Pole. Operátory a výrazy. Abstraktní třídy.

October 16, 2013

1 Pole

1.1 Pole v Javě jako objekt

Pole v Javě je speciálním **objektem**Můžeme mít pole jak primitivních, tak objektových hodnot

- pole primitivních hodnot tyto **hodnoty obsahuje**
- pole objektů obsahuje **odkazy na objekty**

Kromě pole v Javě existují i jiné objekty na ukládání více hodnot - tzn. *kontejnery*, viz další přednášky.

1.2 Deklarace

- Syntaxe *deklarace*:
- `TypHodnoty[] jménoPole`
- na rozdíl od C/C++ nikdy neuvádíme při deklaraci počet prvků pole - ten je podstatný až při **vytvoření objektu pole**

1.3 Vytvoření

- Syntaxe *vytvoření objektu* pole: jako u jiného objektu - operátorem new:
- *jménoPole = new TypHodnoty[početprvků];* nebo vzniklé pole rovnou naplníme hodnotami/odkazy:
- `jménoPole = new TypHodnoty[] {prvek1, prvek2, ...};`

1.4 Přístup k prvkům

Syntaxe přístupu k prvkům *jménoPole[indexprvku]* používáme

- jak pro **přiřazení** prvku do pole:`jménoPole[indexprvku] = hodnota;`
- tak pro **čtení** hodnoty z pole:`proměnná = jménoPole[indexprvku];`

1.5 Příklad - deklarace, vytvoření

Pole je objekt, je třeba ho před použitím nejen **deklarovat**, ale i **vytvořit**:

```
Person[] lidi;
lidi = new Person[5];
```

Deklaraci a vytvoření lze zkombinovat na jeden řádek:

```
Person[] lidi = new Person[5];
```

Zkrácení je však pouze "syntaktickým cukrem", nemá žádný vliv na rychlost běhu, potřebu paměti atd.

1.6 Příklad - naplnění, použití

Nyní můžeme pole naplnit:

```
lidi[0] = new Person("Václav Klaus");
lidi[1] = new Person("Libuše Benešová");
```

```
lidi[0].writeInfo();
lidi[1].writeInfo();
```

- Nyní jsou v poli `lidi` naplněny první dva prvky odkazy na objekty.
- Zbylé prvky zůstaly naplněny prázdnými odkazy `null`.

1.7 Když deklarujeme, ale nevytvoříme

Pokud ovšem `pole` bude proměnnou objektu/třídy:

```
public class Pokus {
   static String[] pole;
   public static void main(String args[]) {
      pole[0] = "Neco";
   }
}
```

Překladač chybu neodhalí a po spuštění se objeví:

```
Exception in thread "main"
 java.lang.NullPointerException at Pokus.main(Pokus.java:4)
```

1.8 Když deklarujeme, ale nevytvoříme - v rámci metody

Pokud tuto chybu uděláme v rámci metody:

```
public class Pokus {
   public static void main(String args[]) {
      String[] pole;
      pole[0] = "Neco";
   }
}
```

překladač nás varuje:

```
Pokus.java:4: variable pole might not have been
 initialized pole[0] = "Neco"; ^ 1 error
```

1.9 Když deklarujeme, vytvoříme, ale nenaplníme

Co kdybychom pole deklarovali, vytvořili, ale nenaplnili příslušnými prvky:

```
Person[] lidi;
lidi = new Person[5];
lidi[0].writeInfo();
```

Toto by skončilo také s běhovou chybou *NullPointerException*:

- pole existuje, má pět prvků, ale první z nich je prázdný, nelze tudíž volat jeho metody (resp. vůbec používat jeho vlastnosti)!

2 Kopírování polí

2.1 Kopírování odkazu na pole

V Javě obecně přiřazení proměnné objektového typu vede pouze k **duplikaci odkazu**, **nikoli** celého odkazovaného **objektu**.Nejinak je tomu u polí, tj.:

```
Person[] lidi2;
lidi2 = lidi1;
```

V proměnné *lidi2* je nyní odkaz na stejné pole jako je v *lidi1*.

2.2 Kopírování obsahu pole

Zatímco, provedeme-li vytvoření nového pole a zavoláme metodu `System.arraycopy`, pak *lidi2* obsahuje kopii (duplikát) původního pole.

```
Person[] lidi2 = new Person[5];
System.arraycopy(lidi, 0, lidi2, 0, lidi.length);
```

Samozřejmě bychom mohli kopírovat prvky ručně, např. pomocí for cyklu, ale volání *System.arraycopy* je zaručeně nejrychlejší a přitom stále platformově nezávislou metodou, jak kopírovat pole. Také *arraycopy* však do cílového pole zduplikuje jen **odkazy na objekty**, nevytvoří kopie objektů!

3 Operátory a výrazy

3.1 Aritmetické

- + - * / % (zbytek po celočíselném dělení)
- Pozn: operátor dělení / je polymorfní, funguje pro celočíselné argumenty jako *celočíselný*, pro floating-point (float, double) jako "obyčejný".
- Na to je třeba dávat pozor: i když dvě (fakticky i typově) celá čísla nejsou dělitelná, jako podíl se nám vrátí vždy celé číslo (typově i hodnotově).
- Abychom dosáhli desetinného výsledku, musí být aspoň jeden z operandů typově s pohyblivou řádovou čárkou. Lze zařídit např. typovou konverzí `((double)a)/b` pro `int a, b`.

3.2 Logické - součiny

Pracují nad logickými (booleovskými) hodnotami (samozřejmě vč. výsledků porovnávání `<, >, ==`, atd.).*logické součiny* (AND):

- `&` (*nepodmíněný* - vždy se vyhodnotí oba operandy),
- `&&` (*podmíněný* - líné vyhodnocování (lazy evaluation) - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)

3.3 Logické - součty, negace

logické součty (OR):

- `|` (*nepodmíněný* - vždy se vyhodnotí oba operandy),
- `||` (*podmíněný* - líné vyhodnocování - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)

negace (NOT):

- `!`

3.4 Relační (porovnávací)

Tyto lze použít na porovnávání primitivních hodnot:

- `<` , `<=` , `>=` , `>`

Test na rovnost/nerovnost lze použít na porovnávání primitivních hodnot i objektů:

- `==` , `!=`

3.5 Upozornění - porovnávání objektů a čísel

- Pozor na porovnávání objektů: == vrací true jen při rovnosti odkazů, tj. jsou-li objekty *identické*. Rovnost *obsahu* (tedy "rovnocennost") objektů se zjišťuje voláním metody `o1.equals(o2)`
- Pozor na srovnávání floating-point čísel na rovnost: je třeba počítat s *chybami zaokrouhlení*.Místo porovnání na přesnou rovnost raději používejme jistou toleranci: `abs(expected-actual) < delta`.

3.6 Bitové operace

- součin `&`
- součet `|`
- exkluzivní součet (XOR) (znak "stříška")
- negace (bitwise-NOT) (znak "tilda") - obrátí bity argumentu a výsledek vrátí

3.7 Bitové posuny

- vlevo `<<` o stanovený počet bitů
- vpravo `>>` o stanovený počet bitů s respektováním znaménka
- vpravo `>>>` o stanovený počet bitů bez respektování znaménka

3.8 Operátor podmíněného výrazu ? :

Jediný *ternární operátor*, navíc polymorfní, pracuje nad různými typy 2. a 3. argumentu.Platí-li první operand (má hodnotu `true`) →

- výsledkem je hodnota druhého operandu
- jinak je výsledkem hodnota třetího operandu

Typ prvního operandu musí být `boolean`, typy druhého a třetího musí být přiřaditelné do výsledku.

3.9 Operátory typové konverze (přetypování)

- Podobně jako v C/C++
- Píše se *(typ)hodnota*, např. (Person)o, kde o byla proměnná deklarovaná jako Object.
- Pro objektové typy se ve skutečnosti *nejedná o žádnou konverzi* spojenou se změnou obsahu objektu, nýbrž pouze o *potvrzení* (tj. typovou kontrolu), že běhový typ objektu je požadovaného typu - např. (viz výše) že o je typu Person.
- Naproti tomu u primitivních typů se jedná o úpravu hodnoty - např. int přetypujeme na short a ořeže se tím rozsah.

3.10 Operátor zřetězení +

Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např. sekvence `int i = 1; System.out.println("variable i = " + i);` je v pořádku, s řetězcovou konstantou se spojí řetězcová podoba dalších argumentů (např. čísla).Pokud je argumentem zřetězení odkaz na objekt `o` :

- je-li `o == null`: použije se řetězec `"null"`
- je-li `o != null`: použije se hodnota vrácená metodou `o.toString()` (tu lze překrýt a dosáhnout tak očekávaného řetězcového výstupu)

3.11 Priority operátorů a vytváření výrazů

Motto: Pravidla priorit operátorů je dobré znát, ale ještě lepší je závorkovat, aby nedošlo k chybám a každý to přesně pochopil!

- nejvyšší prioritu má násobení, dělení, nejnižší přiřazení

- nízkou prioritu má ternární operátor booleovský výraz ? výraz, když true : výraz, když false
- rozhodně NEZNEUŽÍVEJME *přiřazení* ve smyslu, že jej současně použijeme jako *výraz*, tzn. pracujeme s jeho hodnotu!

4 Abstraktní třídy

4.1 Abstraktní třídy

I když Java disponuje rozhraními, někdy je vhodné určitou specifikaci implementovat pouze *částečně*:

Rozhraní Specifikace

Abstraktní třída Částečná implementace, typicky předek konkrétních tříd, plných implementací

Třída Implementace

4.2 Abstraktní třídy (2)

Abstraktní třída je označena klíčovým slovem abstract v hlavičce, např.:`public abstract class AbstractSearcher ...`Název začínající na Abstract není povinný ani nutný.Obvykle má alespoň jednu *abstraktní metodu*, deklarovanou např.:`public abstract int indexOf(double d);`Od abstraktní třídy *nelze vytvořit instanci*, nelze napsat např.:`Searcher ch = new AbstractSearcher(...);`

5 Reálný příklad použití abstraktní třídy

5.1 Rozhraní - abstraktní třída - neabstraktní třída

Viz demo `searching` pro BlueJ:

Rozhraní - specifikuje, co má prohledávač umět `Searcher`

Abstraktní třída - předek konkrétních plných implementací prohledávače `AbstractSearcher`

Konkrétní třída - plná implementace prohledávače `LinearSearcher`

5.2 Searcher

Rozhraní - specifikuje, co má prohledávač umět

```
public interface Searcher {

    /**Nastav do vyhledávače pole, kde se bude vyhledávat  */
    void set(double[] a);

    /** Zjisti, zda pole obsahuje číslo d */
    boolean contains(double d);
```

```
    /**Zjisti pozici, na níž je v poli číslo d.
       Není-li tam, vrať -1 */
    int indexOf(double d);
}
```

5.3 AbstractSearcher

Abstraktní třída - předek konkrétních plných implementací prohledávače

```
public abstract class AbstractSearcher implements Searcher {
    // třída rozhraní implementuje, ale ne úplně
    // úložiště prvků JE implementováno
    protected double[] array;
    // nastavení úložiště prvků JE implementováno
    public void set(double[] a) {
        array = a;
    }
    // rozhodnutí, zda prvek je přítomen na základě vyhledání jeho pozice
    public boolean contains(double d) {
        return indexOf(d) >= 0;
    }
    // samotné vyhledání prvku není implementováno
    public abstract int indexOf(double d);
}
```

5.4 LinearSearcher

Konkrétní třída - plná implementace prohledávače - tentokrát pomocí lineárního prohledání

```
public class LinearSearcher extends AbstractSearcher {
// doimplementuje se, co zbývá a to je metoda indexOf!
    public int indexOf(double d) {
        for(int i = 0; i < array.length; i++) {
            if(array[i] == d) {
                return i;
            }
        }
        return -1;
    }
}
```