně.
3. Napište program, který přečte řetězec a v závislosti na jeho posledním znaku provede:
 - 1 (l) – převod řetězce na malá písmena (*lower*)
 - u (U) – převod řetězce na velká písmena (*upper*)
 - x (X) – prohození malých a velkých písmen (*exchange*)
 Změny proveděte v řetězci, nikoliv jen na výstupu.
4. Napište metodu `void tiskPI(int desetMist)`, která bude tisknout číslo `Math.PI` s přesností na zadaný počet desetinných míst.

10 Třídy a objekty - pokračování

V této kapitole bude popsáno několik důležitých částí, týkajících se tříd a objektů. Některé podkapitoly sebe přímo nenavazují, ale všechna zde uvedená fakta musíme znát před začátkem výkladu **dědičnosti** a **polymorfizmu**.

10.1 Modifikátory deklarace třídy

Dosud jsme měli celý program v jedné třídě a tato třída byla uložena ve stejnojmenném souboru. Tento jednoduchý způsob ale nelze používat pro větší programy, které sestávají z definic více tříd, které jsou navzájem provázány. Je sice možné uložit do jednoho souboru více tříd, ale snižuje to přehlednost a snadnou modifikovatelnost programu.

Naší snahou by mělo být mít pro každý objekt neprimitivního typu vlastní třídu uloženou (viz dále) v samostatném souboru. Hlavním programem je v ideálním případě třída, která obsahuje pouze metodu `main()`, která deklaruje referenční proměnné na ostatní třídy a pomocí nich volá jejich metody. Tím zajistíme bezproblémovou **znovupoužitelnost** (*reusability*) tříd v jiném programu.

Jakmile v programu použijeme více tříd, musíme se začít starat o to, jak budou tyto třídy navzájem spolupracovat.

Před klíčovým slovem `class` mohou být uvedeny tři modifikátory:

- **public** – označuje **veřejnou třídu**. Ta je přístupná i mimo balík (viz [12/2011]), kde je deklarována. Není-li třída označena jako `public`, je přístupná pouze ve svém balíku.
- **abstract** – označuje **abstraktní třídu**. Od této třídy nelze vytvořit instanci. Smysl abstraktní třídy je v tom, že tvoří společný základ pro více tříd, které od ní budou odděleny.
- **final** – je opak abstraktní třídy. **Finální (koncová)** třída může mít instance (lze podle ní vytvořit objekty), ale nesmí být použita jako rodičovská třída pro přípravu jiných tříd.

Lze spojit dvojice modifikátorů **public abstract** nebo **public final**, nikoliv však **final abstract**.

Nemá-li třída uveden žádný modifikátor, je komplátorem považována za neveřejnou (tj. není obecně přístupná), neabstraktní (může být podle ní vytvořen objekt) a nekoncovou (může být zděděna).

Poznámka:

Abstraktní a finální třídy mají smysl při použití dědičnosti, proto budou podrobněji popsány v [11.5/185] a [11.6/187]. \square

10.1.1 Třídy s modifikátorem **public**

Jakmile použijeme pro třídu modifikátor **public**, musí být tato třída uložena v samostatném souboru pojmenovaném stejně jako třída. Pokud třída modifikátor **public** uveden nemá, může ležet v souboru společně s jinou třídou – to je případ třídy **B** z následujícího příkladu.

Příklad 100:

```
// Soubor A.java
public class A {
    int a;
    public A(int par) { a = par; }
    public void setA(int par) { a = par; }
    public int getA() { return a; }
}

// Soubor Hlavni.java
class B {
    int b;
    public B(int par) { b = par; }
    public void setB(int par) { b = par; }
    public int getB() { return b; }
}

public class Hlavni {
    public static void main(String[] args) {
        B b = new B(3);
        A a = new A(5);
        System.out.println("a = " + a.getA() + ", b = " + b.getB());
    }
}
```

Překlad obou souborů je možný buď jednotlivě:

```
javac A.java
javac Hlavni.java
```

nebo všech souborů .java v adresáři najednou:

```
javac *.java
```

V obou případech se vytvoří soubory A.class, B.class a Hlavni.class. Program se spustí tak, že se spustí třída obsahující metodu main(), tedy:

```
java Hlavni
```

Poznámky:

- Je možné, aby i „nepublic“ třída B byla uložena v samostatném souboru. Ten se pak musí jmenovat B.java.
- Běžné chyby při překladu a spuštění viz v [2.6.2/26].

10.2 Kompozice objektů

Třída může mít členskou proměnnou objektového typu.¹ V tomto případě je třeba deklarovat referenční proměnnou najinou třídu.²

Příklad 101:

V následujícím příkladě vidíme, že třída Zamestnanec využívá dvě referenční proměnné na třídu Datum. Tyto proměnné jsou deklarovány jako proměnné instance, protože každý zaměstnanec musí mít svá vlastní data narození a nástupu. Všimněte si také metod `toString()`, které jsou použity pro výpis hodnot jak objektu typu Datum, tak i Zamestnanec.

```
class Datum {
    public int den, mesic, rok;

    public Datum(int den, int mesic, int rok) {
        this.den = den;
        this.mesic = mesic;
        this.rok = rok;
    }
    public Datum(Datum d) {
        this(d.den, d.mesic, d.rok);
    }
}
```

¹Nejdří na se o dědičnost – viz [11/179].

²Případně i sama na sebe – to je pak nejčastěji prvek spojového seznamu.

Překlad obou souborů je možný buď jednotlivě:

javac A.java

javac Hlavni.java

nebo všech souborů .java v adresáři najednou:

javac *.java

V obou případech se vytvoří soubory A.class, B.class a Hlavni.class.

Program se spustí tak, že se spustí třída obsahující metodu main(), tedy:
java Hlavni

Poznámky:

- Je možné, aby i „nepublic“ třída B byla uložena v samostatném souboru. Ten se pak musí jmenovat B.java.
- Běžné chyby při překladu a spuštění viz v [2.6.2/26].

10.2 Kompozice objektů

Třída může mít členskou proměnnou objektového typu.¹ V tomto případě je třeba deklarovat referenční proměnnou najinou třídu.²

Příklad 101:

V následujícím příkladě vidíme, že třída Zamestnanec využívá dvě referenční proměnné na třídu Datum. Tyto proměnné jsou deklarovány jako proměnné instance, protože každý zaměstnanec musí mít svá vlastní data narození a nástupu. Všimněte si také metod `toString()`, které jsou použity pro výpis hodnot jak objektu typu Datum, tak i Zamestnanec.

```
class Datum {
    public int den, mesic, rok;

    public Datum(int den, int mesic, int rok) {
        this.den = den;
        this.mesic = mesic;
        this.rok = rok;
    }
    public Datum(Datum d) {
        this(d.den, d.mesic, d.rok);
    }
}
```

¹Nejdříve se o dědičnost – viz [11/179].

²Případně i sama na sebe – to je pak nejčastěji prvek spojového seznamu.

```
public String toString() {
    StringBuffer b = new StringBuffer(100);

    b.append(den).append(".");
    b.append(mesic).append(".").append(rok);
    return b.toString();
}

}

class Zamestnanec {
    public String jmeno;
    public Datum narozeni, nastup;

    public Zamestnanec(String jmeno,Datum narozeni,Datum nastup) {
        this.jmeno = new String(jmeno);
        this.narozeni = new Datum(narozeni);
        this.nastup = new Datum(nastup);
    }

    public String toString() {
        StringBuffer b = new StringBuffer(200);
        b.append(jmeno);
        b.append(", narozen: ").append(narozeni.toString());
        b.append("\nnastoupil: ").append(nastup.toString());
        return b.toString();
    }
}

public class Kompozice {
    public static void main(String[] args) {
        Datum narozeni = new Datum(21, 5, 1960);
        Zamestnanec z = new Zamestnanec("Josef Novak", narozeni,
                                         new Datum(1, 10, 1990));
        System.out.println(z.toString());
    }
}
```

Vypíše:

Josef Novak, narozen: 21.5.1960
nastoupil: 1.10.1990

Poznámka:

V metodě `main()` je ve volání konstruktoru `Zamestnanec` použita konstrukce, kdy datum nástupu není předáváno pomocí referenční proměnné, ale je vytvořen nový objekt typu `Datum`. Hodnotou výrazu `new Datum(1, 10, 1990)` je odkaz, který je předán jako parametr konstruktoru. V něm se **trvale** přiřadí do referenční proměnné `nastup`. Není to nejčitelnější způsob zápisu, ale používá se velmi často. □

10.3 Autorizovaný přístup k datům

Zapouzdření dat (*data encapsulation*) je jedním z tří vždy zmiňovaných pilířů objektově orientovaných jazyků.³ Jedná se o explicitní spojení dat a metod dané konstrukcí třídy, tzn. data se vždy vyskytují společně s metodami, které s nimi manipuluji.

Autorizovaný přístup k datům je důsledkem zapouzdření, kdy zajištíme, aby s daty nebylo možno z vnějšku třídy manipulovat jinak, než pomocí metod této třídy.

Jak můžeme snadno zjistit v příkladě z předchozí části, všechny proměnné mají před sebou uvedeno klíčové slovo `public`, které znamená, že tyto proměnné jsou přístupné i z vnějšku třídy.

Dokud jsme měli programy, které sestávaly pouze z jedné třídy, bylo lhostejné, zda tento modifikátor uvedeme či nikoliv, protože všechny deklarované proměnné nám byly přístupné. Třídu jsme totiž vytvářeli my, a proto jsme byli sami zodpovědní za to, jak s těmito proměnnými manipulujeme.

V případě využití více tříd se situace radikálně mění. V našem programu můžeme využít třídu, kterou napsal někdo jiný, protože jednotlivé třídy mohou ležet (a většinou i leží) v různých souborech. V současném stavu věcí, kdy jsou všechny proměnné přístupné všem, není problém napsat v metodě `main()` např.:

```
z.nastup.den--;
```

což pak při výpisu zaměstnance způsobí zajímavý výpis:

`Josef Novak, narozen: 21.5.1960`

`nastoupil: 0.10.1990`

Z tohoto důvodu se snažíme data chránit před neautorizovaným přístupem. To v praxi znamená, že proměnné deklarujeme s přístupovým právem **private**⁴ místo `public`. Tím zabráníme jakémukoliv přístupu

³Zbývající dva pilíře jsou dědičnost – viz [11/179] a polymorfismus – viz [14/224].

⁴Existují ještě další přístupová práva a budou podrobně popsána až v [12.3/206].

k proměnným z vnějšku třídy. Jinak řečeno, mechanismem, jak zajistit autorizovaný přístup, je deklarace přístupových práv prvků třídy pomocí příslušného modifikátoru.

Pokud se však jedná o proměnnou, jejíž hodnota by měla být přístupná i zvnějšku, je třeba připravit **public** metody, které obsah proměnné zpřístupní pro čtení, případně povolí změnit.

Samozřejmě, že metoda, která mění obsah proměnné, může provádět libovolné množství kontrol správnosti, např.⁵

```
public void setDen(int den) {
    if (den >= 1 && den <= 31)
        this.den = den;
}
```

Tentýž postup pomocí klíčového slova **private** můžeme použít i pro označení některých metod, které slouží k implementaci **public** metod. Není ovšem šťastným řešením označit jako **private** jediný existující konstruktor. Tím bychom pak zabránili jakémukoliv použití třídy, protože by nebylo možné vytvořit její instanci.

Poznámka:

Pokud vás dráždí spojení angličtiny (set či get) a češtiny (Den), pak můžete použít „nastav“ a „vrat“, tedy `nastavDen()` a `vratDen()`.

Tento způsob ale není doporučenihodný, protože mnohé vývojové prostředky umožňují nechat si vytvořit tyto metody automaticky na základě deklarovaných proměnných. Pak se použije právě „get“ a „set“. Je to tak zařízené pojmenování, že se používají výrazy „getry“ a „setry“, označující skupiny metod, které vrací/nastavují proměnné instance. Z tohoto důvodu je vhodnější si na uvedený způsob zvyknout, byť „tahá za uši“. □

Příklad 102:

Autorizovaný přístup si ukážeme na příkladu bankomatu, který vydává peníze. Je zřejmé, že k penězům se musí přistupovat autorizovaným přístupem, protože:

- je může vybrat pouze ten, kdo zná PIN,
 - peníze musí být fyzicky k dispozici, tj. bankomat není „vybrán“.
- Z tohoto důvodu jsou označeny jako **private**.

⁵Použili-li jsme metodu `setDen()`, měla by existovat i metoda `getDen()`, která vrací hodnotu proměnné den.

Obě tyto kontroly může provést metoda `getPenize()`, která navíc provádí zápis o datu výběru. Pro načtení PIN se využívá metoda `ctiInt()` známá z [4.3/71].

```
import java.util.*;
public class Bankomat {
    private int penize;

    private boolean overPIN(int pin) {
        return (pin == 1234) ? true : false;
    }

    void setPenize(int kolik) {
        Date d = new Date();
        if (kolik > 0) {
            penize += kolik;
            System.out.println(d.toString() + " vlozeno: " + kolik);
        } else {
            System.out.println(d.toString() + " podezrely vklad: "
                               + kolik);
        }
    }

    int getPenize(int kolik) {
        Date d = new Date();
        System.out.print("Zadejte PIN: ");
        int pin = Vstup.ctiInt();
        if (overPIN(pin) == true) {
            if (penize >= kolik) {
                penize -= kolik;
                System.out.println(d.toString() + " vybrano: " + kolik);
                return kolik;
            }
            else
                System.out.println("Nedostatek hotovosti!");
        }
        else
            System.out.println(d.toString() + " PIN nesouhlasí");
        return 0;
    }
}
```

```
public class TestBankomatu {  
    public static void main(String[] args) {  
        Bankomat b = new Bankomat();  
        // b.penize = 5000; // nelze - autorizovaný přístup  
        b.setPenize(1000);  
        System.out.println("vybrano: " + b.getPenize(200));  
    }  
}
```

Vypíše při prvním (správném) a druhém (nesprávném) spuštění.

Mon Apr 03 16:41:23 GMT+02:00 2000 vloženo: 1000

Zadejte PIN: Mon Apr 03 16:41:23 GMT+02:00 2000 vybrano: 200

vybrano: 200

Mon Apr 03 16:41:30 GMT+02:00 2000 vloženo: 1000

Zadejte PIN: Mon Apr 03 16:41:30 GMT+02:00 2000 PIN nesouhlasí

vybrano: 0

10.4 Pole objektů

Budeme-li používat pole objektů (je jedno zda jednorozměrná, či více-rozměrná), je třeba dávat pozor na jednu (z prvního pohledu) poměrně základnou vlastnost.

V případě polí složených z primativních datových typů stačilo jen alokovat pole např.:

```
int[] pole = new int[10];
```

Tím bylo vytvořeno pole **int** o deseti prvcích a všechny prvky byly nastaveny na nulu. Toto byl průzračně jasný postup, který ale nelze aplikovat na pole objektů.

V případě pole objektů musíme provést ještě jeden krok navíc, a to pomocí **new** vytvořit instance jednotlivých objektů. Na tento krok se často zapomíná a pak je při přístupu k poli vyhodena výjimka **NullPointerException**, protože sice existuje pole referencí, ale tyto reference neukazují na žádný objekt a tudíž mají hodnotu **null**.

Příklad 103:

Vytvoření pole objektů.

```

class IntADouble {
    private int i;
    private double d;
    IntADouble(int i, double d) { this.i = i; this.d = d; }

    public void vypis() {
        System.out.println("i = " + i + ", d = " + d);
    }
}

public class PoleObjektu {
    public static void main(String[] args) {
        IntADouble[] louka;
        louka = new IntADouble[3];
        for (int j = 0; j < louka.length; j++) {
            louka[j] = new IntADouble(j, (double) (j * 2));
        }
        for (int j = 0; j < louka.length; j++) {
            louka[j].vypis();
        }
    }
}

```

Vypíše:

```

i = 0, d = 0.0
i = 1, d = 2.0
i = 2, d = 4.0

```

Co se v metodě main() děje?

Řádka: IntADouble[] louka;

deklaruje referenční proměnnou jménem louka, která umí ukazovat na pole objektů typu IntADouble. Žádné pole ovšem dosud neexistuje stejně jako neexistují žádné objekty typu IntADouble.

Řádka: louka = new IntADouble[3];

vytvoří pole tří referencí (ne objektů!!!) na objekty typu IntADouble a přiřadí jej do referenční proměnné louka. Žádné objekty typu IntADouble ovšem dosud neexistují.

Zde je velmi matoucí použití zdánlivého implicitního konstruktoru:

```
new IntADouble[3];
```

Není to ovšem volání konstruktoru, ale pokyn k vytvoření pole referencí.

Řádka v cyklu: `louka[j] = new IntADouble(j, (double) (j * 2));`
 teprve volá pro jednotlivé prvky pole louka konstruktor `IntADouble()`,
 kterým se až nyní vytvoří jednotlivé objekty typu `IntADouble`.

Pozor:

Uvedený postup se nedá obejít ani použitím implicitního konstruktoru (tzn. žádný konstruktor nevytvoříme) ani použitím konstruktoru bez parametrů. Bude-li třída `IntADouble` obsahovat konstruktor:

```
IntADouble() { this.i = 0; this.d = 0.0; }
```

musí vytvoření pole těchto objektů vypadat:

```
IntADouble[] louka;  
louka = new IntADouble[3];  
for (int j = 0; j < louka.length; j++)  
    louka[j] = new IntADouble();
```

□

Poznámka:

V obou případech (konstruktor s parametry i bez parametrů) lze ovšem napsat:

```
IntADouble[] louka = new IntADouble[3];
```

ale pak musí následovat cyklus vytváření jednotlivých instancí v nezměněné podobě.

□

10.5 Předávání skutečných parametrů metodám

10.5.1 Předávání primitivních datových typů

Java předává skutečné parametry všech primitivních datových typů zásadně jen **hodnotou** (*call-by-value*). To znamená, že metodě je předána kopie skutečného argumentu. To má následující výhody:

- skutečným parametrem může být i konstanta nebo výraz,
- skutečný parametr nelze omylem v metodě změnit (lze změnit jen předanou kopii, ale to se neprojeví navenek metody).

A také nevýhodu:

- skutečný parametr nelze v metodě změnit, i když si to přejeme.

Nutno říci, že tato nevýhoda, tak podstatná v procedurálních jazycích typu C, zde nevýhodou příliš není. Protože, když vlastně potřebujete měnit hodnotu skutečného parametru?

- Metody třídy i metody instance mají k proměnným ve své třídě přímý a neomezený přístup, tj. mohou je trvale změnit kdykoliv, bez toho, že by bylo nutné je předávat jako parametry.⁶
- V naprosté většině případů nevznikne požadavek na změnu skutečného parametru, protože zcela stačí využít návratové hodnoty metod tříd i metod instancí.

Příklad 104:

Ukázka možností volání hodnotou.

```
class Nasobeni {
    public static int nasobDvema(int i) { return i * 2; }
    public int nasobTremi(int i) { return i * 3; }
}

public class Hodnotou {
    public static void main(String[] args) {
        int první = 5, druhý = 7;
        Nasobeni n = new Nasobeni();
        první = Nasobeni.nasobDvema(první);
        druhý = n.nasobTremi(druhý);
        System.out.println("první: " + první + ", druhý: " + druhý);
    }
}
```

Vypíše:

první: 10, druhý: 21

Poznámka:

V příkladu je použita jak metoda třídy `nasobDvema()`, tak i metoda instance `nasobTremi()`, aby bylo vidět, že pokud jde o předávání skutečných parametrů, fungují obě stejným způsobem. □

⁶Jsou to pro ně „globální“ proměnné.

10.5.2 Předávání objektů

Objekty se metodám předávají **referencí**. To znamená, že musíme použít formální parametr objektového typu (referenční proměnnou). Jako skutečný parametr se pak hodnotou přenáší odkaz na již existující objekt. Protože pak má metoda k dispozici referenční proměnnou, může pomocí přistupovat k metodám daného objektu nebo i k jeho datovým prvkům. Tento způsob předávání parametru se zcela běžně používá – jiný nelze použít.

Tím, že lze metodě předat objekt, je možné zrealizovat i předání primativního datového typu „odkazem“,⁷ např. protože chceme v jedné metodě nastavit dvě či více proměnných.⁸

Prakticky to znamená, že si musíme vytvořit novou třídu, jejíž členeskou proměnnou bude primativní datový typ, který chceme nastavovat, pak vytvořit instanci této třídy a přiřadit členské proměnné hodnotu primativního datového typu. Poté se metodě, která má měnit skutečné parametry, předá reference na tuto instanci. Přes referenční proměnnou se změní hodnota členské proměnné, která se po návratu přiřadí (přiřazovacím příkazem) do proměnné primativního datového typu.

Příklad 105:

Volání odkazem pomocí „bezpečného“ postupu – třída `IntOdkazem` – a jednoduššího, ale méně bezpečného postupu – třída `DoubleOdkazem`. Ta je tak jednoduchá, jak jen může být. Využívá se v ní možnosti přímého přístupu k proměnné `d`. Je ale nutné zdůraznit, že princip předání pomocí reference metodě `nasobPeti()` je u tříd `IntOdkazem` a `DoubleOdkazem` naprosto stejný.

```
class IntOdkazem {
    private int i;

    IntOdkazem(int i) { this.i = i; }
    public void setInt(int i) { this.i = i; }
    public int getInt() { return this.i; }
}

class DoubleOdkazem {
    double d;
}
```

⁷Je tím míněn způsob „volání odkazem“ jako protiklad „volání hodnotou“.

⁸Pak už to nelze zajistit pomocí návratové hodnoty.

```

class Nasobeni {
    public static void nasobPeti(IntOdkazem j, DoubleOdkazem f) {
        int pom = j.getInt();
        j.setInt(pom * 5);
        f.d *= 5;
    }
}

public class Odkazem {
    public static void main(String[] args) {
        int prvni = 5;
        double druhý = 3.14;

        IntOdkazem par1 = new IntOdkazem(prvni);
        DoubleOdkazem par2 = new DoubleOdkazem();
        par2.d = druhý;
        Nasobeni.nasobPeti(par1, par2);
        prvni = par1.getInt();
        druhý = par2.d;

        System.out.println("prvni: " +prvni + ", druhý: " +druhý);
    }
}

```

Vypíše:

prvni: 25, druhý: 15.700000000000001

Poznámky:

- Výpis hodnoty: druhý: 15.700000000000001 je hezká ukázka toho, že reálná čísla jsou v počítači zobrazena většinou pomocí approximace.
- Pro tento účel by bylo vhodné použít přímo třídy z Java Core API Integer, Double, Float atd., které zabalují primitivní datové typy do objektu.
Tyto třídy mají metodu typu intValue(), která vrátí hodnotu prvku, ale bohužel žádná z těchto tříd nemá opačnou metodu setValue()⁹. Tyto třídy ani nelze zdědit a metodu přidat, protože jsou označeny jako final.

⁹To považuji za poněkud nevhodné, ale nepodařilo se mi zjistit, proč to tvůrci tříd tak naprogramovali.

- Je vidět, že celý postup je pro primitivní datové typy „poněkud přes ruku“, a proto je třeba omezit jeho používání jen na nezbytně nutné (a odůvodněné) případy.

10.5.3 Předávání polí jako skutečných parametrů metodám

Pole jsou předávány metodám pomocí reference.¹⁰ To znamená, že hodnoty prvků pole lze v metodě trvale změnit. Předávání *referencí* je také výhodné z hlediska efektivity programu, protože při předávání skutečných parametrů není nutné vytvářet kopii celého pole.

Pozor:

Je-li pole složeno pouze z primitivních datových typů a předáváme-li metodě pouze jeden prvek pole, je předáván hodnotou. □

Poznámka pro programátora v C či C++:

Protože nedílnou součástí každého pole je jeho délka uložená v proměnné `length`, není třeba tuto délku předávat metodě dodatečným parametrem. Na druhou stranu ale nemůžeme příslušné metodě předat ke zpracování pouze výsek pole, protože se metodě předá pole vždy celé. □

Příklad 106:

Následující příklad ukáže, jak lze předat pole metodě odkazem, tj. je možné jeho prvky trvale měnit. Kromě toho je v metodě `nastavPole()` použito označení `final` pro oba formální parametry.

Z příkladu je jasné vidět, že `final` nedovolí změnu hodnoty parametru (zde `h`). Pokud se ale jedná o pole (případně o referenční proměnnou), je i přes označení `final` možno měnit hodnoty jednotlivých prvků pole (případně hodnoty proměnných objektu, na které ukazuje referenční proměnná). Hodnotu této proměnné ovšem měnit nelze (zde `p = null;`).

```
class PraceSPolem {
    public void nastavPole(final int[] p, final int h) {
        for (int i = 0; i < p.length; i++)
            p[i] = h + i;
        // h++;           // chyba
        // p = null;      // chyba
    }
}
```

¹⁰Presněji řečeno – metodě je předána hodnotou kopie referenční proměnné odkazující na pole, podobně jako v případě objektů.

```

public static void tiskniPole(int[] p) {
    for (int i = 0; i < p.length; i++)
        System.out.print("[" + i + "] = " + p[i] + ", ");
    System.out.print("\b\b \n");
}
public void nastavPrvek(int prvek, int h) {
    prvek = h;
}
}

public class PoleOdkazem {
    public static void main(String[] args) {
        PraceSPolem obj = new PraceSPolem();
        int[] pole = {5, 4, 3, 2, 1};
        PraceSPolem.tiskniPole(pole);
        obj.nastavPole(pole, 3);
        PraceSPolem.tiskniPole(pole);
        obj.nastavPrvek(pole[0], 5);
        PraceSPolem.tiskniPole(pole);
    }
}

```

Vypíše:

```

[0] = 5, [1] = 4, [2] = 3, [3] = 2, [4] = 1
[0] = 3, [1] = 4, [2] = 5, [3] = 6, [4] = 7
[0] = 3, [1] = 4, [2] = 5, [3] = 6, [4] = 7

```

Poznámky:

- Z poslední řádky výpisu je vidět, že změna jednoho prvku pole neměla žádný efekt, protože byl jako parametr základního datového typu předán metodě `nastavPrvek()` hodnotou.
- Zvláštní příkaz tisku: `System.out.print("\b\b \n");` znamená, že se vytisknou dva znaky „návrat kurzoru“ a pak znak mezery následovaný odřádkováním. Celý tento trik je proto, aby se za posledním prvkem pole vymazala již vytisknutá čárka.

10.5.4 Předávání vícerozměrných polí metodám

Oproti jednorozměrným polím zde není žádný jiný rozdíl než ten, že se ve formálním parametru metody musí pomocí prázdných složených závorek udat, kolikarozměrné pole bude. Jinak je vše zcela stejné jako u jednorozměrných polí.

Příklad 107:

Ukázka tisku vícerozměrného pole předaného metodě.

Oproti příkladu z předchozí části je zde navíc ukázka, že se dá pracovat i s řádkami dvourozměrného pole. Ty se opět předávají pomocí reference, takže je možné prvky řádky trvale měnit.

```
class PraceSPolem {  
    public static void tiskniPole(int[][] p) {  
        for (int i = 0; i < p.length; i++) {  
            for (int j = 0; j < p[i].length; j++)  
                System.out.print(p[i][j] + ", ");  
            System.out.print("\b\b \n");  
        }  
    }  
  
    public static void nastavATiskniRadku(int[] r, int h) {  
        for (int i = 0; i < r.length; i++) {  
            r[i] = i + h;  
            System.out.print(r[i] + ", ");  
        }  
        System.out.print("\b\b \n");  
    }  
}  
  
public class PolePoli {  
    public static void main(String[] args) {  
        int[][] pole1 = {{4, 3, 2, 1}, {1, 2, 3, 4}};  
        int[][] pole2 = {{1}, {2, 3}, {4, 5, 6}};  
        PraceSPolem.tiskniPole(pole1);  
        PraceSPolem.tiskniPole(pole2);  
        PraceSPolem.nastavATiskniRadku(pole1[0], 5);  
    }  
}
```

Vypíše:

4, 3, 2, 1

1, 2, 3, 4

1

2, 3

4, 5, 6

5, 6, 7, 8

Cvičení:

1. Vytvořte třídu `BankovniUcet` s potřebnými datovými prvky a metodami. Ve třídě `Banka` vytvořte pole účtů, které náhodně inicializujete. Vypište celkovou hotovost v bance.
2. Vytvořte třídu `Klient`, která bude mít jako jednu členskou proměnnou referenci na objekt třídy `BankovniUcet`. Modifikujte třídu `Banka`, aby pracovala s klienty místo s účty.
3. Napište statickou metodu `void serad(int[] x, int[] y)`, která seřadí pole `x` do pole `y` vzestupně.
4. Napište statickou metodu `int sude(int[] x, int[] y)`, která zkopíruje z pole `x` do pole `y` jen prvky se sudou hodnotou a vrátí počet těchto prvků.

11 Dědičnost

11.1 Úvodní poznámky

Dědičnost představuje možnost přidat k **základní třídě** (též **bázové, rodičovské, supertřídě** nebo **rodiči** případně **předkovi**) další vlastnosti a vytvořit tak **odvozenou třídu** (**zděděnou třídu** nebo **potomka** či **dcerinou třídu**). Odvozená třída bývá specializovanější než třída základní.

Prvotní otázka ale asi je, jaký je pro dědičnost důvod. V [10.2/164] jsme totiž viděli, že instance třídy mohou být datovými prvky jiných tříd díky využití kompozice, tedy bez jakýchkoli problémů s děděním. Kompozice navíc dovoluje využít opakovaně (tj. vícekrát současně) datové vlastnosti jiné třídy (viz datum narození a datum nástupu ve třídě **Zamestnanec**).

To, co nám však kompozice nedovoluje (a dědění ano), je **změna vlastností třídy**.

Odpověď na otázku, zda tedy používat dědění nebo kompozici, je šalamounská – používat obě dvě, ale vždy tam, kde je to vhodné.

Tato rada okamžitě vyvolá další otázku: „Jak poznáme, kdy je to vhodné?“ Odpověď je velmi jednoduchá, použijeme tzv. „**JE test**“.

Vždy, když se rozhodujeme, zda použít dědění či kompozici, stačí se zeptat, zda třída eventuálně vkládaná kompozicí do jiné třídy je i touto třídou. Například „Je **datum** (nástupu) **Zaměstnancem**?“ Pokud je odpověď záporná, použijeme kompozici (tak jako v [10.2/164]).

Je-li však odpověď kladná, např. „Je **vrátný** **Zaměstnancem**?“, pak použijeme dědění, kdy z obecnější třídy **Zaměstnanec** vytvoříme děděním a přidáním dalších vlastností, které jsou specifické pro vrátného, novou třídu **Vrátný**.

Poznámka:

V opačném smyslu lze použít „**MÁ test**“, např. „Má **Zaměstnanec** **Datum** (nástupu)?“ Pokud ano, je **Datum** (nástupu) vhodným kandidátem na „**zakomponování**“ do třídy **Zaměstnanec**. □

Dědění je i významný nástroj pro vytváření ***opakovatelných*** (*reusable*) programových částí (modulů). Teorie totiž říká, že programový modul by měl být zároveň **uzavřený** i **otevřený**.

Uzavřený znamená, že pro jeho použití není třeba nic přidávat a uživatel nemá možnost provádět jeho změny – v modulu se nejlépe vyzná jeho autor.

Otevřený naopak znamená, že uživatel by měl mít možnost některé, pro něj nevhodné části změnit a nové dodat. Řešením tohoto zdánlivě rozporuplného požadavku je právě dědění.

Dědění nám v odvozené třídě umožňuje:

- vše, co bylo v základní třídě dobré, bez dalšího úsilí ponechat,
- vše, co nám chybělo, jednoduše dodat,
- vše, co se nám nelíbilo, změnit.¹

Poznámka pro programátora v C či C++:

Java zná pouze jednoduchou dědičnost, tj. každá třída může mít pouze jednoho přímého předka. Vícenásobnou dědičnost nahrazuje mechanismus rozhraní – viz [13/212]. □

11.2 Realizace dědičnosti

Musí existovat třída, která se stane rodičem, a jméno této třídy se uvede v deklaraci nové třídy za klíčové slovo **extends** (rozšiřuje).

Příklad 108:

Na následujícím, poněkud delším příkladě budou ukázány všechny základní „figle“, používané při dědění. Ze třídy **Kvadr** vytvoříme třídu **Kvadr**, protože kvádr (s nulovou hloubkou) **JE** obdélník.

Poznámka:

Až při závěrečné korektuře jsem byl jemně upozorněn na skutečnost, že z jiného pohledu na věc **MÁ** kvádr šest obdélníků. To pak přináší problém, že děděním třídy **Kvadr** od třídy **obdelnik** by jeden ze šesti obélníků získal privilegované postavení. Berte prosím proto následující příklad jako ukázku neúplně domyšlené analýzy. Používané jazykové konstrukce však tato analýza neovlivní – tzn. jsou správné. Tatáž poznámka platí i pro třídy **usecka** a **Obdelnik** v [13.6/218]. □

¹Viz též polymorfismus – [14/224].

```
class Obdelnik {  
    public int sirka;  
    public int vyska;  
  
    public Obdelnik(int sirka, int vyska) {  
        this.sirka = sirka;  
        this.vyska = vyska;  
    }  
  
    public double delkaUhlopricky() {  
        double pom;  
        pom = (sirka * sirka) + (vyska * vyska);  
        return Math.sqrt(pom);  
    }  
  
    public int hodnotaSirky() {  
        return sirka;  
    }  
}  
  
public class Kvadr extends Obdelnik {  
    public int hloubka;  
  
    public Kvadr(int sirka, int vyska, int hloubka) {  
        super(sirka, vyska);  
        this.hloubka = hloubka;  
    }  
  
    public double delkaUhlopricky() {  
        double pom = super.delkaUhlopricky();  
        pom = (pom * pom) + (hloubka * hloubka);  
        return Math.sqrt(pom);  
    }  
  
    public static void main(String[] args) {  
        Kvadr kva = new Kvadr(6, 8, 10);  
        System.out.println("Uhlopricka: " + kva.delkaUhlopricky());  
        System.out.println("Sirka je: " + kva.hodnotaSirky());  
        System.out.println("Vyska je: " + kva.vyska);  
    } }
```

Vypíše:

Uhlopricka: 14.142135623730951

Sirka je: 6

Vyska je: 8

Na třídě `Obdelnik` již není (doufejme) nic zajímavého.² Třída `Kvadr` ve své hlavičce uvádí pomocí klíčového slova `extends`, že bude dědit od třídy `Obdelnik`. To znamená, že převezme **bez změny** její proměnné instance `sirka` a `vyska` a její metodu instance `hodnotaSirky()`. Konstruktor `Obdelnik()` se nedědí, ale je ze třídy `Kvadr` využíván, ať již přímo, jako je tomu zde, nebo nepřímo.

V konstruktoru `Kvadr()` je první zajímavost – volání konstruktoru rodičovské třídy. Konstruktor rodičovské třídy se volá pomocí klíčového slova `super` a do kulatých závorek se dají skutečné parametry konstruktoru rodičovské třídy, tj. zde `Obdelnik()`.

Pozor:

Pokud konstruktor předka tímto způsobem nezavoláme, doplní si překladač volání konstruktoru bez parametrů, a v tomto případě musí takový konstruktor existovat, jinak kompilátor ohlásí chybu. Připomeňme si, že pokud jsme deklarovali konstruktor s parametry, překladač již nevytvoří implicitní konstruktor bez parametrů – viz též [8.7/128]. □

Další akcí, která je ve třídě `Kvadr`, je doplnění nových proměnných nebo metod. Zde se jedná jen o proměnnou `hloubka`.

Ve třídě `Kvadr`, ačkoliv deklaruje pouze jednu proměnnou, lze využívat **naprosto stejným přístupem** ve skutečnosti tři proměnné – `hloubka`, která byla deklarována zde, a `sirka` a `vyska`, deklarované v rodičovské třídě `Obdelnik`.

Totéž platí i o metodách instance z rodičovské třídy. Pokud je třída zděděna, může přímo využívat všechny³ metody z rodičovské třídy.

To je vidět na následujících příkazech tisku, kde se volá pomocí referenční proměnné `kva`, která je typu `Kvadr`, metoda `hodnotaSirky()` ze třídy `Obdelnik` nebo se stejným způsobem přistupuje k proměnné `vyska`.

```
System.out.println("Sirka je: " + kva.hodnotaSirky());
System.out.println("Vyska je: " + kva.vyska);
```

²Pokud je tam něco nejasného, vratte se k [8.1/120].

³Pokud nejsou `private` – viz [10.3/166].

Poslední akcí je změna toho, co nám ve třídě Kvadr ze třídy Obdelnik nevyhovuje. Jedná se o metodu delkaUhlopricky(). Tím, že jsme použili zcela stejné jméno, jako ve třídě Obdelnik a zcela stejné (zde žádné) formální parametry, došlo ve třídě Kvadr k **překrytí** (*overriding, zastínění – hiding*) metody delkaUhlopricky() deklarované ve třídě Obdelnik. To znamená, že kdykoliv budeme ve třídě Kvadr volat metodu delkaUhlopricky(), bude se volat ta, která je v ní deklarována, nikoliv ta z rodičovské třídy.⁴

Pozor:

Pokud bychom použili stejné jméno, ale jiné formální parametry, došlo by pouze k **přetížení** (*overloading*) této metody – viz [6.8/104]. □

Ihned v metodě delkaUhlopricky() je však ukázáno, jakým způsobem lze využít i překryté metody z rodičovské třídy. Lze to provést opět pomocí klíčového slova **super**, použitého nyní jako jméno objektu.

```
double pom = super.delkaUhlopricky();
```

Takto lze přistupovat k překrytým i nepřekrytým metodám i proměnným instance rodičovské třídy.

Poznámka:

Tento příklad ještě není ani zdaleka ideální. Například díky přístupovým právům **public** u proměnných **sirka** a **vyska** ve třídě **Obdelnik** lze k témtoto proměnným libovolně přistupovat nebo je měnit. Bylo by vhodné využít autorizovaného přístupu (viz [10.3/166]) a pro obě proměnné vyrobit metody **get...()** a **set...()**. □

11.3 Problémy s neimplicitními konstruktory rodičovské třídy

Při využití dědičnosti musíme mít vždy na zřeteli, že během konstrukce objektu typu potomka musí být **vždy** dána možnost, aby byl vyvolán konstruktor rodiče. Mohou nastat dva případy:

- **V rodiči je konstruktor bez parametrů nebo implicitní**
Konstruktor v potomkovi může být implicitní. Pokud ale existuje, nemusí si konstruktor potomka s voláním konstruktora rodiče dělat starosti.
- **V rodiči je konstruktor alespoň s jedním parametrem**
Konstruktor potomka musí existovat a jako svůj první příkaz musí volat pomocí **super()** konstruktor rodiče.

⁴Tb si lze snadno zapamatovat pomocí přísloví: „Bližší košile než kabát.“

Pokud neexistuje nebo nevolá konstruktor rodiče, je hlášena chyba:
 constructor Rodic() not found in class Rodic

Příklad 109:

Třída Potomek nemá žádné proměnné, takže by ani nebylo nutné, aby měla konstruktor. Protože však třída Rodic má konstruktor s parametrem, musí třída Potomek deklarovat svůj konstruktor, z něhož pomocí super() vyvolá konstruktor předka.

```
class Rodic {
    public int i;
    public Rodic(int parI) { i = parI; }
    // public Rodic() { i = 5; }
}

public class Potomek extends Rodic {
    public Potomek() {
        super(8);
    }
    public static void main(String[] args) {
        Potomek pot = new Potomek();
    }
}
```

Poznámka:

Pokud by ovšem třída Rodic měla další konstruktor bez parametru (např. ten, co je v příkladu zakomentován), nemusí třída Potomek mít žádný konstruktor.

Z toho plyne poučení, že je velmi vhodné připravit v naší třídě mimo ostatní konstruktory i jeden konstruktor bez parametrů – byť jej momentálně nepotřebujeme – protože tím ušetříme starosti budoucím potomkům této třídy. □

11.4 Nechceme, aby bylo možné metodu překrýt – finální metody

Může se stát, že považujeme některou metodu třídy za tak důležitou (dokonalou, atd.), že si nepřejeme, aby ji kdokoliv v případných zděděných třídách mohl překrýt. Tomu lze snadno zabránit pomocí klíčového slova **final**.⁵

⁵Použití **final** pro proměnné viz [3.5.1/43].

Příklad 110:

Jak zabránit překrytí metody.

```
class Rodic {  
    public int i;  
    public Rodic() { i = 1; }  
    final int getI() { return i; }  
}  
  
public class Potomek extends Rodic {  
    // int getI() { return i * 2; } // chyba  
  
    public static void main(String[] args) {  
        Potomek pot = new Potomek();  
        System.out.println("Hodnota je: " + pot.getI());  
    }  
}
```

Vypíše:

Hodnota je: 1

Poznámka:

Pokud je v rodičovské třídě metoda označena jako **final**, můžeme ji ve zděděné třídě přetížit, tzn. **final** zabrání překrytí, ale nikoliv přetížení. □

11.5 Chceme, aby bylo nutné metodu překrýt – abstraktní metody a třídy

Toto je opačný případ než předchozí ukázka. Občas se totiž může stát, zejména při psaní v hierarchické struktuře hodně nadřazených tříd, že tušíme, že bude někdy později ve zděděných třídách vhodné používat nějakou metodu.

Tuto metodu zatím nemůžeme napsat, protože neznáme konkrétní poměry v budoucí třídě potomka, ale rádi bychom „připravili půdu“ pro její použití. To je nejčastěji proto, že všechny objekty potomků bude výhodné reprezentovat pomocí referenční proměnné typované na rodičovský typ.

Pak je možné ve **zděděné třídě** vynutit naprogramování této metody použitím klíčového slova **abstract** u **metody** v **rodičovské třídě**. Takto označenou metodu pak **musí** programátor zděděné třídy v této třídě naprogramovat, jinak se program nepřeloží.

Jako častý příklad se udává knihovna grafických objektů se společným předkem. Ten je označen jako **abstraktní třída** (*abstract class*) a deklaruje **abstraktní metody** typu `vykresli()`, `posun()` atd. Tím je pak zaručeno, že každý grafický objekt, který bude získán děděním z této třídy, bude muset mít své metody pro vykreslení, posun atd. a se všemi grafickými objekty bude možno zacházet zcela stejným způsobem.

Příklad 111:

Tento příklad bude jednodušší než grafické objekty, ale principy abstraktní třídy a metody z něj budou patrné.

```
abstract class Rodic {
    public int i;
    public Rodic() { i = 1; }
    abstract int getI();
    final void setI(int novI) { i = novI; }
}

public class Potomek extends Rodic {
    int getI() { return i * 2; }
    void setI() { i = 5; } // přetížená

    public static void main(String[] args) {
//        Rodic rod = new Rodic(); // chyba
        Potomek pot = new Potomek();
        pot.setI(3);
        System.out.println("Hodnota je: " + pot.getI());
        pot.setI(); // přetížená
        System.out.println("Hodnota je: " + pot.getI());
    }
}
```

Vypíše:

```
Hodnota je: 6
Hodnota je: 10
```

Na tomto kratičkém příkladu je vidět jednak způsob konstrukce abstraktní metody, tak i abstraktní třídy. Jakmile je totiž jedna (nebo více) metoda označena jako **abstract**, pak nelze vytvořit instanci třídy a z tohoto důvodu musí být celá třída označena jako **abstract**. To znamená, že pro použití jejich ostatních kompletně naprogramovaných metod (zde `setI()`), je nutno tuto třídu zdědit a ve zděděné třídě naprogramovat všechny zděděné abstraktní metody.

Poznámka:

Všimněte si, jak vypadá abstraktní metoda:

```
abstract int getI();
```

Je uveden její návratový typ, jméno, formální parametry (zde nejsou použity), ale není uvedeno její tělo. □

Pozor:

Není pravda, že ve zděděné třídě se musí přeprogramovat všechny metody z abstraktní třídy. To je nutné jen pro metody označené v rodičovské třídě jako **abstract**. Všechny ostatní metody se mohou:

- nechat být – jako zde `setI(int noveI)`
- překrýt (nejsou-li ovšem označeny jako **final**)
- přetížit – zde `setI()`

□

Poznámka:

Může existovat abstraktní třída, která nemá žádnou abstraktní metodu. Jsou to speciální případy, kdy např. nutíme uživatele, aby naši třídu pouze zdědil, ale žádnou další činnost od něho nepožadujeme.

Toto je případ třídy `java.lang.ClassLoader` z Java Core API, která umožňuje dynamické sestavování programu. V tomto případě je abstraktní třída použita proto, aby JVM rozeznal, které třídy natáhl sám svojí zděděnou třídou a které natáhl pomocí zděděné třídy uživatele. □

11.6 Nechceme, aby bylo možné třídu zdědit – finální třídy

V [11.4/184] byl ukázán způsob, jak zabránit překrytí určité metody pomocí klíčového slova **final**. Stejně, jako je možné pomocí klíčového slova **abstract** označit metodu i třídu, je možné použít **final** i pro třídu. Taková třída se pak nedá zdědit, tj. nemůže být rodičovskou třídou a označuje se jako **finální třída** (*final class*) nebo **koncová třída**.

Poznámka:

Na rozdíl od **abstract**, kdy libovolná abstraktní metoda si vynutila označení své třídy také jako abstraktní, toto neplatí pro **final**. Označení metody (metod) jako **final** nevyneče označení celé třídy jako **final**. Samozřejmě však, je-li třída **final**, její metody se nedají překrýt prostě proto, že se třída nedá zdědit. □

Příklad 112:

Ukázka koncové třídy.

```
final class Rodic {
    public int i;
    public Rodic() { i = 1; }
    abstract int getI(); // chyba
    void setI(int novI) { i = novI; }
}

public class Potomek extends Rodic { // chyba
    ...
}
```

Poznámky:

- Je zřejmé, že označení **final** a **abstract** mají protichůdný účinek a nesmí se bez rozmyslu míchat, jako je to v tomto případě. Jediný smysl by měla abstraktní třída s metodou **final**. Tato třída by se musela zdědit, ale metoda by se nesměla překrýt – to bylo již ukázáno v příkladě v [11.5/185].
- V [11.4/184] byl uveden jeden z důvodů, proč použít finální metodu – metodu již nebylo možné vylepšit. Jako vážnější důvod, proč se používá **final** pro třídy, je to, že finální třídy mohou být nejrůznějším způsobem optimalizovány již při překladu. Protože jsou finální, není možné využít polymorfismu (viz [14/224]), který je sice z programátorského hlediska příjemný, ale z hlediska rychlosti programu pomalý. Z tohoto důvodu je mnoho tříd z Java Core API (např. `String`) označeno jako **final**.

11.7 Překrytí proměnné

Tak, jako bylo možné překrýt metodu rodičovské třídy, je možné překrýt i proměnnou této třídy. U metod měla tato akce význam – nová metoda poskytovala specializovanější služby vhodnější pro novou třídu – viz příklad v [11.2/183] s metodou `delkaUhlopriicky()`. U proměnných primitivních datových typů je význam této činnosti diskutabilní – význam má snad jen tehdy, když potřebujeme proměnnou stejného jména, ale jiného typu. Smysl má ale překrytí referenčních proměnných, čímž dosáhneme ve třídě potomka „specializace“ této členské proměnné.

Příklad 113:

Překrytí proměnné z rodičovské třídy ve třídě potomka.

```
class Rodic {
    public int i;
    static public long j;
}

public class Potomek extends Rodic {
    public long i;
    static public int j;
    public Potomek(long novel) {
        i = novel;
        super.i = 5;
    }
    public static void main(String[] args) {
        Rodic.j = 6;
        Potomek.j = 7;
    }
}
```

K překryté instanční proměnné se lze dostat v metodách instance pomocí klíčového slova **super**. V metodách třídy (míněny statické metody, jako zde `main()`) není tento přístup možný, protože statické metody nemohou používat proměnné instance.

11.8 Základem je Object

Má-li každá třída jen jedinou rodičovskou třídu, je zřejmé, že třídy jsou organizovány v jedné hierarchické dědičné posloupnosti. Otázkou pak je, jaká třída je na vrcholku pyramidy.

Jméno této třídy je známé – je to třída `java.lang.Object`. Tato třída definuje základní stav a chování všech objektů, které se vyskytnou v programu.

Poznámka:

Všechny třídy (kromě zděděných), které byly dosud uvedeny v příkladech, by bylo možné napsat i např. jako:

```
class MojeTřida extends Object {
```

Tento zápis se však nepoužívá, protože je dědění od `Object` považováno za samozřejmost, tj. neuvědeme-li předka, dosadí překladač třídu `Object`.

□

Třída `Object` nemá žádné z vnějšku přístupné datové prvky. Má ale devět metod (přetížené metody nepočítáme), z nichž některé jsou finální a některé si uživatel přeprogramovat (překrýt je).

Překrýt lze metody:

- `clone()`,
- `equals()`,
- `hashCode()`,
- `finalize()`,
- `toString()`.

Finální jsou metody:

- `getClass()`,
- `notify()`,
- `notifyAll()`,
- `wait()`.

Metoda `finalize()` byla zmínována v [8.14/141].

Smysl a způsob použití metody `toString()` byl vysvětlen v [9.8/155].

Metody `notify()`, `notifyAll()` a `wait()` budou popsány v části týkající se vláken – [20.6.7/333].

Zbývající metody budou uvedeny v následujících částech. Pokud se ale Javu teprve učíte a ještě nevíte nic o výjimkách a rozhraních, uděláte nejlépe, když zbytek této kapitoly přeskočíte a vrátíte se k němu po přečtení celé knížky.

11.8.1 Metoda `clone()`

Tato metoda slouží ke **klonování**, tedy vytváření zcela identických kopí jedné instance. Po provedení klonování jsou tyto instance na sobě nezávislé.

To může být v mnohých případech velmi užitečná vlastnost objektu, například když nějaký objekt složitě převedeme do určitého stavu, lze pak jednoduše (kopírováním) vytvořit tentýž objekt.

Z prvního pohledu je použití jednoduché – každý objekt je potomkem `Object`, tedy musí vlastnit metodu `clone()` a tím ji může použít. Zdánlivě jediná vada na této jednoduchosti je, že `clone()` může vyvolat výjimku `CloneNotSupportedException`, kterou musíme postupy popsánymi v [16/241] ošetřit. Je to o to zálužnější, že při neošetření výjimky překladač nenalezne chybu – to je ukázáno v následujícím příkladě.

1.8. Základem je Object

Príklad 114:

Nesprávné použití klonování.

```
public class Klon1 {
    int i;

    Klon1(int i) { this.i = i; }

    public static void main(String[] args) throws
        CloneNotSupportedException {
        Klon1 kopie, orig = new Klon1(5);
        kopie = (Klon1) orig.clone();
        System.out.println("orig: " + orig.i);
        System.out.println("kopie: " + kopie.i);
    }
}
```

Překlad proběhne bez chyb, zkusíme-li si však tento program spustit, bude vyvolána výjimka `CloneNotSupportedException`. Je to z toho důvodu, že `clone()` je připraveno pro třídu `Object`, která samozřejmě nemůže nic vědět o proměnných své zděděné třídy. Z tohoto důvodu je nutné metodu `clone()` v naší třídě překrýt. Dále je nutné, aby naše třída implementovala rozhraní `Cloneable`.

Poznámka:

Rozhraní `Cloneable` je zvláštní rozhraní, protože nemá žádnou metodu (tedy nemuselo by se vlastně ani implementovat). Ovšem implementovat ho je nezbytné, protože když to neuděláte, bude vyhozena výjimka `CloneNotSupportedException`.

Toto je obecný⁶ účinný způsob, jak vynutit překrytí metody z rodičovské třídy ve třídě potomka, aniž by bylo nutné označit tuto metodu v rodičovské třídě jako abstraktní. □

V překryté metodě `clone()` je nutné vytvořit nový objekt této třídy nikoliv pomocí `new` a konstruktoru, ale pomocí volání `super.clone()` přetypovaného na naši třídu. Pak je nutné překopírovat jednu po druhé obsahy všech proměnných, které naše třída obsahuje.

⁶Můžete ho použít i vy – návod viz v programu na disketě `MusiEdit.java`.

Příklad 115:

Již funkční příklad vypadá takto:

```
public class Klon2 implements Cloneable {
    int i;

    Klon2(int i) { this.i = i; }

    protected Object clone() {
        Klon2 k = null;
        try {
            k = (Klon2) super.clone();
            k.i = this.i;
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return k;
    }

    public static void main(String[] args) {
        Klon2 kopie, orig = new Klon2(5);
        kopie = (Klon2) orig.clone();
        System.out.println("orig: " + orig.i);
        System.out.println("kopie: " + kopie.i);
    }
}
```

Vypíše:

```
orig: 5
kopie: 5
```

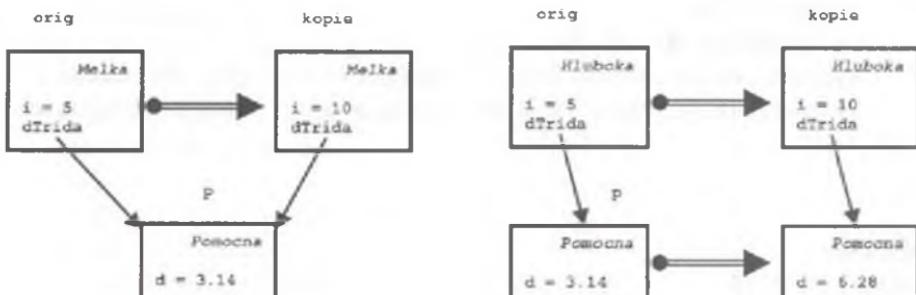
11.8.1.1 Problém mělké a hluboké kopie

Když budete zkoušet výše popsaný způsob na třídy, jejichž proměnné jsou jen základní datové typy, bude vše v pořádku. Pokud ale bude některá proměnná referenční proměnou (tj. odkazuje na objekt jiné třídy), nebude předchozí postup správný. Tímto postupem vytváříme totiž tzv. **mělkou kopii** (*shallow copy*), kdy originál i kopie mají společný podobjekt.

Můžete se ale setkat se situací, kdy originál i kopie mají mít skutečně jeden sdílený objekt, např. ve Windows odkazují dva zástupci na jeden a tentýž program. V tomto případě je použití mělké kopie naprosto správné.

Příklad 116:

Problematiku si ukážeme na příkladu, který využívá třídu Pomocna, která obsahuje pouze jednu proměnnou. Na objekt třídy Pomocna odkazuje referenční proměnná dTrida ze třídy Melka. V metodě `clone()` zkopírujeme obsahy všech proměnných, takže na první pohled se zdá být vše v pořádku. Problém však nastane, když v originálu změníme hodnoty proměnných. Proměnná `i` se změní skutečně jen v originálu, ale proměnná `d` bude změněna v originálu i v kopii, protože objekt ve kterém je uložena, existuje pouze jednou. Z originálu i z kopie na něj vedou pouze odkazy. Na následujícím obrázku je uvedeno i správné řešení pomocí **hluboké kopie** (viz další příklad), aby bylo možné oba přístupy snadno porovnat.



```

class Pomocna {
    double d;
    Pomocna(double d) { this.d = d; }
}

public class Melka implements Cloneable {
    int i;
    Pomocna dTrida;

    Melka(int i, Pomocna dt) { this.i = i; dTrida = dt; }

    protected Object clone() {
        Melka k = null;
        try {
            k = (Melka)super.clone();
            k.i = this.i;
            k.dTrida = this.dTrida;
        }
    }
}

```

```

        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return k;
    }

    public static void main(String[] args) {
        Pomocna p = new Pomocna(3.14);
        Melka kopie, orig = new Melka(5, p);
        kopie = (Melka) orig.clone();
        System.out.println("orig: " + orig.i + "; " + orig.dTrida.d);
        System.out.println("kopie: " + kopie.i + "; " + kopie.dTrida.d);
        orig.i = 10;
        orig.dTrida.d = 6.28;
        System.out.println("orig: " + orig.i + "; " + orig.dTrida.d);
        System.out.println("kopie: " + kopie.i + "; " + kopie.dTrida.d);
    }
}

```

Vypíše:

```

orig: 5; 3.14
kopie: 5; 3.14
orig: 10; 6.28
kopie: 5; 6.28

```

Příklad 117:

Správným řešením je vytvoření tzv. **hluboké kopie** (*deep copy*), kdy pro každou referenční proměnnou musíme vytvořit nový objekt a překopírovat jeho obsah. V případě třídy Pomocna by to bylo možné provést pomocí `new` a konstruktoru. Mnohem bezpečnější způsob ale je, když třída Pomocna bude také implementovat rozhraní `Cloneable` a metodu `clone()`, byť to v tomto jednoduchém případě připomíná použití „kanónu na vrabce“.

```

class Pomocna implements Cloneable {
    double d;
    Pomocna(double d) { this.d = d; }

    protected Object clone() {
        Pomocna k = null;

```

```
try {
    k = (Pomocna)super.clone();
    k.d = this.d;
}
catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
return k;
}

public class Hluboka implements Cloneable {
    int i;
    Pomocna dTrida;
    Hluboka(int i, Pomocna dt) { this.i = i; dTrida = dt; }

    protected Object clone() {
        Hluboka k = null;
        try {
            k = (Hluboka)super.clone();
            k.i = this.i;
            k.dTrida = (Pomocna)dTrida.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return k;
    }

    public static void main(String[] args) {
        Pomocna p = new Pomocna(3.14);
        Hluboka kopie, orig = new Hluboka(5, p);
        kopie = (Hluboka)orig.clone();
        System.out.println("orig: "+orig.i+" "+orig.dTrida.d);
        System.out.println("kopie: "+kopie.i+" "+kopie.dTrida.d);
        orig.i = 10;
        orig.dTrida.d = 6.28;
        System.out.println("orig: "+orig.i+" "+orig.dTrida.d);
        System.out.println("kopie: "+kopie.i+" "+kopie.dTrida.d);
    }
}
```

Vypíše:

```
orig: 5; 3.14
kopie: 5; 3.14
orig: 10; 6.28
kopie: 5; 3.14
```

11.8.2 Metoda equals()

Pomocí této metody lze zjistit, zda se dva objekty rovnají. Jednoduchý výraz „rovnají se“ je v dokumentaci API rozveden do pěti přesně specifikovaných podmínek, které si lze tamtéž přečíst.

Ve třídě `Object` je `equals()` implementována nejpřísnějším možným způsobem, a to tak, že porovnávané objekty musejí být totožné, tj. porovnávají se reference. Pokud mají jen stejný obsah, pak se za stejně nepovažují.

Poznámka:

To znamená, že použití `equals()` má stejný význam jako operátor `==`



Tento nejpřísnější způsob však může být leckdy na závadu, proto je možné metodu `equals()` překrýt. To některé třídy dělají, přičemž většinou pak `equals()` pracuje tak, že vrací `true`, když se rovnají obsahy objektů.

Poznámka:

Chcete-li překrýt metodu `equals()`, je nutné **současně** překrýt i metodu `hashCode()`, protože: „Zjistí-li se použitím `equals()`, že jsou dva objekty stejné, musí metoda `hashCode()` vracet stejné číslo pro oba dva.“



Příklad 118:

Porovnejte výsledky dosažené pomocí tříd `Pomocna` a `Double`⁷ v následujícím případě.

```
class Pomocna {
    double d;
    Pomocna(double d) { this.d = d; }
}
```

⁷ Informace o této třídě naleznete v [9.6/153].

```

public class Rovnost {
    public static void main(String[] args) {
        Pomocna p1 = new Pomocna(3.14);
        Pomocna p2 = new Pomocna(3.14);
        if (p1.equals(p2) == false)
            System.out.println("p1 != p2");

        Pomocna p3 = p1;
        if (p1.equals(p3) == true)
            System.out.println("p1 == p3");

        Double d1 = new Double(3.14);
        Double d2 = new Double(3.14);
        if (d1.equals(d2) == true)
            System.out.println("d1 == d2");
    }
}

```

Vypíše:

```

p1 != p2
p1 == p3
d1 == d2

```

11.8.3 Metoda hashCode()

Zjednodušeně řečeno, metoda `hashCode()` vrací pro každý objekt číslo typu `int`, které je po dobu života objektu stále stejné, tj. nemění se při změně stavu objektu.⁸ Je třeba zdůraznit, že toto číslo není jedinečné, ale pro stejné objekty může být stejné – viz dále příklad a [11.8.2/196].

Metoda `hashCode()` se využívá k nejrůznějším účelům, například spolupracuje s již zmíněnou metodou `equals()` nebo s metodami třídy `java.util.Hashtable`.

Poznámka:

Pokud metodu `hashCode()` překryjete⁹, může vracet libovolné `int` číslo, např. i konstantu. Není to však rozumný postup. Očekává se totiž, že bude pro různé (obsahově) objekty vracet různá čísla. □

⁸Není ale zaručeno, že při opakování spuštění téhož programu bude toto číslo stále stejné.

⁹Viz program `Hash2.java`.

Příklad 119:

Metoda `hashCode()` vrací pro různé objekty čísla různá a pro jiné stejná.

```
class Pomocna {  
    double d;  
    Pomocna(double d) { this.d = d; }  
}  
  
public class Hash {  
    public static void main(String[] args) {  
        Pomocna p1 = new Pomocna(3.14);  
        Pomocna p2 = new Pomocna(3.14);  
        System.out.println("p1: " + p1.hashCode());  
        System.out.println("p2: " + p2.hashCode());  
        p1.d = 6.28;  
        System.out.println("p1: " + p1.hashCode());  
  
        Integer i1 = new Integer(5);  
        Integer i2 = new Integer(5);  
        Byte b = new Byte((byte) 5);  
        System.out.println("i1: " + i1.hashCode());  
        System.out.println("i2: " + i2.hashCode());  
        System.out.println("b: " + b.hashCode());  
    }  
}
```

Vypíše např.:

```
p1: 1747152  
p2: 1747151  
p1: 1747152  
i1: 5  
i2: 5  
b: 5
```

11.8.4 Metoda `getClass()`

Tato finální metoda dokáže poskytnout za běhu programu aktuální informace o libovolném objektu. Metoda `getClass()` vrací objekt třídy `Class`, kdy třída `Class` je přímý potomek třídy `Object`. Instance `Class` zahrnují vše, co se v programu Javy objeví – od primitivních datových typů až po pole. Výhodou metody `getClass()` je tedy to, že pomocí ní a metod třídy

Class dokážeme **za běhu programu** zjistit velké množství informací o jakémkoliv dostupném objektu.

Poznámka:

V začátcích programování v Javě většinu těchto informací pravděpodobně nevyužijeme, ale je dobré do budoucna vědět, že takováto možnost existuje. V následujících příkladech však bude často používána metoda `getName()`, která vrátí jméno třídy. □

Příklad 120:

Zjištění jména třídy za běhu programu.

```
class Rodic {
    public int i;
    public Rodic() { i = 1; }

}

public class Potomek extends Rodic {
    public static void main(String[] args) {
        Potomek dite = new Potomek();
        System.out.println("Jmeno tridy je: " +
                           dite.getClass().getName());
        System.out.println("Jmeno rodicovske tridy je: " +
                           dite.getClass().getSuperclass().getName());
    }
}
```

Vypíše:

```
Jmeno tridy je: Potomek
Jmeno rodicovske tridy je: Rodic
```

Dobré rady:

- Změna v deklaraci třídy předka se automaticky promítne díky dědičnosti i do odvozených tříd potomků.
- Pomocí referenční proměnné na potomka se dostanete po přetypování na všechny metody předka (kromě `private`). Naopak z reference na předka se dostanete jen na metody potomka, které již byly známé v předkovi.

Běžné chyby:

- Chcete-li v potomkovi **překrýt** metodu předka, musí mít zcela stejnou hlavičku.
- Chcete-li v potomkovi **přetížit** metodu předka, musí mít rozdílnou hlavičku – nestačí jen jiný návratový typ.
- V konstruktoru potomka musí být `super()` prvním příkazem (pokud ho vůbec uvedeme).
- Pokud potomek zastiňuje metodu předka, často na začátku volá pomocí `super.` metodu předka, ke které pak přidá další funkcionality. Zapomenutí na `super.` znamená rekurzivní volání metody potomka.
- Pomocí `super.` se lze dostat jen o jednu úroveň hierarchie výše. Nelze tedy volat metodu „dědečka“ pomocí `super.super.ctiInt()`.
- Můžete vytvořit referenci na abstraktní třídu (a v polymorfizmu se to používá často), ale nemůžete vytvořit pomocí `new` objekt abstraktní třídy.
- Je-li byť jen jediná metoda označena jako **abstract**, musí být jako **abstract** označena celá třída.
- Potomek nemůže přistupovat k **private** prvkům předka.

Cvičení:

1. Vytvořte třídu `Kolo` s proměnnými `vyrobce`, `cena` a `boolean maPrehazovacku`. Doplňte do této třídy příslušné metody a konstruktory. Děděním vytvořte třídu `HorskeKolo`, kterému doplňte členské proměnné `int prevodyVzadu`, `prevodyVpredu`. Zajistěte, aby konstruktor správně vytvářel objekty této třídy (např. proměnná `maPrehazovacku` musí být vždy `true`).
2. Vytvořte abstraktní třídu `JednostopeVozidlo` s proměnnou `String druh` a abstraktní metodou `void vypisCoJsiZac()`, která bude v odvozených třídách vypisovat obsah řetězce `druh` a další informace, např. `vyrobce`, `cenu` atd. Od této třídy odvodte třídu `Kolo` a třídu `Motocykl` (např. s proměnnou `obsahNadrze`).
3. Ve třídě `HorskeKolo` překryjte metodu `void vypisCoJsiZac()` tak, aby vypisovala i počty převodů.

12 Balíky

Na **balíky** (*packages*) se lze zjednodušeně dívat jako na knihovny tříd. Vytváření knihoven je u většiny programovacích jazyků ponecháno na překladači, knihovním programu (*librarian*), sestavovacím programu a zadávěcí konkrétního operačního systému na konkrétním počítači (tj. platformě). Java, která je na platformě nezávislá, má možnosti vytváření knihoven zakomponované již do jazyka.

Systém balíků, které dává Java k dispozici, slouží k vytváření nových **prostorů jmen** (*name-spaces*) a umožňuje přehledné vytváření knihoven tříd a rozhraní.

Tím, že jsou třídy uloženy v konkrétním balíku, máme možnost:

- lehce určit, které třídy a rozhraní k sobě patří,
- zabránit konfliktům v pojmenování tříd,
- omezit přístupová práva ke třídám, metodám a proměnným.

Každý balík je obvykle reprezentován adresářem, který obsahuje přeložené zdrojové soubory (.class). Vnořováním adresářů vzniká hierarchie balíků. Jména adresářů odpovídají jménům balíků.

Cesta k balíkům udává systémová proměnná **CLASSPATH**, která standardně obsahuje cestu k balíkům Java Core API a do aktuálního adresáře.

Pro přístup ke třídám, statickým metodám a statickým proměnným se používá tečkové notace, např.:

<code>java.lang.Math</code>	přístup ke třídě <code>Math</code>
<code>java.lang.Math.PI</code>	přístup ke statické konstantě <code>PI</code> ve třídě <code>Math</code>
<code>java.lang.Math.sqrt()</code>	přístup ke statické metodě <code>sqrt()</code> ve třídě <code>Math</code>

Jméno včetně balíku představuje **úplné jméno** (*fully qualified name*), které **můžeme** kdykoliv použít, chceme-li zabránit nejednoznačnostem. Za situace, kdy nepoužíváme **import balíků** (viz dále) úplné jméno **musíme** použít.

12.1 Import balíků

Jak je možné, že jsme ve všech dosud uváděných programech nikde nemuseli nepoužít **plně kvalifikované jméno**? Je to proto, že Java umožňuje používat **neúplná jména** (*simple name*), použitím velmi jednoduchého způsobu. Pomocí klíčového slova **import** se rozšíří oblast platnosti identifikátorů na požadovanou třídu nebo balík.

V každém zdrojovém textu programu je vždy implicitně proveden příkaz:

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

Tímto příkazem se zpřístupní všech 26 tříd, 3 rozhraní a 39 tříd výjimek z tohoto balíku, takže v programu bude možné používat jejich neúplná jména.

Tak se např. zkrátí `java.lang.Math.PI` na `Math.PI` a `java.lang.Math.sqrt()` na `Math.sqrt()`.

Poznámka:

Ve skutečnosti jsou automaticky importovány tři balíky:

1. `java.lang` – standardně importovaný balík,
2. `default` – toto je balík, ve kterém jsou všechny třídy, které nepatří do žádného balíku (nemají na svém začátku uvedeno klíčové slovo `package` – viz dále),
3. **aktuální balík** – ten, který má shodné jméno se jménem, které je uvedeno za klíčovým slovem `package` (viz [12.2/203]) – to se nachází na samém začátku souboru, v němž je příslušná třída.



Někdy však nepotřebujeme zpřístupnit všechny třídy, rozhraní atd. z balíku, ale stačí nám pouze jedna konkrétní třída. V tomto případě se tato třída napiše na konec celého jména balíku. Je to ovšem velmi řídký případ, protože je jednodušší importovat celý balík.

Například pokud bychom zadali příkaz:

```
import java.io.FileReader;
bylo by možné použít příkaz:
FileReader f = new FileReader("a.txt");
```

ovšem pro třídu `LineNumberReader`, která je také z balíku `java.io`, bylo by nutné použít její plně kvalifikované jméno, tedy:

```
java.io.LineNumberReader lf = new java.io.LineNumberReader(f);
```

Příklad 121:

Ukázka nepříliš šťastného importu pouze jedné třídy – význam použitých tříd viz [18.4.2/278].

```
import java.io.FileReader;

public class Balik {
    public static void main(String[] args)
        throws java.io.IOException {
        FileReader f = new FileReader("a.txt");
        java.io.LineNumberReader lf = new
            java.io.LineNumberReader(f);
        System.out.println(lf.readLine());
        f.close();
    }
}
```

Pozor:

Příkaz typu: `import java.io;`

kdy se pokoušíme importovat balík, ne jeho obsah, není možný a překladač hlásí chybu: `class required, but package found`

Stejně tak nefungují pokusy importovat jen třídy začínající např. písmenem F

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



12.2 Vytváření balíků

Java umožňuje programátorovi vytvářet vlastní balíky. Pro tento účel stačí jen na úplném začátku zdrojového textu uvést klíčové slovo **package** následované jménem balíku, např.:

```
package editor;
```

Překlad pak proběhne běžným způsobem a po něm je třeba zajistit, aby přeložené soubory (.class) byly uloženy v adresáři pojmenovaném editor.

Příklad 122:

Ukázka vytvoření vlastního balíku.

```
package editor;

public class Balik {
    public static void main(String[] args) {
        System.out.println("Pokus s balíkem");
    }
}
```

Pozor:

Při spouštění je nutno zadat jako oddělovač adresářů .. (tečku), nikoliv \ nebo /.

Celý postup si můžete prohlédnout na následujícím obrázku, ve kterém byly vynechány pouze nedůležité informace vypisované při příkazu dir.

```
D:\JAVA\ujj\baliky>dir
29.02.00 10:02                               137 Balik.java

D:\JAVA\ujj\baliky>javac Balik.java

D:\JAVA\ujj\baliky>dir
29.02.00 10:02                               426 Balik.class
29.02.00 10:02                               137 Balik.java

D:\JAVA\ujj\baliky>mkdir editor

D:\JAVA\ujj\baliky>copy Balik.class editor
      1 file(s) copied.

D:\JAVA\ujj\baliky>del Balik.class

D:\JAVA\ujj\baliky>dir
29.02.00 10:02                               137 Balik.java
29.02.00 10:03                               <DIR>          editor

D:\JAVA\ujj\baliky>java editor.Balik
Pokus s balíkem

D:\JAVA\ujj\baliky>
```

Pozor:

Všechny třídy, které nemají specifikován balík, patří automaticky do jednoho *implicitního balíku* (*default package*). To znamená, že kdokoliv má téměř neomezený přístup k metodám a proměnným jednotlivých tříd – podrobně viz [12.3/206].

Obecně platí, že do implicitního balíku ukládáme pouze naše pokusné programy, případně malé programy. Pro větší programy je téměř nutností vytvořit příslušný balík. □

12.2.1 Celosvětově platná konvence pro pojmenování

Aby se zabránilo kolizi jmen v rámci celého světa, je doporučována konvence v pojmenování balíku vycházející z doménového jména na Internetu. Toto jméno se zapíše později. Takže např. na Západočeské univerzitě by jméno balíku editor bylo: cz.zcu.editor

Pokud by hrozilo ; -), že na univerzitě bude více programátorů a použijí stejné jméno balíku (editor) pro své programy, přidá se ještě identifikátor oddělení, např.: cz.zcu.kiv.editor což znamená, že balík editor vznikl na Katedře informatiky a výpočetní techniky Západočeské univerzity v České republice.

Pokud ani to nebude stačit, pak lze použít další „zjemnění“:

```
cz.zcu.kiv.herout.editor
```

Příklad 123:

Vytvoření balíku se jménem platným celosvětově.

```
package cz.zcu.kiv.herout.editor;
public class BalikDlouhy {
    public static void main(String[] args) {
        System.out.println("Pokus s dlouhym balikem");
    }
}
```

Na obrázku si můžete prohlédnout způsob překladu a spuštění.

29.02.00 10:02	137 Balik.java
29.02.00 10:28	169 BalikDlouhy.java

```
D:\JAVA\ujj\baliky>mkdir cz\zcu\kiv\herout\editor
D:\JAVA\ujj\baliky>copy BalikDlouhy.class cz\zcu\kiv\herout\editor
1 file(s) copied.

D:\JAVA\ujj\baliky>del BalikDlouhy.class
D:\JAVA\ujj\baliky>java cz.zcu.kiv.herout.editor.BalikDlouhy
Pokus s dlouhym balikem

D:\JAVA\ujj\baliky>
```

Poznámka:

Tento jednoznačný způsob se ale z důvodů lenosti (je nutné připravit mnoho vnořených adresářů) či neznalosti používá velmi zřídka. □

12.3 Přístupová práva

Balíky mají ještě jednu důležitou funkci, a tou je kontrola přístupových práv. V [10.3/166] už byla tato problematika nastíněna, zde bude proveden podrobný rozbor.

Před jakoukoliv proměnnou, konstantu a metodou (nehledě na to, zda se jedná o proměnnou/metodu instance či třídy) lze uvést tři klíčová slova (specifikátory), označující tři možná přístupová práva. Poslední možnost je neuvést žádný specifikátor a ten pak bude implicitní – v literatuře je často označován jako „přátelský“ (*friend*).

V následující tabulce je uveden souhrnný přehled možností, odkud jsou přístupné metody a proměnné v závislosti na jejich specifikátoru.

specifikátor	<i>v téže třídě</i>	<i>v jiné třídě téhož balíku</i>	<i>v podtřídě téhož balíku</i>	<i>v podtřídě jiného balíku</i>	<i>v jiné třídě jiného balíku</i>
private	ano	ne	ne	ne	ne
neuvedeno	ano	ano	ano	ne	ne
protected	ano	ano	ano	ano	ne
public	ano	ano	ano	ano	ano

Z tabulky např. vyplývá, že použijeme-li pro proměnnou specifikátor **private**, bude tato proměnná přístupná pouze a jedině z vnitřku své třídy. Z libovolných jiných tříd není k této proměnné žádný přístup.

12.3.1 Specifikátor **private**

Specifikátor **private** slouží k určení nejvíce restriktivního (omezujícího) přístupu. K takto označeným proměnným či metodám se smí přistupovat pouze uvnitř třídy, ve které jsou deklarovány. Jakýkoliv přístup zvnějšku (tzn. pomocí referenční proměnné) není možný. Stejně tak není možný přístup ze zděděné třídy.

Tento specifikátor se typicky používá u proměnných, protože tím se zajistí **autorizovaný přístup** k datům. Jinak řečeno – při správném objektovém návrhu by měly být všechny proměnné označeny jako **private**.

Pokud se **private** použije pro metody, pak se jedná o nějaké pomocné metody, které mají smysl opravdu jen v rámci třídy. Je značně nevhodné označit všechny metody jako **private**, protože pak třídu nemůžete prakticky použít. Stejně tak je nevhodné označovat jako **private** konstruktoru.

Pozor:

Specifikátor **private** nelze použít pro označení třídy, např.: □

```
private class APriv { // nelze
```

Příklad 124:

Ukázka funkce specifikátoru **private**.

```
class APriv {
    private int i;
    private int getI() { return i; }
}

class BPriv {
    int j;
    BPriv() {
        APriv a = new APriv();
        a.i = 1;           // chyba
        j = a.getI();     // chyba
    }
}
```

Stejně tak nejsou přístupné ze třídy potomka:

```
class ABPriv extends APriv {
    int j;
    ABPriv() {
        i = 1;           // chyba
        j = getI();     // chyba
    }
}
```

Jediná možnost, jak se dostat k takto označeným proměnným či metodám z vnějšku, je přístup k jiné instanci z téže třídy. Je to z toho důvodu, že třída sama před sebou nemusí mít žádné tajnosti.

Příklad 125:

Specifikátor **private** nezabrání přístupu z jiného objektu téže třídy.

```
class CPriv {
    private CPriv jinaC;
    private int i;
```

```

CPriv(int i, CPriv jina) {
    this.i = i;
    jinaC = jina;
}
private int getI() { return i; }
public void porovnani() {
    if (i == jinaC.i)
        System.out.println(i + " == " + jinaC.i);
    else
        System.out.println(i + " != " + jinaC.i);
}
}

public class TestCPriv {
    public static void main(String[] args) {
        CPriv c1 = new CPriv(1, null);
        CPriv c2 = new CPriv(2, c1);
        CPriv c3 = new CPriv(1, c1);
        c2.porovnani();
        c3.porovnani();
    }
}

```

Vypíše:

2 != 1
1 == 1

12.3.2 Specifikátor `protected`

Specifikátor `protected` je mnohem méně restriktivní než `private` – je možný přístup z jakékoliv odvozené třídy a také přístup z libovolné třídy v tomtéž balíku.

Rozdíl mezi velmi podobnými `protected` a `neuvedeno` je v tom, že `protected` dovoluje dědit atributy třídy i třídám z jiného balíku.

Typické použití `protected` je pro proměnné, které nemají být viditelné zvnějšku, ale je vhodné, aby šly použít v odvozené třídě.

Použití `protected` pro metody má význam v tom, že tato metoda jde používat v odvozené třídě, ale navenek bude stále nepřístupná.

Pozor:

Specifikátor **protected** nelze použít pro označení třídy, např.:

```
protected class AProt { // nelze
```

□

Příklad 126:

Ukázka funkce specifikátoru **protected**.

```
class AProt {
    protected int i;
    protected int getI() { return i; }
}

class ABProt extends AProt {
    int j;
    ABProt() {
        i = 1;           // v pořádku
        j = getI();     // v pořádku
    }
}

class BProt {
    int j;
    BProt() {
        AProt a = new AProt();
        a.i = 1;         // v pořádku
        j = a.getI();   // v pořádku
    }
}
```

Poznámka:

Jak je možné, že je v tomto případě přístup v pořádku, když **BProt** nedědí od **AProt**? Toto je poměrně záludná vlastnost, která ukazuje, jak je vhodné vytvářet vlastní balíky.

Protože **AProt** i **BProt** nemají uveden žádný balík, patří obě do implicitního (*default*) balíku a ze stejného balíku¹ je k proměnným a metodám označeným jako **protected** neomezený přístup.

□

Poznámka pro programátora v C či C++:

Pozor na odlišnou funkci specifikátoru **protected**, kdy v Javě jsou takto označené složky přístupné v celém balíku a v C++ jen v podtřídách.

□

¹To, že je zrovna implicitní, nevadí. Důležité je, že je stejný.

12.3.3 Specifikátor public

Tento specifikátor říká, že jsou proměnné i metody volně přístupné komukoliv. Toto přístupové právo je běžné pro metody, ale pro proměnné by mělo být používáno po důkladně rozvaze. Je-li totiž k proměnným neomezený přístup zvenčí, je porušen princip ***autorizovaného přístupu*** k datům (viz [10.3/166]).

Pomocí **public** se také označuje třída, má-li být přístupná i mimo svůj balík. Pozor však na skutečnost, že každá **public** třída musí být v samostatném souboru.

12.3.4 Specifikátor neuveden (je „přátelský“)

Toto přístupové právo je implicitní a omezuje přístupnost jen na jeden balík.² Pokud se pohybujeme v jednom balíku, máme k atributům bez specifikátoru neomezený přístup. Jakmile jsme však „cizi“ (tj. z jiného balíku), ztrácíme jakoukoliv možnost přístupu, a to i včetně dědění.

Toto pravidlo platí i pro třídy – třída bez modifikátoru **public** se nedá v jiném balíku zdědit.

Tímto je určeno použití „přátelského“ specifikátoru – pro **pomocné** třídy, metody a proměnné. Pokud uvažujeme o tom, že by mělo být možné proměnné či metody zdědit i z jiného balíku, musíme použít **protected** nebo pro třídy **public**.

Otázkou je, proč dědit třídy z jiných balíků? Odpověď je prostá – protože již existuje mnoho hotových tříd, např. v balících Java Core API, které zdědit musíme, pokud chceme měnit jejich funkčnost.

Tento problém ale bude týkat zejména našich tříd, kdy si v průběhu programování postupně vytvoříme několik vlastních balíků, z nichž později budeme chtít dědit v nových aplikacích. Je vhodné s touto možností dopředu počítat³ a u každé proměnné nebo metody raději jednoznačně specifikovat přístupová práva pomocí **private**, **protected** či **public**. Je to z toho důvodu, že při vytváření třídy nejlépe víme, co si ke které proměnné či metodě můžeme z vnějšku dovolit. Je pak zbytečné později ztrácat čas tímto bádáním.

²Pozor na přístup z implicitního balíku, jak to již bylo ukázáno v [12.3.2/209].

³Dopředu nedokážeme totiž nikdy s jistotou říci, že naší třídu, původně plánovanou jen pro jeden program, někdy v budoucnu nepoužijeme pro jiný program.

12.4 Při dědění nelze zeslabit přístupová práva

Na všechny dosud zmíněné specifikátory je nutné brát ohled při dědění. Java totiž nepovoluje zeslabit (tj. omezit) ve zděděné třídě přístupová práva. To prakticky znamená, že když je v rodičovské třídě nějaká metoda označena jako **public**, musí být ve třídě potomka také **public**. V opačném případě hlásí překladač chybu:

The method xyz() declared in class ABPrava cannot override the method of the same signature in class APrava. The access modifier is made more restrictive.

Poznámka:

Toto sympatické pravidlo vlastně říká, že pokud autor třídy dal něco k dispozici veřejnosti, nemůže jiný autor využitím pouhého dědění změnit rozhodnutí původního autora. Jinak řečeno – vše, co bylo jednou označeno jako veřejné, musí po celou dobu zůstat veřejné. □

Třídy APrava a ABPrava (viz na disketě) ukazují kombinace všech možností. Úpravou ABPrava a jejím překladem lze dostat následující tabulku, ze které je vidět, jaká musí být nastavena práva ve třídě potomka v závislosti na jejich nastavení ve třídě rodiče.

rodič	potomek			
	private	neuvezeno	protected	public
private	ano	ano	ano	ano
neuvezeno	ne	ano	ano	ano
protected	ne	ne	ano	ano
public	ne	ne	ne	ano

Z této tabulky je také vidět, pořadí od nejvíce k nejméně restriktivním právům. Je to: **private** → **neuvezeno** → **protected** → **public**

Co to prakticky znamená? Pokud je např. ve třídě A označena proměnná nebo metoda jako **protected** a my ji chceme překrýt ve třídě AB a zapomeneme napsat specifikátor **protected**, ohlásí překladač chybu. Nevedený specifikátor by se totiž pokusil zeslabit přístupová práva.

Na druhé straně ale nic nebrání tomu, abychom z původně **protected** metody třídy A udělali ve třídě AB metodu **public**. Je třeba si ale uvědomit, že to provádíme na vlastní riziko a s naší vlastní metodou. Ve třídě AB jsme totiž museli původní metodu překrýt, čili přeprogramovat.

Poznámka:

Zmiňovaná restrikce ale neplatí na úrovni tříd. Třída **public**, může být zděděna třídou bez označení a naopak. □

13 Rozhraní (interface)

Java neumožňuje **vícenásobnou dědičnost** (*multiple inheritance*) známou např. z objektových jazyků C++, ADA atp. Vícenásobná dědičnost znamená, že třída vznikla děděním z více rodičovských tříd najednou. Tato možnost přináší různé výhody, ale také různé problémy¹. Protože autoři Javy byli přesvědčení, že problém je více než výhod, vícenásobnou dědičnost neumožnili. Jako (částečnou) náhradu za ní má ale Java možnost používat **rozhraní**.²

Rozhraní definuje soubor metod, které ale v něm **nejsou implementovány**, tj. v deklaraci rozhraní je pouze hlavička metody, stejně jako je to u abstraktní metody. Třída, která toto rozhraní **implementuje** (tj. jakoby zdědí), **musí** překrýt (v tomto případě lépe řečeno **implementovat**) **všechny** jeho metody.

Rozhraní ale není totéž co abstraktní třída (se všemi metodami abstraktními), protože:

- rozhraní nemůže deklarovat žádné proměnné,
- třída může implementovat **více než jedno** rozhraní,
- rozhraní je nezávislé na dědičné hierarchii tříd, tj. naprosto odlišné třídy (z hlediska stromu dědičnosti) mohou implementovat stejné rozhraní.

Použití rozhraní je výhodné v případech, kdy:

- Chceme třídě pomocí implementace rozhraní **vnutit** zcela konkrétní metody.
- Vidíme jednoznačně podobnosti v různých třídách, ale začlenit tyto podobnosti do vlastností tříd pomocí dědění by vyžadovalo zkonstruovat jejich předka, což je činnost, která:

¹ Nejznámější je dědění od téhož předka dvěma cestami – viz např. [Rac].

² Postupem doby se ukázalo, že princip rozhraní je velmi silnou stránkou Javy.

- nemusí být vůbec možná, a to v případě, že naše třídy vznikly děděním z knihovních (tedy námi neovlivnitelných) rodičovských tříd,
- může být obtížná, protože společný předek našich tříd by byl evidentně vykonstruovaný (viz dále třídy Usecka a Koule).

Poznámky:

- V současné době se místo abstraktních tříd používá téměř výhradně rozhraní. Často se setkáváme i s tím, že neabstraktní třída je dodávána i s rozhraním, které popisuje všechny její metody. Je to např. třída `java.io.DataInputStream` a rozhraní `java.io.DataInput`. Proč tomu tak je, se dozvíme v [13.3/215].
- V začátcích našeho programování v Javě použijeme rozhraní nejčastěji pro konstrukci „adaptérů“, se kterými se setkáváme při konstrukci grafického uživatelského prostředí.

Rozhraní nevynucuje příbuzenské vztahy, jako to činí dědičnost, rozhraní pouze **vnučuje určité dovednosti** těm, kteří jsou těchto dovedností principiálně schopni.

Například železniční společnosti je v podstatě jedno, zda v osobním voze přepravuje člověka nebo kočku, oba ale musí mít dovednost (schopnost) nastoupit (vejít se) do vozu. Tuto schopnost však není dobré získat děděním od společného předka typu `zivocich`, protože v případě např. žirafy či hrocha je její implementace nemožná.

Poznámka pro programátora v C či C++:

Rozhraní v Javě si můžeme představit jako (abstraktní) třídu, která obsahuje pouze čistě virtuální metody a konstanty. Pro rozhraní je povolena vícenásobná dědičnost. □

13.1 Konstrukce rozhraní

Zápis rozhraní se podobá zápisu třídy, např.:

```
public interface Info {  
    public void kdoJsem();  
}
```

Toto rozhraní se jménem `Info` popisuje pouze jednu metodu `kdoJsem()`.

13.2 Použití jednoho rozhraní

Každá třída, která implementuje rozhraní, musí uvést jeho jméno v hlavičce za klíčovým slovem `implements`.

Příklad 127:

Implementace rozhraní. V tomto případě je použití rozhraní `Info` oprávněné, protože `Usecka` a `Koule` mají velmi málo společného. Jistě by se dal nalézt a zkonstruovat jejich společný předek, ale otázka je proč, když pomocí rozhraní dosáhneme podstatně jednodušeji požadované schopnosti jednotným způsobem o sobě informovat.

```
public class Usecka implements Info {
    int delka;
    Usecka(int delka) { this.delka = delka; }
    public void kdoJsem() {
        System.out.println("Usecka");
    }
}

public class Koule implements Info {
    int polomer;
    Koule(int polomer) { this.polomer = polomer; }
    public void kdoJsem() {
        System.out.println("Koule");
    }
}

public class TestKoule {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
        Koule k = new Koule(3);
        u.kdoJsem();
        k.kdoJsem();
    }
}
```

Vypíše:

Usecka
Koule

což nás asi nepřekvapí, protože metody `kdoJsem()` jsou standardní me-

tody obou tříd. Bylo ovšem nutné je implementovat. Pokud bychom na to zapomněli, překladač by hlásil chybu: class Koule should be declared abstract; it does not define method kdoJsem() in interface Info.

To znamená, že pouze abstraktní třída si může dovolit implementovat rozhraní bez toho, že by se v ní naprogramovala i těla metod z tohoto rozhraní.

13.3 Použití rozhraní jako typu referenční proměnné

Pokud je rozhraní nějakou třídou implementováno, lze deklarovat referenční proměnnou typu rozhraní, pomocí které lze pak přistupovat k instancem všech tříd, které toto rozhraní implementují. Případně lze vytvořit novou instanci třídy a přiřadit ji proměnné typu rozhraní.

Tato možnost je velmi důležitá, protože vůbec nemusíme vědět, jaké třídy budou rozhraní implementovat a můžeme přesto vytvářet programy, využívající metod těchto tříd. To je možné proto, že metody jsou známy z popisu rozhraní. Je to také důvod, proč se používá rozhraní popisující metody neabstraktní třídy.

Příklad 128:

Použití referenční proměnné typu rozhraní.

```
public static void main(String[] args) {  
    Koule k = new Koule(3);  
    Info i = new Usecka(5);  
    i.kdoJsem();  
    i = k;  
    i.kdoJsem();  
}
```

Vypíše:

Usecka

Koule

Pozor:

Pomocí referenční proměnné typu rozhraní můžeme přistupovat pouze k metodám instance, které jsou v deklaraci rozhraní uvedeny – podrobně viz [13.5/217]. □

13.4 Implementace více rozhraní jednou třídou

Deklarujeme-li ještě jedno rozhraní:

```
public interface InfoDalsi {
    public void vlastnosti();
}
```

může třída implementovat obě dvě, protože na rozdíl od dědičnosti, kdy se dalо dědit jen od jednoho předka, není počet implementovaných rozhraní omezen.

Třída vypadá tak, že má za klíčovým slovem **implements** uveden seznam všech implementovaných rozhraní (na jejich pořadí nezáleží) oddělený čárkami.³

Příklad 129:

Třída implementující dvě rozhraní.

```
class Usecka implements Info, InfoDalsi {
    int delka;
    Usecka(int delka) { this.delka = delka; }
    public void kdoJsem() {
        System.out.print("Usecka");
    }
    public void vlastnosti() {
        System.out.println(" = " + delka);
    }
}

public class TestDvou {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
        u.kdoJsem();
        u.vlastnosti();
    }
}
```

Vypíše:

Usecka = 5

³V ukázce je uvedena jen třída Usecka, pro třídu Koule je to stejné.

13.5 Instance rozhraní může využívat jen metody rozhraní

V [13.3/215] bylo uvedeno, jak lze využít proměnnou rozhraní pro přístup k metodám třídy. Tuto možnost lze ale využít jen pro metody deklarované v tomtéž rozhraní, nikoliv pro ostatní metody třídy. Z pohledu rozhraní nejsou totiž známy.

Protože rozhraní nemá proměnné, nelze přes referenční proměnnou typu rozhraní přistupovat přímým přístupem ani k žádné instanční proměnné.

Příklad 130:

Referenční proměnná typu rozhraní umožňuje přístup pouze k metodám deklarovaným v rozhraní.

```
class Usecka implements Info, InfoDalsi {
    int delka;

    Usecka(int delka) { this.delka = delka; }

    public void kdoJsem() {
        System.out.print("Usecka");
    }

    public void vlastnosti() {
        System.out.println(" = " + delka);
    }

    public int getDelka() { return delka; }
}

public class Test {
    public static void main(String[] args) {
        Info info = new Usecka(2);
        info.kdoJsem();
        // info.vlastnosti();                                // chyba
        // System.out.println(info.getDelka());              // chyba
    }
}
```

Protože v rozhraní Info nejsou deklarovány metody `vlastnosti()` ani `getDelka()`, nelze k nim pomocí proměnné `info` přistupovat.

13.6 Implementované rozhraní se dědí beze změny

Zdědíme-li třídu, která implementovala rozhraní, bude metoda z rozhraní přístupná v obou třídách, ovšem bude se jednat o tutéž metodu. Pokud ji tedy v potomkovi nepřekryjeme (neimplementujeme znovu), budeme vždy volat metodu z rodičovské třídy.

Příklad 131:

Dědičnost neovlivňuje implementované metody.

```
class Usecka implements Info {
    int delka;
    Usecka(int delka) { this.delka = delka; }
    public void kdoJsem() {
        System.out.println("Usecka");
    }
}
class Obdelnik extends Usecka {
    int sirka;
    Obdelnik(int delka, int sirka) {
        super(delka);
        this.sirka = sirka;
    }
}
public class Test {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
        Obdelnik o = new Obdelnik(2, 4);
        Info iu = new Usecka(6);
        Info io = new Obdelnik(3, 6);
        u.kdoJsem();
        o.kdoJsem();
        iu.kdoJsem();
        io.kdoJsem();
    }
}
```

Vypíše:

```
Usecka
Usecka
Usecka
Usecka
```

Příklad 132:

Pro očekávanou funkci je nutné implementovat metodu `kdoJsem()` i ve třídě `Obdelnik`.

```
class Obdelnik extends Usecka {  
    // ...  
    public void kdoJsem() {  
        System.out.println("Ondřejka");  
    }  
}
```

Pak se vypíše očekávané:

```
Usecka  
Ondřejka  
Usecka  
Ondřejka
```

13.7 Dědění třídy a současná implementace rozhraní

V případě, že dědíme třídu, která má již implementováno rozhraní, je možné ve zděděné třídě libovolně implementovat další rozhraní⁴.

Příklad 133:

Implementace rozhraní v rodiči nijak neovlivňuje implementaci jiného rozhraní v potomkově.

```
class Usecka implements Info {  
    // konstruktor atd. viz dříve  
    public void kdoJsem() {  
        System.out.println("Usecka");  
    }  
}  
  
class Obdelnik extends Usecka implements InfoDalsi {  
    // konstruktor atd. viz dříve  
    public void kdoJsem() {  
        System.out.print("Ondřejka");  
    }  
}
```

⁴A nezapomenout přepsat implementaci zděděného – viz předchozí část.

```

public void vlastnosti() {
    System.out.println(" = " + delka + ", " + sirka);
}

public void vypisSirka() { // není z žádného rozhraní
    System.out.println(sirka);
}
}

public class Test {
    public static void main(String[] args) {
        Obdelnik o = new Obdelnik(2, 4);
        Info io = new Obdelnik(3, 6);
        InfoDalsi iod = new Obdelnik(5, 7);
        o.kdoJsem(); o.vlastnosti();
        io.kdoJsem(); ((Obdelnik)io).vlastnosti();
        ((Obdelnik)iod).kdoJsem(); iod.vlastnosti();
//        InfoDalsi iud = new Usecka(6); // chyba
        ((Obdelnik)io).vypisSirka();
    }
}

```

Vypíše:

```

Ondelnik = 2, 4
Ondelnik = 3, 6
Ondelnik = 5, 7
6

```

Všimněte si, jak lze pomocí přetypování získat z referenční proměnné typu rozhraní přístup k metodám třídy, které by jinak přístupné nebyly – `((Obdelnik)io).vlastnosti();` nebo `((Obdelnik)io).vypisSirka();`

Metoda `vlastnosti()` není součástí rozhraní `Info`, ale `InfoDalsi` a metoda `vypisSirka()` je instanční metoda třídy `Ondelnik` a s rozhraními nemá nic společného.

Protože třída `Usecka` neimplementuje rozhraní `InfoDalsi`, není možné přiřadit instanci této třídy proměnné tohoto rozhraní.

```
//        InfoDalsi iud = new Usecka(6); // chyba
```

13.8 Dědění rozhraní a konstanty rozhraní

Podobně, jako lze dědit třídy, lze dědit i rozhraní. To znamená, že rozhraní, které je potomkem, přebírá všechny deklarované metody od svého rodiče. Rozdíl od dědění tříd je v tom, že v případě rozhraní může být rodičů více a pak se přebírají všechny metody všech rodičů.

Mimo metody lze v rozhraní deklarovat i konstanty. Ačkoliv jejich deklarace vypadá jako deklarace proměnných, jedná se vždy o konstanty rozhraní. Je to totéž, jako by v deklaraci třídy byla použita klíčová slova **final** a současně **static**. Tato klíčová slova je možné použít i v rozhraní, ale je to zbytečné a nedělá se to.

Protože se jedná o konstanty, lze k nim přistupovat pomocí:

JménoRozhraní.jménoKonstanty

Poznámka:

Deklarovat konstantu je možné v kterémkoli rozhraní a dědičností rozhraní není konstanta ovlivněna. □

Příklad 134:

Vznik rozhraní potomka děděním ze dvou rodičovských rozhraní.

```
public interface InfoOba extends Info, InfoDalsi {  
    public int POCET = 3;  
}
```

```
class Usecka implements Info {  
    // vše zbývající viz v nezměněné podobě dříve  
    public void kdoJsem() {  
        System.out.println("Usecka");  
    }  
}
```

```
class Obdelnik extends Usecka implements InfoOba {  
    // vše zbývající viz v nezměněné podobě dříve  
    public void kdoJsem() {  
        System.out.print(POCET + " Obdelnik");  
    }  
}
```

```
public void vlastnosti() {  
    System.out.println(" = " + delka + ", " + sirka);  
}  
}
```

```

public class Test {
    public static void main(String[] args) {
        Info i = new Obdelnik(3, 6);
        InfoDalsi id = new Obdelnik(5, 7);
        InfoOba io = new Obdelnik(2, 4);
        i.kdoJsem(); ((Obdelnik)i).vlastnosti();
        ((Obdelnik)id).kdoJsem(); id.vlastnosti();
        io.kdoJsem(); io.vlastnosti();
        System.out.print("Pocet rozhrani = " + InfoOba.POSET);
    }
}

```

Vypíše:

```

3 Obdelnik = 3, 6
3 Obdelnik = 5, 7
3 Obdelnik = 2, 4
Pocet rozhrani = 3

```

Z programu i z výpisu je zřejmé, že třída `Obdelnik` implementuje jak rozhraní `Info`, tak i `InfoDalsi` a také `InfoOba`, což je vidět z toho, že je přímo přístupná konstanta `POSET`:

```
System.out.print(POSET + " Obdelnik");
```

Tam, kde by `POSET` nebyla přístupná, je nutné použít její celé jméno:

```
System.out.print("Pocet rozhrani = " + InfoOba.POSET);
```

13.9 Využití operátoru instanceof

Za běhu programu lze poznat, zda třída implementuje určité rozhraní, pomocí operátoru `instanceof`. Pokud je toto splněno, lze využívat metody tohoto rozhraní.

To je velmi důležitá možnost. Dovoluje nám totiž vytvářet programy, které nejenže mohou využívat metod, které ještě nejsou implementovány, ale navíc nenastane chyba při překladu, pokud se tyto metody pokusíme vyvolat nad objektem třídy, která dané rozhraní neimplementovala.⁵

Příklad 135:

V příkladu využití `instanceof` jsou třídy `Usecka` a `Obdelnik` naprostě stejné jako v [13.8/221].

⁵Příklad použití najeznete v [14.3/230].

```
public static void main(String[] args) {  
    Usecka u = new Usecka(5);  
    Obdelnik o = new Obdelnik(3, 6);  
    System.out.println("u implementuje Info");  
    if (o instanceof Info)  
        System.out.println("o implementuje Info");  
    if (u instanceof InfoOba)  
        System.out.println("u implementuje InfoOba");  
    if (o instanceof InfoOba)  
        System.out.println("o implementuje InfoOba");  
    if (u instanceof Usecka)  
        System.out.println("u je instancí Usecka");  
}
```

Vypíše:

u implementuje Info
o implementuje Info
o implementuje InfoOba
u je instancí Usecka

Poznámky:

- Operátor `instanceof` nepracuje jen s rozhraními, byť u nich se jeho použití nejčastěji předpokládá. Z posledního příkazu je vidět, že pomocí `instanceof` lze také zjistit, zda je referenční proměnná typu nějaké třídy.
- Pokud se dá již při překladu zjistit, že třídy zkoumané pomocí `instanceof` nemohou mít nic společného, skončí překlad chybou.

Cvičení:

1. Vytvořte třídy Ctverec a Usecka s vhodně zvolenými metodami a proměnnými. Vytvořte rozhraní Zobrazitelny s metodou `void zobraz()`. Zajistěte, aby třídy Ctverec a Usecka implementovaly toto rozhraní tak, že se vykreslí na obrazovku např.:

2. Od třídy Ctverec odvodte třídu DutyCtverec a zajistěte správnou implementaci metody `void zobraz()`, např.:

* *

14 Polymorfizmus

Slovo **polymorfizmus** lze nejbližše vyjádřit českým slovem *vícetvarost* nebo *mnohotvarost*. Jedná se o možnost využívat v programovém textu stejnou syntaktickou podobu pro metody s různou vnitřní reprezentací – jinak řečeno, voláme jakoby pořád stejnou metodu, ale ta provádí pokaždé něco jiného.

Polymorfizmus dovoluje jednotným způsobem manipulovat s prvky příbuzného, ale předem neznámého typu. Teorie ohledně polymorfizmu je značně propracovaná. My se zde omezíme na konstatování, že Java umožnuje polymorfizmus využívat nejpřirozenější možnou cestou.¹

Poznámka:

Některé ukázky polymorfizmu jsme viž viděli v [11.5/185]. Zde bude uveden systematičtější (i když nikoli vyčerpávající) výklad založený na třech nejběžnějších způsobech využití polymorfizmu. □

Základní trik v polymorfizmu je, že potomek může nahradit předka (viz [vir]). To prakticky znamená, že do referenční proměnné, původně deklarované s typem předka, můžeme přiřadit referenci na instanci potomka.² Pak lze přes referenční proměnnou předka využívat i metody, které jsou deklarovány v potomkově.

Otázkou je, proč používat tak komplikovaný přístup, když nám pro totéž stačí deklarovat referenční proměnnou typu potomka. To je pravda, a jen pro jednoho potomka nemá celý systém polymorfizmu valný smysl. Smysl začíná mít, pokud bude potomků více typů (druhů), a my pak k nim můžeme přistupovat jednotným (typově nezávislým) přístupem pomocí referenční proměnné na jejich společného předka.

¹Pokud jste programovali v C++ a strašily vás pojmy typu časná a pozdní vazba, tak na ně můžete v Javě prakticky zapomenout. Ne že by v Javě nebyly, ale běžný uživatel je nepotřebuje rozlišovat.

²Je to možné díky automatické rozšiřující konverzi – viz [3.7.1/47].

14.1 Využití abstraktní třídy

Abstraktní třída je přímo předurčena k tomu, aby polymorfizmu využila.

Jedná se o případ, kdy při konstrukci kořenové třídy již víme, jaké budeme potřebovat metody, ale jejich konkrétní implementace bude silně záviset na povaze zděděných tříd. Z tohoto důvodu není možné tyto metody naprogramovat. Výhodným řešením však je vytvořit abstraktní metodu s jasně definovanými parametry a návratovým typem. Tím donutíme programátory zděděných tříd, aby tuto metodu překryli (přeprogramovali).

Příklad 136:

Celý problém si ukážeme na příkladu ptáka, hada a slona. Pro tuto zvířenu lze snadno nalézt společného předka, kterým bude přirozeně živočich. Abstraktní třída `Zivocich` bude schopna uschovávat typ budoucího živočicha (tj. řetězce "pták", "had" nebo "slon"). Dále bude poskytovat metodu `vypisInfo()`, která vypíše tento typ a zavolá metodu `vypisDelku()`. Bohužel však na úrovni živočicha nelze říci, délka čeho se bude měřit,³ proto bude metoda `vypisDelku()` označena jako abstraktní.

```
abstract class Zivocich {  
    String typ;  
    Zivocich(String typ) { this.typ = new String(typ); }  
  
    public void vypisInfo() {  
        System.out.print(typ + ", ");  
        vypisDelku();  
    }  
  
    public abstract void vypisDelku();  
}
```

Třída `Ptak` již abstraktní nebude, protože v době implementace této třídy je již jasné, že se u ptáka bude měřit délka křídel. Proto je možné naprogramovat metodu `vypisDelku()`.

```
class Ptak extends Zivocich {  
    int delkaKridel;  
  
    Ptak(String typ, int delka) {  
        super(typ);  
        delkaKridel = delka;  
    }
```

³Délka hada by byla asi pochopitelná, délka slona už méně – a délka ptáka?

```
    delkaKridel = delka;
}

public void vypisDelku() {
    System.out.println("delka kridel: " + delkaKridel);
}
}
```

Totéž, co bylo řečeno pro ptáka, platí pro slona, s tou výjimkou, že nás bude zajímat délka chobotu. U hada to bude délka těla.

```
class Slon extends Zivocich {
    int delkaChobotu;
    Slon(String typ, int delka) {
        super(typ);
        delkaChobotu = delka;
    }

    public void vypisDelku() {
        System.out.println("delka chobotu: " + delkaChobotu);
    }
}

class Had extends Zivocich {
    int delkaTela;
    Had(String typ, int delka) {
        super(typ);
        delkaTela = delka;
    }

    public void vypisDelku() {
        System.out.println("delka tela: " + delkaTela);
    }
}
```

Jakmile máme tyto třídy, je možné vytvořit pole živočichů a postupně mu náhodně přiřadit objekty tříd Ptak, Slon a Had. Pak bez toho, že bychom měli nejmenší tušení, jaký živočich se kde skrývá, lze je zcela správně vypsat, protože díky polymorfizmu bude vždy použita správná metoda vypisDelku().

Poznámka:

Uvedený příklad vám může připadat jako vyumělkovaný, ale lze si asi poměrně snadno představit jeho aplikaci v ZOO, kde by se tímto způsobem počítaly náklady. Je asi zřejmé, že náklady na zlatou rybku se budou počítat podle zcela jiných podkladů než náklady na medvěda. Uvedeným postupem to ale lze provést jednotně. Místo, abychom programovali jeden obrovský *switch*, ve kterém se budou počítat v jedné větvi náklady na všechny zlaté rybky, v jiné větvi jiným způsobem náklady na všechny medvědy a v dalších stovkách větví náklady na stovky dalších druhů zvířat, využijeme polymorfizmu. Vždy, když přibude do ZOO nový živočišný druh, postačí – místo toho, abychom přisovali do *switch* další větev – vytvořit pro něj novou třídu, která bude potomkem třídy *Zivocich* a bude implementovat potřebnou metodu.

□

```
public class PolymAbstr {
    public static void main(String[] args) {
        Zivocich[] z = new Zivocich[6];
        for (int i = 0; i < z.length; i++) {
            switch ((int) (1.0 + Math.random() * 3.0)) {
                case 1: z[i] = new Ptak("ptak", i); break;
                case 2: z[i] = new Slon("slon", i); break;
                case 3: z[i] = new Had("had", i); break;
            }
        }

        Zivocich t;
        for (int i = 0; i < z.length; i++) {
            t = z[i];           // zbytecne, staci z[i].vypisInfo();
            t.vypisInfo();
        }
    }
}
```

Vypíše např.:

```
slon, delka chobotu: 0
had, delka tela: 1
had, delka tela: 2
slon, delka chobotu: 3
had, delka tela: 4
ptak, delka kridel: 5
```

Poznámka:

Proměnná typ v abstraktní třídě zivocich je v podstatě zbytečná. Je použita jen proto, aby bylo vidět, že abstraktní třída (na rozdíl od rozhraní) může mít libovolný počet proměnných. Pokud bychom přepsali metodu `vypisInfo()` následujícím způsobem:

```
public void vypisInfo() {
    System.out.print(getClass().getName() + ", ");
    vypisDelku();
}
```

vůbec bychom řetězec typ nepotřebovali (bylo by nutné předělat konstruktory atd.). □

14.2 Použití neabstraktních tříd

Poměrně často tradovaným omylem je tvrzení, že pro využití polymorfizmu je nutné mít abstraktní třídu, což někdy vede k vyumělkovaným konstrukcím. Skutečnost je taková, že je nutné mít kořenovou třídu, která ale nemusí být abstraktní.

Příklad 137:

Toto bude demonstrováno na lehce modifikovaném minulém příkladě. Nyní bude kořenová třída Zivocich neabstraktní a bude poskytovat jednu metodu `void vypisInfo()`, která vypíše druh zvířete. Tuto metodu budeme ve zděděných třídách zastiňovat, ovšem díky možnosti volat pomocí `super.` metody z předka ji také využijeme.

```
class Zivocich {
    public void vypisInfo() {
        System.out.print(getClass().getName() + ", ");
    }
}

class Ptak extends Zivocich {
    int delkaKridel;
    Ptak(int delka) { delkaKridel = delka; }

    public void vypisInfo() {
        super.vypisInfo();
        System.out.println("delka kridel: " + delkaKridel);
    }
}
```

```
class Slon extends Zivocich {  
    ... // analogicky  
}  
  
class Had extends Zivocich {  
    ... // analogicky  
}
```

V hlavním programu opět využijeme pole živočichů, kdy dopředu nevíme, na jaké živočichy bude ve skutečnosti ukazovat.

```
public class PolymDeden {  
    public static void main(String[] args) {  
        Zivocich[] z = new Zivocich[6];  
        for (int i = 0; i < z.length; i++) {  
            switch ((int) (1.0 + Math.random() * 3.0)) {  
                case 1: z[i] = new Ptak(i); break;  
                case 2: z[i] = new Slon(i); break;  
                case 3: z[i] = new Had(i); break;  
            }  
        }  
  
        for (int i = 0; i < z.length; i++)  
            z[i].vypisInfo();  
    }  
}
```

Vypíše např.:

```
Had, delka tela: 0  
Slon, delka chobotu: 1  
Ptak, delka kridel: 2  
Ptak, delka kridel: 3  
Had, delka tela: 4  
Slon, delka chobotu: 5
```

14.3 Použití rozhraní

V případě, kdy se společný předek těžko hledá⁴, je možné použít stejně úspěšně rozhraní. Rozhraní byla podrobně vysvětlena v [13/212], takže zde bude pouze příklad bez další teorie.

Příklad 138:

Z hlediska železniční společnosti může být zajímavá vlastnost hmotnost přepravených kusů. To lze zajistit implementací rozhraní `Vazitelny` ve všech třídách, jejichž instance si činí nároky na přepravu. Všimněte si, že v rozhraní není metoda `getHmotnost()`, což je logická chyba vzniklá již při návrhu rozhraní, ale v hlavním programu bude ukázáno, jak lze pomocí referenční proměnné na typ rozhraní tuto metodu vyvolat.

```
interface Vazitelny {
    public void vypisHmotnost();
}

class Clovek implements Vazitelny {
    int vaha;
    String profese;
    Clovek(String povolani, int tiha) {
        profese = new String(povolani);
        vaha = tiha;
    }
    public void vypisHmotnost() {
        System.out.println(profese + " : " + vaha);
    }
    public int getHmotnost() { return vaha; }
}

class Kufr implements Vazitelny {
    int vaha;
    Kufr(int tiha) { vaha = tiha; }

    public void vypisHmotnost() {
        System.out.println("kufr: " + vaha);
    }
}
```

⁴I když v dále uvedeném případě železniční společnosti by to byla jednoznačně třída `Kus` ; -)

K jednotlivým objektům přepravy pak lze přistupovat hromadně pomocí pole `kusJakoKus`. Proklamovaného individuálního přístupu se dosáhne pomocí indexu a využitím polymorfizmu `:-)`. Všimněte si, jak lze elegantně oddělit živou váhu pomocí operátoru `instanceof`. Protože však metoda `getHmotnost()` není deklarována v rozhraní `Vazitelny`, není možné pomocí referenční proměnné tuto metodu přímo využít. To ukažuje zakomentovaná řádka. Po příslušném přetypování na třídu `Clovek`, je ale přístup možný. Logickou chybou v návrhu rozhraní jsme však přišli o možnost využití polymorfizmu, takže pokud bychom přepravovali ještě objekty třídy `Zvire`, nevyhnuli bychom se použití `switch`, proti kterému jsme v [14.1/226] brojili.

```
public class PolymRozhra {
    public static void main(String[] args) {
        int vahaLidi = 0;
        Vazitelny[] kusJakoKus = new Vazitelny[3];
        kusJakoKus[0] = new Clovek("programator", 100);
        kusJakoKus[1] = new Kufr(20);
        kusJakoKus[2] = new Clovek("modelka", 51);

        System.out.println("CD - individualni pristup");
        for (int i = 0; i < kusJakoKus.length; i++) {
            kusJakoKus[i].vypisHmotnost();
            if (kusJakoKus[i] instanceof Clovek == true)
//                vahaLidi += kusJakoKus[i].getHmotnost();
                vahaLidi += ((Clovek) kusJakoKus[i]).getHmotnost();
        }
        System.out.println("Ziva vaha: " + vahaLidi);
    }
}
```

Vypíše:

```
CD - individualni pristup
programator: 100
kufr: 20
modelka: 51
Ziva vaha: 151
```

15 Vnořené třídy

Z [8/119] víme, že ve třídě mohou být proměnné a metody a obojí mohou být označeny jako **static** – pak patří třídě. Pokud nejsou označeny jako **static**, pak patří instanci.

Od JDK 1.1 může být prvkem třídy i jiná třída¹ – tím není myšlena referenční proměnná typu třída, ale třída deklarovaná uvnitř třídy. Taková třída se označuje jako *vnořená třída* (*nested class, top-less class*). Třída, která obsahuje nějakou vnořenou třídu, se pak označuje jako *třída nejvyšší úrovne* (*top-level class*) nebo – z pohledu vnořené třídy – *vnější třída*.

Pozor:

Nejedná se o problém dědění a rodičovské třídy, ani o problém kompozice, kdy je proměnná třídy referenční proměnou. □

Vnořená třída se typicky použije, pokud má své opodstatnění pouze v rámci své vnější třídy a nikde jinde. Její výhodou je, že má neomezená přístupová práva (podrobně viz [12.3/206]) ke všem proměnným a metodám své vnější třídy, ale přitom tvoří uzavřený celek. To znamená, že k jejím proměnným a metodám nemají přístup metody vnější třídy a samozřejmě ani kdokoliv zvnějšku přes referenční proměnnou typovanou na vnější třídu.

Protože nelze deklarovat referenční proměnnou na vnořenou třídu, není možný ani vnější přístup přes vlastní referenční proměnnou.

Stejně jako u proměnných a metod třídy může být vnořená třída deklarována s modifikátorem **static**. Pak se tato třída nazývá *statická vnořená třída* (*static nested class*). Nemá-li modifikátor **static**, pak se nazývá *vnitřní třída* (*inner class*).

Příklad 139:

Rozdíly mezi statickou vnořenou třídou a vnitřní třídou.

¹Dokonce i rozhraní, ale to už je velká specialita.

```
class Vnejsi {  
    static class StatickaVnorena {  
        // ...  
    }  
  
    class Vnitri {  
        // ...  
    }  
}
```

V Java Core API lze nalézt příklady obou druhů vnořených tříd, ovšem častější jsou vnitřní třídy.

Poznámka:

Kombinace statických vnořených tříd a statických a nestatických proměnných ve vnější třídě (případně téhož pro vnitřní třídy) jsou poměrně komplikované, byť jednoznačně popsané a logicky zdůvodnitelné. V dalším výkladu nebudeme probírat všechny kombinace, ale omezíme se pouze na vnitřní třídy, které implementují rozhraní. Tento typ tříd se totiž velmi často používá pro tzv. adaptéry v grafických prostředích typu AWT², takže pokud budete programovat aplety, tak se s nimi určitě setkáte. □

15.1 Vnitřní třídy

Vnitřní třídu použijeme, chceme-li implementovat rozhraní a současně chceme, aby metoda rozhraní byla přístupná pouze přes referenční proměnnou typu rozhraní. Jinak řečeno, přes referenční proměnnou na vnější třídu nebude možně implementovanou metodu rozhraní z této třídy zavolat.

Při prvním pohledu by se mohlo zdát, že to lze zajistit jednodušeji – pomocí přístupového práva **private** (podrobně viz [12.3.1/206]), ale není tomu tak. Metoda implementující rozhraní musí být přístupná z vnějšku, protože jinak většinou ztrácí svůj smysl.

Poznámka:

To, že mají být metody rozhraní přístupné pouze přes proměnnou typu rozhraní, má své opodstatnění. Vraťme se k příkladu z [13/213], kdy

²AWT znamená *Abstract Windowing Toolkit* a je to balík z Java Core API, který umožňuje vytvářet GUI.

se pomocí rozhraní řešila schopnost člověka (nebo kočky) nastoupit do osobního vagonu. Pravděpodobně bychom nechtěli, aby kdokoliv, kdo na nádraží pracuje s instancí třídy `Clovek`, na této instanci libovolně zkoušel schopnost nástupu do železničního vozu. Může se totiž stát, že tato instance vznikla v programu proto, aby se simulovala délka fronty před pokladnou, a tedy žádný vagón není k dispozici. □

15.1.1 Implementace rozhraní pomocí metody využívající vnitřní třídu

Příklad 140:

V příkladu použijeme již známé rozhraní `Info` – viz [13.1/213] a známou třídu `Usecka` – viz [13.2/214].

```
public interface Info {
    void kdoJsem();
}

class Usecka {
    int delka;
    Usecka(int delka) { this.delka = delka; }

    public Info informace() {
        return new UseckaInfo();
    }
}

class UseckaInfo implements Info {
    public void kdoJsem() {
        System.out.println("Usecka " + delka);
    }
}

public class Test {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
//        u.kdoJsem();      // nelze
//        Info i = u;       // nelze
        Info i = u.informace();
        i.kdoJsem();
    }
}
```

Vypíše:

Usecka 5

Třída Usecka neimplementuje žádné rozhraní, tedy ani metodu `kdoJsem()`, proto nelze použít příkaz:

```
u.kdoJsem();
```

ani příkaz:

```
Info i = u;
```

Místo toho deklaruje metodu

```
public Info informace() {
    return new UseckaInfo();
}
```

která vrací referenci na rozhraní `Info`, které je ve skutečnosti reprezentováno novým objektem třídy `UseckaInfo`. Z toho vyplývá, že třída `UseckaInfo` musí rozhraní `Info` implementovat. Skutečně tomu tak je:

```
class UseckaInfo implements Info {
```

Třída `UseckaInfo` implementuje pouze jednu metodu `kdoJsem()` – víc jich v deklaraci rozhraní není požadováno.

Při použití je nutné deklarovat proměnnou typu rozhraní `Info` a té přiřadit (pomocí volání metody `informace()`) instanci třídy `UseckaInfo`:

```
Info i = u.informace();
```

Teprve poté je možné zavolat metodu rozhraní:

```
i.kdoJsem();
```

Poznámka:

Při překladu vznikne soubor `Usecka.class` (jako obvykle) a navíc soubor `Usecka$UseckaInfo.class`, obsahující kód vnitřní třídy. □

15.1.2 Implementace rozhraní pomocí metody využívající anonymní vnitřní třídu

V adaptérech v AWT se postup uvedený výše ještě zjednoduší³. Vychází se z toho, že v `main()` proměnnou typu rozhraní `Info` nezajímá jméno vnitřní třídy, ale jen její instance. Z toho vychází koncept **anonymních vnitřních tříd**, kdy si ve třídě `Usecka` nemusíme lámat hlavu s hledáním vhodného názvu.

Poznámka:

Výsledný kód je mnohem méně čitelný, proto se anonymní vnitřní

³Nebo komplikuje, podle toho, zda hodnotíte čitelnost zdrojového kódu ; -)

třídu doporučuje používat jen pro velmi jednoduché rozhraní, kde je minimální délka kódu implementovaných metod. Vývojové prostředky typu JBuilder však anonymní vnitřní třídy používají s oblibou, proto je nutné tento způsob znát. □

Příklad 141:

Použití anonymní vnitřní třídy.

```
class Usecka {
    int delka;
    Usecka(int delka) { this.delka = delka; }

    public Info informace() {
        return new Info() {
            public void kdoJsem() {
                System.out.println("Usecka " + delka);
            }
        };      // konec prikazu return - středník nutný
    }          // konec metody informace()
}
```

Opět je zde metoda `informace()` třídy `Usecka`, která z vnějšího pohledu provádí totéž, co její jmenovkyně z předchozí části. Nyní se zde ale vyrábí instance rozhraní⁴, kdy se jakoby volá konstruktor `Info()`. Za ním ihned leží tělo implementovaného rozhraní, v našem případě pouze tělo jediné metody `kdoJsem()`.

Metoda `main()` se nijak neliší od předchozí. Opět je nutné vytvořit referenci na rozhraní a pomocí ní teprve volat metodu `kdoJsem()`, která opět není přístupná z instance vnější třídy `Usecka`.

```
public class Test {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
//        u.kdoJsem();                                // chyba
        Info i = u.informace();
        i.kdoJsem();
    }
}
```

Poznámka:

Při překladu vznikne soubor `Usecka.class` (jako obvykle) a navíc soubor `Usecka$1.class`, obsahující kód vnitřní anonymní třídy. □

⁴Ve skutečnosti je to objekt třídy `Object`.

15.1.3 Proměnná typu rozhraní využívající anonymní vnitřní třídu

Předchozí dva způsoby se používají velmi často. Zde bude jen pro ukázku uveden příklad, kdy pomocí anonymní vnitřní třídy inicializujeme proměnnou instance typu rozhraní.

Příklad 142:

```
class Usecka {  
    int delka;  
    Usecka(int delka) { this.delka = delka; }  
  
    public Info i = new Info() {  
        public void kdoJsem() {  
            System.out.println("Usecka " + delka);  
        }  
    }; // konec deklarace promenne i  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Usecka u = new Usecka(5);  
        u.i.kdoJsem();  
        if (u instanceof Info)  
            System.out.println("u implementuje Info");  
        if (u.i instanceof Info)  
            System.out.println("u.i implementuje Info");  
        if (u.i instanceof Info)  
            System.out.println(u.i.getClass().getName() +  
                " implementuje Info");  
    }  
}
```

Vypíše:

```
Usecka 5  
u.i implementuje Info  
Usecka$1 implementuje Info
```

Poznámka:

V příkladu je ukázka, že operátor `instanceof` funguje i na vnitřní anonymní třídy. V posledním příkazu tisku je použito volání metody `getClass()`, popsané v [11.8.4/198]. □

15.1.4 Vnitřní třída je vytvořena děděním

V následujícím příkladě bude ukázáno, jak lze použít vnitřní třídu pro zajištění v podstatě vícenásobné dědičnosti. Třída `Jmeno` implementuje rozhraní tak dobře, že implementovanou metodu nebude pro naše účely nutné měnit. Pak by bylo možné tuto třídu pouze zdědit a implementace rozhraní by byla zajištěna. Prosté zdědění však není ve třídě `Obdelnik` možné, protože ta dědí od třídy `Usecka`. Z tohoto důvodu je použita vnitřní třída, které v dědění od rodičovské třídy `Jmeno` nic nebrání.

Poznámka:

Tento způsob se používá opět v adaptérech AWT⁵ tam, kde např. z pěti již hotových metod chceme ve vnitřní třídě měnit pouze jednu a nechce se nám čtyři zbývající zastiňovat, což by bylo nutné při implementaci rozhraní. □

Příklad 143:

```
class Jmeno implements Info {
    public void kdoJsem() {
        System.out.println(this.getClass().getName());
    }
}

class Usecka {
    int delka;
    Usecka(int delka) { this.delka = delka; }
}

class Obdelnik extends Usecka {
    int sirka;
    Obdelnik(int delka, int sirka) {
        super(delka);
        this.sirka = sirka;
    }
    public Info informace() {
        return new Vnitri();
    }
    class Vnitri extends Jmeno {
    }
}
```

⁵Viz např. `java.awt.event.MouseAdapter`.

```
public class Test {
    public static void main(String[] args) {
        Obdelnik o = new Obdelnik(3, 6);
        Info i = o.informace();
        i.kdoJsem();
    }
}
```

Vypíše:

Obdelnik\$Vnitrni

U metody `informace()` je použito stejného postupu, jaký byl popsán v [15.1.1/234]. Třída `Vnitrni` je prázdná (tj. nemá žádné tělo), protože vše potřebné je již hotové (tj. metoda `kdoJsem()`) díky zdědění od třídy `Jmeno`. Navíc je ovšem možné cokoliv ze třídy `Jmeno` dodatečně upravit uvnitř třídy `Vnitrni`.

Poznámka:

V tomto případě by program fungoval stejně⁶ i pro:

```
public Info informace() {
    return new Jmeno();
}
```

To znamená, že vnitřní třídu bychom vůbec nepotřebovali, ovšem tím ztratíme možnost případné změny metod zděděných ze třídy `Jmeno`.

□

15.1.5 Vnitřní anonymní třída vznikne děděním

Příklad 144:

Tento příklad je pouze modifikace předchozího. Kromě metody `informace()` se nic nezměnilo. Opět je zde prázdná anonymní třída, protože vše potřebné je již ve třídě `Jmeno`.

```
class Obdelnik extends Usecka {
    int sirka;
    Obdelnik(int delka, int sirka) {
        super(delka);
        this.sirka = sirka;
    }
}
```

⁶Ovšem vypíše `Jmeno`.

16 Výjimky

Mechanismus **výjimek** (*exception*) je jednou z velmi silných bezpečnostních prvků Javy. Většina programovacích jazyků sice umožňuje testovat chybové kódy typu „neotevřený soubor“, „málo paměti“ atd., ale je nutné dodat, že opět „většina z většiny“ těchto jazyků ponechává provedení tohoto typu testů na libovůli programátora.¹

Java je v tomto ohledu světlou výjimkou. Přímo na úrovni kompliátoru nutí programátora, aby ve svém kódu reagoval na některé možné chybové stavy. Pokud nereaguje, program se nepřeloží. Například nám již známá metoda `ctiInt()` z [4.3/71] by bez ošetření výjimek neprošla překladem:

```
public static int ctiInt() {
    byte[] pole = new byte[20];
    String nacteno;
    int i;

    System.in.read(pole);
    nacteno = new String(pole).trim();
    i = Integer.valueOf(nacteno).intValue();
    return i;
}
```

Překladač ohláší: `unreported exception java.io.IOException; must be caught or declared to be thrown`

Dokud programátor tuto závadu neodstraní, nemůže pokračovat dále!

Poznámka:

Není tedy možné použít zavrženíhodný přístup: „Vím, že by se tady měl otestovat chybový stav. Ted' na to nemám čas, doplním to později.“ □

Způsobů, jak reagovat na výjimku, je několik a budou podrobně vyšvětleny dále. Konkrétní způsob reakce závisí na mnoha dalších okolnostech.

¹Ten je však často kvůli své lenosti neproveze :-)

```
public Info informace() {  
    return new Jmeno() { ... // prázdné tělo anonymní třídy  
};      // konec příkazu return - středník nutný  
}        // konec metody informace()  
}
```

Poznámka:

V tomto případě by program fungoval stejně i pro:

```
public Info informace() {  
    return new Jmeno();  
}
```

Ovšem opět bez možnosti případné změny metod zděděných ze třídy Jmeno. □

16 Výjimky

Mechanismus *výjimek* (*exception*) je jednou z velmi silných bezpečnostních prvků Javy. Většina programovacích jazyků sice umožňuje testovat chybové kódy typu „neotevřený soubor“, „málo paměti“ atd., ale je nutné dodat, že opět „většina z většiny“ těchto jazyků ponechává provedení tohoto typu testů na libovůli programátora.¹

Java je v tomto ohledu světlou výjimkou. Přímo na úrovni kompilátoru nutí programátora, aby ve svém kódu reagoval na některé možné chybové stavy. Pokud nereaguje, program se nepřeloží. Například nám již známá metoda `ctiInt()` z [4.3/71] by bez ošetření výjimek neprošla překladem:

```
public static int ctiInt() {
    byte[] pole = new byte[20];
    String nacteno;
    int i;

    System.in.read(pole);
    nacteno = new String(pole).trim();
    i = Integer.valueOf(nacteno).intValue();
    return i;
}
```

Překladač ohláší: `unreported exception java.io.IOException; must be caught or declared to be thrown`

Dokud programátor tuto závadu neodstraní, nemůže pokračovat dále!

Poznámka:

Není tedy možné použít zavrženíhodný přístup: „Vím, že by se tady měl otestovat chybový stav. Ted' na to nemám čas, doplním to později.“



Způsobů, jak reagovat na výjimku, je několik a budou podrobně vyšvětleny dále. Konkrétní způsob reakce závisí na mnoha dalších okolnostech.

¹Ten je však často kvůli své lenosti neprovede :-)

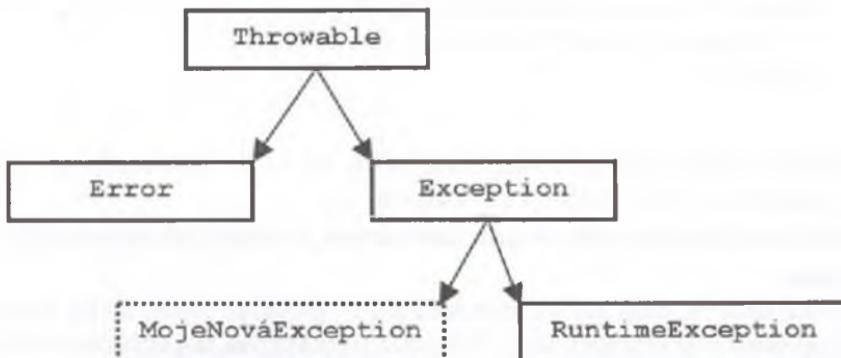
Poznámky:

- Pod pojmem *výjimka* (*exception*) je méněn také *výjimečný stav* (*exceptional event*) nebo (nepřesně) *chyba* (*error*) programu. Všechny uvedené pojmy označují událost, o které si nepřejeme, aby v našem programu vznikla, a bez jejíhož vzniku bude náš program fungovat lépe.
- V dalším textu bude využíván ještě jiný význam slova výjimka. Bude jím méněn objekt, který nese informaci o vzniklé události (tj. výjimce).
- V souvislosti s výskytem výjimky se používá označení *vyvolat výjimku* (též *vyhodit výjimku* – *throw exception*), např.: „Při otevřání souboru byla vyvolána výjimka.“ Jinými slovy řečeno to znamená: „Při otváráni souboru nastala chyba, na kterou interpreter Javy program upozornil zasláním zprávy o výjimce.“

16.1 Možné druhy výjimek

Java rozlišuje tři základní druhy výjimek `Error`, `RuntimeException` a `Exception`. Hned na počátku dodejme, že kompilátor Javy nutí, aby programátor v programu obsloužil pouze poslední z nich, tedy `Exception`. Reakce programátora na ostatní dva druhy je dobrovolná.

Všechny výjimky jsou potomky jedné třídy, třídy `Throwable` (vyhoditelná `:->`). Obrázek znázorňuje základ hierarchie objektů výjimek:



Se třídou `Throwable` většinou nepracujeme, protože se jedná o příliš velkou abstrakci chyby: „Nastala nějaká chyba.“ My spíše potřebujeme chybu více konkretizovat, abychom podle jejího druhu na ní dovedli odpovídajícím způsobem reagovat, např.: „Nastala chyba při otevřání souboru.“

16.1.1 Třída Error

Třída `Error` (a její potomci) představuje závažné chyby, které se mohou vyskytnout ve virtuálním stroji (JVM – interpreteru Javy). Na tyto chyby v našem programu obvykle nereagujeme.² Nereagujeme na ně z toho důvodu, že je v našem programu nedokázeme opravit. Po jejich výskytu program skončí s příslušnými chybovými výpisy.

Vyskytne-li se např. chyba typu `OutOfMemoryError`, nemůžeme provádět zhola nic, protože práci s pamětí neumíme nijak ze svého programu ovlivnit. Navíc všechny naše eventuální záchranné akce by potřebovaly další paměť (pro nové objekty), která ale už došla.

Poznámka:

Jednou z mála smysluplných reakcí na tento neblahý stav je zaslání programu, popisu prostředí našeho počítače a okolností vzniku chyby na: www.javasoft.com



16.1.2 Třída RuntimeException

Další třídou výjimek je třída `RuntimeException` (a její potomci). Tato třída zahrnuje výjimky, na které již dokážeme v našem programu úspěšně reagovat (tj. vzniklou chybu opravit). Jsou to však takové chyby, které se mohou vyskytnout v programu naprosto kdekoli a které testuje a vyvolává samotný JVM. Též se označují jako **asynchronní výjimky**. Jedná se např. o `ArithmaticException`, představující mimo jiné dělení nulou, nebo o `NumberFormatException`, což znamená, že se pokoušíme převést na číslo řetězec znaků, který nemá s číslem nic společného (např. "AHOJ").

Protože se tyto výjimky mohou vyskytnout kdekoli (ne jen při volání určitých metod), nenutí nás překladač, abychom na ně reagovali. **Ovšem my na ně reagovat můžeme, považujeme-li to za užitečné!**

Měli bychom na ně reagovat v těch místech, kde je zvýšená pravděpodobnost jejich výskytu. Například na zmíněnou výjimku `NumberFormatException` má cenu reagovat v místě, kdy uživatel našeho programu má zadat z klávesnice číslo. I když totiž v nápovedě vypíšeme něco jako: zadej počet cihel:, vždy se najde uživatel, který se pokusí napsat hodně .

²Musel by to být velmi speciální program, aby na výjimku typu `Error` reagoval.

Ještě častější případ, než jsou podobní „výzkumníci“, je, že se uživatel pouze překlepne a napiše 12* (kde * měla být původně číslice 8). V tomto případě je namísto reakce typu: „Chybně zadaný počet, zkuste to prosím znovu a pozorněji.“

Nemá samozřejmě větší smysl reagovat na tutéž chybu uprostřed konverzí v našem programu, kde jsme si stoprocentně jistí, že zmíněná situace nemůže nastat.

Poznámka:

Pokud přesto nastane ;-) není nic ztraceno. Java na tuto výjimku zareaguje ukončením programu a množstvím chybových výpisů, ze kterých lze vyčíst, kde chyba nastala. Dalším krokem je pak upravit program tím způsobem, aby bylo možno výskytu této chyby v programu zabránit. □

Příklad 145:

Ukázka výpisu informace o vzniklé výjimce:

Zadej pocet cihel: hodne

```
Exception in thread main java.lang.NumberFormatException:hodne
        at java.lang.Integer.parseInt(Compiled Code)
        at java.lang.Integer.valueOf(Integer.java:511)
        at Vyjim1.ctiInt(Vyjim1.java:11)
        at Vyjim1.main(Vyjim1.java:18)
```

Z výpisu je jasné patrné, že chyba nastala v metodě `ctiInt()` na řádce číslo 11.

Poznámka:

V extrémním případě je samozřejmě možné ošetřit v programu všechny výjimky `RuntimeException`. Pak by ale byl program značně neprehledný, protože těchto výjimek jsou řádově desítky. Jen přímých potomků třídy `java.lang.RuntimeException` je 24. □

16.1.3 Třída `Exception`

Výjimky této třídy a jejích podtříd jsou označovány jako ***kontrolované výjimky*** (*checked exceptions*) a musíme se o ně v programu postarat, tj. nějakým z dale uvedených způsobů je ošetřit.

Tyto výjimky se občas označují jako ***synchronní výjimky*** a na rozdíl od výjimek typu `RuntimeException` se nemohou vyskytnout kdekoli.

Vyskytují se pouze v souvislosti s voláním některých metod, a to těch, u kterých je zvýšená pravděpodobnost, že se při jejich použití může vyskytnout chyba.

Typicky jsou to všechny metody, které pracují se vstupy a výstupy. U těchto metod si jejich programátoři byli vědomi potenciálního nebezpečí a metody napsali tak, aby bylo při jejich použití nutné na nebezpečí reagovat. V začátcích našeho programování v Javě se budeme zabývat pouze tímto typem výjimek.

Pozor:

Práce se všemi zmíněnými typy výjimek je ale zcela stejná. To znamená, že např. `NumberFormatException` ze skupiny `RuntimeException` se ošetruje naprosto stejnými postupy jako `IOException` ze skupiny `Exception`. □

Poznámka:

Pokud budeme potřebovat vytvořit vlastní třídu výjimek, budeme dědit od třídy `Exception` – viz [16.5/255], protože naše výjimky budou jednoznačně synchronní, tj. vzniknou na námi definovaném místě. □

16.2 Způsoby ošetření výjimky

Programátor, který reaguje na (kontrolovanou) výjimku má v zásadě tři možnosti:

- výjimku neumí (nebo nechce) ošetřovat a proto informaci o jejím výskytu předá do nadřazené úrovně (tzn. volající metodě),
- výjimku zachytí a kompletně ošetří v metodě, kde se vyskytla,
- výjimku částečně či kompletně ošetří v metodě a navíc pošle informaci o jejím výskytu do nadřazené úrovně (je to kombinace obou předchozích způsobů).

Každý z těchto způsobů má své výhody a nevýhody a také různé varianty použití, které budou uvedeny dále.

Obecně řečeno – vytváříme-li **modul** (*reusable package*), nevíme dopředu, jak si budoucí uživatel našeho modulu bude přát výjimku ošetřit. V tomto případě je vhodné použít třetí zmíněný postup, kdy výjimku ošetříme jen tak, aby náš objekt zůstal „provozuschopný“ (tj. „nespadl“; $-$) a výjimku propagujeme (rozšíříme) ven z metody. Podrobnosti viz v [16.2.3/249].

16.2.1 Předání výjimky výše – deklarace výjimky

Jedná se o použití tzv. *propagace výjimek* (*propagating exceptions*, též *šíření výjimek*). Je to případ, kdy se metoda, ve které se výjimka vyskytla, „zříká odpovědnosti“ za její zpracování. Toto předání odpovědnosti se jasně **deklaruje** již v hlavičce metody pomocí klíčového slova `throws` (vyhazuje). Řešení problému, co se vzniklou výjimkou, se pouze odsouvá, někdy ale stejně bude muset být provedeno.

Příklad 146:

Deklarace výjimky.

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

```
public class Vyjimka {
    public static int ctiInt() throws IOException {
        byte[] pole = new byte[20];
        String nacteno;
        int i;

        System.in.read(pole);
        nacteno = new String(pole).trim();
        i = Integer.valueOf(nacteno).intValue();
        return i;
    }

    public static void main(String[] args) throws IOException {
        System.out.print("Zadej pocet cihel: ");
        int i = ctiInt();
        System.out.println("Cihel je: " + i);
    }
}
```

Jak je vidět, jedná se o nejjednodušší řešení (alespoň z prvního pohledu). Není třeba provádět žádné zásahy do stávajícího kódu metody `ctiInt()`, stačí jen doplnit její hlavičku:

```
public static int ctiInt() throws IOException {
```

Tato řádka říká, že metoda `ctiInt()` si je vědoma toho, že se v ní může vyskytnout výjimka, ale tento stav nijak dále neřeší. Jak již bylo řečeno, problém se pouze odsouvá, takže pokud i nadřazená úroveň (tj. volající – zde `main()`) použije stejnou strategii, musí i ona uvést ve své hlavičce:

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

Tím se problém ošetření výjimky přesune do další nadřazené úrovně, kterou je JVM, a ta tuto výjimku skutečně ošetří tím, že skončí program a vypíše chybové hlášení – podobně, jako v příkladu 145 na str. 244.

Je zřejmé, že tento způsob není úplně ideální, zvláště pokud pracujeme v týmu. Pak totiž nutíme všechny, kteří použijí metody našich tříd, aby doplnili do svého kódu i ošetření výjimek, o které jsme se my nepostarali. Je-li tedy jediným důvodem této reakce na výjimku pouze naše lenost, (případně alibizmus typu „Mě to nezajímá, ať si to vyřeší jinde.“), je vhodné lenost potlačit a použít jiný způsob ošetření výjimky.

Popsaný způsob ale může mít své odůvodnění. Je-li program (jako ve zde uvedeném příkladě) zcela závislý na úspěšném provedení operace `ctiInt()`, pak nemá smysl výjimku ošetřovat v metodě `ctiInt()`. Tím, že bychom se z porouchaného zařízení pokoušeli číst např. v cyklu, bychom jej pravděpodobně nezprovoznili. Pak je tedy namísto sdělit metodě `main()` zprávu „nepovedlo se“, což právě zajistí výjimka, která se do `main()` rozšíří, a nechat na `main()`, zda chce pokračovat dále.

16.2.2 Kompletní ošetření výjimky

Pokud nechceme propagovat výjimku do nadřazené úrovně, je možné ji v metodě ošetřit. K této činnosti používáme jazykovou konstrukci `try-catch`. Do bloku `try` uzavřeme celý kód, ve kterém se může výjimka vyskytnout, tzv. **chráněný úsek**. Blok `catch`, který za blokem `try` bezprostředně následuje, říká, na jakou výjimku se bude reagovat a jak.

Příklad 147:

Ošetření výjimky.

```
import java.io.*;
public class Vyjimka {
    public static int ctiInt() {
        byte[] pole = new byte[20];
        String nacteno;
        int i;

        try {
            System.in.read(pole);
            nacteno = new String(pole).trim();
            i = Integer.valueOf(nacteno).intValue();
            return i;
        }
```

```

        catch (IOException e) {
            System.out.println("Chyba pri cteni");
            return -1;
        }
    }

public static void main(String[] args) {
    System.out.print("Zadej pocet cihel: ");
    int i = ctiInt();
    System.out.println("Cihel je: " + i);
}
}

```

Poznámka:

Zde je nutno zdůraznit eleganci, s jakou je možné oddělit „výkonný kód“ (blok **try**) od „chybového kódu“ (blok **catch**). Když si vzpomenete, kolik to bylo v procedurálních programovacích jazyčích typu C nebo Pascalu zanořených **if** podmínek při testování otevření a uzavření současně dvou souborů, jistě budete souhlasit, že se zde čitelnost kódu významně zlepšila. □

Když se zamyslíme nad celým kódem, zjistíme, že není úplně ideální. Předpokládáme totiž mlčky, že chyba vstupu nenastane. Pokud nastane, vypíše se pouze chybové hlášení, ale do funkce **main()** se vrátí hodnota **-1**, kterou **main()** bez dalších problémů jako platné celé číslo vytiskne. O trochu lepší řešení by bylo:

```

public static void main(String[] args) {
    System.out.print("Zadej pocet cihel: ");
    int i = ctiInt();
    if (i != -1)
        System.out.println("Cihel je: " + i);
}

```

Zde vidíme, že je nutné použít návratový chybový kód (**-1**) a ten v **main()** dále testovat. V další části bude vidět, jak toto celé lze provést konzistentně.

16.2.3 Ošetření výjimky a následné předání výše

V předchozí části bylo nutné o výjimce informovat volající metodu `main()` předáním chybového kódu -1. Toto je možné provést lépe využitím mechanizmu výjimek, kdy sami programově výjimku vyhodíme příkazem `throw`³.

Příklad 148:

Vyhození výjimky po jejím ošetření.

```
import java.io.*;
public class Vyjimka {
    public static int ctiInt() throws IOException {
        byte[] pole = new byte[20];
        String nacteno;
        int i;

        try {
            System.in.read(pole);
            nacteno = new String(pole).trim();
            i = Integer.valueOf(nacteno).intValue();
            return i;
        }
        catch (IOException e) {
            System.out.println("Chyba pri cteni");
            throw e; // vyhození výjimky
        }
    }

    public static void main(String[] args) {
        System.out.print("Zadej pocet cihel: ");
        try {
            int i = ctiInt();
            System.out.println("Cihel je: " + i);
        }
        catch (IOException e) {
            System.out.println("Program neprobehl spravne");
        }
    }
}
```

³Nezaměňovat s `throws`, který smí být pouze v hlavičce metody.

Všimněte si, že se jedná o spojení obou předchozích metod dohromady. Novinkou je pouze příkaz metody `ctiInt()`

```
throw e;
```

kterým `ctiInt()` **znovu** vyvolává (předává) vyskytnuvší se, již ošetřenou výjimku výše.

Metoda `main()` by na ni mohla reagovat tím, že ji předá také výše (viz [16.2.1/246]). Zde na ni reaguje správně tím, že ji odchytí a kompletně zpracuje. Tímto jednoduchým způsobem se zbavíme nutnosti vymýšlet hodnoty, které se „nikdy v programu nemohou vyskytnout“, pro návratové kódy našich metod.

Poznámka:

V této souvislosti je třeba říci, že na proměnnou `e`, která je parametrem příkazu `throw`, se lze dívat jako na formální parametr metody – podrobnosti viz [16.2.5/251]. □

16.2.4 Naprosto nejhorší reakce na výjimku

Dále popsaný způsob je jen podmnožinou způsobu z [16.2.2/247]. Pro jeho nebezpečnost a rozšíření je mu ale věnována samostatná část. Programátor metody `ctiInt()` nechce obtěžovat šířením výjimky své kolegy. Z důvodu lenosti však také nechce vzniklou výjimku nijak zvlášť ošetřovat, protože „se to nikdy nestane“. Použije proto následující postup.

Příklad 149:

```
public static int ctiInt() {
    byte[] pole = new byte[20];
    String nacteno;
    int i = 0;

    try {
        System.in.read(pole);
        nacteno = new String(pole).trim();
        i = Integer.valueOf(nacteno).intValue();
    }
    catch (IOException e) {
        // !!! skutečně zde není žádný kod !!!
    }
    return i;
}
```

Čili blok `catch` výjimku zachytíl, ale **NIC** nepovedl. **Toto je cesta do pekel!** Pokud budete tento postup používat, počítejte s tím, že při práci na větších projektech vás vaši kolegové po několikahodinovém hledání, u koho se vyskytla chyba, opravdu nebudou mít rádi. Tímto způsobem jste se totiž připravili o všechny výhody mechanizmu výjimek.

16.2.5 Rozumná reakce na výjimku

Jak tedy postupovat, když jsme přesvědčeni, „že se to nikdy nestane“ a tedy nechceme věnovat svůj drahocenný čas na ošetření výjimky, ale způsob uvedený v předchozí části považujeme za skutečně nevhodný?

Pro tyto situace nám dává Java Core API velmi vhodnou metodu `printStackTrace()`, kterou můžeme okamžitě použít. Tato metoda v případě výskytu výjimky vypisuje chybové hlášení, jehož podoba je uvedena v [16.1.2/244]. O hlášení chyby a upřesnění místa chyby tedy nikdy ne přijde me.

Pokud pak v programu zjistíme, že předpoklad „nikdy se to nestane“ není správný, není větším problémem dodatečně nahradit metodu `printStackTrace()` vhodnějším ošetřením výjimky.

V příkladu bude uveden jen výsek kódu:

```
catch (IOException e) {  
    e.printStackTrace();  
}
```

Poznámka:

Všimněte si použití `e.printStackTrace()`. Jméno výjimky v závorkách příkazu `catch` totiž funguje jako formální parametr metody. Jinak řečeno – do bloku `catch` se předává objekt typu výjimky a nad tímto objektem lze volat jeho metody. Často je to právě metoda `printStackTrace()`. Druhý častý případ je volání metody `getMessage()`, která „popisuje chybový stav“ a o níž se zmíníme v další části. □

Pozor:

Tento postup má pouze jednu vadu na kráse, a to, že `printStackTrace()` vypisuje své hlášení na konzoli. V případě, že používáme grafické uživatelské prostředí, se nemusíme o tomto výpisu vůbec dozvědět. Řešení ovšem existuje i v tomto případě – pomocí `System.setErr()` nebo `System.setOut()` přesměrujeme chybový výpis do souboru – podrobně viz [19.2.1/302]. □

16.3 Seskupování výjimek

Když se zamyslíme nad předchozími třemi příklady, zjistíme, že musíme reagovat na výjimku `IOException`, která ale pravděpodobně nikdy nena- stane. Zato výskyt `RuntimeException` výjimky `NumberFormatException` je v případě nedisciplinovaného uživatele velmi pravděpodobný, a proto by bylo mimořádně vhodné tuto výjimku odchytávat. Zpracování výjimky `NumberFormatException` je totiž zcela v kompetenci metody `ctiInt()`.

Máme tedy dvě výjimky a rádi bychom reagovali na obě. To je naštěstí bez problémů možné, protože Java nijak neomezuje počet bloků `catch` následujících za jedním blokem `try`.

Příklad 150:

Použitím dvou bloků `catch` dostaneme téměř ideální metodu `ctiInt()`. Pomocí nekonečného cyklu `while (true)` nutíme uživatele zadávat číslo tak dlouho, dokud jej nezadá správně. Naopak výjimka `IOException` je po chybovém výpisu propagována.

```
public static int ctiInt() throws IOException {
    while (true) {
        try {
            byte[] pole = new byte[20];
            System.in.read(pole);
            String nacteno = new String(pole).trim();
            int i = Integer.valueOf(nacteno).intValue();
            return i;
        }
        catch (NumberFormatException e) {
            System.out.print("Cislo " + e.getMessage());
            System.out.println(" nebylo zadano dobre");
            System.out.print("Zkuste to znova: ");
        }
        catch (IOException e) {
            System.out.println("Chyba cteni");
            throw e;
        }
    }
}

// main() zůstává nezměněna
```

Vypíše např.:

```
Zadej pocet cihel: hodne
Cislo hodne nebylo zadano dobre
Zkuste to znovu: 12*
Cislo 12* nebylo zadano dobre
Zkuste to znovu: 128
Cihel je: 128
```

16.3.1 Postupná selekce výjimek

Pořadí, ve kterém jsou bloky `catch` uváděny, je významové. Při výskytu výjimky se postupně procházejí jednotlivé bloky `catch` v tom pořadí, v jakém jsou v programu uvedeny. Jakmile vyhozená výjimka vyhovuje třídě výjimek uvedené v `catch` nebo i její libovolné rodičovské třídě, provede se tento blok a ostatní bloky `catch` jsou přeskočeny.

Příklad 151:

V kombinaci se stromem dědičnosti výjimek se uvedený postup často využívá pro rozlišení úrovně výjimek.

```
try {
    ...
}
catch (NumberFormatException e) {
    // zpracuje jednu výjimku ze třídy RuntimeException
}
catch (RuntimeException e) {
    // zpracuje všechny zbývající výjimky ze třídy
    //     RuntimeException
}
```

Pozor:

Je asi jasné, že pokud pořadí bloků `catch` prohodíme, pak se na výjimku `NumberFormatException` nikdy nedostane, protože bude zachycena v rámci své rodičovské třídy `RuntimeException`. □

16.4 Vyvolání výjimky

Chceme-li si vyzkoušet, zda náš programový kód ošetruje danou výjimku správně, může to být poněkud obtížné. Výjimka `IOException`, která byla

dosud převážně zmiňována, se vyvolá vnějším zásahem značně obtížně. Naštěstí Java dává možnost vyvolat výjimku programově kdekoliv příkazem `throw`. Ten jsme dosud použili v bloku `catch` pro propagaci výjimky, která už ovšem existovala. Nic nám však nebrání vytvořit nový objekt typu `IOException` a ten příkazem `throw` vyhodit, tj. zajistit, aby se nově vzniklá instance výjimky začala v programu šířit standardním postupem.

Příklad 152:

Vyhození nově vygenerované výjimky.

```
public static int ctiInt() throws IOException {
    //      deklarace zůstávají nezměněny
    try {
        System.in.read(pole);
        nacteno = new String(pole).trim();
        i = Integer.valueOf(nacteno).intValue();
        if (i == 0)
            throw new IOException();
        return i;
    }
    catch (IOException e) {
        System.out.println("Chyba ctení");
        throw e;
    }
}
//      metoda main() zůstává nezměněna
```

Vypíše:

Zadej pocet cihel: 0

Chyba ctení

Program neproběhl spravne

Protože po zadání hodnoty 0 je příkazem: `throw new IOException();` vytvořena nová instance třídy `IOException`, se kterou se dále pracuje jako s regulérní výjimkou. Je tedy zachycena následujícím blokem `catch` a již známým způsobem zpracována.

Poznámka:

Vyvolání nově vzniklé výjimky je možné použít pro otestování naší „záchranné části“ programu. Mnohem častější případ, kdy se tento způsob používá, je konstrukce třídy, která skutečně reaguje na nějaký chybový stav, což je popsáno v následující části. □

16.5 Vytvoření a použití vlastní výjimky

Budeme-li v Javě nějakou dobu programovat, přestane nám stačit reagovat na cizí výjimky a budeme postaveni před nutnost vytvářet výjimky vlastní. Stane se to tehdy, když budeme programovat nějakou třídu, jejíž instance se může dostat do stavu, který považujeme za chybový. Tento stav nedokážeme ve třídě ošetřit, proto použijeme standardního mechanismu výjimek a zprávu o vzniku chybového stavu předáme pomocí výjimky.

Vytvořit takovouto třídu se znalostmi, které již máme, není nic obtížného. Před jejím vytvářením je ovšem vhodné vytvořit i vlastní výjimku,⁴ děděním od třídy `Exception`. To proto, že se bude jednat o typickou *synchronní výjimku*, která vzniká v našem programu na přesně definovaném místě. Druhý dobrý důvod je, že budeme pravděpodobně chtít, aby uživatel naší třídy musel na vzniklou výjimku reagovat, což u výjimek třídy `RuntimeException` není jak známo povinné.

Příklad 153:

V příkladu se budeme snažit o vytvoření třídy `Bankomat`, jejímž účelem je vydávat peníze. Ovšem skutečný bankomat si peníze sám netiskne⁵, takže může vydávat jen do té doby, dokud mu nedojde zásoba peněz, které do něj zřízenec banky jednorázově vložil. Pokud mu peníze dojdou, tj. bylo již realizováno hodně výběrů, přepne se do režimu „Dočasně mimo provoz.“, ohláší to centrále a žádné peníze vám nedá.

Poznámka:

Nenechte se zmást tím, že je bankomat tak primitivní – jde tu v první řadě o princip použití výjimky, nikoliv o peníze ; -) □

```
class BankomatException extends Exception {
    public BankomatException() {
        super("Bankomat docasne mimo provoz");
    }
}

class Bankomat {
    private int penize = 0;
    Bankomat(int kolik) { penize = kolik; }
```

⁴ Čili třídu, jejíž instance bude sloužit k přenosu informací o výjimce.

⁵ Tedy pevně v to doufám ; -)

```
int vydejPenize(int kolik) throws BankomatException {
    if (kolik > penize) {
        System.out.println("Nedostatek hotovosti");
        throw new BankomatException();
    }
    else {
        penize -= kolik;
        return kolik;
    }
}

public class TestBankomatu {
    public static void main(String[] args) {
        Bankomat b = new Bankomat(1000);

        try {
            System.out.println("Vydano: " + b.vydejPenize(200));
            System.out.println("Vydano: " + b.vydejPenize(1000));
        }
        catch (BankomatException be) {
            System.out.println("Vyhledejte jiny bankomat");
            System.out.println(be.getMessage());
            be.printStackTrace();
        }
    }
}
```

Vypíše:

```
Vydano: 200
Nedostatek hotovosti
Vyhledejte jiny bankomat
Bankomat docasne mimo provoz
BankomatException: Bankomat docasne mimo provoz
    at Bankomat.vydejPenize(TestBankomatu.java:14)
    at TestBankomatu.main(TestBankomatu.java:28)
```

Nejdříve jsme vytvořili novou výjimku `BankomatException` jako potomka třídy `Exception`. Protože nám metody třídy `Exception` vyhovují, bylo nutné naprogramovat pouze konstruktor, který nedělá nic jiného, než že zavolá konstruktor třídy `Exception` a předá mu popis naší výjimky, tedy

"Bankomat docasne mimo provoz"⁶

Pak již lze napsat požadovanou třídu Bankomat a v ní metodu:

```
int vydejPenize(int kolik) throws BankomatException {
    která způsobem známým již z [16.4/253], vyvolá nově vzniklou výjimku:
    throw new BankomatException();
```

Od této chvíle je nutné při použití metody `vydejPenize()` vždy ošetřit výjimku `BankomatException` libovolným z již známých postupů. V `main()` jsou volány metody `getMessage()` i `printStackTrace()`, aby bylo zřejmé, že můžeme opravdu použít všeho, nač jsme zvyklí. Z výpisu je pak vidět, jaký řetězec vlastně vrací metoda `getMessage()`.

Poznámka:

Pokud bychom nechtěli ve výpisu identifikovat naši výjimku textem `Bankomat docasne mimo provoz` a stačilo by nám pouze `BankomatException` jako jméno třídy, pak lze nejjednodušší získat novou třídu výjimek takto:

```
class BankomatException extends Exception { }
tedy dědění proběhlo, ale nic se neměnilo ani nepřidávalo.
```

Preferujte tento způsob před tím, kdy metoda `vydejPenize()` vyvolává výjimku `Exception`⁷. Tento druhý způsob není z hlediska přehlednosti chybouvých hlášení dobrý. □

16.6 Konstrukce **try-catch-finally**

Dvojici bloků **try-catch** je možné rozšířit o nepovinný koncový blok **finally**. Konstrukce vypadá takto:

```
try {
    // hlídaný blok
}
catch ( TypVýjimky ) {
    // ošetření výjimky
}
finally {
    // tento kód se provede vždy
}
```

Koncovým blokem pokračuje program po ukončení bloku **try** (tj. když výjimka nenastala) i po ukončení bloku **catch** (tzn. výjimka byla zachycena).

⁶Asi už jste se s tím někdy setkali, že? ; -)

⁷Přece: „Výjimka jako výjimka.“ ; -)

cena). Koncový blok se vykoná i v případě, kdy je v blocích **try** a/nebo **catch**:

- uveden příkaz **return**,
- vyvolána jiná, i nezachycená výjimka, která je propagována výše.

Typické použití koncového bloku **finally** je při práci se soubory – podrobně viz [18/269]. Zde mohou vzniknout minimálně dvě výjimky – při otevření souboru a při čtení ze souboru. Je mimořádně vhodné zajistit, aby byl v každém případě uzavřen otevřený soubor, což lze zajistit právě pomocí bloku **finally**.

Příklad 154:

V příkladu vypisuje metoda **vypisSoubor()** zadaný soubor a vrací počet přečtených znaků. V bloku **try** je možno vyvolat (nesmyslnou) výjimku **ArithmeticException**, aby bylo možné ověřit, že blok **finally** bude vždy proveden.

```
public static long vypisSoubor(String jmeno) throws
                                                IOException {
    File frJmeno = new File(jmeno);
    FileInputStream fr = null;
    try {
        fr = new FileInputStream(frJmeno);
        for (int i = 0; i < frJmeno.length(); i++)
            System.out.print((char)fr.read());
        // throw new ArithmeticException();
        return frJmeno.length();
    }
    catch (FileNotFoundException e) {
        System.out.println("Soubor neotevren");
        throw e;
    }
    catch (IOException e) {
        System.out.println("Chyba cteni");
        throw e;
    }
    finally {
        System.out.println("Konec");
        if (fr != null)
            fr.close();
    }
}
```

Poznámka:

Když se nepodaří soubor otevřít, bude proměnná `fr` stále obsahovat hodnotu `null` a v bloku `finally` nedojde k uzavření neotevřeného souboru. □

16.6.1 Konstrukce **try-finally**

Tato konstrukce představuje zajímavé a velmi užitečné řešení tehdy, když potřebujete, aby byl nějaký úsek kódu vždy proveden. S výjimkami nemusí mít tento kód nic společného!⁸

Příklad 155:

Představte si například, že čtete ze souboru⁹ a chcete, aby bylo možné čtení okamžitě ukončit při výskytu nějakého znaku. To lze snadno zajistit, ale je třeba se postarat i o řádné uzavření souboru. Tento úkol lze elegantně splnit právě použitím bloků `try` a `finally`, protože blok `finally` musí vždy proběhnout.

```
import java.io.*;
public class TryFin {
    public static void main(String[] args) throws IOException {
        File frJm = new File("a.txt");
        FileInputStream fr = new FileInputStream(frJm);
        int c;
        try {
            while ((c = fr.read()) != -1) {
                if (c != 'K')
                    System.out.print((char) c);
                else
                    return;
            }
        }
        finally {
            System.out.println("Soubor uzavren");
            fr.close();
        }
    }
}
```

⁸V příkladu jsme se jich zbavili jejich deklarací.

⁹Čtení ze souborů bude podrobně vysvětleno v [18/269].

Pro soubor `a.txt` vypíše:

<code>ahoj</code>	<code>ahoj</code>
<code>nazdar</code>	<code>nazdar</code>
<code>Konec</code>	<code>Soubor uzavren</code>
<code>cao</code>	

Dobré rady:

- Nepoužívejte bloky `try-catch-finally` pro každou řádku kódu jednotlivě, ale pro celý blok kódu.
- Nenechte blok `catch` prázdný – použijte minimálně metodu `printStackTrace()`.
- Nezneužívejte systém zachycení výjimek, např. tím, že místo `break` pro opuštění cyklu vyhodíte výjimku. Program bude pomalejší a výrazně nepřehlednější.
- Důsledně používejte blok `finally`, pokud část programu, která vyhodila výjimku, alokuje nějaké zdroje. Typicky jsou to otevřené soubory, které musí být v bloku `finally` uzavřeny.

Běžné chyby:

- Blok `catch` má jen jeden argument, nelze napsat:
`catch (IOException e1, EOFException e2)`
- Bloky `catch` se procházejí v pořadí v jakém jsou uvedeny. Pokud je v prvním bloku zachycena výjimka typu `Exception`, jsou ostatní bloky zbytečné.
`catch (Exception e) { ... } // zachytí všechny výjimky`
`catch (IOException e) { ... } // sem se nikdy nedostane`

Cvičení:

1. Vytvořte třídu `SpojovySeznam`, která umí uchovávat prvky typu `Object`. Připravte metody `void vloz(Object obj)`, `Object vyjmniPrvniho()` a `void vypisSeznam()`.
2. Vytvořte třídu výjimek `SeznamException`, které budou vyvolávány metodou `vyjmniPrvniho()` ze třídy `SpojovySeznam` v případě, že bude seznam prázdný.
3. Napište program, který bude využívat instance třídy `SpojovySeznam` a vyzkoušejte propagaci výjimek i jejich ošetření v blocích `try-catch`.

17 Adresáře a soubory

Pro práci se soubory slouží třída `File`. Hned zpočátku je nutné zdůraznit, že tato třída nezajišťuje žádné způsoby čtení nebo zápisu, slouží jen jako jakýsi „manažer“ souborů.

Pozor:

Třída `File` umožňuje též práci s adresáři, a to prakticky stejně jako práci se soubory. V případě pochyb je nutné pomocí metody `isDirectory()` rozlišit adresáře od souborů. □

17.1 Zajištění nezávislosti na operačním systému

Operační systémy mají různé systémy správy souborů a tím také mimo jiné, různé odlišnosti v oddělování jednotlivých adresářů v úplné cestě souborů.

U dvou nejužívanějších operačních systémů se setkáme s těmito konvencemi:

- UNIX (Linux, Solaris, ...) /adresar1/adresar2/soubor.txt
– oddělovačem adresářů je znak / a není více disků
- MS Windows c:\adresar1\adresar2\soubor.txt
– oddělovačem adresářů je znak \ a je více disků

Třída `File` poskytuje čtyři statické proměnné, které tuto závislost přesouvají na JVM. Jsou to:

- `File.separatorChar` – oddělovač adresářů jako `char`
- `File.separator` – oddělovač adresářů jako `String`
- `File.pathSeparatorChar` – oddělovač cest jako `char`
- `File.pathSeparator` – oddělovač cest jako `String`

Běžně používaný je oddělovač adresářů – musí být použit v každé absolutní cestě.

Oddělovač cest se používá, zadáváme-li více cest najednou, např.:

```
c:\jdk1.2\bin;c:\jdk1.2\lib;
```

Kromě toho lze pro zjištění aktuálního adresáře použít volání:

```
System.getProperty("user.dir")
```

které vrátí úplnou cestu do právě aktuálního adresáře jako instanci třídy String.

Poznámka:

Pokud vytváříme programy, které pracují se soubory pouze v aktuálním adresáři, nemusíme nic z dosud popsaných prostředků používat.

Takové programy jsou ale spíše výjimkou než pravidlem. □

Pro systém MS Windows je nastavení následující:

- File.separatorChar \
- File.separator \
- File.pathSeparatorChar ;
- File.pathSeparator ;
- user.dir C:\java\ujj\soubory

17.2 Vytvoření instance třídy File

Třída `File` má tři konstruktory. Všechny tři vyžadují jméno souboru nebo adresáře.

Pozor:

To, že se vytvoří instance využívající jméno souboru nebo adresáře, ještě neznamená, že uvedený soubor nebo adresář existuje fyzicky na disku! Existovat může, ale také nemusí, což volání konstruktoru nijak neovlivní. □

Příklad 156:

V příkladu je zmínován soubor `a.txt`. Bez ohledu na to, zda je či není v aktuálním adresáři, bude výpis této části programu stejný. V příkladu je použito tří konstruktorů. První používá absolutní cestu a jméno souboru (dva parametry), druhý relativní cestu do adresáře `TMP`, oddělovač adresářů a jméno souboru, které spojuje do jednoho parametru. Třetí využívá pouze jméno souboru.

Máme-li instanci třídy `File`, lze z ní získat informace o úplné cestě, jménu souboru nebo adresáře a o rodičovském adresáři.

Oddělovač cest se používá, zadáváme-li více cest najednou, např.:

```
c:\jdk1.2\bin;c:\jdk1.2\lib;
```

Kromě toho lze pro zjištění aktuálního adresáře použít volání:

```
System.getProperty("user.dir")
```

které vrátí úplnou cestu do právě aktuálního adresáře jako instanci třídy String.

Poznámka:

Pokud vytváříme programy, které pracují se soubory pouze v aktuálním adresáři, nemusíme nic z dosud popsaných prostředků používat. Takové programy jsou ale spíše výjimkou než pravidlem. □

Pro systém MS Windows je nastavení následující:

- File.separatorChar \
- File.separator \
- File.pathSeparatorChar ;
- File.pathSeparator ;
- user.dir C:\java\ujj\soubory

17.2 Vytvoření instance třídy File

Třída File má tři konstruktory. Všechny tři vyžadují jméno souboru nebo adresáře.

Pozor:

To, že se vytvoří instance využívající jméno souboru nebo adresáře, ještě neznamená, že uvedený soubor nebo adresář existuje fyzicky na disku! Existovat může, ale také nemusí, což volání konstruktoru nijak neovlivní. □

Příklad 156:

V příkladu je zmiňován soubor a.txt. Bez ohledu na to, zda je či není v aktuálním adresáři, bude výpis této části programu stejný. V příkladu je použito tří konstruktorů. První používá absolutní cestu a jméno souboru (dva parametry), druhý relativní cestu do adresáře TMP, oddělovač adresářů a jméno souboru, které spojuje do jednoho parametru. Třetí využívá pouze jméno souboru.

Máme-li instanci třídy File, lze z ní získat informace o úplné cestě, jménu souboru nebo adresáře a o rodičovském adresáři.

```
String aktDir = System.getProperty("user.dir");
File soubAbs = new File(aktDir, "a.txt");
File soubRel = new File("TMP" + File.separator + "a.txt");
File soub = new File("a.txt");

System.out.println(soubRel.getAbsolutePath());
System.out.println(soubRel.getName());
System.out.println(soubRel.getParent());

System.out.println(soubAbs.getAbsolutePath());
System.out.println(soubAbs.getName());
System.out.println(soubAbs.getParent());
```

Vypíše:

```
C:\java\ujj\soubory\TMP\a.txt
a.txt
TMP
C:\java\ujj\soubory\a.txt
a.txt
C:\java\ujj\soubory
```

Poznámka:

Výpis pro instanci `soub` by byl následující:

```
C:\java\ujj\soubory\a.txt
a.txt
null
```

`null` je vypsán proto, že v konstruktoru `File("a.txt")` nebyla uvedena ani relativní cesta.



17.3 Vytvoření souboru nebo adresáře

Máme-li instanci třídy `File`, lze otestovat, zda soubor nebo adresář daného jména v souborovém systému na disku existuje. K tomu lze použít booleovskou metodu `exists()`.

Nový soubor lze vytvořit voláním `createNewFile()` a nový adresář pomocí `mkdir()`. Existují-li již na disku, lze rozlišit, zda se jedná o soubor nebo adresář voláním Booleovských metod `isFile()`, případně `isDirectory()`.

Potřebujeme-li vytvořit najednou více vnořených adresářů, použijeme metodu `makedirs()`.

Příklad 157:

```

String aktDir = System.getProperty("user.dir");
File novySou = new File("b.txt");
File novyAdr = new File("TMP");
if (novySou.exists() == true)
    System.out.println("b.txt existuje");
else
    novySou.createNewFile();
if (novySou.isFile() == true)
    System.out.println("b.txt je soubor");

if (novyAdr.exists() == true)
    System.out.println("adresar TMP existuje");
else
    novyAdr.mkdir();
if (novyAdr.isDirectory() == true)
    System.out.println("TMP je adresar");

```

Poznámka:

Metoda `createNewFile()` může vyvolat výjimku `IOException`. □

17.4 Práce se souborem nebo adresářem

O existujícím souboru (nebo adresáři) lze zjistit několik důležitých věcí, např. jeho délku – metoda `length()`. Dále pak datum poslední modifikace – `lastModified()`, která ovšem toto datum vrací v počtu milisekund od 1. 1. 1970 a je tedy většinou nutné jej převést na smysluplnější údaj. V následujícím příkladě je proto použita instance třídy `java.util.Date`.

Soubor i adresář lze přejmenovat – `renameTo()`, ale jen pomocí jiné instance třídy `File`. Soubor a prázdný adresář lze také vymazat – `delete()`.

Příklad 158:

```

File soub = new File("b.txt");
File adr = new File("TMP");

System.out.println(new Date(soub.lastModified()));
System.out.println(new Date(adr.lastModified()));

```

```
System.out.println(soub.length());
System.out.println(adr.length());

File jiny = new File("c.txt");
soub.renameTo(jiny);
adr.renameTo(new File("TMP-OLD"));

soub.delete();      // nevymaže c.txt
adr.delete();      // nevymaže TMP-OLD
jiny.delete();     // skutečné vymazání c.txt
```

Vypíše:

Tue Feb 29 14:25:07 GMT+01:00 2000

Tue Feb 29 14:25:07 GMT+01:00 2000

4

0

Pozor:

Po přejmenování souboru nebo adresáře se toto jméno změní pouze na disku, ale ne v datech instance `soub` nebo `adr`. To znamená, že pomocí těchto instancí již nelze přejmenovaný soubor nebo adresář nijak ovlivňovat. Například nelze soubor nebo adresář pomocí `soub` nebo `adr` smazat. Pro vymazání souboru `c.txt` se musí použít instance `jiny`. Přejmenovaný adresář `TMP-OLD` již vymazat nelze, protože nemáme příslušnou referenční proměnnou. □

17.5 Výpis adresáře

Třída `File` dává k dispozici i metody pro výpis všech souborů a podadresářů v daném adresáři. Tyto metody jsou dvě. Metoda `list()` zjistí pouze jména a uloží je do pole typu `String` – je výhodná v tom případě, že nám stačí opravdu jen jména.

Potřebujeme-li však i další informace, např. velikosti souborů, zda se jedná o podadresáře atd. museli bychom pro každě takto získané jméno vytvořit instanci typu `File`. V tomto případě je vhodnější použít metodu `listFiles()`, která vrátí pole typu `File`.

Příklad 159:

```
String jmenoAktDir = System.getProperty("user.dir");
File aktDir = new File(jmenoAktDir);
```

```

String[] jmena;
jmena = aktDir.list();
for (int i = 0; i < jmena.length; i++)
    System.out.println(jmena[i]);

File[] soubory;
soubory = aktDir.listFiles();
for (int i = 0; i < soubory.length; i++)
    System.out.println(soubory[i].getName() + "\t"
        + soubory[i].length());

```

17.5.1 Selektivní výpis adresáře

Zde by bylo možná srozumitelnější použít nadpis typu „Výpis adresáře pomocí masky“. Metody, které jsou k dispozici, však umožňují selekci souborů a adresářů v mnohem větším rozsahu, než jen pomocí „hvězdičkové konvence“ (*.txt“).

Pro tento komfort je však nutné něco učinit, a to naprogramovat třídu implementující rozhraní `FilenameFilter`. V jeho jediné metodě `accept()` se libovolným nám vyhovujícím způsobem rozliší, zda je soubor pro náš výběr přijatelný, či nikoliv.

Pro vlastní výběr se použijí přetížené metody `list()` nebo `listFiles()`, s výhodami a nevýhodami popsanými v předchozí části.

Příklad 160:

V příkladu jsou uvedeny třídy představující dva filtry. První – `FiltrPripony` – rozlišuje podle přípony a první parametr metody `accept()` vůbec nevyužívá.

Druhý – `FiltrVelikosti` – rozlišuje podle velikosti. Metoda `accept()` využívá třetího typu konstruktoru `File`, kdy lze vytvořit instanci i spojením rodičovského adresáře typu `File` a jména souboru typu `String`. Pro získanou instanci `File` lze pak použít libovolné metody popsané výše, protože tyto soubory nebo adresáře určitě existují. Metody `list()` a `listFiles()` totiž pracují tak, že zjistí jména všech souborů a adresářů, tato jména poskytnou metodě `accept()` k ověření, a teprve po tomto ověření je zahrnou do výsledného seznamu.

```

class FiltrPripony implements FilenameFilter {
    String maska;

```

```
FiltrPripomky(String maska) {
    this.maska = maska;
}

public boolean accept(File dir, String name) {
    if (name.lastIndexOf(maska) > 0)
        return true;
    else
        return false;
}
}

class FiltrVelikosti implements FilenameFilter {
    int velikost;

    FiltrVelikosti(int velikost) {
        this.velikost = velikost;
    }

    public boolean accept(File dir, String name) {
        File f = new File(dir, name);
        if (f.length() > velikost)
            return true;
        else
            return false;
    }
}

public class Soubor {
    public static void main(String[] args) {
        String jmenoAktDir = System.getProperty("user.dir");
        File aktDir = new File(jmenoAktDir);

        String[] jmenna;
        FiltrPripomky filtrPr = new FiltrPripomky(".java");
        jmenna = aktDir.list(filtrPr);
        for (int i = 0; i < jmenna.length; i++)
            System.out.println(jmenna[i]);
    }
}
```

```
File[] soubory;
FiltrVelikosti filtrVel = new FiltrVelikosti(1000);
soubory = aktDir.listFiles(filtrVel);
for (int i = 0; i < soubory.length; i++)
    System.out.println(soubory[i].getName() + "\t"
                        + soubory[i].length());
}
}
```

Poznámka:

Jistě by nebyl problém vytvořit třídu `FiltrAdresaru`, či `FiltrJmen-DelsichNezOsmZnaku`. □

Cvičení:

1. Napište program, který vypisuje jména všech podadresářů v zadaném rodičovském adresáři.
2. Modifikujte předchozí program tak, že budou vypisovány i všechny vnořené adresáře z těchto podadresářů.
3. Napište program, který vypíše podle zadанé masky všechny soubory, které se nacházejí v zadaném rodičovském adresáři.
4. Modifikujte předchozí program tak, že budou vypisovány i soubory ze všech vnořených adresářů z těchto podadresářů.

18 Čtení ze vstupů a zápis na výstupy

Protože Java je programovací jazyk podporující distribuovaný a vícevláknový výpočet, nelze říci, že vstupní a výstupní zařízení je pouze soubor.¹ Informace, kterou program čte nebo zapisuje, může ležet principiálně kdekoliv – v souboru, „na síti“, v paměti přístupné jinému programu (nebo vláknu) atd. Tato informace může mít podobu znaků, skupiny bajtů (např. zvuky, obrázky), objektů...

Java se snaží pomocí svých knihoven postihnout všechny tyto (a další) možnosti, což ovšem činí problém vstupů a výstupů poněkud komplikovaný.

Pro jakýkoliv přenos informace je nutné otevřít ***proud*** (*stream*), což si lze představit jako ***kanál***, kterým informace proudí. Proud je otevírána programem, který požaduje nějakou komunikaci. V případě požadavků na čtení otevřá program ***vstupní proud***, ze kterého pak čte, v případě zápisu pak ***výstupní proud***, do kterého zapisuje. To, jak technicky probíhá čtení nebo zápis, není záležitost programu, ale JVM v závislosti na příslušném operačním systému. Po skončení komunikace je vhodné (spíše ale nutné) otevřený proud zavřít.

Pro čtení a zápis dává Java Core API v balíku `java.io` k dispozici množství tříd. Výraz „množství“ je zde naprosto namístě, protože těchto tříd je v JDK 1.3 více než čtyřicet!

Poznámka:

Na začínajícího programátora působí toto poměrně značné množství zpočátku silně depresivním dojmem.² Naštěstí zdaleka ne všechny třídy ihned nutně potřebujeme pro svoji práci. □

Dále se budeme snažit o systematičtější pohled na celý problém a teprve pak bude vysvětlena práce se soubory, jako se základními externími zdroji informace. Pozornost, i když menší, bude věnována i paměťovým proudům a rourám.

¹Když neuvažujeme samozřejmý vstup z klávesnice a výstup na obrazovku.

²Dodnes si ten pocit vybavuji :-)

18.1 Proudy znaků a proudy bajtů

Zatímco v mnoha programovacích jazycích bylo základní rozlišení na textové a binární proudy, Java má jako základní rozlišení **proudý znaků**, kde základní jednotka dat je 16bitová pro Unicode znaky, a **proudý bajtu**, kde je základní jednotka dat osmibitová. Od této dvojice se odvíjí celá hierarchie tříd.

Pro **znakově orientované proudy** jsou k dispozici základní abstraktní třídy Reader a Writer. Pro **bajtově orientované proudy** jsou to abstraktní třídy InputStream a OutputStream.³

Od těchto čtyř abstraktních tříd se dědí další třídy, které se pak již prakticky používají pro zamýšlený vstup a výstup.

Filosoficky vzato, poskytují všechny čtyři zmíněné třídy jeden druh třikrát přetížené metody, kterou si zde pracovně označíme jako **`rw()`**:

- **`rw()`** – práce s jedním prvkem,
- **`rw(typ[] pole)`** – práce s celým polem prvků,
- **`rw(typ[] pole, int index, int pocet)`** – práce s částí pole prvků (část je specifikována indexem svého prvního prvku a počtem prvků)

Dále uvedeme úplné hlavičky metod pro všechny čtyři abstraktní třídy: **znakově orientované proudy**

- Třída Reader

```
int read()
int read(char[] pole)
int read(char[] pole, int index, int pocet)
```

- Třída Writer

```
void write(int i)
void write(char[] pole)
void write(char[] pole, int index, int pocet)
```

bajtově orientované proudy

- Třída InputStream

```
int read()
```

³Jejich názvy jsou dostatečně významové, takže další vysvětlení není nutné.

```
int read(byte[] pole)
int read(byte[] pole, int index, int pocet)
```

- Třída OutputStream

```
void write(int i)
void write(byte[] pole)
void write(byte[] pole, int index, int pocet)
```

Poznámky:

- Všechny metody read() vracejí hodnotu -1, pokud bylo dosaženo konce proudu (tj. už není co číst).
- U třídy Writer přibývají ještě dvě metody:


```
void write(String retez)
void write(String retez, int index, int pocet)
```
- Kromě těchto „hlavních“ metod obsahují zmíněné třídy také další „pomocné“ metody, z nichž nejdůležitější je metoda close().
- Všechny metody mohou vyvolávat výjimku třídy IOException, případně některé její podtřídy. V příkladech bude výjimka většinou jen deklarována, aby byl program co nejjednodušší. Tento postup nemusí být vždy nevhodnější – viz podrobný rozbor v [16/241].

Pokud to značně zjednodušíme, nepotřebujeme k další práci nic více, než znát metody read() nebo write() a vybrat správnou podtřídu některé z abstraktních tříd. Dále si ukážeme, jaké podtřídy jsou k dispozici.

18.2 Dva různé typy tříd zděděných od základních tříd

Od každé ze čtyř základních abstraktních tříd je odvozeno zhruba deset dalších podtříd. Tyto třídy jsou vždy dvou typů. Prvním typem jsou třídy, které slouží pro fyzický přesun dat na určité zařízení nebo z něj. Druhým typem jsou „pomocné“ třídy, které slouží jen pro určité zpracování dat. To bude při čtení provedeno po fyzickém přesunu dat a v případě zápisu před fyzickým přesunem dat.⁴ Tyto třídy jsou často označovány jako *filtry* nebo *vlastnosti* – toto označení bude používáno dále.

⁴Je velmi důležité uvědomit si rozdíl mezi třídami pro fyzický přesun a „pomocnými“ třídami.

18.2.1 Třídy pro práci se zařízeními

V tabulce vidíte přehled tříd pracujících se zařízeními, tedy tříd, které provádějí fyzický přesun dat. U uvedených zařízení vždy existuje možnost zapisovat znaky nebo bajty.

<i>zařízení</i>	<i>přesun znaků</i>	<i>přesun bajtů</i>
paměť	CharArrayReader	ByteArrayInputStream
	CharArrayWriter	ByteArrayOutputStream
	StringReader	StringBufferInputStream
	StringWriter	
soubor	FileReader	FileInputStream
	FileWriter	FileOutputStream
roura	PipedReader	PipedInputStream
	PipedWriter	PipedOutputStream

Do paměti čteme a z paměti zapisujeme typicky tehdy, když budeme data ještě nějakým způsobem zpracovávat – příklad viz v [18.5/288].

Souborové vstupy a výstupy jsou nejběžnější – budou podrobně popsány v [18.3/273] a ve většině dalších používány.

Vstup a výstup pomocí **roury** (*pipe*) je používán pro komunikaci metod nebo vláken jednoho programu – budou zmíněny v [18.6/289].

18.2.2 Třídy vlastností (filtrů)

Tříd filtrů je celkem 19. Slouží pro úpravu dat, která již byla fyzicky načtena, případně dat, která budou vzápětí fyzicky zapisována. Tyto třídy se „nabalují“ na třídy, které provádí fyzické čtení/zápis a dodávají těmto třídám nějakou **vlastnost**. Proto budou dále nazývány „vlastnostmi“.

Většinou se opět vyskytují dvojmo – jednou pro proudy znaků a jednou pro proudy bajtů. Zde bude uveden jejich přehled a odkazy na části, ve kterých jsou vysvětleny.

- Využití vyrovnávacích pamětí – bufferování – viz [18.4.1/278]
 - BufferedReader a BufferedWriter – pro znaky
 - BufferedInputStream a BufferedOutputStream – pro bajty
- Vrácení načteného znaku zpět do vstupního proudu – viz [18.4.4/279]
 - Existují jen pro vstupní proudy – PushbackReader – pro znaky
 - PushbackInputStream – pro bajty

- Formátovaný výstup – viz [18.4.5/280]
Existují jen pro výstupní proudy – `PrintWriter` – pro znaky
`PrintStream` – pro bajty
- Binární čtení nebo zápis základních datových typů – viz [18.4.8/283]
Existují jen pro bajtové proudy –
`DataInputStream` a `DataOutputStream`
- Binární čtení nebo zápis libovolných objektů – viz [18.4.9/285]
Existují jen pro bajtové proudy –
`ObjectInputStream` a `ObjectOutputStream`

Poznámka:

Kromě nich existují ještě třídy pro konverzi mezi znaky a bajty (`InputStreamReader`, `OutputStreamWriter`), filtrování (`FilterReader`, `FilterWriter` a `FilterInputStream`, `FilterOutputStream`) a spojování vstupních proudů (`SequenceInputStream`), které v této knize nebudou zmiňovány. □

18.3 Čtení ze souboru a zápis do souboru

Dále bude popisováno neformátované čtení a zápis. Formátovaný zápis viz v [18.4.5/280].

18.3.1 Vstupy a výstupy znaků

Pro tyto akce slouží třídy `FileReader` a `FileWriter`. Instance obou těchto tříd lze nejlépe vytvořit pomocí již existující instance třídy `File`. To nám dává možnost otestovat, zda čtený soubor skutečně existuje. Pokud by neexistoval a my to netestovali, vyvolal by konstruktor `FileReader` výjimku `FileNotFoundException`.

Příklad 161:

```
import java.io.*;  
  
public class IoZnaky {  
    public static void main(String[] args) throws IOException {  
        File frJm = new File("a.txt");  
        File fwJm = new File("b.txt");
```

```
if (frJm.exists() == true) {  
    FileReader fr = new FileReader(frJm);  
    FileWriter fw = new FileWriter(fwJm);  
    int c;  
  
    while ((c = fr.read()) != -1)  
        fw.write(c);  
  
    fr.close();  
    fw.close();  
}  
}
```

Poznámka pro programátora v C či C++:

To, že `read()` vrací na konci souboru `-1`, nepůsobí žádný problém, protože je to `-1` na čtyřech bajtech. Není možné, aby se dostala do kolize s maximální hodnotou `char`, která je pouze dvoubajtová. Je to stejný princip jako v C, kde je EOF hodnota typu `int` a `char` je vždy jednobajtový. □

Příklad 162:

Stejný příklad trochu jinak. Využívá se toho, že konstruktor `FileWriter` i `FileReader` může mít jako svůj parametr i jméno souboru. Tato možnost je zde použita pouze pro `FileWriter`. Soubor pro čtení se otevírá klasickým způsobem přes `File`, protože to nám dovoluje zjistit dopředu délku souboru a ten číst v cyklu.

```
public static void main(String[] args) throws IOException {
    File frJm = new File("a.txt");
    FileReader fr = new FileReader(frJm);

    FileWriter fw = new FileWriter("b.txt");

    long delka = frJm.length();
    int c;

    for (long i = 0; i < delka; i++) {
        c = fr.read();
        fw.write(c);
    }
}
```

```
fr.close();
fw.close();
}
```

Poznámka:

Tyto třídy pracují se znaky, tj. s 16bitovými čísly. Když ale zkusíte číst soubor s Unicode textem, zjistíte, že jej čte po bajtech. Většina současných souborových systémů je totiž osmibitová a s šestnáctibitovými znaky nepočítají. Proto tyto proudy překódovávají znaky podle nastaveného kódovacího schématu. O jaké přednastavení se jedná, to je možné zjistit pomocí:

```
System.out.println(System.getProperty("file.encoding"));
nebo
```

```
System.out.println(fw.getEncoding());
které na MS Windows vypíší Cp1250.
```

V [UJJ2] je uvedeno jak lze toto nastavení změnit. □

18.3.2 Vstupy a výstupy bajtů

Pro tyto akce slouží třídy `FileInputStream` a `FileOutputStream`. Porovnáme-li dále uvedený program s programem z předchozí části, zjistíme, že se změnily pouze dvě řádky. Jinak je program zcela stejný a zcela stejný je i jeho výstup.

Příklad 163:

```
import java.io.*;

public class IoBajty {
    public static void main(String[] args) throws IOException {
        File frJm = new File("a.txt");
        File fwJm = new File("c.txt");

        if (frJm.exists() == true) {
            FileInputStream fr = new FileInputStream(frJm);
            FileOutputStream fw = new FileOutputStream(fwJm);

            int c;
            while ((c = fr.read()) != -1)
                fw.write(c);
        }
    }
}
```

```
    fr.close();
    fw.close();
}
}
```

18.3.3 Další dovednosti se soubory

Dále budou popisovány metody ze znakových proudů, přičemž pro bajtové proudy existují naprosto stejné (i stejně pojmenované) metody.

Užitečnými metodami třídy `FileReader` jsou:

- `skip(long pocet)` – přeskočí při čtení zadaný počet znaků
 - `boolean markSupported()` – zjišťuje, zda jsou proudem podporovány následující dvě operace
 - `reset()` – vrátí se na začátek souboru
 - `mark(long platnost)` – označí aktuální pozici v souboru značkou, na kterou se pak bude vracet metoda `reset()`; parametr `platnost` udává, kolik znaků můžeme od aktuální označované pozice ještě přečíst, aniž by značka ztratila platnost

Třída `FileWriter` dává ještě k dispozici:

- `FileWriter(String jmeno, boolean append)` – tento další konstruktor umožňuje při zápisu soubor rozšiřovat (jinak by byl přepsán)
 - `flush()` – okamžité zapsání bufferovaných dat na disk

Příklad 164:

Příklad ukazuje, jak otevřít soubor pro připisování na konec – druhé volání konstruktoru:

`fw = new FileWriter("b.txt", true);`
stávající soubor b.txt nevymaže (jak to udělalo první volání). Příkaz `fr.skip()` přeskočí první polovinu souboru a čte až od druhé. Metody `mark()` a `reset()` zde bohužel nejsou podporovány.

```
public static void main(String[] args) throws IOException {
    File frJm = new File("a.txt");
    FileReader fr = new FileReader(frJm);

    FileWriter fw = new FileWriter("b.txt");

    long delka = frJm.length();
    int c;

    // prvni kopirovani od zacatku
    for (long i = 0; i < delka; i++) {
        c = fr.read();
        fw.write(c);
    }

    fr.close(); // poněkud nepřehledný přesun na zač. souboru
    fr = new FileReader(frJm);
//    fr.reset(); není tímto proudem podporováno

    fw.close();
    fw = new FileWriter("b.txt", true);

    // druhe kopirovani z poloviny
    fr.skip(delka / 2);
    while((c = fr.read()) != -1)
        fw.write(c);

    fr.close();
    fw.close();
}
```

18.4 Třídy vlastností

Pro všechny dále uvedené vlastnosti platí, že příslušnou vlastnost (např. bufferování) dodáme otevřenému proudu tím, že jej použijeme jako parametr konstruktoru třídy té které vlastnosti. Protože všechny třídy mají společného předka, je možné používat metody `read()` nebo `write()` bez dalších úprav.

18.4.1 Vlastnost: bufferování

Použití *vyrovnávací paměti* (*buffer*) je známá technika, jak urychlit vstup nebo výstup. Tuto techniku lze použít jak pro proudy znaků – třídy `BufferedReader` a `BufferedWriter`, tak i pro proudy bajtů – třídy `BufferedInputStream` a `BufferedOutputStream`.

Úspora času získaná použitím těchto metod není zanedbatelná. Byly testovány dva programy, které četly soubor o velikosti 99872 B nebufferovaně⁵ a bufferovaně⁶. V případě nebufferovaného čtení bylo dosaženo času 5598 msec a v případě bufferovaného 190 msec, což představuje asi 30násobné zrychlení⁷.

18.4.2 Vlastnost: čtení po řádcích

Pomocí metody `readLine()` třídy `BufferedReader` lze číst soubor po řádcích. Metoda neukládá znak(y) konce řádky do výsledného řetězce. Z tohoto důvodu je použita metoda `newLine()`, která při zápisu řetězce do výstupního souboru zajistí správné ukončení řádky.

Příklad 165:

```
FileReader fr = new FileReader("a.txt");
BufferedReader in = new BufferedReader(fr);
FileWriter fw = new FileWriter("b.txt");
BufferedWriter out = new BufferedWriter(fw);
String radka;

while((radka = in.readLine()) != null) {
    System.out.println(radka);
    out.write(radka);
    out.newLine();
}

fr.close();
out.close();
```

Poznámka:

Všimněte si závěrečného: `out.close()`

⁵Byl to Io8.java – viz na disketě.

⁶Byl to Io9.java – viz na disketě.

⁷Všechny časy byly zprůměrovány z několika pokusů.

Výstup je bufferovaný a kdyby se použilo `fw.close();` uzavřel by se výstupní soubor otevřený přes `FileWriter`, aniž by se do něj zapsala data uložená v bufferu. Výsledkem by bylo, že by soubor `b.txt` byl prázdný. □

Pozor:

Ve třídě `BufferedInputStream` není metoda `readLine()` – ta čte znaky, ne bajty. □

18.4.3 Vlastnost: výběrové čtení po řádcích

Od třídy `BufferedReader` je odvozena třída `LineNumberReader`, která kromě již známě metody `readLine()` nabízí i metodu `getLineNumber()`. Ty slouží k „číslování“ řádek, tj. pomocí `getLineNumber()` můžeme kdykoliv zjistit, kolik jsme již řádek přečetli. Pomocí `setLineNumber()` lze nastavit jiné číslo řádky – viz na disketu `Io12.java`.

18.4.4 Vlastnost: vrácení přečteného znaku

Často se nám stane, že čteme a to, že jsme již měli přestat číst, poznáme podle toho, že jsme už přečetli „něco navíc“. Toto „něco navíc“ by ale chybělo při dalším čtení. Z tohoto důvodu jsou velmi užitečné třídy `PushbackReader` a `PushbackInputStream`, které umožňují nadbytečně přečtený znak, bajt nebo pole vrátit zpět do vstupního proudu.

Příklad 166:

```
FileReader fr = new FileReader("a.txt");
PushbackReader in = new PushbackReader(fr);
int c;

c = in.read();
System.out.print((char) c);
in.unread(c);
c = in.read();
System.out.print((char) c);
```

Vypíše např.:

BB

tedy dva stejné znaky.

18.4.5 Vlastnost: formátování výstupu

Vše, co bylo o výstupu dosud řečeno, se týkalo výstupu neformátovaného. To znamená, že ve výstupním souboru se ocitla naprostě stejná data, jako byla v proměnných, z nichž byla zapisována – viz též [4.2/67].

To tedy prakticky znamená, že při zápisu proměnných typu `int` vytvoříme „binární“ soubor⁸, který nebude běžnými editory čitelný. Z tohoto důvodu často potřebujeme vytvořit i „textový“ soubor, jinými slovy řečeno, potřebujeme zapisovat formátovaně.

Tak, jako pracuje známý `System.out.println()`, pracují i formátované výstupy – přesněji řečeno metoda `System.out.println()` je právě formátovaný výstup.

Potřebujeme-li tedy do textového souboru zapsat libovolná data stejně jako na obrazovku, použijeme bud' `PrintWriter` pro znaky nebo `PrintStream` pro bajty.

Příklad 167:

Následující příklad ukáže rozdíl mezi formátovaným a neformátovaným výstupem. Bude zapisováno deset čísel od 65 do 74, které lze chápout i jako ASCII kódy znaků A, B, ..., I.

```
FileWriter fwForm = new FileWriter("form.txt");
PrintWriter form = new PrintWriter(fwForm);
FileWriter fwNeform = new FileWriter("neform.txt");

for (int i = 65; i < 75; i++) {
    System.out.print(i + " ");
    form.print(i + " ");
    fwNeform.write(i);
}

form.close();
fwNeform.close();
```

Obsah souboru form.txt:

65 66 67 68 69 70 71 72 73 74

Obsah souboru neform.txt:

ABCDEFGHIJ

⁸Každý `int` ale bude zapsán jako jeden znak.

18.4.6 Vlastnost: formátování výstupu s řádkovým bufferováním

Zapisujeme-li formátovaně, bývá často zvykem, že se zápis na výstupní zařízení fyzicky provede až po vytisknutí znaku(ů) konce řádky. Toto se nazývá **řádkové bufferování** a nemá přímou souvislost s třídou BufferedWriter popsanou výše.

Pokud použijeme konstruktor:

```
PrintWriter(Writer out)
```

pak je řádkové bufferování vypnuto, tzn. metoda `print()` vytiskne (zapíše) každý znak okamžitě po svém vyvolání.

Použijeme-li konstruktor:

```
PrintWriter(Writer out, boolean autoFlush)
```

pak při hodnotě `false` bude výpis pozdržen až do výpisu znaku(ů) konce řádky a pak vypsána celá řádku najednou. To má výhodu rychlejšího výstupu, ale nevýhodu, že zapisovaná data nejsou okamžitě na cílovém zařízení.

Při hodnotě `true` je každý vypisovaný znak okamžitě zapsán a výhody a nevýhody jsou právě opačné.

18.4.7 Formátovaný vstup

Tato možnost není bohužel přímo podporována. Ovšem se soubory, ve kterých jsou v textové podobě (tedy zapsány formátovaně) čísla, se setkáváme poměrně často. Následující příklad ukáže, jak si s takovým souborem poradit.

Příklad 168:

Programem `Zapis.java` (viz na disketě) byla do souboru `buf.txt` zapisována čísla od jedné do N , každé číslo na novou řádku. Uvedený program tato čísla ze souboru přečte a vypíše jejich součet.

```
public static void main(String[] args) throws IOException {
    FileReader fr = new FileReader("buf.txt");
    BufferedReader in = new BufferedReader(fr);
    String radka;
    int k, suma = 0;
    while((radka = in.readLine()) != null) {
        k = Integer.valueOf(radka).intValue();
        suma += k;
    }
}
```

```

System.out.println("Součet je: " + suma);
fr.close();
}

```

Příklad 169:

Pokud by na jedné řádce souboru bylo více čísel, byl by program jen o málo složitější. Soubor radky.txt má následující obsah:

```

0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29

```

Zadání je stejné, jako v předchozím případě – vypočítat sumu všech čísel.

Zde s výhodou využijeme třídu `java.util.StringTokenizer`, která dokáže rozdělit řetězec obsahující mezery (mimo jiné) na několik řetězců, kde konec nového řetězce je na místě původní mezery.

```

public static void main(String[] args) throws IOException {
    FileReader fr = new FileReader("radky.txt");
    BufferedReader in = new BufferedReader(fr);
    String radka, cislo;
    int k, suma = 0;

    while((radka = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(radka);
        while (st.hasMoreTokens()) {
            cislo = st.nextToken();
            k = Integer.valueOf(cislo).intValue();
            suma += k;
        }
    }
    System.out.println("Součet je: " + suma);
    fr.close();
}

```

Poznámka:

Pokud by byla čísla v souboru radky.txt oddělena např. znaky ; nebo % nebo mezerou, použil by se jen jiný konstruktor:

```
StringTokenizer st = new StringTokenizer(radka, " ;%"); □
```

18.4.8 Vlastnost: neformátovaný vstup a výstup základních datových typů

Ve všech předchozích příkladech jsme zapisovali neformátovaně pouze znaky nebo bajty. Pokud jsme zapisovali `int` jako v [18.4.5/280], pak byl konvertován na znak. Potřebujeme-li čist nebo zapisovat binární (tj. neformátované) obsahy jednotlivých proměnných jiných primitivních datových typů⁹ (např. `int`, `double`, atd.), použijeme třídy `DataInputStream` a `DataOutputStream`.

Poznámka:

Tato možnost je k dispozici jen pro bajtové proudy, pro znakové proudy nemá význam. □

V těchto třídách jsou samozřejmě implementovány známé metody `read()` a `write()`. Kromě nich však existují metody pro čtení a zápis všech základních datových typů, tedy např. `readBoolean()`, `writeBoolean()`, ..., `readDouble()`, `writeDouble()`.

Ještě navíc jsou implementovány metody pro čtení¹⁰ neznaménkových dat – nepochybň pro případnou kompatibilitu datových souborů vytvořených v jiných programovacích jazycích. Jedná se o:

- `int readUnsignedByte()` – čte bajt jako neznaménkové číslo (rozsah 0 až 255)
- `int readUnsignedShort()` – čte dva bajty jako neznaménkové číslo (rozsah 0 až 65535)

Poznámka:

Chceme-li zapisovat neznaménkové bajty (vysvětlení viz v [3.4.1/36]), převedeme příslušný pomocný `int` na znaménkový bajt a ten zapisujeme. Na data, která budou na disku, se pak můžeme dívat i jako na neznaménkové bajty. □

Jako poslední typ dat, které tyto třídy zpracovávají, je implementován typ UTF-8, který slouží k uložení znaků v kódu Unicode (tj. jakýchkoliv řetězců v Javě) na osmibitová výstupní zařízení. Podrobně o UTF viz [UJJZ2], zde jen stručně řečeno – pokud uložíte do souboru znaky v kódu UTF-8, nemusíte si dělat žádné starosti s odlišným kódováním češtiny

⁹Například potřebujeme uložit na disk pole 100 prvků typu `int` a za něj pole 15 `double`.

¹⁰Pro zápis ne – Java nemá důvod sama vytvářet takové „relikty“ ; -)

na jiných výpočetních systémech, samozřejmě za předpokladu, že je tam zase jako UTF-8 načtete.

Metody jsou `void writeUTF(String s)` a `String readUTF()`.

Příklad 170:

V příkladu se do souboru `data.bin` zapíše náhodně zvolené číslo v rozsahu 2 až 10. Toto číslo udává počet dále vygenerovaných čísel. Za něj se zapíše tento počet náhodných čísel typu `int` a za ně dvě čísla typu `double`. Všechna čísla se pak ze souboru `data.bin` opět přečtou. Na obrazovku se opisují data pro kontrolu.

```
public static void main(String[] args) throws IOException {
    FileOutputStream fwJm = new FileOutputStream("data.bin");
    DataOutputStream fw = new DataOutputStream(fwJm);
    int k, pocet;
    while((pocet = (int) (10.0 * Math.random())) < 2)
    {
        fw.writeInt(pocet);

        for (int i = 0; i < pocet; i++) {
            k = (int) (1000.0 * Math.random());
            System.out.print(k + " ");
            fw.writeInt(k);
        }
        fw.writeDouble(Math.PI);
        fw.writeDouble(Math.E);
        System.out.println("\n" + Math.PI + " " + Math.E);
        fwJm.close();
    }

    FileInputStream frJm = new FileInputStream("data.bin");
    DataInputStream fr = new DataInputStream(frJm);
    pocet = fr.readInt();
    for (int i = 0; i < pocet; i++) {
        k = fr.readInt();
        System.out.print(k + " ");
    }
    double pi = fr.readDouble();
    double e = fr.readDouble();
    System.out.println("\n" + pi + " " + e);
    frJm.close();
}
```

Vypíše na obrazovku například:

```
304 25 977 747 630
3.141592653589793 2.718281828459045
304 25 977 747 630
3.141592653589793 2.718281828459045
```

Soubor `data.bin` má délku 40 bajtů, protože je v něm uloženo:
 $(1 + 5) \times 4 + 2 \times 8 = 40$ (int má 4 bajty a double 8 bajtů).

18.4.9 Vlastnost: serializace objektů

V předchozí části bylo popisováno, jak zapsat do souboru hodnoty základních datových typů. Pokud budeme vytvářet složitější programy, časem se nevyhneme nutnosti ukládat na disk i objekty, které jsou složitější než základní typy. V podstatě se jedná o to, jak uložit na disk jednotlivé instance a tím umožnit jejich „přežití“ i po skončení běhu programu. Takovýmto objektům se říká *perzistentní objekty* a procesu jejich ukládání *serializace* (*serialization*).

Celá problematika je značně složitá (podrobně viz např. v [Cam1]), proto bude dále uvedeno jen minimální množství informace, nutné pro pochopení hlavního rozdílu oproti již zmíněným trídám `DataInputStream` a `DataOutputStream`.

Proces serializace je zajišťován třídami `ObjectInputStream` a `ObjectOutputStream`, které mají kromě množství jiných metod i metody `readObject()` a `writeObject()`. Pomocí nich se zapíší nebo přečtou všechna data, které daný objekt ke své činnosti v paměti JVM potřebuje. Protože v Javě je všechno objekt, je možné po určitém úsilí vytvořit na disku téměř kompletní „stav“ programu a ten pak z disku někdy později obnovit v paměti.

Pozor:

Tímto způsobem lze číst nebo zapisovat pouze objekty tříd, které implementují rozhraní `java.io.Serializable`. To je splněno pro většinu tříd z Java Core API. Budeme-li však chtít takto pracovat s objekty vlastní třídy, musíme tuto implementaci zajistit, což není nijak obtížné, protože toto rozhraní nemá žádné metody. Bez implementace bude vyvolána výjimka `NotSerializableException`. Naše třída však musí mít datové prvky jen základních datových typů nebo typů, které již implementovaly rozhraní `Serializable`. □

Příklad 171:

Program uloží do souboru instanci třídy `java.util.Date`, která obsahuje čas svého vzniku. Když tuto instanci zruší a posléze opětovně přečte, bude se jednat o obsahově tutéž instanci, čili datum se nezmění. Totéž provede i pro instanci naší třídy `ImplSerializable`.

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

class ImplSerializable implements Serializable {
    Date d;
    ImplSerializable() {
        d = new Date();
    }
}

public class Io21 {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        FileOutputStream fwJm = new FileOutputStream("datum.bin");
        ObjectOutputStream fw = new ObjectOutputStream(fwJm);

        Date d = new Date();
        System.out.println("Vznik: " + d);
        ImplSerializable impl = new ImplSerializable();
        System.out.println(impl.d.toString());
        fw.writeObject(d);
        fw.writeObject(impl);
        fwJm.close();
        d = null;          // zrusení instance
        impl = null;

        FileInputStream frJm = new FileInputStream("datum.bin");
        ObjectInputStream fr = new ObjectInputStream(frJm);
        d = (Date)fr.readObject();
        impl = (ImplSerializable)fr.readObject();
        fwJm.close();
        System.out.println("Cteni: " + d);
        System.out.println(impl.d.toString());
    }
}
```

Vypíše např.:

Vznik: Thu Nov 18 20:47:58 GMT+01:00 1999

Thu Nov 18 20:47:58 GMT+01:00 1999

Ctení: Thu Nov 18 20:47:58 GMT+01:00 1999

Thu Nov 18 20:47:58 GMT+01:00 1999

Jako objekty lze uložit i základní datové typy, např. příkazem:

```
writeObject(new Integer(5));
```

Pokud se budeme inspirovat příkladem z [18.4.8/283] a uložíme do souboru šest čísel typu `int` a dvě čísla typu `double`, nebude mít soubor velikost 40 bajtů jako minule, ale 198 bajtů, protože se kromě hodnot `int` a `double` uloží také další informace.

18.4.10 Seskupování vlastností

Občas se stane, že by se nám hodilo použít některou vlastnost z dříve popsaných tříd v kombinaci s jinou vlastností. Například vlastnost bufferování (třída `BufferedWriter`), která urychluje výstup, by bylo vhodné spojit s vlastností formátovaného výstupu (třída `PrintWriter`). Pro tento účel není nutné vytvářet žádné další třídy postupným děděním obou tříd. Je důležité si uvědomit, že obě třídy mají společného předka, takže je lze velmi snadno „zřetezit“.

Příklad 172:

Následující příklad ukazuje, jak lze zajistit vlastnost bufferování formátovaného výstupu.

```
FileWriter fBuf = new FileWriter("buf.txt");
BufferedWriter bBuf = new BufferedWriter(fBuf);
PrintWriter pBuf = new PrintWriter(bBuf);

for (int i = 0; i < 500000; i++)
    pBuf.println(i);

pBuf.close();
```

Poznámka:

Tento postup byl porovnáván s nebufferovaným formátovaným výstupem, který samozřejmě produkoval zcela stejný výstupní soubor. Oba výstupní soubory měly velikost 3 888 890 B. S využitím bufferování se vytvořil za 9 110 msec a bez bufferování za 13 670 msec, tedy o 50% pomaleji. □

18.5 Vstup a výstup do paměti

V [18.2.1/272] byly popisovány tři typy zařízení, kam/odkud mohou proudy zapisovat/číst. Dosud bylo výhradně popisováno jen zařízení typu „file“, tedy vstupy a výstupy do souboru. V této části bude krátká ukázka, jak využít všech dosud popisovaných možností pro zařízení typu paměť. Nebudou samozřejmě popisovány znova všechny třídy, ukázka se bude týkat pouze formátovaného zápisu do instance třídy `String`. V případě potřeby lze (doufejme ; -) snadno odvodit, jak naprogramovat i ostatní možnosti.

Poznámka:

Většinu vlastností popisovaných dříve lze bez jakýchkoli úprav použít i pro práci s pamětí. □

Příklad 173:

Následující (z důvodu jednoduchosti silně vyumělkovaný) příklad ukaže způsob formátovaného zápisu (stejně jako na obrazovku) do řetězcového proudu. Je třeba si uvědomit, že další a další zápisy se **přidávají** na konec proudu, takže pokud se výsledný řetězec tiskne v cyklu, informace se opakují.

```
public static void main(String[] args) throws IOException {
    StringWriter sProud = new StringWriter();
    PrintWriter form = new PrintWriter(sProud);

    for (int i = 1; i <= 3; i++) {
        form.print(i + ". ahoj " + (4 - i) + "\n");
        System.out.println(sProud);
    }
    form.close();
}
```

Vypíše:

1. ahoj 3

1. ahoj 3

2. ahoj 2

1. ahoj 3

2. ahoj 2

3. ahoj 1

18.6 Vstup a výstup do roury

S principem **roury** (*pipe*) jsme se již pravděpodobně setkali na úrovni komunikace programů spuštěných najednou z příkazové řádky terminálu, například v podobě příkazu:

```
dir | more
```

který zajistí, že výpis adresáře (provedený pomocí `dir11`) bude při zaplnění obrazovky čekat na stisk klávesy (pomocí `more`).

To tedy znamená, že `dir` neposílá svůj výstup přímo na obrazovku, ale na vstup programu `more`, který teprve obdržené hodnoty vypisuje na obrazovku a případně čeká na stisk klávesy. Toto předání dat z `dir` do `more` se uskutečňuje právě pomocí roury.

Poznámka:

Pokud používáte Unix, pak vám využití roury jistě připadá jako naprostě běžná věc. □

V předchozím příkladě spolu pomocí roury komunikovaly dva programy. V Javě lze zařídit (využitím tříd `PipedWriter` a `PipedReader`, případně `PipedInputStream` a `PipedOutputStream`) otevření roury i v rámci jednoho programu. Pak je možné, aby byla přes tuto rouru předávána data mezi metodami nebo vláknny.

Jaká je výhoda tohoto způsobu? Roura je ideální zařízení pro uložení dočasných dat, která by se jinak ukládala nejčastěji do pomocného souboru a ten by se po skončení programu vymazal. Protože však roura vzniká v paměti, má oproti souboru výhodu vyšší rychlosti.

Příklad 174:

Celý princip si ukážeme na (opět vyumělkovaném) příkladě.¹² Naším úkolem bude číst soubor, který obsahuje na každé řádce jedno celé číslo, a vypsat z něj jen ta čísla, která mají jako první znak `1`. Z těchto čísel pak dalším čtením vybereme pouze ta, jejichž třetí znak je `9`.

Pro tento účel vytvoříme metodu `vyber()`, která vrací proud typu `Reader` a má tři formální parametry. První vstup je typu `Reader` a určuje, z jakého proudu bude metoda `vyber()` cist. Druhý index říká, jaké pořadí znaku na řádce nás zajímá, a třetí znak, jaká hodnota znaku nás zajímá. Celý princip roury spočívá v řádcích:

¹¹Nebo `ls` v UNIXu.

¹²Tento jednoduchý úkol by bylo možné naprogramovat mnohem snadněji, ale zde se jedná o princip použití.

```
PipedWriter rouraVystup = new PipedWriter();
PipedReader rouraVstup = new PipedReader(rouraVystup);
které vytvářejí začátek a konec jedné roury. Jakmile do roury pomocí
rouraVystup něco zapíšeme, budeme to moci pomocí rouraVstup někdy
přečíst.
```

Řádka: BufferedReader bufVstup = new BufferedReader(vstup);
je použita jen proto, abychom mohli použít metodu readLine().

Stejně tak:

```
PrintWriter formRouraVystup = new PrintWriter(rouraVystup);
dodává rouře vlastnost formátovaného zápisu – můžeme použít println().
```

V cyklu se již známým způsobem čtou řádky ze vstupu, a pokud vyhovují požadavkům, zapisují se do roury. Nakonec je výstup roury uzavřen, ale vstup roury je návratovou hodnotou metody vyber(). To pak umožní v metodě main() vnořit volání metody vyber(), kdy výstup z jedné metody je vstupem pro druhou:

```
Reader jednaDevet = vyber(vyber(fr, 0, '1'), 2, '9');
```

Zde jsou již alespoň částečně vidět výhody tohoto přístupu. Jednak není problém vnořit další volání vyber() a tak např. hledat znak 2 na pozici 6. Další výhodou je, že výsledek je ve standardním proudu (zde jednaDevet), odkud může být libovolně dále zpracován.¹³

Dále je třeba si uvědomit, že metoda vyber() může data – před jejich zapsáním do roury – libovolně upravit, takže další volání vyber() může pracovat nad pozměněnými nebo i zcela jinými daty.

```
import java.io.*;

public class Io30 {
    public static Reader vyber(Reader vstup, int index, char znak)
        throws IOException {
        BufferedReader bufVstup = new BufferedReader(vstup);

        PipedWriter rouraVystup = new PipedWriter();
        PipedReader rouraVstup = new PipedReader(rouraVystup);

        PrintWriter formRouraVystup = new PrintWriter(rouraVystup);

        String radka;
```

¹³ Zde je tradičně vypisován na obrazovku ; -)

```
while((radka = bufVstup.readLine()) != null) {  
    if (index < radka.length()  
        && radka.charAt(index) == znak)  
        formRouraVystup.println(radka);  
}  
formRouraVystup.close();  
return rouraVstup;  
}  
  
public static void main(String[] argv) throws IOException {  
    FileReader fr = new FileReader("data120.txt");  
    Reader jednaDevet = vyber(vyber(fr, 0, '1'), 2, '9');  
    fr.close();  
  
    String radka; // závěrečný tisk  
    BufferedReader br = new BufferedReader(jednaDevet);  
    while((radka = br.readLine()) != null) {  
        System.out.println(radka);  
    }  
    br.close();  
}
```

Protože obsahem souboru jsou čísla od 1 do 120, vypíše tento program:

109

119

18.7 Soubory s náhodným přístupem

Vše, co bylo dosud popisováno, se týkalo proudů, které mohou být zpracovávány pouze sekvenčně.¹⁴

Občas potřebujeme přistupovat k souboru skutečně náhodným přístupem, tj. číst nebo zapisovat do libovolného místa v souboru. Případně potřebujeme provádět totéž, ale ze souboru potřebujeme současně číst a současně do něj i zapisovat.¹⁵

¹⁴Metodu skip() nelze považovat za náhodný přístup, protože umožňuje jen posun dopředu, a metody mark() a reset() nefungují zdaleka ve všech proudech.

¹⁵Současný vstup a výstup není pomocí proudů principiálně možno zajistit, protože u nich je striktně oddělen vstup od výstupu.

Jako typický příklad lze uvést soubor, který je komprimovaným archivem (soubory typu .arj, .zip, ...). V nich běžně potřebujeme nalézt jeden komprimovaný soubor a nahradit jej novým.

Pro tento účel poskytuje Java třídu `RandomAccessFile`, která je v hierarchii tříd zcela nezávislá na třídách výše popisovaných proudů.

Poznámka pro programátora v C či C++:

Práce se soubory pomocí této třídy se nejvíce blíží práci se soubory v C. □

Třída má dva konstruktory, každý má dva parametry. V prvním konstruktoru se využívá známá třída `File`, ve druhém je uvedeno přímo jméno souboru. Druhý parametr obou konstruktorů je shodný a určuje mod otevření souboru. Lze použít jen mody "r" nebo "rw", tedy soubor bude možné jen číst, nebo do něj půjde i zapisovat.

Poznámka pro programátora v C či C++:

Všechny ostatní kombinace modů (např. "rb", "wb", ale i "wr") způsobí vznik výjimky `IllegalArgumentException`. □

Například soubor `a.bin` lze otevřít jen pro čtení příkazem:

```
RandomAccessFile fr = new RandomAccessFile("a.bin", "r");  
a pro čtení i zápis příkazem
```

```
RandomAccessFile frw = new RandomAccessFile("a.bin", "rw");
```

Pohyby v otevřeném souboru jsou zajištěny pomocí metod:

- `length()` – vrátí velikost souboru (vhodné, pokud jsme tuto hodnotu nezískali již z instance třídy `File`),
- `skipBytes(int n)` – přeskočí následujících `n` bajtů,
- `getFilePointer()` – vrací aktuální pozici v souboru,
- `seek(long pozice)` – nastaví aktuální pozici v souboru na pozice měřeno vždy od začátku souboru,
- `setLength(long velikost)` – nastaví novou velikost souboru, což jej umožní zvětšit, ale i oříznout.

V této třídě jsou implementovány tři důvěrně známé metody `read()` a tři metody `write()` – viz [18.1/270]. Kromě nich však existují metody pro čtení a zápis všech základních datových typů, tedy např. `readBoolean()`,

`writeBoolean()`, ..., `readDouble()`, `writeDouble()`, stejně jako v `DataInputStream` a `DataOutputStream`, včetně metod `readUnsignedByte()`, `readUnsignedShort()` a `readUTF()` a jejich `writeXXX()` protějšků.

Navíc je implementována i metoda `String readLine()`, která čte řádku až do znaku konce řádky. Její přímý protějšek typu `writeLine()` není, ale nahrazuje ji např. `writeChars(String s)`.

Pozor:

Všechny údaje jsou zapisovány neformátovaně!

□

Příklad 175:

Příklad použití bude modifikovaný příklad z [18.4.8/284], kdy při čtení přeskočíme všechna čísla typu `int`, kromě posledního. To je přepsáno v souboru novou hodnotou 1234 a ta je znova přečtena. Nakonec je přeskočeno první číslo typu `double` a přečteno až druhé.

```
public static void main(String[] args) throws IOException {
    RandomAccessFile frw = new RandomAccessFile("a.bin", "rw");
    int k, pocet = 5;
    long posun;

    frw.writeInt(pocet);
    for (int i = 0; i < pocet; i++) {
        k = (int) (1000.0 * Math.random());
        System.out.print(k + " ");
        frw.writeInt(k);
    }
    frw.writeDouble(Math.PI);
    frw.writeDouble(Math.E);
    System.out.println("\n" + Math.PI + " " + Math.E);
    System.out.println("Velikost souboru: "
        + frw.getFilePointer());
    System.out.println("Velikost souboru: " + frw.length());

    frw.seek(0L);           // návrat na začátek
    pocet = frw.readInt();
    posun = 4 * pocet;     // int je velký 4 bajty
    frw.seek(posun);
    frw.writeInt(1234);    // přepsání posledního int v souboru
    frw.seek(posun);
    k = frw.readInt();
```

```

System.out.print(k);

frw.skipBytes(8);      // double je velký 8 bajtů
double e = frw.readDouble();
System.out.println("\n" + e);
frw.close();
}

```

Vypíše např.:

```

984 506 380 951 404
3.141592653589793 2.718281828459045
Velikost souboru: 40
Velikost souboru: 40
1234
2.718281828459045

```

Poznámky:

- Při pokusu o zápis do souboru otevřeného jen pro čtení je vyvolána výjimka `IOException`.
- Při zadání modu "wr" se vyhodí `IllegalArgumentException`.
- V modu "rw" se existující soubor nevymaže, ale přepisuje. Pokud je nový (přepisovaný) soubor kratší než původní, zůstává zbytek informací z původního souboru nezměněn na konci. Pak je vhodné použít metodu `setLength()` pro oříznutí souboru.

Vhodný příkaz je např.:

```
frw.setLength(frw.getFilePointer()); // jen pro JDK 1.2
```

Příklad 176:

Další příklad ukazuje vytvoření „indexu řádek“, jako použití `readLine()`.

Soubor je vypsán nejprve popředu a pak pozpátku.

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

```

public class TestRAF {
    public static void main(String[] args) throws IOException {
        RandomAccessFile fr = new RandomAccessFile("f.txt", "r");
        String radka;
        int pocet = 0;
        long[] ofsety = new long[10];

```

```

        ofsety[0] = 0;
        while ((radka = fr.readLine()) != null) {
            pocet++;
            ofsety[pocet] = fr.getFilePointer();
            System.out.println(radka);
            if (radka.endsWith("\r") == true)
                System.out.println("\r");
        }

        fr.seek(0L);
        for (int i = pocet - 1; i > -1; i--) {
            fr.seek(ofsety[i]);
            radka = fr.readLine();
            System.out.println(radka);
        }
        fr.close();
    }
}

```

Pozor:

V JDK 1.1 nechávala `readLine()` na konci řetězce znak `\r`. V JDK 1.2 je již tato chyba odstraněna. □

Poznámky:

- Pokud budete pracovat s řádkami, pak dejte pozor na skutečnost, že ze souboru jsou čteny bajty, které jsou konvertovány na znaky tím, že se v paměti vynuluje vyšší bajt znaku. Tím samozřejmě nelze dosáhnout u všech akcentovaných znaků správného Unicode. Pro neakcentované znaky pracuje správně.

- Budete-li do souboru zapisovat pomocí `writeChars()` a číst pomocí `readLine()`, dostanete výsledek dvojnásobné délky. Správná kombinace metod je totiž `writeBytes()` a `readLine()`.

```

RandomAccessFile fr = new RandomAccessFile("g.txt", "rw");
fr.writeChars("Ahoj");      // A h o j
fr.writeBytes("Ahoj");      // Ahoj
fr.seek(0L);
String radka = fr.readLine();
System.out.println(radka);
fr.close();

```

Cvičení:

1. Textovým (ASCII) editorem vytvořte soubor PISMENA.TXT, ve kterém bude několik řádek složených z malých a velkých písmen a mezer. Napište program, který tento soubor přečte a opíše na obrazovku. Současně do souboru VELKY.TXT zapisuje obsah čteného souboru, ale malá písmena převádí na velká.
2. Napište program, který přečte soubor PISMENA.TXT po řádcích. Každou řádku přesně opíše do souboru KOLIK.TXT a v něm na další řádce uvede, kolik malých písmen na ní bylo.
3. Napište program, který zapíše do souboru POLE.TXT deset řádků celých čísel, takto:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

11, 12, 13, 14, 15, 16, 17, 18, 19, 20

91, 92, 93, 94, 95, 96, 97, 98, 99, 100

4. Napište program, který přečte soubor POLE.TXT a jednotlivé hodnoty uloží do dvourozměrného pole, které kontrolně vypíše na obrazovku. Celé pole pak uloží do souboru POLEINT.BIN jako binární čísla (použijte `writeInt()` ze třídy `DataOutputStream`). Porovnejte velikosti souborů POLE.TXT a POLEINT.BIN.

19 Systémové akce

V této kapitole budou popisovány činnosti, které lze souhrnně označit jako spolupráci programu s jeho okolím. Okolím je méně operační systém, případně počítač či síť, a je nutné si uvědomit, že komunikace programu s okolím probíhá vždy prostřednictvím JVM.

Poznámka:

Mnohé z těchto metod možná hned ke své práci nevyužijeme, ale je dobré vědět, že existují. □

19.1 Parametry příkazové řádky

Program v Javě (aplikace, nikoliv applet) může využít libovolné množství parametrů (argumentů), které byly zadány v příkazové řádce současně se jménem programu při jeho spuštění. To je jeden ze způsobů, jak lze z vnějšku ovlivnit chování programu.

Pozor:

Pokud použijete parametry příkazové řádky, počítejte s tím, že program nebude 100% přenositelný, protože např. Mac OS nemá příkazovou řádku, stejně jako nemá výstup na konzoli pomocí známého: `System.out.println()`. Ostatních běžných operačních systémů, jako Linux, MS Windows, Solaris apod., se to netýká. □

Jak již víme, preložený program se spouští příkazem:

`java NázevTřídy` kde `NázevTřídy` je jméno souboru `.class`, ve kterém je uvedena metoda `main()`.

Pokud uvedeme za tyto dva názvy ještě něco dalšího, je to považováno za parametr příkazové řádky, např.:

`java Ctisoubor a.txt`

Řetězec `a.txt` je předán metodě `main()` pomocí jejího formálního parametru `args`, což je pole řetězců.

Poznámka pro programátora v C či C++:

Toto pole řetězců funguje stejně jako v C pole argv. V jazyce C byl navíc ještě parametr argc, ve kterém byl uložen počet parametrů. Java toto nepotřebuje, protože každé pole si s sebou nese svoji délku. □

Protože je args pole řetězců, je k dispozici proměnná args.length, ve které je počet parametrů. Jednotlivé parametry jsou v poli args uloženy od indexu 0.

Poznámka pro programátora v C či C++:

To je opět významný rozdíl od C, kde v argv[0] byl uložen název programu a první parametr byl až v argv[1]. □

V Javě se jméno programu předávat nemusí, protože to musí být jméno třídy, ve které je uvedena metoda main() – nelze jinak. Potřebujeme-li v programu využít jméno programu, je nejjednodušší jej v editoru opsat ze jména třídy, protože se nebude měnit. Chceme-li ale „čistý objektový přístup“, je možné použít triku, kdy vytvoříme instanci této třídy a pak použijeme již známou metodu getClass() třídy Object – viz následující příklad.

Příklad 177:

Následující program vypíše své jméno a pak opíše všechny své parametry, každý na novou řádku:

```
public class Parametry {
    public static void main(String[] args) {
        Parametry ja = new Parametry();
        System.out.println("Program: " + ja.getClass().getName());
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Po spuštění programu příkazem:

```
java Parametry ahoj nazdar dobrý den
se vypíše:
```

```
Program: Parametry
ahoj
nazdar
dobrý
den
```

Poznámka:

Pokud si říkáte, že ve zdrojovém souboru se třída sice jmenuje `Parametry`, ale vám po překladu do souboru `Parametry.class` nic nemůže zabránit v tom, abyste tento soubor libovolně nepřejmenovali, např. na `Ahoj.class`, pak máte pravdu. Bohužel¹, takto přejmenovaný soubor vám k ničemu nebude, protože po jeho spuštění příkazem:

```
java Ahoj
```

se program nespustí, neboť hned na začátku je vyvolána výjimka:

```
java.lang.NoClassFoundError: Ahoj (wrong name Parametry) □
```

Když spustíme program příkazem:

```
java Parametry ahoj nazdar dobrý den
```

byli bychom pravděpodobně rádi, aby pozdrav „`dobrý den`“ byl vypsán jako jeden parametr. To nyní nebylo možné, protože mezera je bílý znak a ty slouží jako oddělovače parametrů. Tuto funkci mezery lze ale snadno potlačit tím, že slova oddělená mezerou dáme do uvozovek. Tím je sloučíme do jednoho parametru, např.:

```
java Parametry ahoj nazdar "dobrý den"
```

Vypíše:

Program: Parametry

ahoj

nazdar

`dobrý den`

Jak je vidět, znak uvozovky není součástí parametru.

Poznámka:

Z příkazové řádky často předáváme čísla. Je zřejmé, že tato čísla jsou předávána ve formě řetězců číslic a na skutečné číslo je nutné je převést postupem již známým z [9.6/153]. Zde jen pro připomenutí, např.:

```
int i = Integer.valueOf(args[0]).intValue(); □
```

19.2 Systémové atributy a zdroje

Program v Javě běží pod konkrétním operačním systémem, jehož prostředí je popsáno v **systémových atributech** (*system attributes*). Pod těmi si můžeme představit např. jméno operačního systému, domovský adresář uživatele atd.

¹Z bezpečnostních důvodů ale spíše „naštěstí“. ; -)

Java dává možnost tyto atributy čist a tím vlastně výrazně napomáhá k vytváření přenositelných programů.

Systémově nezávislé atributy jsou přístupné pomocí třídy `System`.

Poznámka:

Systémové závislé atributy jsou přístupné pomocí třídy `Runtime`, ale jejich popis již přesahuje „záběr“ této knihy. □

Dále budou uvedeny některé oblasti, ve kterých se se třídou `System` setkáváme. Hned na počátku je ale nutné zdůraznit, že program nikde ne-vytváří objekt (instanci) třídy `System`. Není to zapotřebí, protože všechny proměnné a všechny metody této třídy jsou statické – jsou to proměnné a metody třídy.

Poznámka:

Navíc je konstruktor třídy `System` privátní (`private`), takže pokus o vytvoření instance by selhal. □

19.2.1 Standardní vstupní a výstupní proudy

Setkáváme se s nimi od úplného začátku práce s Javou, nejčastěji v podobě `System.out.println()`.

Filosofie standardních vstupních a výstupních proudů byla přejata z jazyka C a je výhodné ji znát, i když program, který tyto proudy používá, není 100% přenositelný. Dovolují totiž vytvářet programy, na které jsme běžně zvyklí a které umožňují např. přesměrování výstupu do souboru atd.

Pokud ale vytváříme program, který je plně orientován na využití GUI, jsou téměř zbytečné. Slovo „téměř“ znamená, že i přesto se v takových programech občas použijí, a to nejčastěji pro ladící účely nebo pro výpis chybových zpráv přesměrovaný do „logovacího“ souboru.

Třída `System` dává k dispozici tři proudy, přesněji řečeno referenční proměnné odkazující na instanci třídy, která představuje otevřený proud:

- `System.in` – vstupní proud (`InputStream`)
- `System.out` – výstupní proud (`PrintStream`)
- `System.err` – chybový výstupní proud (`PrintStream`)

Všechny tři jsou implementovány pomocí v závorce zmíněných tříd z balíku `java.io` – viz [18/269], takže jejich použití nebude znova opakovat.

Někdy se nám ale může hodit změna těchto proudů, nejčastěji na nějaký námi otevřený souborový proud. K tomu používáme metody `setIn()`, `setOut()` a `setErr()`.

Příklad 178:

Následující příklad ukáže, jak lze všechny „normální“ výpisy přesměrovat do souboru `vypisy.txt` a chybové výpisy (např. hlášení výjimek) do souboru `chyby.log`.

```
import java.io.*;  
  
public class Sy2 {  
    public static void main(String[] args) throws IOException {  
        FileOutputStream fout = new FileOutputStream("vypisy.txt");  
        PrintStream pout = new PrintStream(fout);  
        FileOutputStream ferr = new FileOutputStream("chyby.log");  
        PrintStream perr = new PrintStream(ferr);  
  
        System.out.println("Out pred presmerovanim");  
        System.err.println("Err pred presmerovanim");  
        System.setOut(pout);  
        System.setErr(perr);  
        // System.out = pout; // nelze  
        System.out.println("Out PO presmerovani");  
        System.err.println("Err PO presmerovani");  
  
        fout.flush();  
        // vyhození výjimky, která se zapise do souboru chyby.log  
        throw new IOException();  
    }  
}
```

Program vysal na obrazovku:

Out pred presmerovanim

Err pred presmerovanim

Obsah souboru `vypisy.txt` byl:

Out PO presmerovani

Obsah souboru `chyby.log` byl:

Err PO presmerovani

`java.io.IOException`

at `Sy2.main(Sy2.java:21)`

Poznámka:

Protože jsou `System.in`, `System.out` a `System.err` deklarovány jako `final`, nelze je přímo změnit příkazem typu:

```
System.out = pout; // nelze
```

□

Při používání těchto triků s přesměrováním je nutné si uvědomit, že od standardních vstupů a výstupů se jaksi implicitně předpokládá „standardní“ chování, které přesměrováním silně pozměníme. Je tedy vhodné nejdříve se zamyslit, zda je tento postup vhodný.

Vhodný je například v případě již zmiňovaném v [16.2.5/251], kdy pro ošetření výjimky používáme pouze výpis metodou `printStackTrace()`. Při použití GUI pak nevidíme chybové zprávy směrované touto metodou na konzoli. Řešením je pak přesměrovat chybový výstup do souboru a ten si po skončení programu prohlédnout.

19.2.2 Systémové vlastnosti

Pomocí třídy `System` jsou ovládány také nejrůznější **vlastnosti** (*properties*), definující nejrůznější rysy nebo atributy okolního prostředí.

Tyto vlastnosti jsou uloženy vždy ve dvojici **klic/hodnota** (*key/value*), kde **klic** představuje jméno atributu, např. `"line.separator"`²

Výpis všech vlastností lze provést příkazem:

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

Tento příkaz vypíše asi 44 vlastnosti, např.:

```
-- listing properties --
java.specification.name=Java Platform API Specification
awt.toolkit=sun.awt.windows.WToolkit
java.version=1.2
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
user.timezone=Europe/Prague
java.specification.version=1.2
java.vm.vendor=Sun Microsystems Inc.
***
```

U čtrnácti z této skupiny vlastností je zajištěno, že musí být při startu JVM nastaveny. Jsou to:³

²Skutečně je mezi slovy `line` a `separator` oddělovač tečka.

³Hodnoty jsou samozřejmě závislé na konkrétním prostředí.

<i>klíč</i>	<i>hodnota</i>	<i>význam</i>
file.separator	\	oddělovač adresáru v cestě
line.separator	\r\n	ukončení řádky
path.separator	:	oddělovač jednotlivých cest
java.class.path	.	adresář(e), kde jsou uloženy soubory .class
java.class.version	46.0	verze .class
java.home	c:\program\1\jdk1.2\jre	adresář, ze kterého je spouštěn interpreter Javy
java.vendor	Sun Microsystems Inc.	zdroj (prodejce) této instalace Javy
java.vendor.url	http://java.sun.com/	URL prodejce
java.version	1.2	verze Javy
os.arch	x86	typ procesoru v architektuře operačního systému
os.name	Windows NT	typ operačního systému
user.dir	D:\java\ujj	aktuální adresář
user.home	C:\WINNT\Profiles\herout	domovský adresář
user.name	herout	<i>login name</i> uživatele

Poznámka:

I těchto 14 údajů poskytuje poměrně značné množství „důvěrných“ informací. Z tohoto důvodu není např. apletům dovoleno ani čtení některých z nich. □

Zmiňované vlastnosti lze číst pomocí metody `getProperty()` třídy `System`.

Poznámka:

Hodnoty vlastností lze i měnit, což ale většinou není dobrý nápad, proto zde není uvedeno, jak na to. Pro změnu vlastností musí být velmi dobrý důvod. Ten, kdo jej bude mít, si určitě dokáže v dokumentaci najít způsob, jak to provést. □

Je zřejmé, že mnohé z uvedených vlastností, zejména `file.separator`, `line.separator` a `path.separator` je vhodné používat, protože nám pomáhají vyhnout se nejčastějším rozdílům mezi operačními systémy.

Příklad 179:

Následující příklad ukáže, jak lze vytvořit pod adresář `TMP` a v něm textový soubor `a.txt` tak, aby bylo jedno, zda program spustíme pod Windows nebo pod Unixem.

```
import java.io.*;
public class Sy4 {
    public static void main(String[] args) throws IOException {
        String oddRadek = System.getProperty("line.separator");
        String jmSouboru = System.getProperty("file.separator");
        File adr = new File("TMP");
        adr.mkdir();
        String jmSoub;
        jmSoub = adr.getName().concat(jmSouboru).concat("a.txt");
        FileOutputStream fw = new FileOutputStream(jmenoSouboru);
        PrintStream fwPr = new PrintStream(fw);
        fwPr.print("Jedna radka");
        fwPr.print(oddRadek);
        fwPr.println("Druha radka");
        fwPr.println("Treti radka");
        fw.close();
    }
}
```

Obsah souboru `.\TMP\a.txt` je:

Jedna radka

Druha radka

Treti radka

Tento příklad není příliš průkazný, co se týče použití oddělovače řádek, protože třída `PrintStream` dovoluje tento oddělovač použít automaticky.

Příkazy:

```
fwPr.print("Jedna radka"); fwPr.print(oddRadek);
```

a

```
fwPr.println("Jedna radka");
```

jsou funkčně naprosto shodné, protože třída `PrintStream` využívá hodnotu vlastnosti `line.separator`.

Pokud bychom ale četli soubor, o kterém bychom si nebyli jisti, že je textový (nešlo by použít `DataInputStream.readLine()`), pak by bylo použití hodnoty `line.separator` nezbytné.

19.3 Užitečné metody ze třídy System

19.3.1 Informace o čase

Třída `System` poskytuje metodu `currentTimeMillis()`, která vrací počet milisekund od 1. 1. 1970. Upřímně řečeno, toto číslo nás většinou ani tak moc nezajímá. To, co nás většinou zajímá, je rozdíl dvou těchto časů, což je doba trvání běhu programu. Takovýto údaj je velmi vhodný pro nejrůznější testy výkonnosti programu.

Poznámka:

Je třeba si uvědomit, že moderní operační systémy jsou víceúlohové a víceuživatelské. Náš program nikdy neběží sám a z tohoto důvodu se mohou doby běhu jednoho a téhož programu i dosti podstatně lišit. Provádime-li testy výkonnosti, je vhodné uzavřít všechny nepotřebné spuštěné aplikace. Ty, co uzavřít nelze, se snažíme neaktivovat např. zbytečným nervózním „cvakáním“ myší po ploše. Poslední dobrá rada je spustit měřený program několikrát, vyloučit extrémy a ze zbylých hodnot vypočítat průměr. □

Příklad 180:

Následující program měří dobu výpočtu faktoriálu pomocí rekurze a pomocí cyklu. Všimněte si vnitřního cyklu, pomocí kterého se posouvá celé měření do rozlišitelných hodnot.

```
public class CasFaktorialu {  
    public static void main(String[] args) {  
        long z, k, f;  
  
        for (long i = 10; i <= 20; i += 2) {  
            System.out.print(i + "\t");  
            z = System.currentTimeMillis();  
            for (int j = 1; j <= 500000; j++)  
                f = Fakt.faktRek(i);  
            k = System.currentTimeMillis();  
            System.out.print((k - z) + "\t");  
        }  
    }  
}
```

```
z = System.currentTimeMillis();

for (int j = 1; j <= 500000; j++)
    f = Fakt.faktCykl(i);

k = System.currentTimeMillis();
System.out.print((k - z) + "\n");
}

}
```

Program vypíše např. (blok vpravo je z druhého běhu):

6!	500	361	6!	481	360
8!	681	460	8!	671	471
10!	852	560	10!	921	541
12!	1022	671	12!	1002	650
14!	1192	781	14!	1172	771
16!	1412	961	16!	1382	932
18!	1622	1092	18!	1602	1071
20!	1883	1242	20!	1853	1212

Poznámka:

Všimněte si, že se časy z obou běhu trochu liší. Pokud by tyto časy byly důležité, bylo by vhodné nechat proběhnout např. 100 běhu programu, výsledky ukládat do souboru a výsledné časy zprůměrovat. □

19.3.2 Spuštění garbage collectoru

Automatické čištění paměti provádí JVM na pozadí běhu programu a my se o tuto činnost většinou nemusíme starat. Víme-li však např., že následující část programu bude mít velké paměťové nároky, můžeme před vstupem do této části vynutit běh *garbage collectoru* voláním `System.gc()`.

V tomto případě je ale nutné si uvědomit, že tato aktivita je výpočetně náročná a bude trvat nějakou dobu. To tedy znamená, že není vhodné spouštět *garbage collector* jako součást výpočetně náročných operací a už vůbec ne v cyklu.

19.3.3 Zjištění velikosti dostupné paměti

S činností *garbage collectoru* jsou svázány dvě metody ze třídy `java.lang.Runtime`. První je `totalMemory()` a vrací celkovou velikost paměti dostupné JVM v bajtech. Druhá je `freeMemory()` a vrací velikost právě dostupné paměti.

Pozor:

Jedná se o metody instance, takže je nelze spustit intuitivním:

`Runtime.totalMemory()`

ale je nutné inicializovat referenční proměnnou pomocí volání:

`Runtime r = Runtime.getRuntime();`

□

Příklad 181:

Informace o celkové a o volné dynamické paměti.

```
public static void main(String[] args) {
    Runtime r = Runtime.getRuntime();
    System.out.println("Cela pamet: " + r.totalMemory());
    System.out.println("Volna pamet: " + r.freeMemory());
    int[] pole = new int[1000000];
    System.out.println("Volna pamet: " + r.freeMemory());
    pole = null;
    System.gc();
    System.out.println("Volna pamet: " + r.freeMemory());
}
```

Vypíše:

```
Cela pamet: 1048568
Volna pamet: 736520
Volna pamet: 1039240
Volna pamet: 1039368
```

Na výpisu je velmi zajímavá posloupnost čísel. Nejdříve bylo k dispozici asi 1 MB paměti, z toho 730 KB volné. Po alokaci pole jednoho milionu `int` (tj. 4 000 000 B) se velikost volné paměti **zvýšila** na 1 MB a po uvolnění tohoto pole zůstalo opět 1 MB.

Tento úkaz si lze vysvětlit tak, že JVM pracuje s pamětí „inteligentně“⁴, tzn. v počátcích si alokuje jen nezbytné minimum ; -) 1 MB a až v případě

⁴Záleží ovšem na konkrétní implementaci JVM. Tento výsledek byl získán na Win NT, pod Unixem na JDK 1.1 byl jiný – „neinteligentní“.

potřeby požádá operační systém o další paměť a po její dealokaci pomocí `System.gc()` ji zase dynamicky operačnímu systému vrátí. Stále si ale udržuje zásobu asi 1 MB.

Poznámka:

Z tohoto pokusu vyplývá, že není vhodné psát program, který bude průběžně vypisovat velikost volné paměti, protože výpisem hodnoty kolem 1 MB byste mohli uživatele se 256 MB⁵ lehce frustrovat. ; -) □

19.3.4 Spuštění finalizeru

V [8.14/141] bylo popsáno použití metody `finalize()` jako „destruktoru“ třídy.⁶ Metoda `finalize()` je ovšem JVM volána tehdy, když se to JVM hodí, což může být někdy na závadu. Proto je možné voláním:

```
System.runFinalization()
```

vynutit vyvolání metod `finalize()` všech instancí, které zanikly a dosud jejich metody `finalize()` neproběhly.

19.3.5 Násilné ukončení programu

Musí-li program náhle ukončit svoji činnost, je to možné provést kdekoliv voláním `System.exit(-1)`, přičemž hodnota `-1` je návratovou hodnotou celého programu.

Poznámka:

Toto volání je poměrně „drsné“ ukončení programu. Nelze se spoléhat, že budou uzavřeny otevřené průduvy, soubory atd. Z tohoto důvodu je dobré metodu `exit()` použít jen v případě naprosté nezbytnosti, kdy jiné prostředky už nejsou možné. □

Příklad 182:

Ukázka metody `exit()`.

```
public class Exit {
    public static void konec() {
        System.out.println("Pred exit()");
        System.exit(-1);
        System.out.println("PO exit()");
    }
}
```

⁵To je v únoru 2000 považováno za velmi slušnou velikost paměti. ; -)

⁶A tam se také nachází příklad použití.

```
public static void main(String[] args) {  
    System.out.println("Pred konec()");  
    konec();  
    System.out.println("PO konec()");  
}  
}
```

Vypíše:

Pred konec()
Pred exit()

Cvičení:

1. Napište program, který vypíše soubor, jehož název byl zadán jako parametr příkazové řádky.
2. Modifikujte program ze cvičení v kapitole 18, který vytvářel soubor POLE.TXT, aby zapisoval deset tisíc řádek. Program spusťte dvakrát – poprvé bez využití buferování, podruhé s jeho využitím. Pomocí `System.currentTimeMillis()` změřte časy běhu obou programů.
3. Napište program, který vypíše, jakou verzí překladače Java je překládán (využijte systémovou vlastnost `java.version`).

20 Vlákna

Jak již bylo dříve vícekrát zmiňováno, současné operační systémy jsou víceúlohové a víceuživatelské. To znamená, že v jeden okamžik (z lidského měřítka vnímání času) může v počítači běžet více programů najednou.

Je možné položit si otázku: „Když může v rámci operačního systému běžet více programů najednou, je něco podobného možné i v rámci programu?“

Jinak položená otázka zní: „Je možné, aby program neběžel od začátku do konce jako jeden monolit, ale rozdělil se na několik částí, které budou na sobě víceméně nezávislé a poběží víceméně v náhodném pořadí?“

Odpověď na obě otázky je ANO. Snaha o to, aby se dal program dělit na více částí, je velmi stará. Tyto jednotlivé části se nazývaly v jiných programovacích jazycích *korutinami* (*couroutines*), *kooperujícími procesy*, *light-weight* procesy atp.¹ V Javě se pro „samostatné“ části programu ujal další z běžně používaných názvů, a to *vlákna* (*threads*).

Používání vláken se dnes v programování stále rozšiřuje a Java, jako jeden z mála programovacích jazyků, vlákna přímo podporuje.

„K čemu je takové vlákno?“, zní pravděpodobně další otázka. Odpověď na ni není jednoznačná. Nejjednodušší odpověď zní: „K ničemu,“ a je zcela pravidlivá v tom případě, kdy potřebujeme na jednoprocесорovém počítači vytvořit program, který zpracuje nějaká vstupní data, vytvoří výstupní data a to vše pokud možno rychle a bez interaktivního zásahu uživatele.

Je asi jasné, že takových programů je mnoho, ale velmi rychle (zejména s náruštěm „okenních“ aplikací) přibývají i programy jiné.

Následující stručný a neúplný výčet bude popisovat několik oblastí, v nichž je použití vláken výhodné:

- **Časově náročné akce –** trvá-li nějaký výpočet relativně dlouhou dobu,² je vhodné dát uživateli možnost provádět nějakou jinou akci,

¹Tato terminologie je z pohledu teorie operačních systémů zjednodušující!

²Výzkumy ukazují, že po jedné vteřině zdánlivě nečinnosti počítače začíná být uživatel nervózní.

případně jej ujistit, že počítač opravdu pracuje a za jak dlouho práci zhruba dokončí.

- **Čekání na vstupy od uživatele** – uživatel je proti počítači o několik rádů pomalejší. Pokud je to možné, lze čas strávený čekáním na jednotlivé stisky kláves využít k další činnosti. Například k průběžné kontrole pravopisu, známé z textových procesorů.
- **Opakující se výpočty** – toto je častá záležitost v simulacích, kdy podle jednoho kódu programu běží navzájem nezávisle několik výpočtů; příkladem může být např. simulace pohybu člověka v obchodním domě, kdy lze podle jednoho vzoru vytvořit tisíce zákazníků.
- **Úlohy typu producent–konzument** – producent připravuje data, která konzument průběžně zpracovává.

Vlákna v Javě mohou být na sobě téměř zcela nezávislá, např. jedno počítá faktoriál a druhé mezikódum načítá soubor. Takový program ovšem nemá příliš velký smysl. Vlákna spolu většinou nějakým způsobem spolupracují (proto jsou v jednom programu ; -)) a čím více spolupracují, tím je větší nutnost zajistit jejich správnou interakci.

O vláknech se často říká, že pracují paralelně. To může být pravda pouze na počítačích s více procesory. Má-li počítač jen jeden procesor, je paralelní běh vláken zajištěn jejich střídáním („každý chvíliku tahá pilku“). Protože toto střídání – terminologicky správně je to **předávání řízení** – probíhá velmi rychle, z pohledu uživatele se to jeví, jako by opravdu vlákna běžela paralelně. Správný termín pro tuto činnost je **pseudoparalelně**.

20.1 Třída Thread

Každé vlákno je instancí třídy `java.lang.Thread`, nebo jejího potomka. V nejjednodušším případě stačí pouze odvodit potomka od třídy `Thread` a překrýt klíčovou metodu `run()`, která popisuje, co vlákno při svém běhu vlastně dělá. Metoda `run()` vlákna se nespouští přímo, ale pomocí volání metody `start()` ze třídy `Thread`.

Příklad 183:

Následující příklad je školní příklad, jak podle jednoho kódu programu vytvořit více vláken.

```
public class Vlakno1 extends Thread {
```

```

public Vlaknol(String jmeno) {
    super(jmeno);
}
public void run() {
    for (int i = 1; i <= 3; i++) {
        System.out.println(i + ". " + getName());
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException e) {
            System.out.println("Probudili jste mne");
        }
    }
}

public static void main(String[] args) {
    Vlaknol vlAhoj = new Vlaknol("ahoj");
    vlAhoj.start();
    new Vlaknol("nazdar").start();
    new Vlaknol("cao").start();
}
}

```

V konstruktoru pouze voláme konstruktor třídy `Thread`, kterému předáváme jméno budoucího vlákna. Metoda `run()` využívá metodu `getName()`, která toto jméno vlákna vrátí. Vlákno proběhne třikrát, přičemž pokaždé pouze vypíše pořadové číslo běhu a své jméno (toto je jediná užitečná činnost vlákna). Blok `try` zajišťuje možnost předání řízení (tj. přidělení procesoru) tím, že se vlákno na 1000 milisekund uspí, čímž je dána možnost dalšímu vláknu, aby začalo pracovat. Probuzení vlákna se děje po uplynutí požadovaného časového úseku – zde 1000 ms.

Výjimka `InterruptedException`, zachycovaná v bloku `catch`, bude vyvolána jen tehdy, když vlákno „nedospí“, tj. je jiným vláknem předčasně vzbuzeno – viz dále [20.6.4/329].

Metoda `main()` pouze vytvoří tři nové instance vláken. První způsob (vlákno „ahoj“) je typický. Druhý způsob (zbývající dvě vlákna) není příliš typický a využívá toho, že `main()` pro anonymní instanci vlákna zavolá ihned jeho metodu `start()`. Tím je vlákno spuštěno, tj. existuje na něj odkaz a *garbage collector* nemůže tento objekt zrušit do té doby, než vlákno doběhne.

Program končí tehdy, když doběhnou všechna vlákna, tj. když skončí jejich metody `run()`.

Výstup programu je dle očekávání, přičemž mezi jednotlivými trojicemi výpisu je vteřinová prodleva:

1. ahoj
1. nazdar
1. cao
2. nazdar
2. cao
2. ahoj
3. nazdar
3. cao
3. ahoj

Poznámka:

Pokud jste se s vlákny nikdy nesetkali, pravděpodobně si ted' řeknete:
„Hezké, ale k čemu to prakticky je?“

Zde stačí, když zatím „jen“ oceníte jednoduchost, s jakou Java řeší předávání řízení mezi vlákny. Možná, že si při dalším čtení uvědomíte, jak by vlákna mohla být užitečná i vám. □

20.2 Spolupráce dvou vláken

V předchozím případě byla vlákna na sobě téměř úplně nezávislá.³ V následujícím textu si ukážeme pravděpodobně jeden z typických příkladů použití vláken. Bude se jednat o výpočetně náročný úsek programu, kdy je vhodné, aby byl uživatel informován o tom, že program skutečně pracuje. Tento výpočetně náročný úsek bude představovat čtení souboru se 100 000 řádkami,⁴ ve kterém je na každé řádce jedno celé číslo. Úkolem je spočítat sumu těchto čísel.

Čtení souboru a výpočet sumy bude probíhat v jednom vláknu, informaci o stavu výpočtu bude průběžně zobrazovat druhé vlákno.

³V každém případě spolu na úrovni našeho programu nijak nespolupracovala.

⁴Soubor `data.txt` byl připraven programem `Zapis.java` – viz na disketě. Pokud vám doba čtení souboru bude připadat zanedbatelná, máte zřejmě rychlejší počítač než já ; -) a pak si vytvořte soubor s jedním milionem řádek, případně, pokud budete tuto knihu číst za několik let od doby vydání, deset milionů řádek.

Poznámky:

- Z důvodu maximální jednoduchosti byl zvolen způsob předání informace mezi vlákny pomocí statických proměnných suma a hotovo, nastavovaných prvním vláknem a pouze čtených druhým vláknem. Toto není v praxi nejšťastnější způsob, respektive je to většinou chyba, protože pak nelze smysluplně vytvořit více instancí tohoto vlákna. V [20.6.7/333] uvidíte lepší řešení.
- Vypisovaná informace s průběžnou hodnotou sumy uživateli pravděpodobně nic moc neřekne, navíc se na obrazovce neustále přepisuje. To ovšem v tomto případě nevadí – účelem druhého vlákna je pouze udržovat uživatele v jistotě, že program pracuje, což mění se výpis zajisté splňuje.
- Možná, že jste se již s problémem čtení dlouhého souboru setkali a úspěšně jste jej řešili tím, že jste si na začátku čtení zjistili celkovou velikost souboru a pak po každém čtení jste vypočítali, kolik procent již bylo přečteno. Nyní se právem můžete divit, proč se to řeší tak složitě.

Uvědomte si prosím, že tento způsob důsledně odděluje obě činnosti – výkonnou a informační. Pokud se změní charakter výpisu (což by bylo při použití grafického prostředí pravděpodobné), nebude nutné **nijak** měnit charakter výkonné činnosti. Kromě toho, mimo vlákna výpisu by mohlo být spuštěno jiné vlákno, které by např. uživateli dovolilo, aby zadal jméno dalšího souboru, který potřebuje přečíst.

Tento program budeme postupně rozvíjet. Z tohoto důvodu si připravíme třídu `ReadV1`, která poskytne vlákno, jež bude číst soubor. Tato třída se v dalších pokusech nebude měnit.

Příklad 184:

Třída `ReadV1`, která bude používána v několika dalších programech.

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

```
public class ReadV1 extends Thread {
    FileReader fr;
    BufferedReader in;
    String jmenoSouboru;
    static public long suma = 0;
    static public boolean hotovo = false;
```

```
ReadVl(String jmeno) {
    super("Vlakno pro cteni");
    jmenoSouboru = new String(jmeno);
}

public void run() {
    String radka;
    int k;

    try {
        fr = new FileReader(jmenoSouboru);
        in = new BufferedReader(fr);
        while((radka = in.readLine()) != null) {
            k = Integer.valueOf(radka).intValue();
            suma += k;
            Thread.yield();
        }
        fr.close();
        hotovo = true;
    }
    catch (IOException e) {
        System.out.println("Chyba v souboru!");
    }
}
}
```

V konstruktoru pojmenujeme vlákno jako "Vlakno pro ctení" a uložíme si jméno souboru, který má být čten. Metoda `run()` otevře soubor a metodou `readLine()` známou z [18.4.2/278] čte řádku s celým číslem, které převede a přičte ke statické proměnné třídy `suma`.

Po přečtení celého souboru je nastavena statická proměnná třídy `hotovo` na `true`. Pomocí této proměnné se zastaví druhé výpisové vlákno. Nic z těchto příkazů by nás již nemělo překvapit.

Jediný nový příkaz je volání metody⁵ `public static void yield()`. Tímto příkazem říká vlákno, že se dobrovolně vzdává práva běžet a předává řízení dalšímu vláknu (podrobnosti viz v [20.3/318]). Až další vlákno opět předá řízení, rozběhne se první vlákno od místa, kde odevzdalo řízení příkazem `yield()`.

⁵Slovo `yield` znamená přenechat někomu něco.

20.2.1 Vlákna se pravidelně střídají

Třída Vlakno2 s druhým vláknem je poměrně jednoduchá. Opět vznikla děděním od třídy Thread a opět bylo nutné překrýt pouze metodu run(). Konstruktor se jménem vlákna pro jednoduchost vynecháváme. Pro předání řízení se použije metoda yield() naprosto stejně, jako u čtecího vlákna. To znamená, že se vlákna naprosto pravidelně střídají. Metoda main() má za úkol pouze obě vlákna vytvořit a pak spustit.

Příklad 185:

Předávání řízení pomocí yield() oběma vlákny.

```
public class Vlakno2 extends Thread {
    public void run() {
        while (ReadVl.hotovo == false) {
            System.out.print(ReadVl.suma + "\r");
            Thread.yield();
        }
        System.out.println(ReadVl.suma);
    }

    public static void main(String[] args) {
        ReadVl vlceni = new ReadVl("data.txt");
        vlceni.start();
        Vlakno2 vlVypis = new Vlakno2();
        vlVypis.start();
    }
}
```

Po spuštění program vypisuje průběžně čísla až ke konečnému výsledku 5000050000.

Tento způsob předávání řízení se dá nazvat „gentlemanským“ a pokud jste si program skutečně spustili, vidíte jeho nevýhodu – program běží asi 30 vteřin.⁶ Průběžná informace se totiž vypisuje 100 000 krát, což zabere většinu času. Toto časové zdržení je však v tomto případě zcela zbytečné, protože v předpokladech (viz [20.2/314]) bylo uvedeno, že výpis je pouze proto, aby uživatele „zabavil“.

⁶Je opět třeba zdůraznit, že čas závisí na rychlosti vašeho počítače.

20.2.2 Vlákna se střídají nepravidelně

Není-li z povahy algoritmu nutné, aby se vlákna střídala ve své práci pravidelně, je možné zkombinovat způsoby předání řízení pomocí metod `sleep()` a `yield()`. Vlákno, které čte, bude stále využívat metodu `yield()`, čili po každém čtení odevzdá řízení.

Vlákno, které vypisuje, však toto řízení pokaždě nepřevezme, protože využije metody `sleep(100)` – tedy uspí se na 100 milisekund a teprve pak přijme řízení předávané čtecím vlákнем.

Poznámky:

- Hodnota 100 milisekund je „tak akorát“, aby se uživatel nenudíl.
- Není výhodné, aby metodu `sleep()` využívalo i čtecí vlákno, protože tím by se program jen zcela zbytečně zdržoval. Čtecí vlákno musí pracovat co nejrychleji a pokud nemá jiné vlákno zájem běžet, využije veškerý dostupný čas poskytnutý procesorem.

Příklad 186:

Předání řízení pomocí `yield()` a `sleep()`.

```
public void run() {
    while (ReadVl.hotovo == false) {
        System.out.print(ReadVl.suma + "\r");
        try {
            Thread.sleep(100); // 100 milisekund
        }
        catch (InterruptedException e) {
        }
    }
    System.out.println(ReadVl.suma);
} // main () je nezmenena
```

Program nyní pracuje výrazně rychleji – necelé 4 vteřiny – a výsledný efekt je zcela stejný, jako v předchozím případě – uživatel je dostatečně často informován o tom, že „se něco děje“.

Poznámka:

V chráněném bloku by měl být podle doporučení ze [250/16.2.4] nějaký kód, nejspíše `System.out.println("Predcasne vzbuzen!");`

Není tam proto, že se právě v tomto místě zachycená výjimka **většinou** skutečně nijak neřeší. □

20.3 Stavy vlákna a plánovací algoritmus

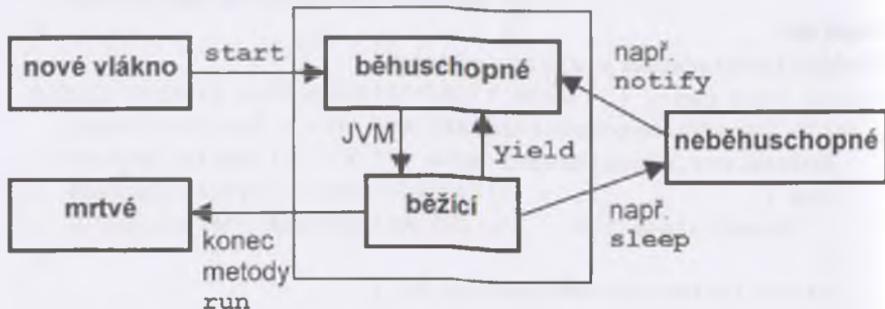
Předchozí dva případy byly založeny na tom, že čtecí vlákno „gentlemanicky“ nabízelo předání řízení po každém čtení. Tento způsob je doporučený – obecně řečeno, pokud víme, že nějaká akce potrvá dlouho, je vhodné umožnit její přerušení.

Co se však stane, pokud vlákno řízení dobrovolně nepředá? Odpověď na tuto otázku záleží na několika okolnostech:

- stavu okolních vláken,
- prioritě vlákna,
- schopnostech operačního systému.

20.3.1 Stavy vlákna

Každé vlákno se musí nacházet v jednom ze čtyř možných stavů, přičemž přechody mezi těmito stavům zajišťují nejčastěji metody třídy Thread.



Stavy vláken jsou:

- **Nové vlákno** – vlákno bylo vytvořeno (nejčastěji pomocí `new`), ale dosud nebylo spuštěno metodou `start()`.
- **Běhuschopné (runnable)** vlákno – metoda `start()` již proběhla; těchto vláken může být více, ale pouze jedno je **běžící**, ostatní čekají na předání řízení.
- **Neběhuschopné** – vlákno, které:
 - bylo uspáno metodou `sleep()`,
 - čeká na `wait()` – viz [20.6.7/333]
 - čeká na I/O – viz [20.5/323]
- **Mrtvé vlákno** – vlákno, jehož metoda `run()` skončila.

20.3.2 Priorita vlákna

Každé vlákno má nastavenou svoji prioritu, podle které se řídí plánovač běhu vláken. Znamená to, že jsou-li běhuschopná dvě vlákna, bude vždy předáno řízení vláknu s vyšší prioritou.

Priorita vlákna se dá zjistit metodou `getPriority()` a nastavit metodou `setPriority()`, přičemž priorita je celé nezáporné číslo.

Nejnižší priorita má hodnotu `MIN_PRIORITY`, nejvyšší `MAX_PRIORITY`, „zlatá střední cesta“ je `NORM_PRIORITY`, což je priorita standardně přidělovaná nově vznikajícím vláknům.

Poznámka:

V JDK 1.2 mají `MIN_PRIORITY`, `NORM_PRIORITY` a `MAX_PRIORITY` hodnoty 1, 5 a 10. □

Od priority se odvíjí tato pravidla plánovače:

- Ze všech běhuschopných vláken běží vždy to, které má nejvyšší prioritu (případně jedno z vláken s nejvyšší prioritou) ze všech běhuschopných vláken.
- Pokud se do běhuschopného stavu dostane vlákno s vyšší prioritou než má vlákno běžící, je běžící vlákno okamžitě donuceno k předání řízení ve prospěch vlákna s vyšší prioritou – tento mechanizmus se nazývá ***preemptivní plánování***.
- Pokud na přidělení procesoru čeká více vláken se stejnou prioritou, plánovač je volí postupně tak, aby na každé vlákno pokud možno došlo.
- Běžící vlákno pomocí metody `yield()` dobrovolně předá řízení některému běhuschopnému vláknu se stejnou prioritou.
- Vlákno s nižší prioritou může získat řízení jen tehdy, když vlákná s vyšší prioritou přejdou do neběhuschopného stavu.

Poznámka:

Má-li vlákno nejvyšší prioritu a dobrovolně se nevzdá řízení, není možné jej k předání řízení obecně⁷ donutit. □

⁷Jednu možnost uvidíte v [20.5/323] – čekání na I/O.

Pozor:

Uvedená pravidla pro plánovač jsou obecně dodržovaná. Není ale dobré spolehnout se při programování algoritmu bezmezně na to, že budou vždy bezpodmínečně dodržována. Lze si snadno představit „inteligentní“ plánovač⁸, který za určitých okolností dá přednost vláknu s nižší prioritou, protože chce zabránit nějakému problému, který detekoval. Priority by se měly proto používat jen tehdy, když ovlivňujeme předávání řízení z hlediska co nejvyšší účinnosti. [Cam1] □

20.3.3 Sdílení času

Pokud operační systém⁹ podporuje tzv. **sdílení času** (*time-slicing* nebo též **casová kvanta**), pak se vlákna se stejnou prioritou pravidelně střídají po pevně přidělených časových intervalech.

To tedy znamená, že i vlákno s nejvyšší prioritou bude pozastaveno, aby mohla běžet i jiná vlákna s touto prioritou.

Poznámky:

- Na vlastnost sdílení času se nedá obecně spolehnout (je to vlastnost jen některých operačních systémů) a je chybou vytvářet programy, jejichž algoritmus na tomto závisí.
- Pokud je v programu jen jedno vlákno s nejvyšší prioritou (a odmítá odevzdat dobrovolně řízení), pak je podpora operačního systému sdílením času neučinná. Nelze totiž spustit vlákno s nižší prioritou, čeká-li na spuštění právě „násilím“ přerušené vlákno, které však má jako jediné nejvyšší prioritu.

20.3.4 Praktické ověření plánování a priorit

Všechna fakta ohledně plánovacího mechanizmu vyjmenovaná v předchozích částech lze snadno ověřit jednoduchou modifikací předchozího programu načítajícího čísla ze souboru.¹⁰

⁸Naprogramovaný nějakou ambiciózní softwarovou firmou.

⁹Systémy Windows 98/NT a Unix toto splňují.

¹⁰Zkoušeno na Windows NT, tedy na OS s podporou sdílení času.

<code>yield()</code> ve čtecím vláknu	priorita čte- cího vlákna	priorita výpi- sového vlákna	střídání vláken
ano	NORM	NORM	ano
ano	MIN	NORM	ano
ano	MAX	NORM	ne
ne	NORM	NORM	ano
ne	MIN	NORM	ano
ne	MAX	NORM	ne

Z tabulky vyplývá, že „gentlemantské“ chování (tj. volání `yield()`) je ostatním vláknům málo platné, pokud nemají dostatečnou prioritu. A naopak, má-li vlákno, u něhož se dá očekávat krátká doba běhu (typicky informační vlákno), vysokou prioritu, bude na běžných operačních systémech pracovat správně, i když se ostatní vlákna budou chovat „negalementanský“.

20.4 Rozhraní Runnable

V předchozích příkladech byla použita třída s vláknem vždy zděděným od třídy `Thread`. Pomocí rozhraní lze však zařídit, aby výhody vlákna mohla využívat i třída, která vznikla děděním od jiné třídy – typicky je to v případě appletu, který musí být potomkem třídy `Applet`.

Vlastnosti vlákna může třída získat implementací rozhraní `Runnable`, které obsahuje pouze jednu metodu – `run()`. Tuto metodu musíme samozřejmě v naší třídě implementovat, ale není to jediná nutná aktivita.

Při vytváření a spuštění vlákna nemůže ten, kdo instanci vlákna vytváří, principiálně vědět, zda třída s vláknem vznikla děděním od `Thread` nebo implementací `Runnable`. Při spuštění vlákna se totiž používá jednotně metoda `start()`, která spustí metodu `run()`. Bohužel při implementaci `Runnable` nejsme povinni metodu `start()` vytvořit, což znamená, že se naše vlákno může z vnějšího pohledu chovat nestandardně. To však asi není naším cílem, takže obvyklý postup je, že při implementaci `Runnable` vytváříme jak metodu `run()`, tak i metodu `start()`.

Uvedený důvod ale není jediný. Ve skutečnosti je potřeba vždy vytvořit instanci třídy `Thread` kvůli datovým prvkům, které jsou sice skryté, ale vnitřně je vlákna využívají. V metodě `start()` proto nemůžeme přímo zavolat metodu `run()`.¹¹ Instance vlákna dosud neexistuje, a proto je

¹¹ Respektive můžeme, ale nebude mít vlastnosti metody `run()` vlákna.

nutné ji vytvořit – nejlépe tak, že vytvoříme přímo instanci třídy Thread a v konstruktoru jí předáme pomocí `this` referenci na instanci naší třídy. Instanci třídy Thread pak spustíme jako vlákno, tj. zavoláme její metodu `start()`.

Metoda `run()` při implementovaném `Runnable` se pak naprosto nijak neliší od `run()` při zděděném `Thread`. Stejně tak se neliší ani vytvoření a spuštění vlákna – zde v metodě `main()`.

Příklad 187:

Vytvoření vlákna implementací `Runnable`.

```
public class Vlakno5 implements Runnable {
    private Thread zobrazVl = null;

    public void start() {
        zobrazVl = new Thread(this);
        zobrazVl.start();
    }

    public void run() {
        while (ReadVl.hotovo == false) {
            System.out.print(ReadVl.suma + "\r");
            try {
                Thread.sleep(100); // 100 milisekund
            }
            catch (InterruptedException e) {
            }
        }
        System.out.println(ReadVl.suma);
    }

    public static void main(String[] args) {
        ReadVl vlCteni = new ReadVl("data.txt");
        vlCteni.start();
        Vlakno5 vlVypis = new Vlakno5();
        vlVypis.start();
    }
}
```

20.5 Čekání na vstup či výstup

V [20.3/318] bylo uvedeno, že vlákno přechází automaticky do stavu neběhuschopné, čeká-li na vstup či výstup. Toto je významná vlastnost vláken, která nám umožňuje bezproblémově vytvářet zejména vstupy z klávesnice. Vstup z klávesnice je totiž o několik řádů pomalejší než ostatní výpočty a není tedy většinou vhodné tyto výpočty brzdit.

Princip je jednoduchý – vstup z klávesnice bude realizován samostatným vláknem, které bude mít přiřazenu nejvyšší prioritu. Protože se však většinu času bude nacházet ve stavu neběhuschopné, nebude tato priorita na závadu. Po stisku klávesy se díky své prioritě okamžitě spustí a načež zadáný znak. Ostatní vlákna, která mají pracovat paralelně, musí mít pouze nižší prioritu, jinak není nutné uvádět **žádný** mechanizmus pro předání řízení (např. `yield()` nebo `sleep()`).

Příklad 188:

V následujícím (vyumělkovaném) příkladě se budou průběžně vypisovat přirozená čísla. Po zadání řetězce, jehož první znak je `K`, celý výpočet skončí. Pokud bude první znak jiný, výpis přirozených čísel bude pokračovat, stejně jako možnost načítání nového řetězce.

Pozor:

Vstup z klávesnice je bufferovaný, tj. po zadání řetězce je nutné stisknout i `<Enter>`. □

Poznámky:

- Třída `Vstup` nemá konstruktor – využívá se implicitní konstruktor, což je možné díky tomu, že třída `Thread` má mimo jiné i konstruktor bez parametrů.
- V metodě `run()` třídy `Vstup` se nastaví maximální priorita vlákna tím, že se statickou metodou `currentThread()` zjistí, jaké vlákno běží. To je samozřejmě vlákno instance `Vstup`, kterému se pak nastaví maximální priorita.

```
import java.io.*;  
  
public class Vstup extends Thread {  
    static public boolean hotovo = false;  
    public void run() {  
        byte[] pole = new byte[10];  
        Thread.currentThread().setPriority(MAX_PRIORITY);  
        while (hotovo == false) {  
            try {  
                System.in.read(pole);  
                if (pole[0] == 'K') {  
                    hotovo = true;  
                }  
            }  
            catch (IOException e) {  
                System.out.println("Chyba vstupu");  
            }  
        }  
    }  
}  
  
public class Vlakno7 extends Thread {  
    public void run() {  
        long i = 0;  
        while (Vstup.hotovo == false) {  
            System.out.print(i++ + "\r");  
        }  
    }  
}  
  
public static void main(String[] args) {  
    Vstup vlVstup = new Vstup();  
    vlVstup.start();  
    Vlakno7 vlVypis = new Vlakno7();  
    vlVypis.start();  
}
```

20.6 Synchronizace vláken

Jak již bylo zmíněno na samém začátku této kapitoly, vlákna mohou být na sobě zcela nezávislá a pak se o sebe navzájem nemusí vůbec starat. Samozřejmě by ale neměla zapomínat na „slušné chování“, tedy zajistit možnost předání řízení.¹²

Mnohem častější ale je, že vlákna spolu více či méně spolupracují a je tedy nutné jejich činnost nějakým způsobem více či méně synchronizovat. V předchozích příkladech byl ukázán nejjednodušší možný způsob synchronizace – pomocí proměnné třídy. Je to ovšem také nejhorší možný způsob, a to z mnoha důvodů. Některé z nich se týkají „čistoty“ programovacího stylu, kdy se snažíme nepředávat parametry pomocí globálních proměnných¹³. Jiné jsou čistě praktické – při použití proměnných třídy je sice možno vytvořit více instancí této třídy, ale program nebude funkční, protože se jednotlivé instance budou „prát“ o jednu jedinou proměnnou.

V této části si ukážeme některé možnosti, jak činnost vláken synchronizovat.

20.6.1 Vlákno čeká trpělivě na konec jiného vlákna

Zde se nejedná o předávání dat, ale z algoritmu úlohy je nutné, aby bylo nějaké vlákno spuštěno až po úplném skončení jiného vlákna.

Poznámka:

Zde se vlákna **nemají** střídat – je to zcela jiná situace, než o jakou jsme se dosud snažili, kdy se vlákna (ne)pravidelně střídala. □

Pokud by se jednalo o program s pouze dvěma vlákny, nemělo by smysl vlákna vůbec používat – stačilo by pouze zavolat sekvenčně první metodu a po jejím skončení pak druhou metodu. Pokud ale bude v programu více vláken (z nichž se některé skutečně střídají), pak je nejjednodušší použít synchronizaci pomocí metody `join()`. Tato metoda čeká na ukončení svého vlákna.

Příklad 189:

Následující příklad bude jemnou modifikací příkladu z [20.2.2/317]. Obě vlákna ponecháme zcela beze změny. To, co se bude měnit, je pouze metoda `main()`, protože budeme chtít změřit čas běhu programu.

¹²Už ale např. víme, že pokud jsou ve vláknu vstupní/výstupní operace, je předání řízení zajištěno automaticky – viz [20.5/323].

¹³Globální proměnné v Javě nejsou, ale proměnné třídy je víceméně nahrazují.

```

public static void main(String[] args) throws
    InterruptedException {
    System.out.println(Thread.currentThread());
    long zac = System.currentTimeMillis();
    ReadV1 vlCteni = new ReadV1("data.txt");
    vlCteni.start();
    Vlakno3 vlVypis = new Vlakno3();
    vlVypis.start();
    //    vlVypis.join();
    long kon = System.currentTimeMillis();
    System.out.println("Konec: " + (kon - zac));
}

```

Pokud bude v metodě `main()` skutečně zakomentována řádku
`vlVypis.join();`

nebude program pracovat tak, jak bychom si přáli. Metoda `main()` běží totiž také v rámci vlákna – o tom se lze snadno přesvědčit příkazem:

```
System.out.print(Thread.currentThread());
```

Takže po spuštění program nejprve vypíše např.:

```
Thread[main,5,main]
```

```
Konec: 81
```

a pak budou spuštěna vlákna realizující vlastní výpočet.

Pokud ale odstraníme komentář před příkazem `vlVypis.join();` bude program fungovat dle očekávání. Vlákno `main` bude čekat na ukončení vlákna `vlVypis` a to, jak víme skončí (využitím statické proměnné `ReadV1.hotovo`) až po ukončení vlákna `vlCteni`. Program pak vypíše:

```
Thread[main,5,main]
```

```
5000050000
```

```
Konec: 3615
```

20.6.2 Vlákno čeká netrpělivě na konec jiného vlákna

Tento příklad je pouze modifikací předchozího příkladu. Metoda `join()` je totiž ve třídě `Thread` přetížena metodami:

```
void join(long milisekund)
```

```
void join(long milisekund, int nanosekund)
```

kdy se na ukončení vlákna čeká jen určený počet milisekund, respektive nanosekund. Toho lze s výhodou využít pro realizaci tzv. *timeoutu*, který

se používá v případě, že nějaká akce nemusí v „rozumném“ čase vůbec proběhnout úspěšně a pak je třeba po určité době umožnit programu nějak na tuto situaci reagovat.

Příklad 190:

V programu se změní jen metoda main(), ve které využijeme metodu isAlive(), jež nás informuje o tom, zda dané vlákno již skončilo či nikoliv. Pokud neskončilo, využijeme metodu join(5000), čili čekání na konec vlákna, nejdéle však 5 sekund. Po této době se vypíše hláška a čeká se dále.

```
public static void main(String[] args) throws
                                                InterruptedException {
    long zac = System.currentTimeMillis();
    ReadVl vlCteni = new ReadVl("data.txt");
    vlCteni.start();
    Vlakno9 vlVypis = new Vlakno9();
    vlVypis.start();
    while (vlVypis.isAlive() == true) {
        vlVypis.join(5000);           // zde čeká 5 vteřin
        System.out.println("\tKde se flakas?!");
    }
    long kon = System.currentTimeMillis();
    System.out.println("Konec: " + (kon - zac));
}
```

20.6.3 Vlákno ukončí předčasně jiné vlákno

V předchozím případě se po vypršení *timeoutu* nedělo nic jiného, než že se po vypsání hlášky *timeout* znova nastavil a to se opakovalo tak dlouho, dokud vlákno testované metodou join() skutečně neskončilo.

Příklad 191:

Zde si ukážeme, jak lze zařídit, aby po vypršení *timeoutu* toto vlákno ihned skončilo – použijeme metodu interrupt(), která dokáže jiné vlákno přerušit. Další metoda je interrupted(), která vrací **true** v případě, že aktuální vlákno bylo přerušeno z jiného vlákna metodou interrupt().

V programu bude nutné kromě main() pozměnit i metodu run(), ve které po zjištění, že byla někým násilně přerušena, dojde k okamžitému ukončení.

```
public class Vlakno10 extends Thread {  
    public void run() {  
        while (ReadVl.hotovo == false) {  
            System.out.print(ReadVl.suma + "\r");  
            yield();  
            if (interrupted() == true) {  
                return;  
            }  
        }  
        System.out.println(ReadVl.suma);  
    }  
  
    public static void main(String[] args) throws  
        InterruptedException {  
        long zac = System.currentTimeMillis();  
        ReadVl vlCteni = new ReadVl("data.txt");  
        vlCteni.start();  
        Vlakno10 vlVypis = new Vlakno10();  
        vlVypis.start();  
        vlVypis.join(1000);  
        if (vlVypis.isAlive() == true) {  
            System.out.println("\t Vyprsel ti cas - koncis!");  
            vlVypis.interrupt();  
        }  
        long kon = System.currentTimeMillis();  
        System.out.println("Konec: " + (kon - zac));  
    }  
}
```

Poznámky:

- Tento program není dotažen do konce. Stejným způsobem by totiž bylo vhodné přerušit i vlákno `vlCteni` a v metodě `run()` třídy `ReadVl` doplnit podmíněný `return` naprosto stejně, jako ve třídě `Vlakno10`.
- Metoda `interrupt()` vlákno pouze přeruší, tj. nepřevede jej do stavu **mrtvé vlákno** – viz též [20.3/318].
- V JDK 1.1 byly metody `stop()`, `suspend()` atp. které umožňovaly vláknu zrušit jiné vlákno. Tyto metody jsou však v JDK 1.2 označeny jako *deprecated* (odmítané) a to z důvodů bezpečnosti.

- JDK 1.2 obsahuje metodu `destroy()`, která vlákno zruší, ovšem tato metoda je prázdná a uživatel si ji musí naprogramovat (tj. překrýt) – z důvodu existence jednoduššího řešení nebyl tento postup použit.

Dobrá rada:

Obecně platí, že přímé ovlivňování stavu jednoho vlákna druhým vláknem je nebezpečná akce, která může vést k zablokování výpočtu. Správný postup je, že se vlákna „neznají“ a komunikují jen přes objekty s metodou označenou `synchronized` – viz [20.6.5/331]. □

20.6.4 Vlákno je násilně probuzeno

V [20.2.2/317] bylo řečeno, že pokud se vlákno uspí metodou `sleep()`, musí být ošetřena výjimka `InterruptedException`, která je vyhozena v případě, že je vlákno násilně probuzeno před uplynutím doby spánku („nedospí“).

Tohoto mechanizmu se dá s výhodou využít, protože volání metody `interrupt()` je právě ta zpráva, která způsobí předčasné probuzení. V programu se změní pouze metoda `run()`, ostatní třídy a metody se nezmění.

Příklad 192:

Vlákno uspané pomocí `sleep()` je probuzeno pomocí metody `interrupt()`.

```
public void run() {
    while (ReadVl.hotovo == false) {
        System.out.print(ReadVl.suma + "\r");
        try {
            Thread.sleep(100); // 100 milisekund
        }
        catch (InterruptedException e) {
            System.out.println("Predcasne vzbuzen");
            break;
        }
    }
    System.out.println(ReadVl.suma);
}
```

Vypíše:

Vyprsel ti cas - koncis!

Predcasne vzbuzen

Konec: 1032

Poznámka:

Protože ani tady není zaslána zpráva o ukončení i vláknu `v1Cteni`, program po výpisu `Konec: 1032` ještě chvíli běží dále, dokud vlákno `vlCteni` nedočte soubor do konce.

Řešení jsou dvě. Můžeme zaslat zprávu o konci i tomuto vláknu, ale pak v něm musí být i zastavovací mechanizmus. Druhým a lepším řešením je vytvořit vlákno `vlCteni` jako démona – viz [20.7.3/340]. □

20.6.5 Kritické sekce – synchronizované metody

Jakmile začneme vytvářet programy, jejichž vlákna pracují nad stejnými daty, dostaneme se k problému označovanému jako **kritické sekce** (*critical sections*). Jedná se o úseky kódu, jejichž paralelní běh může způsobit problémy. Tyto problémy jsou ale mimořádně nepříjemné, protože se vyskytují pouze někdy, když dojde k časovému souběhu nepříznivých událostí.¹⁴

Z tohoto důvodu je nutné kritické sekce tzv. synchronizovat, čili zajistit, aby nemohly probíhat paralelně, ale pouze sériově.

Poznámka:

Je to trochu zvláštní, protože z původního sériového běhu programu (kde k témtoto potížím principiálně nemohlo dojít) se pomocí vláken a určitého úsilí snažíme o paralelní běh. A v tomto paralelním běhu se v některých částech opět snažíme za pomoci dalšího úsilí o seriový běh. ; -)

□

Příklad 193:

Školní ukázka kritické sekce je v následujícím programu. Třída `Bod`¹⁵ má metody `nastav()` a `cti()`, které obě pracují **současně s dvěma údaji**.

```
public class Bod {
    private int[] xy = {0, 0};
    public void nastav(int x, int y) {
        xy[0] = x;      // kritické místo
        xy[1] = y;
    }
    public int[] cti() {
        return new int[] {xy[0], xy[1]};
    }
}
```

¹⁴K tomu ovšem dojde – podle Murphyho zákonů – nikoliv při ladění, ale při předávání programu ; -)

¹⁵Z logického hlediska není příliš šťastně napsaná, ovšem pro ukázku kritické sekce dostatečně vyhovuje.

Pokud bude instance třídy `Bod` využívána jako překladiště dat, např. jedno vlákno bude číst data ze souboru a pomocí metody `nastav()` je ukládat a druhé vlákno je bude odebírat, je to jen otázka času, kdy dojde k chybě. Vlákno, které nastavuje, bude donuceno k předání řízení (viz časové úseky v [20.3.3/320]) v okamžiku označeném jako „kritické místo“, tj. první prvek pole `xy` bude již vyplněn, zatímco druhý nikoliv. Druhé vlákno metodou `cti()` přečte první údaj nový a druhý údaj starý.

Poznámka:

Můžete si to vyzkoušet s třídami `Nastavuje`, `Cte` a `Test1`. □

Aby se ve třídě `Bod` zabránilo popsané kolizi, dovoluje Java použít velmi jednoduchý způsob, pracující na principu **monitoru**.¹⁶

V Javě má každý objekt (díky zdědění od `Object`) svůj jedinečný monitor, který stačí jen jednoduše aktivovat, a to pomocí klíčového slova `synchronized`. Tímto klíčovým slovem se označí všechny metody, které lze považovat za kritické sekce, a od této chvíle jsou tyto metody nepřerušitelné. To znamená, že předávání řízení mezi vlákny nemůže nikdy nastat uprostřed této metody.

Příklad 194:

Nyní už správná (z hlediska kritických sekcí) třída `BodSynchr` bude vypadat takto:

```
public class BodSynchr {
    private int[] xy = {0, 0};

    synchronized public void nastav(int x, int y) {
        xy[0] = x;
        xy[1] = y;
    }

    synchronized public int[] cti() {
        return new int[] {xy[0], xy[1]};
    }
}
```

Při vstupu do **synchronizované kritické sekce** vlákno získá monitor (jakoby uzamkne data) a po jejím opuštění opět monitor uvolní (data

¹⁶ „Monitor je programový modul, určený pro práci v paralelním prostředí. V monitoru jsou zapouzdřena sdílená data spolu s podprogramy, které tato data zpřístupňují. Podprogramy monitoru mají speciální vlastnost – vždy pouze jeden proces může v jednom časovém okamžiku aktivně provádět monitorový podprogram v daném monitoru.“ [Jež]

odemkne). Po získání monitoru jedním vláknem nemůže jiné vlákno zahájit provádění žádné jiné (ani též) kritické sekce patřící témuž objektu. Je nutné zdůraznit, že všechny tyto akce se dějí zcela automaticky bez jakýchkoliv dodatečných zásahů programátora.

Poznámka:

Kritické sekce jsou vždy spojeny se společnými daty nebo zařízením. Principiálně nic nebrání tomu, aby běžely současně dvě kritické sekce, které nesdílí stejný monitor, např. čtení ze dvou různých souborů. □

20.6.6 Kritické sekce - synchronizované bloky

Označení metody jako `synchronized` je velmi jednoduchý a účinný způsob, který nám bude v naprosté většině případů vyhovovat. Může se však stát, že nám vyhovovat přestane, když:

- Chceme jako kritickou sekci označit menší úsek, než je celá metoda.
- Potřebujeme, aby byla kritická sekce svázána s jiným objektem. Toto je hlavní důvod, protože v případě synchronizovaných metod se implicitně používá monitor instance, v jejíž třídě jsou tyto metody deklarovány.

Příklad 195:

Jako příklad si vezmeme na pomoc program z [18.7/291], kdy jsme pracovali se souborem s náhodným přístupem (`RandomAccessFile`). V nějaké třídě (zde `BlokSynchr`), která se čtením ze souboru zabývá pouze okrajově, chceme jako kritickou sekci označit jenom ty úseky, ve kterých se pracuje se souborem, a to tak, aby byly **závislé na objektu souboru**. To je z toho důvodu, že zcela jiná třída může využívat tentýž soubor a pak by monitory nad objekty tříd, které spolu soupeří o soubor, nebyly nic platné.

```
import java.io.*;
public class BlokSynchr {
    RandomAccessFile file;
    BlokSynchr(RandomAccessFile f) {
        file = f;
    }
    public void presun(long kam) throws IOException {
        synchronized (file) {
            file.seek(kam);
        }
    }
}
```

```
public int ctiInt() throws IOException {
    synchronized (file) {
        return file.readInt();
    }
}
// další metody třídy bez vztahu k souboru
}
```

20.6.7 Synchronizace časové posloupnosti vláken

V předchozích dvou částech bylo ukázáno, jak lze jednoduše vytvořit a použít kritické sekce, které zabraňují vzniku nekonzistentních (nehotových) dat. U příkladu z [20.6.5/331] je si ale třeba uvědomit, že není nijak zaručeno, že z „překladiště dat“ budou vždy odebírány ty souřadnice, které byly předchozím nastavením uloženy.

Klidně se může stát, že se jedny a tytéž souřadnice odeberou vícekrát, případně, že se nestáčí odebrat vůbec, protože jsou přepsány novými. Tento problém kritické sekce neřeší!

V literatuře najdete tento problém často popisován jako „úloha producent–konzument“, což velmi přesně vystihuje podstatu problému.

Podstatou je, že producent musí uvědomit konzumenta o připravnosti nových dat a konzument musí čekat, dokud toto upozornění nedostane. Až poté, co data „zkonzumuje“, musí upozornit producenta, že může pokračovat v přípravě dalších dat.

I pro tento případ jsou k dispozici prostředky Javy, protože třída `Object`, od které všichni dědí, obsahuje metody `wait()` (čekej), `notify()` (uvědom) a `notifyAll()`. Všechny tři využívají již zmíněných monitorů kritických sekcí.

Metoda `wait()` zastaví vlákno až do jeho probuzení metodou `notify()` nebo `notifyAll()`, kterou volá jiné vlákno. Z tohoto důvodu je důležité, že se po zavolení `wait()` uvolní monitor objektu, protože jinak by nemohlo dojít k vyvolání `notify()` jiným vláknem.

Stejně jako u metody `join()` (viz [20.6.2/326]) je i metoda `wait()` přetížena metodami

```
void wait(long milisekund)
void wait(long milisekund, int nanosekund)
které čekají jen určenou dobu timeoutu.
```

Metoda `notifyAll()` probudí všechna vlákna, která byla v tomto objektu pozastavena metodou `wait()`. Plánovač pak na základě priority nebo dalších pravidel (viz [20.3.2/319]) určí, které vlákno poběží.

Metoda `notify()` pracuje stejně, ale probudí jen jedno vlákno, přičemž není určeno které.

Příklad 196:

Celý problém si ukážeme na trochu rozsáhlejším příkladě, kdy budeme opět číst ze souboru čísla a počítat jejich součet.¹⁷

Začneme tím, že si vytvoříme třídu `Cteni` pro čtení dat ze souboru. Tato třída nebude odvozena od třídy `Thread`, ale bude využívat metod `wait()` a `notifyAll()`, což je možné díky tomu, že tyto metody pocházejí ze třídy `Object` a nikoliv `Thread`.

Konstruktor `Cteni` pouze otevře požadovaný soubor, který zavře až metoda `finalize()`. Metoda `nacti()` zajistí načtení jednoho čísla ze souboru a jeho uložení do pomocné proměnné `hodnota`. Metoda `predej()` toto číslo předá dále. Po přečtení celého souboru bude aktuální vlákno o této skutečnosti informováno zasláním zprávy o přerušení:

```
Thread.currentThread().interrupt();
```

Poznámka:

Zde je třeba ještě jednou zdůraznit, že metody třídy `Cteni` nejsou vlákna, pouze jsou z vláken volány. □

V této třídě stojí za pozornost také logická proměnná `precteno`, pomocí níž (a metody `wait()`) je zajištěno, že dříve bude vykonávat svoji činnost metoda `nacti()` a ne metoda `predej()`, a to nezávisle na tom, v jakém pořadí budou volány.

```
import java.io.*;

public class Cteni {
    private FileReader fr;
    private BufferedReader in;
    public String jmenoSouboru;
    private int hodnota;
    private boolean precteno = false;
    private boolean konecSouboru = false;
```

¹⁷V tomto případě však nebudeme uživatele o průběhu výpočtu informovat pomocí vypisované sumy, protože to už dostatečně dobře umíme.

```
Cteni(String jmeno) {
    jmenoSouboru = new String(jmeno);
    try {
        fr = new FileReader(jmenoSouboru);
        in = new BufferedReader(fr);
    }
    catch (IOException e) {
        System.out.println("Chyba pri otvirani souboru!");
    }
}

synchronized public void nacti() {
    while (precteno == true) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    String radka;
    try {
        if ((radka = in.readLine()) != null) {
            hodnota = Integer.valueOf(radka).intValue();
            System.out.print(jmenoSouboru + " precteno: "
                + hodnota + "   ");
        }
        else {
            konecSouboru = true;
            Thread.currentThread().interrupt();
        }
    }
    catch (IOException e) {
        System.out.println("Chyba pri cteni souboru!");
    }

    precteno = true;
    notifyAll();
}
```

```
synchronized public int predej() {
    while (precteno == false) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }
    precteno = false;
    if (konecSouboru == false) {
        System.out.println(jmenoSouboru + " predano: "
                           + hodnota + " ");
        notifyAll();
        return hodnota;
    }
    else {
        Thread.currentThread().interrupt();
        return 0;
    }
}

protected void finalize() {
    try {
        fr.close();
    }
    catch (IOException e) {
        System.out.println("Chyba pri zavirani souboru!");
    }
}
```

Instanci třídy Cteni vytvoří main() ve třídě TestPrKon a odkaz na ni předá objektu třídy Producent a současně i Konzument.

```
public class TestPrKon {
    public static void main(String[] args) {
        Cteni ct1 = new Cteni("data10.txt");
        Producent vlPr1 = new Producent(ct1);
        Konzument vlKon1 = new Konzument(ct1);
        vlKon1.start();
        vlPr1.start(); // schválne později než producent
    }
}
```

Obě třídy Producent i Konzument jsou potomky třídy Thread, čili jejich instance zajišťují běhy vláken. Protože obě instance pracují nad jedním objektem třídy Cteni, jsou pomocí metod wait() a notifyAll() volaných v této třídě vlastně synchronizována vlákna vlKon1 a vlPr1.

```
class Producent extends Thread {
    private Cteni c;
    Producent(Cteni c) {
        this.c = c;
    }

    public void run() {
        while (!interrupted() == false) {
            c.nacti();
        }
        System.out.print(" " + c.jmenoSouboru + " - konec cteni ");
    }
}

public class Konzument extends Thread {
    private Cteni c;
    private int suma = 0;

    Konzument(Cteni c) {
        this.c = c;
    }

    public void run() {
        int cislo;
        while (true) {
            cislo = c.predej();
            if (!interrupted() == false)
                suma += cislo;
            else
                break;
        }

        System.out.println(" " + c.jmenoSouboru
                           + " - vysledna suma: " + suma);
    }
}
```

Program vypíše např.:

```
data10.txt precteno: 1  data10.txt predano: 1
data10.txt precteno: 2  data10.txt predano: 2
data10.txt precteno: 3  data10.txt predano: 3
data10.txt precteno: 4  data10.txt predano: 4
data10.txt precteno: 5  data10.txt predano: 5
data10.txt precteno: 6  data10.txt predano: 6
data10.txt precteno: 7  data10.txt predano: 7
data10.txt precteno: 8  data10.txt predano: 8
data10.txt precteno: 9  data10.txt predano: 9
data10.txt precteno: 10 data10.txt predano: 10
    data10.txt - vysledna suma: 55
    data10.txt - konec cteni
```

Možná, že vám připadá uvedený příklad příliš složitý na tak jednoduchou záležitost. Je třeba si ale uvědomit, že tímto důsledným oddělením producenta a konzumenta dostáváme možnost číst stejným způsobem najednou více souborů, a to bez jakýchkoliv dalších starostí o jejich synchronizaci.

Příklad 197:

Více objektů typu producent–konzument, kdy se mění pouze metoda main().

```
public class TestPrKon {
    public static void main(String[] args) {
        Cteni ct1 = new Cteni("data10.txt");
        Producent vlPr1 = new Producent(ct1);
        Konzument vlKon1 = new Konzument(ct1);
        vlKon1.start();
        vlPr1.start(); // schválně později než producent

        Cteni ct2 = new Cteni("data20.txt");
        Producent vlPr2 = new Producent(ct2);
        Konzument vlKon2 = new Konzument(ct2);
        vlPr2.start();
        vlKon2.start();
    }
}
```

20.7 Další informace o vláknech

Problematika vláken je značně rozsáhlá. Dříve uvedené příklady sice stačí na běžnou práci s vlákny, ale pokud budete vytvářet složitější, na vláknech založené programy, budete muset získat mnohem více informací. Vhodným zdrojem je kniha *Concurrent Programming in Java* [Lea], která se zabývá výhradně touto problematikou.

20.7.1 Problematika hladovění a uváznutí

V případě použití vláken musíme být velmi opatrní na to, aby každé vlákno mělo šanci na dostatek zdrojů, čímž je mírněn nejen procesorový čas.

V této souvislosti se používá pojmu **hladovění** (*starvation*), kdy jedno (či více) vláken sice obdrží procesor, ale není schopno provést dostatečný pokrok ve výpočtu, protože je mu bud' procesor vzápětí odebrán, nebo celý vymezený čas jen čeká na uvolnění nějakého zdroje. Ostatní výpočty přitom mohou pokračovat předpokládaným tempem.

Pokud hladovění způsobí zablokování celého výpočtu, hovoříme o **uváznutí** (*deadlock*). Nejčastější příčinou tohoto jevu je situace, kdy dvě vlákna alokují každé jeden zdroj a vzájemně čekají, až ten druhý jej uvolní.

Neexistuje jednoznačný návod, jak se těmto problémům vyhnout. Významnou pomocí jsou však přetížené metody `join()` (viz [20.6.2/326]) a `wait()`, které umožňují čekat jen omezenou dobu. Po uplynutí této doby lze podniknout záchranné akce, např. že vlákno uvolní všechny alokované zdroje.

Ukázku, jak vypadá hladovění (byť ne úplně podle definice) si můžete lehce vyzkoušet, pokud ve třídě `Cteni` z předchozího příkladu upravíte metodu `nacti()` zakomentováním řádky.

```
else {  
    konecSouboru = true;  
//    Thread.currentThread().interrupt();  
}
```

Pak nastane situace, že po přečtení souboru se v metodě `nacti()` čeká ve `wait()` na uvolnění monitoru, které ovšem nikdy nenastane, protože vlákno producenta již úspěšně skončilo.

20.7.2 Skupiny vláken

Pro rozsáhlejší „vláknové“ aplikace je vhodné použít seskupování stejných typů vláken do jednotlivých skupin, se kterými pak lze pracovat najednou. Pro práci se skupinami vláken existuje ve třídě `Thread` množství metod.

Poznámka:

Protože jsme dosud žádné skupiny pro naše vlákna nevyužívali, patřila všechna vlákna do jedné přednastavené (*default*) skupiny. □

20.7.3 Vlákna typu démon

Démon (*daemon*) je speciální typ vlákna. Pokud program používá běžná vlákna, nemůže skončit dříve, než jsou ukončena všechna vlákna. Pokud však některé vlákno označíme jako démona, program skončí bez ohledu na to, zda již toto vlákno doběhlo, či nikoliv.

Příklad 198:

Následující program by s použitím „normálního“ vlákna vypsal během deseti vteřin desetkrát hlášku `daemon je tu` a pak skončil. O tom se lze snadno předvědět zakomentováním řádky:

```
vld.setDaemon(true);
```

Protože je však na této řádce vlákno označeno jako démon, hláška se vypíše pouze jednou a pak program skončí, protože skončilo vlákno metody `main()` a s ním i celý program.

```
public class Daemon extends Thread {  
    public void run() {  
        for (int i = 1; i <= 10; i++) {  
            System.out.println(i + ". daemon je tu");  
            try {  
                Thread.sleep(1000);  
            }  
            catch (InterruptedException e) {  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Zacatek programu");  
        Daemon vld = new Daemon();
```

```
    vld.setDaemon(true);
    vld.start();
    if (vld.isDaemon() == true)
        System.out.println("Program hned skonci");
    else
        System.out.println("Program pobezi dlouho");

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

Vypíše:

```
Zacatek programu
Program hned skonci
Konec programu
1. daemon je tu
```

Těto vlastnosti lze s výhodou využít, pokud máme v programu vlákna, která pouze poskytují služby, a když už o tyto služby není zájem, mohou klidně skončit. To je právě případ vlákna producenta z příkladu v [20.6.7/337]. Označíme-li jej jako démona, nemusíme si dělat starosti s jeho ukončením. V příkladu jsou použity třídy CteniDae, ProducerDae a TestPrKonDae, ze kterých zde budou uvedeny pouze výseky, kde došlo k důležitým změnám oproti již uvedenému příkladu. Ve třídě KonzumentDae k žádným změnám (kromě přejmenování třídy) nedošlo.

```
public class CteniDae {

    . . .

    synchronized public void nacti() {
        . . .

        else {
            konecSouboru = true;
        }
    }

    . . .
}
```

```
public class ProducentDae extends Thread {  
    * * *  
    public void run() {  
        while (true)  
            c.nacti();  
    }  
}  
  
public class TestPrKonDae {  
    public static void main(String[] args) {  
        CteniDae s1 = new CteniDae("data10.txt");  
        ProducentDae vlPrl = new ProducentDae(s1);  
        KonzumentDae vlKonl = new KonzumentDae(s1);  
        vlKonl.start();  
        vlPrl.setDaemon(true);  
        vlPrl.start();  
    }  
}
```

Bez toho, že by vlákno producenta bylo označeno jako démon, by program nikdy neskončil (což jsme si ověřili v [20.7.1/339]). V uvedeném příkladě však program proběhne bez problémů a nekonečná smyčka v metodě `run()` producenta bude přerušena okamžitě po skončení vlákna konzumenta.

Cvičení:

1. Napište program, který bude řešit úlohu producent–konzument pomocí vláken, přičemž vlákna budou spolu komunikovat pomocí roury.

Literatura

- [Cam1] **Campione, M. – Walrath, K.: The Java Tutorial Second Edition.**
Addison-Wesley, U.S.A., 1999
- [Cam2] **Campione, M. – Walrath, K. – Huml, A.: The Java Tutorial Continued.**
Addison-Wesley, U.S.A., 1999
- [Dei] **Deitel, H. M. – Deitel, P. J.: Java – How to Program.**
Prentice Hall, U.S.A., 1997
- [Eck] **Eckel, B.: Thinking in Java.**
Prentice Hall, U.S.A., 1998
- [Fla] **Flanagan, D.: Programování v jazyce Java.**
Computer Press, Brno, 1997
- [Jež] **Ježek, K. – Matějovic, P. – Racek, S.: Paralelní architektury a programy.**
Vydavatelství ZČU, Plzeň, 1999
- [Lea] **Lea, D.: Concurrent Programming in Java.**
Addison-Wesley, U.S.A., 1997
- [Rac] **Racek, S. – Kvoch, M.: Třídy a objekty v C++.**
KOPP, České Budějovice, 1998
- [Tom] **Toman, P. – Kotala, Z.: Java.**
Vydavatelství ZČU, Plzeň, 1999
dione.zcu.cz/java/sbornik.html
- [UJC1] **Herout, P.: Učebnice jazyka C.**
KOPP, České Budějovice, 1997
- [UJC2] **Herout, P.: Učebnice jazyka C – 2. díl.**
KOPP, České Budějovice, 1995
- [Vir] **Virius, M. – Štrupl, D.: JBuilder verze 3, podrobný průvodce.**
Grada, Praha, 1999
- [Wal] **Walrath, K. – Campione, M.: The JFC Swing Tutorial.**
Addison-Wesley, U.S.A., 1999
- [WWW] www.javasoft.com

Rejstřík

- abstract** 162, 163, 185–188, 200
abstract class 186
abstraktní metoda 186
abstraktní třída 186
aktuální balík 202
alert-BELL 38
anonymní vnitřní třída 235
aplet 21
aplikace 21
applet 21
Application Programming Interface 23
array 119
assigment 45
asynchronní výjimky 243
atributy 119
autorizovaný přístup 166, 206, 210

backslash 38
backspace-BS 38
bajtkód 21
bajtově orientované průdy 270
balíky 33, 67, 201
bázová třída 179
běhuschopné vlákno 318
běžící vlákno 318
binární 51
blank final 44
boolean 36, 40, 42, 46, 48, 75, 76,
 81, 108, 112, 152, 160
break 79–81, 85, 87, 88, 90, 91, 260

buffer 278
bugfix version 22
byte 36, 37, 48, 49, 53, 61, 89
byte-code 21

call-by-value 104, 171
carriage return-CR 38
case 89, 90, 96
catch 73, 247, 248, 251–254, 257,
 258, 260, 312
class 120, 162
class 119
Command Prompt 25
Concurrent Programming in Java 339
constructor 125
continue 79–81, 87, 88, 91, 93
couroutines 310
critical sections 330

časová kvanta 320
členské proměnné 113, 119

daemon 340
data encapsulation 166
datové složky 119
dceřiná třída 179
deadlock 339
dědičnost 162
deep copy 194
default 89, 91, 92, 202, 209, 340
default constructor 127

- default package** 204
démon 340
deprecated 328
do 80, 82, 87, 89, 94
double 33, 36, 40, 47–50, 59, 72, 89, 152, 283–285, 287, 293
else 75–78, 95
error 242
„escape“ sekvence 38
exception 241, 242
exceptional event 242
explicitní konverze 47
expression 45
extends 180, 182
false 40, 42, 55–58, 75, 93, 108, 112, 148, 281
field 119
file 288
filtr 271
final 43, 66, 109, 162, 163, 174, 175, 184, 185, 187, 188, 221, 302
final class 187
finalizer 141, 142
finally 257–260
finální třída 162, 187
float 36, 40, 41, 48–50, 52, 59, 89, 108
for 50, 55, 65, 80, 83–89, 95, 96, 102, 109, 141
formální parametry 98, 99, 103
formfeed – FF 38
fully qualified name 109, 201
garbage collector 140, 141, 154, 306, 307, 312
globální proměnná 107, 108
GUI – Graphical User Interface 67
hiding 183
hladovění 339
hluboká kopie 193, 194
hodnotou 104, 171
char 36, 37, 42, 89, 108, 145, 160, 261, 274
checked exceptions 244
chráněný úsek 247
chyba 242
if 55, 75–78, 95, 248
implementovat rozhraní 212
implements 214, 216
implicitní konstruktor 127
implicitní konverze 47
implicitní balík 204
import 202
import balíku 201
inner class 232
instance třídy 119
instance 119
instanceof 222, 223, 231, 237
instanční proměnné 121
int 36, 37, 40, 41, 46, 48–50, 52, 53, 59, 70, 71, 81, 89, 108, 112, 113, 117–119, 169, 197, 274, 280, 283–285, 287, 293, 307

- Java Core API 24
JDK – Java Development Kit 21
Java platforma 24
JVM – Java Virtual Machine 24
jedničkový doplněk 61
JIT kompilátor 24
jméno pole 111
Just In Time 24
- kanál 269
key/value 302
kladné nekonečno 41
klíč 302
klíč/hodnota 302
klonování 190
komentář 31
koncová metoda 162
koncová třída 187
konstanty 33, 43
konstruktor 125
kontrolované výjimky 244
kooperující procesy 310
korutiny 310
kritické sekce 330
- l-hodnota* 45
l-value 45
librarian 201
light-weight 310
load 20
login name 303
lokální proměnná 108
long 36, 37, 48–50, 89, 118
- major version* 22
mělká kopie 192
member variable 119
method 119
metoda 33, 97, 119
metoda třídy 97, 134, 135
metoda instance 97
minor version 22
mnohotvarost 224
modul 245
monitor 331
mrtvé vlákno 318, 328
multiple inheritance 212
- name-spaces* 201
NaN 41
neautorizovaný přístup 136
neběhuschopné vlákno 318
nested class 232
nested comments 32
neúplná jména 202
new 112, 114, 116, 121, 124, 131, 140, 169, 191, 194, 200, 318
newline, linefeed – LF 38
Not a Number 41
nové vlákno 318
null 111, 169, 259, 263
null statement 50
- Oak* 20
obalující třída 35
object 119
objekt 119

- objektový typ 119
oblast viditelnosti 107
odkaz 111
odvozená třída 179
on-line 24
one's complement 61
opakovaně využitelný 180
operátor čárka 85
overloaded 104
overloading 183
overriding 183

package 202, 203
package 201
perzistentní objekty 285
pipe 272, 289
plně kvalifikované jméno 109, 201,
 202
pointer 104
pole 111
polymorfismus 162, 224
potomek 179
prázdná konstanta 44
prázdný příkaz 50
preemptivní plánování 319
prefix 51
primitive data types 35
primitivní datové typy 35
private 136, 166, 167, 199, 200,
 206–208, 210, 211, 233, 300
Products & APIs 21
proměnná instance 108, 121, 131
proměnná třídy 108, 131
proměnná 33
propagace výjimek 246
propagating exceptions 246
properties 302
prostor jmen 107, 201
protected 206, 208–211
proud 269
proudý bajtů 270
proudý znaků 270
předávání parametrů 104
předávání řízení 311
předešek 179
překrytí 183
přetečení 41
přetížená metoda 104
přetížení 183
příkaz 45
přiřazení 45
pseudoparalelní běh vlákna 311
public 34, 109, 120, 156, 162, 163,
 166, 167, 183, 206, 210, 211

quote 38

reference 111
referencí 173, 175
referenční 111
referenční proměnná 111
return 81, 90, 94, 100, 126, 143,
 258, 328
reusability 162
reusable package 245
reusable 180

- rodič 179
 roura 272, 289
 rozhraní 33, 212
 rozšiřující konverze 47, 48
runable 318
 rádkové bufferování 281
 sdílení času 320
self-tested 34
 serializace 285
serialization 285
shallow copy 192
short 36, 48, 49, 89
short circuit 55
simple name 202
single quote 38
sizeof() 42
 skutečné parametry 103
SDK – Standard Development Kit 21
starvation 339
statement 45
static 35, 97, 108, 109, 119–121,
 124, 131, 134, 141, 221, 232
static initialization block 138
static nested class 232
 statická proměnná 108
 statická vnořená třída 232
 statická metoda 97
 statický inicializační blok 44, 138
stream 269
suffix 51
super 182, 183, 189
 supertřída 179
switch 80, 89–91, 93, 227, 231
synchronized 329, 331, 332
 synchronizované kritické sekce 331
 synchronní výjimka 244, 255
system attributes 299
 systémový atribut 299
 šíření výjimek 246
tab - HT 38
 ternární operátor 51
this 123, 127, 128, 130, 133, 137,
 143, 322
thread 310
throw 249, 250, 254
throw exception 242
throws 246
time-slicing 320
timeout 326, 327, 333
top-less class 232
top-level class 232
true 40, 55–58, 75, 78, 82, 83, 148,
 160, 196, 281, 315, 327
try 73, 247, 248, 252, 257–260,
 312
 třída 33, 119
 třída nejvyšší úrovně 232
 unární 50
 úplné jméno 201
 úplné vyhodnocování 57
 uváznutí 339

- veřejná třída 162
vícenásobná dědičnost 212
víceťvarost 224
virtuální stroj 24
vlákno 310
vlastnosti 271, 302
vnější třída 232
vnitřní třída 232
vnořená třída 232
vnořený komentář 32
void 35, 47, 99, 100, 126, 141, 143
volání odkazem 104
vstupní proud 269
vyhodit výjimku 242
výjimečný stav 242
výjimka 241, 242
výraz 45
vyrovnávací paměť 278
výstupní proud 269
vyvolat výjimku 242

while 50, 80–84, 87, 89, 93–95
wrapper class 35

základní datové typy 35
základní třída 179
záporné nekonečno 41
zapouzdření dat 166
zasílání zpráv 124
zastínění 183
zděděná třída 179
zkrácené vyhodnocování 55
znakové orientované proudy 270
znovupoužitelnost 162
zužující konverze 47, 49



Učebnice jazyka Java je knihou, která Vám pomůže překonat první úskalí tohoto programovacího jazyka. Naleznete v ní to, co pro začátky svého programování v Javě potřebujete, ale nečekejte, že je v ní vše – takové množství informací není možné vtěsnat do žádné knížky. Protože problematika programovacího jazyka Java je značně rozsáhlá, jsou v této knize podrobně vysvětleny aspekty jazyka, který se pravděpodobně již nebude výrazně měnit. Programování grafických uživatelských rozhraní bude náplní dalšího dílu.

Kniha je založena na výkladu pomocí příkladů, jichž zde naleznete asi 200. Kromě nich se setkáte s množstvím poznámek, dobrých rad a varování, které byly sepsány na základě zkušeností s tímto jazykem. V knize také naleznete množství odkazů na vzájemné souvislosti, což by Vám mělo pomoci k vytvoření potřebného nadhledu.

Ke knížce si můžete dokoupit disketu s více než čtyřmi sty kompletně odladěnými programy.

Autor (*1961) pracuje jako odborný asistent na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Zabývá se výukou softwarově orientovaných předmětů včetně jazyka Java. Je autorem sedmi odborných knih a čtyř skript. Jejich oblíba je přelíčitána i tomu, že je v nich uvedeno něco víc, než bývá v manuálech – jsou psány srozumitelnou formou, jsou v nich uvedeny osobní praktické zkušenosti a popisovaná problematika je ta, se kterou se při programování setkáte.



Zdrojový kód všech příkladů a cvičení najdete na disketě, kterou je možno ke knize objednat v nakladatelství.



nakladatelství

Šumavská 3, 370 01 České Budějovice
Tel./fax: 038 - 646 04 74, e-mail: knihy@kopp.cz
Aktuální nabídka: Internet: <http://www.kopp.cz>
Cena knihy 199,- Kč včetně DPH
Cena diskety 69,- Kč včetně DPH

ISBN 80-7232-115-3

9 788072 321155