
Datové typy: primitivní, objektové, pole. Výrazy.

Obsah

Úvod k datovým typům v Javě	2
Primitivní vs. objektové datové typy - opakování	2
Přiřazení proměnné primitivního typu - opakování	2
Přiřazení objektové proměnné - opakování	2
Primitivní datové typy	3
Primitivní datové typy	3
Integrální typy - celočíselné	4
Integrální typy - "char"	5
Typ char  - kódování	5
Čísla s pohyblivou řádovou čárkou	6
Vestavěné konstanty s pohyblivou řádovou čárkou	6
Typ logických hodnot - <i>boolean</i>	7
Typ void 	7
Pole	7
Pole v Javě	7
Pole (2)	8
Pole - co když deklarujeme, ale nevytvoříme?	9
Pole - co když deklarujeme, vytvoříme, ale nenaplníme?	9
Kopírování polí	10
Operátory a výrazy	10
Aritmetické	11
Logické	11
Relační (porovnávací)	12
Bitové	12
Operátor podmíněného výrazu ? : 	13
Operátory typové konverze (přetypování)	13
Operátor zřetězení + 	14
Priority operátorů a vytváření výrazů	14
Porovnávání objektů	14
Relační (porovnávací)	15
Porovnávání objektů	15
Porovnávání objektů - příklad	16
Metoda hashCode	16
Metoda hashCode - příklad	17

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

- 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:

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

Přiřazení objektové proměnné - opakování

- Příklad,

deklarujeme

třídu

Counter



[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=Counter] takto:

```
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);  
    }  
}
```

- nyní ji použijeme:

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

dostaneme:

```
3.23456  
3.23456
```

Primitivní datové typy

Primitivní datové typy

Proměnné těchto typů nesou **elementární**, z hlediska Javy **atomické, dále nestrukturované** hodnoty.

Deklarace takové proměnné (kdekoli) způsobí:

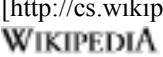
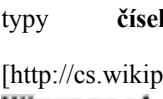
1. rezervování příslušného paměťového prostoru (např. pro hodnotu `int`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=int] čtyři bajty)
2. zpřístupnění (pojmenování) tohoto prostoru identifikátorem proměnné

V Javě existují tyto skupiny primitivních typů:

1. **integrální typy** (obdoba ordinálních typů v Pascalu) - zahrnují typy *celočíselné* (byte

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=byte],
short

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=short], int
long

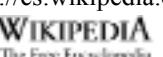
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=long]) a typ char

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=char];
2. typy čísel s pohyblivou řádovou čárkou (float

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=float] a double

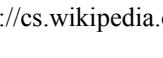
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=double])
3. typ logických hodnot (boolean

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=boolean]).

Integrální typy - celočíselné

V Javě jsou celá čísla vždy interpretována jako znaménková

"Základním" celočíselným typem je 32bitový int

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=int] s rozsahem -2 147 483
648

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=-2147483647]
648] až 2147483647

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=2147483647]

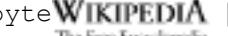
větší rozsah (64 bitů) má long

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=long], cca +/- 9*10^18

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=+-9*10^18]

menší rozsah mají

- short

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=short] (16 bitů), tj. -32768..32767
- byte

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=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



[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_typer:Search?search= 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 [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_typer:Search?search=char]

představuje jeden 16bitový znak v kódování UNICODE

Konstanty typu

char

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_typer:Search?search=char] zapisujeme

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

Typ char

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_typer:Search?search=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 uvedených 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

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_typer:Search?search=java.util.ResourceBundle]).

Čísla s pohyblivou řádovou čárkou

Kódována podle ANSI/IEEE 754-1985

- float 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=float] - 32 bitů
- double 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=double] - 64 bitů

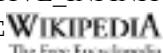
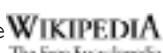
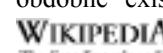
Možné zápisů literálů typu float 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=float] (klasický i semilogaritmický tvar) - povšimněte si "f" za číslem - je u float nutné!:

float f = -.777f, g = 0.123f, h = -4e6f, 1.2E-15f; 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=float f = -.777f, g = 0.123f, h = -4e6f, 1.2E-15f;]

double 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=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 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=Float.POSITIVE_INFINITY Y], totéž s NEGATIVE 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=NEGATIVE]...
- totéž pro Double 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=Double]
- obdobně existují pro oba typy konstanty uvádějící rozlišení daného typu - MIN_VALUE  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=MIN_VALUE], podobně s MAX 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=MAX]...

Konstanta NaN 
[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=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**  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=false] a **true**  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=true].

Na rozdíl od Pascalu na nich *není* definováno uspořádání, nelze je porovnávat pomocí <, >, <=, >=.

Typ **void**

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=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

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

typ hodnoty [] jménopole 
[[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=typodnoty \[\] jménopole](http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=typodnoty [] 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 *jménopole[indexprvku]* Používáme

- jak pro **přiřazení** prvku do pole: *jménopole[indexprvku] = hodnota;* 
[[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search= jménopole\[indexprvku\] = hodnota;](http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search= jménopole[indexprvku] = hodnota;)]
- tak pro **čtení** hodnoty z pole proměnná = *jménopole[indexprvku];* 
[[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search= proměnná = jménopole\[indexprvku\];](http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search= proměnná = jménopole[indexprvku];)]

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

jménopole = new typ[početprvků]; nebo vzniklé pole rovnou naplníme hodnotami/odkazy

jménopole = new typ[] {prvek1, prvek2, ...};

Pole (2)

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

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

Clovek 

[<http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search=Clovek>]

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 
[<http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search=lidi>] naplněny první dva prvky odkazy na objekty.
- Zbylé prvky zůstaly naplněny prázdnými odkazy null 
[<http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD%AD:Search?search=null>].

Pole - co když deklarujeme, ale nevytvoříme?

Co kdybychom pole pouze deklarovali a nevytvořili:

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

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  [http://cs.wikipedia.org/wiki/Special%CA%11n%C3%AD:Search?search=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:

```
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)!

Kopírování polí

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

Nejinak je tomu u polí, tj.:

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

V proměnné *lidi2* je nyní odkaz na stejné pole jako je *vlidi1*.

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

```
Person[] lidi2 = new Person[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 [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=for] cyklu, ale volání *System.arraycopy* je zaručeně nejrychlejší a přitom stále platformově nezávislou metodou, jak kopírovat pole.

Také *arraycopy* však do cílového pole zduplicuje jen **odkazy na objekty**, nevytvorí kopie objektů!

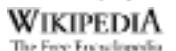
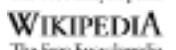
Operátory a výrazy

Cíl Zvládnout použití operátorů v Javě a naučit se sestavovat výrazy různých typů

Předpoklady Znát obecné principy syntaxe a vyhodnocování výrazů v pg. jazycích (např. Pascalu)

- Operátory v Javě: aritmetické, logické, relační, bitové
- Ternární operátor podmíněného výrazu
- Typové konverze
- Operátor zřetězení

Aritmetické

+ 	[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%2B],	-
- 	[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%2D],	*
/ 	[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%2F],	/
% 	[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%25] a %	
celočíselném dělení	[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%25] (zbytek po celočíselném dělení)	

Pozn: operátor dělení / je polymorfní, funguje pro celočíselné argumenty jako *celočíselný*, pro floating-point (float, double) jako "obyčejný".

Logické

Pracují nad logickými (booleovskými) hodnotami (samozřejmě vč. výsledků porovnávání <, >, ==, atd.).

logické součiny (AND):

- &  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%26] (*ne-podmíněný* - vždy se vyhodnotí oba operandy),
- &&  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%26%26] (*podmíněný* - líné vyhodnocování - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)

logické součty (OR):

- |  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%27] (*ne-podmíněný* - vždy se vyhodnotí oba operandy),
- ||  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%27%27] (*podmíněný* - líné vyhodnocování - druhý operand se vyhodnotí, jen nelze-li o výsledku rozhodnout z hodnoty prvního)

negace (NOT):

- !  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=!] [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=!=]

Relační (porovnávací)

Tyto lze použít na porovnávání primitivních hodnot:

- <  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=<], <=  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=<=], >=  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=>=], >  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=>]

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

- ==  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search==], !=  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search!=]

Upozornění:

- 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(Object o2)
- Pozor na srovnávání floating-points čísel na rovnost: je třeba počítat s chybami zaokrouhlení; místo porovnání na přesnou rovnost raději používejme jistou toleranci: abs(expected-actual) < delta

Bitové

Bitové:

- součin &  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=&]
- součet |  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=|]
- exkluzivní součet (XOR) ^ 

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=%] (znak "stříška")

- negace (bitwise-NOT)

~



[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=~] (znak "tilda") - obrátí bity argumentu a výsledek vrátí

Posuny:

- vlevo

<<WIKIPEDIA
The Free Encyclopedia

[http://cs.wikipedia.org/wiki/Specifikace_Alin%C3%A1k] o stanovený počet bitů

- vpravo

>> [WIKIPEDIA](#)
The Free Encyclopedia

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=>>] o stanovený počet bitů s respektováním znaménka

- vpravo

>>> **WIKIPEDIA**
The Free Encyclopedia

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=>>>] o stanovený počet bítů bez respektování znaménska

Dále viz např. Bitové operátory [<http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/operatory/Bitove.java>]

Operátor podmíněného výrazu ? : WIKIPEDIA The Free Encyclopedia

[<http://cs.wikipedia.org/wiki/Special%20Search?search=?:>]

Jediný *ternární operátor*, navíc polymorfní, pracuje nad různými typy 2. a 3. argumentu.

Platí-li první operand (má hodnotu true

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=true] ->

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

Typ prvního operandu musí být boolean **WIKIPEDIA**

[http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD_Search?search=boolean], typy druhého a třetího musí být přiřaditelné do výsledku.

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.

Operátor zřetězení +

[<http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=+>]

Výsledkem je vždy řetězec, ale argumenty mohou být i jiných typů, např.

sekvence `int i = 1; System.out.println("promenna i="+i);` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=int i = 1; System.out.println("promenna i="+i);] je v pořadku

s řetězcovou konstantou se spojí řetězcová podoba dalších argumentů (např. čísla).

Pokud je argumentem zřetězení odkaz na objekt o [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=o]->

• je-li o == null [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=o == null] -> použije se řetězec "null" [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search="null"]

• je-li o != null [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=o != null] -> použije se hodnota vrácená metodou o.toString() [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=o.toString()] (tu lze překrýt a dosáhnout tak očekávaného řetězcového výstupu)

Priority operátorů a vytváření výrazů

nejvyšší prioritu má násobení, dělení, nejnižší přiřazení

Porovnávání objektů

- Porovnávání primitivních hodnot a objektů je zásadně odlišné.
- U objektů lze kromě `==` a `!=` použít metodu `equals`.

Relační (porovnávací)

Na objekty nelze použít:

- `<`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= <], `<=`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= <=], `>`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= >], `>=`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= >=]

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

- `==`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= ==],
`!=`  [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search= !=]

Znovu podstatné upozornění:

- 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(Object o2)`

Porovnávání objektů

Použití `==`

- Porovnáme-li dva objekty (tzn. odkazy na objekty) prostřednictvím operátoru `==` dostaneme rovnost jen v případě, jedná-li se o dva odkazy na tentýž objekt - tj. dva *totožné* objekty.
- Jedná-li se o dva obsahově stejné objekty existující samostatně, pak `==` vrátí `false`.

Chceme-li (intuitivně) chápout rovnost objektů podle obsahu, tj.

- dva objekty jsou *rovné* (*rovnocenné*, nikoli *totožné*), mají-li stejný obsah, pak
- musíme pro danou třídu překrýt metodu `equals`, která musí vrátit `true`, právě když se *obsah* výchozího a srovnávaného objektu rovná.

- Fungování equals lze srovnat s porovnáváním dvou databázových záznamů podle primárního klíče.
- Nepřekryjeme-li equals, funguje původní equals přísným způsobem, tj. *rovné si budou jen totožné objekty*.

Porovnávání objektů - příklad

Příklad: objekt třídy Clovek nese informace o člověku. Dva objekty položíme stejné (rovnocenné), ne-sou-li stejná příjmení:

Obrázek 1. Dva lidí jsou stejní, mají-li stejná příjmení

```
public class Person implements Comparable {  
    private String firstname, surname;  
    public Person (String j, String p) {  
        firstname = j;  
        surname = p;  
    }  
    public boolean equals(Object o) {  
        if (o instanceof Person) {  
            Person c = (Person)o;  
            // dva lidé se (v našem případě) rovnají, mají-li stejná příjmení  
            return surname.equals(c.surname);  
        } else {  
            // porovnáváme-li osobu s objektem jiného typu, nikdy se nerovnají  
            return false;  
        }  
    }  
}
```

Méně agresivní verze by nemusela při porovnávání s jiným objektem než Person vyhodit výjimku, pouze vrátit false.

Metoda hashCode

Jakmile u třídy překryjeme metodu equals, měli bychom současně překrýt objektů i metodu hashCode:

- *hashCode* vrací celé číslo (int) „co nejlépe“ charakterizující obsah objektu, tj.
- pro dva stejné (*equals*) objekty *musí vždy vrátit stejnou hodnotu*.
- Pro dva obsahově různé objekty by *hashCode* naopak měl vracet *různé* hodnoty (ale není to stoprocentně nezbytné a ani nemůže být vždy splněno).

Metoda *hashCode* totiž nemůže vždy být prostá.

Metoda **hashCode** - příklad

V těle *hashCode* s oblibou „přehráváme“ (delegujeme) řešení na volání *hashCode* jednotlivých složek objektu - a to těch, které figurují v *equals*:

Obrázek 2. Třída Clovek s metodami *equals* a *hashCode*

```
public class Person implements Comparable {  
    private String firstname, surname;  
    public Person (String j, String p) {  
        firstname = j;  
        surname = p;  
    }  
    public boolean equals(Object o) {  
        if (o instanceof Person) {  
            Person c = (Person)o;  
            // dva lidé se (v našem případě) rovnají, mají-li stejná příjmení  
            return surname.equals(c.surname);  
        } else {  
            // porovnáváme-li osobu s objektem jiného typu, nikdy se nerovnají  
            return false;  
        }  
    }  
    public int hashCode() {  
        return surname.hashCode();  
    }  
}
```