KAPITOLA 2

Seznámení s datovými typy a operátory

Klíčové dovednosti a pojmy

- Seznámení s primitivními typy jazyka Java.
- Používání literálů.
- Inicializace proměnných.
- Seznámení s pravidly oborů proměnných uvnitř metody.
- Používání aritmetických operátorů.
- Používání relačních a logických operátorů.
- Porozumění operátorům přiřazení.
- Používání zkrácených přiřazení.
- Porozumění převodu typu v přiřazeních.
- Přetypování nekompatibilních typů.
- Pochopení převodu typu ve výrazech.

V základech každého programovacího jazyka jsou jeho datové typy a operátory, přičemž Java není žádnou výjimkou. Tyto prvky definují limity jazyka a určují druh úloh, na které jej lze aplikovat. Jazyk Java naštěstí podporuje bohatý sortiment datových typů i operátorů, díky čemuž je vhodný pro jakýkoli typ programování.

Datové typy a operátory jsou rozsáhlé téma. Začneme studiem základních datových typů jazyka Java a jeho nejčastěji používaných operátorů. Kromě toho se blíže podíváme na proměnné a prozkoumáme výrazy.

Proč jsou datové typy důležité

Datové typy jsou v Javě zvláště důležité, protože jde o silně typovaný jazyk. To znamená, že u všech operací kompilátor kontroluje kompatibilitu typů. Nelegální operace se nezkompilují. Silná typová kontrola tedy pomáhá předcházet chybám a zlepšuje spolehlivost. Silná typová kontrola ovšem vyžaduje, aby všechny proměnné, výrazy a hodnoty měly nějaký typ. Není zde tedy například žádná koncepce „beztypové" proměnné. Kromě toho typ hodnoty určuje, které operace jsou na něm povolené. Operace povolená na jednom typu nemusí být povolená na jiném.

Primitivní typy jazyka

Java obsahuje dvě obecné kategorie vestavěných datových typů: objektově orientované a neobjektově orientované. Objektově orientované typy jazyka Java definují třídy, jimž se budeme věnovat později. Nicméně v srdci jazyka Java je osm primitivních (nebo také elementárních či jednoduchých) typů dat, které ukazuje tabulka 2.1. Pojem *primitivní* se zde používá pro signalizaci, že tyto typy nejsou v objektově orientovaném smyslu objekty, ale spíše normální binární hodnoty. Tyto primitivní typy nejsou objekty kvůli efektivitě. Všechny ostatní datové typy Javy se konstruují z těchto primitivních typů.

Tabulka 2.1: Vestavěné primitivní datové typy jazyka Java

Typ	Význam
`boolean`	Představují hodnoty pravda a nepravda
`byte`	8bitové celé číslo
`char`	Znak
`double`	Plovoucí desetinná čárka s dvojnásobnou přesností
`float`	Plovoucí desetinná čárka s dvojnásobnou přesností
`int`	Celé číslo
`long`	Dlouhé celé číslo
`short`	Krátké celé číslo

Java přesně určuje rozsah a chování pro každý primitivní typ, který musejí všechny implementace virtuálního stroje Javy podporovat. Vzhledem k požadavku Javy na přenositelnost je Java v tomto ohledu nekompromisní. Kupříkladu typ `int` je stejný ve všech prováděcích prostředích. Díky tomu mohou být programy plně přenositelné, takže není nutné přepisovat kód pro konkrétní platformu. Přestože striktní specifikace rozsahu primitivních typů může v některých prostředích způsobit malou ztrátu výkonu, pro přenositelnost je to naprosto nezbytné.

Celá čísla

Jazyk Java definuje čtyři celočíselné typy: `byte`, `short`, `int` a `long`. Jejich popis uvádí následující tabulka:

Typ	Šířka v bitech	Rozsah
`byte`	8	–128 až 127
`short`	16	–32 768 až 32 767
`int`	32	–2 147 483 648 až 2 147 483 647
`long`	64	–9 223 372 036 854 775 808 až 9 223 372 036 854 775 807

Jak je z této tabulky patrné, všechny celočíselné typy mají kladné i záporné hodnoty se znaménkem. Java nepodporuje bezznaménková (pouze kladná) celá čísla. Řada jiných počítačových jazyků podporuje znaménková i bezznaménková celá čísla. Návrháři Javy však cítili, že bezznaménková celá čísla jsou zbytečná.

POZNÁMKA

Z technického hlediska může běhový systém Javy používat pro uchovávání primitivního typu libovolnou požadovanou velikost. Nicméně ve všech případech se typy musejí chovat dle uvedené specifikace.

Nejčastěji používaným celočíselným typem je `int`. Proměnné typu `int` se často používají pro řízení cyklů, indexování polí a provádění obecných celočíselných výpočtů.

Potřebujete-li celé číslo s větším rozsahem, než má typ `int`, použijte typ `long`. Zde je například program, který vypočítá počet kubických palců obsažených v krychli o velikosti strany jedna míle:

```
/*
  Vypočítá počet kubických palců v 1 krychlové míli.
*/
class Palce {
  public static void main(String args[]) {
    long kb;
    long pm;

    pm = 5280 * 12;

    kb = pm * pm * pm;

    System.out.println("V krychlové míli je " + kb +
                       " kubických palců.");
  }
}
```

Zde je výstup z tohoto programu:

```
V krychlové míli je 254358061056000 kubických palců.
```

Je jasné, že výsledek by nemohla uchovávat proměnná typu `int`.

Nejmenším celočíselným typem je `byte`. Proměnné typu byte jsou zvláště užitečné při práci s holými binárními daty, jež nemusejí být přímo kompatibilní s dalšími vestavěnými typy Javy. Typ `short` vytváří krátká celá čísla. Proměnné typu `short` se hodí tehdy, když nepotřebujete velký rozsah nabízený typem `int`.

Otázka pro experta

Otázka: Říkáte, že existují čtyři celočíselné typy: `int`, `short`, `long` a `byte`. Jenže já jsem slyšel(a), že mezi celočíselné typy jazyka Java lze zařadit také typ `char`. Můžete to vysvětlit?

Odpověď: Formální specifikace pro jazyk Java definuje kategorii typů s názvem integrální (celé) typy, jež obsahuje `byte`, `short`, `int`, `long` a `char`. Nazývají se integrální typy, protože uchovávají celočíselné, binární hodnoty. Jenže smyslem prvních čtyř je reprezentovat numerické celočíselné veličiny. Účelem typu `char` je reprezentovat znaky. Principiální použití typu `char` se tedy podstatně liší od principiálního použití ostatních integrálních typů. Kvůli těmto odlišnostem je v této knize typ `char` považován za samostatný typ.

Typy s plovoucí desetinnou čárkou

Jak jsme si řekli v kapitole 1, typy s plovoucí desetinnou čárkou mohou reprezentovat čísla, jež mají zlomkové části. Existují dva druhy typů s plovoucí desetinnou čárkou, `float`, resp. `double`, které představují čísla s jednoduchou, resp. dvojitou, přesností. Typ `float` je široký 32 bitů a typ `double` je široký 64 bitů.

Z obou typů se nejčastěji používá typ `double`, protože veškeré matematické funkce v knihovně třídy Javy používají hodnoty typu `double`. Například metoda `sqrt()` (kterou definuje standardní třída `Math`) vrací hodnotu typu `double`, která je odmocninou jejího argumentu typu `double`. V níže uvedeném příkladu se metoda `sqrt()` používá pro výpočet délky přepony z daných délek odvěsen:

```
/*
  Používá Pythagorovu větu pro nalezení délky přepony
  ze zadaných délek odvěsen.
*/
class Prepona {
  public static void main(String args[]) {
    double x, y, z;

    x = 3;
```

```
    y = 4;

    z = Math.sqrt(x*x + y*y); <-- Všimněte si, jak se metoda sqrt() volá. Je
                                  před ní uveden název třídy, jejíž je členem.
    System.out.println("Přepona je " +z);
  }
}
```

Výstup z tohoto programu vypadá následovně:

```
Přepona je 5.0
```

Ještě jedna věc ohledně předchozího příkladu: Jak již bylo řečeno, metoda `sqrt()` je členem standardní třídy `Math`. Všimněte si, jak se metoda `sqrt()` volá – je před ní uveden název `Math`. To je podobné, jako když je před `println()` uvedeno `System.out`. I když se ne všechny standardní metody volají s uvedením jména jejich třídy, některé se volají tímto způsobem.

Znaky

Znaky v Javě nejsou jako v řadě dalších počítačových jazyků 8bitové veličiny. Java totiž používá znakovou sadu Unicode. Unicode definuje znakovou sadu, která může reprezentovat všechny znaky ve všech lidských jazycích. V jazyce Java je `char` bezznaménkový 16bitový typ s rozsahem 0 až 65 536. Standardní 8bitová znaková sada ASCII je podmnožinou znakové sady Unicode a používá rozsah 0 až 127. Znaky ASCII jsou tedy rovněž platnými znaky Javy.

Znakové proměnné lze přiřadit hodnotu uzavřením znaku do jednoduchých uvozovek. Kupříkladu tímto způsobem se přiřadí písmeno „X“ do proměnné `zn`:

```
char zn;
zn = 'X';
```

Znakovou hodnotu můžete vypsat pomocí příkazu `println()`. Například tento řádek vypíše hodnotu v proměnné `zn`:

```
System.out.println("Toto je zn: " + zn);
```

Typ `char` je bezznaménkovým 16bitovým typem, a proto je možné provádět na proměnné typu `char` nejrůznější aritmetické manipulace. Podívejte se například na následující program:

```
// Se znakovými proměnnými lze pracovat jako s celými čísly.
class UkazkaZnakAritm {
  public static void main(String args[]) {
    char zn;

    zn = 'X';
```

```
    System.out.println("zn obsahuje " + zn);

    zn++; // inkrementuj zn  <-- Znak lze dekrementovat
    System.out.println("zn je nyní " + zn);

    zn = 90; // přiřaď do zn hodnotu Z  <-- Znaku lze přiřadit celočíselnou hodnotu
    System.out.println("zn je nyní " + zn);
  }
}
```

Výstup generovaný tímto programem vypadá takto:

```
zn obsahuje X
zn je nyní Y
zn je nyní Z
```

V tomto programu se do proměnné `zn` nejdříve uloží hodnota X. Potom se proměnná `zn` inkrementuje. Výsledkem je, že proměnná `zn` obsahuje Y, což je další znak v posloupnosti znakové sady ASCII (a Unicode). Dále se do proměnné `zn` přiřadí hodnota 90, což je hodnota znakové sady ASCII (a Unicode) pro písmeno Z. Vzhledem k tomu, že znaková sada ASCII zabírá prvních 127 hodnot ve znakové sadě Unicode, budou všechny „staré triky", které jste možná prováděli se znaky v jiných jazycích, fungovat také v jazyce Java.

Otázka pro experta

Otázka: Proč Java používá znakovou sadu Unicode?

Odpověď: Java byla navržená pro celosvětové použití. Z tohoto důvodu musí používat znakovou sadu, která dokáže reprezentovat jazyky na celém světě. Unicode je standardní znaková sada navržená právě k tomuto účelu. Používání znakové sady Unicode je samozřejmě neefektivní v případě jazyků, jako je angličtina, němčina, španělština nebo čeština, jejichž znaky lze uchovávat v rámci 8 bitů. To je ale cena, kterou je nutné zaplatit za globální přenositelnost.

Pravdivostní typ

Typ boolean představuje hodnoty pravda a nepravda. Java definuje hodnoty pravda a nepravda pomocí rezervovaných slov `true` a `false`. Proměnná či výraz typu `boolean` bude mít tedy jednu z těchto hodnot.

Zde je program, který demonstruje typ `boolean`:

```
// Demonstruje hodnoty typu boolean.
class UkazkaBool {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b je " + b);
    b = true;
    System.out.println("b je " + b);

    // hodnota typu boolean může řídit příkaz if
    if (b) System.out.println("Toto se provede.");

    b = false;
    if (b) System.out.println("Toto se neprovede.");

    // výsledkem relačního operátoru je hodntoa typu boolean
    System.out.println("10 > 9 je " + (10 > 9));
  }
}
```

Výstup vygenerovaný tímto programem je uveden níže:

```
b je false
b je true
Toto se provede.
10 > 9 je true
```

Na tomto programu si můžete všimnout tří zajímavých věcí. Za prvé, při vypisování hodnoty typu boolean metodou `println()` se zobrazí `"true"` nebo `"false"`. Za druhé, již samotná hodnota proměnné typu boolean stačí pro řízení příkazu `if`. Není tedy nutné psát příkaz `if` takto:

```
if (b == true) ...
```

Za třetí, výsledkem relačního operátoru, jako je <, je hodnota typu `boolean`. Proto výraz 10 > 9 zobrazí hodnotu `"true"`. Kromě toho je nutné uvést kolem výrazu `10 > 9` další sadu závorek, protože operátor + má vyšší prioritu než operátor >.

Vyzkoušejte 2.1: Jak daleko uhodil blesk?

Zvuk.java

V tomto projektu vytvoříte program, který vypočítá, jak daleko (v metrech) je posluchající osoba od místa, kde uhodil blesk. Zvuk se pohybuje vzduchem přibližnou rychlostí 335 metrů za vteřinu. Znáte-li tedy čas mezi okamžikem, kdy se objeví blesk,

a okamžikem, kdy dorazí zvuk, můžete vypočítat vzdálenost k místu jeho úderu. Pro tento projekt předpokládáme, že tento čas činí 7,2 vteřin.

1. Vytvořte nový soubor s názvem `Zvuk.java`.
2. Pro výpočet vzdálenosti budete potřebovat použít hodnoty s plovoucí desetinnou čárkou. Proč? Protože čas (7,2) má zlomkovou část. Přestože by bylo přípustné použít hodnoty typu `float`, použijeme v tomto příkladu typ `double`.
3. Pro výpočet vzdálenosti vynásobíte 7,2 hodnotou 335. Výsledek pak přiřadíte do proměnné a poté jej zobrazíte.
4. Zde je celý program `Zvuk.java`:

```
/*
  Vyzkoušejte 2.1
  Vypočítá vzdálenost k místu úderu blesku,
  jehož zvuku trvá cesta k vám 7,2 vteřin.
*/
class Zvuk {
  public static void main(String args[]) {
    double dist;

    dist = 7.2 * 335;

    System.out.println("Blesk uhodil " + dist +
                       " metrů daleko.");
  }
}
```

5. Program zkompilujte a spusťte. Zobrazí se následující výsledek:

```
Blesk uhodil 2412.0 metrů daleko.
```

6. Mimořádná výzva: Můžete vypočítat vzdálenost k velkému objektu, jako je skalnatá stěna, podle doby ozvěny. Pokud například tlesknete rukama a změříte čas, než uslyšíte ozvěnu, pak znáte celkovou dobu cesty zvuku tam i zpět. Při podělení této hodnoty dvěma získáte čas, který zvuk potřebuje na cestu jedním směrem. Pomocí této hodnoty pak můžete vypočítat vzdálenost k danému objektu. Upravte tedy předchozí program tak, aby počítal tuto vzdálenost s předpokladem, že časovým intervalem odpovídá ozvěně.

Literály

V jazyce Java označují *literály* fixní hodnoty, které jsou reprezentované ve formě čitelné pro člověka. Kupříkladu číslo `100` je literál. Literály se také obvykle nazývají *konstanty*. Použití literálů je povětšinou tak intuitivní, že jsme je v té či oné podobě používali ve všech předchozích ukázkových programech. Nyní nastal čas představit si je na formální úrovni.

Literály jazyka Java mohou být kteréhokoli z primitivních datových typů. Způsob, jakým se každý literál reprezentuje, závisí na jeho typu. Jak jsme si řekli již dříve, znakové konstanty se uzavírají do jednoduchých uvozovek. Například `'a'` i `'%'` jsou znakové konstanty.

Celočíselné literály se uvádějí jako čísla bez zlomkové části. Například `10` a `-100` jsou celočíselné literály. Literály s plovoucí desetinnou čárkou vyžadují použití desetinné tečky, za níž následuje zlomková část daného čísla. Kupříkladu `11.123` je literál s plovoucí desetinnou čárkou. Java dále umožňuje používat pro zápis čísel s plovoucí desetinnou čárkou vědeckou notaci.

Celočíselné literály jsou standardně typu `int`. Chcete-li uvést literál typu `long`, připojte k němu `l` nebo `L`. Například `12` je typu `int`, ale `12L` je typu `long`.

Literály s plovoucí desetinnou čárkou jsou standardně typu `double`. Chcete-li uvést literál typu `float`, připojte ke konstantě `f` nebo `F`. Například `10.19F` je typu `float`.

Třebaže celočíselné literály vytvářejí standardně hodnotu typu `int`, lze je i tak přiřazovat do proměnných typu `char`, `byte` nebo `short`, je-li ovšem možné přiřazovanou hodnotu reprezentovat cílovým typem. Celočíselný literál je možné vždy přiřadit do proměnné typu `long`.

Počínaje verzí JDK 7 můžete do celočíselného literálu nebo do literálu s plovoucí desetinnou čárkou vložit jedno či více podtržítek. To pak může usnadnit čtení hodnot s mnoha ciframi. Při kompilování literálu se podtržítka jednoduše zahodí. Zde je příklad:

```
123_45_1234
```

Tento výraz stanoví hodnotu 123 451 234. Použití podtržítek je zvláště užitečné při kódování takových věcí, jako jsou typová označení, identifikátory zákazníků a stavové kódy, které se obvykle rozdělují na podskupiny číslic.

Šestnáctkové, osmičkové a binární literály

Jak asi možná víte, v programování je někdy snadnější používat číselnou soustavu nezaloženou na 10, ale spíše na 8 nebo 16. Číselná soustava založená na 8 se nazývá osmičková a používá číslice 0 až 7. V osmičkové soustavě je číslo 10 stejné jako 8 v desítkové. Číselná soustava o základu 16 se nazývá šestnáctková a používá číslice 0 až 9 plus písmena A až F, která znamenají 10, 11, 12, 13, 14 a 15. Například šestnáctkové číslo 10 je 16 v desítkové soustavě. Vzhledem k častému využívání těchto dvou číselných soustav umožňuje jazyk Java zapisování celočíselných literálů také v šestnáctkové nebo osmičkové soustavě. Šestnáctkový literál musí začínat `0x` nebo `0X` (nula následovaná x nebo X). Osmičkový literál začíná nulou. Zde je několik příkladů:

```
s16 = 0xFF; // 255 v desítkové soustavě
s8 = 011;   // 9 v desítkové soustavě
```

Zajímavé je, že Java navíc umožňuje zápis šestnáctkových literálů s plovoucí desetinnou čárkou, které se však používají jen velice zřídka.

Počínaje verzí JDK 7 můžete zapisovat celočíselný literál pomocí binární soustavy. K tomu stačí uvést před binárním číslem `0b` nebo `0B`. Například takto můžete zapsat hodnotu 12 v binární soustavě: `0b1100`.

Znakové řídicí sekvence

Uzavření znakových konstant do jednoduchých uvozovek funguje pro většinu tisknutelných znaků, je tu ale několik znaků, jako je návrat na začátek, představujících zvláštní problém při používání textového editoru. Kromě toho některé další znaky, jako jsou jednoduché a dvojité uvozovky, mají v Javě zvláštní význam, takže je nemůžete použít přímo. Z těchto důvodů nabízí Java speciální *řídicí sekvence*, někdy označované jako lomítkové znakové konstanty, uvedené v tabulce 2.2. Tyto sekvence se používají místo znaku, který reprezentují.

Tabulka 2.2: Znakové řídicí sekvence

Řídicí sekvence	Popis
`\'`	Jednoduché uvozovky
`\"`	Dvojité uvozovky
`\\`	Zpětné lomítko
`\r`	Návrat na začátek
`\n`	Nový řádek
`\f`	Posun strany
`\t`	Vodorovný tabulátor
`\b`	Backspace
`\ddd`	Osmičková konstanta (kde ddd je osmičková konstanta)
`\uxxxx`	Šestnáctková konstanta (kde xxxx je šestnáctková konstanta)

Kupříkladu tento příkaz přiřadí proměnné `zn` znak tabulátoru:

```
zn = '\t';
```

Další příkaz přiřadí proměnné `zn` jednoduché uvozovky:

```
zn = '\'';
```

Řetězcové literály

Java podporuje ještě jeden typ literálů: řetězec. *Řetězec* je skupina znaků uzavřená dvojitými uvozovkami. Například

```
"toto je test"
```

je řetězec. Příklady řetězců jste viděli v mnoha příkazech `println()` v předchozích ukázkových programech.

Kromě běžných znaků může řetězcový literál obsahovat také jednu či více právě popsaných řídicích sekvencí. Podívejte se například na následující program, který používá řídicí sekvence `\n` a `\t`:

```
// Demonstruje řídicí sekvence v řetězcích.
class UkazkaRetezec {
  public static void main(String args[]) {
    System.out.println("První řádek\nDruhý řádek"); ← Pro vygenerování nového řádku používejte \n
    System.out.println("A\tB\tC");      ← Pro zarovnání výstupu
    System.out.println("D\tE\tF") ;     ← používejte tabulátory
  }
}
```

Výstup vypadá takto:

```
První řádek
Druhý řádek
A       B       C
D       E       F
```

Otázka pro experta

Otázka: Je řetězec sestávající z jednoho znaku totéž jako znakový literál? Je tedy například `"k"` stejné jako `'k'`?

Odpověď: Ne. Nesmíte si plést řetězce se znaky. Znakový literál reprezentuje jediné písmeno typu `char`. Řetězec obsahující pouze jedno písmeno je i nadále řetězec. Přestože se řetězce skládají ze znaků, nejsou téhož typu.

Všimněte si, jak je použitá řídicí sekvence `\n` pro vygenerování nového řádku. Pro víceřádkový výstup tedy nemusíte používat několik příkazů `println()`. Stačí jen do delšího řetězce vložit na místa, kde se má přejít na nový řádek, sekvenci `\n`.

Bližší pohled na proměnné

Proměnné jsme si představili v kapitole 1. Zde se na ně podíváme blíže. Jak jste se dozvěděli již dříve, proměnné se deklarují pomocí příkazu

```
typ název-proměnné;
```

kde *typ* je datový typ proměnné a *název-proměnné* je její název. Můžete deklarovat proměnnou libovolného platného typu, včetně právě popsaných jednoduchých typů. Při vytváření proměnné vytváříte instanci jejího typu. Schopnosti proměnné tedy určuje její typ. Například proměnné typu `boolean` nelze používat pro uchovávání hodnot s plovoucí desetinnou čárkou. Kromě toho typ proměnné se během jejího života nemůže měnit. Kupříkladu proměnná typu `int` se nemůže změnit na proměnnou typu `char`.

V jazyce Java je nutné všechny proměnné před jejich použitím deklarovat. To je nezbytné, protože kompilátor musí před náležitým zkompilováním jakéhokoli příkazu používajícího danou proměnnou vědět, jaký typ dat tato proměnná obsahuje. Java tak navíc může provádět striktní typovou kontrolu.

Inicializace proměnné

Obecně lze říci, že proměnné musíte před jejím použitím dát nějakou hodnotu. Jeden ze způsobů, jak dát proměnné hodnotu, nabízí přiřazovací příkaz, s nímž jste se již setkali. Další možností je dát proměnné počáteční hodnotu při její deklaraci. V takovém případě stačí jen za název proměnné napsat rovnítko a přiřazovanou hodnotu. Obecný tvar inicializace vypadá následovně:

```
typ prom = hodnota;
```

Zde je *hodnota* hodnotou přiřazovanou do proměnné *prom* při vytváření této proměnné. Přiřazovaná hodnota musí být kompatibilní s uvedeným typem. Zde je několik příkladů:

```
int pocet = 10; // dej proměnné pocet počáteční hodnotu 10
char zn = 'X';  // inicializuj proměnnou zn písmenem X
float f = 1.2F; // proměnná f je inicializovaná hodnotou 1.2
```

Při deklarování dvou nebo více proměnných téhož typu pomocí seznamu s prvky oddělenými čárkami můžete dát jedné nebo více proměnným počáteční hodnotu:

```
int a, b = 8, c = 19, d; // b a c mají inicializace
```

V tomto případě se inicializují pouze proměnné `b` a `c`.

Dynamická inicializace

I když předchozí příklady používaly pro inicializaci pouze konstanty, umožňuje jazyk Java inicializovat proměnné dynamicky, pomocí libovolného výrazu platného v době deklarace dané proměnné. Například zde je krátký program, který počítá objekt válce podle průměru základy a výšky.

```
// Demonstruje dynamickou inicializaci.
class DynInicializace {
  public static void main(String args[]) {
```

```
        double polomer = 4, vyska = 5;

        // dynamicky inicializuj objem
        double objem = 3.1416 * polomer * polomer * vyska;
                                        proměnná objem se dynamicky inicializuje za běhu

        System.out.println("Objem je " + objem);
    }
}
```

Zde se deklarují tři lokální proměnné: `polomer`, `vyska` a `objem`. První dvě, `polomer` a `vyska`, se inicializují konstantou. Nicméně `objem` se inicializuje dynamicky na objem válce. Podstatné je zde to, že inicializační výraz může používat libovolné prvky platné v době inicializace, včetně volání metod, dalších proměnných nebo literálů.

Obor a doba platnosti proměnných

Až dosud byly všechny námi používané proměnné deklarované na začátku metody `main()`. Jazyk Java ovšem umožňuje deklarovat proměnné uvnitř libovolného bloku. Jak jsme si vysvětlili v kapitole 1, blok začíná otevírací složenou závorkou a končí uzavírací složenou závorkou. Blok definuje *obor*. Proto při každém zahájení nového bloku vytváříte nový obor. Obor určuje, které objekty jsou viditelné ostatním částem programu. Dále určuje dobu platnosti těchto objektů.

Řada ostatních počítačových jazyků definuje dvě obecné kategorie oborů: globální a lokální. Java je sice také podporuje, nejedná se ale o nejlepší způsob kategorizace oborů v Javě. Nejdůležitější obory v Javě jsou ty, které definuje třída, a ty, které definuje metoda. Oboru tříd (a v něm deklarovaným proměnným) se budeme věnovat v pozdější části této knihy při popisu tříd. Prozatím budeme zkoumat pouze obory definované metodou nebo uvnitř metody.

Obor definovaný metodou začíná otevírací složenou závorkou. Má-li ovšem metoda parametry, jsou i ony zahrnuty do oboru dané metody.

Obecným pravidlem je, že proměnné deklarované uvnitř oboru nejsou viditelné (tj. přístupné) v kódu, který je definovaný mimo tento obor. Proto při deklarování proměnné uvnitř nějakého oboru vlastně provádíte její ohraničení, které ji chrání před neautorizovaným přístupem či modifikací. Pravidla oborů tak poskytují základ pro zapouzdření.

Obory lze vnořovat. Například při každém vytvoření bloku kódu vytváříte nový, vnořený obor. Vnější obor pak obklopuje vnitřní obor. To znamená, že objekty deklarované ve vnějším oboru budou viditelné pro kód uvnitř vnitřního oboru. Obráceně to ovšem neplatí. Objekty deklarované uvnitř vnitřního oboru nebudou mimo něj viditelné.

Pro pochopení efektu vnořených oborů si ukážeme následující program:

```
// Demonstruje obor bloku.
class UkazkaObor {
  public static void main(String args[]) {
    int x; // dostupná pro celý kód uvnitř metody main

    x = 10;
    if (x == 10) { // začni nový obor

      int y = 20;  // dostupná pouze pro tento blok

      // zde jsou dostupné obě proměnné x a y.

      System.out.println("x a y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Chyba! Proměnná y zde není dostupná
                                    Zde je proměnná y mimo svůj obor
    // x je zde i nadále dostupná.
    System.out.println("x je " + x);
  }
}
```

Jak již naznačují komentáře, proměnná x je deklarovaná na začátku oboru metody `main()`, a je tedy přístupná všemu následujícímu kódu uvnitř metody `main()`. Uvnitř bloku `if` se deklaruje proměnná `y`. Blok definuje obor, a proto je proměnná `y` viditelná pouze pro další kód v rámci tohoto bloku. Z tohoto důvodu je řádek `y = 100` vně tohoto bloku deaktivovaný komentářem. Pokud byste počáteční symbol komentáře odstranili, došlo by k chybě v době kompilace, protože proměnná `y` není mimo svůj blok viditelná. Uvnitř bloku `if` lze používat také proměnnou `x`, protože kód uvnitř bloku (tj. vnořeného oboru) má přístup k proměnným, které jsou deklarované obklopujícím oborem.

Uvnitř bloku lze deklarovat proměnné na libovolném místě, platné jsou však až za místem deklarace. Pokud tedy definujete proměnnou na začátku metody, bude k dispozici pro veškerý kód uvnitř této metody. A naopak, deklarujete-li proměnnou na konci bloku, bude v podstatě k ničemu, protože k ní nebude mít přístup žádný kód.

Zde je další důležitá věc na zapamatování: Proměnné se vytvářejí při vstupu do jejich oboru a po opuštění jejich oboru se likvidují. To znamená, že proměnná si neuchová svoji hodnotu, jakmile se provádění dostane mimo její obor. Proměnné deklarované uvnitř metody si proto neuchovají své hodnoty mezi jednotlivými voláními dané metody. Dále proměnná deklarovaná uvnitř bloku ztratí po opuštění tohoto bloku svoji hodnotu. Doba platnosti proměnné je tedy omezená na její obor.

Obsahuje-li deklarace proměnné inicializační část, provede se při každém vstupu do bloku, v němž je deklarovaná, opětovná inicializace. Podívejte se například na tento program:

```
// Demonstruje dobu platnosti proměnné.
class UkazkaInicialProm {
  public static void main(String args[]) {
    int x;

    for (x = 0; x < 3; x++) {
      int y = -1; // proměnná y se inicializuje při každém vstupu do bloku
      System.out.println("y je: " + y); // zde se vždy vypíše -1
      y = 100;
      System.out.println("y je nyní: " + y);
    }
  }
}
```

Výstup vygenerovaný tímto programem vypadá takto:

```
y je: -1
y je nyní: 100
y je: -1
y je nyní: 100
y je: -1
y je nyní: 100
```

Jak můžete vidět, proměnná `y` se při každém vstupu do vnitřního cyklu vždy znovu inicializuje na hodnotu `-1`. Přestože je jí následně přiřazená hodnota `100`, tuto hodnotu ztratí.

S pravidly oborů jazyka Java souvisí jedna zvláštnost, která vás možná překvapí. Třebaže lze bloky vnořovat, nemůže mít žádná proměnná deklarovaná uvnitř vnitřního oboru stejný název jako proměnná deklarovaná obklopujícím oborem. Například následující program, který se pokouší deklarovat dvě samostatné proměnné se stejným názvem, se nezkompiluje.

```
/*
  Tento program se snaží deklarovat ve vnitřním oboru proměnnou
  se stejným názvem, jako má proměná definovaná ve vnějším oboru.

  *** Tento program se nezkompiluje. ***
*/
class VnorProm {
  public static void main(String args[]) {
```

```
    int pocet; ←──────────────────────────────────────┐
                                                      │
    for (pocet = 0; pocet < 10; pocet = pocet + 1) {  │
      System.out.println("Proměnná pocet: " + pocet);│
                                                      │
      int pocet; // neplatné!!! ←──┐                  │
      for (pocet = 0; pocet < 2; pocet++)             │
        System.out.println("Tento program obsahuje chybu!");
    }
  }
}
```

Nelze znovu deklarovat proměnnou `pocet`, protože je již deklarovaná

Pokud znáte jazyk C/C++, pak nejspíše víte, že na názvy přidělované proměnným deklarovaným ve vnitřním oboru se nevztahuje žádné omezení. V jazyce C/C++ je tedy deklarace proměnné `pocet` uvnitř bloku vnějšího cyklu `for` zcela platná a překryje vnější proměnnou. Návrháři jazyka Java cítili, že toto skrývání názvů by mohlo snadno vést k programovacím chybám, a proto jej zakázali.

Operátory

Java poskytuje bohaté prostředí operátorů. *Operátor* je symbol, který říká kompilátoru, aby provedl určitou matematickou nebo logickou manipulaci. Java obsahuje čtyři obecné třídy operátorů: aritmetické, bitové, relační a logické. Java dále definuje několik dalších operátorů, jež obstarávají určité speciální situace. V této kapitole prozkoumáme aritmetické, relační a logické operátory. Dále prostudujeme operátor přiřazení. Na bitové a speciální operátory se podíváme později.

Aritmetické operátory

Jazyk Java definuje následující aritmetické operátory:

Operátor	Význam
+	Sčítání (také unární plus)
–	Odčítání (také unární minus)
*	Násobení
/	Dělení
%	Modulo (zbytek po dělení)
++	Inkrement
––	Dekrement

Všechny operátory +, –, * a / fungují v Javě stejným způsobem jako v jakémkoli jiném počítačovém jazyce (nebo v algebře). Lze je aplikovat na libovolný vestavěný číselný datový typ. Dále je lze používat na objekty typu `char`.

Ačkoliv činnosti aritmetických operátorů jsou všem čtenářům dobře známé, zaslouží si několik zvláštních situací krátké vysvětlení. Pamatujte si, že při aplikování operátoru / na celé číslo se jakýkoliv zbytek ořízne. Například `10/3` se v celočíselném dělení bude rovnat `3`. Zbytek tohoto dělení můžete získat pomocí operátoru modulo (%). V Javě funguje podobně jako v ostatních jazycích: vrací zbytek po celočíselném dělení. Například `10 % 3` je `1`. V Javě lze operátor % aplikovat na celočíselné typy i na typy s plovoucí desetinnou čárkou. Proto je `10.0 % 3.0` také `1`. Následující program demonstruje operátor modulo.

```
// Demonstruje operátor %.
class UkazkaModulo {
  public static void main(String args[]) {
    int ivysledek, izbytek;
    double dvysledek, dzbytek;

    ivysledek = 10 / 3;
    izbytek = 10 % 3;

    dvysledek = 10.0 / 3.0;
    dzbytek = 10.0 % 3.0;

    System.out.println("Výsledek a zbytek pro 10 / 3: " +
                       ivysledek + " " + izbytek);
    System.out.println("Výsledek a zbytek pro 10.0 / 3.0: " +
                       dvysledek + " " + dzbytek);
  }
}
```

Výstup z tohoto programu vypadá následovně:

```
Výsledek a zbytek pro 10 / 3: 3 1
Výsledek a zbytek pro 10.0 / 3.0: 3.3333333333333335 1.0
```

Jak vidíte, operátor % dává zbytek `1` pro celočíselné operace i pro operace s plovoucí desetinnou čárkou.

Inkrementace a dekrementace

V kapitole 1 jsme si řekli, že operátory inkrementace a dekrementace v Javě jsou ++ a --. Jak uvidíte, mají několik speciálních vlastností, díky nimž jsou docela zajímavé. Začněme prozkoumáním, co přesně tyto operátory dělají.

Operátor inkrementace přičítá 1 ke svému operandu a operátor dekrementace odečítá 1. Proto

```
x = x + 1;
```

je totéž jako

```
x++;
```

a

```
x = x - 1;
```

je stejné jako

```
x--;
```

Oba operátory inkrementace a dekrementace mohou svůj operand buď předcházet (prefix), nebo následovat (postfix). Například

```
x = x + 1;
```

lze zapsat jako

```
++x; // prefixový tvar
```

nebo jako

```
x++; // postfixový tvar
```

Ve výše uvedeném příkladu není žádný rozdíl mezi tím, zda se inkrementace aplikuje jako prefix nebo jako postfix. Když se ale inkrementace či dekrementace použije jako součást většího výrazu, je zde podstatný rozdíl. Pokud totiž operátor inkrementace či dekrementace předchází svůj operand, Java provede uvedenou operaci před získáním hodnoty operandu pro zbytek výrazu. Pokud operátor následuje svůj operand, Java získá hodnotu operandu ještě před jeho inkrementací či dekrementací. Uvažte toto:

```
x = 10;
y = ++x;
```

V tomto případě bude proměnná y nastavená na hodnotu 11. Pokud ale kód přepíšeme

```
x = 10;
y = x++;
```

bude mít proměnná y hodnotu 10. V obou případech je proměnná x nastavena na hodnotu 11. Rozdíl je jen v tom, kdy k tomu dojde. Možnost určovat, kdy se má operace inkrementace či dekrementace provést, totiž přináší podstatné výhody.