
Kapitola 1. Přednáška 5 - řídicí struktury, datové typy

Obsah

Datové typy v Javě	1
Primitivní vs. objektové datové typy - opakování	1
Přiřazení proměnné primitivního typu - opakování	2
Přiřazení objektové proměnné - opakování	2
Primitivní datové typy	3
Integrální typy - celočíselné	3
Integrální typy - "char"	3
Typ <code>char</code> - kódování	4
Čísla s pohyblivou řádovou čárkou	4
Vestavěné konstanty s pohyblivou řádovou čárkou	4
Typ logických hodnot - boolean	5
Typ <code>void</code>	5
Pole v Javě	5
Pole (2)	6
Pole - co když deklarujeme, ale nevytvoříme?	6
Pole - co když deklarujeme, vytvoříme, ale nenaplníme?	7
Kopírování polí	7

Datové typy v Javě

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

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**

Přiřazení proměnné primitivního typu - opakování

- Příklad:

```
float a = 1.23456;  
float b = a;  
a += 2;
```

Přiřazení objektové proměnné - opakování

- Příklad, deklarujeme třídu

Cislo

takto:

```
public class Cislo {  
    private float hodnota;  
    public Cislo(float h) {  
        hodnota = h;  
    }  
    public void zvysO(float kolik) {  
        hodnota += kolik;  
    }  
    public void vypis() {  
        System.out.println(hodnota);  
    }  
}
```

- nyní ji použijeme:

```
Cislo c1 = new Cislo(1.23456);  
Cislo c2 = c1;  
c1.zvysO(2);  
c1.vypis();  
c2.vypis();
```

dostaneme:

```
3.23456
3.23456
```

Primitivní datové typy

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

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 **čísels** s **pohyblivou řádovou čárkou** (`float` a `double`)
3. typ **logických hodnot** (`boolean`).

Integrální typy - celočíselné

V Javě jsou celá čísla vždy interpretována jako znaménková

"Základním" celočíselným typem je 32bitový `int` s rozsahem -2^{31} až $2^{31}-1$

větší rozsah (64 bitů) má `long`, cca $\pm 9 \cdot 10^{18}$

menší rozsah mají

- `short` (16 bitů), tj. -32768..32767
- `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...` Viz např.

Minimální	a	maximální	hodnoty
-----------	---	-----------	---------

[<http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/hodnoty/MinMaxHodnoty.java>]

Integrální typy - "char"

`char` představuje jeden 16bitový znak v kódování UNICODE

Konstanty typu `char` zapisujeme

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

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.

Problém může nastat při interpretaci kódování znaků národních abeced přímo ve zdrojovém textu programu.

Ve zdroj. textu správně napsaného javového vícejazyčného programu by žádné národní znaky VŮBEC neměly vyskytovat.

Je vhodné umístit je do speciálních souborů tzv. *zdrojů* (v Javě objekty třídy `java.util.ResourceBundle`).

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" 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!, s větší povolenou přesností a rozsahem

Vestavěné konstanty s pohyblivou řádovou čárkou

Kladné a záporné nekonečno:

- `Float.POSITIVE_INFINITY` , totéž s `NEGATIVE...`
- totéž pro `Double`
- obdobně existují pro oba typy konstanty uvádějící rozlišení daného typu - `MIN_VALUE` , podobně s `MAX...`

Konstanta `NaN` - *Not A Number*

Viz také Minimální a maximální hodnoty
[<http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/hodnoty/MinMaxHodnoty.java>]

Typ logických hodnot - boolean

Přípustné hodnoty jsou `false` a `true`.

Na rozdíl od Pascalu na nich *není* definováno uspořádání.

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*.

Pole v Javě

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 dále

Syntaxe deklarace

`typhodnoty [] jménopole`

Poznámka

na rozdíl od C/C++ nikdy neuvádíme při deklaraci počet prvků pole - ten je podstatný až při **vytvoření objektu pole**

Syntaxe přístupu k prvkům `jmenoPole[indexPrvku]` Používáme

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

Syntaxe vytvoření *objektu* pole: jako u jiného objektu - voláním konstruktoru:

`jmenoPole = new TypHodnoty[početPrvků];` nebo vzniklé pole rovnou naplníme hodnotami/odkazy

`jmenoPole = new TypHodnoty[] {prvek1, prvek2, ...};`

Pole (2)

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

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

Clovek

Nyní můžeme pole naplnit:

```
lidi[0] = new Clovek("Václav Klaus", Clovek.MUZ);  
lidi[1] = new Clovek("Libuše Benešová", Clovek.ZENA);
```

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

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

Pole - co když deklarujeme, ale nevytvoříme?

Co kdybychom pole pouze deklarovali a nevytvořili:

```
Clovek[] lidi;  
lidi[0] = new Clovek("Václav Klaus", Clovek.MUZ);
```

Toto by skončilo s běhovou chybou "NullPointerException", pole neexistuje, nelze do něj tudíž vkládat prvky!

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
```

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)
```

Pole - co když deklarujeme, vytvoříme, ale nenaplníme?

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

```
Clovek[] lidi;  
lidi = new Clovek[5];  
lidi[0].vypisInfo();
```

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)!

Kopírování polí

V Javě obecně přiřazení proměnné objektového typu vede pouze k **duplikaci odkazu**, **nikoli** celého odkazovaného **objektu**.

Nejinak je tomu u polí, tj.:

```
Clovek[] lidi2;  
lidi2 = lidi1;
```

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

Zatímco, provedeme-li vytvoření nového pole + `arraycopy`, pak `lidi2` obsahuje duplikát/klon/kopii původního pole.

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

viz též Dokumentace API třídy "System" [<http://java.sun.com/j2se/1.4/docs/api/java/lang/System.html>]

Poznámka

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!