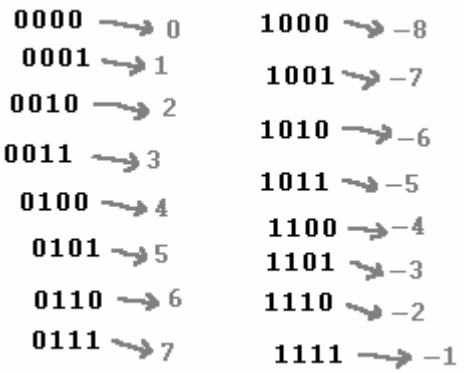
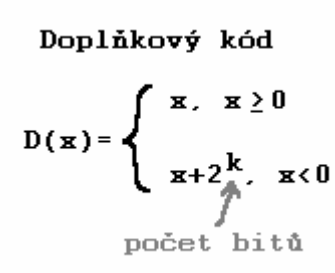
7 Celá čísla

Pro práci s celými čísly jsou v Javě typy byte, short, int a long. Všechny jsou *znaménkové* (připouštějí záporné hodnoty) a všechny používají *doplňkový kód*. Doplňkový kód definuje, jak jsou číselné hodnoty uloženy v paměti. Celočíselné typy se navzájem liší rozsahem přípustných hodnot. Přípustné hodnoty pro daný typ jsou dány počtem bajtů, které jsou potřeba pro uložení hodnoty tohoto typu. Např. typ byte je definován jako jednobajtový a jeho přípustné hodnoty jsou v rozmezí $< -128,127 >$.

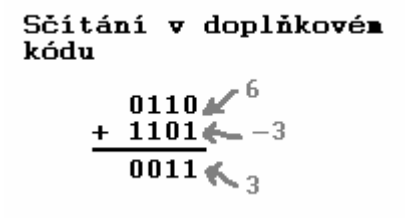
typ	počet bajtů	rozsah
byte	1	<−128,127>
short	2	<−32768,32767>
int	4	<−2147483648,2147483647>
long	8	<−9223372036854775808,9223372036854775807>

Pro objasnění principu doplňkového kódu si popíšeme, jak by vypadal čtyřbitový celočíselný typ. Přípustné hodnoty jsou určeny tím, že k uložení hodnoty tohoto typu máme k dispozici pouze čtyři bity. Čtyři bity připouštějí 16 kombinací nul a jedniček: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111. Každá tato kombinace bude obrazem jedné přípustné hodnoty. Kombinace, které mají v nejvyšším bitu (tj. prvním zleva) nulu, použijeme pro kladná čísla a nulu. Kombinace, které mají v nejvyšším bitu jedničku, budou zobrazovat čísla záporná. Z toho plyne, že interval přípustných hodnot bude $< -8,7 >$. Obecně při použití k bitů je interval přípustných hodnot $< -2^{k-1}, 2^{k-1} - 1 >$. Zbývá přiřadit každé přípustné hodnotě jednu ze 16 kombinací nul a jedniček. Nezáporným číslům přiřadíme jejich zápis ve dvojkové soustavě. Např. číslo 5 bude uloženo jako 0101. Záporným číslům přiřadíme dvojkový zápis jejich doplňku do hodnoty 2^k, kde k je počet bitů. Např. pro číslo −5 je tento doplněk −5 + 16 = 11. Hodnota −5 bude tudíž uložena jako 1011.

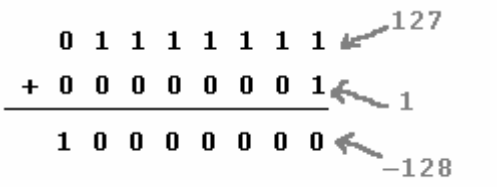




Doplňkový kód má několik příjemných vlastností. Např. sečtení dvou čísel v doplňkovém kódu provedeme tak, že sečteme jejich obrazy. Vznikne-li přenos z nejvyššího řádu, budeme jej ignorovat.
Stane-li se, že výsledek operace neleží v přípustném intervalu, nastává *přetečení* (angl. overflow): 0110 + 0011 = 1001, 6 + 3 = −7. V takovém případě má výsledek opačné znaménko než výsledek matematické operace.



V Javě nezpůsobí přetečení chybu, je však třeba s ním počítat. Např. pokud máme v proměnné typu byte hodnotu 127 a přičteme k ní 1, bude v proměnné −128.

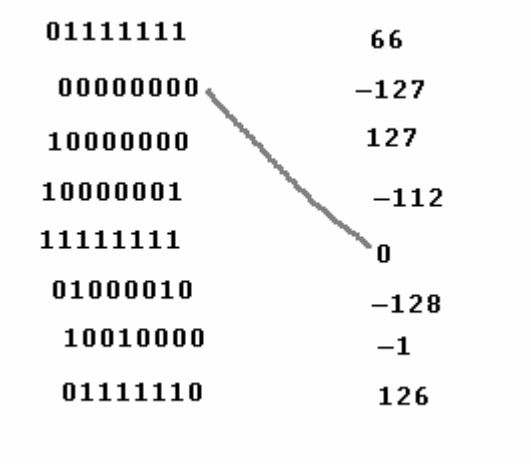


Zápis celočíselné hodnoty lze v Javě provést několika způsoby:

- desítkově: např. 123,
- šestnáctkově: zápis začíná znaky 0x nebo 0X, např. 0x2A, 0xff (lze používat malá i velká písmena),
- osmičkově: zápis začíná znakem 0, např. 012 je desítkově 10.

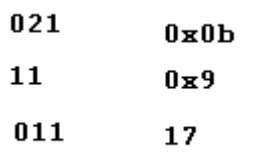
Úloha 1

Spojte binární obrazy na obrázku s hodnotami, které reprezentují.



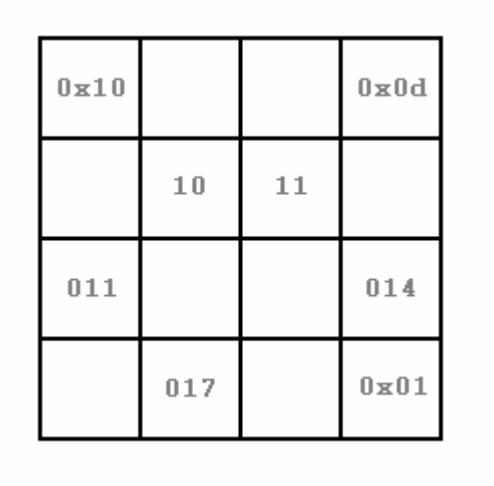
Úloha 2

Spojte na obrázku stejné hodnoty.



Úloha 3

Magický čtverec je čtverec o velikosti strany *n*, který obsahuje čísla $1..n^2$ a pro který platí, že součet čísel v každém řádku, sloupci i na diagonálách je stejný. Doplňte magický čtverec tak, aby byl součet v každém řádku, sloupci i na diagonálách 34.



Úloha 4

Doplňte věty.

1. Hodnota typu long je uložena v paměti na …… bitech.
2. Proměnná typu byte může nabývat …… různých hodnot.
3. Pokud sečteme dvě celočíselné kladné hodnoty a výsledek je záporný, nebo sečteme dvě celočíselné záporné hodnoty a výsledek je kladný, nastalo ……
4. Celočíselné hodnoty lze v Javě zapisovat třemi způsoby:
   ……

Řešení úlohy 1

66 je uloženo jako 01000010, −127 jako 10000001, 127 jako 01111111, −112 jako 10010000, −128 jako 10000000, −1 jako 11111111, 126 jako 01111110.

Řešení úlohy 2

Hodnota 021 je desítkově 17, 0x0b je desítkově 11, 011 je desítkově i šestnáctkově 9.

Řešení úlohy 3

První řádek: 0x10, 3, 2, 0x0d, druhý řádek: 5, 10, 11, 8, třetí řádek: 011, 6, 7, 014, čtvrtý řádek: 4, 017, 14, 0x01.

Řešení úlohy 4

1. Hodnota typu long je uložena v paměti na *64* bitech.
2. Proměnná typu byte může nabývat *256* různých hodnot.
3. Pokud sečteme dvě celočíselné kladné hodnoty a výsledek je záporný, nebo sečteme dvě celočíselné záporné hodnoty a výsledek je kladný, nastalo *přetečení*.
4. Celočíselné hodnoty lze v Javě zapisovat třemi způsoby: *desítkově, šestnáctkově a osmičkově*.

Otázky a odpovědi

Studentka: Mistře, proč se používá „složitý“ doplňkový kód, když by stačilo ukládat čísla jako dvojici znaménko a absolutní hodnota? Pro uložení znaménka nám stačí jeden bit. Kladné znaménko bychom uložili jako nulu a záporné jako jedničku. Pak by např. číslo 6 bylo ve čtyřbitovém kódu uloženo jako 0110 a číslo −6 jako 1110.

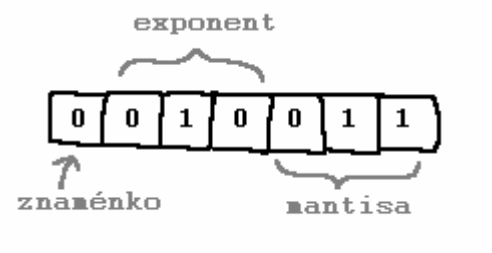
Java guru: Tato reprezentace celých čísel se také někdy používá. Říká se jí přímý kód. Oproti doplňkovému kódu má dvě nevýhody: nula má dva obrazy (0000 a 1000) a operace s čísly jsou malinko složitější. Proto se většinou dává přednost doplňkovému kódu.

Studentka: Pro uložení hodnoty typu byte potřebujeme 1 bajt a pro uložení hodnoty typu int potřebujeme 4 bajty. Znamená to, že proměnná typu int zabírá stejně místa jako čtyři proměnné typu byte, mistře?

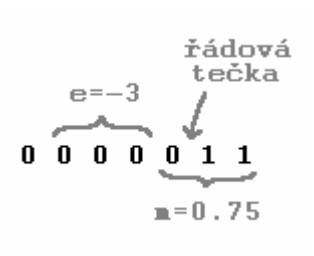
Java guru: Ne. Pro proměnné v zásobníku platí, že každá zabírá nejméně 4 bajty. Proměnná typu byte tedy zabírá v JVM stejně paměti jako proměnná typu short nebo int, a to 4 bajty. Proměnná typu long zabírá dvakrát více místa, tj. 8 bajtů.

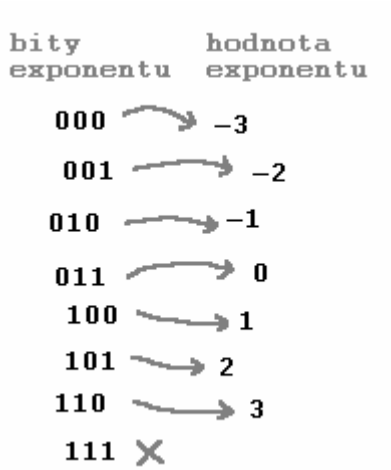
8 Reálná čísla

Pro práci s reálnými čísly má Java dva primitivní datové typy: float a double. Oba používají stejný způsob reprezentace: reálné číslo je uloženo jako trojice znaménko, mantisa a exponent. Označíme-li znaménko, mantisu a exponent po řadě *z*, *m* a *e*, pak bude tato trojice reprezentovat číslo $(-1)^z * m * 2^e$. Např. 0.25 je dvojkově 0.01, mantisa tedy bude 1_2 a exponent $-2_{10} = -10_2$. Číslo 4, uložené jako reálné číslo, bude mít mantisu 1 a exponent $2_{10} = 10_2$. V obou případech bude znaménko kladné, tj. $z = 0$.



Pro další přiblížení reprezentace reálných čísel si navrhneme 7-bitový datový typ, který bude analogií javovských typů float a double. První bit vyhradíme pro znaménko, další tři bity pro exponent a poslední tři bity pro mantisu. Budou-li všechny bity exponentu jedničkové, nepůjde o reprezentaci reálného čísla, ale o některou z nečíselných hodnot, které probereme později. Každá jiná kombinace nul a jedniček bude představovat reálné číslo. Abychom mohli ukládat čísla se zápornými exponenty, použijeme pro uložení exponentu aditivní kód +3, tj. před uložením k němu přičteme 3. Např. exponent −1 se uloží jako $2_{10} = 010_2$. Budou-li všechny bity exponentu nulové (tj. exponent je -3), bude řádová tečka v mantise za prvním bitem. Např. 0 000 011 je reprezentací čísla 0.00011_2. Pokud bude alespoň jeden bit exponentu nenulový, bude mantisa v tzv. normalizovaném tvaru, tj. z intervalu $<1,2)$. Každá normalizovaná mantisa má právě jednu jedničku před řádovou tečkou. Tato jednička nebude uložena (říkáme jí skrytá jednička, angl. hidden one). Bity mantisy budou reprezentovat pouze necelou část. Např. 0 011 100 je reprezentací čísla 1.1_2 (exponent je 0 a mantisa 1.1_2).





Podívejme se na to, jaká čísla můžeme v této reprezentaci uložit. Je zřejmé, že to nebudou všechna reálná čísla, protože exponent musí být z intervalu <−3,3>. Navíc iracionální čísla uložit nedokážeme. Zaměřme se tedy na čísla racionální. I s těmi je potíž, protože některá mají ve dvojkové soustavě nekonečný periodický rozvoj (např. $0.4_{10} = 0.\overline{0110}_2$). Taková čísla většinou uložíme přibližně. Výjimkou je např. $0.\bar{1}_2$, protože $0.\bar{1}_2 = 1_2$. Přibližně uložíme také čísla, jejichž mantisa má více bitů než čtyři (pro exponent z intervalu <−2,3>), příp. tři (pro exponent −3). Nedokážeme-li uložit číslo přesně, použijeme pro jeho reprezentaci nejbližší zobrazitelné číslo (např. 0.4 uložíme jako 0 001 100, což je $0.011_2 = 0.375_{10}$).

Nejmenší zobrazitelné kladné číslo má reprezentaci 0 000 001, což představuje hodnotu $0.00001_2 = (1/32)_{10}$. Pokud tuto hodnotu vydělíme dvěma, očekáváme nenulový výsledek. V našem typu ovšem výsledek nelze přesně zobrazit. Nejbližší zobrazitelná čísla jsou

0.00001_2 a 0 (má reprezentaci 0 000 000). Pro reprezentaci výsledku se vybere hodnota, která má v nejnižším bitu 0, tj. 0. Ačkoliv očekáváme nenulovou hodnotu, výsledkem bude 0. Tento jev označujeme jako *podtečení* (angl. underflow). Největší záporné číslo má reprezentaci 1 000 001, což je $-0.00001_2 = -(1/32)_{10}$. Vydělíme-li tuto hodnotu dvěma, nastane podtečení. Výsledek operace je v tomto případě záporný. Proto bude výsledkem tzv. *záporná nula* (má reprezentaci 1 000 000). Podtečení tedy nastává, pokud je výsledek aritmetické operace nenulový a přitom je tak blízký nule, že se v daném typu nedá zobrazit.

Největší zobrazitelné číslo má reprezentaci 0 110 111, což představuje číslo $1111_2 = 15_{10}$. Co dostaneme, když toto číslo vynásobíme dvěma? Výsledek nelze v našem typu zobrazit a přitom je natolik vzdálen hodnotě 15, že se tato hodnota nehodí pro jeho reprezentaci. Pro tuto situaci zavedeme speciální hodnotu *nekonečno* (její reprezentace bude 0 111 000). Výsledkem tedy bude hodnota nekonečno (kladné nekonečno).

Nejmenší zobrazitelné číslo je −15 (má reprezentaci 1 110 111). Pokud tuto hodnotu vynásobíme dvěma, výsledek bude mnohem menší než nejmenší zobrazitelné číslo. Pro tuto situaci zavedeme *záporné nekonečno* (bude mít reprezentaci 1 111 000). S nekonečny je možné provádět aritmetické operace, málokdy je však výsledkem něco jiného než nekonečno. Např. pokud vydělíme nekonečno dvěma, výsledkem bude nekonečno. Kromě kladného a záporného nekonečna zavedeme ještě jednu speciální hodnotu. Označíme ji *NaN* (z anglického Not-a-Number). Bude znamenat, že výsledek aritmetické operace nelze blíže určit. NaN dostaneme např. sečteme-li kladné a záporné nekonečno. Jeho reprezentace bude 0 111 100.

Vraťme se k javovským typům float a double. Typ float používá 32 bitů: 1 bit zabírá znaménko, 8 bitů exponent a 23 bitů mantisa. Pro uložení exponentu se používá aditivní kód +127. V tomto kódu je číslo *x* uloženo jako binární reprezentace čísla *x* + 127. Např. číslo -1 bude uloženo jako 01111110. Typ double používá 64 bitů: 1 bit pro znaménko, 11 bitů pro exponent a 52 bitů pro mantisu. Exponent je uložen v aditivním kódu +2047. V obou případech je formát čísla podle normy IEEE 754.

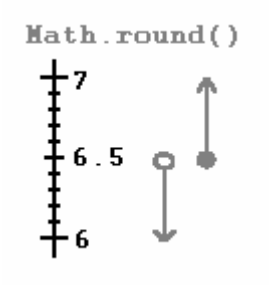
Pro reprezentaci reálných hodnot používáme obvykle typ double. Hodnoty typu double zapisujeme pomocí desetinné tečky.

```
double d1 = 1.43;
double d2 = -2.0;
```

Chceme-li použít typ float, je nutné za hodnotu připojit malé či velké písmeno f, kterým řekneme, že jde o hodnotu typu float:

```
float f1 = 2.35f;
float f2 = -3.67F;
```

Zaokrouhlování reálných čísel na celá provádíme pomocí Math.floor(), Math.ceil() a Math.round(). Math.floor() provádí zaokrouhlení dolu, Math.ceil() zaokrouhlení nahoru a Math.round() běžné zaokrouhlení, tj. k hodnotě přičte 0.5 a výsledek zaokrouhlí dolu.




```
double d = 9.5;
double d1 = Math.floor( d );  // v d1 bude 9
double d2 = Math.ceil( d );   // v d2 bude 10
double d3 = Math.round( d );  // v d3 bude 10
```

Zaokrouhlovat můžeme i záporná čísla.

```
double d = -9.5;
double d1 = Math.floor( d );  // v d1 bude -10
double d2 = Math.ceil( d );   // v d2 bude -9
double d3 = Math.round( d );  // v d3 bude -9
```

Hodnota π je v Javě dostupná jako Math.PI a Eulerova konstanta *e* jako Math.E. Obě hodnoty jsou typu double.

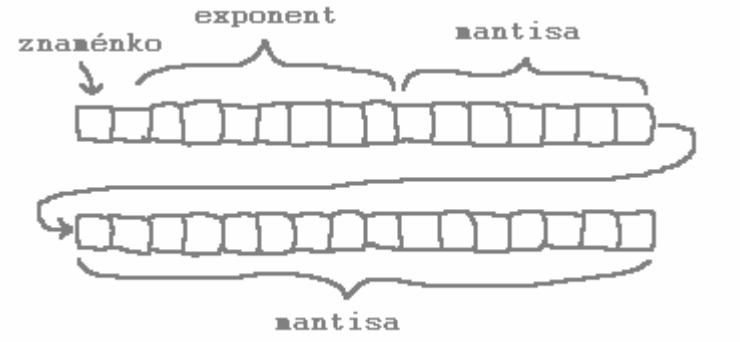
```
double r = 4.5;
double obvod = 2 * Math.PI * r;
System.out.println( obvod );
```

K výpočtu druhé odmocniny slouží Math.sqrt(). Použití si ukážeme na příkladu výpočtu délky přepony v pravoúhlém trojúhelníku:

```
double a = 1.2, b = 2.8;
double c = Math.sqrt( a * a + b * b );
System.out.println( c );
```

Úloha 1

Zapište bitovou reprezentaci nejmenšího kladného čísla zobrazitelného v typu float.



Úloha 2

Vytiskněte Ludolfovo číslo a základ přirozených logaritmů.

```
System.out.println(            );
System.out.println(            );
```

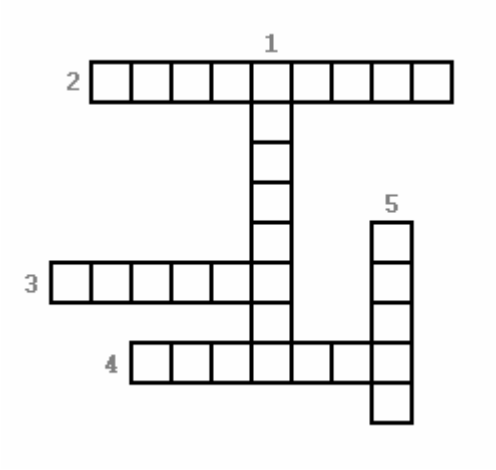
Úloha 3

Doplňte věty.

1. Každá hodnota typu double je uložena na …… bajtech.
2. Pro zaokrouhlení hodnoty typu double slouží ……
3. Druhou odmocninu počítáme pomocí ……
4. Je-li očekávaný výsledek aritmetické operace záporný a při aritmetické operaci dojde k podtečení, bude výsledkem ……

Úloha 4

Doplňte křížovku: 1 – jedna část obrazu reálného čísla, 2 – výsledek aritmetické operace je nenulový, ale pro jeho zobrazení je použita nula, 3 – osmibajtový reálný typ, 4 – jedna část obrazu reálného čísla, 5 – čtyřbajtový reálný typ.



Řešení úlohy 1

Nejmenší kladné číslo má reprezentaci 0 (znaménko), 00000000 (exponent) a 00000000000000000000001 (mantisa). Exponent je −127 a mantisa 2^{-22}. Jde tedy o reprezentaci čísla 2^{-149}.

Řešení úlohy 2

```
System.out.println( Math.PI );
System.out.println( Math.E );
```

Řešení úlohy 3

1. Každá hodnota typu double je uložena na *osmi* bajtech.
2. Pro zaokrouhlení hodnoty typu double slouží *Math.round()*.
3. Druhou odmocninu počítáme pomocí *Math.sqrt()*.
4. Je-li očekávaný výsledek aritmetické operace záporný a při aritmetické operaci dojde k podtečení, bude výsledkem *záporná nula*.

Řešení úlohy 4

1 – exponent, 2 – podtečení, 3 – double, 4 – mantisa, 5 – float.

Otázky a odpovědi

Student: Mistře, proč musím za hodnotou psát f, jestliže ji chci přiřadit do proměnné typu float?

Java guru: Pokud tam f nenapíšeš, překladač hodnotu uloží jako double. Bude tedy uložena na 64 bitech. Když se ji pak pokusíš přiřadit do proměnné typu float, která zabírá 32 bitů, překladač to odmítne, protože by při přiřazení mohlo dojít ke ztrátě informace. Písmenem f překladači sděluješ, že má s hodnotou zacházet jako s hodnotou typu float.
Student: Mistře, mám pro reprezentaci reálných čísel používat typ float a psát za každou hodnotou f nebo mám používat typ double? Vím, že typ double nabízí větší rozsah i přesnost než typ float, na druhou stranu však každá proměnná typu double zabírá dvakrát více paměti než proměnná typu float.

Java guru: Máš pravdu, typ double je paměťově náročnější. Obvykle se mu však dává přednost a to právě pro jeho větší rozsah a přesnost. Také většina funkcí v javovském API pracuje s typem double. Nemáš-li tedy nějaký zvláštní důvod používat float, používej double.

Student: Vím, že některá čísla, např. 0.1 a 0.4, mají ve dvojkové soustavě nekonečný peoridický rozvoj a tudíž nemohou být v počítači uložena přesně. Jak je potom možné, že když takovou hodnotu uložím do proměnné a hodnotu proměnné vytisknu, zobrazí se přesně? Např. kód

```
float f = 0.1F;
double d = 0.4;
System.out.println( f );
System.out.println( d );
```

vytiskne 0.1 a 0.4.

Java guru: To je dobrá otázka. Čísla 0.1 a 0.4 jsou skutečně v počítači uložena přibližně. Např. 0.1_{10} má v typu float reprezentaci 0 (znaménko), 01111011 (exponent) a 10011001100110011001101 (mantisa). Exponent je tedy −4 a uložené číslo je binárně 0.00011001100110011001101 (včetně skryté jedničky). Převodem do desítkové soustavy dostaneme

$(2^{23}+2^{22}+2^{19}+2^{18}+2^{15}+2^{14}+2^{11}+2^{10}+2^{7}+2^{6}+2^{3}+2^{2}+2^{0})/2^{27}=\frac{13421773}{134217728}$, což je

přibližně, nikoliv však přesně, 0.1. (Přesně to je 0.100000001490116119384765625). Před tiskem této hodnoty ji Java musí převést do desítkové soustavy. Při tomto převodu se používá „chytrý“ algoritmus, který ji převede na 0.1 a proto se vytiskne přesně tato hodnota. Může nastat i opačná situace, kdy si zaokrouhlení nepřejeme a přesto k němu dojde. Např. kód

```
float f = 0.10000000149F;
System.out.println( f );
```

vytiskne 0.1.

Student: Aha. Je potřeba si potom dělat těžkou hlavu s tím, že reálná čísla jsou v počítači uložena přibližně, když tento „chytrý“ algoritmus najde „přesnou“ reprezentaci?

Java guru: Bohužel ano. Při aritmetických operacích se používá vnitřní reprezentace čísel a to může vést k překvapením. Např. kód

```
double d1 = 0.01;
double d2 = 0.09;
System.out.println( d1 + d2 );
```

vytiskne 0.09999999999999999.

Student: K zaokrouhlení reálného čísla na celé slouží Math.round(). Jak se ovšem provede zaokrouhlení např. na dvě desetinná místa?

Java guru: Zaokrouhlování na určitý počet desetinných míst Java nemá. Souvisí to s tím, že většina reálných čísel je v počítači uložena přibližně. Představ si, že bys chtěl zaokrouhlit 0.38 na jedno desetinné místo. Výsledkem by měla být hodnota 0.4, tu však není možné uložit přesně (ve dvojkové soustavě má nekonečný periodický rozvoj). Z toho je vidět, že takové zaokrouhlování nemůže fungovat.
Nicméně určitou možnost zaokrouhlování máš. Např. při tisku reálné hodnoty pomocí printf() můžeš říci, kolik desetinných míst se má vytisknout. Zaokrouhlení se provede automaticky.

```
double d = 0.38;
System.out.printf( "%.1f", d );
```

V tomto případě se vytiskne 0.4.

Student: Mistře, jak to, že se někdy místo desetinné tečky vytiskne desetinná čárka?

Java guru: Je to proto, že Java se přizpůsobuje národním zvyklostem. A protože je u nás zvykem používat desetinnou čárku, tiskne se desetinná čárka. Tedy za předpokladu, že provozuješ Javu na lokalizovaném operačním systému a používáš pro tisk printf(). Např.

```
System.out.printf( "%.2f", Math.PI );
```

vytiskne 3,14. Obdobně se Java chová při čtení vstupní hodnoty. Na počeštěném operačním systému očekává čárku, zatímco v anglickém prostředí tečku.