Datové typy, hodnota null, statické metody a proměnné, konstanty, řídicí struktury

August 28, 2014

1

1.1 Úvod k datovým typům v Javě

Cíl Naučit se pracovat s primitivními a objektovými datovými typy v Javě, vymezit to vůči obecně známým principům (např. z Pascalu)

Předpoklady Znát základní datové typy (číselné, logické, znakové) - např. z Pascalu, C či odjinud

- Primitivní vs. objektové typy
- Kategorie primitivních typů: integrální, boolean, čísla s pohyblivou řádovou čárkou
- Pole: deklarace, vytvoření, naplnění, přístup k prvkům, rozsah indexů

1.2 Primitivní vs. objektové datové typy - opakování

Java striktně rozlišuje mezi hodnotami

- **primitivních datových typů** (čísla, logické hodnoty, znaky) a
- **objektových typů** (řetězce a všechny uživatelem definované [tj. vlastní] typy-třídy)

Základní rozdíl je v práci s proměnnými:

- proměnné primitivních typů *přímo obsahují danou hodnotu*, zatímco
- proměnné objektových typů obsahují pouze *odkaz na příslušný objekt*

Důsledek → **dvě objektové proměnné** mohou nést odkaz na **tentýž objekt**

1.3 Přiřazení proměnné primitivního typu - opakování

Příklad:

```
double a = 1.23456;
double b = a;
a += 2;
// poté bude v a hodnota 3.23456,
// v b stále hodnota 1.23456
```

2 Předávání parametrů do metod

2.1 Předávání parametru primitivního typu

Při volání metod s parametry primitivního typu (čísla, boolean, char) postupuje Java vždy takto:

1. **vyhodnotí** výraz předávaný jako skutečný parametr, získá tím jeho hodnotu příslušného typu, který musí odpovídat typu uvedenému v deklaraci (hlavičce) metody
2. **zavolá** metodu, vytvoří v ní (tzn. na zásobníku daného vlákna/procesu) lokální proměnnou
3. do ní **zkopíruje** předanou hodnotu
4. **počítá** s touto hodnotou (ale neměla by ji měnit, i když to fyzicky je možné - neuvedeme-li final před deklarací parametru!)
5. po ukončení metody lokální proměnná **mizí** i s případnými změnami

2.2 Vracení výsledku

Chceme-li předat výsledek "nahoru" (volajícímu kódu), pak buďto:

- **návratovou hodnotou** (přes return - běžné a bezproblémové)
- modifikací **atributu** objektu, na němž byla metoda zavolána (nebezpečné, protože metoda má pak tzv. vedlejší efekt, t.j. mění objekt, na němž je zavolána) nebo
- modifikací **obsahu objektu**, který byl předán jako jiný **parametr** (s podobnými či ještě horšími riziky!, oba poslední případy musí být popsány v dokumentaci metody)

V Javě tedy nelze to, co v Pascalu, tzn. předat do metody (tedy pascalsky procedury či funkce) ODKAZ na proměnnou typu např. celé číslo a pak s ní v metodě plně pracovat (čtení, modifikace)!!!

2.3 Předávání parametru primitivního typu: příklad

```
public class IntParamDemo {
   // tato metoda při zavolání navenek nic neprovede,
   // modifikace i je jen modifikací lokální proměnné,
   // která se při opuštění zapomene
   private static void addTwo(int i) {
      i += 2; // přičteme k i dvojku
   }
   public static void main(String[] args) {
      int value = 50;
      addTwo(value);
      System.out.println(value); // vypíše 50!!!
   }
}
```

2.4 Předávání parametru objektové typu

Při volání metod s parametry objektového typu (objekty předdefinovaných, cizích i vlastních tříd) postupuje Java vždy takto:

1. vezme **odkaz** na objekt předávaný do metody
2. **zavolá metodu**, vytvoří v ní (tzn. na zásobníku daného vlákna/procesu) lokální odkazovou (objektovou) proměnnou
3. do ní **zkopíruje** předaný odkaz (NEDUPLIKUJE OBJEKT, jen ODKAZ, objekt zůstává, jak byl)
4. **počítá** s předaným objektem
5. po ukončení metody lokální odkazová proměnná mizí, ale objekt metodou mohl být změněn a změny se pak navenek **projeví**

2.5 Odlišnosti Javy

V Javě tedy nelze to, co v Pascalu, tzn. předat do metody (tedy pascalsky procedury či funkce) hodnotu objektu, tedy tak, aby se obsah objektu **zduplikoval** a uvnitř metody pracovalo s tímto duplikátem!!! Když tu kopii (zřídka) potřebujeme, lze ji vyrobit ručně.

2.6 Přiřazení objektové proměnné – deklarace

```
public class Counter {
   private double value;
   public Counter(double v) {
      value = v;
   }
   public void add(double v) {
      value += v;
   }
   public void show() {
      System.out.println(value);
   }
}
```

2.7 Přiřazení objektové proměnné - použití

Napíšeme-li kód:

```
Counter c1 = new Counter(1.23456);
Counter c2 = c1;
c1.add(2);
c1.show();
c2.show();
```

Vypíše se:

```
3.23456
3.23456
```

3 Primitivní datové typy

3.1 Primitivní datové typy - deklarace

Proměnné těchto typů nesou **elementární**, z hlediska Javy **atomické**, **dále nestrukturované** hodnoty.Deklarace takové proměnné (kdekoli) způsobí:

1. rezervování příslušného paměťového prostoru (např. pro hodnotu `int` čtyři bajty)
2. zpřístupnění (pojmenování) tohoto prostoru identifikátorem proměnné
3. Místo, kde je paměťový prostor pro proměnnou rezervován, závisí na tom, zda se jedná o proměnnou lokální (tzn. buď parametr metody nebo proměnná v metodě deklarovaná), pak se vyhradí na zásobníku, nebo zda jde o proměnnou objektu či třídy – pak má místo v rámci paměťového prostoru objektu.

3.2 Primitivní datové typy - kategorie

V Javě existují tyto skupiny primitivních typů:

1. **integrální typy** (obdoba ordinálních typů v Pascalu) - zahrnují typy *celočíselné* (`byte`, `short`, `int` a `long`) a typ `char`;
2. typy **čísel s pohyblivou řádovou čárkou** (`float` a `double`)
3. typ **logických hodnot** (`boolean`).

3.3 Integrální typy - celočíselné

- V Javě jsou celá čísla vždy interpretována jako znaménková (tj. nelze změnit modifikátory jako v C++)
- "Základním" celočíselným typem je 32bitový `int` s rozsahem $-2147483648..2147483647$, dobrá volba, když dopředu neznáme přesně požadovaný rozsah celých čísel
- větší rozsah (64 bitů) má `long`, cca $+/-9*10^{18}$
- menší rozsah mají `short` (16 bitů), tj. $-32768..32767$ a
- `byte` (8 bitů), tj. $-128..127$

Pro celočíselné typy existují (stejně jako pro floating-point typy) konstanty - *minimální a maximální hodnoty* příslušného typu. Tyto konstanty mají název vždy `Typ.MIN_VALUE`, analogicky MAX.

3.4 Celočíselné typy - použití

- Nejpoužívanějším celočíselným typem je 32bitový `int` s rozsahem $-2147483648..2147483647$, dobrá volba, když dopředu neznáme přesně požadovaný rozsah celých čísel
- Jelikož Java je od počátku jazyk pro 32bitové (a "širší") architektury, nevede volba "uzšího" typu (short, byte) ve většině případů ke zrychlení běhu kódu, spíše naopak.

- Užší typy mají výhody nižší paměťové náročnosti v případě jejich struktur, tzn. polí těchto hodnot. Pak se může evt. projevit i zrychlení dané nutností číst méně bajtů dat.
- Tento faktor má ale smysl uvažovat až při počtu hodnot nejméně řádově milióny.

3.5 Zápis hodnot celočíselných typů od Javy 7

Počínaje Java 7 lze použít notaci s podtržítkem k oddělení řádů dlouhých čísel (většinou po tisících).

3.6 Integrální typy - "char"

`char` představuje jeden 16bitový znak v kódování UNICODEKonstanty typu `char` zapisujeme

- v apostrofech - 'a', 'Ř'
- pomocí escape-sekvencí - \n (konec řádku) \t (tabulátor)
- hexadecimálně - \u0040 (totéž, co 'a')
- oktalově - \127

3.7 Typ char - kódování

Java vnitřně kóduje znaky a řetězce v UNICODE, pro vstup a výstup je třeba použít některou za serializací (převodu) UNICODE na sekvence bajtů:

- např. vícebajtová kódování UNICODE: **UTF-8** a UTF-16
- osmibitová kódování ISO-8859-x, Windows-125x a pod.

Co se znaky národních abeced?

- Problém může nastat při interpretaci kódování znaků národních abeced uvedených přímo ve zdrojovém textu programu.
- Ve zdrojovém textu správně napsaného javového vícejazyčného programu by žádné národní znaky VŮBEC neměly vyskytovat.
- Běžně se umisťují do speciálních souborů tzv. *zdrojů* (v Javě objekty třídy `java.util.ResourceBundle`).

3.8 Čísla s pohyblivou řádovou čárkou

Kódována podle ANSI/IEEE 754-1985

- `float` - 32 bitů
- `double` - 64 bitů

Možné zápisy literálů typu `float` (klasický i semilogaritmický tvar) - povšimněte si "f" nebo "F" za číslem - je u float nutné!:`float f = -.777f, g = 0.123f, h = -4e6F, 1.2E-15f;double`: tentýž zápis, ovšem bez "f" za konstantou a s větší povolenou přesností a rozsahem

3.9 Vestavěné konstanty s pohyblivou řádovou čárkou

- Kladné "nekonečno": `Float.POSITIVE_INFINITY` , podobně záporné: `Float.NEGATIVE`...
- totéž pro `Double`
- Obdobně existují pro oba typy konstanty uvádějící rozlišení (nejmenší uložitelnou absolutní hodnotu různou od 0) daného typu - `MIN_VALUE` , podobně pro `MAX_VALUE`...
- Konstanta `NaN` - *Not A Number*

3.10 Typ logických hodnot - boolean

- Přípustné hodnoty jsou `false` a `true`.
- Na rozdíl od Pascalu na nich *není* definováno uspořádání, nelze je porovnávat pomocí `<`, `<=`, `>=`, `>`

3.11 Typ void

- Význam podobný jako v C/C++.
- Není v pravém slova smyslu datovým typem, nemá žádné hodnoty.
- Označuje "prázdný" typ pro sdělení, že určitá metoda *nevrací žádný výsledek.*

3.12 Všechno, co jste chtěli vědět o primitivních datových typech...

...najdete na Oracle: The Java Tutorial: Primitive Data Types (`http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`).

4 Statické proměnné a metody

4.1 Proměnné a metody třídy - statické

Dosud jsme zmiňovali **proměnné a metodyobjektu**.Lze deklarovat také metody a proměnné patřící *celé třídě*, tj. skupině všech objektů daného typu – statická proměnná existuje pro jednu *třídu* jen jednou!Takové metody a proměnné nazýváme **statické** a označujeme v deklaraci modifikátorem `static`

4.2 Příklad statické proměnné a metody (1)

Představme si, že si budeme pamatovat, kolik lidí se nám během chodu programu vytvořilo a vypisovat tento počet.Budeme tedy potřebovat do třídy *Person* doplnit:

- jednu proměnnou `peopleCount` společnou pro celou třídu `Person` - každý člověk ji při svém vzniku zvýší o jedna.
- jednu metodu `howManyPeople`, která vrátí počet dosud vytvořených lidí.

4.3 Příklad statické proměnné a metody (2)

```
public class Person {
   private String name;
   private int age;
   private static int peopleCount = 0;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
      peopleCount++;
   }
   public static int howManyPeople() {
      return peopleCount;
   }
}
```

Pozn: Všimněte si v obou případech modifikátoru/klíčového slova *static*.

5 Konstanty v Javě, tj. statické nemodifikovatelné "proměnné"

5.1 Problém: V programu potřebujeme (většinou na více místech) tutéž konkrétní hodnotu.

- Takové hodnoty se také označují jako "magic numbers", protože nemusí být na první pohled zřejmé, proč je to zrovna právě ta hodnota.
- Konstantu je vhodné zavést i tehdy, když ji použijeme (zatím!) jen jednou.
- Nemusí pochopitelně jít jen o čísla, ale libovolné typy – znaky, řetězce, data, ale i jiné objektové typy.
- V ideálním případě nejsou v kódu programu mimo konstanty žádné jiné konkrétní hodnoty - s výjimkou 0, "", true/false, případně znaků či hodnoty 1. Vše ostatní by mělo být definováno v konstantách.

5.2 Příklad konstanty

```
public class Person {
   private static final int MAX_PEOPLE_COUNT = 100;

   private String name;
   private int age;
   private static int peopleCount;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
      peopleCount++;
   }
```

```
   public static int howManyPeople() {
      return peopleCount;
   }
   public boolean maxPeopleCountReached() {
      return peopleCount >= MAX_PEOPLE_COUNT;
   }
}
```

6 Příkazy a řídicí struktury v Javě

6.1 Příkazy v Javě

V Javě máme následující příkazy:

- Přiřazovací příkaz = a jeho modifikace (kombinované operátory jako je += apod.)
- Řízení toku programu (větvení, cykly)
- Volání metody
- Návrat z metody - příkaz **return**
- Příkaz je ukončen středníkem **;**
- v Pascalu středník příkazy *odděluje*, v Javě (C/C++) *ukončuje*

7 Přiřazení

7.1 Přiřazení v Javě

- Operátor přiřazení = (assignment)
- na levé straně musí být proměnná
- na pravé straně výraz *přiřaditelný* (assignable) do této proměnné
- Rozlišujeme přiřazení primitivních hodnot a odkazů na objekty

7.2 Přiřazení primitivní hodnoty

- Na pravé straně je výraz vracející hodnotu primitivního typu:
- číslo, logická hodnotu, znak (ale ne např. řetězec)
- Na levé straně je proměnná téhož nebo širšího typu jako přiřazovaná hodnota:
- např. `int` lze přiřadit do `long`
- Při zužujícím přiřazení se také provede konverze, ale může dojít ke ztrátě informace:
- např. `int` → `short`
- Přiřazením primitivní hodnoty se hodnota zduplikuje ("opíše") do proměnné na levé straně.

7.3 Přiřazení odkazu na objekt

Konstrukci `=` lze použít i pro přiřazení do objektové proměnné:`Person z1 = new Person();`Co to udělalo?

1. vytvořilo nový objekt typu `Person` (`new Person()`)
2. přiřadilo jej do proměnné `z1` typu `Person`

Nyní můžeme *odkaz* na tentýž vytvořený objekt znovu přiřadit - do `z2`:`Person z2 = z1;`Proměnné `z1` a `z2` ukazují nyní na stejný objekt typu osoba!!!Proměnné objektového typu obsahují *odkazy* (reference) na objekty, ne objekty samotné!!!

8 Volání metod a návrat z nich

8.1 Volání metody

Metoda objektu je vlastně procedura/funkce, která realizuje svou činnost primárně s proměnnými objektu.Volání metody určitého objektu realizujeme:***identifikaceObjektu.názevMetody(skutečné parametry)***

- **identifikaceObjektu**, jehož metodu voláme
- **.** (tečka)
- **názevMetody**, jíž nad daným objektem voláme
- v závorách uvedeme *skutečné parametry* volání (záv. může být prázdná, nejsou-li parametry)

8.2 Návrat z metody

Návrat z metody se děje:

1. Buďto automaticky posledním příkazem v těle metody
2. nebo explicitně příkazem return návratováhodnota

způsobí ukončení provádění těla metody a návrat, přičemž může být specifikována návratová hodnotatyp skutečné návratové hodnoty musí korespondovat s deklarovaným typem návratové hodnoty

9 Řízení toku uvnitř metod - větvení, cykly

9.1 Podmíněný příkaz

Podmíněný příkaz neboli *neúplné větvení*`if (logický výraz) příkaz`platí-li logický výraz (má hodnoty true), provede se příkaz

9.2 Větvení

Příkaz *úplného větvení*if - else

```
if (logický výraz)
   příkaz1
else
 příkaz2
```

platí-li `logický výraz` (má hodnoty `true`), provede se `příkaz1` neplatí-li, provede se `příkaz2` Větev else se **nemusí uvádět**

9.3 Cyklus s podmínkou na začátku

Tělo cyklu se provádí tak dlouho, **dokud** platí podmínkaobdoba while v Pascalu, C a dalšíchv těle cyklu je jeden jednoduchý příkaz ...

```
while (podmínka)
 příkaz;
```

... nebo příkaz složený

```
while (podmínka) {
   příkaz1;
   příkaz2;
   příkaz3;
   ...
}
```

Tělo cyklu se nemusí provést ani jednou - pokud už hned na začátku podmínka neplatí

9.4 Doporučení k psaní cyklů/větvení

Větvení, cykly: doporučuji vždy psát se **složeným příkazem v těle** (tj. se složenými závorkami)!!! Jinak hrozí, že se v těle větvení/cyklu z neopatrnosti při editaci objeví něco jiného, než chceme, např.:

```
while (i < a.length)
 System.out.println(a[i]); i++;
```

Provede v cyklu jen ten výpis, inkrementaci ne a program se zacyklí. Pišme proto vždy takto:

```
while (i < a.length) {
    System.out.println(a[i]); i++;
}
```

9.5 Poznámka k úpravě

Pišme ale ještě raději takto:

```
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```

9.6 Příklad použití "while" cyklu

Dokud nejsou přečteny všechny vstupní argumenty – vč. toho případu, kdy není ani jeden:

```
int i = 0;
while (i < args.length) {
   //"přečti argument args[i]"
   i++;
}
```

Dalším příkladem je použití while pro realizaci celočíselného dělení se zbytkem:Příklad: Celočíselné dělení se zbytkem (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/DeleniOdcitanim.java`)

9.7 Cyklus s podmínkou na konci

Tělo se provádí **dokud** platí podmínka (vždy aspoň jednou)obdoba repeat v Pascalu (podmínka je ovšem *interpretována opačně*)Relativně málo používaný - je méně přehledný než whileSyntaxe:

```
do {
   příkaz1;
   příkaz2;
   příkaz3;
   ...
} while (podmínka);
```

9.8 Příklad použití "do-while" cyklu

Dokud není z klávesnice načtena požadovaná hodnota:

```
String vstup = "";
float number;
boolean isOK;
// create a reader from standard input
BufferedReader in = new BufferReader(new InputStream(System.in));
// until a valid number is given, try to read it
do {
   String input = in.readLine();
   try {
      number = Float.parseFloat(input);
      isOK = true;
   } catch (NumberFormatException nfe) {
      isOK = false;
   }
} while(!isOK);
System.out.println("We've got the number " + number);
```

Příklad: Načítej, dokud není zadáno číslo (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/DokudNeniZadano.java`)

9.9 Cyklus "for"

obecnější než for v Pascalu, podobně jako v C/C++. De-facto jde o rozšíření while, lze jím snadno nahradit.

```
for(počáteční op.; vstupní podm.; příkaz po každém průch.)
 příkaz;
```

anebo (obvyklejší, bezpečnější)

```
for (počáteční op.; vstupní podm.; příkaz po každém průch.) {
   příkaz1;
   příkaz2;
   příkaz3;
   ...
}
```

9.10 Příklad použití "for" cyklu

Provedení určité sekvence určitý počet krát

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

Vypíše na obrazovku deset řádků s čísly postupně 0 až 9

1. Příklad: Pět pozdravů (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/PetPozdravu.java`)
2. Příklad: Výpis prvků pole objektů "for" cyklem (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/PolozkyForCyklem.java`)

9.11 Doporučení k psaní for cyklů (1)

Používejte asymetrické intervaly (ostrá a neostrá nerovnost):

- podmínka daná počátečním přiřazením i = 0 a inkrementací i++ je *neostrou nerovností*, zatímco
- opakovací podmínka `i < 10` je *ostrou nerovností*: i už nesmí hodnoty 10 dosáhnout!

Vytvarujte se složitých příkazů v hlavičce (kulatých závorkách) for cyklu:

- je lepší to napsat podle situace před cyklus nebo až do jeho těla

9.12 Doporučení k psaní for cyklů (2)

Někteří autoři nedoporučují psát deklaraci řídicí proměnné přímo do závorek cyklu`for (int i = 0; ...`ale rozepsat takto:

```
int i;
for (i = 0; ...
```

potom je proměnná `i` přístupná ("viditelná") i mimo cyklus - za cyklem, což se však ne vždy hodí.

9.13 Vícecestné větvení "switch - case - default"

Obdoba pascalského select - case - else Větvení do více možností na základě ordinální hodnoty. Chová se ale spíše jako switch-case v C.

```
switch(výraz) {
   case hodnota1: prikaz1a;
                  prikaz1b;
                  prikaz1c;
                  ...
                  break;
   case hodnota2: prikaz2a;
                  prikaz2b;
                  ...
                  break;
   default:       prikazDa;
                  prikazDb;
                  ...
}
```

Je-li *výraz* roven některé z hodnot, provede se sekvence uvedená za příslušným case. Sekvenci obvykle ukončujeme příkazem break, který předá řízení ("skočí") na první příkaz za ukončovací závorkou příkazu switch.Příklad: Vícecestné větvení (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/VicecestneVetveni.java`)

9.14 Vnořené větvení

Větvení if - else můžeme samozřejmě vnořovat do sebe:

```
if(podmínka_vnější) {
   if(podmínka_vnitřní_1) {
      ...
   } else {
      ...
   }
} else {
   if(podmínka_vnitřní_2) {
      ...
   } else {
      ...
   }
}
```

9.15 Vnořené větvení (2)

Je možné "šetřit" a neuvádět složené závorky, v takovém případě se else vztahuje vždy k nejbližšímu neuzavřenému if, např. znovu předchozí příklad:

```
if(podmínka_vnější)
   if(podmínka_vnitřní1)
      ...
```

```
   else // vztahuje se k nejbližšímu if
        // s if (podmínka_vnitřní_1)
      ...
else // vztahuje se k prvnímu if,
     // protože je v tuto chvíli
     // nejbližší neuzavřené
   if (podmínka_vnitřní_2)
      ...
   else // vztahuje se k if (podmínka_vnitřní_2)
 ...
```

Tak jako u cyklů ani zde tento způsob zápisu (bez závorek) nelze v žádném případě doporučit!!!Příklad: Vnořené větvení (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/VnoreneVetveni.java`)

9.16 Řetězené "if - else if - else"

Někdy rozvíjíme pouze druhou (negativní) větev:

```
if (podmínka1) {
   ...
} else if (podmínka2) {
   ...
} else if (podmínka3) {
   ...
} else {
   ...
}
```

Neplatí-li podmínka1, testuje se podmínka2, neplatí-li, pak podmínka3..., neplatí-li žádná, provede se příkaz za posledním – samostatným – else. Opět je dobré všude psát složené závorky!!!Příklad: Řetězené if (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/VicecestneVetveniIf.java`)

9.17 Příkazy "break"

Realizuje "násilné" ukončení průchodu cyklem nebo větvením switch Syntaxe použití `break` v **cyklu**:

```
int i = 0;
for (; i < a.length; i++) {
   if(a[i] == 0) {
      break; // skoci se za konec cyklu
   }
}
if (a[i] == 0) {
    System.out.println("Nasli jsme 0 na pozici "+i);
} else {
    System.out.println("0 v poli neni");
}
```

použití u `switch` jsme již viděli

9.18 Příkaz "continue"

Používá se v těle cyklu.Způsobí přeskočení zbylé části průchodu tělem cyklu

```
for (int i = 0; i < a.length; i++) {
   if (a[i] == 5)
      continue;
   System.out.println(i);
}
```

Výše uvedený příklad vypíše čísla `1`, `2`, `3`, `4`, `6`, `7`, `8`, `9`, nevypíše hodnotu `5`.Příklad: Řízení průchodu cyklem pomocí "break" a "continue" (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/BreakContinue.java`)

9.19 "break" a "continue" s návěštím

Umožní ještě jemnější řízení průchodu vnořenými cykly:

- pomocí návěští můžeme naznačit, který cyklus má být příkazem break přerušen nebo
- tělo kterého cyklu má být přeskočeno příkazem continue.

Příklad: Návěští (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/rizeni/Navesti.java`)

9.20 Doporučení k příkazům break a continue

Až na jisté typické obraty raději NEPOUŽÍVAT, ale jsou menším zlem než by bylo goto (kdyby v Javě existovalo...), protože nepředávají řízení dále než za konec struktury (cyklu, větvení). Toto však již neplatí pro break a continue na návěští! Poměrně často se používá break při sekvenčním vyhledávání prvku.