Záznam přednášek z předmětu

Počítače a programování 1

KIV/PPA1

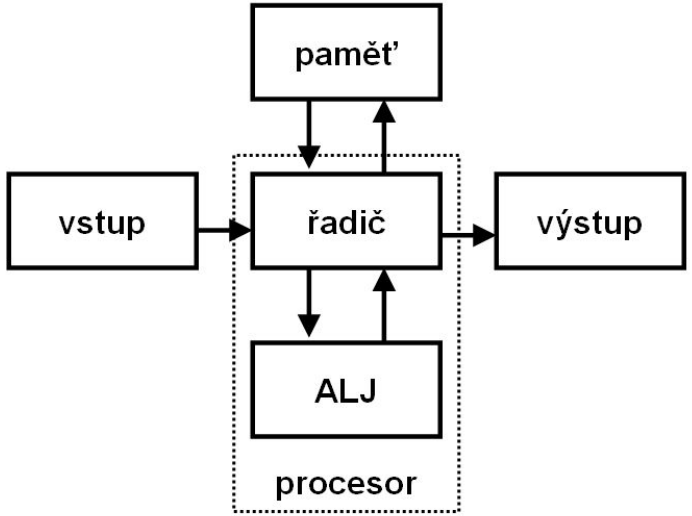
Pavel Herout

Obsah

Kapitola 1. Základní informace

1.1. Struktura počítače z pohledu programátora

- pro základní strukturu se stále používá asi 50 let staré schéma zavedené von Neumannem



- **Vstup** – klávesnice, soubor na HD, myš, scanner, čidla, mikrofon, CD, atd.
  - poskytuje data ke zpracování počítačem
- **Výstup** – obrazovka, soubor na HD, plotter, vypalovačka, reproduktory, atd.
  - produkuje data pro uživatele

  Poznámka

  Souhrnný název pro oba je **periferní zařízení** nebo I/O (*input/output*) či V/V (vstup/výstup)
- **Paměť** – vnitřní paměť (RAM)
  - uschovává program a dočasná data
- **Řadič** (řídící jednotka, *controller*)
  - podle programu dává pokyny ke zpracování dat a produkuje výstupní data
- **ALJ** – aritmeticko-logická jednotka
  - podle pokynů řadiče provádí jednotlivé výpočty a zpracování dat

- úkolem programátora je napsat předpis (= program, postup, algoritmus) pro řadič, tak, aby provedl požadovanou činnost

1.2. Nutné znalosti o jednotlivých částech

1.2.1. Paměť – hardwarový pohled

- vždy je tvořena posloupností bitů (bit = *binary digit*), zkratka **b**
- nejmenší část paměti, která se střídavě může nacházet ve dvou různých stabilních stavech a tak uchovávat elementární jednotku informace s hodnotami 0 nebo 1
- bit je pro adresování příliš malé množství informace
  - bity se sdružují po osmicích do paměťových míst nazývaných **bajt** (*byte*, též **slabika**), zkratka **B**
  - kapacita paměti se udává v MB nebo GB

Výstraha

Pozor na zkratky

K („kilo“)	2^{10} = 1024 ~ 1000
k	= 1000
M („mega“) = K * K	2^{20} = 1024 * 1024 = 1 048 576 ~ 1 000 000
G („giga“) = K * M	2^{30} = 1024 * 1024 * 1024 = 1 073 741 824 ~ 1 000 000 000
T ("terra") = M * M	2^{40} = 1024 * 1024 * 1024 * 1024 = 1 099 511 627 776 ~ 1 000 000 000 000

- rozlišujeme vnitřní (operační) paměť a vnější paměť
  - **vnitřní paměť** (RAM – *Random Access Memory*)
    - správně RAM-RWM (*Read Write Memory*) versus RAM-ROM (*Read Only Memory*)

      (srovnej s CD-RW versus CD-ROM)
    - umožňuje rychlé čtení (jednotky nanosekund) a zápis (jen RWM) uložených dat
    - typická velikost 256 MB až 4 GB
    - technologicky je dnes vyráběna výhradně z elektronických obvodů
    - způsob adresace – vždy adresujeme bajty
      - každý bajt má svou adresu
      - nezáporné celé číslo od 0
      - slouží pro jeho jednoznačnou identifikaci

  - **vnější paměť**
    - je asi 10^6 krát pomalejší než vnitřní
    - kapacita je jednotky GB až TB
    - používají se různé technologie:
      - elektronické obvody (různé druhy paměťových karet – SmartMedia,CompactFlash atd.)
      - magnetický záznam (pevné disky, diskety, pásky)
      - optický záznam (CD, DVD atd.)
    - způsob adresace – často ne bajty, ale větší celky = soubory
      - pojem adresy je složitější a udává se různě (např. disk, adresář, soubor, případně i pozice v souboru)
      - protože fyzická adresa může být udávána jako kombinace cylindru, hlavy a sektoru, udává se někdy kapacita v mocninách 10 ne 2

1.2.2. Paměť – softwarový pohled

- termín **bit** se používá ve více významech:
  - jako **elementární jednotka informace**, která říká, která ze dvou možností nastala (první nebo druhá, pravá nebo levá, pravda nebo nepravda atd.)
  - jako **jedna číslice** čísla zaznamenaného **v binární** (tj. dvojkové) **soustavě** (binární resp. dvojková číslice)
  - jako **nejmenší část paměti** počítače – viz dříve hardwarový pohled
- data jsou informace, které jsou v průběhu výpočtu uloženy v paměti vnitřní nebo vnější
  - typy dat: čísla, texty, obrázky, zvuky atd.
- aby mohla být data uložena ve vnitřní paměti musí být napřed převedena (zakódována) jako posloupnost bitů
  - základní posloupnost osmi bitů (bajt) je často velikostně nedostačující
- bajty se sdružují do větších celků – **slov**
  - typicky 2, 4 nebo 8 bytů dlouhých
  - označují se **W** (*word*), **DW** (*double word*) nebo **QW** (*quad word*)
- posloupnost *n* bitů umožňuje zobrazit celkem 2^n různých hodnot
- přehled všech souvislostí

bajtů	zkratka	bitů	zobrazených hodnot	rozsah	max. hodnota	řád
1	B	8	2^8	256	255	10^2
2	W	16	2^{16}	$2^6 * 2^{10} = 64K$	65 535	10^5
4	DW	32	2^{32}	$2^2 * 2^{30} = 4G$	4 294 967 295	10^9
8	QW	64	2^{64}	$2^4 * 2^{30} * 2^{30} = 16G*G$	18 446 744 073 709 551 615	10^{19}

1.2.3. ALJ – aritmeticko-logická jednotka

- podle pokynů řadiče provádí jednotlivé výpočty rozdělené do miniaturních (atomických) operací, např. sečtení dvou čísel
- operace jsou prováděny v registrech
  - jejich velikost v bitech (nikdy v bajtech) určuje typ počítače
    - dnes typicky 32 bitové, existují i 64 bitové

1.3. Algoritmizace

- úkolem programátora je napsat předpis (= program, postup, algoritmus) pro řadič, tak aby provedl požadovanou činnost
- **algoritmus**
  - postup při řešení určité třídy úloh, který je tvořen seznamem jednoznačně definovaných příkazů
  - zaručuje, že pro každou přípustnou kombinaci vstupních dat se po provedení konečného počtu kroků dospěje k požadovaným výsledkům
- vlastnosti algoritmu:
  - **hromadnost** (někdy není požadována)

    měnitelná vstupní data
  - **determinovanost**

    každý krok je jednoznačně definován
  - **konečnost a resultativnost**

    pro přípustná vstupní data se po provedení konečného počtu kroků dojde k požadovaným výsledkům
- algoritmus může být velmi jednoduchý i velmi složitý
- pro překonání potíží s rozsáhlými úlohami se používají dva základní principy
  - **dekompozice**
    - složité problémy se chápou jako kolekce jednodušších problémů
    - tak se hierarchicky postupuje tak dlouho, až máme množství jednoduchých akcí, které lze zvládnout dostupnými prostředky
    - každý krok dekompozice popisuje část systému v podrobnějších detailech
    - úzce souvisí s metodologií nazývanou **shora dolů** kdy vyjadřujeme řešení v termínech podproblémů, jejichž řešení považujeme za možné, ale zatím je neřešíme
      - tak hierarchicky postupujeme dolů, až se dostaneme na jednoduché akce vyjádřitelné programovacím jazykem
    - opačný postup **zdola nahoru** – vytváříme nejdříve základní stavební díly, ze kterých sestavujeme vyšší celky
      - mnohem méně používaný postup
  - **abstrakce**
    - koncepční zjednodušení složitého zanedbáním nedůležitých nebo nerealizovatelných detailů
- některé prostředky pro zápis algoritmu
  - přirozený jazyk, vývojové diagramy, pseudojazyk, programovací jazyk, atd.

Příklad 1.1. Ukázka různých zápisů triviálního algoritmu

Úloha: Vypiš dvakrát slovo `Ahoj`

- **přirozený jazyk**

  „vhodným příkazem vypiš Ahoj a pak vhodným příkazem vypiš Ahoj“

  - *dekompozice*

    „příkaz výpisu bude směrován na obrazovku a po výpisu se odřádkuje“

  - *abstrakce*

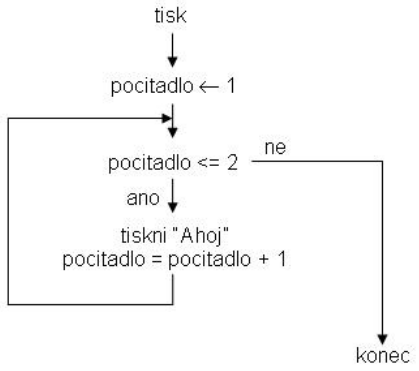
    - nezabýváme se podružnostmi, jako je např. použitý font, barva písma, znaková sada apod.
    - protože ale v konečném výsledku musejí být specifikovány, řešíme to použitím implicitních hodnot

    Poznámka

    algoritmus je funkční, ale postrádá vlastnost hromadnosti

    Oprava: „**v cyklu** dvakrát vhodným příkazem vytiskni řetězec Ahoj“

- **vývojový diagram**



- **pseudojazyk**

```
pocitadlo = 1
dokud platí pocitadlo <= 2 prováděj
begin
  println(Ahoj)
  inkrementuj pocitadlo
end
```

- **programovací jazyk**

```
pocitadlo = 1;
while (pocitadlo <= 2) {
  System.out.println("Ahoj");
  pocitadlo = pocitadlo + 1;
}
```


Poznámka

Nadále budeme zapisovat algoritmy programovacím jazykem

výhoda – spuštěním programu lze nejsnadněji zjistit, zda je algoritmus zapsán správně

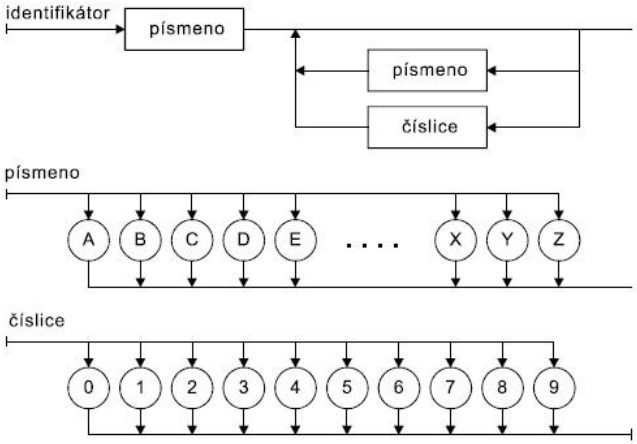
1.4. Programy a programovací jazyky

- program je předpis pro provedení určitých akcí počítačem zapsaný v programovacím jazyku
- programovací jazyky:
  - **strojově orientované**
    - strojový jazyk = jazyk fyzického procesoru
    - jazyk asembleru (jazyk symbolických adres/instrukcí, asembler) = strojový jazyk zapsaný symbolicky
  - **vyšší jazyky**
    - procedurální (příkazové, imperativní)
    - neprocedurální, např. funkcionální – Lisp, logické – Prolog
    - jazyky pro speciální oblasti, např. SQL
- hlavní rysy procedurálních jazyků (např. C, C++, C#, Java, Pascal, Basic, ...)
  - zpracovávané údaje mají formu datových objektů různých typů, které jsou v programu reprezentovány pomocí proměnných resp. konstant
  - program obsahuje deklarace a příkazy
  - deklarace definují význam jmen (identifikátorů)
  - příkazy předepisují akce s datovými objekty nebo způsob řízení výpočtu

1.4.1. Syntaxe a sémantika programovacích jazyků

- **Syntaxe**
  - souhrn pravidel udávajících přípustné tvary dílčích konstrukcí a celého programu
  - prostředky pro popis
    - syntaktické diagramy

Příklad 1.2. Identifikátor je posloupnost písmen a číslic začínající písmenem



- různé formy Backus-Naurovy formy (BNF)
  - rozšířená Backus-Naurova forma – EBNF

Příklad 1.3. Identifikátor je posloupnost písmen a číslic začínající písmenem

```
identifikátor = písmeno {písmeno | číslice}
písmeno = 'A' |'B' |'C' |'D' |'E' |... |'X' |'Y' |'Z'
číslice = '0' |'1' |'2' |'3' |'4' |'5' |'6' |'7' |'8' |'9'
```

- **Neterminály**:
  - `identifikátor`, `písmeno`, `číslice`
- **Terminály**:
  - ‘A’, ‘B’, ..., ‘0’, ‘1’, ...
- Význam metasymbolů:

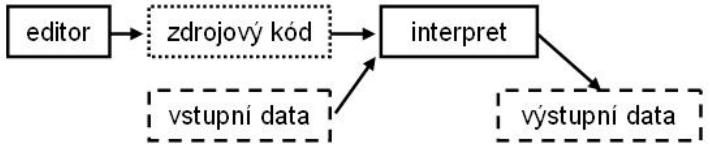
{x}	žádný nebo několik výskytů `x`
x | y	`x` nebo `y`

- **Sémantika**
  - udává význam jednotlivých konstrukcí
  - prostředky pro popis – obvykle popsána slovně

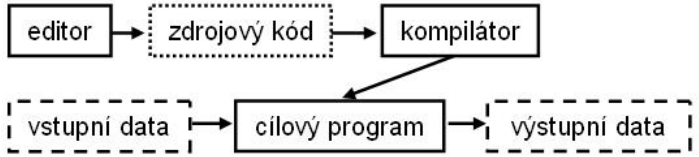
1.4.2. Implementace programovacích jazyků

Dvě základní metody:

- **interpretační**
  - větší přenositelnost programů, ale menší rychlost, syntaktické chyby odhaleny až při spuštění



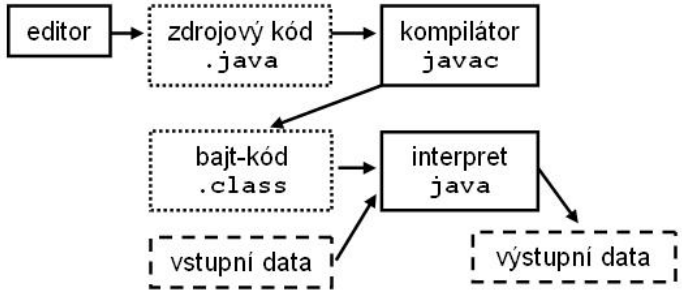
- **kompilační**
  - nulová přenositelnost mezi **platformami**, větší rychlost, syntaktické chyby odhaleny při překladu (platforma = procesor + operační systém)



- obě metody se dají kombinovat

1.5. Úvod do jazyka Java

- jazyk Java je příkladem kombinace kompilační a interpretační metody



  - program je tvořen jedním nebo několika zdrojovými soubory s příponou `.java`
  - zdrojové soubory se přeloží překladačem `javac` (v terminologii firmy Sun je to kompilátor) do vnitřní formy (*byte code*, **bajt-kód**), která je platformově nezávislá
    - překladem souboru `Jmeno.java` vznikne soubor s názvem `Jmeno.class`

- interpretaci vnitřní formy provede program `java` (JVM – *Java Virtual Machine*)
- program obvykle využívá řadu knihoven (*Java Core API – Application Programming Interface*), které je třeba mít k dispozici jak při překladu, tak při interpretaci
  - ♦ JVM + Java Core API = Java platforma
- rychlost interpretované Javy a kompilovaného jazyka (např. C) je srovnatelná
  - ♦ technologie JIT (*Just In Time*) a Hot-Spot

1.5.1. Vývojové prostředky

doporučené (tj. nainstalované v počítačové učebně) jsou dva prostředky

oba jsou volně šiřitelné a jsou dostupné na poskytnutém CD

1. **JDK, SciTe, příkazová řádka** – budeme používat při několika prvních cvičeních
   - JDK (*Java Development Kit*) – základní programy poskytované firmou Sun

     k dispozici na `http://java.sun.com`
   - SciTe – jednoduchý editor se základní podporou Javy

     k dispozici na `http://www.scintilla.org`
   - **Výhody**:
     - naprostá kontrola nad zdrojovým souborem, jednoduché pro naučení, minimální systémové nároky, editor je vhodný i pro operace s jinými typy souborů
   - **Nevýhody**:
     - pro složitější programy těžkopádné, přicházíme o výhodu podpory moderních vývojových prostředků, méně komfortní programování
2. **Eclipse** – budeme používat od zhruba druhé třetiny semestru
   - RAD (*Rapid Application Development*) nástroj špičkové profesionální kvality

     k dispozici na `http://www.eclipse.org/platform`
   - **Výhody**:
     - po zacvičení a poznání i základních možností výrazně zvyšuje produktivitu a komfort programování
       - ♦ významná pomoc při ladění (*debug*)
     - odhady založené na zkušenosti říkají, že pro zkušeného programátora použitím (kvalitního) RAD se efektivita zvyšuje 2 až 3krát
   - **Nevýhody**:
     - pro začátečníka komplikované prostředí

       - ♦ využijeme pouze omezenou část základních možností
     - značné systémové nároky, zejména na operační paměť (min. 256 MB, lépe 512 MB)

1.5.2. První program v Javě

vypíše daný text na obrazovku

```
public class PrvniProgram {
  public static void main(String[] args) {
    System.out.println("Ahoj, toto je prvni program");
  }
}
```

po překladu:

```
javac PrvniProgram.java
```

a spuštění:

```
java PrvniProgram
```

se na obrazovku vypíše:

```
Ahoj, toto je prvni program
```

- nejjednodušší program v jazyku Java je tvořen jedním **zdrojovým souborem**
- obsahuje deklaraci **veřejné třídy** (*public class*) pojmenované `PrvniProgram`
  - Konvence („štábní kultura“):
    - ♦ jména tříd se píší s prvním velkým písmenem
    - ♦ vnořené úseky kódu se odsazují (ideálně dvěma mezerami, ne tabulátorem)
- v ní je deklarována hlavní **metoda** (funkce) `main()`
  - je to **veřejná statická metoda** (*public static method*)
  - její první řádek se nazývá **hlavička metody**
  - klíčové slovo `void` vyjadřuje, že metoda **nevrací žádnou hodnotu** (jde o **proceduru**)
  - v závorkách je specifikace **formálního parametru** (*`String[] args`*), který zpočátku nevyužijeme
- zdrojový soubor **musí** mít jméno shodné se jménem veřejné třídy a příponu `.java`, tedy `PrvniProgram.java`
  - jakýkoliv jiný název, včetně `prvniprogram.java`, je chybný

Výstraha

Java **důsledně** rozlišuje malá a velká písmena (*case sensitive*)

1.5.3. Druhý program

vypíše daný text dvakrát (*n*-krát) na obrazovku:

```
public class DruhyProgram {
  public static void main(String[] args) {
    int n = 2;
    int pocitadlo = 1;
    while (pocitadlo <= n) {
      System.out.println("Ahoj");
      pocitadlo = pocitadlo + 1;
    }
  }
}
```

1.5.4. Jednoduchý grafický nástroj DrawingTool

- bude sloužit pro vizualizaci výstupů mnoha programů na cvičeních
  - byl vytvořen ing. Tesařem z KIV jako pomůcka pro PPA1
  - podrobný návod je v souboru `DrawingTool.pdf`
- velmi jednoduchý – je možné:
  - jednorázově zvolit velikost kreslícího plátna
  - jednorázově nastavit barvu pozadí
  - nakreslit čáru tloušťky 1 pixel z počátečního bodu do koncového bodu
  - nastavit barvu čáry
    - nastavenou barvou čáry se kreslí až do její další změny
- používá kartézské souřadnice
  - bod [0, 0] leží v levém horním rohu
  - souřadnice x roste doprava
  - souřadnice y roste dolů
- barvy jsou popsány pomocí příkazu: `import java.awt.*;`
  - k dispozici jsou:

    `Color.BLACK`, `Color.RED`, `Color.PINK`, `Color.ORANGE`, `Color.YELLOW`, `Color.GREEN`, `Color.MAGENTA`, `Color.CYAN`, `Color.BLUE`, `Color.GRAY`
- pro použití v programu potřebujeme soubor `DrawingTool.java` uložený ve stejném adresáři jako náš zdrojový soubor
- příklad použití


```
import java.awt.*;

public class Kresleni {
  public static void main(String[] args) {
    // inicializace: sirka = 300, vyska = 200, barva pozadi = bila
    DrawingTool dt = new DrawingTool(
                     300, 200, Color.WHITE, true);
    // nastaveni barvy cary na cernou
    dt.setColor(Color.BLACK);
    // nakresleni uhlopricky z leveho horniho rohu [0, 0]
    // do praveho dolniho rohu [300 - 1, 200 - 1]
    dt.line(0, 0, 299, 199);
  }
}
```

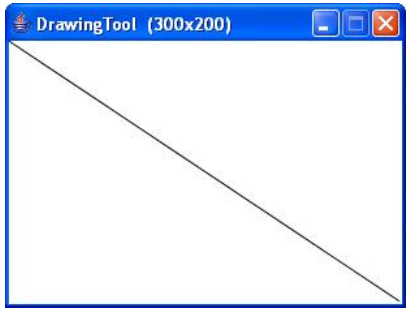
po překladu:

`javac Kresleni.java`

a spuštění

`java Kresleni`

vykreslí



Kapitola 2. Reprezentace dat, datové typy

2.1. Číselné soustavy

- **polyadické** – zobrazené mnohočlenem
- číslice tvořící zápis čísla jsou koeficienty polynomu obsahujícího mocniny základu
- základem soustavy může být libovolné přirozené číslo větší než 1
- posloupnost číslic c_n, c_{n-1}, …, c_2, c_1, c_0, c_{-1}, c_{-2}, …, c_{-m} v soustavě o základu *z* reprezentuje číslo

  $c_n * z^n + c_{n-1} * z^{n-1} + \ldots + c_2 * z^2 + c_1 * z^1 + c_0 * z^0 + c_{-1} * z^{-1} + c_{-2} * z^{-2} + \ldots + c_{-m} * z^{-m}$

  např.:

  $8274{,}56 = \mathbf{8} * 10^3 + \mathbf{2} * 10^2 + \mathbf{7} * 10^1 + \mathbf{4} * 10^0 + \mathbf{5} * 10^{-1} + \mathbf{6} * 10^{-2}$
- častý omyl je, že takto lze zapsat pouze celé číslo
- může-li být pochybnost o základu soustavy, zapisuje se číslo jako

  $(c_n\ c_{n-1} \ldots c_2\ c_1\ c_0, c_{-1}\ c_{-2} \ldots c_{-m})_z$

  např.:

  $(8274{,}56)_{10}\ (10011{,}01)_2\ (567{,}12)_8\ (\text{A1B2,9F})_{16}$
- pro celá čísla platí:
  - kapacita soustavy: $K = z^n$
  - největší hodnota soustavy: $N_{max} = z^n - 1$

  Příklad 2.1. Největší číslo desítkové soustavy na tři řádová místa

  z=10, n=3 N_{max} = 999 K = 1000

- významné soustavy **z hlediska počítačů**
  - dvojková (binární)
  - osmičková (oktalová)
  - šestnáctková (hexadecimální)
- významné soustavy **z hlediska lidí**
  - desítková (dekadická)

2.1.1. Dvojková (binární) soustava

- nativní soustava pro počítače – odpovídá fyzikální podstatě pamětí

- základem soustavy je **2**
- číslo se někdy zapisuje jako 1101_B místo známého 1101_2
- aritmetické operace
  - důležité pouze sčítání
  - odčítání je přičtení záporného čísla
  - násobení je opakované sčítání
  - dělení je opakované přičítání záporného čísla
- princip sčítání

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
```

- např.:

```
 101    5
 111    7
----   --
1100   12
```

- převod z desítkové soustavy do dvojkové: 12,6875

  xc = 12

  xd = 0,6875

  převádíme odděleně celou část (dělíme dvěma) a desetinnou část (násobíme dvěma)

xc	xc / 2	xc % 2	xd	xd * 2
12	6	0	0,6875	1,3750
6	3	0	0,3750	0,7500
3	1	1	0,7500	1,5000
1	0	1	0,5000	1,0000

  celou část sepisujeme pozpátku: 1100

  desetinnou část sepisujeme popředu: ,1011

  dohromady: $1100{,}1011_B \sim 12{,}6875_D$

2.1.2. Šestnáctková (hexadecimální) soustava

- základ je $16 = 2^4$
- používá se v textech a výpisech – je lépe čitelná a 4x úspornější než binární

- používané číslice **0** až **9** a **A** až **F** (ve významu 10 až 15) – možno i **a** až **f** (nepoužívat)
- číslo se často zapisuje jako $A1B2_H$ místo známého $A1B2_{16}$
- převod z dvojkové na šestnáctkovou
  - rozdělíme číslo na čtveřice od prava do leva
  - každá čtveřice je jedna šestnáctková číslice
  - 1011001001 = 10 1100 1001 = 2C9

2.1.3. Osmičková (oktalová) soustava

- základ je $8 = 2^3$
- používá se pouze z historických důvodů
- používané číslice 0 až 7

2.2. Kódování aneb data v počítači

- data převádíme na posloupnost bitů, respektive bajtů, tzn. je pevně daný rozsah (počet řádů)
  - nevýznamové nuly jsou důležité
- převod se nazývá kódování

2.2.1. Základní dělení kódů (datových typů)

- číselné
  - celočíselné
    - beznaménkové
    - znaménkové
      - přímý kód
      - inverzní kód
      - doplňkový kód
      - kód s posunutou nulou
  - reálné
    - v jednoduché přesnosti
    - ve dvojnásobné přesnosti
- znakové
- logické (booleovské)

2.2.2. Nejdůležitější datové typy

Poznámka

Podrobný výklad bude uveden z časových důvodů v jedné z posledních přednášek.

Poznámka

Příklady u celých čísel jsou udávány na osmi bitech. Pro větší rozsahy platí tytéž principy.

2.2.2.1. Celočíselné beznaménkové

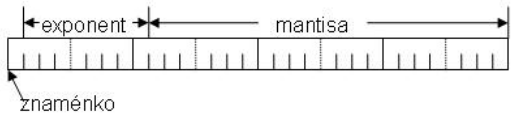
- zápis ve dvojkové soustavě zarovnaný na násobky bajtů (někdy nazývaný „přímý kód“)
- pouze pro nezáporná čísla (*unsigned*)
- např.: 1000 0011 = 131 nebo 0000 0011 = 3
- v Javě neexistuje

2.2.2.2. Celočíselné znaménkové ve dvojkovém doplňku

- v programovacích jazycích výhradně používané zobrazení celých znaménkových čísel
- kladná čísla stejná, jako u beznaménkového
- záporná – změní se hodnoty jednotlivých bitů a přičte se 1, např.:
  - 1111 1101 = -3
  - -1 ~ 1111 1111

2.2.2.3. Reálné

- *aproximace* reálných čísel – číslo nemusí být zobrazeno zcela přesně (rozdíl od celočíselných)
- též „zobrazení v **pohyblivé řádové čárce**“ (*floating point*)
- dle IEEE 754 na 4 bajtech (`float` – viz dále) nebo na 8 bajtech (`double`)



- mnohem komplikovanější, než celá čísla
  - používat s rozmyslem jen tam, kde nestačí celé číslo
  - neporovnávat navzájem na rovnost!

2.2.2.4. Znakové

- přiřazují každému znaku zvolené abecedy nezáporné celé číslo
- existuje (a používá se) mnoho kódů (podrobně později)
  - český text běžně v 11 různých kódováních!
- základní kódy
  - **ASCII** (*American Standard Code for Information Interchange*) – 7bitový kód pro anglickou abecedu, nejvyšší bit bajtu (**MSB** – *Most Significant Bit*) je vždy nulový
    - ♦ základ, který všechny další kódování rozšiřují
    - ♦ používané znaky začínají od čísla 20_H (32_D) znakem <mezera>
    - ♦ **číslice** jsou od 30_H (48_D) ~ **0** do 39_H (57_D) ~ **9**
    - ♦ velká neakcentovaná písmena jsou od 41_H (65_D) ~ **A** do $5A_H$ (90_D) ~ **Z**
    - ♦ malá neakcentovaná písmena jsou od 61_H (97_D) ~ **a** do $7A_H$ (122_D) ~ **z**
  - **Unicode** – nepřesné (souhrnné) označení pro 21bitový kód (vnitřně používán v Javě)

Výstraha

Je rozdíl mezi jednociferným číslem a číslicí (tj. znakem)

- jednociferné číslo (např. 2) má v jednom bajtu obraz 0000 0010
- číslice má obraz 0011 0010
  - číslice přicházejí z klávesnice a před použitím v programu jako číslo se musí převést
    - ♦ převod se provede odečtením 30_H
  - naopak po výpočtu chceme číslo vypsat na obrazovku jako znak a je opět nutný převod
    - ♦ převod se provede přičtením 30_H

2.2.2.5. Logické

- též **booleovské** podle pana Booleho (Booleova algebra)
- ukládají jen dvě hodnoty – *true* (pravda, **logická 1**) a *false* (nepravda, **logická nula**)
- prakticky se realizují pomocí celočíselného bezznaménkového typu

2.3. Datové typy

- ve vyšších programovacích jazycích abstrahujeme od binární podoby dat v paměti počítače
- využívají se kódy (tj. datové typy) zmíněné dříve

- s daty pracujeme jako s hodnotami různých datových typů, které jsou uloženy v datových objektech
- **primitivní datové typy** (též **základní** nebo **jednoduché**) – představují kolekci celých a reálných čísel, znaků a booleovských hodnot
  - vyskytují se v každém programovacím jazyce a jmenují se přibližně stejně (`int` versus `integer`)
- datový typ specifikuje
  - **množina hodnot**, např. celá čísla z intervalu -128 až +127
  - **množina operací**, které lze s datovým typem provádět, např.:
    - ♦ aritmetické operace – sčítání, odčítání, násobení, dělení celočíselné a modulo
      - – výsledkem je tentýž datový typ
    - ♦ relační operace – rovno, nerovno, menší, větší, apod.
      - – výsledkem je booleovská hodnota
    - ♦ bitové operace – posuvy, rotace (málokdy používané)
      - – výsledkem je tentýž datový typ

2.4. Datové typy v Javě

- konstantám se říká **literály** – rozdíl mezi proměnnou s konstantní hodnotou (viz dále) a literálem
- pozor na primitivní datové typy (vždy malými písmeny) a jejich **obalovací třídy** (*wraper class*) (první písmeno velké)

  např.: `byte` versus `Byte`

2.4.1. Celočíselné typy a jejich konstanty

- pouze znaménkové, ve dvojkovém doplňku
  - nelze použít neznaménková celá čísla
- celočíselné typy se od sebe liší pouze svojí velikostí a tím i rozsahem zobrazitelných čísel
  - `byte` (8b) -128 až +127
  - `short` (16b) -32 768 až +32 767
  - `int` (32b) -2 147 438 648 až +2 147 438 647
  - `long` (64b) -9 223 372 036 854 775 808 až +9 223 372 036 854 775 807

Poznámka

V začátcích programování používat jen `int`

- celočíselné konstanty mohou být zapsány ve třech číselných soustavách

- **desítkové** – posloupnost číslic 0 až 9, z nichž první nesmí být 0

  např.: 86, 15, 0, 1

- **osmičkové** – číslice 0 (nula) následovaná posloupností osmičkových číslic 0 až 7

  např.: 0126, 015, 0, 01

- **šestnáctkové** – číslice 0 (nula) následovaná znakem x (nebo X) a posloupností šestnáctkových číslic 0 až 9, a až f nebo A až F

  např.: 0x56, 0X56, 0x3A, 0XCD, 0xCD, 0xCd, 0x0, 0x1

2.4.2. Reálné datové typy a jejich konstanty

- `float` a `double` – standard IEEE 754 (viz výše)
  - prakticky používat pouze `double`, který je na 64 bitech o rozsahu asi 4,9E-324 až 1,7E+308
- reálné konstanty mohou začínat a končit **desetinou tečkou**, obsahovat znaménka, exponenty atd.

  např.: `15. 56.8 .84 3.14 5e6 7E23 -7E+23 +7E-23`

  - reálná konstanta je automaticky typu `double`

  Poznámka

  při operacích s reálnými čísly může výsledek operace nabýt několika „nenormálních“ hodnot, které ale nejsou chybové

  1. kladné a záporné nekonečno – `Double.POSITIVE_INFINITY` a `Double.NEGATIVE_INFINITY`
  2. NaN (*Not a Number*) – `Double.NaN`
- maximální a minimální hodnoty celočíselných i reálných typů lze získat pomocí konstant `MIN_VALUE` a `MAX_VALUE`, např.:

```
int i = Integer.MIN_VALUE;

double d = Double.MAX_VALUE;
```

2.4.3. Znakový typ a jeho konstanty

- pouze jeden – `char` a má velikost 16 bitů
- Java **vnitřně** pracuje se znaky v kódování Unicode – to jednoznačně řeší problémy s různými kódováními češtiny
- znakové konstanty jsou **vždy uzavřeny do apostrofů** a mohou být představovány
  1. jedním znakem – v případě (neakcentovaných) znaků běžně dostupných na klávesnici

     např.: `'A'`, `'1'`, `'%'`

  2. posloupností `'\uXXXX'`, kde *XXXX* jsou šestnáctkové číslice kódových bodů Unicode
     - tento zápis se často používá v případě akcentovaných znaků – obrana proti různým kódováním češtiny ve zdrojových souborech
     - lze tak ale zapsat libovolný znak

       např.: `'\u00C1'` ~ `'Á'`, `'\u00E1'` ~ `'á'`, `'\u011B'` ~ `'ě'`, `'\u0041'` ~ `'A'`
  3. „escape“ sekvencí

     `'\n'` ~ `'\u000A'` nová řádka (*newline*, *linefeed* – <LF>)

     `'\r'` ~ `'\u000D'` návrat na začátek řádky (*carriage return* – <CR>)

     `'\t'` ~ `'\u0009'` tabulátor (*tab* – <HT>)

     `'\b'` ~ `'\u0008'` posun doleva (*backspace* – <BS>)

     `'\\'` ~ `'\u005C'` zpětné lomítko (*backslash*)

     `'\''` ~ `'\u002C'` apostrof (*single quote*)

     `'\"'` ~ `'\u0022'` uvozovky (*quote*)
- znaky <CR>, <LF>, <tabulátor>, <BS>, <mezera> se souhrnně nazývají **bílé znaky** (*white spaces*)
  - vyskytují se běžně v **textových souborech**

2.4.3.1. Textové versus binární soubory

- textové soubory
  - čitelné běžným editorem (Notepad, SciTe apod.), protože obsahují jen zobrazitelné znaky a bílé znaky
  - organizovány po řádcích, řádky ukončeny <CR><LF> (Windows) nebo jen <LF> (Unix)
  - je záležitostí editoru, zda správně rozpozná ukončení řádky z jiné platformy
  - snadný převod pomocí SciTe
    - ♦ požadované ukončení řádku se zvolí v Nastavení/Konce řádků
    - ♦ záměna se provede pomocí Nastavení/Převést konce řádků
- binární soubory
  - v běžném editoru se objeví nečitelný výpis, protože obsahují všechny možné hodnoty bajtů
  - je třeba použít binární prohlížeč, který obsah zobrazí nejčastěji jako hexa čísla
  - existují i binární editory, kde lze hexa čísla měnit
    - ♦ je jich mnohem méně než textových editorů a jsou určeny pro profesionály
    - ♦ nepoužívat, pokud přesně nevíme, co děláme

2.4.4. Řetězcové konstanty

- zatím budeme používat pouze pro výpisy
- jsou složeny ze znaků
- tvoří se stejně, jako se tvoří znakové konstanty, pouze jsou uzavřeny do uvozovek

  např. typicky: `"pocet studentu"`
- lze použít i akcentované znaky

  např.: `"háček a čárka"`
- lze použít i různé způsoby zápisu znaků v jednom řetězci

  např.: `"Program kon\u010D\u00ED!\n"`
- lze použít automatické zřetězování dlouhých literálů oddělených mezerami, tabelátory nebo novými řádkami
  - spojovací znak je `+`

    např.: tři rovnocené zápisy:

    1. `"Takhle vypada velmi dlouhy retezec"`
    2. `"Takhle" + " vypada " + "velmi dlouhy retezec"`
    3. `"Takhle vypada " + "velmi "`
       `+ "dlouhy retezec"`
- potřebujeme-li pojmenovanou konstantu, použijeme (podrobnosti později)

```
final String JMENO = "Pavel";
```

Výstraha

Pozor na různé typy uvozovek dodávané (měněné) automaticky textovými procesory

- "správně" `"správně"`
- “nesprávně – anglické uvozovky” `“nesprávně – anglické uvozovky”`
- „nesprávně – české uvozovky“ `„nesprávně – české uvozovky“`

Poznámka

Znak je jiný datový typ než řetězec, dokonce i jednoznakový řetězec je jiný typ a nesmí být zaměňován se znakem.

- problém číslice versus číslo je o řád vyšší, protože v řetězci je několik číslic za sebou
- např. řetězec `"125"` představuje v paměti sekvenci tří bajtů


```
0011 0001    0011 0010    0011 0101
```

- jako neznaménkové celé číslo je uloženo v jednom bajtu

```
0111 1101
```

- neexistuje jednoduchý převod typu „odečíst 30_H“ jako u znaků
- naštěstí mají programovací jazyky vstupní a výstupní funkce, které tento převod provádějí více-méně automaticky
  - je třeba si uvědomit, že konverze řetězec obsahující číslice na číslo probíhá vždy při vstupu z klávesnice
  - opačná konverze číslo na řetězec obsahující číslice probíhá vždy při výpisu na obrazovku

2.4.5. Logický typ a jeho konstanty

- `boolean`
- může nabývat pouze dvou hodnot, představovaných logickými konstantami `true` (= logická 1) a `false` (= logická 0)
- základní operace s logickým typem

  `&&` – logický součin (konjunkce) – „jeden `false` zruší vše“

  `||` – logický součet (disjunkce) – „jeden `true` stačí“

  `!` – negace

x	y	x && y	x || y	!x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Výstraha

Operátory `&&` a `||` musejí být ve dvojici. Jsou-li samostatně, mají význam bitových operací.

2.5. Proměnné, deklarace a přiřazení

- **proměnná** je datový objekt v operační paměti označený (významovým) **symbolickým jménem** (vyjadřuje účel) a je v něm uložena hodnota, která se může v průběhu měnit
  - výhodou je, že se do paměti odkazujeme pomocí významových jmen, nikoliv pomocí adres
    - ♦ převod jména na adresu provede překladač

  např.:

symbolické jméno proměnné: `stavUctu`

datový typ: celočíselný znaménkový

hodnota: `12345`

- s proměnnou lze provádět libovolné operace dané množinou operací příslušného datového typu

2.5.1. Identifikátory

- jména proměnných, konstant, podprogramů, tříd, atd.
  - v podstatě vše, co píšeme do programu a nejsou to **klíčová slova** nebo **řetězcové konstanty**
- v Javě (a v mnoha běžných programovacích jazycích) platí:
  - délka identifikátorů není omezena
  - každý identifikátor musí začínat písmenem nebo podtržítkem, jako další znaky se mohou vyskytnout i číslice
  - identifikátory rozlišují velká a malá písmena (jsou *case sensitive*)
- způsob zápisu identifikátorů v Javě – **nutno dodržovat**, protože je důsledně dodržováno v celém Java Core API
  - **třídy** a rozhraní – identifikátor začíná vždy velkým písmenem a ostatní písmena jsou malá

    v případě, že je identifikátor tvořen více slovy, začíná každé slovo velkým písmenem

    např.: pro třídy `PrvniProgram`, `Pokus`, `String` a `StringBuffer` a pro rozhraní `Cloneable`
  - **metody** a **proměnné** – identifikátor začíná (a pokračuje) malým písmenem

    stejně jako u tříd se při použití více slov označuje začátek dalšího slova velkým písmenem

    např.: pro proměnné `pocet`, `pocetPrvku` a pro metody `start()`, `getSize()`.
  - **konstanty** – používají se pouze velká písmena

    ve víceslovných identifikátorech je oddělovačem podtržítko

    např.: `PI`, `MAX_VALUE`
  - **balíky** – identifikátor se skládá pouze z malých písmen

    ve složených jménech je oddělovačem tečka

    např.: `java.lang`

2.5.2. Deklarace

- proměnnou je nutno před prvním použitím **deklarovat**, tj. stanovit symbolické jméno, datový typ a požadavek na uložení do paměti (děje se automaticky)

  např.: `int stavUctu;`

- je možná deklarace s inicializací, kdy se proměnné navíc stanoví počáteční hodnota

  např.: `int stavUctu = 0;`
- to se často používá, protože před prvním čtením musí mít proměnná definovanou hodnotu
- chybný kód:

```
int i, j;

j = i + 2; // chyba při překladu
```

- Java dovoluje deklarovat proměnné kdekoliv v kódu a uvnitř třídy
- jedním příkazem lze vytvořit několik proměnných stejného typu
  - používat jen pro pomocné proměnné

  např.: `int i, j, k;`

2.5.2.1. Pojmenovaná konstanta

- speciální typ proměnné – hodnotu lze jednou nastavit (typicky inicializací při deklaraci) a pak ji nelze měnit, jen číst
  - od proměnné se vizuálně liší v deklaraci použitím dalšího **klíčového slova** (`final`)

  např.: `final int LIMIT_PUJCKY = 100000;`
- pokus o změnu konstanty skončí chybou

  např.: `LIMIT_PUJCKY = 500000;` // chyba

2.5.3. Přiřazení

- hodnotu proměnné lze nastavit nebo změnit přiřazovacím příkazem (není to rovnice!)

  např.: `stavUctu = stavUctu + 12345;`
- obecný tvar přiřazovacího příkazu je

  proměnná = *výraz*;

2.5.3.1. Výraz

- předepisuje výpočet hodnoty určitého typu
- může být složen z
  - **operandů**
    - proměnných
    - pojmenovaných konstant
    - literálů

- ♦ volání metod
- **operátorů** (`+ - * / %` atp.) – podrobně viz dále
- závorek

  např.: `x = (y + MAX) * 8 / pocetPrvku();`

  kde:

  - ♦ `y` je proměnná
  - ♦ `MAX` je pojmenovaná konstanta
  - ♦ `8` je literál
  - ♦ `pocetPrvku()` je volání metody

■ pořadí operací prováděných ve výrazu je dáno

- závorkami !!!
- prioritou operátorů

  např.: `x + y * z` se vyhodnocuje jako `x + (y * z)` protože `*` má vyšší prioritu než `+`
- asociativitou operátorů

  např.: `x + y + z` se vyhodnocuje jako `(x + y) + z` protože `+` je asociativní zleva

Poznámka

Kromě nejjednodušších výrazů se nespoléháme na prioritu ani asociativitu a použijeme důsledně závorky.

2.5.4. Typové konverze

■ proměnné lze přiřadit jen hodnotu stejného typu

■ chceme-li přiřadit jiný typ, je nutné **přetypování** (typová konverze)

- **implicitní** – proběhne automaticky
  - ♦ je to konverze z typů s nižším rozsahem na typy s vyšším rozsahem

    např.: `char` –> `int` –> `double`

```
double d; int i = 5;
d = i;
```

- **explicitní** – je nutno ji zapsat do programu
  - ♦ je to konverze z typů s vyšším rozsahem na typy s nižším rozsahem
  - ♦ dochází k možné ztrátě přesnosti nebo rozsahu


```
int i; double d = 3.14;
i = (int) d;   // i bude 3
```

Poznámka

Je dobrým zvykem zapisovat explicitní konverzi i na místo implicitní. Dáváme tím najevo, že víme, co děláme.

```
d = (double) i;
```

Výstraha

Přetypování má nejvyšší prioritu

```
int i = (int) d + f;     // chybně – přetypuje se jen d
int i = (int) (d + f);  // dobře – sečte se d a f a výsledek se přetypuje
```

2.5.5. Operátory

■ výrazy (viz výše) jsou složeny z operandů a operátorů

■ operátory lze dělit podle

- počtu operandů okolo nich na
  - ♦ unární – existují čtyři: `- + ++ --`
  - ♦ binární – je jich více
  - ♦ ternární – existuje pouze jeden: `?:`
- podle výsledku operace na
  - ♦ aritmetické – sčítání, odčítání, ...
  - ♦ relační – větší, menší, ...
  - ♦ bitové – málo používané (zájemci naleznou v literatuře)

2.5.5.1. Unární operátory

■ nejjednodušší – unární plus a mínus

např.: `i = -j;` nebo `i = +j;`

■ komplikovanější – inkrementační `++` a dekrementační `--`

- oba lze použít pouze na proměnné a to
  - ♦ před proměnnou – *prefix*

    např.: `++i //` zvětší hodnotu proměnné i o 1

    – je to inkrementování před použitím

– proměnná je nejprve zvětšena o jedničku a pak je tato nová hodnota vrácena jako hodnota výrazu

♦ za proměnnou – *suffix* (občas též *postfix*)

např.: `i++ //` zvětší hodnotu proměnné i o 1

– je to inkrementování po použití

– je vrácena původní hodnota proměnné a ta je pak zvětšena o jedničku

♦ v nejjednodušším (a nejpoužívanějším) případě nemá smysl *prefix* a *suffix* rozlišovat – výsledek je stejný

např.: `i++;` je stejné jako `++i;`

nebo: `--i;` je stejné jako `i--;`

♦ odlišnosti nastávají, jsou-li tyto unární operátory součástí výrazu

```
int i = 5, j = 1, k;
i++;                  // i bude 6
j = ++i;              // j bude 7, i bude 7
j = i++;              // j bude 7, i bude 8
k = --j + 2;          // k bude 8, j bude 6, i bude 8
```

Poznámka

Jednoduchá rada – nejsme-li si jisti, rozepíšeme výraz do více příkazů, např.:

```
--j;
k = j + 2;
```

2.5.5.2. Aritmetické operátory

■ výsledek je stejného typu jako operandy

`+` sčítání

`-` odčítání

`*` násobení

`/` dělení – celočíselné i reálné

`%` dělení modulo (zbytek po celočíselném dělení)

■ příklady

- `7 / 2 = 3`
- `-7 / 2 = -3`
- `7.0 / 2 = 3.5` // celočíselná 2 se převedla implicitní konverzí na reálnou 2.0 a dělení proběhlo jako reálné

- `7 / 2.0 = 3.5`
- `7 % 2 = 1`
- `-7 % 2 = -1`

Poznámka

Aritmetické operátory jsou typicky binární, ale unární `++` a `--` jsou taktéž aritmetické operátory.

Výstraha

U operací `+` a `*` může dojít k přetečení, tj. výsledek neodpovídá skutečnosti (podrobně viz později)

2.5.5.3. Relační operátory

■ hodnoty všech primitivních datových typů jsou uspořádané, tzn. lze je navzájem porovnávat

`>` větší

`<` menší

`>=` větší nebo rovno

`<=` menší nebo rovno

`==` rovno

`!=` nerovno

■ výsledkem relační operace je `true` (relace je platná) nebo `false`

■ v případě, že se typy operandů liší (např. `int` a `double`), proběhne implicitní konverze

Výstraha

Relační operátory v začátcích používat pouze ve spojitosti s příkazy `if`, `for`, `while` (viz dále).

Nepokoušet se o výrazy typu:

```
boolean b = ++i > 10;
```

Poznámka

Relační operátory mají nižší prioritu než aritmetické. Lze tedy psát bez závorek výrazy typu:

```
if (x * 2 < y - 3)
```

navíc operace s logickými typy (`&&` a `||` viz dříve) mají ještě nižší prioritu, než relační operátory. Lze napsat:

```
if (x * 2 < y - 3 && i / 5 == 8)
```

rozumnější ale je takto složité výrazy závorkovat

```
if ( (x * 2 < y - 3) && (i / 5 == 8) )
```

nebo dokonce

```
if ( ((x * 2) < (y - 3)) && ((i / 5) == 8) )
```

2.5.5.4. Přiřazovací operátory

- zjednodušují zápis, pokud je v přiřazovacím výrazu přítomna na levé i pravé straně stejná proměnná, např.:

```
x = x + 5;
```

lze zapsat zkráceně jako:

```
x += 5;
```

- přiřazovací operátory existují pro všechny aritmetické operátory

```
+= -= *= /= %=
```

- oba příklady pro: `int i = 4, j = 3;`
  - `j += i;` // j bude 7, i bude 4
  - `j *= i - 2;` // j bude 14, protože `j = j * (i - 2);`

Poznámka

Používejte tento zkrácený zápis uvážlivě a pokud si nejste jisti:

1. **závorkujte**, např.:

```
j *= (i - 2);
```

2. použijte **pomocnou proměnnou**, např.:

```
int pom = i - 2;
j *= pom;
```

Poznámka

Máme-li proměnnou `citac`, lze ji zvětšit o jednu čtyřmi způsoby

```
citac = citac + 1;
citac += 1;
++citac;
citac++;
```

Programátor v Javě použije vždy poslední způsob.

2.5.6. Komentáře

- pomáhají vyznat se ve zdrojovém kódu
- v Javě jsou komentáře tří typů

1. **do konce řádky** `//`

```
i++;  // inkrementace proměnné i
```

2. **blokový** `/* */`

```
/* inkrementace proměnné i,
   protože to algoritmus vyžaduje
*/
i++;
```

Poznámka

Nachází se zásadně **před** komentovaným kódem.

3. **dokumentační** `/** */` – viz podrobně dále

```
/**
  metoda sečte dvě čísla
*/
int sectiDveCisla(int a, int b) {
```

- je nutné je používat, míru (množství) si musí každý zvolit sám (není-li stanoveno např. firemní politikou)
  - komentujeme vždy, když:
    - použijeme nestandardní jazykovou nebo algoritmickou konstrukci

```
cena *= 1.19;  // přičtení 19% DPH
```

```
jmeno = sc.next();
/* přečtení a zahození všech zbývajících znaků
   na řádce včetně konce řádku
*/
sc.nextLine();
```

    - pokud jsme v daném místě kódu museli přemýšlet, rozhodovat se nebo hledat chybu

```
/* roky 1700, 1800 a 1900 nejsou přestupné
   roky 1600 a 2000 jsou přestupné, proto: rok % 400 == 0
*/
if (rok % 4 == 0  &&  rok % 100 != 0  ||  rok % 400 == 0) {
  prestupnyRok = true;
}
```

- naopak **ne**komentujeme, pokud je z kódu jeho účel zřejmý

```
cena += DPH_19;   // přičtení 19% DPH - tento komentář je zbytečný
```

- komentáře jsou při překladu ze zdrojového kódu vypuštěny, tzn. v programu se neobjeví

2.5.6.1. Dokumentační komentáře

- vyskytují se nejčastěji před názvy tříd a metod
  - tyto komentáře zásadně používáme (byť se to zdá v krátkých školních příkladech zcela zbytečné)
  - je-li v programu pouze jedna metoda (tj. `main()`), stačí dokumentační komentář pro třídu
- musí se vyskytnout bezprostředně před názvem třídy nebo metody

  dobře

```
import java.util.*;

/**
 * Načtení vstupního řetězce z klávesnice
 * a jeho výpis na obrazovku
 * @author P.Herout
*/
public class Vstup {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String jmeno = sc.next();
    System.out.println(jmeno);
  }
}
```

  špatně – mezi komentářem a názvem třídy je příkaz `import` – dokumentace se sice vygeneruje, ale bez našich textů

```
/**
 * Načtení vstupního řetězce z klávesnice
 * a jeho výpis na obrazovku
 * @author P.Herout
*/
import java.util.*;

public class Vstup {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String jmeno = sc.next();
    System.out.println(jmeno);
  }
}
```

- programem `javadoc` se z nich automaticky vygeneruje dokumentace v HTML

  použijeme příkaz

```
javadoc -d mojeDokumentace Vstup.java
```

  který založí podadresář `mojeDokumentace` a v něm vytvoří dalších 14 (čtrnáct!) `.HTML` souborů a jeden podadresář

  - z těchto souborů prohlížíme soubor `index.html`, všechny ostatní jsou pomocné a dají se z něj „doklikat“

  informace o autorovi není implicitně do HTML generována; pokud ji požadujeme, musíme použít příkaz

```
javadoc -author -d mojeDokumentace Vstup.java
```

- další podrobnosti viz později u metod

Kapitola 3. Terminálové V/V, řídící struktury, enum, ladění

3.1. Terminálové formátované vstupy a výstupy (V/V nebo I/O)

- **terminálový** = souhrnné označení pro nejprimitivnější V/V v příkazovém okénku
  - vstup z klávesnice
  - výstup na obrazovku v textovém režimu
- **formátovaný** = čísla jsou při výstupu automaticky převedena z binární formy do řetězce číslic v dekadické soustavě

  např.: číslo 0111 1101_B zabírající v paměti 1 bajt je převedeno do řetězce tří číslic „125“ a ten je vypsán
  - pro vstupy platí opačný postup, tj. (automatický) převod z řetězce číslic na číslo v binární reprezentaci

Poznámka

Převod z řetězce na binární reprezentaci a naopak nemusí být jen automaticky u V/V. Lze jej provést i kdekoliv v programu. Podrobnosti později.

3.1.1. Klasický výstup na obrazovku

- používá se metoda `System.out.print(parametr)`, která vypíše `parametr` a kurzor ponechá za posledním vypsaným znakem na stejné řádce
  - `parametr` může být proměnná nebo konstanta libovolného primitivního datového typu nebo řetězec
    - princip fungování je ten, že pro každý primitivní typ existuje implicitní konverze na typ `String`

```
int i = 15;
System.out.print(i);
System.out.print('B');
System.out.print(3.14);
System.out.print("ahoj lidi");
```

    vypíše: `15B3.14ahoj lidi`
  - jedním příkazem lze vypsat i více proměnných atd. najednou
    - jednotlivé položky se oddělují znakem + a je nutné, aby první položkou byl řetězec

```
int i = 15;
System.out.print(i + 'B');          // vypíše: 81
System.out.print("i = " + i + 'B'); // vypíše: i = 15B
```


Poznámka

81 je 15 + 66, kde 66 je kód znaku 'B'

- nenapadá-li nás vhodný úvodní řetězec, použijeme prázdný řetězec

```
System.out.print("" + i + 'B');  // vypíše: 15B
```

- prakticky je vhodné každou položku uvést vysvětlujícím textem nebo alespoň oddělit bílým znakem

```
System.out.print("i = " + i + " znak = " + 'B');
  // vypíše: i = 15 znak = B
System.out.print("i = " + i + " " + 'B');
  // vypíše: i = 15 B
```

- někdy si ale naopak přejeme vyhodnotit výraz před výpisem – pak se použijí závorky

```
int i = 1, j = 2;
System.out.print(i + j);
  // vypíše: 3
System.out.print("soucet = " + i + j);
  // vypíše: soucet = 12
System.out.print("soucet = " + (i + j));
  // vypíše: soucet = 3
```

- častý požadavek na odřádkování lze řešit několika způsoby
  - nejméně vhodný – za výpisem použít příkaz `System.out.println();`
  - nejpoužívanější – místo `System.out.print(parametr)` použít `System.out.println(parametr)`
    - provede totéž, co předchozí výpisy a navíc za posledním vypsaným znakem odřádkuje
  - sofistikovaný – znak `'\n'` (nebo řetězec `"\n"`) kdekoliv ve vypisovaných položkách způsobí odřádkování
    - takto lze v jednom výpisu odřádkovat i několikrát
    - tento způsob nezávisí na použití metody `print()` nebo `println()`

      např. následující čtyři příkazy mají stejnou funkčnost a vždy vypíší `i = 15` a na novou řádku `B`

```
System.out.print("i = " + i + "\n" + 'B' + "\n");
System.out.print("i = " + i + '\n' + 'B' + '\n');
System.out.println("i = " + i + "\n" + 'B');
System.out.println("i = " + i + '\n' + 'B');
```

Poznámka

Použijeme-li řetězec s dvojicí znaků `"\r\n"`, bude na platformě MS Windows výstup stejný. Při výpisu jen znaku `'\n'` se před něj automaticky doplní znak `'\r'`.

- ♦ pragmatický přístup – používejte vždy `System.out.println()` a každou řádku vypisujte samostatným příkazem

```
System.out.println("i = " + i);
System.out.println('B');
```

- při výpisu lze změnit typ vypisované položky

```
char c = 'A';
System.out.println("Znak " + c
                   + " ma ASCII hodnotu: "+ (int) c);
```

3.1.2. Komplexní řešení výstupu

Poznámka

Používáme jen tehdy, když nám nestačí možnosti `System.out.println()`.

- přichází od JDK1.5 (dříve se řešilo pomocí `java.text.NumberFormat`)
- jasná inspirace možnostmi jazyka C
- možnost volit šířku výpisu, zarovnávání doprava či doleva, nevýznamové mezery a množství dalších věcí
  - vyčerpávající informace i s příklady viz `java.util.Formatter`
- základní metoda `System.out.format(parametr)`
- vyřešeno přenositelně odřádkování pomocí `%n`, např.:

```
System.out.format("nova radka%n");
```

- základní princip je formátovací řetězec jako první parametr a pak jednotlivé položky jako další parametry
- základní pravidla
  - ve formátovacím řetězci jsou za znakem `%` formátovací znaky
  - kolik je znaků `%`, tolik musí být dalších parametrů
  - `System.out.format("i = %d, j = %d%n", i, j);`

3.1.2.1. Výpis celého čísla v desítkové soustavě

- používá se `%d`, např. pro `int i = -1234;`

```
System.out.format("i = %d%n", i);        // i = -1234
```

- počet míst lze stanovit, pak se doplňují mezery zleva, tj. zarovnání doprava, např.:

```
System.out.format("i = %7d%n", i);       // i =   -1234
```


- počet míst lze stanovit a zarovnat doleva (zbylé místo se doplní mezerami), např.:

```
System.out.format("i = %-7dahoj%n", i);  // i = -1234   ahoj
```

- lze vynutit výpis i + znaménka, např. pro `int i = 1234;`

```
System.out.format("i = %+7d%n", i);      // i =   +1234
```

- vynutí se výpis nevýznamových nul, např.:

```
System.out.format("i = %07d%n", i);      // i = -001234
```

- vypisuje se i oddělovač řádů, např.:

```
System.out.format("i = %,7d%n", i);      // i = -1 234
```

Poznámka

Oddělovač řádů je pevná mezera šířky čtvrt čtverčíku („čtverčík“ je typografický termín). Není-li tento znak k dispozici v používaném fontu, zobrazí se nesmyslný znak, např.:

```
i = -1á234
```

3.1.2.2. Výpis celého čísla v jiných soustavách

- osmičková soustava, např. pro `int j = 30;`

```
System.out.format("j = %o%n", j); // j = 36
```

- šestnáctková soustava, např. pro `int j = 30;`

```
System.out.format("j = %X%n", j); // j = 1E
```

- počet míst lze určit, např.:

```
System.out.format("j = %3X%n", j); // j =  1E
```

- lze vynutit nevýznamové nuly, např. pro: `int j = 10;`

```
System.out.format("j = %02X%n", j); // j = 0A
```

3.1.2.3. Výpis znaku

- používá se `%c`, např. pro `char c = 'a';`

```
System.out.format("c = %c%n", c); // c = a
```

- lze použít přetypování a lze vypsat více proměnných najednou

```
System.out.format("Znak %c ma ASCII hodnotu: %d%n", c, (int) c);
```

```
// Znak a ma ASCII hodnotu: 97
```

3.1.2.4. Výpis reálného čísla

- výpis jako běžné reálné číslo `%f`, např. pro `double d = 1234.567;`

```
System.out.format("d = %f%n", d); // d = 1234,567000
```

Poznámka

Desetinný oddělovač je závislý na lokalitě – pro ČR je to čárka, nikoliv tečka.

- výpis ve vědeckotechnické notaci `%g`, např. pro `double d = 1234.567;`

```
System.out.format("d = %g%n", d); // d = 1.234567e+03
```

Poznámka

Desetinný oddělovač je tečka.

- lze nastavit počet míst celkem (10) a počet míst za desetinným oddělovačem (1), číslo bude zaokrouhleno

```
System.out.format("d = %10.1f%n", d); // d = 1234,6
```

- lze použít zarovnání doleva, výpis nevýznamových nul, oddělovač řádů apod. stejně jako u celého čísla

3.1.2.5. Výpis řetězce

- používá se `%s`, např. pro `String s = "Ahoj lidi";`

```
System.out.format("s = %s%n", s); // s = Ahoj lidi
```

- řetězec lze vypsat velkými písmeny

```
System.out.format("s = %S%n", s); // s = AHOJ LIDI
```

- lze stanovit šířku výpisu, výpis bude zarovnán doprava

```
System.out.format("s = |%11s|%n", s);  // s = |  Ahoj lidi|
```

- výpis lze zarovnat i doleva

```
System.out.format("s = |%-11s|%n", s); // s = |Ahoj lidi  |
```

3.1.3. Formátovaný vstup

- přichází od JDK1.5, dříve se řešilo vlastními metodami
  - řešení je jednoduché a funguje stejně i pro čtení ze souborů (viz později)
- je poskytován třídou `java.util.Scanner`

  - na začátku zdrojového kódu musí být příkaz

```
import java.util.Scanner;
```

    nebo stačí příkaz

```
import java.util.*;
```

- čtení je nutné „inicializovat“ (přesné vysvětlení a podrobnosti později) příkazem

```
Scanner sc = new Scanner(System.in);
```

- často se jako bezprostředně další příkaz dává přepnutí do US lokality, aby při čtení reálných čísel byl desetinný oddělovač tečka (podrobnosti viz dále a též v předmětu KIV/JXT)

```
sc.useLocale(Locale.US);
```

- čtení celého čísla

```
int i = sc.nextInt();
```

- čtení reálného čísla

```
double d = sc.nextDouble();
```

- čtení znaku – není tak elegantní, je třeba přečíst celou řádku a vyseparovat první znak

```
char c = sc.nextLine().charAt(0);
```

- čtení řetězce – řetězec je čten do prvního bílého znaku
  - je-li více řetězců na řádce (např.: `ahoj lidi`), přečte se jen první, tj. `ahoj`

```
String s = sc.next();
```

- čtení celé řádky do řetězce, znak(y) odřádkování jsou přečteny, ale zahodí se – nejsou součástí načteného řetězce (řeší problém, jak se zbavit znaku(ů) konce řádky – viz dále)

```
String radka = sc.nextLine();
```

- ilustrativní příklad

```
import java.util.*;

public class NacitaniRealnychCisel {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    sc.useLocale(Locale.US);

    System.out.print("Zadej 1. odvesnu: ");
    double d1 = sc.nextDouble();
    System.out.print("Zadej 2. odvesnu: ");
    double d2 = sc.nextDouble();
    double prepona = Math.sqrt(d1 * d1 + d2 * d2);
    System.out.println("Prepona je: " + prepona);
```

```
    }
  }
```

3.1.3.1. Problém vyprázdnění vstupu

- neznalost tohoto problému při vícenásobném (opakovaném) vstupu z klávesnice způsobí značné potíže
  - problémy jsou nejčastěji při čtení v cyklech (viz též dále)
- ilustrativní příklad problému

```
import java.util.*;

public class VyprazdneniVstupuProblem {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej prvni cele cislo: ");
    int i1 = sc.nextInt();
    System.out.println("Zadal jsi: " + i1);

    System.out.print("Zadej retezec: ");
    String s = sc.next();
    System.out.println("Zadal jsi: " + s);

    System.out.print("Zadej znak: ");
    char c = sc.nextLine().charAt(0);
    System.out.println("Zadal jsi: " + c);

    System.out.print("Zadej druhe cele cislo: ");
    int i2 = sc.nextInt();
    System.out.println("Zadal jsi: " + i2);
  }
}
```

vypíše např.:

```
Zadej prvni cele cislo: 123
Zadal jsi: 123
Zadej retezec: abc
Zadal jsi: abc
Zadej znak: Exception in thread "main" ►
java.lang.StringIndexOutOfBoundsException
: String index out of range: 0
        at java.lang.String.charAt(Unknown Source)
        at VyprazdneniVstupuProblem.main(VyprazdneniVstupuProblem.java:16)
```

na zadání znaku se vůbec nečeká a program v tom místě skončí chybou

- je ovšem zajímavé, že při bezprostředně předcházejícím čtení řetězce se nic podobného nestane
- vysvětlení je jednoduché

- metody `nextInt()`, `nextDouble()` a `next()` fungují jako „žravé“ (*greedy*)
- program tedy funguje takto:
  - ♦ všechny případné bílé znaky (mezery, tabulátory, konce řádek) před vlastním `int` jsou metodou `nextInt()` automaticky přečteny a „požrány“
    - skutečné čtení čísla (zde `123`) začíná až od prvního nebílého znaku (zde `1`)
    - čtení končí po přečtení prvního bílého znaku za číslem (nejčastěji znaku(ů) konce řádky po stisku klávesy Enter)
      - tento znak se ale „nepožírá“ v tomto čtení, ale zůstává ve **vyrovnávací paměti** (*buffer*) klávesnice
  - ♦ následuje čtení řetězce pomocí `next()`, které nejprve „požere“ všechny předcházející bílé znaky
    - skutečné čtení řetězce (zde `abc`) začíná až od prvního nebílého znaku (zde `a`)
    - čtení opět skončí po přečtení prvního bílého znaku za řetězcem
    - tím jsou znaky odřádkování, které zůstávají v bufferu klávesnice
  - ♦ metoda `nextLine()` narozdíl od předchozích metod nefunguje jako „žravá“
    - přečte vše až do konce řádky
    - problém je v tom, že znaky konce řádky jsou v bufferu klávesnice uloženy již od předchozího čtení, takže tato metoda načte pouze ukončení řádky
    - znaků konce řádky se metoda automaticky zbaví – výsledkem je tedy prázdný řetěz
    - z prázdného řetězu se pokoušíme získat jeho první znak, což vyvolá výjimku (chybu v programu)
- řešení je jednoduché – po každém čtení pomocí metod `nextInt()`, `nextDouble()` a `next()` vyprázdnit buffer klávesnice metodou `nextLine()`
  - to zaručí, že každé následující čtení začíná „s čistým stolem“
- funkční řešení

```
import java.util.*;

public class VyprazdneniVstupuOprava {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej prvni cele cislo: ");
    int i1 = sc.nextInt();
    sc.nextLine();   // zde ve skutecnosti zbytecne
    System.out.println("Zadal jsi: " + i1);

    System.out.print("Zadej retezec: ");
    String s = sc.next();
    sc.nextLine();   // zde nutne
    System.out.println("Zadal jsi: " + s);
```

```
    System.out.print("Zadej znak: ");
    char c = sc.nextLine().charAt(0);
//    sc.nextLine();   // zde nesmi byt
    System.out.println("Zadal jsi: " + c);

    System.out.print("Zadej druhe cele cislo: ");
    int i2 = sc.nextInt();
    System.out.println("Zadal jsi: " + i2);
  }
}
```

vypíše např.:

```
Zadej prvni cele cislo:    123
Zadal jsi: 123
Zadej retezec: abc
Zadal jsi: abc
Zadej znak: e
Zadal jsi: e
Zadej druhe cele cislo: 987
Zadal jsi: 987
```

3.2. Základní matematické funkce

- potřebné při číselných výpočtech
- jsou poskytovány třídou `java.lang.Math`, kterou není třeba připojit příkazem `import` na začátku zdrojového kódu
  - jedná se o statické metody (viz později), takže je nutné je volat jako `Math.`*`jmenoMetody`*`()`
  - většina metod má své parametry typu `double`, tzn. případný typ `int` není nutné přetypovávat (funguje implicitní konverze), ale je to velmi vhodné
  - obsahuje též konstanty `Math.PI` a `Math.E`
- stručný přehled některých metod
  - absolutní hodnota – pro `int` i `double`

```
int j = -5;
int i = Math.abs(j);
double d = 3.14;
double f = Math.abs(d);
```

  - větší a menší číslo – pro `int` i `double`

```
int i = 3, j = 5;
int vetsi = Math.max(i, j);
double d = 3.14, f = 1.3;
double mensi = Math.min(d, f);
```


  - druhá odmocnina – pro `double`

```
double d = 9.0;
double f = Math.sqrt(d);
```

  - libovolná mocnina (tzn. i odmocnina) – pro `double`

```
int i = 2;
int kilo = (int) Math.pow(i, 10);
double d = 27;
double tretiOdmocnina = Math.pow(d, 1 / 3.0);
```

  - e^x a přirozený logaritmus – pro `double`

```
double d = Math.exp(3.0);
double logaritmus = Math.log(d);
```

  - desítkový logaritmus – pro `double`

```
double d = 1E4;
double logaritmus = Math.log10(d);
```

  - převody mezi stupni a radiány – pro `double`

```
double stupen = 180;
double radian = Math.toRadians(stupen);
stupen = Math.toDegrees(radian);
```

  - trigonometrické funkce – pro `double`

```
double stupen = 180;
double radian = Math.toRadians(stupen);
double sin = Math.sin(radian);
double cos = Math.cos(radian);
double tan = Math.tan(radian);
```

3.2.1. Náhodná čísla

- při mnoha výpočtech využíváme jako zkušební data (pseudo)náhodná čísla
  - ta získáváme nejsnáze pomocí třídy `Random`
    - programu musí předcházet příkaz: `import java.util.*;`
    - třídu musíme (podobně jako `Scanner`) inicializovat příkazem:

```
Random r = new Random();
```

  - v začátcích nám z mnoha možností postačí jen metoda `nextInt(int max)`

- ♦ vrací celé číslo v intervalu `<0, max-1>`
- potřebujeme-li reálné číslo, použijeme `nextDouble()`
  - ♦ vrací reálné číslo v intervalu `<0, 1)`

■ ilustrativní příklad

```
import java.util.*;

public class NahodnaCisla {
  public static void main(String[] args) {
    Random r = new Random();
    int i = r.nextInt(10);
    System.out.println("prvni: " + i);
    System.out.println("druhe: " + r.nextInt(10));
    System.out.println("realne: " + r.nextDouble());
  }
}
```

vypíše např.:

```
prvni: 3
druhe: 8
realne: 0.38141405942277307
```

3.3. Řídící struktury

■ jsou to programové konstrukce, které se skládají z dílčích příkazů a pro ně předepisují způsob provedení

- česky – pomocí nich ovlivňujeme směr provádění programu
  - ♦ jinak by program probíhal postupně od prvního příkazu do posledního (tak fungují jen nejtriviálnější školní programy)

■ existují tři druhy řídících struktur (též nazývané **strukturované příkazy**)

1. **posloupnost** – příkazy v posloupnosti se vždy provedou v pořadí, jak jsou zapsány (to byly všechny dříve uvedené programy)
   - **složený příkaz** – několik příkazů uzavřených do `{ }` (též nazývaný **blok**)

```
{
  int i = 5;
  int j;
  j = i * 10;
  System.out.println("j = " + j);
}
```

Poznámka

Důsledně odsazujte jednotlivé příkazy! Významně to zvyšuje čitelnost programu.

2. **větvení** – v závislosti na splnění podmínky se provede jen určitá část programu

   - příkaz `if` pro jednoduché větvení (neúplná podmínka)
   - příkaz `if-else` pro dvojité větvení (úplná podmínka)
   - příkaz `switch` pro mnohonásobné větvení
3. **cyklus** – část programu se provede opakovaně
   - příkaz `for` pro cyklus s předem známými mezemi („načti deset čísel“)
   - příkaz `while` pro cyklus s předem neznámými mezemi („čti čísla tak dlouho, dokud nenarazíš na konec souboru“)
   - příkaz `do-while` podobný příkazu `while`

Poznámka

Ovládnutí těchto několika málo jazykových konstrukcí znamená umět programovat na základní úrovni.

3.3.1. Motivační příklad – od sekvence k cyklu

■ základním způsobem výpočtu programu je vykonávání jednotlivých příkazů v pořadí, jak jsou zapsány v programu

- nevýhoda – použitelné pouze pro triviální algoritmy (navíc s omezenou použitelností – jednorázově)

■ úkol pro všechny následující programy je vypsat na obrazovku největší ze zadaných čísel

3.3.1.1. Nefunkční řešení pomocí sekvence

začínáme pouze s dvěma čísly

```
import java.util.*;

public class Sekvence1 {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej cele cislo: ");
    int prvniCislo = sc.nextInt();
    System.out.print("Zadej cele cislo: ");
    int druheCislo = sc.nextInt();

    int vetsiZCisel;

    // algoritmus
    vetsiZCisel = prvniCislo;
    vetsiZCisel = druheCislo;

    // závěrečný tisk
    System.out.println("Vetsi cislo = " + vetsiZCisel);
  }
}
```

vypíše např.:

```
Zadej cele cislo: 3
Zadej cele cislo: 5
Vetsi cislo = 5
```

program pracuje správně pouze, je-li větší číslo zadáno jako druhé

```
Zadej cele cislo: 5
Zadej cele cislo: 3
Vetsi cislo = 3
```

v tomto případě potřebujeme „přeskočit“ (tj. vykonat podmíněně) přiřazovací příkaz: `vetsiZCisel = druheCislo;`

3.3.1.2. Nevhodné řešení pomocí neúplné podmínky

- podmínka je speciální příkaz, který se při provádění může podmíněně přeskočit, tedy neprovést
  - většinou se snažíme napsat podmínku obráceně, tak, aby se příkaz nepřeskakoval, ale aby se vykonával

```
// algoritmus
vetsiZCisel = prvniCislo;
if (prvniCislo < druheCislo) {
  vetsiZCisel = druheCislo;
}
```

vypíše např.:

```
Zadej cele cislo: 5
Zadej cele cislo: 3
Vetsi cislo = 5
```

3.3.1.3. Nevhodné řešení pomocí dvou neúplných podmínek

pokud neumíme rozhodnout, jaké přiřazení provést vně podmínky, použijeme:

```
// algoritmus
if (prvniCislo > druheCislo) {
  vetsiZCisel = prvniCislo;
}
if (prvniCislo < druheCislo) {
  vetsiZCisel = druheCislo;
}
```

zde jsou dva problémy:

- relace < není opakem (logickou negací) relace >
  - pro stejně velká čísla se neprovede žádné přiřazení
- protože je proměnná `vetsiZCisel` nastavena v podmínce, překladač soudí, že by nemusela být nastavena vůbec a program nepřeloží:


```
variable vetsiZCisel might not have been initialized
```

vylepšené řešení je:

```
int vetsiZCisel = 0;

// algoritmus
if (prvniCislo > druheCislo) {
  vetsiZCisel = prvniCislo;
}
if (prvniCislo <= druheCislo) {
  vetsiZCisel = druheCislo;
}
```

3.3.1.4. Vhodné řešení pomocí úplné podmínky

místo inverzní podmínky použijeme klíčové slovo `else`:

```
int vetsiZCisel = 0;

// algoritmus
if (prvniCislo > druheCislo) {
  vetsiZCisel = prvniCislo;
}
else {
  vetsiZCisel = druheCislo;
}
```

3.3.1.5. Nevhodné řešení pomocí zanořených podmínek

tentýž problém se pokusíme řešit pro tři čísla

použitím předchozího postupu je program řešitelný, ale značně nabývá na složitosti

```
int nejvetsiZCisel = 0;

// algoritmus
if (prvniCislo >= druheCislo) {
  if (tretiCislo >= prvniCislo) {
    nejvetsiZCisel = tretiCislo;
  }
  else {
    nejvetsiZCisel = prvniCislo;
  }
}
else {
  if (tretiCislo >= druheCislo) {
    nejvetsiZCisel = tretiCislo;
  }
  else {
    nejvetsiZCisel = druheCislo;
  }
}
```

uvedený postup je nevhodný:

- pro každé další číslo se rozsah programu zdvojnásobí – složitost programu roste exponenciálně
- každé další číslo vyžaduje další paměťovou proměnnou – program je lineárně paměťově závislý
- program hledá maximum pro pevný počet čísel daný při **zápisu algoritmu** – potřebujeme, aby bylo možné zadat počet čísel až **po spuštění programu**

3.3.1.6. Převedení exponenciální složitosti programu na lineární

současně též odstraníme lineární paměťovou náročnost

```
int posledneZadanaHodnota;
int dosudNejvetsiHodnota = 0;

System.out.print("Zadej cele cislo: ");
posledneZadanaHodnota = sc.nextInt();
if (posledneZadanaHodnota > dosudNejvetsiHodnota) {
  dosudNejvetsiHodnota = posledneZadanaHodnota;
}

System.out.print("Zadej cele cislo: ");
posledneZadanaHodnota = sc.nextInt();
if (posledneZadanaHodnota > dosudNejvetsiHodnota) {
  dosudNejvetsiHodnota = posledneZadanaHodnota;
}

System.out.print("Zadej cele cislo: ");
posledneZadanaHodnota = sc.nextInt();
if (posledneZadanaHodnota > dosudNejvetsiHodnota) {
  dosudNejvetsiHodnota = posledneZadanaHodnota;
}

// závěrečný tisk
System.out.println("Nejvetsi cislo = " + dosudNejvetsiHodnota);
```

počátečním inicializací `dosudNejvetsiHodnota = 0;` omezujeme program jen na kladná čísla – později si ukážeme jiné řešení

3.3.1.7. Použití cyklu

místo opakujícího se kódu použijeme programové opakování – cyklus

```
int posledneZadanaHodnota;
int dosudNejvetsiHodnota = 0;

do {
  System.out.print("Zadej cele cislo (0 = konec): ");
  posledneZadanaHodnota = sc.nextInt();
  if (posledneZadanaHodnota > dosudNejvetsiHodnota) {
    dosudNejvetsiHodnota = posledneZadanaHodnota;
  }
} while (posledneZadanaHodnota != 0);
```



```
    // závěrečný tisk
    System.out.println("Nejvetsi cislo = " + dosudNejvetsiHodnota);
```

toto řešení ještě není ideální (s ukončovací hodnotou pracuje jako s platnou hodnotou), ale ideálu se blíží

3.3.1.8. Problém magických čísel

Poznámka

1. Tento problém se netýká sekvencí, podmínek ani cyklů.
2. Číselným (nebo znakovým) konstantám použitým přímo v textu programu se říká **magická čísla**. Snahou programátora je se jich důsledně zbavit pomocí **pojmenovaných konstant** (též **symbolických konstant**).
3. Speciálním případem pojmenovaných konstant je **výčtový typ** – viz dále.

- ukončující `0` může být nevýhodná (může to být jedno z porovnávaných čísel) a pak je v programu nutné ji měnit na třech místech
- lepším řešením je použít pojmenovanou konstantu `KONEC`
  - z mnoha důvodů (zde zatím jen důvod lepší čitelnosti a viditelnosti) umísťujeme pojmenovanou konstantu v programu hned za jméno třídy a dáváme jí prefixy `final static`
    - zapomeneme-li na `static`, program nelze přeložit a kompilátor hlásí např.:

```
Sekvence9.java:10: non-static variable KONEC cannot be referenced from ►
a static context
```

- současně program upravíme tak, aby šel používat i pro záporná čísla
  - proměnnou `dosudNejvetsiHodnota` nebudeme inicializovat hodnotou `0` (jako dříve), ale hodnotou `Integer.MIN_VALUE`
    - to je nejmenší možné číslo (viz dříve) a jakékoliv jiné námi zadané číslo bude větší (případně stejné)
    - zápis `Integer.MIN_VALUE` je ukázkou pojmenované konstanty z knihovny Java – konstanta `KONEC` používá stejný princip vytvoření pojmenované konstanty

```
import java.util.*;

public class Sekvence9 {
  final static int KONEC = 0;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int posledneZadanaHodnota;
    int dosudNejvetsiHodnota = Integer.MIN_VALUE;

    do {
      System.out.print("Zadej cele cislo ("
```

```
                    + KONEC + " = konec): ");
      posledneZadanaHodnota = sc.nextInt();
      if (posledneZadanaHodnota > dosudNejvetsiHodnota) {
        dosudNejvetsiHodnota = posledneZadanaHodnota;
      }
    } while (posledneZadanaHodnota != KONEC);

    // závěrečný tisk
    System.out.println("Nejvetsi cislo = " + dosudNejvetsiHodnota);
  }
}
```

vypíše např.:

```
Zadej cele cislo (0 = konec): 8
Zadej cele cislo (0 = konec): 3
Zadej cele cislo (0 = konec): 5
Zadej cele cislo (0 = konec): 0
Nejvetsi cislo = 8
```

3.3.2. Podmínka – příkaz if

- v závislosti na splnění podmínky se provede jen určitá část programu
  - **podmínka** je logický výraz uzavřený do závorek
    - obsahuje relační operátory `> < <= >= != ==`

      `(i <= 2)`

    - často je podmínka složena z více logických výrazů, které jsou pak spojeny logickými operátory `&&` a/nebo `||`

      `(i <= 2  &&  i != 0)`

    - má booleovskou hodnotu `true` – „podmínka splněna“ nebo `false` – „podmínka nesplněna“
- `if` má dvojí tvar
  - **neúplná podmínka**
    - při splněné podmínce se provede část kódu
    - při nesplněné podmínce se neprovede nic

```
boolean mamPenize = true;  // případně false
int pocetPiv = 0;
if (mamPenize == true) {
  pocetPiv++;
}
System.out.println("Vypil jsem " + pocetPiv + " piv");
```


Poznámka

Za `if` vždy použijte složený příkaz, tj. `{ }` – vyhnete se tím mnoha problémům.

Znak profesionality není napsat co nejúspornější zdrojový kód, ale zdrojový kód, ve kterém se každý vyzná.

  - **úplná podmínka**
    - při splněné podmínce se provede jedna část kódu a druhá ne
    - při nesplněné podmínce se provede druhá část kódu a první ne

```
boolean buduRidit = true;  // případně false
int pocetPiv = 0;
int pocetLimonad = 0;
if (buduRidit == false) {
  pocetPiv++;
  System.out.println("Neridim - vypil jsem "
                     + pocetPiv + " piv");
}
else {
  pocetLimonad++;
  System.out.println("Ridim - vypil jsem "
                     + pocetLimonad + " limonad");
}
```

- do `if` lze vložit libovolný příkaz, tedy i další `if`

```
if (dalSiPivo == true) {
  if (desitku == true) {
    pocetDesitek++;
  }
}
else {
  pocetLimonad++;
}
```

  - tento program lze zdánlivě stejně napsat i pomocí složené podmínky

```
if (dalSiPivo == true  &&  desitku == true) {
  pocetDesitek++;
}
else {
  pocetLimonad++;
}
```

    ale funkčnost je trochu jiná – snaha vyhýbat se složeným podmínkám

    - použít místo nich zanořené `if` – program se lépe ladí
- příkaz `else-if` – používá se při několikanásobném rozhodování
  - máme-li více rozhodování

1. použijeme více `if` za sebou

   - pomalejší
   - méně možností

```
char c = 'B';
if (c >= 'A'  &&  c <= 'Z') {
  System.out.println("Velke pismeno " + c);
}
if (c >= 'a'  &&  c <= 'z') {
  System.out.println("Male pismeno " + c);
}
if (c >= '0'  &&  c <= '9') {
  System.out.println("Cislice " + c);
}
```

2. použijeme konstrukci `else-if`

   - rychlejší – neprochází zbytečně nesplnitelné podmínky
   - možnost závěrečného `else`

```
if (c >= 'A'  &&  c <= 'Z') {
  System.out.println("Velke pismeno " + c);
}
else if (c >= 'a'  &&  c <= 'z') {
  System.out.println("Male pismeno " + c);
}
else if (c >= '0'  &&  c <= '9') {
  System.out.println("Cislice " + c);
}
else {
  System.out.println("Interpunkce " + c);
}
```

- příklady
  - při složených podmínkách důsledně závorkovat

```
int rok = 2004;
if (rok % 4 == 0
    && ((rok % 100 != 0)  ||  (rok % 400 == 0))) {
  System.out.println("Prestupny rok " + rok);
}
```

  - prohození obsahu proměnných

```
if (x < y) {
  int pom = x;
  x = y;
  y = pom;
}
```

chybou je:

```
if (x < y) {
  x = y;
  y = x;
}
```

3.3.3. Cykly

- používáme, pokud je výhodné, aby se sekvence příkazů opakovala
- terminologie
  - **řídící proměnná cyklu** – proměnná, na které závisí ukončení cyklu
    - v ideálním stavu existuje pouze jedna
  - **podmínka ukončující cyklus** – logický výraz obsahující řídící proměnnou cyklu
  - **hlavička cyklu** – klíčové slovo `for` nebo `while` a výraz(y) v následujících `( )`
    - hlavička představuje nutnou administrativu
  - **tělo cyklu** – příkazy, které potřebujeme opakovaně provést
    - tělo představuje výkonný kód

Poznámka

Pokud je tělo všech tří dále uvedených cyklů tvořeno jen jedním příkazem, nevyžaduje překladač (narozdíl od přednášejícího ;-) kolem těla cyklu `{ }`.

Dobrá rada – **zásadně toto zjednodušení nepoužívat**. Pokud tělo cyklu uzavřete vždy do `{ }`, tzn. považujete jej vždy za složený příkaz, vyhnete se mnoha problémům.

3.3.3.1. Cyklus for

- vhodný v případě, že předem známe omezující kriteria, tedy:
  - počáteční nastavení řídící proměnné
  - koncovou hodnotu řídící proměnné
  - způsob ovlivnění řídicí proměnné po každé otáčce cyklu
- testuje podmínku cyklu před vykonáním těla cyklu
  - tělo cyklu proběhne, pokud je podmínka splněna
  - cyklus nemusí proběhnout ani jednou
- obecná struktura

```
for (inicializace; ukončující podmínka; změna řídící proměnné) {

tělo cyklu

}
```

- **typický** zápis cyklu `for`

```
final static int MAX = 10;
for (int i = 1;  i <= MAX;  i++) {
  System.out.println(i);
}
```

Poznámka

Syntaxe příkazu `for` umožňuje zápis mnoha různými dalšími způsoby. Dobrá rada – nepoužívat je (proto jejich příklady zde nebudou uváděny). Nejhorší „prohřešek“ je změna řídící proměnné v těle cyklu (viz též dále).

- výpočet faktoriálu

```
public class Faktorial {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej cele cislo: ");
    int n = sc.nextInt();
    int f = 1;
    for (int i = 2;  i <= n;  i++) {
      f *= i;
    }
    System.out.println("f! = " + f);
  }
}
```

- změna řídící proměnné nemusí být jen inkrementace
  - program počítá součin všech lichých čísel

```
final static int MAX = 9;
int soucin = 1;
for (int i = 3;  i <= MAX;  i += 2) {
  soucin *= i;
}
```

- typická a velmi nepříjemná chyba – středník navíc za hlavičkou cyklu
  - příkaz výpisu není ve skutečnosti příkazem v těle cyklu, ale samostatný příkaz za cyklem, proto proběhne pouze jednou
  - tato chyba je mnohem pravděpodobnější, pokud nepoužíváte `{ }` okolo těla cyklu


```
final static int MAX = 10;
for (int i = 1;  i <= MAX;  i++) ; {
  System.out.println("ahoj");
}
```

3.3.3.2. Cyklus while

- vhodný, závisí-li ukončovací podmínka na nějakém příkazu v těle cyklu
  - nevíme dopředu, kolikrát cyklus proběhne
  - typická akce je čtení nějakých hodnot až do jejich vyčerpání
- testuje podmínku cyklu před vykonáním těla cyklu
  - tělo cyklu proběhne, pokud je podmínka splněna
  - cyklus nemusí proběhnout ani jednou
- obecná struktura

```
správné nastavení řídící proměnné

while (ukončující podmínka) {

tělo cyklu, ve kterém se mění řídící proměnná

}
```

- zápis cyklu `while` – načítání čísel, dokud není zadána ukončující nula

```
System.out.println("Zadavej cela cisla, 0 = konec: ");
int i = sc.nextInt();
while (i != 0) {
  System.out.println(i * 2);
  i = sc.nextInt();
}
```

- **typický zápis cyklu** `while` – načítání čísel, dokud není zadána ukončující nula
  - využívá se toho, že přiřazení je výraz
  - závorky kolem přiřazení jsou nutné – bez nich chyba překladu

```
System.out.println("Zadavej cela cisla, 0 = konec: ");
int i;
while ((i = sc.nextInt()) != 0) {
  System.out.println(i * 2);
}
```

Poznámka

I když lze cyklus `for` vždy nahradit cyklem `while`, zásadně to neděláme. Každý má svou jasnou oblast použití.

3.3.3.3. Cyklus do-while

- podobný cyklu `while`
  - používá se mnohem méně často
- testuje podmínku cyklu **po** vykonání těla cyklu
  - tělo cyklu proběhne znovu, pokud je podmínka splněna
  - cyklus je opuštěn při nesplněné podmínce
  - cyklus proběhne minimálně jednou
- zápis cyklu `do-while` – načítání čísel, dokud není zadána ukončující nula

```
System.out.println("Zadavej cela cisla, 0 = konec: ");
int i;
do {
  i = sc.nextInt();
  System.out.println(i * 2);
} while (i != 0);
```

3.3.3.4. Příkazy break a continue

- mění podmínky zpracování cyklu nezávisle na řídící proměnné
  - lze je použít ve všech uvedených typech cyklů
  - pokud jsou cykly vnořeny do sebe, působí na ten cyklus, ve kterém jsou uvedeny
- `break` – ukončuje (ihned opouští) cyklus
  - prakticky se používá pro
    - předčasné ukončení cyklu, např. při výskytu chyby
    - řádné ukončení nekonečného cyklu (viz dále)
- `continue` – skáče na konec těla cyklu (tj. přeskočí zbytek těla cyklu) a tím vynutí další iteraci smyčky – cyklus neopouští
  - používá se mnohem méně než `break`, příklad viz dále
- program načítá celá čísla, pokud jsou kladná, vypíše jejich dvojnásobek, záporných si nevšímá, nula ukončí cyklus
  - je použit nekonečný cyklus `while(true)`


```
System.out.println("Zadavej cela cisla, 0 = konec: ");
while (true) {
  int i = sc.nextInt();
  if (i < 0) {
    continue;    // dalsi nacitani
  }
  if (i == 0) {
    break;       // konec cyklu
  }
  System.out.println(i * 2);
}
```

- ukázka použití `continue` – zamezí mnohonásobnému vnoření podmínek
  - mnohonásobné vnoření podmínek se při ladění stává „noční můrou“

```
System.out.println("Zadavej cela cisla, 0 = konec: ");
int i;
while ((i = sc.nextInt()) != 0) {
  if (i < 0) {
    continue;
  }
  if (i % 2 == 0) {
    continue;
  }
  System.out.println("Liche kladne: " + i);
}
```

bez použití `continue` jsou podmínky obrácené a program je kratší; pokud by bylo ale podmínek více a výkonná část delší (zde je jen jeden příkaz `System.out.println(...)`), bylo by zanoření neúnosné a ladění by bylo obtížnější

```
System.out.println("Zadavej cela cisla, 0 = konec: ");
int i;
while ((i = sc.nextInt()) != 0) {
  if (i > 0) {
    if (i % 2 != 0) {
      System.out.println("Liche kladne: " + i);
    }
  }
}
```

3.3.3.5. Příklady použití cyklů a ukončovací podmínky

- budeme vždy zpracovávat posloupnost celých čísel načítaných z klávesnice
  - zpracování bude představovat součet těchto čísel
- algoritmus zapsaný v pseudojazyce je

```
suma = 0
dokud nejsou přečtena všechna čísla {
```

```
    číslo = načti číslo
    suma = suma + číslo
  }
  výpis suma
```

- možnosti, jak určit, zda jsou již přečtena všechna čísla
  1. počet čísel bude vždy stejný, např. 5
  2. počet sčítaných čísel bude určen prvním načteným číslem, které nebude sumováno
  3. počet čísel je určen **koncovou zarážkou**, tj. domluvenou hodnotou, která se nemůže mezi regulérně zpracovávanými čísly vyskytnout – zde 0
- formální struktura vstupních dat
  1. $c_1\ c_2\ c_3\ c_4\ c_5$
  2. $n\ c_1\ c_2 \ldots c_n$
  3. $c_1\ c_2 \ldots c_k\ 0$ kde $c_i \neq 0$
- řešení 1. – pevný počet dat
  - vstupní data: $c_1\ c_2\ c_3\ c_4\ c_5$
  - zde je typické použití cyklu `for`
  - počet dat je **nezbytné** zadat pomocí pojmenované konstanty – zde `MAX`
    - pro jiný počet vstupních dat je sice nutné změnit zdrojový kód, ale jen na jednom přesně definovaném místě

```
public class PosloupnostPevnyPocet {
  final static int MAX = 5;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int suma = 0;
    System.out.println("Zadej " + MAX
                       + " celych cisel");
    for (int i = 1;  i <= MAX;  i++) {
      int cislo = sc.nextInt();
      suma += cislo;
    }
    System.out.println("Suma = " + suma);
  }
}
```

- řešení 2. – počet dat zadán jako první číslo
  - vstupní data: $n\ c_1\ c_2 \ldots c_n$

- opět typické použití cyklu `for`
- program je „pružnější“ – pro jiný počet čísel není nutno měnit zdrojový kód

```
public class PosloupnostZadanyPocet {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n;
    int suma = 0;
    System.out.print("Zadej pocet cisel: ");
    n = sc.nextInt();
    System.out.println("Zadej " + n
                       + " celych cisel");
    for (int i = 1;  i <= n;  i++) {
      int cislo = sc.nextInt();
      suma += cislo;
    }
    System.out.println("Suma = " + suma);
  }
}
```

- řešení 3. – data ukončena zarážkou
  - vstupní data: $c_1\ c_2 \ldots c_k\ 0$ kde $c_i \neq 0$
  - typické použití cyklu `while`
  - velký problém je zvolit vhodnou hodnotu zarážky
    - pokud bychom např. chtěli program modifikovat na počítání aritmetického průměru, hodnota zarážky 0 by přestala vyhovovat

```
public class PosloupnostZarazka {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int suma = 0;
    System.out.println("Zadavej cela cisla,"
                       + " 0 = konec: ");
    int cislo;
    while ((cislo = sc.nextInt()) != 0) {
      suma += cislo;
    }
    System.out.println("Suma = " + suma);
  }
}
```

- cykly mohou být vnořené
  - u cyklů `for` zcela běžná záležitost

```
final static int MAX_I = 3;
final static int MAX_J = 10;

for (int i = 0;  i < MAX_I;  i++) {
  for (int j = 1;  j <= MAX_J;  j++) {
    int k = (i * MAX_J) + j;
    System.out.format("%2d ", k);
  }
  System.out.println();  // odradkovani na konci cyklu j
}
```

vypíše

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

- u cyklů `while` je třeba dbát zvýšené opatrosti, aby cyklus skončil
  - ♦ v případě zanořování dvou a více cyklů `while` je dobré se zamyslet, zda vnořený cyklus není spíše typu `for`

3.3.3.6. Konečnost cyklů

■ jedna ze zásadních podmínek algoritmu – **konečnost** – se nejsnáze poruší chybou v cyklu

- cyklus má špatně nastavenou podmínku ukončení
  - ♦ program se „zacyklí“ a je nutné jej násilně ukončit, v krajním případě vypnutím počítače

■ základní pravidlo – řídící proměnná cyklu se musí při každé otáčce cyklu změnit

- dodržujeme-li doporučené použití `for` a `while`, je porušení tohoto pravidla obtížné a většinou je způsobeno překlepem

```
for (int i = 0;  i < MAX_I;  i++) {
  for (int j = 1;  i <= MAX_J;  j++) {   // zde je chyba
    int k = (i * MAX_J) + j;
    System.out.format("%2d ", k);
  }
  System.out.println();
}
```

■ druhé pravidlo – do cyklu se musí vstoupit s jasně definovanými počátečními podmínkami

- to opět splňují typizovaná použití `for` a `while`
- čím nestandardněji napsaný cyklus, tím vyšší pravděpodobnost chyby
  - ♦ příklad porušuje několik zásad
    - – řídící proměnná cyklu je deklarována vně cyklu a do cyklu přichází s nejasnou hodnotou (deklarace a nastavení `i` mohou proběhnout mnohem dříve)

    - – není použit relační operátor `<=` ale `!=` což společně s nevhodným počátečním nastavením (11) způsobí, že cyklus proběhne cca 4 miliardkrát (než se `i` přetečením dostane na hodnotu 10)
    - – řídící proměnná cyklu není měněna v hlavičce cyklu, ale v jeho těle
      - je to na špatně viditelném místě
      - při pozdější změně těla cyklu může příkaz `i++` zcela zmizet a cyklus se stane nekonečným

```
int i = 11;
for (; i != 10; ) {
  System.out.print(i++);
}
```

Poznámka

Je možné porušit beztrestně každou zásadu, pokud víme, co děláme. Porušíme-li jich ale více najednou, pravděpodobnost problémů se prudce zvyšuje.

Výstraha

Každé porušení zásady vyžaduje důsledný komentář, ve kterém je vysvětlen důvod.

■ cyklus nemusí proběhnout ani jednou – opět je problém ve vstupních podmínkách

- proměnná `cislo` je deklarovaná daleko od hlavičky cyklu
- je inicializovaná na běžnou hodnotu 0, která je ale v tomto případě nešťastná
- hodnota `cislo` se mění až v těle cyklu, kam se program nedostane

```
int suma = 0;
int cislo = 0;
System.out.println("Zadavej cela cisla,"
                   + " 0 = konec: ");
while (cislo != 0) {
  cislo = sc.nextInt();
  suma += cislo;
}
System.out.println("Suma = " + suma);
```

vypíše:

```
Zadavej cela cisla, 0 = konec:
Suma = 0
```

3.3.4. Přepínač – příkaz switch

■ umožňuje několikanásobné větvení programu – **větev přepínače**

■ nepoužívá se příliš často

■ základní tvar příkazu

```
switch (výraz) {
  case konstanta_1 :
    příkazy_1;
    break;
        . . .
  case konstanta_n :
    příkazy_n;
    break;

  default :
    příkaz_def;
    break;
}
```

- konstanty jsou stejného typu, jako výraz – nejčastěji `int` nebo `char`

Výstraha

Nelze napsat výčet konstant nebo jejich interval. Tyto případy je nutné řešit sekvencí `else-if` – příklad viz dále

- příkazy představují složené příkazy, které se neuzavírají do `{ }`
- sémantika (zjednodušeně)
  - vypočte se hodnota výrazu
  - nalezne se konstanta odpovídající vypočtené hodnotě
  - provedou se všechny příkazy v této větvi
  - program opouští přepínač
  - není-li nalezena odpovídající konstanta, provedou se příkazy ve větvi `default`
- typický příklad
  - je zde vidět „neduh“ přepínače – potřeba velké administrativy
  - na druhou stranu je naprosto jasné, co se děje v každé větvi

```
public class Switch {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej pocet piv: ");
    int pp = sc.nextInt();
    switch (pp) {
      case 1 :
        System.out.println("" + pp + " pivo");
        break;

      case 2 :
        System.out.println("" + pp + " piva");
```



```
        break;

      case 3 :
        System.out.println("" + pp + " piva");
        break;

      case 4 :
        System.out.println("" + pp + " piva");
        break;

      default :
        System.out.println("" + pp + " piv");
        break;
    }
  }
}
```

- zkrácený příklad – jen pro pokročilé
  - vynecháme-li ve větvi `break`, program vykonává všechny příkazy další větve

```
public class SwitchZkracene {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Zadej pocet piv: ");
    int pp = sc.nextInt();
    switch (pp) {
      case 1 :
        System.out.println("" + pp + " pivo");
        break;

      case 2 :
      case 3 :
      case 4 :
        System.out.println("" + pp + " piva");
        break;

      default :
        System.out.println("" + pp + " piv");
        break;
    }
  }
}
```

- tentýž příklad za použití sekvencí `else-if`
  - program je kratší, otázka je, zda je stejně dobře čitelný

```
public class SwitchElseIf {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
```

```
      System.out.print("Zadej pocet piv: ");
      int pp = sc.nextInt();
      if (pp == 1) {
        System.out.println("" + pp + " pivo");
      }
      else if (pp >= 2  &&  pp <= 4) {
        System.out.println("" + pp + " piva");
      }
      else {
        System.out.println("" + pp + " piv");
      }
    }
  }
```

Poznámka

Příkaz `break` ve `switch` je naprosto stejný jako `break` v cyklech. Stane-li se, že je `switch` v cyklu nebo naopak cyklus v některé větvi `switch`, ukončí `break` provádění jemu bližšího příkazu.

3.4. Ladění

Motto: Program se podařilo přeložit bez chyby, takže musí také správně fungovat. `;-)))`

- kromě nejtriviálnějších programů je nutné programy ladit, tzn. odstraňovat jejich chybné či nesprávné chování
  - činnost se nazývá ladění nebo častěji *debugging* a používaný nástroj je *debugger*
- vyšší fází ladění je **testování** – podrobnosti např. v KIV/ZSWI

3.4.1. Metoda ladících výpisů

- používáme, nemáme-li k dispozici debugger
  - na vhodná místa programu vložíme `System.out.println()` pomocí nějž vypisujeme obsahy důležitých proměnných
    - tak lze zjistit, jakými cestami program běžel a analýzou hodnot vypsaných proměnných umíme najít i důvod pro tuto činnost
    - tato metoda je stále používaná pro svoji jednoduchost
    - zásadní nevýhodou je, že musíme měnit zdrojový kód
      - před odladěním přidávat
      - po odladění ubírat
- ukázka na příkladu výpočtu funkce $e^x = 1 + x^1/1! + x^2/2! + x^3/3! + x^4/4! + ...$
  - pro urychlení výpočtu nepočítáme faktoriály a mocniny vždy znovu, ale využijeme iterační výpočet:

$člen_0 = 1$

$člen_1 = člen_0 * x / 1$

$člen_2 = člen_1 * x / 2$

$člen_3 = člen_2 * x / 3$

- pro zvýšení přehlednosti výpisů se vyplatí „pohrát“ si s formáty výpisů, zde:

```
System.out.format("i=%2d, clen=%.3f, suma=%.3f%n", i, clen, suma);
```

```
import java.util.*;

public class LadiciVypisy {
  final static double EPS = 1e-3;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.println("Iteracni vypocet e**x");
    System.out.print("Zadej x: ");
    int x = sc.nextInt();
    // ladici vypis
    System.out.println("x=" + x);

    // vypocet
    double suma = 1.0;
    double clen = 1.0;
    int i = 1;

    while (clen > EPS) {
      clen *= (double) x / i;
      suma += clen;
      // ladici vypis
      System.out.format("i=%2d, clen=%.3f, suma=%.3f%n", i, clen, suma);
      i++;
    }

    System.out.println("Vysledek: " + suma);

    // ladici vypisy
    System.out.println("Funkci:   " + Math.exp(x));
    System.out.println("Presnost: " + Math.abs(Math.exp(x) - suma));
  }
}
```

vypíše např.:

```
Iteracni vypocet e**x
Zadej x: 2
x=2
i= 1, clen=2,000, suma=3,000
```

```
i= 2, clen=2,000, suma=5,000
i= 3, clen=1,333, suma=6,333
i= 4, clen=0,667, suma=7,000
i= 5, clen=0,267, suma=7,267
i= 6, clen=0,089, suma=7,356
i= 7, clen=0,025, suma=7,381
i= 8, clen=0,006, suma=7,387
i= 9, clen=0,001, suma=7,389
i=10, clen=0,000, suma=7,389
Vysledek: 7.388994708994708
Funkci:   7.38905609893065
Presnost: 6.138993594273501E-5
```

3.4.2. Využití grafického debuggeru

- současné vývojové nástroje dávají k dispozici mnohem komfortnější možnosti pro ladění – debuggery
- běžné možnosti debuggeru:
  - nastavení míst přerušení programu (*breakpoints*)
    - jsou to speciálně označené řádky zdrojového kódu
    - pokud by program měl začít zpracovávat tuto řádku, zastaví se
  - výpisy všech dostupných proměnných
    - lokální proměnné se vypisují automaticky všechny
    - nelokální proměnné (může jich být mnoho) jsou vypisovány „na žádost“
  - krokování programu řádku po řádce
    - vynikající při zkoumání komplikovanějších podmínek
  - krokování dovnitř podprogramů (*Step into*) nebo provedení podprogramu jako jeden příkaz (*Step over*)
  - nastavení přerušení při přístupu nebo při změně proměnné
  - a mnoho dalších
- použití debuggeru významně urychluje ladění programu a je vhodné se naučit alespoň základní možnosti
  - doporučovaný RAD nástroj Eclipse má debugger na vysoké úrovni

3.5. Výčtový typ

Poznámka

Součástí Javy od JDK 1.5.

- je to sofistikovanější řešení pojmenovaných konstant (aneb jak se vyhnout magickým číslům)
  - výhodný v případech, kdy:
    - pojmenované konstanty spolu souvisejí
    - na hodnotách konstant nám nezáleží
- kontrapříklad – nevhodné použití pojmenovaných konstant

```
public class PojmenovaneKonstanty {
  final static int ASISTENT         = 1;
  final static int ODBORNY_ASISTENT = 2;
  final static int DOCENT           = 3;
  final static int PROFESOR         = 4;

  public static void main(String[] args) {
    int vyucujici = DOCENT;

    if (vyucujici == DOCENT  ||  vyucujici == PROFESOR) {
      System.out.println("Muze vest doktorandy");
    }

    System.out.println("vyucujici: " + vyucujici);

    vyucujici *= 2;    // nesmysl, ale mozny
    System.out.println("vyucujici: " + vyucujici);
  }
}
```

vypíše:

```
Muze vest doktorandy
vyucujici: 3
vyucujici: 6
```

- na kontrapříkladu jsou vidět nevýhody tohoto řešení
  - je třeba přidělit čísla konstantám („Proč má být ASISTENT jednička?“)
    - možné problémy při budoucích změnách
  - při výpisu se zobrazí pouze nic neříkající číslo (zde 3)
  - konstrukce není typově bezpečná, lze s ní provádět nesmyslné operace (zde např. násobení dvěma)
- řešení pomocí výčtového typu
  - používáme klíčové slovo `enum` (od *enumeration* – výčet, vyjmenování)
  - v programu `enum` umísťujeme (stejně jako pojmenovanou konstantu) hned za název třídy
  - můžeme přidat specifikátory `public` a/nebo `static` (zdůvodnění později)
  - pro název výčtového typu platí stejná pravidla jako pro název třídy (první písmeno názvu velké)

- jednotlivé položky výčtového typu se píší velkými písmeny, neinicializují se žádnými čísly a na jejich pořadí nezáleží
- použití v programu je `JménoVýčtovéhoTypu.JMÉNO_POLOŽKY`

■ vhodnější řešení předchozího příkladu

```
public class PouzitiEnum {
  enum Vyucujici {ASISTENT, ODBORNY_ASISTENT,
                  DOCENT, PROFESOR};

  public static void main(String[] args) {
    Vyucujici vyucujici = Vyucujici.DOCENT;

    if (vyucujici == Vyucujici.DOCENT
       ||  vyucujici == Vyucujici.PROFESOR) {
      System.out.println("Muze vest doktorandy");
    }

    System.out.println("vyucujici: " + vyucujici);

/* nelze prelozit
    vyucujici *= 2;
    System.out.println("vyucujici: " + vyucujici);
*/
  }
}
```

vypíše

```
Muze vest doktorandy
vyucujici: DOCENT
```

■ nutnost používat dvojici `JménoVýčtovéhoTypu.JMÉNO_POLOŽKY`, se může zdát zbytečná, ovšem pouze do okamžiku, než použijeme více výčtových typů se stejně pojmenovanými položkami

```
public class ViceEnum {
  enum Karty {CERVENA, ZELENA, KULE, ZALUDY};
  enum Barvy {MODRA, ZLUTA, CERVENA, ZELENA};

  public static void main(String[] args) {
    Karty k = Karty.CERVENA;
//    Barvy b = Karty.CERVENA;     // nelze
    Barvy b = Barvy.CERVENA;

    System.out.println("karta: " + k);
    System.out.println("barva: " + b);
  }
}
```

vypíše

```
karta: CERVENA
barva: CERVENA
```


Poznámka

Všechny uváděné informace ukazují jen nejzákladnější použití `enum`. Výčtový typ `enum` umožňuje mnohem širší využití.

Kapitola 4. Metody

4.1. Důvody použití

- jedna ze základních zásad v programování – „jednu věc programovat pouze jednou“

1. napíšeme-li (koncepčně chybně!) rozsáhlý kód (netriviální programy jsou rozsáhlé), je:
   - málo přehledný
   - obsahuje části, které
     - jsou zcela shodné
     - jsou funkčně shodné (liší se od sebe navzájem jen v detailech)
   - špatně se ladí
   - špatně se mění či opravuje

Příklad 4.1. Odstrašující příklad opakujícího se kódu

```
public class PravouhlyTrojuhelnikBlok {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    sc.useLocale(Locale.US);

    // prvni cteni
    double d1;
    do {
      System.out.print("Zadej odvesnu: ");
      d1 = sc.nextDouble();
      if (d1 <= 0) {
        System.out.println("Chybne zadani");
        System.out.println("Zadej kladne realne cislo");
      }
    } while (d1 <= 0);

    // druhe cteni
    double d2;
    do {
      System.out.print("Zadej odvesnu: ");
      d2 = sc.nextDouble();
      if (d1 <= 0) {         // chyba z kopirovani
        System.out.println("Chybne zadani");
        System.out.println("Zadej kladne realne cislo");
      }
    } while (d2 <= 0);

    double prepona = Math.sqrt(d1 * d1 + d2 * d2);
    System.out.println("Prepona je: " + prepona);
  }
}
```

Příklad 4.2. Lepší řešení využívající metody

```
public class PravouhlyTrojuhelnikMetody {
  public static double nactiOdvesnu(Scanner sc) {
    double d;
    do {
      System.out.print("Zadej odvesnu: ");
      d = sc.nextDouble();
      if (d <= 0) {
        System.out.println("Chybne zadani");
        System.out.println("Zadej kladne realne cislo");
      }
    } while (d <= 0);

    return d;
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    sc.useLocale(Locale.US);

    double d1 = nactiOdvesnu(sc);
    double d2 = nactiOdvesnu(sc);

    double prepona = Math.sqrt(d1 * d1 + d2 * d2);
    System.out.println("Prepona je: " + prepona);
  }
}
```

2. netriviální programy jsou rozsáhlé – většinou nejsme schopni je napsat jako jednu sekvenci kódu

   - využíváme dříve zmíněné dekompozice a metody shora dolů – tj. od globálního problému k menším a menším částem

- výhody použití metod
  - program se snáze navrhuje – díky metodě shora dolů, což také umožňuje vytvářet program po krocích
  - zdrojový kód je přehlednější
    - soustřeďujeme se jen na jednu konkrétní část algoritmu
  - program se lépe ladí, zejména za použití debugerů
    - i bez použití debugerů se chyba snáze lokalizuje – např.: „je to problém vstupu“
  - změny či opravy jsou pouze na jednom místě
  - řadu knihovních metod již stejně používáme (např.: `println()` nebo `nextDouble()`), tak proč nepoužívat i vlastní
- nevýhody použití metod
  - zvýší se administrativa – je nutné navíc dodat:
    - hlavičku metody
    - návratovou hodnotu přes `return`
  - metodám je často třeba předat parametry (tj. upřesnění použití)
  - zpočátku se použití metod zdá jako komplikovanější a pomalejší způsob

Poznámka

Metod je více druhů – obecné označení pro všechny je **podprogram**. Pokud metoda vrací hodnotu, říká se jí též **funkce**. Nevrací-li metoda hodnotu, nazývá se **procedura**. Příklady viz dále.

4.2. Terminologie

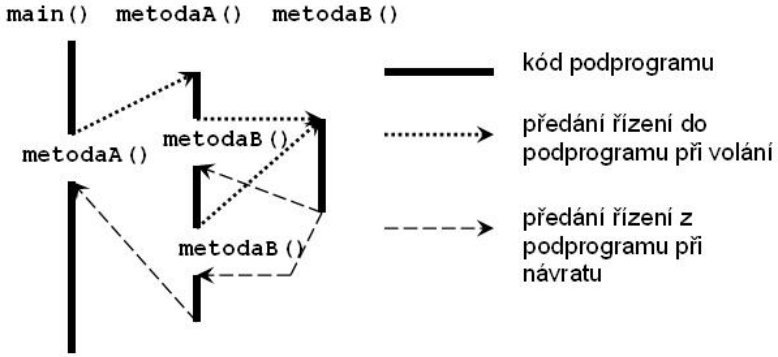
- použijeme-li výše uvedený příklad, pak:
  - `public static double nactiOdvesnu(Scanner sc)`

    je **hlavička metody** v **deklaraci funkce**, přičemž:
    - `public` je **přístupové právo** – podrobnosti později
    - `static` je označení „kategorie“ metody – podrobnosti později
      - bez `static` nelze program přeložit
    - `double` je typ návratové hodnoty
      - ideální metoda totiž provede jeden ucelený úsek činnosti a vrátí jednu hodnotu jako svůj celkový výsledek
    - `nactiOdvesnu` – jméno metody (dodržovat konvenci pojmenování!)
    - `Scanner sc` – deklarace **formálního parametru**
      - hodnota předávaná do metody zvnějšku
      - v tomto případě pro nás nemá (jako parametr) praktický význam (viz dále)
  - **tělo funkce** (výkonný kód) – program, který je zapsán mezi `{ }`
  - `return d;` – vrácení **návratové hodnoty**
    - výsledkem činnosti metody je hodnota dočasně uložená v proměnné `d`
  - `double d1 = nactiOdvesnu(sc);`

    je **volání metody**, přičemž:
    - návratová hodnota metody se uloží do proměnné `d1`
    - `sc` je **skutečný parametr** volání metody
      - jeho případná změna by ovlivnila (parametrizovala) chování metody – příklad viz dále

- metoda `main()` je **volající** a metoda `nactiOdvesnu(sc);` je v ní **volaná**
  - ♦ často je volaná metoda současně i metodou volající

    např. metoda `nactiOdvesnu()` volá ve svém těle metody `nextInt()` a `println()`
  - ♦ toto zřetězení volání není ničím omezeno
- principiální schéma předávání řízení při volání metod
  - ♦ metoda `main()` volá metodu `metodaA()`
  - ♦ metoda `metodaA()` volá ve svém těle na dvou různých místech metodu `metodaB()`



- metody nemohou být **vnořené**
  - ♦ metoda nesmí ve svém těle obsahovat deklaraci jiné metody
  - ♦ Pozor – deklarace není volání – viz předchozí příklad volání
  - ♦ chybně

```
public static int nactiASectiDveCisla() {
  public static int nactiCislo() {    // chyba !!!
    int i = sc.nextInt();
    return i;
  }
  int c1 = nactiCislo();
  int c2 = nactiCislo();
  return c1 + c2;
}
```

- místo deklarace metod je víceméně libovolné
  - volaná metoda (zde `tisk()`) může být uvedena před metodou volající (zde `main()`), což je častější případ


```
public class VolanaPredVolajici {
  public static void tisk(int n) {
    System.out.println("n = " + n);
  }

  public static void main(String[] args) {
    tisk(5);
  }
}
```

- volaná metoda může být uvedena za metodou volající

```
public class VolanaZaVolajici {
  public static void main(String[] args) {
    tisk(5);
  }

  public static void tisk(int n) {
    System.out.println("n = " + n);
  }
}
```

- příkazů `return` může být v metodě více – nemusí být jen jeden jako poslední příkaz metody
  - pak má praktický význam, je-li je `return` součástí podmínky (nebo méně často i cyklu)
    - ♦ v tomto případě provedení `return` znamená předčasný návrat z metody

```
public class ViceReturn {
  public static void tiskKladnych(int n) {
    if (n <= 0) {
      return;
    }
    System.out.println("n = " + n);
  }

  public static void main(String[] args) {
    tiskKladnych(-5);
    tiskKladnych(5);
  }
}
```

4.3. Příklad dekompozice a různých způsobů volání

- v předchozím příkladě se metoda snadno nalezla – opakovaný kód
- je běžné, že se metody použijí, aniž se kód opakuje

- vodítkem může být, že kód metody nemá být delší než jedna obrazovka a má provádět jednu jasně ohraničenou (definovanou) činnost
- v následujícím případě řešíme na sobě nezávisle
  - ♦ **vstup** – `nactiPrirozeneCislo()`

    metoda bez parametrů – častý případ, kdy metoda provádí stále stejnou činnost, kterou není třeba parametrizovat (viz též dále)
  - ♦ **výpočet** – `faktorial(int n)`

    metoda s jedním formálním parametrem

    – narozdíl od předchozího příkladu má parametr skutečné opodstatnění
  - ♦ **hlavní program** – `main()` jen volá obě metody

```
public class FaktorialMetody {
  public static int nactiPrirozeneCislo() {
    Scanner sc = new Scanner(System.in);
    sc.useLocale(Locale.US);
    int n;
    do {
      System.out.print("Zadej prirozene cislo: ");
      n = sc.nextInt();
      if (n <= 0) {
        System.out.println("Chybne zadani: "
                        + n + " neni prirozene cislo");
      }
    } while (n <= 0);

    return n;
  }

  public static int faktorial(int n) {
    int f = 1;
    for (int i = 2;  i <= n;  i++) {
      f *= i;
    }
    return f;
  }

  public static void main(String[] args) {
    int n = nactiPrirozeneCislo();
    int nf = faktorial(n);
    System.out.println("" + n + "! = " + nf);
  }
}
```

4.3.1. Různé způsoby volání metod

■ každá metoda zvlášť


```
public static void main(String[] args) {
  int n = nactiPrirozeneCislo();
  int nf = faktorial(n);
  System.out.println("" + n + "! = " + nf);
}
```

- nejzřejmější – doporučeno pro začátečníky
- dají se dobře vypisovat ladící výsledky (hodnoty mezikroků) a tím určit, ve které metodě je případná chyba

```
int n = nactiPrirozeneCislo();
System.out.println("n = " + n);  // ladici vypis
int nf = faktorial(n);
System.out.println("" + n + "! = " + nf);
```

■ metoda volá jinou metodu jako svůj skutečný parametr

```
public static void main(String[] args) {
  int nf = faktorial(nactiPrirozeneCislo());
  System.out.println("" + nf);
}
```

- kratší zápis
- méně přehledné

■ vše v jednom – trojnásobné zanoření volání

```
public static void main(String[] args) {
  System.out.println("" + faktorial(nactiPrirozeneCislo()));
}
```

- velmi kompaktní zápis na hranici čitelnosti
- málo přehledné – pro začátečníky nedoporučené

4.3.2. Skutečné parametry metody

■ různé možnosti skutečných parametrů

- metoda `int faktorial(int n)` může mít jako svůj skutečný parametr (při volání)
  - ♦ literál – použijeme nejčastěji při ladění metody

    ```
    int fn = faktorial(5);
    ```
  - ♦ pojmenovanou konstantu

    ```
    int fn = faktorial(MAX);
    ```
  - ♦ proměnnou – nejčastěji

    ```
    int fn = faktorial(n);
    ```

- ♦ volání jiné metody – běžné

```
int fn = faktorial(nactiPrirozeneCislo());
```

- ♦ obecný výraz – rozmyslet si předem, zda má smysl

```
int fn = faktorial((n + MAX) * 8 + nactiPrirozeneCislo());
```

  - – třeba dbát zvýšené opatrnosti, aby byl výraz správně
  - – lepší je použít pomocnou proměnou pro vyhodnocení výrazu

```
int pom = (n + MAX) * 8 + nactiPrirozeneCislo();

int fn = faktorial(pom);
```

- obecně platí, že skutečný parametr musí být stejného typu, jako formální parametr
  - jinak je nutné použít přetypování

```
int fn = faktorial((int) 5.0);
```

4.3.3. Způsob předání parametrů z volající do volané

- parametry (primitivních datových typů) se předávají vždy **hodnotou**
  - způsob předání nelze zvolit ani ovlivnit
- proto bylo v předchozích způsobech lhostejné, zda je skutečným parametrem proměnná či konstanta
- předání probíhá takto:
  - (je-li to nutné) vypočte se hodnota výrazu představujícího skutečný parametr
  - tato hodnota se automaticky uloží do datové struktury nazývané **zásobník**
    - ♦ je-li skutečným parametrem proměnná, je do zásobníku uložena její kopie
  - pak je vyvolána metoda, která ve svém těle umí ze zásobníku hodnotu číst
    - ♦ pokud se ji pokusí změnit, je to možné, ale změna se neprojeví ve volající metodě

typická chyba

```
public class ZmenaParametru {
  public static void zmenParametr(int n) {
    n = 5;   // nesmysl
  }

  public static void main(String[] args) {
    int i = 3;
    System.out.println("i = " + i);
    zmenParametr(i);
    System.out.println("i = " + i);
  }
}
```


vypíše:

```
i = 3
i = 3
```

- výhody předávání hodnotou
  - skutečným parametrem může být i výraz
- nevýhody
  - parametr je ve funkci jen vstupní

4.3.4. Typická chyba – parametry jsou považovány za deklaraci

- začátečníky občas zmate, že formální parametry nemají při svém prvním použití hodnotu
  - tuto hodnotu se snaží načíst nebo nastavit přiřazovacím příkazem
    - ♦ to není syntaktická chyba – formální parametr lze změnit (ale téměř nikdy se to nedělá)
    - ♦ je to logická chyba, protože se zničí hodnota předávaná v budoucnu při volání přes skutečný parametr

```
public static int faktorial(int n) {
  n = 5;          // logicka chyba
  int f = 1;
  for (int i = 2;  i <= n;  i++) {
    f *= i;
  }
  return f;
}
```

4.4. Parametry a návratová hodnota

- metody se nejčastěji rozlišují podle
  - počtu (formálních) parametrů
  - návratové hodnoty

4.4.1. Metody bez parametrů

- nemá-li metoda formální parametry, opakuje svůj algoritmus beze změny
- výhodné pro jednoduché aktivity typu `nacteniCisla()`
- v hlavičce metody jednoduše parametry neuvedeme
  - je ale nutné uvést prázdné kulaté závorky

- smysluplné příklady jsou často různé vstupy, např. výše:

```
public static int nactiPrirozeneCislo() {
```

- volání je:

```
int i = nactiPrirozeneCislo();
```

  - prázdné kulaté závorky je opět nutné uvést

4.4.2. Metody bez návratové hodnoty

- nazývají se též **procedury**
  - typickou procedurou je `main()`
- používají se pro aktivity, které jsou provedením ukončené
  - do volající metody se nevrací žádná hodnota, tj. výsledek činnosti volané metody
  - typicky jsou to algoritmy pro výpis, kde provedená činnost je výstup informace
  - místo návratového typu se použije klíčové slovo `void` (prázdný, nulitní)

```
public static void vypisNasobekPi(int n) {
  System.out.println("" + n + " * Pi = " + n * Math.PI);
}
```

- volání je:

```
public static void main(String[] args) {
  final int MAX = 10;
  for (int i = 1;  i <= MAX;  i++) {
    vypisNasobekPi(i);
  }
}
```

4.4.3. Metody bez parametrů a bez návratové hodnoty

- spojení předchozích dvou možností
- smysluplné použití jen zřídka

```
public static void vypisOblibenyPozdrav() {
  System.out.println("ahoj");
}
```

- volání je:

```
vypisOblibenyPozdrav();
```


4.4.4. Metody s více formálními parametry

- metoda může mít „neomezené“ množství parametrů
  - prakticky metoda nesmí mít více než pět parametrů (lépe tři parametry)
    - má-li jich více, jedná se o špatnou dekompozici problému – metoda se snaží najednou provést více činností než jednu
- parametry se navzájem oddělují čárkami
- pojmenování parametrů musí být významové, např.:
  - správně:

```
public static int podil(int delenec, int delitel) {
```

  - chybně – mnohem hůře použitelnější pro budoucího uživatele metody

```
public static int podil(int a, int b) {
```

- jsou-li parametry stejného typu, musí se typy opakovat, např. chybně:

```
public static int podil(int delenec, delitel) {
```

- na pořadí parametrů nezáleží – mělo by ale odpovídat logice věci
  - chybně: `void vypisDatum(int mesic, int rok, int den)`
  - správně: `void vypisDatum(int den, int mesic, int rok)`
    - při volání může být každý skutečný parametr jiného charakteru, např.:

```
vypisDatum(dnesniDen, RIJEN, 2005);
```

4.4.5. Metoda s více formálními parametry různého typu

- parametry mohou být různých datových typů
  - na pořadí opět nezáleží
  - typy se mohou opakovat
  - platí všechna již uvedená pravidla
- např.:

```
void vypisZaokrouhlene(double cislo, int pocetMist) {
```

4.5. Lokální proměnné versus statické proměnné

- proměnné se dělí podle místa deklarace

4.5.1. Lokální proměnné

- proměnné deklarované v metodě
  - jejich viditelnost je pouze v metodě, ve které jsou deklarovány a to od místa deklarace
  - navíc při deklaraci ve složeném příkazu je proměnná viditelná jen do `}`

```
if (x < y) {
  int pom = x;
  x = y;
  y = pom;
}               // konec viditelnosti pom
int k = pom;    // chyba
```

- chceme-li použít (hodnotu) lokální proměnné ve volané metodě, musíme ji předat jako skutečný parametr

4.5.2. Statické proměnné

- proměnné deklarované mimo jakoukoliv metodu
  - typicky je místo deklarace ihned za hlavičkou třídy
  - při deklaraci se musí použít klíčové slovo `static`
- výhody
  - jsou viditelné ve všech metodách
    - metody je mohou číst i nastavovat
    - tím se vlastně dají vyloučit formální parametry
      - náhrada formálních parametrů bez rozmyslu je velmi špatná začátečnická taktika!
- nevýhody
  - jsou viditelné ve všech metodách
    - metody je mohou libovolně nastavovat – zvýšená možnost chyby, která se špatně hledá
- statické proměnné se používají jen pro minimální počet proměnných
  - musí být významově pojmenovány
    - neexistuje, aby se jmenovaly např.: `i k pom neco`
  - je-li jich více stejného typu, každá musí být deklarována na samostatné řádce, případně doplněná komentářem
    - špatně

```
static int celkovyPlat, sumaPenez;
```


    - dobře

```
static int celkovyPlat = 0;
static int sumaPenez;        // celková výše účtu
```

  - musí být nainicializovány
    - typicky v `main()`
    - případně ve speciální inicializační metodě, je-li kód inicializace delší
    - jedná-li se o primitivní datové typy, může být inicialiace v místě deklarace (viz výše: `celkovyPlat = 0;`)
  - ostatní metody je pokud možno jen čtou
    - změnit jejich hodnotu (je-li to nutné) má dovoleno jen jedna speciální metoda

Výstraha

Je třeba si uvědomit, že při použití statických proměnných nás chrání překladač mnohem méně, než u lokálních proměnných!

Příklad 4.3. Příklad rozumného použití

- řešíme problém zbytečného formálního parametru z prvního příkladu
  - proměnná `scanner` je inicializována v samostatné metodě `nastavScanner()`
  - do metody `nactiOdvesnu()` se již nemusí předávat jako formální parametr

```
public class PravouhlyTrojuhelnikStatickePromenne {
  static Scanner scanner;

  public static void nastavScanner() {
    scanner = new Scanner(System.in);
    scanner.useLocale(Locale.US);
  }

  public static double nactiOdvesnu() {
    double d;
    do {
      System.out.print("Zadej odvesnu: ");
      d = scanner.nextDouble();
      if (d <= 0) {
        System.out.println("Chybne zadani");
        System.out.println("Zadej kladne realne cislo");
      }
    } while (d <= 0);

    return d;
  }

  public static void main(String[] args) {
    nastavScanner();
    double d1 = nactiOdvesnu();
    double d2 = nactiOdvesnu();

    double prepona = Math.sqrt(d1 * d1 + d2 * d2);
    System.out.println("Prepona je: " + prepona);
  }
}
```

Poznámka

Vedlejší výhodou tohoto řešení je i fakt, že objekt Scanneru existuje pouze jednou. To bude důležité v případě domácích úloh při kontrole programu validátorem.

Příklad 4.4. Odstrašující příklad chybného použití

- statické proměnné jsou použité bez rozmyslu jen pro zdánlivé ulehčení práce
  - přes statické proměnné lze vracet i návratovou hodnotu metody
    - zde proměnná `f`
    - jako začátečníci nikdy nedělat!
  - je zde smícháno vše dohromady, nikdo není za nic odpovědný
    - proč se to vyskytuje?
      - kód vypadá kratší a elegantnější
      - hrubá neznalost základních principů

```
public class FaktorialKatastrofa {  // NIKDY timto zpusobem !!!
  static int i, f = 1, n;

  public static void faktorial() {
    for (i = 2;  i <= n;  i++) {
      f *= i;
    }
  }

  public static void main(String[] args) {
    n = 5;
    faktorial();
    System.out.println("" + n + "! = " + f);
  }
}
```

4.5.3. Zastínění statických proměnných lokálními

- pokud dodržujeme konvence, že statických proměnných je málo a významově pojmenované, tento **problém nenastane**
- porušíme-li konvence, pamatujeme si pravidlo „bližší košile než kabát“
  - lokální **zastiňuje** (*hide*) statickou

```
public class Metody {
  static int i = 5;
  static void tiskni() {
    double i = 6.5;
    System.out.println(i);           // tiskne 6.5
  }
```

4.5.4. Trocha teorie – přidělování paměti proměnným

- přidělení paměti znamená určení adresy umístění proměnné v paměti počítače
  - provádí automaticky kompilátor a JVM
- statické proměnné
  - paměť je přidělena na začátku programu
  - je k dispozici až do konce programu
- lokální proměnné a formální parametry metod
  - paměť je přidělena v okamžiku vstupu do metody
  - je k dispozici jen do konce metody
    - po opuštění metody je proměnná nedostupná a její obsah se nenávratně ztrácí
    - paměť je uvolněna pro jiné použití
  - paměť se přiděluje ze **zásobníku** (*stack*)
    - při opakovaném volání metody se paměť může přidělit na jiném místě
- zásobník se též nazývá fronta **LIFO** (*last in, first out* – „poslední dovnitř, první ven“)
  - zásobník je často konstruován jako **vratný** – roste k nižším adresám

Příklad 4.5. Příklad přidělování paměti

```
public static void main(String[] args) {
  int a = 8;  // 1
  f();        // 6
}

static void f() {
  int b = 6;  // 2
  g(b);       // 5
}

static void g(int x) {
  int c = 4;  // 3
  x = c;      // 4   nesmysl
}
```

```
                     c (4)     c (4)
                     x (6)     x (4)
           b (6)     b (6)     b (6)     b (6)
  a (8)    a (8)     a (8)     a (8)     a (8)     a (8)
  args     args      args      args      args      args
  1        2         3         4         5         6
```


4.6. Přetěžování metod

- *overloading of names*
- jako začátečníci nebudeme potřebovat
  - musíme ale znát princip, protože se jedná o běžný jev z Java Core API, např. `print()` je v Java Core API přetížená celkem devětkrát – pro všechny základní datové typy a navíc pro typ `Object`
- metody se mohou jmenovat stejně, přesněji mají stejná jména, ale různé hlavičky
  - jejich formální parametry se musí lišit jedním z těchto způsobů:
    - počtem – nejlepší
    - typem (nikoliv pojmenováním!) – dobré a časté
    - pořadím – nevhodné
    - kombinacemi těchto způsobů
  - metodu nelze přetížit pouhou změnou typu návratové hodnoty

```
static int ctverec(int i) { return i * i; }

static long ctverec(int i) { return i * i; }  // chyba
```

- většinou se jedná o několik funkčně příbuzných metod
  - provádějí stejnou základní operaci, ale s různými datovými typy

```
static int ctverec(int i) { return i * i; }

static double ctverec(double i) { return i * i; }

static long ctverec(long i) { return i * i; }

public static void main(String[] args) {
  int j = ctverec(5);
  double d = ctverec(5.5);
  long l = ctverec(12345L);
```

  - provádí stejnou základní operaci, ale upřesněně

```
static void tiskPenez(int koruny) {
  System.out.println("Cena: " + koruny + ",-- Kc");
}

static void tiskPenez(int koruny, int halere) {
  System.out.println("Cena: " + koruny + "," +
                     halere + " Kc");
}
```

```
public static void main(String[] args) {
  tiskPenez(12);
  tiskPenez(12, 50);
}
```

- při volání kompilátor vybere tu z metod, která přesně vyhovuje počtu, typům a pořadí skutečných parametrů

4.7. Komentování tříd a metod – pokračování

- jakmile začínáme používat metody (program je rozsáhlý), je nezbytné dokumentovat zdrojový kód pomocí dokumentačních komentářů `/** */`
  - v nich je možné používat pro zvýšení přehlednosti základní značky HTML, např. `<p> <br> <ol> <ul> <li> <code>` apod.
  - u krátkých školních programů může délka dokumentačních komentářů přesáhnout délku zdrojového kódu
- u třídy musí být komentář uveden bezprostředně před řádkou s klíčovým slovem `class`
  - chybně

```
/**
 * Trida pro ukazku zakladnich moznosti dokumentacnich komentaru
 * @author P.Herout
 */
import java.util.*;

public class DokumentacniKomentare {
```

  - správně

```
import java.util.*;

/**
 * Trida pro ukazku zakladnich moznosti dokumentacnich komentaru
 * @author P.Herout
 */
public class DokumentacniKomentare {
```

  - uvádíme vždy stručný účel, pro který byla třída vytvořena
    - dále se často uvádí i autor zdrojového kódu
    - další možnosti, např. `@version`, `@since` viz v dokumentaci k `javadoc`

Poznámka

Způsob, jak komentovat třídu, je opakování z druhé přednášky. Zde je uveden pouze proto, aby se na jednom místě vyskytovala ucelená informace o dokumentačním komentování.

- u každé metody uvádíme
  - stručný účel, pro který byla vytvořena
  - `@param` důsledný popis formálních parametrů – každého parametru zvlášť
  - `@return` – popis návratové hodnoty
  - komentář musí být opět uveden bezprostředně před hlavičkou metody

```
import java.util.*;

/**
 * Trida pro ukazku zakladnich moznosti dokumentacnich komentaru
 * @author P.Herout
 */
public class DokumentacniKomentare {

  /**
    * Nacita prirozene cislo
    * @param sc objekt <tt>Scanner</tt>, ktery cte z klavesnice
    * @return nactene cislo
    */
  public static int nactiPrirozeneCislo(Scanner sc) {
    System.out.print("Zadej prirozene cislo: ");
    int n = sc.nextInt();
    return n;
  }

  /**
    * Nacita realne cislo vetsi nebo rovno zadane dolni mezi
    * <br>
    * desetinny oddelovac je vzdy tecka (.)
    * @param sc objekt <tt>Scanner</tt>, ktery cte z klavesnice
    * @param dolniMez nejmensi cislo, ktere je mozne nacist
    * @return nactene cislo
    */
  public static double nactiRealneCislo(Scanner sc, double dolniMez) {
    double d;
    do {
      System.out.print("Zadej realne cislo >= " + dolniMez + ": ");
      d = sc.nextDouble();
      if (d < dolniMez) {
        System.out.println("Chybne zadani: "
                           + d + " je mensi nez " + dolniMez);
      }
    } while (d < dolniMez);

    return d;
  }

  /**
    * Pocita mocninu
    * @param zaklad zaklad mocniny
```

```
    * @param exponent exponent mocniny
    * @return hodnota mocniny
    */
  public static double vypoctiMocninu(double zaklad, int exponent) {
    double m = Math.pow(zaklad, exponent);
    return m;
  }

  /**
    * Hlavni metoda, ktera:
    * <ul>
    * <li> pripravi <tt>Scanner</tt></li>
    * <li> nacte zaklad</li>
    * <li> nacte exponent</li>
    * <li> vypocte mocninu a vypise vysledek</li>
    * </ul>
    * @param args parametry prikazove radky
    */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    sc.useLocale(Locale.US);

    double d = nactiRealneCislo(sc, 2.0);
    int n = nactiPrirozeneCislo(sc);
    double mocnina = vypoctiMocninu(d, n);
    System.out.println("" + d + "^" + n + " = " + mocnina);
  }
}
```

4.7.1. Generování dokumentace

- pro generování dokumentace se používá program `javadoc`, který je součástí JDK
- základní použití je:

  ```
  javadoc -d adresar-generovanych-souboru *.java
  ```

  - kde `adresar-generovanych-souboru` je úplná nebo relativní cesta, nejčastěji:
    - `javadoc -d ./doc *.java`

      který založí v aktuálním adresáři podadresář `doc` a do něj zkopíruje vygenerované soubory
- `javadoc` generuje i z jednoho jediného `.java` souboru množství HTML souborů
  - důležitý je soubor `index.html`

Poznámka

Toto je opět opakování z druhé přednášky.

Kapitola 5. Pole

5.1. Motivace

- v programování se často vyskytují úlohy, kdy se provádí výpočet opakovaně nad proměnnými stejného datového typu
  - používáme-li pro tyto úlohy samostatné proměnné (jako dosud), je řešení těžkopádné a změny (rozšíření) krajně obtížné

Příklad 5.1. Přečíst teploty naměřené v jednotlivých dnech týdne a pro každou z nich vypsat odchylku od průměrné teploty

```
public class TeplotyJednotlive {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int t1, t2, t3, t4, t5, t6, t7;
    int prumer;
    t1 = sc.nextInt();
    t2 = sc.nextInt();
    t3 = sc.nextInt();
    t4 = sc.nextInt();
    t5 = sc.nextInt();
    t6 = sc.nextInt();
    t7 = sc.nextInt();
    prumer = (t1 + t2 + t3 + t4 + t5 + t6 + t7) / 7;
    System.out.println("1. " + t1 + " " + (t1 - prumer));
    System.out.println("2. " + t2 + " " + (t2 - prumer));
    System.out.println("3. " + t3 + " " + (t3 - prumer));
    System.out.println("4. " + t4 + " " + (t4 - prumer));
    System.out.println("5. " + t5 + " " + (t5 - prumer));
    System.out.println("6. " + t6 + " " + (t6 - prumer));
    System.out.println("7. " + t7 + " " + (t7 - prumer));
  }
}
```

- z příkladu je zřejmé:
  - kód se opakuje s minimálními změnami
    - je ale natolik jednoduchý, že řešení pomocí dalších metod (viz dříve) by ke zjednodušení nepomohlo

Příklad 5.2. Nevhodné řešení pomocí metod

```
public class TeplotyJednotliveMetody {
  static Scanner sc = new Scanner(System.in);
  static int prumer;

  public static int nactiTeplotu() {
    return sc.nextInt();
  }

  public static void vypisOdchylku(int poradi, int t) {
    System.out.println("" + poradi + ". "
                       + t + " " + (t - prumer));
  }

  public static void main(String[] args) {
    int t1, t2, t3, t4, t5, t6, t7;
    t1 = nactiTeplotu();
    t2 = nactiTeplotu();
    t3 = nactiTeplotu();
    t4 = nactiTeplotu();
    t5 = nactiTeplotu();
    t6 = nactiTeplotu();
    t7 = nactiTeplotu();
    prumer = (t1 + t2 + t3 + t4 + t5 + t6 + t7) / 7;
    vypisOdchylku(1, t1);
    vypisOdchylku(2, t2);
    vypisOdchylku(3, t3);
    vypisOdchylku(4, t4);
    vypisOdchylku(5, t5);
    vypisOdchylku(6, t6);
    vypisOdchylku(7, t7);
  }
}
```

- pokud by se mělo pracovat s teplotami za celý měsíc nebo rok, program by byl velmi nepřehledný

Poznámka

Toto je jiný typ příkladu, než dřívější zpracování posloupností – ty jsme jen načetli a rovnou průběžně sumovali. S jednotlivými údaji se po přečtení již nikdy nemuselo pracovat.

5.2. Pole

5.2.1. Základní informace

- je to strukturovaný datový typ
  - skládá se z pevně daného počtu **prvků pole** (složek, položek)
    - počet prvků se stanoví při vzniku pole a od této chvíle se počet nesmí měnit
  - prvky jsou zásadně stejného typu – pole je **homogenní** datová struktura
  - prvky se navzájem odlišují pomocí **indexu**
    - celé číslo vyjadřující pořadí prvku v poli
    - v Javě má index vždy hodnoty 0 až počet prvků – 1
  - hodnoty prvků (narozdíl od jejich počtu) se smějí měnit libovolně
- vytvoření pole s názvem `teploty` o `N` prvcích typu `int`

```
static final int N = 7;
int[] teploty = new int[N];
```

- pole může být libovolného datového typu

```
double[] presneTeploty = new double[N];
```

- přístup k prvkům pole pomocí indexu, kterým je typicky proměnná `i` typu `int`

`teploty[i]`

- tento zápis představuje jednu proměnnou typu `int`
  - ♦ má stejné vlastnosti, jako např. proměnná deklarovaná jako: `int t1;`
- místo proměnné `i` může být libovolný celočíselný výraz, např.:

  `teploty[0]` nebo `teploty[i + 1]` atp.
- povolený rozsah indexů je `teploty[0]` až `teploty[N - 1]`
- **překročení mezí pole** (přetečení nebo podtečení indexů) hlídá automaticky JVM
  - ♦ při *`index < 0`* nebo *`index >= N`* výpočet končí výpisem chybového hlášení (výjimkou)

■ každé pole si v sobě nese informaci o své celkové délce

- je uložena v konstantě *`jménoPole`*`.length`, např.:

  `teploty.length`
- představuje celkový počet prvků pole, nikoliv nejvyšší index

■ typický začátek práce s polem je:

```
for (int i = 0; i < teploty.length; i++) {
```

- častou chybou je: `i <= teploty.length;`

Poznámka

Naprosto stejnou funkčnost by v tomto případě měl i příkaz:

```
for (int i = 0; i < N; i++) {
```

který je navíc kratší.

Doporučuji přesto používat první způsob, protože:

1. už z prvního pohledu na hlavičku cyklu je zřejmé, že bude zpracovávat konkrétní pole
2. horní mez pole může být „trochu“ jiná než očekávaná, např.:

   ```
   int[] teploty = new int[N + 1];
   ```

   Standardní zápis `teploty.length` funguje bez problémů vždy.

5.2.2. Řešení odchylek teplot pomocí pole

■ úlohu výpisu odchylek jednotlivých teplot vyřešíme pomocí pole např. takto

- pokud bychom později chtěli počítat průměr měsíčních teplot, znamenalo by to jen úpravu jedné řádky programu:

  `static final int N = 31;`


```
public class TeplotyPole {
  static final int N = 7;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int[] teploty = new int[N];

    // nacitani hodnot
    for (int i = 0;  i < teploty.length;  i++) {
      teploty[i] = sc.nextInt();
    }

    // vypocet prumeru
    int suma = 0;
    for (int i = 0;  i < teploty.length;  i++) {
      suma += teploty[i];
    }

    int prumer = suma / teploty.length;

    // vypis hodnot
    for (int i = 0;  i < teploty.length;  i++) {
      System.out.println("" + (i + 1) + ". "
                         + teploty[i] + " "
                         + (teploty[i] - prumer));
    }
  }
}
```

■ použití pole neznamená, že nesmíme použít metody – naopak, používáme je stejně často

- metody důsledně oddělují jednotlivé části algoritmu
  - ♦ pole lze deklarovat jako statickou proměnnou
    - – odpadne tak nutnost jeho předávání přes formální parametry

      (jiné řešení – přenos přes parametry – viz dále)

```
public class TeplotyPoleMetody {
  static final int N = 7;
  static int[] teploty;

  public static void nacteniHodnot() {
    Scanner sc = new Scanner(System.in);
    for (int i = 0;  i < teploty.length;  i++) {
      teploty[i] = sc.nextInt();
    }
  }

  public static int vypocetPrumeru() {
    int suma = 0;
    for (int i = 0;  i < teploty.length;  i++) {
```

```
      suma += teploty[i];
    }

    return (suma / teploty.length);
  }

  public static void vypisHodnot(int prumer) {
    for (int i = 0;  i < teploty.length;  i++) {
      System.out.println("" + (i + 1) + ". "
                         + teploty[i] + " "
                         + (teploty[i] - prumer));
    }
  }

  public static void main(String[] args) {
    teploty = new int[N];
    nacteniHodnot();
    int prumer = vypocetPrumeru();
    vypisHodnot(prumer);
  }
}
```

5.2.3. Praktické poznámky

5.2.3.1. Počáteční hodnoty pole

- po vzniku pole je zajištěno, že všechny prvky nového pole budou mít implicitní (předem známou, defaultní – *default*) hodnotu a to hodnotu nulovou (pro typ `double` pak 0.0, pro `boolean` pak `false`)
  - nevyhovuje-li nám nulová hodnota nebo pokud bezprostředně po vzniku pole nenásleduje jeho naplnění hodnotami, je vhodné pole inicializovat, např.:

```
public static void vytvoreniANastaveni(int pocatecniHodnota) {
  teploty = new int[N];
  for (int i = 0;  i < teploty.length;  i++) {
    teploty[i] = pocatecniHodnota;
  }
}
```

5.2.3.2. Rozdíl mezi délkou pole a aktuální délkou

- máme-li např. pole o délce tří prvků, např.:

```
int[] pole = new int[3];
```

  - je jeho délka jednou provždy 3
    - můžeme však z tohoto pole využívat jen některé prvky
    - budeme-li např. po určitou dobu práce s polem využívat jen první dva prvky, je **aktuální délka** pole 2

  - Java neposkytuje žádné prostředky pro zjištění aktuální délky
  - je to záležitost jen a jen programátora
  - dobrá rada – vytvářet pole přesně na potřebnou velikost (bez rezerv) a používat jej celé
    - budeme-li mít algoritmus, který vyžaduje pole s proměnnou délkou, řeší se to jiným způsobem, např. pomocí třídy `ArrayList`

5.2.3.3. Inicializované pole

- pole lze vytvořit také výčtem inicializačních hodnot
  - nejčastější použití je jako vzorová data pro usnadnění ladění

```
int[] poleFibon = { 0, 1, 1, 2, 3, 5, 8 };
```

  - velikost pole si překladač vypočte sám z počtu uvedených konstant
- častý omyl je, že toto pole je konstantní, tj. hodnoty nelze měnit
  - není to pravda, lze (nesmyslně) nastavit např.: `poleFibon[1] = -6;`
  - nepomůže ani

```
final int[] poleFibon = { 0, 1, 1, 2, 3, 5, 8 };
```

    - to jen zabrání změně referenční proměnné, nikoli hodnotám pole

5.2.3.4. Výpis všech prvků pole najednou

- tato možnost přichází od JDK 1.5
- využívá možností třídy `java.util.Arrays`
  - jako první příkaz programu musí být:

```
import java.util.*;
```

- používá se nejčastěji jako kontrolní výpis pro ladící účely
- příkaz je:

```
System.out.println(Arrays.toString(teploty));
```

- pole vypíše včetně počátečních a koncových `[ ]` např. jako:

```
[10, 11, 12, 13, 14, 16, 15]
```

5.2.3.5. Použití konstrukce for-each pro pole

- přístup **postupně ke všem prvkům** pole lze zrealizovat i bez indexů
  - tento způsob se používá často pro pole či kontejnery objektů (podrobně viz později)

- pro pole primitivních datových prvků je spíše kuriozitou, kterou nebudeme používat, ale nesmí nás překvapit

■ příklad sečte prvky celého pole

```
public class ForEach {
  public static void main(String[] args) {
    int[] pole = { 1, 3, 5, 7 };
    int suma = 0;

    for (int hodnota : pole) {
      suma += hodnota;
    }

    System.out.println("Suma = " + suma);
  }
}
```

5.2.3.6. Vstup hodnot ze souboru

Poznámka

Podrobně bude tento princip vyložen později v části o souborech.

■ při ladění programů je únavné opakovaně zadávat stále stejné hodnoty prvků polí

- to lze obejít:
  - ♦ dočasným inicializovaným polem – viz výše
  - ♦ přesměrováním vstupu ze souboru

■ ukázka části kódu pro načtení prvků pole

```
import java.util.*;

public class NacteniPole {
  final static int N = 5;

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int[] pole = new int[N];

    for (int i = 0;  i < pole.length;  i++) {
      System.out.print("pole[" + i + "]: ");
      pole[i] = sc.nextInt();
    }

    // ladici kontrolni vypis nacteneho pole
    System.out.println(Arrays.toString(pole));
    System.out.println("... a dalsi zpracovani nacteneho pole");
  }
}
```


po spuštění

```
java NacteniPole
```

vypíše např.:

```
pole[0]: 92
pole[1]: 4
pole[2]: 8
pole[3]: 63
pole[4]: 123
[92, 4, 8, 63, 123]
... a dalsi zpracovani nacteneho pole
```

■ bez toho, že bychom zdrojový kód program jakkoliv upravovali, lze zadávané hodnoty načíst ze souboru

- připravíme si jednorázově editorem textový soubor `hodnoty.txt` s obsahem:

  ```
  92 4 8 63 123
  ```

  za posledním číslem může nebo nemusí být odřádkováno

- spustíme program příkazem:

  ```
  java NacteniPole < hodnoty.txt
  ```

- program nečeká na vstup z klávesnice a rovnou vypíše:

  ```
  pole[0]: pole[1]: pole[2]: pole[3]: pole[4]: [92, 4, 8, 63, 123]
  ... a dalsi zpracovani nacteneho pole
  ```

5.3. Trocha teorie

5.3.1. Princip vytvoření pole

■ deklarace: `int[] abc;`

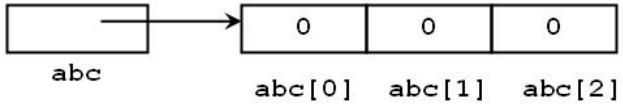
- nevytváří pole (několik stejných prvků ve skupině)
- vytváří **referenční proměnnou pole**, která umí ukazovat na libovolně velké pole typu `int`
  - ♦ použití této proměnné jako pole povede v tomto okamžiku k chybě programu

    ```
    abc[1] = 5; // chyba java.lang.NullPointerException
    ```

■ příkaz: `abc = new int[3];`

- vytvoří (**alokuje**) v paměti pole o třech prvcích typu `int`
  - ♦ prvky mají implicitní nulové hodnoty

- odkaz (adresa) tohoto bloku paměti (pole) se přiřadí do referenční proměnné `abc`
- od této chvíle ukazuje referenční proměnná na pole o konstantním počtu tří prvků



- od této chvíle můžeme pole `abc` používat již známými způsoby, např.:

```
abc[1] = 5;
```

- pole se tedy vytváří v Javě pouze **dynamicky** pomocí operátoru `new`

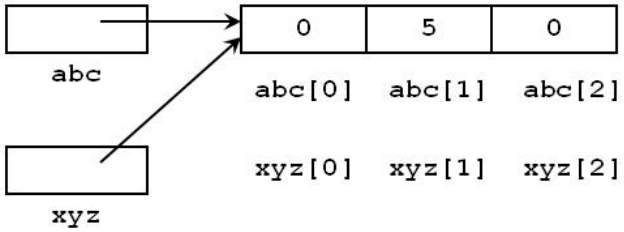
5.3.2. Práce s referenčními proměnnými polí

- protože referenční proměnná pole je proměnná, lze jí během výpočtu měnit hodnotu
  - to většinou není rozumné, protože tím nenávratně ztrácíme pole, na které původně ukazovala

```
int[] abc;
abc = new int[3];
abc[1] = 5;
abc = new int[8];            // zcela nove pole
System.out.println(abc[1]);   // vypise 0
```

  - tato zdánlivě nesmyslná možnost má své opodstatnění
    - na jedno a téže pole (blok paměti) může ukazovat více referenčních proměnných pole
      - stávají se tak synonymy pro přístup k prvkům pole
      - praktické použití je pro předávání polí jako parametrů a návratových hodnot metod (viz dále)

```
int[] abc = new int[3];
abc[1] = 5;
int[] xyz;
xyz = abc;
System.out.println(xyz[1]);   // vypise 5
```



5.4. Pole jako parametr a/nebo návratová hodnota metody

5.4.1. Motivace

- proč se o to snažíme, když stačilo deklarovat referenční proměnnou pole jako statickou proměnnou a pole tak bylo viditelné a měnitelné ve všech metodách?
  - můžeme snadno vytvořit univerzální metody, které pracují s poli různé velikosti
  - v knihovnách Java Core API je mnoho užitečných metod, které takto fungují a my je musíme umět používat

5.4.2. Příklad

- budeme modifikovat program pro zjišťování odchylek teplot
  - odchylky se budou ukládat do samostatného pole
    - to se vytvoří automaticky, dle velikosti pole teplot
    - hodnoty v samostatném poli by v budoucnu umožňovaly snadné výpočty typu
      - hledání maxima a minima
      - průměrná odchylka apod.
  - metoda `int[] vytvoreniPoleANacteniHodnot(int velikostPole)`
    - dostává jako svůj parametr velikost budoucího pole teplot
    - toto pole vytvoří a postupně naplní načtenými hodnotami
    - vrací odkaz na toto pole

```
public static int[] vytvoreniPoleANacteniHodnot(int velikostPole) {
  Scanner sc = new Scanner(System.in);
  int[] pole = new int[velikostPole];
  for (int i = 0;  i < pole.length;  i++) {
    pole[i] = sc.nextInt();
  }
  return pole;
}
```

  - metoda `int[] vypocetOdchylek(int[] poleTeplot)`
    - dostává jako svůj parametr pole teplot
    - nejprve vypočte sumu teplot a jejich průměr
    - pak vytvoří pole odchylek stejné velikosti, jako je pole teplot

- pole odchylek postupně naplní vypočtenými hodnotami
- vrací odkaz na pole odchylek

```
public static int[] vypocetOdchylek(int[] poleTeplot) {
  int suma = 0;
  for (int i = 0;  i < poleTeplot.length;  i++) {
    suma += poleTeplot[i];
  }

  int prumer = suma / poleTeplot.length;
  int[] odchylky = new int[poleTeplot.length];
  for (int i = 0;  i < poleTeplot.length;  i++) {
    odchylky[i] = poleTeplot[i] - prumer;
  }
  return odchylky;
}
```

- metoda `void vypisHodnot(int[] poleT, int[] poleO)`
  - dostává jako své parametry pole teplot a pole odchylek
  - hodnoty obou polí formátovaně vypíše
  - nevrací nic

```
public static void vypisHodnot(int[] poleT, int[] poleO) {
  for (int i = 0;  i < poleT.length;  i++) {
    System.out.println("" + (i + 1) + ". "
                       + poleT[i] + " "
                       + poleO[i]);
  }
}
```

- hlavní program pouze volá připravené metody
  - je zde ukázáno, jak lze bez problémů použít vytvořené metody pro zpracování údajů za měsíc

```
public class TeplotyPoleMetodyParametry {
  static int TYDEN = 7;
  static int MESIC = 31;

  public static int[] vytvoreniPoleANacteniHodnot(int velikostPole) {
    // ...
  }

  public static int[] vypocetOdchylek(int[] poleTeplot) {
    // ...
  }

  public static void vypisHodnot(int[] poleT, int[] poleO) {
    // ...
  }
```



```
  public static void main(String[] args) {
    int[] teplotyTydne = vytvoreniPoleANacteniHodnot(TYDEN);
    int[] odchylkyTydne = vypocetOdchylek(teplotyTydne);
    vypisHodnot(teplotyTydne, odchylkyTydne);

    int[] teplotyMesice = vytvoreniPoleANacteniHodnot(MESIC);
    int[] odchylkyMesice = vypocetOdchylek(teplotyMesice);
    vypisHodnot(teplotyMesice, odchylkyMesice);
  }
}
```

5.4.3. Pole lze v metodě měnit

- když byl předáván do metody skutečný parametr primitivního datového typu, nebyla jeho případná změna v metodě viditelná vně metody
- v případě polí jako formálních parametrů to neplatí!
  - do metody se předá kopie referenční proměnné, která ale ukazuje na původní pole (viz dříve)
  - prvky pole lze v metodě trvale nastavit – změna bude viditelná i po opuštění metody

Příklad 5.3. Trvalá změna hodnoty prvku pole předaného do metody

```
import java.util.*;

public class ZmenaPole {
  public static void zmenaTretihoPrvku(int[] pole) {
    pole[2] = 15;
  }

  public static void main(String[] args) {
    int[] poleFibon = { 0, 1, 1, 2, 3, 5, 8 };
    System.out.println(Arrays.toString(poleFibon));
    zmenaTretihoPrvku(poleFibon);
    System.out.println(Arrays.toString(poleFibon));
  }
}
```

vypíše:

```
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 15, 2, 3, 5, 8]
```

5.5. Pole jako tabulka

- předchozí příklady ukazovaly použití pole pro uložení (časové) posloupnosti údajů
- když uvážíme, že každý prvek pole je neoddělitelně spojen se svým indexem, můžeme pole používat jako jednoduchou tabulku (zobrazení)
  - první (neexistující) sloupec představují indexy – je to hodnota x

- druhý sloupec jsou hodnoty jednotlivých prvků – je to hodnota `y = f(x)`
- je třeba si uvědomit, že první index pole je vždy 0 a indexy rostou po jedné
  - ♦ v případě jiných hodnot je třeba přepočítat indexy – viz dále

■ ve všech třech následujících příkladech budeme zkoumat kvalitu generátoru pseudonáhodných čísel

- ten se nachází v balíku `java.util`, tj. jako první příkaz programu musí být:

```
import java.util.*;
```

- generátor by měl poskytovat čísla v **rovnoměrném rozložení**, tj. každé číslo s přibližně stejnou pravděpodobností
  - ♦ budeme zkoumat četnosti čísel – pro každé vygenerované číslo přičteme jedničku odpovídajícímu prvku pole
  - ♦ v případě kvalitního generátoru (a dostatečného počtu pokusů) se četnosti jednotlivých čísel musejí přibližně rovnat
    - – vždy provedeme 5 milionů pokusů, tj. vygenerujeme 5 milionů pseudonáhodných čísel

```
final static int MAX = 5000000;
```

- generátor připravíme příkazem

```
Random gen = new Random();
```

- pseudonáhodné čísla v daném rozsahu, např.: <0, 4> se generují příkazem:

```
int cislo = gen.nextInt(5);
```

5.5.1. Bezproblémový případ

■ budeme zkoumat četnosti pseudonáhodných čísel 0 až 4

- použijeme pole o 5 prvcích a nebude nutno žádné přepočítávání indexů

```
import java.util.*;

public class CetnostCisel {
  final static int N = 5;
  final static int MAX = 5000000;

  public static void main(String[] args) {
    int[] cetnosti = new int[N];
    Random gen = new Random();
    for (int i = 0;  i < MAX;  i++) {
      int cislo = gen.nextInt(N);
      cetnosti[cislo]++;
    }

    for (int i = 0;  i < cetnosti.length;  i++) {
      System.out.print("" + i + " = "
                       + cetnosti[i] + ", ");
    }
  }
}
```

vypíše např.:

```
0 = 1000278, 1 = 1000178, 2 = 999093, 3 = 1000798, 4 = 999653,
```

■ z výsledů je vidět, že odchylka je menší než 1 % (přesněji +0,8 promile a –0,91 promile)

5.5.2. Posunutí indexů

■ budeme zkoumat četnosti pseudonáhodných čísel 10 až 14

- použijeme pole o 5 prvcích
- indexy bude nutno přepočíst – posunout o 10
  - ♦ při ukládání – odečítáme
  - ♦ při výpisu – přičítáme

```
import java.util.*;

public class CetnostCiselPosunutiIndexu {
  final static int N = 5;
  final static int MAX = 5000000;
  final static int DOLNI_MEZ = 10;

  public static void main(String[] args) {
    int[] cetnosti = new int[N];
    Random gen = new Random();
    for (int i = 0;  i < MAX;  i++) {
      int cislo = (gen.nextInt(N) + DOLNI_MEZ);
      int index = cislo - DOLNI_MEZ;
      cetnosti[index]++;
    }

    for (int i = 0;  i < cetnosti.length;  i++) {
      System.out.print("" + (i + DOLNI_MEZ) + " = "
                       + cetnosti[i] + ", ");
    }
  }
}
```

vypíše např.:

```
10 = 1002007, 11 = 999561, 12 = 999590, 13 = 999685, 14 = 999157,
```

5.5.3. Posunutí indexů a přepočet rozsahu

■ budeme zkoumat četnosti pseudonáhodných čísel 10 až 109 ve skupinách po 20, tj.

10 – 29, 30 – 49, 50 – 69, 70 – 89, 90 – 109

- použijeme pole o 5 prvcích
- indexy bude nutno přepočíst – generované číslo přiřadit skupině

```
import java.util.*;

public class CetnostCiselPrepocet {
  final static int N = 5;
  final static int MAX = 5000000;
  final static int SKUPINA = 20;
  final static int DOLNI_MEZ = 10;
  final static int HORNI_MEZ = DOLNI_MEZ + N * SKUPINA;

  public static void main(String[] args) {
    int[] cetnosti = new int[N];
    Random gen = new Random();
    for (int i = 0;  i < MAX;  i++) {
      int cislo = gen.nextInt(HORNI_MEZ - DOLNI_MEZ);
      cislo += DOLNI_MEZ;
      int index = (cislo - DOLNI_MEZ) / SKUPINA;
//      System.out.println("" + cislo + " = " + index);
      cetnosti[index]++;
    }

    for (int i = 0;  i < cetnosti.length;  i++) {
      System.out.print("" + (i * SKUPINA + DOLNI_MEZ)
                       + "-"
                       + ((i + 1) * SKUPINA + DOLNI_MEZ - 1)
                       + " = "
                       + cetnosti[i] + ", ");
    }
  }
}
```

vypíše např.:

```
10-29 = 1001653, 30-49 = 1000302, 50-69 = 999735, 70-89 = 998393, 90-109 = ▶
999917,
```

5.6. Pole reprezentující množinu

- prvky polí nemusejí být pouze čísla
- použijeme-li pole typu `boolean`, pak hodnota prvku může znamenat
  - `true` – prvek je přítomen v množině
  - `false` – prvek není přítomen v množině
- nejjednodušší případ pak je, že prvek má hodnotu danou jeho indexem

5.6.1. Prvočísla pomocí Eratosthenova síta

- úkol je nalézt všechna prvočísla menší nebo rovna zadanému maximálnímu číslu (MAX)

5.6.1.1. Algoritmus

1. vytvoříme množinu obsahující všechna přirozená čísla od 2 do MAX včetně
2. z množiny vypustíme všechny násobky čísla 2
3. najdeme nejbližší vyšší číslo k předchozímu zpracovávanému a vypustíme všechny jeho násobky
4. opakujeme krok 3. dokud jsou v množině nějaká nezpracovaná čísla
5. čísla, která v množině zůstávají, jsou hledaná prvočísla

Poznámka

Algoritmus lze urychlit, respektive výpočet lze zkrátit, když si uvědomíme, že největší zpracovávané číslo je nejvýše druhá odmocnina z MAX.

5.6.1.2. Řešení

```
public class Prvocisla {
  final static int MAX = 11;

  static boolean[] vytvorANastav(int max) {
    boolean[] mnozina = new boolean[max + 1];
    for (int i = 0;  i < mnozina.length;  i++) {
      mnozina[i] = true;
    }
    return mnozina;
  }

  static void sito(boolean[] mnozina) {
    int mez = (int) Math.sqrt(mnozina.length);
    for (int i = 2;  i <= mez;  i++) {
      if (mnozina[i] == false) {
        continue;
      }
      for (int j = i * 2;  j < mnozina.length;  j += i) {
        mnozina[j] = false;
      }
    }
  }

  static void vypisPrvocisel(boolean[] mnozina) {
    for (int i = 2;  i < mnozina.length;  i++) {
      if (mnozina[i] == true) {
        System.out.print("" + i + ", ");
      }
    }
    System.out.println();
```

```
  }

  public static void main(String[] args) {
    boolean[] prvocisla = vytvorANastav(MAX);
    sito(prvocisla);
    vypisPrvocisel(prvocisla);
  }
}
```

5.7. Vícerozměrná pole

- pole, k jehož prvkům se přistupuje pomocí více než jednoho indexu
- v Javě se s vícerozměrnými poli pracuje jako s jednorozměrnými poli, jejichž prvky jsou opět pole
  - tato znalost má význam při použití sofistikovaných konstrukcí, např. pole s nestejnou délkou řádek apod.

5.7.1. Deklarace

- deklarace referenční proměnné – kolik rozměrů, tolik prázdných `[]`

```
int[][] matice;
```

- vytvoření matice o dvou řádcích a třech sloupcích

```
matice = new int[2][3];
```

Poznámka

Je-li počet řádků a sloupců konstantní (jako v tomto případě), používá se občas označení **pravoúhlé pole**.

- i vícerozměrné pole lze vytvořit výčtem inicializačních hodnot

```
int[][] jednotkovaMatice = {{ 1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
```

- při předání vícerozměrného pole do metody:
  - hodnoty lze v metodě trvale měnit – platí stejná pravidla, jako pro jednorozměrné pole
  - počet řádků se zjistí jako

    jmenoMatice`.length`
  - počet sloupců se zjistí jako

    jmenoMatice`[0].length`

5.7.2. Součet dvou matic

- program umožní zadat počet řádek a sloupců
- pak se zadají obě matice

- nakonec se zadané matice vypíší a vypíše se i jejich součet

```
import java.util.*;
public class SoucetMatic {
  static Scanner sc = new Scanner(System.in);

  static int[][] vytvorANacti(int radky, int sloupce) {
    int[][] m = new int[radky][sloupce];
    System.out.println("Zadejte matici "
                       + radky + " x " + sloupce);
    for (int i = 0;  i < radky;  i++) {
      for (int j = 0;  j < sloupce;  j++) {
        m[i][j] = sc.nextInt();
      }
    }
    return m;
  }

  static void vypis(int[][] m) {
    int radky = m.length;
    int sloupce = m[0].length;
    for (int i = 0;  i < radky;  i++) {
      for (int j = 0;  j < sloupce;  j++) {
        System.out.format("%2d ", m[i][j]);
      }
      System.out.println();
    }
  }

  static int[][] secti(int[][] m1, int[][] m2) {
    int radky = m1.length;
    int sloupce = m1[0].length;
    int[][] ms = new int[radky][sloupce];
    for (int i = 0;  i < radky;  i++) {
      for (int j = 0;  j < sloupce;  j++) {
        ms[i][j] = m1[i][j] + m2[i][j];
      }
    }
    return ms;
  }

  public static void main(String[] args) {
    System.out.print("Zadej pocet radku: ");
    int radky = sc.nextInt();
    System.out.print("Zadej pocet sloupcu: ");
    int sloupce = sc.nextInt();
    int[][] m1 = vytvorANacti(radky, sloupce);
    int[][] m2 = vytvorANacti(radky, sloupce);
    int[][] soucet = secti(m1, m2);
    vypis(m1);
    System.out.println("plus");
    vypis(m2);
    System.out.println("rovna se");
```

```
    vypis(soucet);
  }
}
```

5.7.2.1. Načtení hodnot vstupních matic ze souboru

- při ladění si můžeme se zadáváním hodnot pomoci opět (jako u jednorozměrného pole) využitím souboru se vstupními hodnotami
  - připravíme si jednorázově editorem textový soubor `hodnoty-matic.txt` s obsahem např.:

```
4
5

 1  2  3  4  5
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35

100 100 100 100 100
200 200 200 200 200
300 300 300 300 300
400 400 400 400 400
```

    kde jednotlivá čísla jsou ta, která bychom jinak zadávali po spuštění programu, např. číslo 5 na druhé řádce má význam „počet sloupců“

    ### Poznámka

    Jak je zřejmé, na počtu a umístění bílých znaků nezáleží. Zadáváme je dle uvážení tak, aby byl soubor `hodnoty-matic.txt` co nejpřehlednější.

  - po spuštění příkazem

    `java SoucetMatic < hodnoty-matic.txt`

    vypíše:

```
Zadej pocet radku: Zadej pocet sloupcu: Zadejte matici 4 x 5
Zadejte matici 4 x 5
 1  2  3  4  5
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
plus
100 100 100 100 100
200 200 200 200 200
300 300 300 300 300
400 400 400 400 400
rovna se
101 102 103 104 105
211 212 213 214 215
321 322 323 324 325
431 432 433 434 435
```

Kapitola 6. Řazení

6.1. Základní terminologie

- anglický výraz *sort* má tyto překlady:
  - třídit, roztřiďovat, rozdělovat podle druhů
  - uspořádat, seřadit
- **řazení** – činnost, kdy přeskupujeme prvky (nejčastěji v poli) tak, aby byly v určité posloupnosti, ve které mezi prvky platí vztahy „větší-menší“ či „předchůdce-následník“
  - „seřaď studenty podle data narození“
- **třídění** – činnost, kdy prvky z určité skupiny (množiny) rozdělujeme do podskupin podle nějaké společné vlastnosti
  - „rozřaď (roztřiď) studenty podle měsíců data narození do 12 skupin“
- pojmy řazení a třídění se běžně zaměňují (i v odborné literatuře)
  - většinou však oba znamenají „přeskupit v posloupnosti“
  - zde bude důsledně používán pojem **řazení**
- účelem řazení je (významně) usnadnit pozdější vyhledávání
  - jedná se o základní a téměř univerzálně prováděnou činnost nejen v počítačích
    - „dej si věci do pořádku“ znamená „proveď nějakou vhodnou formu třídění a řazení“
    - uvádí se, že řazení zabere až 25 % výpočetního času zpracování dat
  - protože se jedná o univerzální věc
    - existuje množství rozmanitých algoritmů
      - na nich lze ukázat nejrůznější principy a postupy
        - velmi vhodné pro výuku algoritmizace v programování
      - Java má ve svých knihovnách samozřejmě tuto problematiku více než uspokojivě řešenu – viz dále
- při řazení je důležitý způsob (směr) řazení
  - řazení **vzestupně** – předchozí prvek je menší nebo roven následujícímu prvku
  - řazení **sestupně** – předchozí prvek je větší nebo roven následujícímu prvku
- pro pole primitivních datových prvků v Javě tedy platí
  - vzestupně: `p[i - 1] <= p[i]`, pro `i = 1, ..., p.length - 1`

  - sestupně: `p[i - 1] >= p[i]`, pro `i = 1, ..., p.length - 1`

Poznámka

Pokud dále nebude explicitně uvedeno, že se jedná o sestupné řazení, jedná se implicitně o řazení vzestupné. To je v praxi mnohem běžnější.

- další důležitý fakt je, kde řazení fyzicky probíhá – rozeznáváme
  - **řazení vnitřní** – všechny řazené prvky jsou v jeden okamžik dostupné ve vnitřní paměti
    - zaměříme se pouze na tento způsob („paměti je dost“ – není problém takto řadit pole o velikosti desítek milionů prvků)
  - **řazení vnější** – řazených prvků je extrémní množství a do vnitřní paměti se najednou nevejdou
    - jsou uloženy v jednom, či více souborech na vnější paměti
    - v jeden okamžik je přímo dostupná (tj. ve vnitřní paměti) pouze část prvků
    - tento způsob viz v předmětech KIV/PPA2, KIV/PT, KIV/DB1
- pojem **klíč**
  - dosud jsme předpokládali řazení primitivních datových prvků
  - řadit lze i **záznamy**, tj. datové struktury s více položkami
    - např. záznam o osobě má položky jméno, věk, váha, výška
      - technickou realizaci záznamu v Javě viz později
  - pak při řazení vybereme jednu z položek, kterou pro účely řazení nazveme klíčem, a podle ní řadíme celé záznamy
    - přirozeně chceme, aby údaje o jedné osobě zůstaly pohromadě
    - při použití klíče jsou všechny ostatní položky záznamu pro samotný proces řazení nepodstatné (pojem **hodnotové prvky**)
    - v případě primitivních datových prvků je klíčem celý prvek
  - je-li možné řadit podle více položek, může se jednat o
    - **složený klíč** – použijí se hodnoty více položek najednou
      - např. položkami jsou den, měsíc a rok narození a úkolem je seřadit podle stáří
      - pak se z těchto položek vypočítá jedna společná hodnota
        - v tomto případě a v Javě to bude počet milisekund od 1.1.1970
      - další častý případ je příjmení a jméno
      - podstatné je, že položky by měly mít stejný (fyzikální) význam
        - nemá moc smysl klíč složený z položek výška a věk

♦ **primární**, **sekundární** a **terciální** řazení – postupné řazení

– narozdíl od předchozího případu postupujeme po částech

1. primárním řazením podle jedné (nejdůležitější) položky (klíče) seřadíme pole
2. položky se stejnou hodnotou primárního klíče se tím seskupily a tyto úseky pole řadíme jednotlivě podle sekundárního klíče (druhé nejvýznamnější položky)
3. nevyhovuje-li výsledek řazení (tj. zůstávají-li stejné podskupiny), postup se opakuje pro terciální klíč atd.

– tento způsob je univerzálnější

• klíče pro jednotlivé stupně řazení spolu nemusejí souviset

• proto se používá více a to i v případech, kdy by bylo možné zkonstruovat složený klíč

• **stabilita** řazení – mnohdy velmi důležitý faktor

Poznámka

Má význam pouze při řazení **záznamů**. Při řazení primitivních datových prvků se význam ztrácí.

♦ **metoda řazení je stabilní**, pokud relativní pořadí prvků se stejnou hodnotou klíče zůstává v seřazeném poli stejné jako v původním poli

– bude-li záznam obsahovat položky jméno a váhu a v poli budou sekvenčně záznamy pro členy celé rodiny, pak při stabilním řazení se seskupí položky vždy pro jednoho člena rodiny, ale časová posloupnost zůstane zachována

```
Pavel 80, Karel 75, Petr 45, Pavel 85, Pavel 83, Karel 74
```

po seřazení stabilní metodou

```
Karel 75, Karel 74, Pavel 80, Pavel 85, Pavel 83, Petr 45
```

♦ **metoda řazení je nestabilní**, pokud relativní pořadí prvků se stejnou hodnotou klíče bude v seřazeném poli jiné než v poli původním

– pro předchozí příklad bychom ztratili časovou řadu

```
Pavel 80, Karel 75, Petr 45, Pavel 85, Pavel 83, Karel 74
```

po seřazení nestabilní metodou

```
Karel 74, Karel 75, Pavel 83, Pavel 80, Pavel 85, Petr 45
```

– náhradním řešením by bylo dodat do záznamu položku o čase a podle ní sekundárně řadit

■ **lexikografické** řazení

• při řazení čísel očekáváme výsledek např.:


```
1, 3, 8, 11, 12, 17, 31, 40
```

• řadíme-li ale řetězce (typicky jména souborů) dostaneme výsledek např. (přípona `.txt` je volena náhodně a nemá zde význam):

```
1.txt, 11.txt, 12.txt, 17.txt, 3.txt, 31.txt, 40.txt, 8.txt
```

• toto řazení probíhá tak, že se řetězce řadí nejprve podle hodnoty 1. znaku, tj. `1`

♦ tím se k sobě dostanou (vytvoří se skupiny):

– `1.txt`, `11.txt`, `12.txt`, `17.txt`

– `3.txt`, `31.txt`

– `40.txt`

– `8.txt`

♦ ve druhém a dalších krocích se řadí uvnitř těchto skupin – skupiny se navzájem již nemíchají; jedná se o postupné řazení

– porovnává se znak `.` (tečka) z `1.txt` oproti znaku `1` z `11.txt` atd.

– tečka má menší hodnotu než `1`

♦ v případě stejného obsahu prvních (několika) znaků dvou řetězců je menší kratší řetězec

– např. `ahoj` je „menší“ než `ahojlidi`

• proto se v podobných případech snažíme, aby délka řetězců byla shodná, např. si vypomůžeme nevýznamovými nulami (zde jsou ale pro řazení významové)

```
01.txt, 03.txt, 08.txt, 11.txt, 12.txt, 17.txt, 31.txt, 40.txt
```

Poznámka

V případě řetězců složených jen z ASCII znaků pak nenastávají (většinou) další problémy. V případě řazení řetězců s akcentovanými znaky je situace složitější – záleží na národních zvyklostech. Prostředky Javy je však plně řešitelná i v případě češtiny. Podrobnosti v KIV/JXT.

Poznámka

■ Pro jednoduchost budou principy některých algoritmů řazení vysvětlovány na poli prvků typu `int` a řazení bude vždy vzestupné.

■ Při výkladu jednotlivých algoritmů nebude diskutována jejich efektivita – viz později.

■ Pro všechny algoritmy lze nalézt řadu dílčích vylepšení (některé jsou uvedeny). Pragmaticky vzato – tato vylepšení nemají přílišný praktický smysl a jsou uváděna jen jako ukázka používání základních programátorských „triků“.

6.2. Řazení výběrem – SelectSort

- též **řazení s výběrem mezního prvku** nebo **řazení s přímým výběrem**

6.2.1. Princip řešení

1. v poli nalezneme prvek s nejmenší hodnotou a zapamatujeme si jeho pořadí
2. vyměníme tento prvek s prvkem na prvním místě v poli
3. body 1. a 2. opakujeme pro druhý nejmenší prvek pole a druhé pořadí až pro předposlední nejmenší prvek

```
[5, 1, 4, 2, 3]  // nalezena 1, záměna s 5
[1, 5, 4, 2, 3]  // nalezena 2, záměna s 5
[1, 2, 4, 5, 3]  // nalezena 3, záměna s 4
[1, 2, 3, 5, 4]  // nalezena 4, záměna s 5
[1, 2, 3, 4, 5]
```

6.2.2. Řešení v Javě

- `iMin` je index prvku s minimální hodnotou

```
import java.util.*;

public class RazeniVyberem {  // Select sort
  final static int[] pole = {5, 1, 4, 2, 3};

  public static void main(String[] args) {
    for (int i = 0;  i < pole.length - 1;  i++) {
      int iMin = i;
      for (int j = i + 1;  j < pole.length;  j++) {
        if (pole[j] < pole[iMin]) {
          iMin = j;
        }
      }
      if (i != iMin) {
        int pom = pole[iMin];
        pole[iMin] = pole[i];
        pole[i] = pom;
      }
    }
    System.out.println(Arrays.toString(pole));
  }
}
```

6.3. Řazení vkládáním – InsertSort

- postup používaný při řazení karet

6.3.1. Princip řešení

1. máme již seřazený úsek k prvků
2. vezmeme $(k + 1)$ prvek a v předchozím seřazeném poli nalezneme místo, kam se zařadí
3. zařazení proběhne tak, že se všechny následující prvky posunou o jedno místo dále
4. body 2. a 3. provádíme pro všechny další prvky

Poznámka

Posunutí karet je snadné a jednorázové. Posunutí prvků pole je nutno provést v cyklu – představuje časově náročné výměny prvků.

```
[5, 1, 4, 2, 3]

// zpracovává 1
[5, 5, 4, 2, 3]  posunutí 5
[1, 5, 4, 2, 3]  vložení 1

// zpracovává 4
[1, 5, 5, 2, 3]  posunutí 5
[1, 4, 5, 2, 3]  vložení 4

// zpracovává 2
[1, 4, 5, 5, 3]  posunutí 5
[1, 4, 4, 5, 3]  posunutí 4
[1, 2, 4, 5, 3]  vložení 2

// zpracovává 3
[1, 2, 4, 5, 5]  posunutí 5
[1, 2, 4, 4, 5]  posunutí 4
[1, 2, 3, 4, 5]  vložení 3

[1, 2, 3, 4, 5]
```

6.3.2. Řešení v Javě

- vnitřní cyklus
  - jde od konce k začátku (od vyšších indexů k nižším) – nutné pro posun prvků v poli, abychom je nepřepsali
  - má dvojitou ukončovací podmínku
    - `j >= 0` – dokud jsme v poli
      - tato podmínka se uplatní pouze jednou – viz dále
    - `pole[j] > pom` – dokud je právě zařazovaný prvek menší

Poznámka

Je nezbytné mít podmínku zapsanou jako:

```
j >= 0 && pole[j] > pom
```

nikoliv

```
pole[j] > pom && j >= 0 // chyba
```

protože funguje **zkrácené vyhodnocování logických výrazů**

není-li první část podmínky, tj. `j >= 0` splněna, k vyhodnocování druhé části, tj. `pole[j] > pom` nedojde, a tím ani podtečení indexu

```
import java.util.*;

public class RazeniVkladanim {  // Insert sort
  final static int[] pole = {5, 1, 4, 2, 3};

  public static void main(String[] args) {
    for (int i = 1;  i < pole.length;  i++) {
      int pom = pole[i];
      int j;
      for (j = i - 1;  j >= 0  &&  pole[j] > pom;  j--) {
        pole[j + 1] = pole[j];
      }
      pole[j + 1] = pom;
    }
    System.out.println(Arrays.toString(pole));
  }
}
```

6.3.3. Komplikovanější řešení v Javě

Výstraha

Pouze pro zájemce!

- pokud nás bude zajímat efektivita řešení, lze vynechat dvojitou podmínku ve vnitřním cyklu
  - použijeme programovací trik nazývaný **zarážka**
    - pokud si schováme hodnotu prvního prvku a dočasně ji přepíšeme hodnotou právě zařazovaného prvku, můžeme použít pouze jednoduchou podmínku

      ```
      pole[j] > pom
      ```

    - za urychlení vnitřního cyklu ale zaplatíme složitějším programem s více přesuny
      - tyto přesuny jsou ale ve vnějším cyklu, takže proběhnou nejvýše N krát

```
import java.util.*;

public class RazeniVkladanimSeZarazkou {
  final static int[] pole = {5, 1, 4, 2, 3};

  public static void main(String[] args) {
    for (int i = 1;  i < pole.length;  i++) {
      int pom = pole[i];
      int prvni = pole[0];
      pole[0] = pom;
      int j;
      for (j = i - 1;  pole[j] > pom;  j--) {
        pole[j + 1] = pole[j];
      }
      if (pom > prvni) {
        pole[0] = prvni;
        pole[j + 1] = pom;
      }
      else {
        pole[0] = pom;
        pole[j + 1] = prvni;
      }
    }
    System.out.println(Arrays.toString(pole));
  }
}
```

Poznámka

Urychlení programu v tomto případě je asi 7 %, což je zanedbatelné. Přesto je však užitečné si trik s použitím zarážky pamatovat – je to často používaný trik při práci s poli.

6.4. Řazení zaměňováním – BubbleSort

- též **řazení s přímou výměnou**

6.4.1. Princip řešení

1. porovnáváme dva sousední prvky počínaje posledním prvkem
2. pokud je prvek s nižším indexem větší, prohodíme je
3. končíme na prvním prvku
   - tím se prvek s nejnižší hodnotou dostane na první pozici („vybublá“ nahoru)
   - kromě toho se částečně seřadí zbytek pole, což způsobí, že celkový počet výměn prvků je v průměru menší než u řazení vkládáním
4. opakujeme body 1. až 3. přičemž končíme u druhého prvku v pořadí atd.

```
[5, 1, 4, 2, 3]

// krok 1
[5, 1, 2, 4, 3]  // probublává 2
[1, 5, 2, 4, 3]  // probublává 1
```

```
  // krok 2
  [1, 5, 2, 3, 4]  // probublává 3
  [1, 2, 5, 3, 4]  // probublává 2

  // krok 3
  [1, 2, 3, 5, 4]  // probublává 3

  // krok 4
  [1, 2, 3, 4, 5]  // probublává 4

  [1, 2, 3, 4, 5]
```

6.4.2. Řešení v Javě

- je třeba dávat zvýšený pozor na indexy (zpracování) krajních prvků – meze cyklů
  - program lze napsat více podobnými způsoby

```
import java.util.*;

public class RazeniZamenovanim {  // Bubble sort
  final static int[] pole = {5, 1, 4, 2, 3};

  public static void main(String[] args) {
    for (int i = 1;  i < pole.length;  i++) {
      for (int j = pole.length - 1;  j >= i;  j--) {
        if (pole[j - 1] > pole[j]) {
          int pom = pole[j];
          pole[j] = pole[j - 1];
          pole[j - 1] = pom;
        }
      }
    }
    System.out.println(Arrays.toString(pole));
  }
}
```

6.4.3. Komplikovanější řešení v Javě

Výstraha

Pouze pro zájemce!

- pokud nás bude zajímat efektivita řešení, lze po průchodu vnitřního cyklu např. zjistit, zda v něm proběhla alespoň jedna výměna prvků
  - pokud ne, je pole již seřazené a algoritmus je možné předčasně ukončit
  - to, zda došlo k výměně prvků, zjistíme použitím booleovské proměnné `zamena`


```
import java.util.*;

public class RazeniZamenovanimZkracene {  // Bubble sort
//  final static int[] pole = {5, 1, 4, 2, 3};
  final static int[] pole = {1, 2, 3, 4, 5};

  public static void main(String[] args) {
    for (int i = 1;  i < pole.length;  i++) {
      boolean zamena = false;
      for (int j = pole.length - 1;  j >= i;  j--) {
        if (pole[j - 1] > pole[j]) {
          int pom = pole[j];
          pole[j] = pole[j - 1];
          pole[j - 1] = pom;
          zamena = true;
        }
      }
      if (zamena == false) {
        break;          // pole je serazeno
      }
    }
    System.out.println(Arrays.toString(pole));
  }
}
```

6.5. Prakticky používané řešení

- všechny předchozí způsoby řazení by se v praxi použily jen ve velmi speciálních případech
- pro naprostou většinu běžných použití využíváme řazení z balíku `java.util`
  - jedná se o metodu `sort()` třídy `Arrays`
  - metoda je přetížená
    - `Arrays.sort(pole)` – seřadí celé pole
    - `Arrays.sort(pole, odVcetne, doKrome)` – seřadí část pole od indexu `odVcetne` do indexu `doKrome`
  - používá se algoritmus QuickSort pro pole primitivních datových prvků a MergeSort pro pole objektů (viz KIV/PPA2)
    - algoritmus je velmi rychlý – pole o 10 milionech prvků seřadí za 2,5 sec (Pentium Centrino, 1,5 GHz)

```
import java.util.*;

public class RazeniJavaCas {  // Quick sort
  final static int MAX = 15;
  static int[] pole;

  public static void pripravPole() {
```

```
    pole = new int[MAX];
    for (int i = 0;  i < pole.length;  i++) {
      pole[pole.length - 1 - i] = i + 1;
    }
  }

  public static void main(String[] args) {
    pripravPole();
    System.out.println(Arrays.toString(pole));

//    Arrays.sort(pole);
    Arrays.sort(pole, pole.length/3, (pole.length/3)*2);

    System.out.println(Arrays.toString(pole));
  }
}
```

vypíše:

```
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[15, 14, 13, 12, 11, 6, 7, 8, 9, 10, 5, 4, 3, 2, 1]
```


Kapitola 7. Třída

7.1. Motivační kontrapříklad

Výstraha

Toto je odstrašující příklad použití, jak to nedělat!

Poznámka

Dále bude používán lékařský termín BMI (*Body Mass Index*), který nemá nic společného s programováním. Zde je použit pro příklad výpočtu dat odvozených z dat základních.

BMI se vypočítá jako podíl hmotnosti [kg] a čtverce výšky [m], tj.

BMI = m / (v * v)

Hodnoty BMI:

- < 18 – podváha
- 18 až 25 – ideální hmotnost
- 25 až 30 – nadváha
- > 30 – obezita ohrožující zdraví

- v řazení jsme používali pole primitivních datových typů, např. `int`
- v případě, že chceme *uchovávat* o jednom objektu (např. člověku) více informací najednou (např. jméno, výšku a váhu), můžeme použít více polí
  - pro jméno budeme používat typ `String` (podrobnosti později)

    ```
    String[] jmena = {"Pavel", "Hana", "Karel", "Dana" };
    ```

  - pro váhu typ `double`

    ```
    double[] vahy = {90, 50, 85.5, 70};
    ```

  - pro výšku typ `int`

    ```
    int[] vysky = {186, 170, 190, 175};
    ```

  - údaje o jedné konkrétní osobě budou v těchto třech na sobě nezávislých polích spojeny pořadím, tj. indexem
  - použití může vypadat např.:

    ```
    public class OsobyNevhodne {
      static String[] jmena = {"Pavel", "Hana", "Karel", "Dana" };
      static double[] vahy = {90, 50, 85.5, 70};
      static int[] vysky = {186, 170, 190, 175};
    ```

```
  public static int vypoctiBMI(int poradi) {
    double vyskaMetry = vysky[poradi] / 100.0;
    double bmi = vahy[poradi] / (vyskaMetry * vyskaMetry);
    return (int) Math.round(bmi);  // zaokrouhleni
  }

  public static void vypisOsobu(int poradi) {
    System.out.println(jmena[poradi]
                       + " vazi: " + vahy[poradi] + " [kg],"
                       + " meri: " + vysky[poradi] + " [cm],"
                       + " BMI: " + vypoctiBMI(poradi));
  }

  public static void main(String[] args) {
    vypisOsobu(2);
    // nebo v cyklu
    for (int i = 0;  i < jmena.length;  i++) {
      vypisOsobu(i);
    }
  }
}
```

vypíše:

```
Karel vazi: 85.5 [kg], meri: 190 [cm], BMI: 24
Pavel vazi: 90.0 [kg], meri: 186 [cm], BMI: 26
Hana vazi: 50.0 [kg], meri: 170 [cm], BMI: 17
Karel vazi: 85.5 [kg], meri: 190 [cm], BMI: 24
Dana vazi: 70.0 [kg], meri: 175 [cm], BMI: 23
```

- tento způsob má množství nevýhod
  - nejvážnější je obtížná synchronizace třech nezávislých polí
    - ♦ pokud jednou porušíme synchronizaci, nikdy už ji nejsme schopni zrekonstruovat
    - ♦ jakákoliv změna (přidání, ubrání či přemístění osoby) musí být pečlivě provedena ve všech třech polích
      - pragmaticky
        - přidat osobu do polí nelze, protože pole mají konstantní velikost a jsou zcela zaplněny
        - ubrat osobu by znamenalo její přemístění na konec polí a vynulování všech hodnot a nepoužívání celého rozsahu pole
        - prakticky realizovatelné je tedy jen změna pořadí (tj. přemístění)
    - ♦ typický příklad požadavku na změnu pořadí je řazení
    - ♦ budeme-li uvažovat např. řazení podle váhy, pak
      - při prohození dvou prvků v poli váhy musíme zajistit prohození týchž prvků i v poli jména a v poli výšky

      - tento algoritmus sice lze napsat, ale je krajně nepřehledný
        - navíc bude velmi pomalý, protože se stále přesouvají všechna data

7.1.1. Řešení pomocí pole indexů

Výstraha

Jen pro zájemce!

- jedná se o techniku používanou občas v programování
- princip je nepřesouvat data v polích, což může být časově i organizačně náročné, ale pouze jejich indexy
- zavedeme pomocné pole indexů (zde `poradi`), které je z počátku inicializované posloupností přirozených čísel

```
static String[] jmena = {"Pavel", "Hana", "Karel", "Dana" };
static double[] vahy = {90, 50, 85.5, 70};
static int[] vysky = {186, 170, 190, 175};
static int[] poradi = {0, 1, 2, 3};
```

- jakýkoliv přístup do ostaních polí provádíme přes pole indexů – vnořené indexování

```
System.out.println(jmena[poradi[cislo]]);
```

- při řazení pak všechna pole zůstávají v původním pořadí a mění se jen pole indexů

```
public static void seradPodleVahy() {
  // pouzit SelectSort
  for (int i = 0;  i < vahy.length - 1;  i++) {
    int iMin = i;
    for (int j = i + 1;  j < vahy.length;  j++) {
      if (vahy[poradi[j]] < vahy[poradi[iMin]]) {
        iMin = j;
      }
    }
    if (i != iMin) {
      int pom = poradi[iMin];
      poradi[iMin] = poradi[i];
      poradi[i] = pom;
    }
  }
}

public static int vypoctiBMI(int cislo) {
  double vyskaMetry = vysky[poradi[cislo]] / 100.0;
  double bmi = vahy[poradi[cislo]] / (vyskaMetry * vyskaMetry);
  return (int) Math.round(bmi);  // zaokrouhleni
}

public static void vypisOsobu(int cislo) {
```

```
    System.out.println(jmena[poradi[cislo]]
                       + " vazi: " + vahy[poradi[cislo]] + " [kg],"
                       + " meri: " + vysky[poradi[cislo]] + " [cm],"
                       + " BMI: " + vypoctiBMI(cislo));
  }

  public static void main(String[] args) {
    seradPodleVahy();
    for (int i = 0;  i < jmena.length;  i++) {
      vypisOsobu(i);
    }

    System.out.println("Poradi v polich:");
    System.out.println(Arrays.toString(jmena));
    System.out.println(Arrays.toString(vahy));
    System.out.println(Arrays.toString(vysky));
  }
```

vypíše:

```
Hana vazi: 50.0 [kg], meri: 170 [cm], BMI: 17
Dana vazi: 70.0 [kg], meri: 175 [cm], BMI: 23
Karel vazi: 85.5 [kg], meri: 190 [cm], BMI: 24
Pavel vazi: 90.0 [kg], meri: 186 [cm], BMI: 26
Poradi v polich:
[Pavel, Hana, Karel, Dana]
[90.0, 50.0, 85.5, 70.0]
[186, 170, 190, 175]
```

Poznámka

I přes tento užitečný trik je použitý způsob tří nezávislých polí krajně nešťastný a nebudeme jej používat!

7.2. Třída jako nový datový typ

- v předchozím případě bychom potřebovali mít data o jedné osobě pohromadě v jednom **záznamu** (*record*)
- různé programovací jazyky pro tuto akci dávají k dispozici jazykové konstrukce – např. Pascal `record`, jazyk C `struct` apod.
- Java jako objektově orientovaný jazyk jde koncepčně dále – k dispozici je **třída** (*class*)
  - ta má více možností, než pouhý datový záznam, umožňuje deklarovat:
    - data
    - metody pracující nad těmito daty
  - třída je **šablona** (*template*), pomocí níž se vytvářejí datové objekty typu této třídy
    - vytvářejí se zásadně dynamicky pomocí operátoru `new`

Poznámka

Statické vytváření (opak dynamického) známe již dlouho, např.: `int i;`

- datové objekty se nazývají **objekty** (*objects*) nebo **instance** (*instance*)
- přistupuje se k nim pomocí **referenční proměnné** (podobně jako v polích)

- tříd bývá v jednom programu více a každá třída nemusí obsahovat metodu `main()`
  - typicky `main()` obsahuje pouze jedna třída v programu – organizační podrobnosti viz dále

Poznámka

Dále bude používán pojem „datová třída“, což označuje třídu sloužící především k uchování dat a manipulaci s nimi. Opakem bude „řídicí třída“ (též „hlavní třída“ nebo „třída aplikační logiky“), která bude realizovat vlastní výpočet – ta má v sobě typicky metodu `main()`. Tyto pojmy jsou částečně převzaty z **třívrstvé architektury**.

7.2.1. Vytvoření datové třídy

Výstraha

V datové třídě téměř nikdy nepoužíváme před deklarací dat a metod označení `static` – podrobnosti viz dále.

7.2.1.1. Data (atributy)

- data deklarovaná ve třídě mohou být různých typů
  - primitivních datových typů – zde `int` a `double`
  - referenčních proměnných – zde `String`
- další používané názvy pro data jsou **položky** (*fields*), **složky**, **instanční proměnné**, **atributy** (*atributes*), ...
- deklarují se vně metod
  - jsou to **nelokální proměnné**
  - metody k nim mají přímý přístup bez nutnosti používat je ve formálních parametrech
- data představují **stav objektu**
  - např. stav objektu je: Hana, 50, 170

7.2.1.2. Metody

- píšeme je dle běžných konvencí
- nazýváme je **instančními metodami**
  - narozdíl od **statických metod**, před kterými je použito klíčové slovo `static`

- metody představují **schopnosti objektu**
  - např. umí vypočíst BMI, umí zjistit výšku, ...

7.2.1.3. Zapouzdření a autorizovaný přístup k datům

- data mohou být přímo přístupná z vnějšku objektu
  - tento postup zásadně nepoužíváme
  - používáme **zapouzdření** (*encapsulation*) dat, tj. jejich skrytí před vnějšími vlivy pomocí klíčového slova `private` (je to **přístupové právo**)
- z vnějšku pak přistupujeme k nepřístupným datům **autorizovaným přístupem** pomocí (speciálních) metod
  - výhodou je, že nikdo nemůže z vnějšku nastavit data objektu chybně (např. zápornou váhu)
  - nevýhodou je, že se zvyšuje administrativa
- pro speciální metody, které pouze čtou nebo nastavují hodnoty dat, se používá název **getry** a **setry**
  - z angličtiny *get* – získat a *set* – nastavit
  - v příkladě použijeme pouze jednu nastavovací metodu – `setVaha()`
    - to odpovídá běžné realitě, kdy změnit jméno a výšku (narozdíl od váhy) je velmi obtížná záležitost (ve třídě `Osoba` proto nemožná)
- přístupové právo metod je `public` – chceme, aby se instance třídy dala zvnějšku prostřednictvím metod používat a ovládat
  - v řídkých případech může být pomocná metoda označená jako `private` – nechlubíme se s ní navenek

7.2.1.4. Konstruktor

- protože instance třídy vzniká dynamicky pomocí operátoru `new`, je vhodné umožnit počáteční nastavení stavu objektu
- to se typicky provádí ve speciální metodě nazývané **konstruktor** (*constructor*)
- konstruktor se od ostatních metod liší tím, že:
  - jeho jméno si nemůžeme libovolně volit – musí se přesně shodovat se jménem třídy
  - nemá návratovou hodnotu (ani `void`)
  - nesmí být volána rekurzivně
  - je to metoda, která se spustí automaticky prostřednictvím operátoru `new` – viz dále
- jinak vypadá jako běžná metoda
- použití klíčového slova `this`:
  - není to jen výsada konstruktoru, ale v konstruktoru se typicky vyskytuje

  - formální parametry konstruktoru se často jmenují stejně jako atributy
    - stejné jméno formálního parametru překrývá jméno nelokální proměnné
    - řešení – `this.`*`jmenoAtributu`* – odkazuje vždy na atribut bez ohledu na jméno formální proměnné

7.2.1.5. Metoda toString()

- každá datová třída by měla mít tuto speciální metodu (podrobnosti později)
- její hlavička **musí** být

```
public String toString() {
```

- metoda vrací řetězec, který co nejlépe popisuje stav třídy, tj. nejčastěji zahrnuje hodnoty atributů třídy
  - může však zahrnovat i cokoliv dalšího, co tvůrce třídy uzná za vhodné
    - zde např. hodnotu BMI
- metoda nic nevypisuje!
  - typickou chybou je použití `System.out.println()` v této metodě
- výhodou této metody je, že se při použití třídy nemusíme nijak starat o správný výpis stavu – pouze stačí vypsat hodnotu vracenou metodou `toString()`

7.2.1.6. Příklad třídy

- vytvoříme třídu `Osoba`
  - s atributy: `jmeno`, `vaha`, `vyska`
  - s konstruktorem `Osoba()`
  - s metodami: `getJmeno()`, `getVaha()`, `setVaha()`, `getVyska()` a `vypoctiBMI()`

```
class Osoba {
  // atributy
  private String jmeno;
  private double vaha;
  private int vyska;

  // konstruktor
  public Osoba(String jmeno, double vaha, int vyska) {
    this.jmeno = jmeno;
    setVaha(vaha);
    this.vyska = vyska;
  }

  // getry a setry
  public String getJmeno() {
    return jmeno;
  }
```

```
  public double getVaha() {
    return vaha;
  }

  public void setVaha(double vaha) {
    if (vaha > 0.0) {
      this.vaha = vaha;
    }
  }

  public int getVyska() {
    return vyska;
  }

  // ostatni metody
  public int vypoctiBMI() {
    double vyskaMetry = vyska / 100.0;
    double bmi = vaha / (vyskaMetry * vyskaMetry);
    return (int) Math.round(bmi);  // zaokrouhleni
  }

  public String toString() {
    return "\n<" + jmeno
           + ", vaha: " + vaha
           + ", vyska: " + vyska
           + ", BMI: " + vypoctiBMI()
           + ">";
  }
}
```

Poznámka

Znaky `\n` a < a > nemají z hlediska principu použití metody `toString()` žádný význam. Jejich význam bude ukázán později.

7.2.1.7. Odstrašující příklad datové třídy

- předchozí příklad třídy má jednu nevýhodu a tou je velká administrativa („užvaněnost“)
  - začátečník proto rád věří „dobré radě“ „zkušenějšího“ kolegy typu: „Proč to děláš tak složitě, když to jde mnohem jednodušeji.“
- následující třída bude víceméně stejně použitelná (alespoň, co se týče objemu dat), jako mnohem rozsáhlejší třída předchozí

```
class OsobaSporiva {
  String jmeno;
  double vaha;
  int vyska;
}
```

- třída nemá žádné metody a atributy nemají specifikovány přístupová práva (nejsou `private`)

  - atributy jsou tak přímo přístupné z vnějšku instance
- tento způsob šetří práci jen zdánlivě
  - kód, který se ušetří při vytváření třídy, se bude totiž muset mnohokrát opakovaně napsat při každém použití třídy
  - navíc tato třída bude mnohem náchylnější k případným chybám
- stejným nesmyslem jsou i postupné kompromisy typu: „až to budu potřebovat, tak tam dodám alespoň konstruktor“
  - tato snaha končí tím, že postupně do třídy dodáte vše (nebo většinu) z toho, co je ve třídě původní, ale bude vás to stát spoustu času a zlobení
  - napište datovou třídu pořádně hned!
- Závěr: Tento způsob zápisu zásadně nepoužívat. Konstrukce třídy je základní stavební kámen budoucího programu. Pokud budou místo „základních kamenů“ použity (na mnoha různých místech) podivné „slepence z nepálené hlíny“, nelze se pak divit, že se program zhroutí.

7.2.2. Použití datové třídy

7.2.2.1. Kde se datová třída fyzicky vyskytuje

- zdrojový kód třídy `Osoba` může být uložen ve stejném souboru, jako kód hlavní třídy, která bude třídu `Osoba` používat (zde `OsobyObjektyZaklad`)
  - v tomto případě nesmí být před klíčovým slovem `class` modifikátor `public`

    pokud je, kompilátor vypíše chybovou zprávu:

    ```
    class Osoba is public, should be declared in a file named Osoba.java
    ```
  - pro jednoduché školní případy je takový postup přípustný s odůvodněním, že vše je přehledně v jednom souboru
  - překlad se provede příkazem

    ```
    javac OsobyObjektyZaklad.java
    ```

    nebo **Ctrl + F7** ve SciTE
  - spuštění se provede příkazem

    ```
    java OsobyObjektyZaklad
    ```
- prakticky ve větších programech platí následující pravidla
  - všechny zdrojové kódy jednoho programu jsou v samostatném adresáři (součástí jednoho projektu)
  - v něm se zásadně nevyskytují jakékoliv soubory, které k programu nepatří
  - každá třída je deklarována s klíčovým slovem `public` a tudíž je v samostatném souboru s příponou `.java`

- překlad všech tříd najednou se provede příkazem

```
javac *.java
```

nebo **F7** ve SciTE

- spuštění se provede příkazem

```
java OsobyObjektyZaklad
```

7.2.2.2. Jak se používá

■ ve třídě, která datovou třídu používá, je nutno vytvořit

1. referenční proměnnou typu této třídy, např.:

```
Osoba o;
```

2. instanci této třídy pomocí operátoru `new` se správně nastavenými parametry konstruktoru, např.:

```
o = new Osoba("Hana", 50, 170);
```

Poznámka

Tyto dva příkazy se běžně spojují v jeden:

```
Osoba o = new Osoba("Hana", 50, 170);
```

instance (objekt) tedy představuje vyhrazené místo v paměti, kde se společně nachází všechny atributy tohoto objektu a metody, které s nimi dokáží pracovat

■ od této chvíle existuje objekt třídy `Osoba` v paměti a je možné s ním pracovat

- zásadně jen pomocí referenční proměnné `o` a **tečkové notace**
- protože je třída `Osoba` napsaná svědomitě, komunikujeme s objektem **zasíláním zpráv**
  - ♦ to představuje volání instančních metod třídy, tj. povely instanci, aby něco provedla
  - ♦ nejjednodušší je žádost o vrácení řetězce popisujícího stav objektu a jeho okamžitý výpis

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

což vypíše

```
<Hana, vaha: 50.0, vyska: 170, BMI: 17>
```

  - ♦ můžeme volat i jednotlivé metody, např.:

```
System.out.println(o.getJmeno() + " ma BMI " + o.vypoctiBMI());
```

což vypíše

```
Hana ma BMI 17
```


Výstraha

Typická chyba je, že zapomeneme poslat zprávu **konkrétnímu** objektu, tedy voláme instanční metodu bez uvedení referenční proměnné, např.:

```
System.out.println(getJmeno() + " ma BMI " + vypoctiBMI());
```

Na to kompilátor zareaguje chybovou zprávou:

```
cannot find symbol
```

  - ♦ můžeme měnit stav objektu
    - – protože je ve třídě `Osoba` deklarována pouze jedna metoda typu „setr“ – `setVaha()`, lze objektu měnit jen váhu

```
o.setVaha(55.0);
```

  - ♦ „kouzelná věc“ metody `toString()` je, že pokud je od objektu očekáván řetězec, nemusí se volání metody `toString()` vůbec uvádět, protože si jej kompilátor doplní sám
    - – proto lze běžně použít výpis:

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

což vypíše

```
<Hana, vaha: 55.0, vyska: 170, BMI: 19>
```

  - ♦ pokud se pokusíme změnit stav objektu nedovoleným způsobem, objekt změnu neakceptuje

```
o.setVaha(-10.0);

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

což vypíše

```
<Hana, vaha: 55.0, vyska: 170, BMI: 19>
```

■ celý kód je

```
public class OsobyObjektyZaklad {
  public static void main(String[] args) {
    Osoba o;
    o = new Osoba("Hana", 50, 170);
    System.out.println(o.toString());
    System.out.println(o.getJmeno() + " ma BMI " + o.vypoctiBMI());
    o.setVaha(55.0);
    System.out.println(o);
  }
}
```

■ je zřejmé, že vzájemně nezávislých osob (tj. instancí třídy `Osoba`) můžeme vytvořit libovolné množství

7.2.2.3. Použití datové třídy v metodách třídy hlavní

- aneb princip, jak předat objekt do metody

Poznámka

Zde bude ukázán způsob, jak pracovat s objekty v metodách. Bude ukazován na koncepčně špatném příkladu, kdy vynecháme ze třídy `Osoba` výpočet BMI, čímž vznikne datová třída `OsobaBezBMI`. Ukázaný princip předávání objektů do metod je ale univerzálně platný.

- řešíme případ, kdy musíme BMI vypočítávat „zvnějšku“, tj. ve třídě, která datovou třídu `OsobaBezBMI` používá
  - do metody hlavní třídy můžeme data potřebná pro výpočet BMI předat dvěma způsoby
    1. do metody se předává celý objekt jako jediný formální parametr, kterým je referenční proměnná na objekt
       - ♦ zápis je principiálně stejný, jako u primitivního datového typu nebo pole
       - ♦ princip fungování je stejný jako u pole – předává se kopie referenční proměnné
       - ♦ výhodný způsob – jednoduchost
         - – formální parametr je pouze jeden
         - – snadné volání s jedním skutečným parametrem

```
public static int vypoctiBMI(OsobaBezBMI o) {
  double vyskaMetry = o.getVyska() / 100.0;
  double bmi = o.getVaha() / (vyskaMetry * vyskaMetry);
  return (int) Math.round(bmi);
}
```

volání

```
System.out.println(o.getJmeno() + " ma BMI "
                   + vypoctiBMI(o));
```

       - ♦ tomuto způsobu dáváme zásadně přednost – objekt je ucelený záznam, se kterým pracujeme vždy najednou (jako s celkem)
    2. do metody se předávají jednotlivé atributy, jako seznam formálních parametrů primitivních datových typů
       - ♦ nevýhodný způsob – složitost
         - – více formálních parametrů
         - – nepřehledné (dlouhé) vyplňování skutečných parametrů

```
public static int vypoctiBMIPoCastech(double vaha, int vyska) {
  double vyskaMetry = vyska / 100.0;
  double bmi = vaha / (vyskaMetry * vyskaMetry);
  return (int) Math.round(bmi);
}
```

volání

```
System.out.println(o.getJmeno() + " ma BMI "
     + vypoctiBMIPoCastech(o.getVaha(), o.getVyska()));
```

       - ♦ tento způsob nepoužíváme
  - celá hlavní třída vypadá takto

```
public class OsobyObjektyFormalniParametr {
  // rozumne pouziti formalniho parametru
  public static int vypoctiBMI(OsobaBezBMI o) {
    double vyskaMetry = o.getVyska() / 100.0;
    double bmi = o.getVaha() / (vyskaMetry * vyskaMetry);
    return (int) Math.round(bmi);
  }

  // nerozumne pouziti formalnich parametru
  public static int vypoctiBMIPoCastech(double vaha, int vyska) {
    double vyskaMetry = vyska / 100.0;
    double bmi = vaha / (vyskaMetry * vyskaMetry);
    return (int) Math.round(bmi);
  }

  public static void main(String[] args) {
    OsobaBezBMI o;
    o = new OsobaBezBMI("Hana", 50, 170);
    System.out.println(o.toString());
    System.out.println(o.getJmeno() + " ma BMI "
                       + vypoctiBMI(o));
    System.out.println(o.getJmeno() + " ma BMI "
     + vypoctiBMIPoCastech(o.getVaha(), o.getVyska()));
  }
}
```

a vypíše

```
<Hana, vaha: 50.0, vyska: 170>
Hana ma BMI 17
Hana ma BMI 17
```

7.3. Použití datové třídy v poli

7.3.1. Vytvoření pole objektů

- použití instancí v poli je přímočaré
- oproti polím primitivních datových typů je však při vytváření pole nutné provést jeden krok navíc

1. připravit referenční proměnnou typu pole typu `Osoba`, např.:

```
Osoba[] osoby;
```

2. vytvořit (alokovat) pole referenčních proměnných na objekty typu `Osoba`

```
osoby = new Osoba[POCET_OSOB];
```

3. vytvořit pomocí `new` a volání konstruktoru jednotlivé instance třídy `Osoba` a přiřadit je do přístušných prvků pole, tj. referenčních proměnných – toto se typicky provádí v cyklu

```
for (int i = 0; i < osoby.length; i++) {
  osoby[i] = new Osoba(jmena[i], vahy[i], vysky[i]);
}
```

Poznámka

Tento třetí krok nebyl v případě pole z primitivních datových prvků zapotřebí a proto se na něj často zapomíná. Kompilátor však nezapomene ;-)

- máme-li pole správně vytvořeno, je možné jej ihned vypsat

```
System.out.println(Arrays.toString(osoby));
```

což vypíše

```
[
<Pavel, vaha: 90.0, vyska: 186, BMI: 26>,
<Hana, vaha: 50.0, vyska: 170, BMI: 17>,
<Karel, vaha: 85.5, vyska: 190, BMI: 24>,
<Dana, vaha: 70.0, vyska: 175, BMI: 23>]
```

Poznámka

Zde je vidět důvod použití znaků `\n` a `<` a `>` v metodě `toString()`. Znaky `< >` uzavírají informace o jedné osobě a znak `\n` způsobí odřádkování.

- jednotlivým objektům (osobám uloženým v poli) lze zasílat zprávy (tj. požadavky, co mají dělat), např.:

```
System.out.println("Jednotlive osoby se jmenuji:");
for (int i = 0;  i < osoby.length;  i++) {
  System.out.print(osoby[i].getJmeno() + ", ");
}
```

vypíše

```
Jednotlive osoby se jmenuji:
Pavel, Hana, Karel, Dana,
```

- zasílané zprávy mohou představovat i požadavek na změnu stavu, např.:

```
System.out.println("V menze dobre vari - vsichni ztloustli o 5 kg:");
for (int i = 0;  i < osoby.length;  i++) {
  osoby[i].setVaha(osoby[i].getVaha() + 5.0);
}
System.out.println(Arrays.toString(osoby));
```

vypíše

```
V menze dobre vari - vsichni ztloustli o 5 kg:
[
<Pavel, vaha: 95.0, vyska: 186, BMI: 27>,
<Hana, vaha: 55.0, vyska: 170, BMI: 19>,
<Karel, vaha: 90.5, vyska: 190, BMI: 25>,
<Dana, vaha: 75.0, vyska: 175, BMI: 24>]
```

7.3.2. Seřazení pole objektů

- při řazení přehazujeme pouze referenční proměnné v poli osob
  - výměna je rychlá – nejedná se o výměnu objektů, ale referencí na ně
    - je to stejný princip, jako bylo řazení pomocí vložených indexů, ovšem na mnohem komfortnější úrovni
  - je zřejmé, že řazení nebude nijak narušovat vnitřek objektů osob
- pole osob nadeklarujeme jako nelokální s modifikátorem `static` – viz později
  - tím bude toto pole viditelné ve všech metodách – použito pro zjednodušení kódu
  - vlastní vytvoření a naplnění pole se provede v metodě `vytvorPoleOsob()`

```
static final int POCET_OSOB = 4;
static String[] jmena = {"Pavel", "Hana", "Karel", "Dana" };
static double[] vahy = {90, 50, 85.5, 70};
static int[] vysky = {186, 170, 190, 175};
static Osoba[] osoby;

public static void vytvorPoleOsob() {
  osoby = new Osoba[POCET_OSOB];
  for (int i = 0;  i < osoby.length;  i++) {
    osoby[i] = new Osoba(jmena[i], vahy[i], vysky[i]);
  }
}
```

- řazení proběhne (jako v předchozích případech) pomocí algoritmu SelectSort
  - při porovnávání je třeba od jednotlivých osob získat jejich váhu pomocí `getVaha()`

Poznámka

Zde teprve nabývá na významu termín **klíč**. Klíčem je zde hodnota váhy.

  - při prohazování prvků pole je třeba mít pomocnou proměnnou typu `Osoba`

```
public static void seradPodleVahy() {
  for (int i = 0;  i < osoby.length - 1;  i++) {
    int iMin = i;
    for (int j = i + 1;  j < osoby.length;  j++) {
      if (osoby[j].getVaha() < osoby[iMin].getVaha()) {
        iMin = j;
      }
    }
    if (i != iMin) {
      Osoba pom = osoby[iMin];
      osoby[iMin] = osoby[i];
      osoby[i] = pom;
    }
  }
}
```

- metoda `main()` jen volá předchozí metody

```
public static void main(String[] args) {
  vytvorPoleOsob();
  seradPodleVahy();
  System.out.println(Arrays.toString(osoby));
}
```

- výsledek je

```
[
<Hana, vaha: 50.0, vyska: 170, BMI: 17>,
<Dana, vaha: 70.0, vyska: 175, BMI: 23>,
<Karel, vaha: 85.5, vyska: 190, BMI: 24>,
<Pavel, vaha: 90.0, vyska: 186, BMI: 26>]
```

7.4. String jako příklad knihovní třídy

- jedna z nejpoužívanějších tříd z knihoven Javy – řetězce se používají prakticky všude
- zvláštnosti `String`
  - inicializovaný objekt řetězce může vzniknout i bez použití `new`

    ```
    String s = "Ahoj";
    ```

    nepoužívat: `String s = new String("Ahoj");`
  - spojení dvou řetězců je možné pomocí operátoru `+`
    - ♦ to jsme již od začátků běžně používali v `System.out.println()`

      ```
      String s1 = "Ahoj";

      String s2 = s1 + " lidi"; // "Ahoj lidi"
      ```


  - ♦ navíc platí, že při setkání řetězce jako prvního operandu a jiného datového typu jako druhého operandu se druhý automaticky převede na řetězec

    ```
    String s = "Ahoj" + 10; // "Ahoj10"
    ```
  - jednou vytvořený řetězec je konstantní – nelze jej měnit
    - ♦ běžně se to řeší tím, že vytvoříme jiný řetězec – toto omezení nás ve skutečnosti moc neomezuje

      Poznámka

      Existuje měnitelný řetězec `StringBuffer` – podrobnosti viz v literatuře.
    - ♦ i když to vypadá, že se řetězec mění, ve skutečnosti se jedná o více různých řetězců, na které ukazuje jedna referenční proměnná

      ```
      String s = "Ahoj";
      s = s + ", jak";
      s += " se mas?" // "Ahoj, jak se mas?"
      ```

7.4.1. Knihovní metody pro práci s řetězci

- existuje asi 50 metod (mnohé jsou přetížené)
  - stačí znát jen několik z nich, ostatní v literatuře nebo v dokumentaci
- protože řetězec je objekt, volají se jeho instanční metody jako:

  `s.`*`jmenoMetody`*`(`*`skutecneParametry`*`);`

  nikoliv jako:

  `String.`*`jmenoMetody`*`(`*`s, skutecneParametry`*`);`

  nebo jako:

  `jmenoMetody``(`*`s, skutecneParametry`*`);`

7.4.1.1. Práce s jednotlivými znaky řetězce

- `String` je speciální pole znaků (typ `char`)
  - v příkladech bude používán řetězec: `String s = "abcdaxy";`
- délka řetězce – Pozor: je to metoda, tj. se závorkami, nikoli konstanta jako u polí

  ```
  int i = s.length(); // i = 7
  ```
- znak na pozici

  ```
  char c = s.charAt(1); // c = 'b'
  ```
- index prvního výskytu znaku (-1 není-li nalezen)

  ```
  int p = s.indexOf('a'); // p = 0
  ```

- index posledního výskytu znaku

```
int k = s.lastIndexOf('a'); // k = 4
```

7.4.1.2. Porovnávání řetězců

- je potřeba porovnávat obsahy, nikoli referenční proměnné
  - chybné porovnávání – porovnávají se referenční proměnné

```
String s1 = "abcd";
String sp = "ab";
String s2 = sp + "cd";
if (s1 != s2) {
  System.out.println(s1 + " != " + s2);
}

// abcd != abcd
```

- porovnání s výsledkem typu `boolean` – `true` = rovnají se, `false` = nerovnají se

```
boolean b = s1.equals(s2);
```

- porovnání s výsledkem `int` – 0 = rovnají se, záporné číslo = první je menší, kladné číslo = druhý je menší

```
int i = s1.compareTo(s2);
```

- porovnání s výsledkem `int` bez ohledu na velikost písmen

```
int k = s1.compareToIgnoreCase("ABCD");
```

7.4.1.3. Náhrada všech znaků v řetězci

```
String s2, s1 = "cacao";

s2 = s1.replace('c', 'k');
```

7.4.1.4. Práce s podřetězci

- v příkladech bude používán řetězec: `String s = "abcdaxy";`
- vyhledávání – index prvního nalezeného (-1 není-li nalezen)

```
int p = s.indexOf("da"); // p = 3
```

- získání podřetězce od indexu včetně do konce

```
String vse = s.substring(3); // daxy
```

- získání podřetězce od indexu včetně do indexu bez

```
String odDo = s.substring(2, 5); // cda
```


7.4.1.5. Test začátku a konce řetězce

```
String s = "obr1234.jpg";

boolean z = s.startsWith("obr"); // true

boolean k = s.endsWith(".jpg"); // true
```

7.4.1.6. Oříznutí bílých znaků na okrajích

- provádíme při práci s řetězcem načteným ze vstupu – na jeho konci jsou znaky odřádkování
- ubere všechny mezery, tabulátory a konce řádek na konci i na začátku řetězce

```
String s2, s1 = "\r\n\t ahoj\t \r\n";

s2 = s1.trim();
```

7.4.1.7. Konverze řetězce na základní datové typy

- nutné, pokud při vstupu nepoužíváme třídu `Scanner`
- před převodem je vhodné použít `trim()`
- používáme metodu `parseXyz()` z příslušné obalovací třídy

```
String s = " 123 ";
s = s.trim();
int i = Integer.parseInt(s);
long l = Long.parseLong("5678901234");
double d = Double.parseDouble("1.3E-6");
```

7.4.2. Příklad použití řetězce

- toto je reálný příklad ze života
- v souboru s digitální fotografií mohou být na začátku údaje ve formátu Exif, popisující okolnosti vzniku fotografie
  - kromě jiného tam lze nalézt i čas vzniku fotografie ve formátu *hh:mm:ss rrrr:mm:dd*, např.:

```
10:05:43 2005:11:06
```

  - úkolem bude připravit řetězec, pomocí něhož později přejmenujeme soubor (obvykle pojmenovaný velmi nevýznamově, např. `PICT0031.JPG`) na `2005-11-06_10-05-43.jpg`

```
String casADatum = "10:05:43 2005:11:06";
int zacDatum = casADatum.indexOf(' ');
String cas = casADatum.substring(0, zacDatum);
String datum = casADatum.substring(zacDatum + 1);
String jmeno = datum + "_" + cas;
jmeno = jmeno.replace(':', '-');
```

```
jmeno += ".jpg";
System.out.println(jmeno);   // 2005-11-06_10-05-43.jpg
```

Poznámka

Ve skutečnosti je v Exif formát datumu a času *rrrr:mm:dd hh:mm:ss*, což je ale moc jednoduché pro ilustrační příklad ;-)

7.4.3. Zpracování příkazové řádky

- vstupní hodnoty se programu dají předat i při jeho spuštění na tzv. „příkazové řádce“
  - je to mnohem běžnější způsob, než zadávat hodnoty z klávesnice po spuštění programu
    - ♦ tento způsob využíváme nevědomky od začátku programování, např.:

      `javac PrvniProgram.java`

      kde řetězec `PrvniProgram.java` je parametr příkazové řádky
- pro přečtení hodnot těchto parametrů uvnitř našeho programu slouží (dosud nepoužívaný) formální parametr `args` metody `main()`

  Poznámka

  Skutečné parametry nikdy nenastavujeme – to za nás provede operační systém po spuštění programu.

  - `args` představuje pole typu `String`, které dokážeme zpracovat
    - ♦ význam těchto řetězců je velmi různorodý a určuje jej programátor aplikace
      - konvence stanoví, že pokud má řetězec obsah `-h` nebo `-help` nebo `/?` nebo `/h` program pouze vypíše nápovědu
        - některé programy, které vyžadují parametry příkazové řádky, vypisují nápovědu také v případě, že žádný parametr není uveden (např. `java.exe`)
      - jinak mají parametry nejčastěji význam jmen souborů a přepínačů činností (viz např. výpis po **`javac -help`**)
      - jakýkoliv jiný (smysluplný) význam je ale akceptovatelný – v každém případě nesmíme zapomenout, že se vždy jedná o řetězce (čísla je nutné převést)
    - ♦ jednotlivé parametry jsou od sebe na příkazové řádce odděleny nejčastěji mezerou
- v příkladu bude mít první parametr význam počtu opakování a druhý parametr význam jména

```
public class PrikazovaRadka {
  public static void help() {
    System.out.println("Pouziti:");
    System.out.println("  java PrikazovaRadka pocetOpakovani jmeno");
    System.exit(1);
  }
```



```
  public static void main(String[] args) {
    if (args.length == 0) {
      // zadny parametr neuveden - vypis napovedy
      help();
    }

    // dale predpokladame, ze jsou spravne zadany oba parametry
    // v realnem programu by se musel otestovat jejich pocet i jejich typy
    int pocetOpakovani = Integer.parseInt(args[0]);
    String jmeno = args[1];

    for (int i = 1;  i <= pocetOpakovani;  i++) {
      System.out.println("" + i + ". " + jmeno);
    }
  }
}
```

po spuštění

`java PrikazovaRadka`

vypíše

```
Pouziti:
  java PrikazovaRadka pocetOpakovani jmeno
```

po spuštění

`java PrikazovaRadka 3 Pavel`

vypíše

```
1. Pavel
2. Pavel
3. Pavel
```

7.5. Statické versus instanční

- proč někde `static` je a někde není
- toto klíčové slovo rozděluje atributy a stejně tak i metody do dvou základních skupin
  1. bez `static` – **instanční** atributy a/nebo metody
     - vztahují se pouze a jedině ke konkrétní instanci
     - další instance téže třídy nemohou být nijak ovlivněny
  2. se `static` – **třídní** atributy a/nebo metody
     - vztahují se k celé třídě, tzn. mohou být ovlivněny všechny instance
     - nebezpečná možnost – používat s rozvahou

- pochopení rozdílu je významný krok k pochopení konceptu objektově orientovaného programování
  - bez pochopení rozdílu můžeme programovat bez problémů na základní úrovni

7.5.1. Přístup – Nechápu rozdíl mezi třídními a instančními

- není to žádný problém pro ty, co nechtějí programovat profesionálně
- pro vcelku úspěšné programování typu:
  - matematický výpočet – např. řešení kvadratické rovnice, výpočet určitého integrálu, ...
  - transformace dat – např. přečtení čísel ze souboru a jejich zvětšení o 19 % DPH a zápis do souboru
  - práce s polem objektů datových tříd – např. pole osob z předchozího příkladu

  je možné se držet velmi jednoduchých pravidel:

  1. datová třída (typu `Osoba`), jejíž instance vytvořené pomocí `new` se budou vyskytovat nejčastěji v polích
     - ♦ klíčové slovo `static` *zásadně nepoužíváme* ani před atributy, ani před metodami

       instanční metody pracují s instančními atributy
  2. hlavní třída programu ve které se vyskytuje metoda `main()`
     - ♦ klíčové slovo `static` *zásadně používáme* před atributy i před metodami

       třídní metody pracují s třídními atributy
- porušit tato jednoduchá pravidla lze
  1. když rozumíme tomu, co děláme – viz dále
  2. když nám nevadí, že si způsobíme problémy

  tzn. porušit tato pravidla prakticky nelze ;-)
- dále následují ukázky problémů při porušení pravidel

Výstraha

V následujících ukázkách budou vynechána přístupová práva `private` u atributů a `public` u metod, ale pouze z toho důvodu, aby bylo co nejviditelnější použití či nepoužití `static`.

7.5.1.1. Vynechané static u hlavní třídy

- jakmile jen u jediné metody a/nebo atributu vynecháme `static`, program nepůjde přeložit
  - na výskytu ostatních `static` nezáleží
- jak to má vypadat

```
public class HlavniOK {
  static double vaha;
  static int vyska;

  static void vypisVahu() {
    System.out.println("vaha = " + vaha);
  }

  static void vypisVysku() {
    System.out.println("vyska = " + vyska);
  }

  public static void main(String[] args) {
    vaha = 90;
    vyska = 186;
    vypisVahu();
    vypisVysku();
  }
}
```

- jak vypadá při porušení pravidel

```
public class HlavniChyba {
  static double vaha;
  int vyska;            // chyba

  static void vypisVahu() {
    System.out.println("vaha = " + vaha);
  }

  void vypisVysku() {   // chyba
    System.out.println("vyska = " + vyska);
  }

  public static void main(String[] args) {
    vaha = 90;
    vyska = 186;
    vypisVahu();
    vypisVysku();
  }
}
```

chybový výpis

```
HlavniChyba.java:15: non-static variable vyska cannot be referenced
from a static context
    vyska = 186;
    ^
HlavniChyba.java:17: non-static method vypisVysku() cannot be referenced
from a static context
    vypisVysku();
    ^
```

- jak se to neobratně spraví

```
public class HlavniOprava {
  static double vaha;
  int vyska;          // poruseni pravidel

  static void vypisVahu() {
    System.out.println("vaha = " + vaha);
  }

  void vypisVysku() {   // poruseni pravidel
    System.out.println("vyska = " + vyska);
  }

  public static void main(String[] args) {
    HlavniOprava objekt = new HlavniOprava();
    vaha = 90;
    objekt.vyska = 186;
    vypisVahu();
    objekt.vypisVysku();
  }
}
```

je vidět, že musíme vyrobit z hlavní třídy její instanci pomocí `new` a k atributům nebo metodám bez `static` přistupovat pomocí reference na objekt

7.5.1.2. Přidané static u datové třídy

- tato chyba je závažnější – program lze bez problémů přeložit
  - pokud však vytvoříme více než jednu instanci datové třídy, program funguje chybně
- jak to má vypadat

Poznámka

Všimněte si, že osoby se nemusejí vytvářet v cyklu.

```
class OsobaOK {
  double vaha;
  int vyska;

  OsobaOK(double vaha, int vyska) {
    this.vaha = vaha;
    this.vyska = vyska;
  }

  double getVaha() {
    return vaha;
  }

  int getVyska() {
    return vyska;
  }

  public String toString() {
    return "\n<"
           + "vaha: " + vaha
           + ", vyska: " + vyska
           + ">";
  }
}

public class DatovaVHlavniOK {
  public static void main(String[] args) {
    OsobaOK[] osoby = new OsobaOK[2];
    osoby[0] = new OsobaOK(90, 186);
    osoby[1] = new OsobaOK(50, 170);
    System.out.println(Arrays.toString(osoby));
  }
}
```

vypíše:

```
[
<vaha: 90.0, vyska: 186>,
<vaha: 50.0, vyska: 170>]
```

- jak vypadá při porušení pravidel

Poznámka

Kód v hlavní třídě zůstal zcela nezměněn.

```
class OsobaChyba {
  private double vaha;
  static private int vyska;  // poruseni pravidel

  public OsobaChyba(double vaha, int vyska) {
    this.vaha = vaha;
    this.vyska = vyska;
  }

  public double getVaha() {
    return vaha;
  }
    // poruseni pravidel
  static public int getVyska() {
    return vyska;
  }

  public String toString() {
    return "\n<"
           + "vaha: " + vaha
           + ", vyska: " + vyska
           + ">";
  }
}
```

```
public class DatovaVHlavniChyba {
  public static void main(String[] args) {
    OsobaChyba[] osoby = new OsobaChyba[2];
    osoby[0] = new OsobaChyba(90, 186);
    osoby[1] = new OsobaChyba(50, 170);
    System.out.println(Arrays.toString(osoby));
  }
}
```

vypíše chybně – výška 170 se opakuje

```
[
<vaha: 90.0, vyska: 170>,
<vaha: 50.0, vyska: 170>]
```

Poznámka

Důvodem chyby je, že třídní atribut (se `static`) je ve všech instancích třídy pouze v jednom originále. Každá jeho změna se tak projeví ve všech existujících instancích.

- tento problém nelze spravit, a to ani neobratně
  - jedná se o chybu v návrhu třídy

7.5.2. Přístup – Chci pochopit rozdíl mezi třídními a instančními

- nezbytné pro ty, kteří budou Javu (nebo C# nebo C++ apod.) používat na netriviální úrovni
- následují ukázky, proč porušit výše popsaná rozumná pravidla

7.5.2.1. Přidané static u datové třídy

- `static` u atributu znamená, že existuje pouze jedno paměťové místo společné pro všechny instance
  - atribut je **třídní**, tzn. jeden společný pro celou třídu a všechny její instance
  - používá se velmi zřídka
  - typické použití je ve významu počitadla již vzniklých objektů
- pokud je atribut třídní, většinou s ním pracuje i třídní metoda (se `static`) – je to rozumné
  - není to ale podmínkou – je možná i instanční metoda – důvody v KIV/PGS
- příklad na použití počítadla existujících objektů
  - rozlišuje se třídní `pocitadlo` a instanční `poradi`
  - použití třídní metody je pomocí *`JmenoTridy.jmenoMetody()`*, tj. `OsobaPocitadlo.getPocitadlo()`

```
class OsobaPocitadlo {
  private static int pocitadlo = 0;
  private int poradi;
  private double vaha;
  private int vyska;

  public OsobaPocitadlo(double vaha, int vyska) {
    pocitadlo++;
    this.poradi = pocitadlo;
    this.vaha = vaha;
    this.vyska = vyska;
  }

  public static int getPocitadlo() {
    return pocitadlo;
  }

  public double getVaha() {
    return this.vaha;
  }

  public int getVyska() {
    return this.vyska;
  }

  public String toString() {
    return "\n<" + this.poradi
           + ".: vaha: " + this.vaha
           + ", vyska: " + this.vyska
           + ">";
  }
}

public class DatovaVHlavniPocitadlo {
  public static void main(String[] args) {
    OsobaPocitadlo o1 = new OsobaPocitadlo(90, 186);
    OsobaPocitadlo o2 = new OsobaPocitadlo(50, 170);
    System.out.print("Existuji "
                + OsobaPocitadlo.getPocitadlo()
                + " osoby. Jsou to:");
    System.out.print(o2);
    System.out.println(o1);
  }
}
```

vypíše:

```
Existuji 2 osoby. Jsou to:
<2.: vaha: 50.0, vyska: 170>
<1.: vaha: 90.0, vyska: 186>
```

7.5.2.2. Přidané static u nedatové třídy

- nejedná se o hlavní třídu s `main()`

- je to typicky třída s konstantami s všeobecně užitečnými metodami, tzv. *utility class*
  - všechny atributy jsou `static final`, tj. konstanty
  - všechny metody jsou `static`
  - třída nemá konstruktor – její instance se nevytvářejí
- typický příklad je známá třída `java.lang.Math`
- metody této třídy se volají

  `Math.`*`jménoMetody(parametry)`*`;`

  např.: `double d = Math.pow(x, 2.0);`
- konstanty z této třídy se použijí

  `Math.`*`JMÉNO_KONSTANTY`*`;`

  např.: `double pi = Math.PI;`

```
class Prvocisla {
  private static final int[] prvocisla = {
    2, 3, 5, 7, 11, 13, 17, 19 };
  public static final int POCET_PRVOCISEL = prvocisla.length;

  public static int prvocisloVPoradi(int n) {
    if (n - 1 < POCET_PRVOCISEL) {
      return prvocisla[n - 1];
    }
    else {
      return 0;
    }
  }

  public static boolean jePrvocislo(int cislo) {
    for (int i = 0;  i < prvocisla.length;  i++) {
      if (cislo == prvocisla[i]) {
        return true;
      }
    }
    return false;
  }
}

public class HlavniPrvocisla {
  public static void main(String[] args) {
    System.out.println("K dispozici je "
              + Prvocisla.POCET_PRVOCISEL + " prvocisel");

    int c = 6;
    System.out.print("Cislo " + c);
    if (Prvocisla.jePrvocislo(c) == true) {
      System.out.println(" je prvocislo");
    }
    else {
      System.out.println(" neni prvocislo");
    }

    System.out.println("Sedme prvocislo je "
                + Prvocisla.prvocisloVPoradi(7));
  }
}
```


vypíše

```
K dispozici je 8 prvocisel
Cislo 6 neni prvocislo
Sedme prvocislo je 17
```

7.5.2.3. Hlavní třída

- typicky obsahuje pouze metodu `main()`
- další častá třídní metoda je zpracování parametrů příkazové řádky – viz dříve
- pro naše účely budou v této třídě ještě další třídní metody – přesně tak, jak bylo ukázáno dříve
  - profesionálně používaný postup viz v pozdějších předmětech

7.5.2.4. Vztah mezi třídními a instančními

- uvažujeme stále autorizovaný přístup, tj. atributy jsou zvnějšku třídy přístupné pouze přes příslušné getry a setry
- instanční metody smějí
  - používat atributy
    - ♦ třídní
    - ♦ instanční
  - volat metody
    - ♦ třídní
    - ♦ instanční
- statické metody smějí
  - používat atributy
    - ♦ třídní
  - volat metody
    - ♦ třídní
- příklad na všechny kombinace

- v instanční metodě `toString()` jsou zároveň ukázány způsoby přístupu k atributům i k metodám

```
class TridniInstancni {  // povolene kombinace
  private static int pocitadlo = 0;
  private int poradi;

  public TridniInstancni() {
    pocitadlo++;
    this.poradi = getPocitadlo();
  }

  public static int getPocitadlo() {
    return pocitadlo;
  }

  public int getPoradi() {
    return poradi;
  }

  public String toString() {
    String s = "" + poradi + ". z " + pocitadlo;
    return "" + getPoradi() + ". z " + getPocitadlo();
  }
}

public class HlavniTridniInstancni {
  public static void main(String[] args) {
    System.out.println("Zacatek - existuje "
                + TridniInstancni.getPocitadlo()
                + " instanci");
    TridniInstancni o1 = new TridniInstancni();
    TridniInstancni o2 = new TridniInstancni();
    System.out.println("Existuji "
                + TridniInstancni.getPocitadlo()
                + " instance. Jsou to:");
    System.out.println(o1);
    System.out.println(o2);
  }
}
```

vypíše

```
Zacatek - existuje 0 instanci
Existuji 2 instance. Jsou to:
1. z 2
2. z 2
```


Kapitola 8. Výjimky, adresáře a soubory

8.1. Výjimky

8.1.1. Úvodní informace

- mechanismus výjimek (*exception*) je jedním z velmi silných bezpečnostních prvků Javy
- mnoho (starších) programovacích jazyků sice umožňuje testovat chybové kódy typu „neotevřený soubor“, „málo paměti“ atd.
  - většina z těchto jazyků ale ponechává provedení tohoto typu testů na libovůli programátora
- Java přímo na úrovni kompilátoru nutí programátora, aby ve svém kódu reagoval na některé možné chybové stavy
  - pokud nereaguje, program se nepřeloží, byť je kód správně

```
public static int[] vytvorANactiPole() {
  Scanner sc = new Scanner(new File("data.txt"));
  int n = sc.nextInt();
  int[] pole = new int[n];
  for (int i = 0;  i < n;  i++) {
    pole[i] = sc.nextInt();
  }
  return pole;
}
```

  vypíše chybu při překladu:

```
VyjimkaPropagovana.java:8:
  unreported exception java.io.FileNotFoundException;
  must be caught or declared to be thrown
    Scanner sc = new Scanner(new File("data.txt"));
                 ^
```

  dokud programátor tuto závadu neodstraní, nemůže pokračovat dále!
- je několik způsobů, jak reagovat na výjimku
  - konkrétní způsob reakce závisí na mnoha dalších okolnostech
- pod pojmem **výjimka** (*exception*) je míněn také **výjimečný stav** (*exceptional event*) nebo (nepřesně) **chyba** (*error*)
  - všechny uvedené pojmy označují událost, o které si nepřejeme, aby v našem programu vznikla
  - bez jejího vzniku bude náš program fungovat lépe (dle očekávání)
- v souvislosti s výskytem výjimky se používá termín **vyhodit výjimku** (*throw exception*)

Poznámka

Zde bude vysvětlena problematika výjimek jen do té míry, aby postačovala pro budoucí práci se soubory. Kompletní práci s výjimkami viz literatura nebo navazující předměty.

8.1.2. Možné druhy výjimek

- z hlediska (běžného) programátora se rozlišují dva základní druhy výjimek
  - `Exception` – nutné ošetření ve zdrojovém kódu
  - `RuntimeException` – dobrovolné ošetření
- způsob práce s oběma typy je zcela stejný
- od těchto dvou základních typů je odvozeno (odděděno) množství dalších výjimek
  - protože vznikají děděním od základních, stačí při ošetřování výjimek vždy uvádět názvy základních druhů
    - např. místo `FileNotFoundException` lze použít jen `Exception`
      - toto je ovšem zjednodušení, které může použít jen začátečník

8.1.2.1. Exception

- též **kontrolované výjimky** (*checked exceptions*)
- musíme se o ně v programu postarat, tj. ošetřit je přidáním speciální části kódu
- vyskytují v souvislosti s voláním metod, u kterých je zvýšená pravděpodobnost, že se při jejich použití může vyskytnout problém
  - typicky jsou to všechny metody, které pracují se vstupy a výstupy
  - u těchto metod si jejich programátoři byli vědomi potenciálního nebezpečí a metody napsali tak, aby bylo při jejich použití nutné na nebezpečí reagovat

8.1.2.2. RuntimeException

- reakce na chyby, které se mohou vyskytnout v programu naprosto kdekoliv, např.:
  - `ArithmeticException` – dělení nulou apod.
  - `ArrayIndexOutOfBoundsException` – index mimo rozsah pole
- protože se tyto výjimky mohou vyskytnout kdekoliv (ne jen při volání určitých metod), nenutí nás překladač, abychom je ošetřili pomocí speciální části kódu
  - považujeme-li to za užitečné, můžeme je ošetřit!
    - typicky je to v místech, kde je zvýšená pravděpodobnost problému (vstupy od uživatele)
  - nemá větší smysl ošetřovat je v místě, kde jsme si stoprocentně jisti, že zmíněná situace nemůže nastat

- pokud přesto nastane, Java na tuto výjimku zareaguje ukončením programu a chybovým výpisem, ze kterého lze vyčíst, kde chyba nastala

  ukázka neošetřené výjimky

```
public class VyjimkaNeosetrena {
  public static void main(String[] args) {
    int[] abc = new int[3];
    abc[3] = 5;
  }
}
```

  ukázka chybového výpisu

```
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException: 3
     at VyjimkaNeosetrena.main(VyjimkaNeosetrena.java:4)
```

- následujícím krokem je pak úprava zdrojového kódu a ošetření této výjimky, která „nemůže nastat“

8.1.3. Způsoby reakce na výjimku

- programátor, který reaguje na (kontrolovanou) výjimku, má v zásadě tři možnosti:
  1. výjimku neumí (nebo nechce) ošetřovat a proto informaci o jejím výskytu předá do nadřazené úrovně (tzn. volající metodě)
     - **propagace výjimek**, **deklarace výjimek**
  2. výjimku zachytí a kompletně ošetří v metodě, kde se vyskytla
     - **ošetření výjimek**
  3. výjimku částečně či kompletně ošetří v metodě a navíc pošle informaci o jejím výskytu do nadřazené úrovně
     - kombinace obou předchozích způsobů
- každý z těchto způsobů má své výhody a nevýhody a také různé varianty použití
  - pro účely začátečnického programování probereme jen jednoduché verze dvou prvních případů

8.1.3.1. Předání výjimky výše – deklarace výjimky

- metoda, ve které se výjimka vyskytla, se zříká odpovědnosti za její zpracování
  - předání odpovědnosti výše se jasně deklaruje již v hlavičce metody pomocí klíčového slova `throws` (vyhazuje)

Poznámka

Tento postup je velmi jednoduchý, ale ve skutečnosti se řešení problému pouze odsouvá. Někdy bude muset být provedeno.

- je to nejjednodušší řešení, alespoň z prvního pohledu
  - ♦ není třeba provádět žádné zásahy do stávajícího kódu metody, stačí jen doplnit její hlavičku o:

```
throws Exception
```

```
import java.util.*;
import java.io.*;

public class VyjimkaPropagovana {
  public static int[] vytvorANactiPole() throws Exception {
    Scanner sc = new Scanner(new File("data.txt"));
    int n = sc.nextInt();
    int[] pole = new int[n];
    for (int i = 0;  i < n;  i++) {
      pole[i] = sc.nextInt();
    }
    return pole;
  }

  public static void main(String[] args) {
    int[] abc = vytvorANactiPole();
    System.out.println(Arrays.toString(abc));
  }
}
```

po překladu se ale vypíše chybové hlášení

```
VyjimkaPropagovana.java:16:
  unreported exception java.lang.Exception;
  must be caught or declared to be thrown
    int[] abc = vytvorANactiPole();
                               ^
```

což je přesně doklad toho, že problém byl z metody `vytvorANactiPole()` pouze přesunut do metody `main()`

použijeme-li v hlavičce metody `main()` stejnou taktiku, překlad proběhne bez problémů

```
import java.util.*;
import java.io.*;

public class VyjimkaPropagovana {
  public static int[] vytvorANactiPole() throws Exception {
    Scanner sc = new Scanner(new File("data.txt"));
    int n = sc.nextInt();
    int[] pole = new int[n];
    for (int i = 0;  i < n;  i++) {
      pole[i] = sc.nextInt();
    }
    return pole;
  }

  public static void main(String[] args) throws Exception {
    int[] abc = vytvorANactiPole();
    System.out.println(Arrays.toString(abc));
  }
}
```

- v případě, že soubor `data.txt` neexistuje, je po spuštění vypsáno následující chybové hlášení:

```
Exception in thread "main"
  java.io.FileNotFoundException: data.txt
  (Systém nemůže nalézt uvedený soubor)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:106)
  at java.util.Scanner.<init>(Scanner.java:621)
  at VyjimkaPropagovana.vytvorANactiPole(VyjimkaPropagovana.java:6)
  at VyjimkaPropagovana.main(VyjimkaPropagovana.java:16)
```

z něj je jasně patrné, na jakém místě zdrojového kódu (čísla řádek) došlo k problému

Poznámka

Metoda `main()` předala odpovědnost výše – virtuálnímu stroji (JVM), který výjimku zpracoval, tj. vypsal chybové hlášení a ukončil program.

- výhody
  - jednoduchost – stačí jen změna hlavičky metody
  - neřešíme to, na co nemáme kompetence
    - ♦ pokud je metoda vyvolávající výjimku hluboce zanořena (což je typické pro různá čtení ze souborů), je vhodné, aby se o neúspěchu akce dozvěděla i metoda, která je nadřazena mnohem výše (typicky z GUI) a mohla tak odpovídajícím způsobem reagovat
- nevýhody
  - nutíme všechny, co naši metodu používají, aby se postarali o problémy, které způsobí naše metoda
    - ♦ je-li naše metoda častokrát volaná z různých míst, je to nešťastné řešení, pokud mohl být problém vyřešen v této metodě

8.1.3.2. Kompletní ošetření výjimky

- případná výjimka nepronikne ven z metody
  - nadřazená úroveň se nemusí o nic starat
- ošetření výjimky se provede pomocí konstrukce `try-catch`
  - vlastní výkonný kód se nijak nemění, jen se uzavře do bloku začínající `try`
    - ♦ prakticky je často nutné přesunout některé deklarace vně bloku `try`
  - blok `catch`, který za blokem `try` bezprostředně následuje, stanoví, na jakou výjimku se bude reagovat a jak

8.1.3.2.1. Primitivní, ale účinný způsob

- v bloku `catch` se nesnažíme o nápravu problému, pouze vypíšeme chybové hlášení a často i ukončíme program
  - v mnoha případech nejrozumnější reakce
  - pro začátečníky doporučovaná

```
public class VyjimkaOsetrenaVypisem {
  public static int[] vytvorANactiPole() {
    int[] pole = null;   // vysunuti deklarace ven z bloku
    try {
      Scanner sc = new Scanner(new File("data.txt"));
      int n = sc.nextInt();
      pole = new int[n];
      for (int i = 0;  i < n;  i++) {
        pole[i] = sc.nextInt();
      }
      return pole;
    }
    catch (Exception e) {
      e.printStackTrace(); // vypis vyjimky
      System.exit(1);      // ukonceni programu
    }
    return pole;
  }

  public static void main(String[] args) {
    int[] abc = vytvorANactiPole();
    System.out.println(Arrays.toString(abc));
  }
}
```

po spuštění a nenalezení souboru `data.txt` vypíše

```
java.io.FileNotFoundException: data.txt
  (Systém nemůže nalézt uvedený soubor)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:106)
  at java.util.Scanner.<init>(Scanner.java:621)
  at VyjimkaOsetrenaVypisem.vytvorANactiPole(VyjimkaOsetrenaVypisem.java:8)
  at VyjimkaOsetrenaVypisem.main(VyjimkaOsetrenaVypisem.java:24)
```

Poznámka

Ukončení práce programu (zde pomocí `System.exit(1);`) je nutností, protože po výstupu z metody `vytvorANactiPole()` se jinak očekává, že vše proběhlo v pořádku, což není pravda. Chybový výpis sám o sobě nezpůsobí ukončení programu!

- pokud by se později stalo, že výjmečná situace nastává často, máme vždy možnost přejít k následujícímu řešení

8.1.3.2.2. Náprava problému

- použijeme, pokud umíme v metodě problém zcela vyřešit
  - většinou se použije cyklus, ve kterém se umožní zadání správných hodnot

```
import java.util.*;
import java.io.*;

public class VyjimkaOsetrenaKompletne {
  public static int[] vytvorANactiPole() {
    int[] pole = null;
    String jmeno = "data.txt";
    while (true) {
      try {
        Scanner sc = new Scanner(new File(jmeno));
        int n = sc.nextInt();
        pole = new int[n];
        for (int i = 0;  i < n;  i++) {
          pole[i] = sc.nextInt();
        }
        break;
      }
      catch (Exception e) {
        System.out.println("Soubor " + jmeno + " nenalezen!");
        System.out.print("Zadej spravne jmeno: ");
        Scanner scKlavesnice = new Scanner(System.in);
        jmeno = scKlavesnice.next();
      }
    }
    return pole;
  }

  public static void main(String[] args) {
    int[] abc = vytvorANactiPole();
    System.out.println(Arrays.toString(abc));
  }
}
```

po spuštění vypíše např.:

```
Soubor data.txt nenalezen!
Zadej spravne jmeno: jiny.txt
Soubor jiny.txt nenalezen!
Zadej spravne jmeno: data-pole.txt
[1, 2, 3, 4, 5]
```

8.1.4. Naprosto nejhorší reakce na výjimku

Poznámka

Není to ve skutečnosti žádná speciální kategorie, jen podmnožina způsobu ošetření výjimky.

- důvod použití je v naprosté většině lenost programátora
  - nechce výjimku propagovat, protože by to následně musel udělat i ve všech volajících metodách
  - do bloku `catch` nic nenapíše, protože se to „nikdy nestane“

```
public class VyjimkaOsetrenaChybne {
  public static int[] vytvorANactiPole() {
    int[] pole = null;
    try {
      Scanner sc = new Scanner(new File("data.txt"));
      int n = sc.nextInt();
      pole = new int[n];
      for (int i = 0;  i < n;  i++) {
        pole[i] = sc.nextInt();
      }
      return pole;
    }
    catch (Exception e) {  // zadna akce
    }
    return pole;
  }

  public static void main(String[] args) {
    int[] abc = vytvorANactiPole();
    System.out.println(Arrays.toString(abc));
  }
}
```

- tímto způsobem jsme se připravili o všechny výhody mechanismu výjimek
  - výjimka proběhne, ale nikdo se nedozví kdy a kde

8.1.4.1. Elegantní řešení této situace

- někdy skutečně potřebujeme na zachycenou výjimku nijak nereagovat a také ji nepropagovat
- pak jsou k dispozici dva základní přístupy – oba se používají v závislosti na typu problému
  - blok `catch` je prázdný, ale existuje v něm dostatečně podrobný a zdůvodňující komentář
    - typický případ je při synchronizaci vláken

```
catch (InterruptedException e) {
  // zde není žádná reakce na výjimku, protože ...
}
```

  - výjimka kontrolovaná se převede na Runtime výjimku (tu není nutné ošetřovat), která se vyhodí
    - typický případ je zavírání souborů metodou `close()` – viz dále


```
catch (IOException e) {
  throw new RuntimeException(e);
}
```

8.1.5. Konstrukce `try-catch-finally`

- dříve zmíněná konstrukce `try-catch` se často doplňuje ještě o blok uvedený klíčovým slovem `finally`
  - tento blok pak proběhne vždy, tzn. ať již výjimka nenastane (provádí se jen kód v bloku `try`) nebo i když nastane (provádí se i kód v bloku `catch`)

```
try {
  // výkonný kód
  // provádí se celý, pokud nevznikne výjimka
}
catch (Exception e) {
  // kód zachycení výjimky
  // provádí se po výskytu výjimky
}
finally {
  // ukončovací kód
  // provádí se vždy jako poslední po try či catch
}
```

- typické použití je při práci se soubory – viz dále

8.2. Adresáře a soubory pomocí třídy File

Poznámka

Zde se budeme na soubory dívat jako na nedělitelný celek, tj. nebude nás (zatím) zajímat jejich obsah.

- pro práci se soubory slouží třída `File`
  - nezajišťuje žádné způsoby čtení nebo zápisu, slouží jako „manažer“ souborů
  - slouží zcela stejným způsobem i pro adresáře (*directory*)
    - soubor se od adresáře rozliší pomocí metody `isDirectory()` nebo `isFile()`

      Poznámka

      Pojem **složka** se zde nepoužívá.
  - třída mj. umožňuje
    - test existence
    - vytváření a mazání
    - přejmenování a přesunutí

- ♦ získávání informací (velikost, čas vzniku apod.)

- je uložena v balíku `java.io`, takže na začátku programu musíme napsat

```
import java.io.*;
```

8.2.1. Terminologie

- **přípona souboru** – vše od poslední tečky do konce `.java`
- **jméno souboru**
  - většinou vše včetně přípony – `Pokus.java`
  - ve speciálních případech vše bez přípony, např.

    jméno souboru je `Pokus`

    - ♦ `Pokus.java` je zdrojový kód
    - ♦ `Pokus.class` je přeložený kód souboru `Pokus.java`
- **aktuální adresář** (*actual directory*) – značen `.`
- **nadřazený adresář** (**rodičovský adresář** – *parent directory*) – značen `..`
- **podadresář** (**vnořený adresář**, **podřízený adresář** – *subdirectory*) – adresář vnořený do nějakého adresáře (nejčastěji aktuálního)
- **kořenový adresář** (*root directory*)
  - Windows `\` (zpětné lomítko – *backslash*)
    - ♦ ve Windows je kořenový adresář ještě spojen s označením disku, např.:

      `D:\`

      Poznámka

      Jméno disku se píše z konvence velkým písmenem. Není to však nezbytné.
  - Linux `/` (lomítko – *slash*)
- **úplná cesta** (*absolute path*) – adresářová cesta od kořenového adresáře do daného adresáře, např.:

```
D:\ppa1\prednasky-priprava\10\soubory\
```

  nebo

```
D:\ppa1\prednasky-priprava\10\soubory
```

  s existencí nebo neexistencí závěrečného (zpětného) lomítka je nutno v programu počítat
- **úplné jméno souboru** – adresářová cesta od kořenového adresáře a jméno souboru
  - jednoznačná identifikace souboru v souborovém systému, např.:


```
D:\ppa1\prednasky-priprava\10\soubory\Pokus.java
```

- **relativní cesta** (*relative path*), **neúplná cesta**, **relativní jméno souboru** – používat co nejméně (značná možnost omylu)
  - je to cesta k souboru (nejčastěji) z aktuálního adresáře, např.:

    `..\html.css` – soubor `html.css` v nadřazeném adresáři

    `..\08\ppa1-08.xml` – soubor `ppa1-08.xml` v adresáři `08`, který je podadresářem nadřazeného adresáře

    `.\soubory\AktualniAdresar.java` – soubor `AktualniAdresar.java` v podadresáři `soubory`

    `soubory\AktualniAdresar.java` – totéž jako předchozí

    `\soubory\AktualniAdresar.java` – významný rozdíl od předchozího – soubor `AktualniAdresar.java` v podadresáři kořenového adresáře (pravděpodobně nebude existovat)

8.2.2. Zajištění nezávislosti na operačním systému

- existují různé systémy správy souborů a tím také různé odlišnosti v oddělování jednotlivých adresářů v úplné cestě souborů
  - Linux – oddělovačem adresářů je znak `/` a není více disků

    `/adresar1/adresar2/soubor.txt`
  - Windows – oddělovačem adresářů je znak `\` a je více disků

    `C:\adresar1\adresar2\soubor.txt`
- třída `File` poskytuje statickou proměnnou `File.separator` typu `String`
  - pro Linux má hodnotu `"/"`
  - pro Windows má hodnotu `"\"` – neřeší problém se jménem disku
  - řešení závislosti na operačním systému je přesunuto na JVM
  - použití

```
String jmeno = File.separator + "adresar1"
             + File.separator + "adresar2"
             + File.separator + "soubor.txt";
```

8.2.3. Vytvoření instance třídy File

Výstraha

Vytvoření instance využívající jméno souboru nebo adresáře neznamená, že uvedený soubor nebo adresář existuje fyzicky na disku. Existovat může, ale také nemusí.

- k dispozici jsou tři základní způsoby – konstruktory

1. nejjednodušší – pouze jméno souboru v aktuálním adresáři

```
File soubor = new File("a.txt");
System.out.println(soubor.getCanonicalPath());
```

vypíše

```
D:\ppa1\prednasky-priprava\10\soubory\a.txt
```

Poznámka

Kromě jména souboru v aktuálním adresáři lze zadat i úplnou nebo relativní cestu. Je to ovšem platformně nepřenositelná konstrukce a v případě Windows je nutné zdvojit všechna zpětná lomítka. Například:

```
File soubor = new File("D:\\ppa1\\a.txt");
```

2. objekt podadresáře a souboru v tomto podadresáři

```
File aktualniAdr = new File(".");
System.out.println(aktualniAdr.getCanonicalPath());
File soubor2 = new File(aktualniAdr, "b.txt");
System.out.println(soubor2.getCanonicalPath());
```

vypíše

```
D:\ppa1\prednasky-priprava\10\soubory
D:\ppa1\prednasky-priprava\10\soubory\b.txt
```

3. jméno podadresáře v aktuálním adresáři a souboru v tomto podadresáři

```
File souborVAdresari =
          new File("nexistujiciAdresar", "c.txt");
System.out.println(souborVAdresari.getCanonicalPath());
```

vypíše

```
D:\ppa1\prednasky-priprava\10\soubory\nexistujiciAdresar\c.txt
```

Poznámka

Ani jeden z těchto souborů a adresářů na disku neexistuje.

8.2.4. Práce s existujícím souborem nebo adresářem

8.2.4.1. Získávání informací

- `boolean exists()` – zjištění, zda existuje
- `boolean isFile()` – je to soubor
- `boolean isDirectory()` – je to adresář

- `long length()` – délka souboru v počtu bajtů
- `long lastModified()` – datum poslední modifikace – Pozor – počet milisekund od 1.1.1970

```
Calendar c = new GregorianCalendar();
c.setTimeInMillis(soubor.lastModified());
System.out.format("%tF%n", c);
```

vypíše např.:

```
2005-11-12
```

8.2.4.2. Změny

- `delete()` – vymazání souboru
- `renameTo(File dest)` – přejmenování nebo přesun souboru

```
File soubor = new File("a.txt");
System.out.println(soubor.getName() + " " + soubor.exists());

File souborNovy = new File("b.txt");
soubor.renameTo(souborNovy);
System.out.println(souborNovy.getCanonicalPath());
System.out.println(soubor.getName() + " " + soubor.exists());

File souborJinde = new File("..", "b.txt");
souborNovy.renameTo(souborJinde);
System.out.println(souborJinde.getCanonicalPath());
```

vypíše:

```
a.txt true
D:\ppa1\prednasky-priprava\10\soubory\b.txt
a.txt false
D:\ppa1\prednasky-priprava\10\b.txt
```

8.2.4.3. Vytvoření adresáře

- `mkdirs()` – vytvoří adresář nebo skupinu adresářů

```
File adr = new File("prvni" + File.separator + "druhy");
System.out.println(adr.getCanonicalPath() + " "
                   + adr.exists());
adr.mkdirs();
System.out.println(adr.getCanonicalPath() + " "
                   + adr.exists());
```

vypíše

```
D:\ppa1\prednasky-priprava\10\soubory\prvni\druhy false
D:\ppa1\prednasky-priprava\10\soubory\prvni\druhy true
```

8.2.5. Výpis položek adresáře

- `String[] list()` – jména souborů a podadresářů

```
File adr = new File("..");
String[] jmena = adr.list();
for (int i = 0;  i < jmena.length;  i++) {
  System.out.println(jmena[i]);
}
```

- `File[] listFiles()` – soubory a podadresáře jako objekty `File`
  - výhodnější než předchozí – lze ihned získávat informace o jednotlivých položkách

    v příkladu se vypisují soubory pouze jménem a podadresáře celou cestou

```
File[] polozky = adr.listFiles();
for (int i = 0;  i < polozky.length;  i++) {
  if (polozky[i].isFile() == true) {
    System.out.println(polozky[i].getName());
  }
  else {
    System.out.println(polozky[i].getCanonicalPath());
  }
}
```

vypíše např.:

```
b.txt
D:\ppa1\prednasky-priprava\10\images
ppa1-10.xml
D:\ppa1\prednasky-priprava\10\soubory
D:\ppa1\prednasky-priprava\10\vyjimky
```


Kapitola 9. Souborový vstup a výstup

9.1. Proudový vstup a výstup

- na vstupní a výstupní operace lze pohlížet tak, že se jedná o **proud dat** (*stream*)
  - to má výhodu, že **zdroj dat** (*source*) (v případě vstupu) nebo **cíl dat** (*target*) (v případě výstupu) může být různých typů (např. jednou je to klávesnice a podruhé vstupní soubor), ale program tento proud dat zpracovává vždy zcela stejným způsobem
    - prakticky se liší pouze počáteční inicializace zdroje nebo cíle
  - typické zdroje jsou klávesnice, soubor a síťové spojení
  - typické cíle jsou obrazovka, soubor a síťové spojení
- základní vlastností proudu dat je, že „teče spojitě vpřed“
  - není možné se vracet nebo přeskakovat dopředu
    - prakticky vzato, tyto možnosti změny pořadí dat potřebujeme málokdy
- podrobnosti o proudech dat viz v navazujících předmětech
  - zde budou uvedeny pouze nejnutnější poznatky potřebné k praktickému používání
  - omezíme se pouze na souborové I/O

9.1.1. Proudový vstup a výstup s využitím prostředků operačního systému

- tato možnost není příliš známá, byť je velmi užitečná
  - důvodem malé znalosti je všeobecné používání programů s **GUI** (*Graphical User Interface* – **grafické uživatelské rozhraní**)
- aniž by program měl cokoliv se soubory společného, je možné z vnějšku programu snadno zařídit souborový vstup a výstup
  - uživatel toho dosáhne spuštěním programu speciálním způsobem, tzv. **přesměrováním I/O**

Poznámka

Toto je opakování již vyložené problematiky. Zde je uvedeno, aby informace o práci se soubory byla kompletní na jednom místě.

Příklad 9.1. Program pracující pouze s klávesnicí a obrazovkou

```
import java.util.*;

public class PresmerovaniOperacniSystem {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Zadej prvni cislo: ");
    int a = sc.nextInt();
    System.out.print("Zadej druhe cislo: ");
    int b = sc.nextInt();
    int suma = a + b;
    System.out.println("suma = " + suma);
  }
}
```

- v tomto programu není ani náznak práce se vstupními nebo výstupními soubory
  - trik je ale v tom, že program nečte přímo z klávesnice, ale ze `System.in`, který je prostředky operačního systému po spuštění napojen na klávesnici
  - stejně tak výstup není přímo na obrazovku (neadresuje se grafická karta), ale do `System.out`, který je opět propojen pomocí operačního systému na obrazovku
- pokud tento program spustíme z příkazové řádky standardním způsobem dostaneme např.:

```
D:\ppa1>java PresmerovaniOperacniSystem
Zadej prvni cislo: 1
Zadej druhe cislo: 2
suma = 3
```

- pokud si uvědomíme, že zdroj dat nemusí být klávesnice, ale soubor např. s názvem `vstup.txt`, je možné využít příkaz operačního systému a při spuštění určit, že zdrojem dat není klávesnice, ale tento soubor

  obsah souboru `vstup.txt`

```
1
2
```

  nové spuštění programu s přesměrováním vstupu

```
D:\ppa1>java PresmerovaniOperacniSystem < vstup.txt
Zadej prvni cislo: Zadej druhe cislo: suma = 23
```

  program nečeká na vstupy z klávesnice, ale načte je ze souboru `vstup.txt`

- tentýž postup funguje i pro výstup, který lze přesměrovat např. do souboru s názvem `vystup.txt`

  nové spuštění programu s přesměrováním výstupu

  - přesměrují se všechny výpisy, tedy i pomocné výpisy pro zadání prvního a druhého čísla, takže čísla z klávesnice musíme zadávat bez vypsané nápovědy (zde čísla 1 a 2)


```
D:\ppa1>java PresmerovaniOperacniSystem > vystup.txt
1
2
```

obsah souboru `vystup.txt`

```
Zadej prvni cislo: Zadej druhe cislo: suma = 3
```

- oba dva případy lze spojit, tj. zdrojem je `vstup.txt` a cílem je `vystup.txt`
  - obsahy obou souborů jsou stejné, jako v předchozích případech

```
D:\ppa1>java PresmerovaniOperacniSystem < vstup.txt > vystup.txt
```

9.1.2. Proudový vstup a výstup s využitím knihoven Javy

- přesměrování vstupu i výstupu lze provést i programově
  - to má např. výhodu, že vyloučíme možné chyby či omyly při spouštění našeho programu nezkušeným uživatelem, který by zapomněl na příkaz přesměrování

9.1.2.1. Přesměrování vstupu

- k přesměrování vstupu využijeme skutečnost, že třída `Scanner` má několik přetížených konstruktorů
  - ty umožňují stanovit zdroj dat pro objekt `Scanner`u
  - nejčastěji jsou to tyto zdroje
    - klávesnice

```
Scanner sc = new Scanner(System.in);
```

    - soubor

```
Scanner sc = new Scanner(new File("vstup.txt"));
```

  - zůstává zachována hlavní výhoda přesměrování
    - bez ohledu na to, odkud data skutečně přitékají (je to dáno počátečním nastavením `Scanner`u) je následující práce s načítáním hodnot stále stejná
      - používáme stále stejný příkaz, např.: `int a = sc.nextInt();`

Výstraha

Čteme-li pomocí Scanneru reálná čísla, může i při čtení ze souboru nastat problém s použitým desetinným oddělovačem (tzn. `,` (čárka) versus `.` (tečka)). Řešením je např. použít `sc.useLocale(Locale.US);` – viz dříve.

Příklad 9.2. Stejný program jako předchozí, jen s jiným nastavením `Scanneru`

```
import java.util.*;
import java.io.*;

public class PresmerovaniVProgramuVstup {
  public static void main(String[] args) throws IOException {
    Scanner sc = new Scanner(new File("vstup.txt"));
    System.out.print("Zadej prvni cislo: ");
    int a = sc.nextInt();
    System.out.print("Zadej druhe cislo: ");
    int b = sc.nextInt();
    int suma = a + b;
    System.out.println("suma = " + suma);
  }
}
```

- aby bylo možné program přeložit, bylo (oproti předchozímu programu) navíc nutné
  - importovat knihovnu (**balík**, *package*) vstupů a výstupů příkazem:

    ```
    import java.io.*;
    ```

  - ošetřit výjimku, která může vzniknout tím, že uvedený soubor nemusí existovat:
    - zde výjimku ošetřujeme nejprimitivnějším způsobem, tj. deklarací

      ```
      public static void main(String[] args) throws IOException {
      ```

    - jiné způsoby viz dříve
- po spuštění programu dostáváme

```
D:\ppa1>java PresmerovaniVProgramuVstup
Zadej prvni cislo: Zadej druhe cislo: suma = 3
```

- na výstupu je vidět drobná vada na kráse a to zbytečné výzvy uživateli, aby zadal čísla
  - ty lze odstranit mnoha způsoby, např.:

```
import java.util.*;
import java.io.*;

public class PresmerovaniVProgramuVstupLepsi {
  static final String JMENO_SOUBORU = "vstup.txt";
  static Scanner sc;

  public static boolean zajistiZdrojVstupu() {
    boolean klavesnice = true;
    try {
      File f = new File(JMENO_SOUBORU);
      if (f.exists() == true) {
        sc = new Scanner(f);          // ze souboru
        klavesnice = false;
      }
      else {
        sc = new Scanner(System.in);  // z klavesnice
        klavesnice = true;
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    return klavesnice;
  }

  public static void main(String[] args) {
    boolean klavesnice = zajistiZdrojVstupu();
    if (klavesnice == true) {
      System.out.print("Zadej prvni cislo: ");
    }
    int a = sc.nextInt();
    if (klavesnice == true) {
      System.out.print("Zadej druhe cislo: ");
    }
    int b = sc.nextInt();
    int suma = a + b;
    System.out.println("suma = " + suma);
  }
}
```

- v metodě `zajistiZdrojVstupu()` se
  - v případě existence souboru `vstup.txt` nastaví objekt `Scanner` na tento soubor
  - v případě, že soubor `vstup.txt` neexistuje, nastaví se `Scanner` na vstup z klávesnice
  - metoda plně ošetřuje výjimky, takže s nimi „neobtěžuje“ volající metodu
  - metoda vrací booleovskou proměnou s hodnotou `true`, pokud je vstup z klávesnice
    - tato hodnota pak může být použita pro „odfiltrování“ přebytečných pokynů pro uživatele v případě vstupu ze souboru
- pokud čteme soubor, o kterém nevíme, jak je dlouhý, můžeme pro test konce souboru použít metody:
  - `boolean hasNext()` – existuje další řetězec?
  - `boolean hasNextInt()` – existuje další celé číslo?
  - `boolean hasNextDouble()` – existuje další reálné číslo?
  - `boolean hasNextLine()` – existuje další řádka?

  použití viz dále

9.1.2.2. Přesměrování výstupu

- u přesměrování výstupu do souboru není situace tak jednoduchá, jako v případě vstupu

- vypisujeme jinou jazykovou konstrukcí než `System.out.println()`
  - ♦ používáme jen metodu `println()`
- je to vlastně regulérní práce se souborem
- celý princip ale funguje podobně, jako u vstupu
  - ♦ na začátku inicializujeme typ cíle a pak jednotným způsobem zapisujeme

```
p.println("suma = " + suma);
```

  - ♦ po ukončení zápisu do souboru je vhodné soubor uzavřít voláním metody `close()`

```
import java.util.*;
import java.io.*;

public class PresmerovaniVProgramuVystup {
  static boolean obrazovka = false;
  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(new File("vstup.txt"));
    int a = sc.nextInt();
    int b = sc.nextInt();
    int suma = a + b;
    PrintStream p;
    if (obrazovka == true) {
      p = System.out;
    }
    else {
      p = new PrintStream("vystup.txt");
    }
    p.println("suma = " + suma);
    p.close();
  }
}
```

9.2. Textové versus binární soubory

9.2.1. Textové soubory

- jsou (respektive zdají se být) mnohem běžnější
  - připomínají vstup z klávesnice a výstup na obrazovku, které jsou též textové
  - zdrojové kódy jsou v textových souborech
- typická univerzální přípona je `.txt`
  - není to však žádná podmínka, ale dodržovaná konvence
- kromě této přípony existuje u textových souborů množství dalších přípon, které udávají „specializaci“ souboru, např.:
  - `.html` – textové soubory určené pro umístění na WWW

  - `.xml` – textové soubory pro platformově nezávislou výměnu dat
  - i tyto soubory lze číst, vytvářet a měnit libovolnými textovými editory
- textové soubory jsou organizovány po řádcích
  - řádky mohou mít různou délku
    - ♦ končí domluveným(i) znakem(y), který je platformově závislý
      - Windows – dvojice <CR><LF> (`"\r\n"`)
      - Linux – <LF> (`"\n"`)
      - Macintosh – <CR> (`"\r"`)
- výhody oproti binárním souborům:
  - je možné je prohlížet nebo upravovat libovolným textovým editorem
  - mají pochopitelnější obsah
- nevýhody oproti binárním souborům:
  - stejné množství informace zabírá většinou více místa
  - pomalejší zpracovávání – informace se musí překódovat do/z počítačového kódování

9.2.1.1. CSV soubory

- je to speciální druh textových souborů – *comma separated values*
- poměrně často se používají jako přenosový formát pro aplikace, které pracují implicitně s binárními soubory – typicky Excel
- jednotlivé hodnoty jsou na jedné textové řádce oddělené čárkou (*comma*) nebo častěji středníkem
  - tuto řádku lze pak snadno rozdělit na jednotlivé podřetězce – **parsovat**
  - podrobnosti o parsování viz dále

9.2.2. Binární soubory

- při běžném používání počítačů jsou mnohem častější
  - obrázky
  - hudba
  - filmy
  - texty ve Wordu
  - tabulky v Excelu apod.
- jejich vnitřní organizace záleží na:

- dodržení obecně platného formátu
  - ♦ o jaký formát se jedná se pozná z přípony souboru – `.mp3`, `.jpg`, `.avi`
    - – použití jiné přípony většinou způsobí nečitelnost souboru, byť je jeho obsah správně
- fantazii jejich tvůrce – **proprietární formáty**

■ výhody oproti textovým souborům:
- paměťově úspornější
- rychlejší zpracování

■ nevýhody oproti textovým souborům:
- pro každý obecně známý formát je zapotřebí speciální prohlížeč a/nebo editor
  - ♦ u textových souborů stačí pouze jeden (oblíbený) editor
- naprostá nečitelnost v případě nedokumentovaných proprietárních formátů

9.3. Zpracování souborů v Javě

Poznámka

Budeme se zabývat pouze proudově orientovanými postupy zpracování souborů. Soubory s náhodným přístupem viz v literatuře (třída `RandomAccessFile`), stejně jako proudově orientovaná práce s rourou apod.

■ platí zásady, že:
- soubor (pro vstup i výstup) je nutné před prvním použitím otevřít
  - ♦ nezbytné, přičemž otevření **není** příkaz typu:

```
File f = new File("vstup.txt");
```

- po posledním použití uzavřít
  - ♦ není nutné dodržet (respektive překladači nevadí), ale hrozí ztráta dat

■ Java odlišuje zpracování textových souborů a binárních souborů
- pro každý typ dává jiné prostředky, ovšem s podobným chováním
- záleží jen na programátorovi, zda je správně použije

■ všechny třídy jsou v balíku `java.io`, který importujeme příkazem

```
import java.io.*;
```

■ v naprosté většině případů se musíme postarat o ošetření výjimek

9.3.1. Zpracování textových souborů

Poznámka

Ze souboru se načítají bajty, které se v implicitním kódování převzatém z operačního systému (např. windows-1250) převádějí v programu na Unicodové znaky (typ `char`) – podrobnosti viz v KIV/JXT.

9.3.1.1. Nejprimitivnější způsob

■ pro čtení použijeme třídu `FileReader`
- instanci třídy, tzn. otevření souboru pro čtení (např. `text.txt`) je možné provést dvojím způsobem:

1. `FileReader fr = new FileReader(new File("text.txt"));`
2. `FileReader fr = new FileReader("text.txt");`

   častější způsob

- ze souboru můžeme číst:
  - ♦ jednotlivé znaky: `int read()`
    - – skutečně čte znaky, i když je vrací jako typ `int` – to kvůli ukončení čtení
  - ♦ pole znaků: `int read(char[] pole)`

    tento způsob není dobré používat – vždy přečte tolik znaků, kolik je rozsah pole
    - – načtené pole znaků se pak často převádí na `String` příkazem

```
String s = new String(pole);
```

  - ♦ Pozor: v této jednoduché podobě není možné přečíst najednou celou řádku – viz dále
- čtení až do konce souboru
  - ♦ testujeme návratovou hodnotu `-1`, která znamená konec souboru

```
int c;
while ((c = fr.read()) != -1) {
  System.out.print((char) c);    // vypis precteneho znaku na obrazovku
}
```

- uzavření souboru

```
close()
```

- při celém zpracování je třeba ošetřit výjimku typu `IOException`

Příklad 9.3. Čtení souboru po znacích a jejich kontrolní výpis na obrazovku

```
import java.io.*;

public class CteniSouboruPoZnacich {
  public static void main(String[] args) {
    try {
      FileReader fr = new FileReader("text.txt");
      int c;
      while ((c = fr.read()) != -1) {
        System.out.print((char) c);
      }
      fr.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

Poznámka

Tento primitivní způsob je vhodný pouze v případě, že chceme číst soubor po znacích. Chceme-li číst formátovaně čísla, je vhodné použít již známý `Scanner` nebo způsob uvedený dále.

- pro zápis použijeme třídu `FileWriter`
  - instanci třídy, tzn. otevření souboru pro zápis (např. `zapis.txt`) je možné provést dvojím způsobem:
    1. `FileWriter fw = new FileWriter(new File("zapis.txt"));`
    2. `FileWriter fw = new FileWriter("zapis.txt");`

       častější způsob
  - do souboru můžeme zapisovat:
    - jednotlivé znaky: `void write(int c)`
    - řetezec: `void write(String str)`
    - o odřádkování se musíme postarat sami
  - uzavření souboru

    `close()`
  - při celém zpracování je třeba ošetřit výjimku typu `IOException`

Příklad 9.4. Zápis znaků a řetězce do souboru

```
import java.io.*;

public class ZapisDoSouboruZnaky {
  public static void main(String[] args) {
    try {
      FileWriter fw = new FileWriter("zapis.txt");
      char c = 'A';
      fw.write(c);
      fw.write("hoj lidi\n");
      String s = "jak se mate\r\n";
      fw.write(s);
      fw.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

- tento způsob má nevýhodu, že se musíme postarat o odřádkování a navíc i o správnou kombinaci znaků na konci řádky
  - v prvním případě je řádka ukončena pouze `\n`, což ve Windows může způsobit v některých editorech potíže
  - v druhém případě je ukončena `\r\n`, což je správně pro Windows ale nadbytečně pro Linux
  - systémové řešení viz dále
- významná nevýhoda totoho primitivního způsobu je, že neumožňuje elegantní zápis čísel
  - čísla (typu `int` i `double`) se musejí před zápisem převést na `String` např.:

    `fw.write("" + i);`
  - elegantnější řešení viz dále

9.3.1.2. Vylepšený způsob

- nad uvedeným primitivním zpracováním lze snadno „vystavět nadstavbu“, která práci se soubory zpříjemní
  - použijí se doplňkové třídy tzv. **filtry** nebo **vlastnosti**
    - bufferování – třídy `BufferedReader` a `BufferedWriter`
      - významně urychlují zpracování souborů použitím **vyrovnávací paměti** (*buffer*)
      - `BufferedReader` umožňuje čtení po řádcích, přičemž znaky odřádkování (`\n` nebo `\r\n`) automaticky odřezává

metoda: `String readLine()`

- v případě, že již není co číst (tj. konec souboru) vrací `null`

Poznámka

Třída `Scanner` dává stejné nebo větší možnosti čtení jako `BufferedReader`, ale ta je asi 10krát rychlejší. Pro čtení krátkých souborů lze ale bez problémů používat již známý `Scanner`.

– `BufferedWriter` umožňuje transparentní zápis konce řádky – zvolí se podle aktuální platformy

metoda: `void newLine()`

Poznámka

Pokud používáme výstupní formátování, je tato metoda nahrazena známou metodou `println()`.

♦ výstupní formátování – třída `PrintWriter`

– dává k dispozici známé metody `print()`, `println()` a `format()`

Poznámka

Vstupní formátování, tj. převod řetězců na číslice je zde zajišťováno pomocí známé třídy `Scanner`.

● způsob zapojení tříd vlastností

♦ v konstruktorech se postupně vytvářejí objekty všech zainteresovaných tříd

Příklad 9.5. Čtení souboru s celými čísly, násobení těchto čísel dvěma a zapsání výsledků do souboru

```
import java.util.*;
import java.io.*;

public class CteniAZapisCiselDoSouboru {
  public static void main(String[] args) {
    try {
      Scanner sc = new Scanner(new File("cisla.txt"));
      PrintWriter pw = new PrintWriter(
                       new BufferedWriter(
                       new FileWriter("nasobky.txt")));
      while (sc.hasNextInt() == true) {
        int i = sc.nextInt();
        pw.println(i * 2);
      }
      sc.close();
      pw.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

■ soubor pro zápis má přidané vlastnosti:

- buferování – kvůli rychlosti
- formátovaný zápis – kvůli eleganci převodu celých čísel na řetězec a jejich zápisu

■ i soubor pro čtení se doporučuje uzavřít – `sc.close();`

Příklad 9.6. Tentýž příklad – rychlé čtení

```
import java.io.*;

public class CteniAZapisCiselDoSouboruRychle {
  public static void main(String[] args) {
    try {
      BufferedReader bfr = new BufferedReader(
                           new FileReader("cisla.txt"));

      PrintWriter pw = new PrintWriter(
                       new BufferedWriter(
                       new FileWriter("nasobky2.txt")));
      String radka;
      while ((radka = bfr.readLine()) != null) {
        int i = Integer.parseInt(radka);
        pw.println(i * 2);
      }
      bfr.close();
      pw.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

- soubor pro čtení má přidanou vlastnost buferování
- načítáme celou řádku, na které je ale jen jedno číslo
  - to pak musíme převést z řetězce na typ `int`, aby bylo možné provést aritmetickou operaci násobení 2
- tento program je o něco málo komplikovanější než předchozí, ale asi 30krát rychlejší

Příklad 9.7. Příklad na převedení souboru na velká písmena po řádcích

```
import java.io.*;

public class CteniAZapisRetezcuDoSouboru {
  public static void main(String[] args) {
    try {
      BufferedReader bfr = new BufferedReader(
                           new FileReader("text.txt"));

      PrintWriter pw = new PrintWriter(
                       new BufferedWriter(
                       new FileWriter("velky-text.txt")));
      String radka;
      while ((radka = bfr.readLine()) != null) {
        radka = radka.toUpperCase();
        pw.println(radka);
      }
      bfr.close();
      pw.close();
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

- metoda `readLine()` odstraní znaky konce řádky
- metoda `println()` je tam zase přidá

Příklad 9.8. Příklad transparentního řešení desetinného oddělovače

- při používání `Scanner`u nastávaly problémy s desetinným oddělovačem (tečka versus čárka – 3.14 versus 3,14)
  - tento problém lze elegantně řešit načtením čísla z textového vstupu jako řetězce
    - pak v řetězci provedeme záměnu čárky za tečku
    - řetězec převedeme na `double` číslo
  - toto řešení lze uplatnit jak pro `Scanner` tak i po `FileReader`

```
public static double preved(String s) {
  s = s.replace(',', '.');
  double d = Double.parseDouble(s);
  return d;
}

public static void main(String[] args) {
  try {
    String s;
    Scanner sc = new Scanner(new File("desetinny-odd.txt"));
    s = sc.nextLine();
    System.out.println(s + " --> " + preved(s));
    s = sc.nextLine();
    System.out.println(s + " --> " + preved(s));
    sc.close();

    BufferedReader bfr = new BufferedReader(
          new FileReader(new File("desetinny-odd.txt")));
    s = bfr.readLine();
    System.out.println(s + " --> " + preved(s));
    s = bfr.readLine();
    System.out.println(s + " --> " + preved(s));
    bfr.close();
  } ...
```

- vypíše např.:

```
3.14 --> 3.14
6,28 --> 6.28
3.14 --> 3.14
6,28 --> 6.28
```

9.3.1.3. Ošetření výjimek a uzavírání souboru

Poznámka

Pouze pro zájemce.

- předchozí (a i následující) ukázky se z důvodů jednoduchosti příliš nezabývají problematikou ošetření výjimek a uzavírání souboru

- tyto postupy jsou vhodné jen pro nejjednodušší školní případy
  - v případě výskytu problému (výjimky) je na konzoli vypsán chybový výpis (`e.printStackTrace()`) a program je ukončen (`System.exit(1);`)
    - je třeba si ale uvědomit, že otevřený soubor se neuzavře pomocí `close()`, protože tato část kódu v `try` již neproběhne
      - to ale není fakticky problém, protože `System.exit()` zaručuje, že mimo jiné uzavře všechny otevřené soubory

- programujeme-li složitější program, není možné tento postup použít

1. program má v naprosté většině grafické uživatelské rozhraní (GUI), kde není výpis na konzoli vůbec vidět
2. program je složitý a nechceme, aby při nenalezeném souboru okamžitě skončil
   - spíše chceme, aby mohl uživatel na problém reagovat např. volbou jiného souboru
3. všechny otevřené soubory je nutno uzavřít, jinak dochází k postupnému vyčerpávání zdrojů a celý program se někdy zhroutí

1. **řešení problému s GUI**

- pro výpis chyby použijeme výpis do okénka (místo výpisu na konzoli)
- používáme statickou metodu `showMessageDialog()` třídy `javax.swing.JOptionPane`, která má čtyři parametry

a. odkaz na nadřazenou komponentu – typicky jej nevyužíváme, proto je `null`

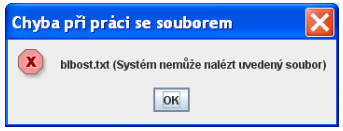
b. vlastní zpráva – typicky ji získáme výpisem ze zachycené výjimky pomocí `e.getLocalized-Message()`

c. titulek okna – typicky řetězec `"Chyba při práci se souborem"`

d. typ okna – určuje se konstantou, typicky `JOptionPane.ERROR_MESSAGE`

```
catch (IOException e) {
  JOptionPane.showMessageDialog(
        null,
        e.getLocalizedMessage(),
        "Chyba při práci se souborem",
        JOptionPane.ERROR_MESSAGE);
}
```

při chybně zadaném názvu souboru (zde `blbost.txt`) zobrazí okénko:



2. **řešení problému okamžitého ukončení programu**

- okamžité ukončení programu pomocí `System.exit(1);` bylo vhodné, protože v případě výjimky nebyla načtena požadovaná data a nemělo tedy smysl v programu dále pokračovat
- v rozsáhlých programech ale použijeme způsob deklarace výjimky (tzn. jejího předání výše) než ošetření výjimky pomocí `try-catch`
  - nadřazená úroveň se pak musí postarat o nápravu
  - toto je převládající postup u vícevrstvých programů a nemá nic společného s leností programátora, který píše kód pracující se souborem

3. **řešení problému neuzavřeného souboru**

- využijeme konstrukci `try-catch-finally` (viz dříve výjimky), kde soubor zavíráme v části `finally`, která musí vždy proběhnout
  - je zde malá nepříjemnost – při uzavírání souboru metodou `close()` je opět nutné ošetřit výjimku, čili konstrukce `try` se zanořují
    - při reakci na zachycenou výjimku v bloku `catch` se používá změna typu výjimky na `RuntimeException` – viz dříve

Varování

Všechna tři dosud popsaná řešení se dají používat v kombinacích společně. Je ale nutné rozvážit, co vlastně od programu chceme a zvolit vhodnou kombinaci!

Příklad načítání obsahu souboru do jednoho řetězce, který je metodou vrácen. Pokud se při čtení vyskytne chyba, je o tom uživatel informován v novém okénku. Pokud byl soubor otevřen, je vždy i uzavřen v bloku `finally`. Případný problém s uzavíráním souboru je řešen vyhozením Runtime výjimky, kterou není nutné dále nijak ošetřovat.

```
import java.io.*;
import javax.swing.JOptionPane;

public class CteniSouboru {

  public static String prectiObsahSouboru(String jmenoSouboru) {
    String obsahSouboru = "";
    BufferedReader bfr = null;
    try {
      bfr = new BufferedReader(
            new FileReader(
            new File(jmenoSouboru)));

      String radka = null;
      while ((radka = bfr.readLine()) != null) {
        obsahSouboru += radka;
      }
    }
    catch (IOException e) {
      // chybové okénko
      JOptionPane.showMessageDialog(
          null,
```



```
          e.getLocalizedMessage(),
          "Chyba při práci se souborem",
          JOptionPane.ERROR_MESSAGE);
    }
    finally {
      if (bfr != null) {
        // uzavírání souboru, pokud byl předtím otevřen
        try {
          bfr.close();
        }
        catch (IOException ex) {
          throw new RuntimeException(ex);
        }
      }
      if (obsahSouboru.length() == 0) {
        // ukončení programu pokud nebyla načtena žádná data
        System.exit(1);
      }
      return obsahSouboru;
    }
  }

  public static final void main(String[] args) {
    String text;

    text = prectiObsahSouboru("data.txt");
    System.out.println(text);

    text = prectiObsahSouboru("blbost.txt");
    System.out.println(text);
  }
}
```

9.3.2. Zpracování CSV souborů – parsování řetězců

Poznámka

Pouze pro zájemce.

- řešíme problém, kdy ve vstupním souboru je na jedné řádce několik čísel (obecně hodnot)
  - nejčastější oddělovače jsou
    - mezera nebo mezery
    - středník nebo čárka – často soubory s příponou `.csv` (*Comma Separated Values*)
    - dvojtečka
- načteme celou řádku jako řetězec
  - je jednodušší rozdělit řetězec na jednotlivé podřetězce (parsovat) v programu než jej postupně načítat ze souboru

- před převodem jednotlivých číslic je nutné načtenou řádku **parsovat**
  - dříve se používala třída `java.util.StringTokenizer`
  - jednodušší způsob dnes (od JDK 1.4) je použití metody `split()` třídy `String`
    - metoda rozdělí původní řetězec na jednotlivé podřetězce, které umístí do vráceného pole řetězců
    - pak se postupně převedou všechny podřetězce na čísla

9.3.2.1. Základní použití `split()`

- jednotlivé podřetězce jsou odděleny jen jedním unikátním dopředu známým znakem (viz nejčastější oddělovače výše)
- oddělovací znak se zapíše jako skutečný parametr metody `split()`

Příklad 9.9. Rozdělení podřetězců oddělených mezerou

```
String radka = "123 45 6 789";
String[] podretezce = radka.split(" ");
for (int i = 0;  i < podretezce.length;  i++) {
  System.out.println("|" + podretezce[i] + "|");
}
```

vypíše

```
|123|
|45|
|6|
|789|
```

z výpisu je vidět, že oddělovací mezery jsou „požrány“

- pro jiný oddělovač je situace analogická

```
String radka = "123;45;6;789";
String[] podretezce = radka.split(";");
```

- je-li v původním řetězci některý z oddělovacích znaků zdvojen, bude výsledkem prázdný řetězec

```
String radka = "123  45 6 789";
String[] podretezce = radka.split(" ");
```

vypíše

```
|123|
||
|45|
|6|
|789|
```


- to prakticky znamená, že když si nejsme jisti vícečetností oddělovačů, musíme pak ve výsledném poli řetězců testovat výskyt prázdného řetězce, např.:

```
String radka = "123  45 6 789";
String[] podretezce = radka.split(" ");
for (int i = 0;  i < podretezce.length;  i++) {
  if (podretezce[i].length() > 0) {
    int cislo = Integer.parseInt(podretezce[i]);
    System.out.println(cislo);
  }
}
```

vypíše

```
123
45
6
789
```

- je-li v původním řetězci některý z oddělovacích znaků na samém konci nebo je-li zdvojen na samém konci, není prázdný řetězec součástí pole řetězců

```
String radka = "123;45;6;789;;";
String[] podretezce = radka.split(";");
```

vypíše

```
|123|
|45|
|6|
|789|
```

9.3.2.2. Použití regulárních výrazů

- zde bude ukázáno pouze základní použití – pro sofistikované použití viz `java.util.regex.Pattern`
- jednodušší případ je, když jsou oddělovače sice různé, ale je jich konečný počet (jsou konkrétně předem známy)
  - pak se všechny potenciální oddělovače vypíší do hranatých závorek
    - na jejich pořadí nezáleží
    - pokud bude nějaký oddělovač navíc, nevadí to

```
String radka = "123 45;6:789";
String[] podretezce = radka.split("[; >:]");
for (int i = 0;  i < podretezce.length;  i++) {
  System.out.println("|" + podretezce[i] + "|");
}
```

vypíše (znak `>` nebyl v původním řetězci a proto nebyl použit)

```
|123|
|45|
|6|
|789|
```

Varování

Snažíme se uvádět co nejmenší počet možných oddělovačů, protože některé znaky mohou mít v regulárních výrazech jiný význam. Typickým problémovým znakem je znak `.` (tečka).

```
String radka = "soubor.txt";
String[] podretezce = radka.split("[.]");
```

správné použití, které vypíše

```
|soubor|
|txt|
```

nesprávné použití (bez hranatých závorek) je:

```
String radka = "soubor.txt";
String[] podretezce = radka.split(".");
```

to nevypíše nic, protože tečka má význam „jakýkoliv znak“, takže jsou všechny znaky vstupního řetězce považovány za oddělovače a „požerou“ se

- komplikovanější případ je, když dopředu neznáme konkrétní znaky, ale pouze jejich kategorie
  - v podstatě až zde by se mělo mluvit o regulárních výrazech
  - znaky stejné kategorie lze zadat jako jeden výraz
    - v dokumentaci k `java.util.regex.Pattern` je uvedeno přehledně pět různých oblastí skupin znaků (plus další možnosti)
    - pro běžné použití stačí pouze jedna oblast – *POSIX character classes*
      - Pozor na to, že fungují jen pro znakovou sadu US-ASCII (tj. ne na akcentovaná písmena – viz později)!
    - přehled možností *POSIX character classes*
      - `\p{Lower}` – malá písmena
      - `\p{Upper}` – velká písmena
      - `\p{Alpha}` – malá a velká písmena
      - `\p{Digit}` – číslice
      - `\p{XDigit}` – hexadecimální číslice
      - `\p{Alnum}` – malá a velká písmena a číslice
      - `\p{Punct}` – interpunkce – libovolný ze znaků `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`
      - `\p{Blank}` – mezera a tabulátor
      - `\p{Space}` – bílé znaky – mezera, tabulátor, <CR>, <LF>, nová stránka
      - `\p{Cntrl}` – řídící znaky – `\x00` až `\x1F`
      - `\p{Graph}` – viditelné znaky – `\p{Alnum}` a `\p{Punct}`
      - `\p{Print}` – tisknutelné znaky – `\p{Graph}` a mezera
      - `\p{ASCII}` – všechny ASCII znaky, tj. sedmibitové znaky
    - Pozor: při použití ve zdrojovém kódu je nutno zdvojit úvodní zpětné lomítko!
    - použití oddělovačů z kategorie `\p{Punct}`

```
String radka = "soubor.txt";
String[] podretezce = radka.split("\\p{Punct}");
```

    - použití oddělovačů z kategorie `\p{Alpha}`

```
String radka = "123a45A6";
String[] podretezce = radka.split("\\p{Alpha}");
```

vypíše

```
|123|
|45|
|6|
```

Poznámka

Regulární výrazy se nepoužívají jen pro parsování řetězců, ale za stejných podmínek např. i pro vyhledávání vzorů v řetězci – viz metodu `String.matches()`.

9.3.3. Zpracování binárních souborů

Poznámka

1. Pouze pro zájemce.
2. Ze souboru se načítají bajty. Neprobíhá žádná konverze – bajt v souboru je tentýž bajt v programu.
3. Omezíme se jen na binární soubory složené z dat primitivních datových typů (nikoliv objektů).
4. Všechna data jsou v pořadí *big-endian* – viz později.

- binární soubory čteme a vytváříme (v začátcích programování) málokdy
- platí velmi podobné postupy, jako u textových souborů, jen se používané třídy jinak jmenují
- používat primitivní způsob nemá prakticky význam – je nutné současně použít třídu vlastností

9.3.3.1. Základní třídy

- pro čtení použijeme třídu `FileInputStream`
  - instanci třídy, tzn. otevření souboru pro čtení (např. `data.bin`) je možné provést dvojím způsobem:
    1. `FileInputStream fis = new FileInputStream(new File("data.bin"));`
    2. `FileInputStream fis = new FileInputStream("data.bin");`

       častější způsob
  - ze souboru můžeme číst:
    - jednotlivé bajty: `int read()`
      - skutečně čte bajty, i když je vrací jako typ `int` – to kvůli ukončení čtení
    - pole bajtů: `int read(byte[] pole)`

      vždy přečte tolik bajtů, kolik je rozsah pole
  - uzavření souboru

    `close()`
  - při celém zpracování je třeba ošetřit výjimku typu `IOException`
- pro zápis použijeme třídu `FileOutputStream`
  - instanci třídy, tzn. otevření souboru pro zápis (např. `vystupni-data.bin`) je možné provést dvojím způsobem:
    1. `FileOutputStream fos = new FileOutputStream(new File("vystupni-data.bin"));`
    2. `FileOutputStream fos = new FileOutputStream("vystupni-data.bin");`

       častější způsob
  - do souboru můžeme zapisovat smysluplně pouze jednotlivé bajty: `void write(int b)`
  - uzavření souboru

    `close()`
  - při celém zpracování je třeba ošetřit výjimku typu `IOException`

9.3.3.2. Třídy vlastností

- používáme již známé buferování – zde jen pro zvýšení rychlosti práce se soubory
  - `BufferedInputStream`
  - `BufferedOutputStream`

- klíčové jsou třídy
  - `DataInputStream` – čtení primitivních datových typů
    - dává k dispozici metody
      - `int readInt()` – přečte jeden `int`
      - `double readDouble()` – přečte jeden `double`
      - a další stejně významově pojmenované metody
    - pokud již v souboru není co číst, vyhazují tyto metody výjimku `EOFException`
      - toho se využívá k ukončení čtení souboru neznámé délky
        - čtení provádíme v nekonečném cyklu, který je ukončen ošetřením výjimky `EOFException`
  - `DataOutputStream` – zápis primitivních datových typů
    - dává k dispozici metody
      - `void writeInt(int i)` – zapíše jeden `int` jako posloupnost čtyř bajtů
      - `void writeDouble(double d)` – zapíše jeden `double`
      - a další stejně významově pojmenované metody
- je třeba si uvědomit, že záleží jen na programátorovi, zda smíchá různé primitivní datové typy do jednoho souboru
  - pokud ano, musí se v něm vyznat a nejlépe tento soubor ihned zdokumentovat
  - vhodnější je, aby byl binární soubor složen jen z dat stejného typu

Příklad 9.10. Zápis čísel do binárního souboru

```
import java.io.*;

public class ZapisBinCiselDoSouboru {
  static final int MAX = 1000000;

  public static void main(String[] args) {
    try {
      DataOutputStream dos = new DataOutputStream(
                             new BufferedOutputStream (
          new FileOutputStream("testovaci-cisla.bin")));

      for (int i = 1;  i <= MAX;  i++) {
        dos.writeInt(i);
      }
      dos.close();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

- soubor pro zápis má přidané vlastnosti:
  - buferování – kvůli rychlosti
  - binární zápis – kvůli možnosti zápisu datového typu `int` do souboru

Příklad 9.11. Čtení čísel z binárního souboru a výpočet jejich sumy

- v programu jsou použity dva vnořené bloky `try-catch`
  - vnější blok je pro již známé ošetření vjimky typu `IOException` – nenalezení souboru, chyba při čtení apod.
    - tato výjimka se pravděvděpodobně nikdy nevyskytne, proto ji nijak zvlášť neošetřujeme
  - vnitřní blok ošetřuje výjimku typu `EOFException`
    - tato výjimka se vyskytne vždy – při dosažení konce souboru, proto v ní můžeme uzařít soubor a vypsat závěrečný výsledek

```
import java.io.*;

public class CteniBinCiselZeSouboru {
  public static void main(String[] args) {
    DataInputStream dis = null;
    try {
      dis = new DataInputStream(
            new BufferedInputStream (
            new FileInputStream("testovaci-cisla.bin")));

      long suma = 0;
      try {
        while (true) {
          suma += dis.readInt();
        }
      }
      catch (EOFException e) {
        dis.close();
        System.out.println("suma: " + suma);
      }
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

- soubor pro čtení má přidané vlastnosti:
  - buferování – kvůli rychlosti
  - binární zápis – kvůli možnosti čtení datového typu `int` ze souboru

9.3.4. Porovnání rychlostí zpracování textových a binárních souborů

- budeme číst soubory obsahující celá čísla od 1 do 1 000 000

- textový soubor `testovaci-cisla.txt` má každé číslo na samostatné řádce a velikost 7 888 896 B
  - ♦ čteme pomocí
    1. `Scanner`
    2. `BufferedReader`
- binární soubor `testovaci-cisla.bin` obsahuje čísla typu `int` a má velikost 4 000 000 B
  - ♦ čteme pomocí `BufferedInputStream`

■ v obou souborech je stejné množství informace, pouze jinak zakódované!

■ z těchto čísel budeme počítat jejich sumu, která musí být vždy 500 000 500 000

■ pokusy byly prováděny na Pentium 4 Centrino 1,5 GHz

čtení pomocí	**typ souboru**	**čas** [msec]
`Scanner`	`.txt`	6 960
`BufferedReader`	`.txt`	880
`BufferedInputStream`	`.bin`	140

Kapitola 10. Vyhledávání v poli

10.1. Obecné informace

■ často vykonávaná činnost

■ hledáme, zda se prvek určité hodnoty (též prvek s klíčem) nalézá v poli, s výsledkem:

- ● někdy stačí pouze informace typu *true* / *false*
- ● většinou ale požadujeme i index nalezeného prvku

■ máme možnosti:

1. prvek není v poli – je třeba o nenalezení informovat nějakou domluvenou hodnotou
   - ● často hodnota –1, což nemůže být použitý index
2. prvek je v poli právě jednou
3. prvek je v poli vícekrát

■ pole lze rozlišovat podle

1. organizace pole
   a. neuspořádané – tak jak prvky postupně přicházely
   b. uspořádané – před vyhledáváním proběhlo seřazení pole
2. zaplnění pole
   a. zaplněné jen částečně – některé prvky na konci pole nejsou použity
   b. zcela zaplněné

Poznámka

Principy vyhledávání jsou poměrně jednoduché. Budou ale uváděny podrobně, protože na nich lze ukázat některé techniky programování či algoritmizace.

10.2. Neuspořádané pole

■ pole prohledáváme vždy sekvenčně – od prvního prvku v pořadí

Poznámka

Pokud nalezneme vyhovující prvek, hledání končí bez ohledu na to, zda se v poli mohou dále vyskytovat prvky se stejnou hodnotou. Řešení opakujícího se výskytu bude uvedeno dále.

10.2.1. Pole není zcela zaplněné

■ vždy předpokládáme, že nevyužité prvky jsou na konci pole

- informace o tom, kolik prvků pole je aktuálně použito, může být:
  - známou hodnotou indexu koncového prvku
    - výhodný a běžně používaný způsob
    - dvě možnosti indexu:
      - naposledy použitý (nejvyšší)
      - první nepoužitý – výhody:
        - ukončovací podmínka je dle zvyku < nikoliv <=
        - metody lze použít i pro celé pole, kdy koncový index je *pole*.length
    - vyskytuje se i v knihovnách Javy zásadně ve verzi „první nepoužitý“
    - v příkladu v podmínce cyklu `for` nemusí být dvojitá podmínka typu

```
i < koncovyIndex && pole[i] != hodnota
```

protože při nalezeném výskytu prvku je cyklus okamžitě opuštěn pomocí příkazu `return`

```
final static int NEUSPECH = -1;

/**
  nalezne prvni vyskyt prvku v poli
  @param pole prohledavane pole
  @param hodnota hledana hodnota
  @param koncovyIndex prvni nevyuzity index
  @return <ul>
            <li> index nalezeneho prvku v pripade uspechu</li>
            <li> NEUSPECH (-1) v pripade neuspechu</li>
          </ul>
 */
public static int nalezniPrvek(int[] pole, int hodnota,
                               int koncovyIndex) {
  for (int i = 0; i < koncovyIndex;  i++) {
    if (pole[i] == hodnota) {
      return i;         // uspech
    }
  }
  return NEUSPECH;      // neuspech
}
```

pro vypsání výsledku použijeme metodu

```
public static void vypisVysledek(int[] pole, int hodnota,
                                 int vysledek) {
  System.out.println("Prvek s hodnotou " + hodnota
                     + " v poli:");
  System.out.println(Arrays.toString(pole));
  if (vysledek != NEUSPECH) {
    System.out.println("je na indexu: " + vysledek);
  }
  else {
    System.out.println("neni");
  }
}
```

v `main()` se obě metody použijí pro zcela zaplněné pole

```
int[] poleCele = {5, 1, 4, 2, 3};
int vys;
int prvek = 2;
vys = nalezniPrvek(poleCele, prvek, poleCele.length);
vypisVysledek(poleCele, prvek, vys);
prvek = 6;
vys = nalezniPrvek(poleCele, prvek, poleCele.length);
vypisVysledek(poleCele, prvek, vys);
```

a vypíše se

```
Prvek s hodnotou 2 v poli:
[5, 1, 4, 2, 3]
je na indexu: 3
Prvek s hodnotou 6 v poli:
[5, 1, 4, 2, 3]
neni
```

pokud použijeme tyto metody na nezaplněné pole, je použití velmi podobné

```
// vytvoreni nezaplneneho pole
int[] poleNecele = new int[poleCele.length + 3];
for (int i = 0;  i < poleCele.length;  i++) {
  poleNecele[i] = poleCele[i];
}
int nepouzityIndex = poleCele.length;

prvek = 2;
vys = nalezniPrvek(poleNecele, prvek, nepouzityIndex);
vypisVysledek(poleNecele, prvek, vys);
prvek = 6;
vys = nalezniPrvek(poleNecele, prvek, nepouzityIndex);
vypisVysledek(poleNecele, prvek, vys);
```

a vypíše se

```
Prvek s hodnotou 2 v poli:
[5, 1, 4, 2, 3, 0, 0, 0]
je na indexu: 3
Prvek s hodnotou 6 v poli:
[5, 1, 4, 2, 3, 0, 0, 0]
neni
```

  - vyplněním nepoužitých prvků pole domluvenou hodnotou (dále konstanta `VYPLN`)
    - ta se nesmí vyskytnout mezi použitými prvky

- stanovit hodnotu konstanty `VYPLN` bývá obtížné a časem to může být zdroj nepříjemných chyb (viz problémy roku 2000, kdy nepoužité hodnoty byly 00)
  - častá hodnota je např. `Integer.MIN_VALUE`
- nepoužívat, pokud nemáme skutečně dobrý důvod
- v podmínce cyklu `for` většinou je dvojitá podmínka typu

```
i < pole.length && pole[i] != VYPLN
```

```
  final static int NEUSPECH = -1;
  final static int VYPLN = Integer.MIN_VALUE;

  /**
    nalezne prvni vyskyt prvku v poli;
    <br>
    nepouzite prvky maji hodnotu <code>VYPLN</code>
    @param pole prohledavane pole
    @param hodnota hledana hodnota
    @return <ul>
              <li> index nalezeneho prvku v pripade uspechu</li>
              <li> NEUSPECH (-1) v pripade neuspechu</li>
             </ul>
   */
  public static int nalezniPrvek(int[] pole, int hodnota) {
    for (int i = 0;
         i < pole.length  &&  pole[i] != VYPLN;  i++) {
      if (pole[i] == hodnota) {
        return i;   // uspech
      }
    }
    return NEUSPECH;      // neuspech
  }
```

pro vypsání výsledku použijeme metodu `vypisVysledek()` zcela stejně, jako v předchozím případě

v `main()` se obě metody použijí pro zcela zaplněné pole

```
    int[] poleCele = {5, 1, 4, 2, 3};
    int vys;
    int prvek = 2;
    vys = nalezniPrvek(poleCele, prvek);
    vypisVysledek(poleCele, prvek, vys);
    prvek = 6;
    vys = nalezniPrvek(poleCele, prvek);
    vypisVysledek(poleCele, prvek, vys);
```

a vypíše se

```
Prvek s hodnotou 2 v poli:
[5, 1, 4, 2, 3]
je na indexu: 3
```



```
Prvek s hodnotou 6 v poli:
[5, 1, 4, 2, 3]
neni
```

pokud použijeme tyto metody na nezaplněné pole, je použití velmi podobné

pro vyplnění nepoužitých částí pole byla použita knihovní metoda

```
Arrays.fill(int[] pole, int pocatecniIndex, int koncovyIndex, int vypln)
```

```
    // vytvoreni nezaplneneho pole
    int[] poleNecele = new int[poleCele.length + 3];
    for (int i = 0;  i < poleCele.length;  i++) {
      poleNecele[i] = poleCele[i];
    }
    Arrays.fill(poleNecele, poleCele.length,
                poleNecele.length, VYPLN);

    prvek = 2;
    vys = nalezniPrvek(poleNecele, prvek);
    vypisVysledek(poleNecele, prvek, vys);
    prvek = 6;
    vys = nalezniPrvek(poleNecele, prvek);
    vypisVysledek(poleNecele, prvek, vys);
```

a vypíše se

```
Prvek s hodnotou 2 v poli:
[5, 1, 4, 2, 3, -2147483648, -2147483648, -2147483648]
je na indexu: 3
Prvek s hodnotou 6 v poli:
[5, 1, 4, 2, 3, -2147483648, -2147483648, -2147483648]
neni
```

- použití **zarážky**
  - zarážka je speciální hodnota, kterou předem uložíme do prvního nevyužitého prvku pole
  - pomocí ní se vyhneme dvojité podmínce ukončení cyklu
  - nevýhody
    - lze použít jen pro nezaplněné pole
    - použitím zarážky pole modifikujeme a po skončení vyhledávání prvku jej nesmíme zapomenout vrátit do původního stavu
  - nevýhody převažují – pro použití zarážky musí být skutečně dobrý důvod
  - z příkladu je vidět, že ušetření jedné podmínky je vyváženo zvýšenou administrativou

```
public class VyhledavaniNeserazeneZarazka {
  final static int NEUSPECH = -1;

  /**
```

```
    nalezne prvni vyskyt prvku v poli;
    <br>
    prvni nepouzity prvek ma docasne hodnotu hledaneho prvku
    @param pole prohledavane pole
    @param hodnota hledana hodnota
    @return index nalezeneho prvku nebo NEUSPECH (-1)
   */
  public static int nalezniPrvek(int[] pole, int hodnota,
                                 int nepouzityIndex) {
    int pom = pole[nepouzityIndex];
    pole[nepouzityIndex] = hodnota;  // nastaveni zarazky
    int i;
    for (i = 0; i < pole.length;  i++) {
      if (pole[i] == hodnota) {
        break;
      }
    }
    pole[nepouzityIndex] = pom;  // obnoveni puvodni hodnoty
    if (i == nepouzityIndex) {
      return NEUSPECH;
    }
    return i;   // uspech
  }

  public static void vypisVysledek(int[] pole, int hodnota,
                                   int vysledek) {
    ...
  }

  public static void main(String[] args) {
    int[] poleCele = {5, 1, 4, 2, 3};
    int vys;
    int prvek;
    // vytvoreni nezaplneneho pole
    int[] poleNecele = new int[poleCele.length + 3];
    for (int i = 0;  i < poleCele.length;  i++) {
      poleNecele[i] = poleCele[i];
    }
    int nepouzityIndex = poleCele.length;

    prvek = 2;
    vys = nalezniPrvek(poleNecele, prvek, nepouzityIndex);
    vypisVysledek(poleNecele, prvek, vys);

    prvek = 6;
    vys = nalezniPrvek(poleNecele, prvek, nepouzityIndex);
    vypisVysledek(poleNecele, prvek, vys);
  }
}
```


10.2.2. Pole je zcela zaplněné

- nejčastější stav
- odpadá nutnost výplně nebo koncového indexu
  - tím se zjednoduší jak vyhledávací metoda, tak i její použití

```
final static int NEUSPECH = -1;

/**
  nalezne prvni vyskyt prvku v poli
  @param pole prohledavane pole
  @param hodnota hledana hodnota
  @return <ul>
            <li> index nalezeneho prvku v pripade uspechu</li>
            <li> NEUSPECH (-1) v pripade neuspechu</li>
          </ul>
 */
public static int nalezniPrvek(int[] pole, int hodnota) {
  for (int i = 0; i < pole.length;  i++) {
    if (pole[i] == hodnota) {
      return i;   // uspech
    }
  }
  return NEUSPECH;
}
```

pro vypsání výsledku použijeme metodu `vypisVysledek()` zcela stejně, jako v předchozích případech

```
public static void main(String[] args) {
  int[] poleCele = {5, 1, 4, 2, 3};
  int vys;
  int prvek = 2;
  vys = nalezniPrvek(poleCele, prvek);
  vypisVysledek(poleCele, prvek, vys);
  prvek = 6;
  vys = nalezniPrvek(poleCele, prvek);
  vypisVysledek(poleCele, prvek, vys);
}
```

10.3. Opakující se výskyt prvků

- chceme-li informaci o všech shodných prvcích, je výhodné, aby vyhledávací metoda vracela pole s indexy nalezených prvků
  - použijeme návratovou hodnotu `null` v případě, že nebudou nalezeny žádné výskyty prvku
    - `null` je klíčové slovo, které (v tomto případě) říká „žádné pole“

```
/**
  nalezne vsechny vyskyt prvku v poli
```

```
    @param pole prohledavane pole
    @param hodnota hledana hodnota
    @return <ul>
              <li>pole indexu nalezeneho prvku v pripade uspechu</li>
              <li>null v pripade neuspechu</li>
            </ul>
   */
  public static int[] nalezniPrvek(int[] pole, int hodnota) {
    int[] vysledky = new int[pole.length];
    int pocet = 0;
    for (int i = 0; i < pole.length;  i++) {
      if (pole[i] == hodnota) {
        vysledky[pocet] = i;
        pocet++;
      }
    }
    if (pocet == 0) {
      return null;      // neuspech
    }
    int[] pom = new int[pocet];
    for (int i = 0; i < pom.length;  i++) {
      pom[i] = vysledky[i];
    }
    return pom;         // uspech
  }
```

metoda `vypisVysledek()` se maličko změní

```
  public static void vypisVysledek(int[] pole, int hodnota,
                                   int[] vysledek) {
    System.out.println("Prvek s hodnotou " + hodnota
                       + " v poli:");
    System.out.println(Arrays.toString(pole));
    if (vysledek != null) {
      System.out.println("je na indexech: "
                         + Arrays.toString(vysledek));
    }
    else {
      System.out.println("neni");
    }
  }
```

použití

```
  public static void main(String[] args) {
    int[] vys;
    int prvek = 6;
    // zadny vyskyt
    int[] pole1 = {5, 1, 4, 2, 3};
    vys = nalezniPrvek(pole1, prvek);
    vypisVysledek(pole1, prvek, vys);

    // jeden vyskyt
    int[] pole2 = {5, 1, 6, 2, 3};
    vys = nalezniPrvek(pole2, prvek);
    vypisVysledek(pole2, prvek, vys);

    // vice vyskytu
    int[] pole3 = {6, 1, 6, 2, 6};
    vys = nalezniPrvek(pole3, prvek);
    vypisVysledek(pole3, prvek, vys);
  }
```

vypíše

```
Prvek s hodnotou 6 v poli:
[5, 1, 4, 2, 3]
neni
Prvek s hodnotou 6 v poli:
[5, 1, 6, 2, 3]
je na indexech: [2]
Prvek s hodnotou 6 v poli:
[6, 1, 6, 2, 6]
je na indexech: [0, 2, 4]
```


10.4. Uspořádané pole

Poznámka

Pro jednoduchost budeme uvažovat zcela zaplněné pole. Postupy pro nezaplněné pole viz dříve, prakticky by se ale použil jen způsob s koncovým indexem.

- pokud budeme v poli vyhledávat častěji a nezáleží nám na původním pořadí vkládaných hodnot, vyplatí se pole před vyhledáváním seřadit
  - prvky tvoří monotonní posloupnost

Poznámka

Předpokládáme opět řazení vzestupně.

- pak je nejužívanější technikou **metoda půlení intervalu**, známá též jako **binární vyhledávání**
  - v rámci každého kroku cyklu se rozpůlí prohledávaný interval
  - výhodou je, že prvek je nalezen (nebo nenalezen) do nejvýše $\log_2$(pocetPrvkuPole) kroků
    - dělíme-li pocetPrvkuPole opakovaně 2, pak po $\log_2$(pocetPrvkuPole) krocích dostaneme číslo menší nebo rovno 1
    - říkáme, že **složitost** (viz později) je logaritmická
  - algoritmus je
    - zjistíme hodnotu ležící na prostředním indexu
    - je-li rovna hledané hodnotě, cyklus končí

- je-li menší, hledáme v levé polovině
- je-li větší, hledáme v pravé polovině

Výstraha

Nevýhodou metody je, že při opakujících se hodnotách prvků končí při nalezení libovolného vyhovujícího prvku, tzn. ne nutně prvního.

```
final static int NEUSPECH = -1;

/**
  nalezne nejaky vyskyt prvku v poli
  @param pole prohledavane pole
  @param hodnota hledana hodnota
  @return <ul>
            <li> index nalezeneho prvku v pripade uspechu</li>
            <li> NEUSPECH (-1) v pripade neuspechu</li>
          </ul>
 */
public static int puleniIntervalu(int[] pole, int hodnota) {
  int dolni = 0;
  int horni = pole.length - 1;
  while (dolni <= horni) {
    int stred = (dolni + horni) / 2;
    if (pole[stred] < hodnota) {
      dolni = stred + 1;
    }
    else if (pole[stred] > hodnota) {
      horni = stred - 1;
    }
    else {
      return stred;     // uspech
    }
  }
  return NEUSPECH;      // neuspech
}
```

použití je, např.

```
public static void main(String[] args) {
  // lichy pocet prvku
  int[] pole1 = {1, 2, 3, 4, 5};
  int vys;
  int prvek = 2;
  vys = puleniIntervalu(pole1, prvek);
  vypisVysledek(pole1, prvek, vys);
  prvek = 6;
  vys = puleniIntervalu(pole1, prvek);
  vypisVysledek(pole1, prvek, vys);

  // sudy pocet prvku
  int[] pole2 = {1, 2, 3, 4, 5, 7};
  prvek = 2;
  vys = puleniIntervalu(pole2, prvek);
  vypisVysledek(pole2, prvek, vys);
  prvek = 6;
  vys = puleniIntervalu(pole1, prvek);
  vypisVysledek(pole2, prvek, vys);

  // opakujici se prvky
  int[] pole3 = {1, 2, 2, 2, 5, 7};
  prvek = 2;
  vys = puleniIntervalu(pole3, prvek);
  vypisVysledek(pole3, prvek, vys);
}
```

vypíše

```
Prvek s hodnotou 2 v poli:
[1, 2, 3, 4, 5]
je na indexu: 1
Prvek s hodnotou 6 v poli:
[1, 2, 3, 4, 5]
neni
Prvek s hodnotou 2 v poli:
[1, 2, 3, 4, 5, 7]
je na indexu: 1
Prvek s hodnotou 6 v poli:
[1, 2, 3, 4, 5, 7]
neni
Prvek s hodnotou 2 v poli:
[1, 2, 2, 2, 5, 7]
je na indexu: 2
```

10.4.1. Praktické použití

- metodu pro binární vyhledávání nebudeme programovat sami, protože je součástí knihoven Javy
  - je ze třídy `java.util.Arrays` a jmenuje se `binarySearch()`
    - prvním příkazem musí být `import java.util.*;`
  - je přetížena pro pole všech primitivních datových prvků
  - pro typ `int` má hlavičku `int binarySearch(int[] pole, int hodnota)`
  - návratovým typem je vždy typ `int` s významem
    - index nalezeného prvku v případě úspěchu
    - záporné číslo v případě neúspěchu
      - to v absolutní hodnotě +1 dává index, kam by byl hledaný prvek zařazen, kdyby byl v hledaném poli
      - prakticky se použije málokdy a proto testujeme jen na `< 0`

- příklad použití

```
import java.util.*;

public class VyhledavaniBinarniKnihovna {
  public static void vypisVysledek(int[] pole, int hodnota,
                                   int vysledek) {
    System.out.println("Prvek s hodnotou " + hodnota
                       + " v poli:");
    System.out.println(Arrays.toString(pole));
    if (vysledek >= 0) {
      System.out.println("je na indexu: " + vysledek);
    }
    else {
      System.out.println("neni");
    }
  }

  public static void main(String[] args) {
    // lichy pocet prvku
    int[] pole1 = {1, 2, 3, 4, 5};
    int vys;
    int prvek = 2;
    vys = Arrays.binarySearch(pole1, prvek);
    vypisVysledek(pole1, prvek, vys);
    prvek = 6;
    vys = Arrays.binarySearch(pole1, prvek);
    vypisVysledek(pole1, prvek, vys);

    // sudy pocet prvku
    int[] pole2 = {1, 2, 3, 4, 5, 7};
    prvek = 2;
    vys = Arrays.binarySearch(pole2, prvek);
    vypisVysledek(pole2, prvek, vys);
    prvek = 6;
    vys = Arrays.binarySearch(pole1, prvek);
    vypisVysledek(pole2, prvek, vys);

    // opakujici se prvky
    int[] pole3 = {1, 2, 2, 2, 5, 7};
    prvek = 2;
    vys = Arrays.binarySearch(pole3, prvek);
    vypisVysledek(pole3, prvek, vys);
  }
}
```

Vypíše:

```
Prvek s hodnotou 2 v poli:
[1, 2, 3, 4, 5]
je na indexu: 1
Prvek s hodnotou 6 v poli:
[1, 2, 3, 4, 5]
```



```
neni
Prvek s hodnotou 2 v poli:
[1, 2, 3, 4, 5, 7]
je na indexu: 1
Prvek s hodnotou 6 v poli:
[1, 2, 3, 4, 5, 7]
neni
Prvek s hodnotou 2 v poli:
[1, 2, 2, 2, 5, 7]
je na indexu: 2
```

Kapitola 11. Složitost, kódování dat

Výstraha

Následující výklad je pouze pragmatický úvod do problematiky bez podrobnějšího rozboru. Má za účel pouze prvotní pochopení, proč jsou dříve uvedené algoritmy řazení tak málo efektivní v porovnání s knihovní metodou `Arrays.sort()`. Podrobný výklad viz v KIV/PPA2.

11.1. Úvodní informace

- při praktické realizaci algoritmu jsme omezení prostředky, které máme k dispozici, zejména
  - časem
  - velikostí paměti
- složitost je vztah daného algoritmu k daným prostředkům
  - časová
    - každé množině vstupních dat přiřazuje počet operací vykonaných při výpočtu podle stanoveného algoritmu
      - čas se tedy měří počtem nutně provedených operací, přičemž doba provedení operace nezávisí na rozsahu vstupních dat
  - paměťová
    - závislost paměťových nároků na vstupních datech
- časová složitost bývá často podceňována, protože:
  - v začátcích programování se nesetkáváme s úlohami s vysokou časovou složitostí a současně s rozsáhlými daty
  - „tento počítač je pomalý, použitím rychlejšího počítače se čas výrazně sníží“
  - bývá snaha zrychlit dosavadní algoritmus dílčími úpravami
    - programátorským trikem, např. použití zarážky – viz dříve (urychlení o 7 %)
    - přepsání některých částí do jazyka asembleru apod.
  - řešením ale je použít rychlejší algoritmus – existuje-li a známe-li jej

11.2. Přesné zjištění složitosti

- časovou složitost je možné stanovit v závislosti na
  - konkrétních datech
  - na základě rozsahu dat – mnohem častější
    - obecně se získává analýzou algoritmu – často velmi složité

- u triviálních algoritmů je možné ji stanovit detailní analýzou programu

Příklad 11.1. Ukázka zjištění časové složitosti pro součet prvků pole

```
static int soucetPrvku(int[] pole) {
  int suma = 0;          // prirazeni p1
  for (int i = 0;        // prirazeni p2
       i < pole.length; // porovnani c1
       i++) {           // soucet a prirazeni s1 + p3
    suma += pole[i];    // soucet a prirazeni s2 + p4
  }
  return suma;
}
```

– budeme-li považovat komentované instrukce za operace, je časová složitost pro *n* prvků pole

C(n) = p1 + p2 + (n + 1) * c1 + n * (s1 + p3) + n * (s2 + p4)

– pokud budeme pro zjednodušení považovat akce (*a*) porovnávání, sčítání a přiřazení za stejně složité (a = p = c = s), pak

C(n) = a + a + (n + 1) * a + n * (a + a) + n * (a + a) = 3 * a + 5 * n * a = a * (3 + 5 * n)

– bude-li akce *a* trvat jednotku času, je složitost C(n) = 3 + 5n

Příklad 11.2. Ukázka zjištění časové složitosti pro součet prvků čtvercové matice

```
static int soucetPrvku(int[][] matice) {
  int suma = 0;              // prirazeni p1
  for (int i = 0;            // prirazeni p2
       i < matice.length;    // porovnani c1
       i++) {                // soucet a prirazeni s1 + p3
    for (int j = 0;          // prirazeni p4
         j < matice.length; // porovnani c2
         j++) {              // soucet a prirazeni s2 + p5
      suma += matice[i][j]; // soucet a prirazeni s3 + p6
    }
  }
  return suma;
}
```

– časová složitost pro čtvercovou matici řádu *n*

C(n) = p1 + p2 + (n + 1) * c1 + n * (s1 + p3) + n * (p4 + (n + 1) * c2 + n * (s2 + p5) + n * (s3 + p6))

– po převedení na akce:

C(n) = a + a + (n + 1) * a + n * (a + a) + n * (a + a) + n * (a + a + (n + 1) * a + n * (a + a) + n * (a + a)) =

$= 3a + 5an + n(3a + 5an) = 3a + 5an + 3an + 5an^2 = 3a + 8an + 5an^2$

– bude-li akce *a* trvat jednotku času, je $C(n) = 3 + 8n + 5n^2$

Příklad 11.3. Vypočtené hodnoty

```
   n  pole   matice [msec]
 10:    53      583
 20:   103     2163
 40:   203     8323
 80:   403    32643
```

Příklad 11.4. Naměřené hodnoty (pro 100 000 opakování)

```
   n  pole   matice [msec]
 10:    40      541
 20:    71     2053
 40:   140     8011
 80:   261    32967
```

11.3. Odhad složitosti

- zpravidla nás nezajímají konkrétní počty operací pro různé rozsahy vstupních dat *n*, ale **tendence jejich růstu** při zvětšujícím se *n*
  - pak lze výrazy udávající složitost výrazně zjednodušit
    - ♦ zanedbáme aditivní konstanty (u předchozího příkladu pole 3, u matice 3)
    - ♦ zanedbáme multiplikativní konstanty (u pole 5, u matice 8 a 5)
    - ♦ zanedbáme všechny složky s nižším řádem růstu než nejvyšším (u pole takové nejsou, u matice n)
  - u pole je C(n) = n – **složitost je lineární**
  - u matice je C(n) = n^2 – **složitost je kvadratická**

Příklad 11.5. Vypočtené hodnoty po zjednodušení

```
   n  pole   matice [msec]
 10:    10      100
 20:    20      400
 40:    40     1600
 80:    80     6400
```

11.3.1. Obecně používané vyjádření složitosti

- časovou složitost vyjadřujeme pomocí asymptotické notace
  - o dvou funkcích *f* a *g* definovaných na množině přirozených čísel a s nezáporným oborem hodnot říkáme

    „*f* roste řádově nejvýše tak rychle, jako *g*“
  - píšeme

$f(n) = O(g(n))$

pokud existuje přirozené číslo *K* tak, že platí:

$f(n) \leq K.g(n)$

- pro předchozí příklad s maticí je např. K = 6

- v tomto způsobu zápisu, který bude dále používán, je tedy časová složitost
  - součtu pole: O(n)
  - součtu matice: $O(n^2)$

11.3.2. Hrubý odhad složitosti pro triviální algoritmy

- lze provést jednoduše – každý vnořený cyklus zvyšuje mocninu složitosti o jednu
  - součet pole – jeden cyklus – složitost O(n)
  - součet matice – dva cykly – složitost $O(n^2)$
- pozor na to, že složitost je odvozena od algoritmu, nikoliv od charakteru dat
  - všechny tři dříve uvedené řadicí algoritmy zpracovávaly pole, ale měly dva cykly
    - ♦ u všech je složitost $O(n^2)$
    - ♦ zanedbáváme skutečnost, že vnitřní cyklus byl vždy postupně zkracován
- u vyhledávání půlením intervalu jsme se setkali se složitostí O(log n) – **složitost je logaritmická**

  Poznámka

  Dále bude používán zápis *log n* ve smyslu $log_2 n$
- s další složitostí jsme se setkali u metody `Arrays.sort()`
  - používá algoritmus typu QuickSort nebo MergeSort se složitostí O(n log n)
  - podrobně v KIV/PPA2

11.4. Praktický význam složitosti

- tabulka ukazuje doby výpočtu pro různé složitosti a počty prvků za předpokladu, že jedna operace trvá jednu nanosekundu (10^{-9} sec), což je při rychlosti dnešních procesorů (3 GHz) splnitelné

složitost / n	10	20	40	100	1000	10 000	100 000
log n	3,3 ns	4,3 ns	5,3 ns	6,6 ns	10 ns	13,3 ns	16,6 ns
n	10 ns	20 ns	40 ns	100 ns	1000 ns	10 µs	100 µs
n log n	33 ns	86 ns	200 ns	660 ns	10 µs	133 µs	1660 µs

složitost / n	10	20	40	100	1000	10 000	100 000
n^2	100 ns	400 ns	1600 ns	10 µs	1 ms	100 ms	10 s

- z tabulky vyplývá několik skutečností:

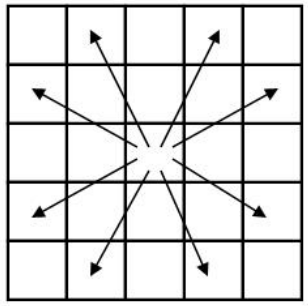
1. pokud bychom použili desetinásobně rychlejší počítač (nebo desetinásobně prodloužili čas výpočtu na stávajícím počítači) bylo by možné zpracovávat x-krát delší data
   - logaritmická 33 krát
   - lineární 10 krát
   - kvadratická 3 krát
2. v současnosti je horní mez rozsahu dat (tj. výpočet je proveden v „rozumném“ čase)
   - pro kvadratickou složitost (dříve uvedené metody řazení) **100 tisíc** prvků
   - pro O(n log n) **100 milionů** prvků, což přesahuje současné možnosti RAM pamětí

11.5. Ukázka algoritmu s časově neřešitelnou složitostí

Varování

Tyto úlohy nutně nemusejí mít velký počet (např. stamiliony prvků) zpracovávaných dat!

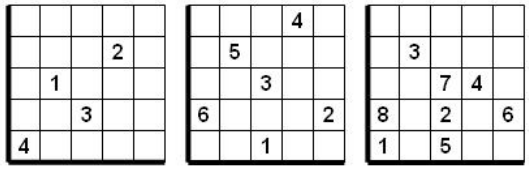
- jedná se o známou úlohu, kdy má jezdec na šachovnici postupně navštívit všechna políčka šachovnice, ovšem každé pouze jednou
  - nehledáme pouze jedno vyhovující řešení, ale všechna existující řešení pro jedno počáteční pole šachovnice
- úlohu budeme řešit tzv. **hrubou silou** (*brutal force*), kdy prověříme postupně všechny existující možnosti bez pokusů o zjednodušení (např. zrcadlové možnosti)
  - k tomu využijeme postup zvaný **zpětné trasování** (*backtracing*) – podrobnosti viz KIV/PPA2 nebo KIV/PT
- tato úloha má **exponenciální složitost** $O(8^{n * n})$
  - z jednoho políčka šachovnice lze totiž v ideálních podmínkách skočit až na 8 jiných polí



  - to by pro klasickou šachovnici teoreticky znamenalo 8^{64} možností, tj. přibližně:

    6 277 101 735 386 680 763 835 789 423 207 700 000 000 000 000 000 000 000 000

    $6 . 10^{57}$ možností
  - v případě, že by ověření jedné možnosti trvalo 1 nanosekundu, celý výpočet by trval asi 2.10^{41} let
- ve skutečnosti tolik možností není, protože:
  - tahů je ve skutečnosti jen 63
  - množství pokusů skončí neúspěchem mnohem dříve, než po 63 tazích
    - lze skončit už po 4, 6 či 8 tazích



  - políčka u krajů mají mnohem méně možností pokračujících skoků – 2, 3, 4 a 6
  - na osm políček lze skočit pouze z prvního tahu
    - toto lze zobecnit, protože na políčko se vždy (s výjimkou prvního tahu) musíme dostat skokem z nějaké přípustné pozice, takže možností výskoků z políčka je vždy (n – 1)
  - poslední tah má pouze jednu možnost, ať je kamkoliv, předposlední max. dvě možnosti apod.
    - navíc u mnoha případů budou tzv. vynucené tahy, kdy bude zbývat pouze jedna možnost pro celý řetězec tahů
  - zpřesněným horním odhadem tedy dostaneme „mnohem příznivější“ počet možností

    $1^4 * 2^8 * 3^{20} * 5^{16} * 7^{15} = 1 * 256 * 3486784401 * 152587890625 * 4747561509943 =$

    646 629 820 924 111 671 841 523 437 500 000 000

$6 . 10^{35}$ možností, což je „jen“ 20.10^{18} let

- je zřejmé, že pro šachovnici 8 x 8 polí není tato úloha zcela řešitelná
  - není dosud řešitelná ani pro 7 x 7 polí
  - pro takto velké šachovnice je řešitelná pouze částečně, např. prozkoumáme jen první jednu miliardu možností
    - i tento rozsahem velmi malý pokus poskytne překvapivě značné množství řešení
      - 7 x 7: 11 160 řešení
      - 8 x 8: 6 243 řešení
- pro 6 x 6 polí a méně je úloha na současných počítačích řešitelná kompletně
  - zpřesněný horní odhad možností je:

    $1^4 * 2^8 * 3^{12} * 5^8 * 7^3 = 1 * 256 * 531441 * 390625 * 343 =$

    18 228 426 300 000 000 možností
  - skutečný počet možností je 188 888 353 094, což představovalo v Javě 2958 sekund výpočtu na Pentiu Centrino 1,5 GHz
    - skutečné možnosti představují asi 10^{-5} z horního odhadu
    - výpočet jedné možnosti trval asi 16 nanosekund
  - celkový počet řešení, tj. kompletních 36 tahů, je 524 486
- použijeme-li znalost, že pouze 10^{-5} ze zpřesněného horního odhadu je skutečný počet možností, bude pro 8 x 8 polí kvalifikovaný odhad

  $6 . 10^{35} * 10^{-5} = 6 . 10^{30}$ možností, tj. asi 2.10^{14} let výpočtu

11.6. Kódování aneb data v počítači

Poznámka

Toto je podrobné rozvedení problematiky, která byla informativně zmíněna na druhé přednášce.

- data převádíme na posloupnost bitů, respektive bajtů, tzn. je pevně daný rozsah (počet řádů)
  - nevýznamové nuly jsou důležité
- převod se nazývá kódování

11.6.1. Základní dělení kódů (datových typů)

- číselné
  - celočíselné
    - beznaménkové
    - znaménkové
      - přímý kód
      - inverzní kód
      - doplňkový kód
      - kód s posunutou nulou
  - reálné
    - v jednoduché přesnosti
    - ve dvojnásobné přesnosti
- znakové
- logické (booleovské)

11.6.2. Podrobnější pohled na datové typy

Poznámka

Příklady u celých čísel jsou udávány na osmi bitech. Pro větší rozsahy platí tytéž principy.

11.6.2.1. Celočíselné beznaménkové

- zápis ve dvojkové soustavě zarovnaný na násobky bajtů (někdy nazývaný „přímý kód“)
- pouze pro nezáporná čísla (*unsigned*)
- např.: 1000 0011 = 131 nebo 0000 0011 = 3

11.6.2.2. Celočíselné znaménkové

- **přímý kód**
  - kladná čísla stejná, jako u beznaménkového
  - znaménko je učeno v MSB (*Most Significant Bit*)
  - např.: 1000 0011 = –3
  - nevýhody: dvojí nula 0000 0000 (kladná) a 1000 0000 (záporná)
  - používá se pro zobrazení mantisy u reálných čísel
- **inverzní kód** (jedničkový doplněk)
  - kladná čísla stejná, jako u beznaménkového
  - pro záporná čísla se mění 0 na 1 a 1 na 0

- např.: 1111 1100 = –3
- nevýhody: dvojí nula 0000 0000 (kladná) a 1111 1111 (záporná)
- používá se v bitových operacích případně jako mezikrok v doplňkovém kódu

■ **doplňkový kód** (dvojkový doplněk)

- matematická definice
  - ♦ D(x) = x pro x>= 0
  - ♦ D(x) = x + K pro x < 0, kde K = 2^n (kapacita soustavy)
- kladná čísla stejná, jako u bezznaménkového
- záporná: „inverzní + 1“
- např.: 1111 1101 = –3
- -1 ~ 1111 1111
- výhradně používané zobrazení znaménkových čísel
- výhody:
  - ♦ pouze jedna nula: 0000 0000
  - ♦ odečítání je přičítání záporného čísla (zjednodušení ALJ)

```
   3    0000 0011
+(-1)   1111 1111
-----   ---------
   2    0000 0010
```

- pozor na nesymetrický rozsah: –128 až +127
- častá chyba: 127 + 1 = –128
  - ♦ tento jev se nazývá **přetečení** a jsou principiálně možné dvě reakce
    - – přerušení výpočtu jako reakce na chybu
    - – pokračování výpočtu s nesprávným výsledkem – používá Java

■ **kód s posunutou nulou**

- lineární posun nuly po číselné ose
  - ♦ 0000 0000 je v absolutní hodnotě největší záporné číslo
  - ♦ nula je ve středu rozsahu 2^{n-1}-1 např. 0 ~ 0111 1111
  - ♦ 1111 1111 je největší kladné číslo
- kladná čísla **odlišná** od bezznaménkového – základní nevýhoda

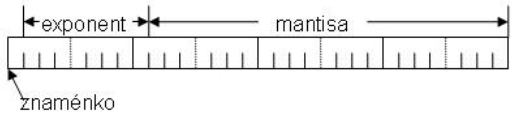
- např.: 1000 0010 = +3
- použití: exponent u reálných čísel

11.6.2.3. Reálné

■ aproximace reálných čísel

■ též „zobrazení v **pohyblivé řádové čárce**“ (*floating point*)

■ dle IEEE 754 na 4 bajtech (nebo na 8 bajtech – viz dále)



■ znaménkový bit patří mantise: 0 ~ +, 1 ~ –

■ mantisa je v přímém kódu, základ 2

- je normalizovaná: 1 <= mantisa <= základ
- její první bit musí být 1 –> neukládá se a představuje jedničku před desetinnou čárkou

■ exponent je na 8bitech v kódu s posunutou nulou, kde 0 ~ 0111 1111

■ mantisa určuje **přesnost**, exponent určuje **rozsah** čísla

- formát je kompromis mezi přesností a rozsahem

■ rozsah je v absolutní hodnotě asi od 10^{-45} do 10^{+38}, přesnost asi na 6 až 7 desetinných (desítkových) číslic

■ hodnota 0 reálného čísla je uložena jako samé nuly

■ hodnota čísla je

$x = (-1)^{znam} * 2^{exp-127} * 1,mantisa$

■ hodnoty některých čísel

```
 0,0   00 00 00 00
 0,5   3F 00 00 00
 1,0   3F 80 00 00
-1,0   BF 80 00 00
 2,0   40 00 00 00
+min   00 00 00 01  ~ 1,4E-45
+max   7F 7F FF FF  ~ 3,4E+38
+nek   7F 80 00 00
-nek   FF 80 00 00
 NaN   7F C0 00 00
```

■ Pozor: některá reálná čísla nelze uložit přesně (mantisa má omezenou velikost), např.

0,1 ~ 3D CC CC CD

- to může způsobit chyby při aritmetických operacích

■ je-li hodnota čísla < min, číslo nelze zobrazit, zaokrouhluje se automaticky na 0 – **podtečení**

např.: 1,2E-50 se zaokrouhlí na 0

■ je-li hodnota čísla > max, číslo nelze zobrazit, zaokrouhluje se automaticky na nekonečno – **přetečení**

např.: 8,2E+50 se zaokrouhlí na +nekonečno

■ Pozor: přetečení i podtečení může být pro kladné i záporné hodnoty

např.: –1,2E-50 se zaokrouhlí na 0 a –8,2E+50 se zaokrouhlí na –nekonečno

■ **NaN** (*Not a Number*) – definovaná hodnota při chybné operaci s reálným číslem (např. dělení nuly nulou)

■ zobrazení na 8 bajtech dodržuje naprosto stejnou filosofii, pouze exponent má 11 bitů a mantisa 52 bitů

- rozsah je v absolutní hodnotě asi od 10^{-308} do 10^{+308}
- přesnost asi na 15 až 16 desetinných (desítkových) číslic

■ operace s reálnými čísly

- **sčítání** – porovnáním exponentů se zjistí menší číslo, tomu se zvětší exponent na úroveň většího čísla (tj. zmenšuje se mantisa, tj. ztrácí se přesnost) – pro hodně odlišná čísla se může v důsledku přičítat nula
- **násobení** – exponenty se sečtou, mantisy se vynásobí
- Pozor – nepoužívat reálná čísla tam, kde stačí celá čísla – menší rychlost (cca o 25 % – záleží na mnoha okolnostech)
  - ♦ zejména v cyklech nahradit reálná čísla celými – zaokrouhlovací problémy
  - ♦ zásadně nepoužívat porovnání na rovnost (==)

11.6.2.4. Znakové

■ přiřazují každému znaku zvolené abecedy nezáporné celé číslo

■ existuje (a používá se) mnoho kódů (podrobně později)

- český text běžně v 11 různých kódováních
  - ♦ proto bude této problematice věnována samostatná přednáška

11.6.2.5. Řetězec

■ řetězec je složen z jednotlivých znaků (v daném kódování)

■ u řetězce se rozlišuje

- kapacita – počet znaků, které může řetězec nejvýše uschovat

- aktuální velikost – počet platných znaků
- např. řetězec může mít kapacitu 10 znaků, ale aktuální velikost je pouze 4 znaky, protože je v něm např. slovo `ahoj`

■ způsob uložení v paměti a zejména označení aktuální velikosti závisí nejčastěji na použitém programovacím jazyce, např.:

- Java – rozeznává dva základní typy řetězců, způsoby jejich implementace nejsou obecně známy
  - ♦ řetězec typu `String` má aktuální velikost vždy rovnou kapacitě
  - ♦ řetězec typu `StringBuffer` je měnitelný řetězec s různou aktuální velikostí a kapacitou
- jazyk C – řetězec je pole bajtů začínající indexem 0, za posledním platným znakem řetězce je přidán znak s hodnotou 0 binárně (`'\0'`)
  - ♦ aktuální velikost se zjistí spočtením všech znaků předcházejících znaku `'\0'`
  - ♦ výhodou je, že řetězec může mít libovolnou kapacitu
- Pascal – řetězec je pole bajtů, kdy jsou využívány bajty od indexu 1
  - ♦ na indexu 0 je uložena hrodnota představující aktuální velikost
  - ♦ nevýhodou je, že kapacita je max. 255 znaků

11.6.2.6. Logické

■ též **booleovské** podle pana Booleho (Booleova algebra)

■ ukládají jen dvě hodnoty – *true* (pravda, **logická 1**) a *false* (nepravda, **logická nula**)

■ prakticky se realizují pomocí celočíselného bezznaménkového typu

11.6.2.7. Problém ukládání dat do fyzické paměti

Poznámka

Dále popisovaný problém se týká všech datových typů. **V praxi se s ním setkáme až při práci s binárními soubory**, ale na tyto termíny narazíme mnohem dříve.

■ většinou má datový typ větší délku než 1 bajt, pak záleží na tom, v jakém pořadí jsou jednotlivé bajty ukládány do paměti (vnitřní i vnější)

např. celé neznaménkové číslo na čtyřech bajtech o hodnotě $01A2B3C4_H$ (tj. 27440068_D) budeme do paměti ukládat od adresy 1000

■ existují dvě možnosti

1. **big-endian** (BE) – „vyšší řády na nižší adrese“ – „přirozené“ uložení

   např.: 1000:01 1001:A2 1002:B3 1003:C4

   tedy: 01 A2 B3 C4

2. **little-endian** (LE) – „vyšší řády na vyšší adrese“ – „obrácené“ uložení

   např.: 1000:C4 1001:B3 1002:A2 1003:01

   tedy: C4 B3 A2 01

- obecně se nedá říci, s jakým způsobem uložení dat se setkáme a je běžné, že nalezneme oba způsoby na tomtéž počítači
- způsob záleží na
  - procesoru
    - Intel LE
    - Motorola BE
  - operačním systému
    - Windows LE
  - programovacím jazyce
    - Java BE – zaručuje 100% přenositelnost i binárních souborů
    - v mnoha programovacích jazycích není určen a přebírá se způsob od operačního systému

Kapitola 12. Kódování znaků (nejen češtiny)

12.1. Základní terminologie

- velmi nejednotná
  - liší se u velkých firem a organizací
    - mnohá větší organizace si „klade za čest“ přijít s novou terminologií místo toho, aby „se snížila“ k použití terminologie od konkurence
  - stejné pojmy se používají pro různé věci
- dále je použita terminologie z *Unicode – Character Encoding Model*

  `http://www.unicode.org/reports/tr17/tr17-5.html`
- pro kódování lze používat několikaúrovňovou organizaci

12.1.1. Znaková sada (množina kódovaných znaků – *Coded Character Set* – CCS)

- mapování mezi množinou (abstraktních) znaků a množinou nezáporných celých čísel (**kódové body** – *code points*)
  - A = 41 (abstraktní znak A má kódový bod 41)
  - US-ASCII, ISO 8859-1, JIS X 0201, Unicode nebo ISO 10646-1 jsou příklady **znakových sad**

12.1.2. *Character Encoding Form* – CEF

- mapování množiny nezáporných celých čísel (prvků CCS) na množinu kódových jednotek dané šířky, např. 32bitových `int`
  - A = 00000041 (znak A namapovaný na 32bitový `int`)

12.1.3. Kódovací schéma, kódování (*Character Encoding Scheme* – CES)

- způsob mapování kódových jednotek z CEF do posloupnosti **oktetů** (osmibitových bajtů)
- další používané názvy jsou *character map*, *charmap*, *character set*, *charset*, *code page*
  - A = 00 41 (pro znakovou sadu Unicode a kódovací schéma UTF-16BE)
- dále bude výhradně používán pojem „**charset**“, protože existuje stejně pojmenovaná příslušná třída `java.nio.charset.Charset`
- pro jednu znakovou sadu může být více charsetů
  - např. pro znakovou sadu Unicode existují charsety

- ♦ UTF-8
- ♦ UTF-16
- ♦ UTF-32

- často je ale charset určen jen pro jednu znakovou sadu, např.: ISO 8859-2
- naopak pro několik různých asijských znakových sad (**ideografická písma**) existuje jedno kódovací schéma EUC
  - charsety jsou např. EUC-JP, EUC-KR, x-EUC-TW
- jeden dokument může být napsán současně
  - ve více znakových sadách (Unicode, ISO 8859-2)
  - za použití různých charsetů (UTF-8, UCS-2, ISO-8859-2)
  - což je nejhorší možný případ, pokud je tento dokument uložen v jednom souboru
- běžně je jeden soubor vždy jen v jednom charsetu
  - ovšem je běžné, že soubory z jednoho projektu mají různé charsety – není to šťastné řešení

Výstraha

Charsety nikdy nerozlišují tvar znaku (*glyph* – nesprávně „font“)

Characters Versus Glyphs

Glyphs	Unicode Characters
A A A A A A A	U+0041 LATIN CAPITAL LETTER A
a a a a a a a	U+0061 LATIN SMALL LETTER A
fi fi	U+0066 LATIN SMALL LETTER F + U+0069 LATIN SMALL LETTER I
п *n* *ū*	U+043F CYRILLIC SMALL LETTER PE
ه ه ه ه	U+0647 ARABIC LETTER HEH

12.2. Historie

- sedmibitový ASCII kód (US-ASCII)
  - MSB (*Most Significant Bit*) je vždy nulový
  - základ všech dalších kódování (*zero extending* – viz dále)
  - popisuje „všechny znaky z anglické klávesnice“, tj. velká a malá neakcentovaná písmena latinky, číslice, interpunkci a speciální znaky
  - bez 32 řídících znaků je tedy k dispozici 128 – 32 = 96 znaků
  - malých a velkých písmen je 52, číslic 10, zbývajících znaků 34
  - stačí pouze pro čistě anglické texty

- pro mnoho jazyků nestačí jen neakcentovaná písmena a pro akcentovaná písmena už není v US-ASCII místo
  - proto vznikaly osmibitové znakové sady
    - ♦ využívá se osmý bit (MSB), což dává možnost pro dalších až 128 znaků
    - ♦ čeština např. využívá áčďéěíňóřšťúůýž (tj. 30 dalších znaků – včetně velkých písmen)
  - problémy pojmenování viz dále
- protože jsou osmibitové, není třeba používat speciální kódovací schéma
  - celé číslo znaku ze znakové sady vždy představuje 1 bajt
  - znaková sada má stejný název jako kódovací schéma a ten je stejný jako název charsetu
- existuje mnoho různých charsetů, protože 128 volných znaků nestačí pro všechny požadavky
  - k dohodě nedošlo bohužel ani mezi evropskými státy používajícími latinku
  - to znamená, že např. česko-francouzská organizace musí používat pro své dokumenty dva charsety
- dle ISO normy vznikají charsety ISO-8859-1 až ISO-8859-15
- další charsety vznikají v národních standardizačních organizacích
- kromě nich existují znakové sady závislé na platformě (proprietární formáty)
  - např. CP-1250, MacCentralEurope, které jsou bohužel většinou jako jediné na konkrétní platformě plně podporovány
- celkově se odhaduje, že existují stovky těchto charsetů
- pokud kódují znaky latinky, je v naprosté většině prvních 128 znaků totožných s US-ASCII (*zero-extending*)
  - existují výjimky, např. EBCDIC (*Extended Binary-Coded Decimal Interchange Code*), který popisuje jen neakcentované znaky, ale ty jsou uloženy i v horních 128 pozicích
- některé aplikace označují osmibitové charsety jako ANSI
- společné označení pro sedmi a osmibitové charsety je SBCS (*Single-Byte Character Set*)

12.3. Současnost

- s rozvojem Internetu je zvýšená potřeba výměny dokumentů mezi různými platformami a různými národními jazyky
- řešením je 16bitový CEF
- do její přípravy se bohužel najednou pouštějí dvě různé organizace

  a. ISO (*International Organization for Standardization*)
     - od 1989

- znaková sada ISO/IEC 10646
- ve zkratce UCS (*Universal Character Set*)

b. Unicode (*Unicode Consortium*)

- od 1990
- znaková sada Unicode
- ve zkratce Unicode

- naštěstí se obě organizace domlouvají hned na počátku prací a od roku 1991 pracují na sjednocení obou znakových sad, což bylo dokončeno v 1993
  - v současnosti obě znakové sady nejsou absolutně totožné, liší se např. v drobných specifikacích jednotlivých znaků, částečně odlišné terminologii apod.
  - podstatné je, že číselné hodnoty (*code points*) jednotlivých znaků jsou identické
- z běžného pohledu nemá význam obě sady rozlišovat a běžně se považují za synonyma a používá se společné označení Unicode
  - velmi dobře zdokumentovaná znaková sada viz `www.unicode.org`
- *code points* jednotlivých znaků se označují jako `U+hexa_číslo`, např. znak A je `U+0041`
  - prvních 128 znaků je stejných jako u US-ASCII (*zero-extending*)

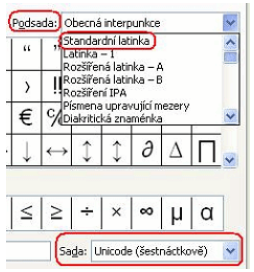
bitů	charset	binárně	hexa	znak
7	US-ASCII	`1000001`	41	A
8	ISO-8859-2	`01000001`	41	A
16	UTF-16, UCS-2	`00000000 01000001`	41	A
32	UTF-32, UCS-4	`00000000 00000000 00000000 01000001`	41	A

12.4. Historie Unicode

verze 1.	1991	28292 znaků
verze 2.	1996	38869 znaků – rozšíření na 32 bitů (resp. 21 bitů)
verze 3.	2000	49170 znaků
verze 4.	2003	96248 znaků
verze 5.	?	připravovaná verze

- 16bitová CEF umožňuje použít asi 65 tisíc znaků (U+0000 až U+FFFF)
  - ty jsou ve skupině označené jako BMP (*Basic Multilingual Plane*)
  - BMP obsahuje všechny znaky používané v Evropě a Americe plus základní ideografická písma čínštiny, japonštiny a korejštiny (HAN písmo)

- samotné BMP je vnitřně děleno do bloků, kde např. ASCII je v *Basic Latin block* (dělení je občas používáno ve fontech)



- od Unicode verze 2.0 přestal rozsah 16 bitů dostačovat a Unicode přešlo na 32 bitů
  - z nich ale využívá jen 21 bitů, tj. `U+000000` až `U+10FFFF`
  - to znamená celkem 16 skupin (**sféra** – *plane*), každá o velikosti asi 65 tisíc znaků, dohromady asi 1 milion znaků
  - první skupina v pořadí je BMP (`U+0000` až `U+FFFF`)
  - další jsou v rozsahu `U+10000` až `U+10FFFF` a běžně se téměř nikdy (v našich zeměpisných šířkách) nepoužívají
  - znaky v těchto skupinách se nazývají *supplementary characters*
  - jak Unicode, tak i ISO neplánují ani v budoucnu překročit počet 21 bitů

12.5. Problém pořadí bajtů

- ukládáme-li do paměti (do souboru) vícebajtové entity, je třeba rozlišovat pořadí bajtů
  - A = 0041
  - *little-endian* LE (vyšší řády na vyšší adrese) `41 00`
  - *big-endian* BE (vyšší řády na nižší adrese) `00 41`
  - toto platí i pro např. čtyřbajtové entity `00 00 00 41` či `41 00 00 00`
  - způsob ukládání *little-* nebo *big-endian* závisí na platformě (Windows LE), procesoru (Intel LE, Motorola BE), programovacím jazyce (Java vždy BE), aplikaci, ...
- proč to potřebujeme?
  - většina současných souborových systémů pracuje s bajty – pro správnou serializaci znaků do bajtového proudu

12.6. Problém kódovacích schémat

- znak přesahuje jeden bajt – je možné používat více kódovacích schémat

- pro detailní pochopení je vhodné rozlišovat Unicode a ISO 10646 (pro praktické použití ne)
- Unicode samo o sobě nepopisuje mechanismy pro identifikaci proudu bajtů

■ ISO 10646 má dva základní charsety

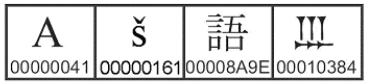
- UCS-2 (2 bajty) stačí právě pro BMP, ale pro nic víc!
- UCS-4 (4 bajty) pro celý rozsah

  UCS = *Universal Multiple-Octet Character Set*
- dolní polovina UCS-4 je identická s UCS-2
- ISO 10646 využívá i možností UTF-x

■ Unicode má sedm platných charsetů:

UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE

- UTF = *Unicode* (nebo UCS) *Transformation Format*
- UTF-x se používají i pro ISO 10646
- jakýkoliv UTF-x může sloužit pro uložení celého rozsahu Unicode (21 bitů)



UTF-32

UTF-16

UTF-8

■ obecný omyl je že *UCS-2 = UTF-16*

- rovnost platí pro všechny znaky z BMP (proto se zaměňují)
- UCS-2 nemůže popsat znaky z oblastí nad BMP (*supplementary characters*) a nemůže využívat **zástupné páry** (*surrogate pairs*) jako UTF-16 (viz dále)

■ UTF-xy (ne UTF-8) mohou přímo určit pořadí bajtů – LE nebo BE

UTF-32BE

UTF-32LE

UTF-16BE

UTF-16LE

UTF-8

12.7. Značka bajtového pořadí

■ UTF-x mohou pro identifikaci pořadí bajtů využívat počáteční **značku bajtového pořadí**

- *initial byte order mark* (BOM), která je umístěna na samém začátku souboru
- pro tento účel definuje Unicode dva kódové body
  - ♦ `U+FEFF` = ZERO WIDTH NO-BREAK SPACE (*byte order mark*) (pevná mezera nulové délky)
  - ♦ `U+FFFE` = not a character code

■ pro UTF-16 pro BE má tvar `FE FF` a pro LE pak `FF FE`

■ je-li načtena správně, pak se pevná mezera nulové délky ze své podstaty nemůže zobrazit

- při nesprávném načtení (záměna BE za LE nebo naopak) se opět nezobrazí, protože se jedná o neplatný znak
- k nesprávnému načtení by ale principiálně nemělo dojít

■ BOM se ale někdy nemusí nebo nesmí vyskytovat, což je dáno definicí charsetu

charset	BOM	hodnota	
UTF-8	volitelný	EF BB BF	
UTF-16	doporučený	FE FF nebo FF FE	
UTF-16LE	zakázaný	FF FE	na BOM se nebere zřetel, tj. i opačný FE FF musí být ignorován
UTF-16BE	zakázaný	FE FF	na BOM se nebere zřetel, tj. i opačný FF FE musí být ignorován
UTF-32	doporučený	00 00 FE FF nebo FF FE 00 00	
UTF-32LE	zakázaný	FF FE 00 00	na BOM se nebere zřetel
UTF-32BE	zakázaný	00 00 FE FF	na BOM se nebere zřetel

12.8. UTF-8

- bylo vytvořeno proto, aby se znaky Unicode daly zakódovat posloupností bajtů, protože s bajty umí pracovat každá aplikace a každý souborový systém
- vzniká v 1992
- staré a nepoužívané označení je UTF-2
- je to obecně rozšířený a přijímaný formát, popsaný např. v
  - RFC 2279
  - Amendment 2 v ISO 10646-1
  - Unicode Standard
  - např. řetězce v Javě v `.class` souborech jsou v UTF-8
- ze zmíněných sedmi charsetů Unicode se (v našich zeměpisných šířkách) často používá právě UTF-8
- výhody
  - pro texty využívající jen znaky anglické abecedy je UTF-8 totožná s US-ASCII
    - využívá se jen jeden bajt na jeden znak
    - s US-ASCII umí pracovat každá aplikace
  - pro akcentované znaky se využívají dva bajty
- soubory kódované v UTF-16 zabírají pro většinu textů psaných latinkou (nejen angličtinou) zbytečně dvojnásobné místo
  - zkoumáme-li četnost výskytu akcentovaných znaků v českém textu, zjistíme, že mají asi 10% výskyt

    „Například v tomto odstavci napsaném zcela prokazatelně češtinou s plným využitím akcentů je z celkových 129 písmen 114 písmen bez akcentů a jen 15 písmen s akcenty.“
  - započítáme-li do předchozího příkladu i mezery, interpunkci a číslice (všechny jsou z US-ASCII) dostaneme celkově 164 znaků a z toho 15 akcentovaných
- základní nevýhoda UTF-8 – znaky nemají stejnou délku, tzn. není možné skočit přímo na určitý znak („přeskoč prvních 20 znaků“)
  - pravděpodobnost „omylu“ (považování poloviny znaku za celý znak) je omezena principem kódovacího schématu

znak Unicode	max. význ. bitů	bajty UTF-8
U+0000 až U+007F	7	`0xxx xxxx`
U+0080 až U+07FF	11	`110x xxxx 10yy yyyy`
U+0800 až U+FFFF	16	`1110 xxxx 10yy yyyy 10zz zzzz`

znak Unicode	max. význ. bitů	bajty UTF-8
U+10000 až U+10FFFF	21	`1111 0xxx 10yy yyyy 10zz zzzz 10ss ssss`

- princip čtení
  - má-li bajt nastaveno MSB, pak počet jedničkových bitů za ním udává počet následujících bajtů znaku, které vždy začínají bity `10`
  - „trefíme-li“ se náhodně doprostřed znaku, poznáme to podle začátku `10` a pak je nutné přeskočit všechny následující bajty začínající `10`

Poznámka

principiálně je možné zakódovat pomocí UTF-8 až 31 bitů

`1111 110x 10yy yyyy 10zz zzzz 10ss ssss 10tt tttt 10uu uuuu`

- ale protože Unicode končí na významových 21 bitech, se tento způsob nevyužívá
- nejširší znak v UTF-8 má tedy 4 bajty

- u UTF-8 je z principu zbytečná značka bajtového pořadí (BOM)
  - specifikace říká, že není ani vyžadována, ani doporučována, ale pokud je použita, nesmí vadit
  - mnoho aplikací ale BOM vyžaduje a vytváří a bez BOM pracují chybně, např. nejsou schopné automaticky rozpoznat charset
  - některé (např. SciTe) dokonce považují BOM za nezbytnou část a označení „UTF-8“ znamená „UTF-8 + BOM“ a pro UTF-8 bez BOM používají označení „UTF-8 Cookie“
- praktický příklad

znak	Unicode	Unicode bitově	UTF-8 bitově	UTF-8 bajtově
D	U+0044	`0000 0000 0100 0100`	`0100 0100`	44
á	U+00E1	`0000 0000 1110 0001`	`1100 0011 1010 0001`	C3 A1
š	U+0161	`0000 0001 0110 0001`	`1100 0101 1010 0001`	C5 A1
a	U+0061	`0000 0000 0110 0001`	`0110 0001`	61
BOM	U+FEFF	`1111 1110 1111 1111`	`1110 1111 1011 1011 1011 1111`	EF BB BF
	U+FFFE	`1111 1111 1111 1110`	`1110 1111 1011 1111 1011 1110`	EF BF BE

- jediná správná značka bajtového pořadí je v UTF-8 `EF BB BF`
  - posloupnost `EF BF BE` vznikla zakódováním neexistujícího znaku, není tedy BOM a žádná aplikace ji nerozpozná

- slovo „Dáša“ může tedy v UTF-8 vypadat

  a. `44 C3 A1 C5 A1 61`

  b. `EF BB BF 44 C3 A1 C5 A1 61`

     ale nikdy jako

  c. `EF BF BE 44 C3 A1 C5 A1 61`

Poznámka

Existuje též kódovací schéma UTF-7, které využívá pouze 7 bitů bajtu

- je popsáno v RFC-2152
- prakticky by se použilo, pokud by (zastaralý) přenosový kanál dovoloval přenášet jen sedmibitové bajty
- slovo „Dáša“ v UTF-7 je: `44 2B 41 4F 45 42 59 51 2D 61`

  což znamená, že každý akcentovaný znak využívá čtyři bajty

12.9. Problém UTF-16

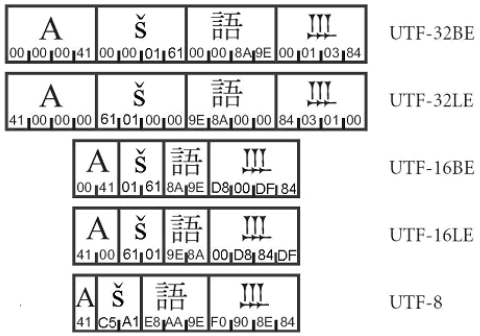
- dokud Unicode využívalo jen `U+0000` až `U+FFFF` kódovací schéma UTF-16 neexistovalo a používalo se UCS-2 s pevnou šířkou 2 bajty
- po expanzi na 21 bitů (od Unicode verze 2.0) UCS-2 s pevnou šířkou přestává stačit, ale není možné jej opustit, protože je široce používáno
  - proto je od 1994 navrženo a od 1996 nastupuje UTF-16 (přijímané též v ISO)
  - použil se stejný trik, jako ve vztahu US-ASCII a UTF-8
    - vše staré musí zůstat nezměněno, nové se přidá zvětšením šířky
- UTF-16 je kódovací schéma s proměnnou šířkou, čímž se odlišuje od UCS-2 (pro znaky mimo BMP)
  - kódování znaků v BMP se nemění!!!
- možného zvětšení šířky se dosáhlo trikem, že byly vyhrazeny dva bloky kódových bodů 1024 velké, které se mohou vyskytovat pouze společně, tj. jeden znak je zakódován pomocí 2 x 16 bitů
  - to dává možnost zakódovat 1024 x 1024 znaků nad BMP
  - oba „znaky“ dohromady se nazývají **zástupné páry** (*surrogate pairs*)
- první blok je v rozsahu `U+D800` až `U+DBFF`, tj.:
  - `1101 1000 0000 0000` až `1101 1011 1111 1111`
  - a označuje se jedním z termínů
    - *high-half zone*
    - *high surrogate area*
    - *leading-surrogate code unit*
- druhý blok je v rozsahu `U+DC00` až `U+DFFF`, tj.:
  - `1101 1100 0000 0000` až `1101 1111 1111 1111`
  - a označuje se jedním z termínů
    - *low-half zone*
    - *low surrogate area*
    - *trailing-surrogate code unit*
- v každém bloku je volných 10 bitů, dohromady 20 bitů
  - Unicode ale potřebuje až 21 bitů – použije se trik, kdy se využije toho, že BMP je kódováno jen 16 bity a proto lze 21 bitů „posunout dolů“ odečtením `0x10000` (viz dále)
  - zástupné páry tedy umí zakódovat znaky `U+010000` až `U+10FFFF`
- pro vyšší znaky nelze použít UTF-16 (ale není to třeba) a musí se použít UTF-32 nebo UCS-4
- způsob kódování znaků nad BMP, tj. `U+010000` až `U+10FFFF`
  - od hodnoty znaku odečteme `0x10000`, čímž se dostaneme do rozsahu `0x00000` až `0xFFFFF`
    - binárně `yyyy yyyy yyxx xxxx xxxx`
  - přidáme tyto bity k základům zástupných párů, tj. k `0xD800` a `0xDC00`
    - binárně `1101 10yy yyyy yyyy` a `1101 11xx xxxx xxxx`
- prakticky:
  - znak `U+10384`

    `0x10384 - 0x10000 = 0x00384 = 0000 0000 00|11 1000 0100`
    - vyšší pár: `1101 10|00 0000 0000 = 0xD800`
    - nižší pár: `1101 11|11 1000 0100 = 0xDF84`
- protože i UTF-16 potřebujeme serializovat (zapisovat po bajtech do souboru), používá se BOM, respektive UTF-16 existuje ve verzích UTF-16, UTF-16LE, UTF-16BE (viz dříve)

12.10. UTF-32

- byl definován v 1999
- pro prvních 21 bitů se neliší od UCS-4
- nad 21 bitů není definován (narozdíl od UCS-4), což je ale pouze akademický problém

- opět existují tři verze UTF-32, UTF-32LE, UTF-32BE



12.11. Problémy pojmenování charsetů

- každý významnější uživatel (nikoli tvůrce!) charsetu považuje za svoji povinnost jej nově pojmenovat, nejlépe úplně odlišně od již užívaných pojmenování
  - vzniká situace, kdy jeden a týž charset má množství různých jmen
  - např. US-ASCII má dalších 14 oficiálně evidovaných jmen:

    ISO646-US, IBM367, ASCII, cp367, default, ascii7, ANSI_X3.4-1986, iso-ir-6, us, 646, iso_646.irv:1983, csASCII, ANSI_X3.4-1968, ISO_646.irv:1991
- takže i když je uvedeno jméno charsetu, příjemce dokumentu nemusí být schopen text přečíst, protože toto pojmenování (nikoli charset!) nezná
- pořádek v pojmenování zavádí *Internet Assigned Numbers Authority* (IANA) `http://www.iana.org/assignments/character-sets`
- rozlišuje v pojmenování základní (nejoficiálnější) jméno, které se nazývá **kanonické jmén**o a ostatní evidovaná jména, kterým se říká **aliasy**
  - např. US-ASCII je kanonické jméno a pokud není speciální důvod, mělo by se používat
  - ANSI_X3.4-1986 je evidovaný alias od US-ASCII
- může se stát, že jméno charsetu není evidováno v IANA, ale charset je přesto podporován některými aplikacemi
  - pak se používá stejný princip kanonického jména a aliasů, ale kanonické jméno musí začínat znaky „x-“ nebo „X-“
  - např. API Javy poporuje charset s kanonickým jménem x-MacCentralEurope a aliasem MacCentralEurope
  - API JDK 1.5.0 podporuje 148 charsetů a 655 aliasů

12.12. Praktický dopad na uživatele počítačů v České republice

- zcela běžně se můžeme setkat s dokumentem kódovaným těmito charsety
  - uvažujeme pouze dokumenty psané latinkou

12.12.1. US-ASCII – American Standard Code for Information Interchange

aliasy: ISO646-US, IBM367, ASCII, cp367, default, ascii7, ANSI_X3.4-1986, iso-ir-6, us, 646, iso_646.irv:1983, csASCII, ANSI_X3.4-1968, ISO_646.irv:1991

- základní sedmibitový charset
- nemůže zobrazit akcenty
- je běžný v dokumentech psaných angličtinou nebo ve zdrojových kódech programů

12.12.2. ISO-8859-2 – Latin Alphabet No. 2

aliasy: ibm912, l2, ibm-912, cp912, ISO_8859-2:1987, ISO_8859-2, latin2, csISOLatin2, iso8859_2, 912, 8859_2, ISO8859-2, iso-ir-101

- základní osmibitový charset pro východeovropské země – mezinárodní standard dle ISO
- na platformě Unix/Linux se používalo zcela výhradně, ale poslední verze mnoha distribucí implicitně používají UTF-8

12.12.3. windows-1250 – Windows Eastern European

aliasy: cp1250, cp5346

- osmibitový proprietární charset fy Microsoft
- podporován operačními systémy a aplikacemi této firmy
- od ISO-8859-2 se liší pouze ve znacích š, Š, ž, Ž, ť, Ť

12.12.4. IBM852 – MS-DOS Latin-2

aliasy: 852, ibm-852, csPCp852, cp852, ibm852

- osmibitový proprietární charset fy IBM
- používaný charset v MS-DOSu a v konzolovém okénku Windows

12.12.5. x-MacCentralEurope – Macintosh Latin-2

alias: MacCentralEurope

- osmibitový proprietární charset fy Macintosh

12.12.6. UTF-8 – Eight-bit UCS Transformation Format

alias: UTF8, unicode-1-1-utf-8

- základní charset pro uživatele znakové sady Unicode
- znaky mají šířku 1 nebo 2 bajty
- na začátku může mít BOM `EF BB BF` , který ale nemá pro charset význam (může mít význam pro aplikaci, která podle něj snadno pozná UTF-8)

12.12.7. UTF-16 – Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

aliasy: utf16, UTF_16

- charset pro uživatele znakové sady Unicode
- v aplikacích se často označuje jako UCS-2, což není evidovaný alias
- znaky mají šířku 2 bajty
- na začátku může mít BOM `FE FF` nebo `FF FE`, který má význam pro určení pořadí bajtů
- neobsahuje-li BOM, musí se pořadí bajtů (*big-endian* nebo *little-endian*) vyzkoušet

12.12.8. UTF-16BE – Sixteen-bit Unicode Transformation Format, big-endian byte order

aliasy: X-UTF-16BE, UnicodeBigUnmarked, UTF_16BE, ISO-10646-UCS-2

- charset pro uživatele znakové sady Unicode
- znaky mají šířku 2 bajty a jsou vždy v *big-endian* pořadí
- na začátku nesmí mít BOM, je-li tam, je ignorován ve verzi `FEFF` i `FFFE`

12.12.9. UTF-16LE – Sixteen-bit Unicode Transformation Format, little-endian byte order

aliasy: UnicodeLittleUnmarked, X-UTF-16LE, UTF_16LE

- charset pro uživatele znakové sady Unicode
- znaky mají šířku 2 bajty a jsou vždy v *little-endian* pořadí
- na začátku nesmí mít BOM, je-li tam, je ignorován ve verzi `FEFF` i `FFFE`

Příklad 12.1. Jak vypadá bajtově dokument obsahující pouze slovo „Dáša“

1.	US-ASCII	44 nelze zobrazit akcenty 61
2.	ISO-8859-2	44 E1 B9 61
3.	windows-1250	44 E1 9A 61
4.	IBM852	44 A0 E7 61
5.	x-MacCentralEurope	44 87 E4 61
6.	UTF-8	44 C3 A1 C5 A1 61
7.	UTF-8 (s BOM)	EF BB BF 44 C3 A1 C5 A1 61
8.	UTF-16 (s BOM b-e)	FE FF 00 44 00 E1 01 61 00 61
9.	UTF-16 (s BOM l-e)	FF FE 44 00 E1 00 61 01 61 00
10.	UTF-16BE	00 44 00 E1 01 61 00 61
11.	UTF-16LE	44 00 E1 00 61 01 61 00

12.13. Zdroje

- `Java\jdk1.5.0\docs\guide\intl\encoding.doc.html`
- `Java\jdk1.5.0\docs\api\java\nio\charset\Charset.html`
- `http://www.iana.org/assignments/character-sets`
- `http://www.unicode.org/`
- `http://www.unicode.org/faq/`
- `http://www.unicode.org/reports/tr10/`
- `http://www.unicode.org/reports/tr19/tr19-9.html`
- `http://www.unicode.org/reports/tr17/tr17-5.html`

Kapitola 13. XML – obecné informace

- hlavní informační zdroj/rozcestník v ČR – `www.kosek.cz`

13.1. Základní charakteristiky

- rozšiřitelný značkovací jazyk (*eXtensible Markup Language*)
- pochází z oblasti zaměřené na uchování a zpracování textových dokumentů
  - jeho „otec“ je SGML (*Standard Generalized Markup Language*)
  - jeho „bratr“ je HTML (*Hypertext Markup Language*)
- standard W3C
  - velmi rozšířen
- jednoduchý otevřený formát – lze libovolně doplňovat
- popisuje data nezávisle na platformě
  - „Java poskytuje přenositelný kód, XML přenositelná data“
- volná množina značek – rozdíl s HTML – s pevnou gramatikou
  - lze programově kontrolovat správnost struktury (a částečně i obsahu)
- principiálně textový soubor (textová informace) – textové jsou značky i data
- značkování jasně popisuje účel (rozdíl od proprietárních binárních formátů)
  - značky neurčují vzhled dat, ale jejich strukturu
    - HTML určuje jak se data zobrazí
    - XML určuje, jaký mají data význam
- na velmi jednoduchou myšlenku je navázáno velké množství technologií – viz dále
- XML je jeden z nejdůležitějších formátů výměny dat strukturovaným způsobem
- jeden konkrétní příklad

```
<lekar>
  <jmeno>
    <prijmeni>Petr</prijmeni>
    <krestni>Pavel</krestni>
  </jmeno>
  <telefon>377 123 456</telefon>
</lekar>
```


13.1.1. Co XML není

- všespasitelná technologie
- programovací jazyk – nelze přeložit do spustitelného programu
- síťový protokol (asociace HTML versus HTTP)
- databáze, ačkoliv s databázemi může spolupracovat
- není vhodný pro rozsáhlé bitové sekvence – obrázky, zvuky, video
- málo vhodný pro značně (stovky MB) rozsáhlá data (značky zvyšují velikost souboru)

13.1.2. Ověření správnosti

- díky pevné struktuře lze nezávisle na budoucí aplikaci ověřovat správnost dat
  - velká výhoda – kontrola dat se přesouvá na jejich pořizovatele
  - vstupní data se zkontrolují před zasláním do aplikace
  - aplikace může mít pak mnohem jednodušší vstup – nemusí kontrolovat chybové stavy
- několik zvyšujících se úrovní ověřování správnosti
  1. jen XML – **správně strukturovaný** (*well-formed*) dokument
     - značky se nekříží, pouze jedna je kořenová, atd. (asi 7 jednoduchých pravidel – podrobně viz dále)
  2. XML a DTD – **validní dokument**
     - je použita jasně definovaná množina značek, ve správném pořadí
  3. XML schémata (XSD)
     - jako u DTD + data mají správné typy a částečně kontrolovatelné hodnoty

13.1.3. Vývoj XML

- SGML (*Standard Generalized Markup Language*)
  - značkovací jazyk pro textové dokumenty (armáda USA, letectví)
    - správa technické dokumentace rozsahů desítek tisíc stran
  - extrémně složitý
- nejúspěšnější aplikací SGML je HTML
- vývoj XML od 1996 do 1998 verze 1.0
  - v současné době verze 1.1

13.1.4. Dvě hlavní oblasti použití

- ve skutečnosti je jich mnohem více – použití XML je rychle se rozšiřující oblast

1. **dokumenty orientované na sdělení**
   - knihy, WWW stránky, dokumentace – DocBook
   - navazující technologie např. XSLT, XSLT-FO
   - pořídí se (jednou) označkovaný text a pak se automaticky transformuje do mnoha výstupních (čitelných) formátů, např. HTML, PS, PDF, RTF, plain textu
2. **datově orientované dokumenty**
   - B2B (*bussines to bussines*) aplikace – strukturovaná data pro výměnu informací mezi aplikacemi
     - textový soubor odstiňuje:
       - rozsah čísel (int je dvoubajtové)
       - typ čísel (znaménkové int je v doplňkovém kódu)
       - způsob uložení čísel *big-* nebo *little-endian*
   - konfigurační soubory a mnohé další

13.2. Syntaxe a prvky XML

- dokument XML se skládá z textového obsahu označkovaného pomocí **textových značek** (**tagů**)
- používá se slovo „dokument“, protože to nemusí být nutně soubor (XML dokument po síti)
- značky si lze libovolně dodávat – otevřený formát – rozdíl od HTML
- značky popisují obsah nikoliv formátování
- na značky jsou mnohem přísnější pravidla než v HTML
  - dokument se hůře píše
  - dá se ale snadno správně načíst (zkontrolovat), protože je parser jednodušší
    - asi 50% kódu prohlížečů HTML řeší chybové situace, tj. problémy v datech
- **element** = počáteční a koncová značka a informace mezi nimi
- musí existovat jeden element, který je **kořenový** (vše obaluje)
- nejjednodušší XML dokument, který ale nemá žádný smysl

```
<a/>
```

  - má jeden element typu `a`, který nemá obsah
- nejjednodušší XML dokument


```
<pozdrav>ahoj</pozdrav>
```

  - má jeden element typu `pozdrav`, jehož obsah je `ahoj` typu „znaková data“

13.2.1. Názvy značek v XML

- obecně lze použít akcentovaná i neakcentovaná písmena, číslice a znaky „-“ (pomlčka) „_ “(podtržítko) a „.“(tečka)
  - nesmí začínat číslicí
- rozlišuje se velikost písmen `<POZDRAV>`, `<Pozdrav>`, `<pozdrav>` – různé
- měly by být významové – `<z1>`, `<z12>`, `<z3>` jsou sice správně, ale nic neříkají o obsahu

13.2.1.1. Praktické doporučení pro datově orientované dokumenty

- s ohledem na budoucí možnost automatizovaného generování programů (*data binding*) je výhodné
  - vytvářet názvy značek pomocí stejných pravidel jako identifikátory v Javě, např.:

    `pozdrav`, `uvitaciPozdrav`, `pozdravNaRozloucenou`
  - v identifikátorech nepoužívat
    - akcenty – teoreticky by ale neměly dělat problémy
    - pomlčky a tečky – určitě budou dělat problémy

13.2.2. Obecně platná pravidla pro značky

- každá značka musí mít svoji **uzavírací značku**
  - platí i pro **prázdnou značku**

    ```
    <bezPozdravu></bezPozdravu>
    ```

    - lze zkrátit na
      - `<bezPozdravu/>`
      - `<bezPozdravu />`
- dokumenty mají vždy strukturu stromu
  - jen jeden element je **kořenový** (**element dokumentu**)
  - všechny další elementy jsou elementy potomků
- mezery ve formátování jsou ignorovány

```
<lekar>
  <jmeno>
    <prijmeni>Petr</prijmeni>
    <krestni>Pavel</krestni>
```

```
  </jmeno>
  <telefon>377 123 456</telefon>
</lekar>
```

- každý potomek má nejvýše jednoho rodiče – nelze křížit značky – chyba:

```
<jmeno>
  <prijmeni>Petr</prijmeni>
  <krestni>Pavel</jmeno>
</krestni>
```

- elementy obsahují:
  - jiné elementy, např.:
    - `<jmeno>` obsahuje dva elementy a to `<prijmeni>` a `<krestni>`
  - data, např.:
    - `<prijmeni>` obsahuje řetězec (tj. data) `Petr`
- je možný element s kombinovaným (smíšeným) obsahem

```
<telefon>
  <predvolba> +420 </predvolba> 377 123 456
</telefon>
```

  - kterému se ale snažíme vyhnout

13.2.3. Atributy

- dvojice *název-hodnota* umístěná do počáteční značky
- hodnoty musejí být zavřeny do uvozovek (pozor " nikoliv „ nebo “ nebo ”)

```
<vaha jednotka="kg">150</vaha>
```

  - název atributu je „jednotka“
  - hodnota atributu je „kg“
- je-li v hodnotě znak uvozovek, lze hodnotu uzavřít do apostrofů

```
<znak vzhled='"' popis="uvozovky" />
```

- atributů může mít značka víc a na jejich pořadí nezáleží

```
<vaha jednotka="kg" datumVazeni="2004-12-03"> 150 </vaha>
```

- atributy představují kompaktnější zápis
  - často se snadněji zpracovávají podpůrnými technologiemi
- někdy dilema, zda použít atribut nebo obsah elementu či **vnořený element**

  - tři víceméně stejné možnosti

1. `<vaha jednotka="kg" hodnota="150"/>`
2. `<vaha jednotka="kg">150</vaha>`
3.
```
<vaha>
  <jednotka>kg</jednotka>
  <hodnota>150</hodnota>
</vaha>
```

  - nebo (nejméně vhodně) **element se smíšeným obsahem**

```
<vaha>
  <jednotka>kg</jednotka>
  150
</vaha>
```

- atributy různých elementů mohou mít stejná jména

```
<vaha jednotka="kg">45,5</vaha>

<vyska jednotka="cm">170</vyska>
```

- atribut lze vždy nahradit vnořeným elementem

13.2.4. Kdy použít elementy a kdy atributy

13.2.4.1. Atributy

- dopředu je jasná doména hodnot
  - datumy a časy
  - číselné údaje s omezeným rozsahem (`vek="1"` až `"150"`)
  - výčty čísel (`dph="22"` nebo `"19"` nebo `"5"`)
  - výčty textů (`jednotka="kg"` nebo `"g"` nebo `"libra"`)
- informaci už nelze dále strukturovat
- musí existovat pevně daný (tj. nebude se měnit) vztah k elementu
- potřeba kompaktního zápisu – hodnota atributu je uzavřená v uvozovkách, není potřeba ji uzavírat koncovou značkou
- je nevhodné umístit do atributu obecný text

```
<citát text="Byli jsme a budem!"/>
```

- u dokumentů orientovaných na sdělení by měl atribut představovat pouze doplňkovou informaci

```
<citat jazyk="čeština">
  Byli jsme a budem!
</citat>
<citat jazyk="angličtina">
  To be or not to be?
  <poznamkaPodCarou cislo="10">
    Shakespeare.
  </poznamkaPodCarou>
</citat>
```

Poznámka

Prakticky se pro označení jazyka používá **jmenný prostor** (viz dále)

```
<citat xml:lang="cs">Byli jsme a budem!</citat>

<citat xml:lang="en">To be or not to be?</citat>
```

13.2.4.2. Elementy

- opakuje-li se vícekrát – atributy musejí mít rozdílná jména

```
<vahoveZmeny periodaVazeni="7 dnů">
  <hodnota>150</hodnota>
  <hodnota>140</hodnota>
  <hodnota>130</hodnota>
</vahoveZmeny>
```

- všechny další případy, které nejsou vyjmenovány u atributů

13.2.5. Entitní reference

- náhrada problematických znaků znakového obsahu elementu
  - chybně `<nerovnost>3 < 5</nerovnost>`
  - dobře `<nerovnost>3 < 5</nerovnost>`
- předefinované entity

&lt;	<
&amp;	&
&gt;	>
&quot;	"
&apos;	'

13.2.6. Sekce CDATA

- pro výpis textu „tak, jak je“ – většinou u dokumentů orientovaných na sdělení

```
<nerovnost><![CDATA[ 3 < 5 < 7 < 9]]></nerovnost>
```

- nechává všem znakům původní význam
- použití pro výpisy programů, příkazy vnořených jazyků apod.

```
<usekProgramu jazyk="Java">
  <![CDATA[
    for (int i = 0; i < 5; i++) {
      System.out.format("%d < 5\n", i);
    }
  ]]>
</usekProgramu>
```

13.2.7. Komentáře

```
<!-- komenář -->
```

- nesmí se vnořovat a nesmí být součástí značky
- chyba `<vaha <!-- spisovně hmotnost --> >`

13.2.8. Zpracovávací instrukce

- uzavřeny do značky `<? ?>`
- příklad nejčastější instrukce, která je v naprosté většině případů jako první řádka XML dokumentu

```
<?xml version="1.0" encoding="utf-8"?>
```

13.2.9. Deklarace XML a použitý charset

- de facto povinný začátek každého XML dokumentu

```
<?xml version="1.0" encoding="utf-8"?>
```

  - pro určení charsetu zásadně používat kanonická jména – viz dříve
- XML implicitně používá znakovou sadu Unicode
  - bez určení charsetu v atributu `encoding` je dokument implicitně v UTF-8

```
<?xml version="1.0" ?>
```

- nezávisle na použitém kódování lze do dokumentu vložit jakýkoliv znak
  - je třeba znát jeho Unicode *code point* (`www.unicode.org/charts`)
  - např. Ω je:

- `&#x03A9;` (hexadecimálně)
- `&#937;` (dekadicky – nepoužívat – mate)

13.3. Ukázka výhod datově orientovaného XML dokumentu

- použití pro předávání dat mezi aplikacemi
- možnosti:
  1. binární soubor v proprietárním formátu
  2. textový soubor
  3. XML dokument

13.3.1. Binární soubor v proprietárním formátu

- nejhorší možnost, problémy s:
  - délkou řetězců
  - kódováním češtiny
  - datovými typy čísel a jejich rozsahem
  - organizací dat, atd.

13.3.2. Textový soubor

- odpadají problémy s délkou řetězců a datovými typy čísel
- zůstávají problémy kódování češtiny a hlavně význam jednotlivých dat
- lze použít jen pro jeden typ záznamu a navíc s neměnnou strukturou

```
Petr;Pavel;377 123 456
Vanda;Alexandra
mu┼┘;┼át├¡hl┠╜;Ji┼Ö├¡;150;150
┼┘ena;Ot┠╜lie;Otyl├í;45,5;170
```

Poznámka

Data jsou záměrně v neznámém charsetu, který neumí editor zobrazit.

13.3.3. XML dokument

- je jednoznačně určen charset (kódování češtiny)
- je zcela zřejmý význam jednotlivých dat

- příklad obsahující stejná data jako předcházející textový soubor

```
<?xml version="1.0" encoding="UTF-8"?>
<obezitologie>
  <personal>
    <lekar>
      <jmeno>
        <prijmeni>Petr</prijmeni>
        <krestni>Pavel</krestni>
      </jmeno>
      <telefon>377 123 456</telefon>
    </lekar>
    <sestra>
      <jmeno>
        <krestni>Vanda</krestni>
        <prijmeni>Alexandra</prijmeni>
      </jmeno>
    </sestra>
  </personal>
  <leceneOsoby>
    <nadvaha>
      <pacient pohlavi="muž">
        <jmeno>
          <prijmeni>Štíhlý</prijmeni>
          <krestni>Jiří</krestni>
        </jmeno>
        <vyska jednotka="cm">150</vyska>
        <vaha jednotka="kg">150</vaha>
      </pacient>
    </nadvaha>
    <podvyziva>
      <pacient pohlavi="žena">
        <jmeno>
          <krestni>Otýlie</krestni>
          <prijmeni>Otylá</prijmeni>
        </jmeno>
        <vaha>45,5</vaha>
        <vyska jednotka="cm">170</vyska>
      </pacient>
    </podvyziva>
  </leceneOsoby >
</obezitologie>
```

13.3.4. Jiné způsoby zápisu XML dokumentu

- předchozí dokument splňuje všechny podmínky XML (je *well-formed*) – lze jej **validovat**
- ale z hlediska dalšího zpracování (pohlížíme na něj jako na datově orientovaný dokument) není dobrý
  - má příliš „stupňů volnosti“, např.:
    - `<prijmeni>` a `<krestni>` lze prohodit

- ♦ `<telefon>` u `<sestra>` je nepovinný
- ♦ atribut `jednotka` elementu `<vaha>` je nepovinný
- ♦ elementy `<vyska>` a `<vaha>` lze prohodit

■ následují ukázky dvou různých přístupů lépe strukturovaných dokumentů

13.3.4.1. Přísně hierarchické schéma

```
<?xml version="1.0" encoding="UTF-8"?>

<obezitologie>
  <personal>
    <lekar>
      <jmeno>
        <krestni>Pavel</krestni>
        <prijmeni>Petr</prijmeni>
      </jmeno>
      <telefon>377 123 456</telefon>
    </lekar>
    <sestra>
      <jmeno>
        <krestni>Vanda</krestni>
        <prijmeni>Alexandra</prijmeni>
      </jmeno>
      <telefon>377 123 789</telefon>
    </sestra>
  </personal>

  <leceneOsoby>
    <nadvaha>
      <pacient pohlavi="muž">
        <jmeno>
          <krestni>Jiří</krestni>
          <prijmeni>Štíhlý</prijmeni>
        </jmeno>
        <vaha jednotka="kg">150</vaha>
        <vyska jednotka="cm">150</vyska>
      </pacient>
    </nadvaha>

    <podvyziva>
      <pacient pohlavi="žena">
        <jmeno>
          <krestni>Otýlie</krestni>
          <prijmeni>Otylá</prijmeni>
        </jmeno>
        <vaha jednotka="kg">45,5</vaha>
        <vyska jednotka="cm">170</vyska>
      </pacient>
    </podvyziva>
  </leceneOsoby>
</obezitologie>
```


13.3.4.2. Ploché schéma

```
<?xml version="1.0" encoding="UTF-8"?>

<obezitologie>
  <osoba role="lékař">
    <jmeno>
      <prijmeni>Petr</prijmeni>
      <krestni>Pavel</krestni>
    </jmeno>
    <telefon>377 123 456</telefon>
  </osoba>

  <osoba role="sestra">
    <jmeno>
      <krestni>Vanda</krestni>
      <prijmeni>Alexandra</prijmeni>
    </jmeno>
  </osoba>

  <osoba role="pacient - nadváha">
    <jmeno>
      <prijmeni>Štíhlý</prijmeni>
      <krestni>Jiří</krestni>
    </jmeno>
    <pohlavi hodnota="muž"/>
    <vyska jednotka="cm">150</vyska>
    <vaha jednotka="kg">150</vaha>
  </osoba>

  <osoba role="pacient - podvýživa">
    <jmeno>
      <krestni>Otýlie</krestni>
      <prijmeni>Otylá</prijmeni>
    </jmeno>
    <pohlavi/>
    <vaha>45,5</vaha>
    <vyska jednotka="cm">170</vyska>
  </osoba>
</obezitologie>
```

13.3.4.3. Porovnání obou způsobů

■ nelze obecně říci, který přístup je lepší

- pokud lze očekávat v budoucím dokumentu množství organizačních změn, je lepší ploché schéma

  `<osoba role="primář"> <osoba role="vrchní sestra">`

  - ♦ nové role se přidávají snadno bez nutnosti zásahu do stávající struktury

- je-li struktura víceméně neměnná, je lepší využít co nejvíce hierarchické schéma, tj. dodat co možná nejvíce informací

- z dlouhodobých zkušeností víme, že se data mění méně (pomaleji) než aplikace:
  - čím více informace dokument obsahuje, tím lépe
  - nevadí, když se v jednotlivých aplikacích všechny informace pokaždé nevyužijí

13.4. Kontrola XML dokumentu

- nejnižší úroveň kontroly (validace)
  - dokument je **správně strukturován** (*well-formed*)
- existuje sedm oblastí základní kontroly (všechny příklady jsou chybové):

1. musí existovat právě jeden kořenový element

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální</jazykC>
<jazykJava>objektový</jazykJava>
```

2. každá počáteční značka má odpovídající ukončující značku

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální
```

3. elementy se nesmějí překrývat (křížit)

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC>procedurální
  <charakteristika>univerzální
  </jazykC>
</charakteristika>
```

4. hodnoty atributů musí být v uvozovkách (nebo v apostrofech)

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC vyucujici=Herout>
  procedurální
</jazykC>
```

5. element nesmí mít stejně pojmenované atributy

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC Herout="přednáší" Herout="zkouší">
  procedurální
</jazykC>
```

6. komentáře nesmí být vnořené a ani uvnitř značek


```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC <!-- už dlouho --> přednáší="Herout">
  procedurální
</jazykC>
```

7. ve znakových datech nejsou znaky `<` a `&`

```
<?xml version="1.0" encoding="UTF-8"?>
<jazykC přednáší="Herout">
  operátor & vrátí adresu
</jazykC>
```

- při průběhu kontroly se nic automaticky neopravuje!
  - jednoznačnost zdrojového XML dokumentu
- nejjednodušší kontrola – zobrazit v HTML prohlížeči, např. MSIE 6
- prakticky jsou kontroly prováděny již hotovými parsery
- validace pomocí parseru xerces

```
>xerces.bat soubor.xml
```

- kde `xerces.bat` je soubor uložený v cestě `PATH` s jednořádkovým obsahem

```
@java -cp "c:\Program Files\Java\xerces\xercesImpl.jar";"c:\Program Files\Ja-
va\xerces\xmlParserAPIs.jar";"c:\Program Files\Java\xerces\xercesSamples.jar"
sax.Counter %*
```

- pro správně formulovaný dokument vypíše:

```
>xerces obezitologie-pevne.xml
obezitologie-pevne.xml: 71 ms (27 elems, 6 attrs, 0 spaces, 358 chars)
```

13.5. Jmenné prostory (XML namespaces)

- možnost kombinovat více sad značek v jednom dokumentu
- každá sada značek je jednoznačně definována svojí URI adresou
  - URI (*Uniform Resource Identifier*) je nadmnožina URL (*Uniform Resource Locator*)
  - URL jednoznačně určuje umístění dokumentu v síti
- proč se to dělá?
  - používají se již hotové sady značek buď z DTD nebo z XSD
  - dokument vytváří/doplňuje více lidí
- pro použití elementu z určité sady značek je třeba určit identifikátor (*prefix*) této sady

- jméno prefixu je libovolné, mělo by být co nejkratší
- prefix se určuje pomocí speciálního atributu `xmlns` (XML namespaces)

```
xmlns:jmenoPrefixu="URI sady značek"
```

- prefix se pak používá před každým názvem elementu nebo atributu

```
<?xml version="1.0" encoding="UTF-8"?>
<prj5:evidencePredmetuPRJ5
  xmlns:prj5="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
  xmlns:osobni="http://www.kiv.zcu.cz/~herout/xml/osobni-sada" >

  <osobni:identifikace>
    <osobni:cislo>
      A12345
    </osobni:cislo>
    <osobni:jmeno>
      Pavel Herout
    </osobni:jmeno>
  </osobni:identifikace>

  <prj5:jmeno>
    Praktické poznatky z využití XML v lihovarnictví
  </prj5:jmeno>
  <prj5:hodnoceni prj5:bodu="100"/>
</prj5:evidencePredmetuPRJ5>
```

- URI sady značek musí být unikátní
  - většinou se používá URL a odpovídající soubor nemusí existovat
  - URI pouze zajišťuje jednoznačné pojmenování
- v XML dokumentu pak lze použít dva elementy pojmenované `<jmeno>` rozlišené prefixem
  - `<osobni:jmeno>`
  - `<prj5:jmeno>`
- je možné jeden jmenný prostor deklarovat jako implicitní
  - typicky pokud jedna sada značek výrazně převažuje nad druhou
  - lze pak vynechat jeho prefix

```
<?xml version="1.0" encoding="UTF-8"?>
<evidencePredmetuPRJ5
  xmlns="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
  xmlns:osobni="http://www.kiv.zcu.cz/~herout/xml/osobni-sada"
>

  <osobni:identifikace>
    <osobni:cislo>
      A12345
    </osobni:cislo>
    <osobni:jmeno>
      Pavel Herout
    </osobni:jmeno>
  </osobni:identifikace>

  <jmeno>
    Praktické poznatky z využití XML v lihovarnictví
  </jmeno>
  <hodnoceni bodu="100"/>
</evidencePredmetuPRJ5>
```

- pozor na to, že implicitní jmenný prostor se nevztahuje na atributy
  - zde to nevadí, v budoucnu při použití XSD bude
- jmenné prostory mají platnost pro všechny podřazené elementy
- dále platí pravidlo, že později deklarovaný zastiňuje dříve deklarovaný

```
<?xml version="1.0" encoding="UTF-8"?>
<evidencePredmetuPRJ5
  xmlns="http://www.kiv.zcu.cz/~herout/xml/prj5-sada"
>

  <identifikace
    xmlns="http://www.kiv.zcu.cz/~herout/xml/osobni-sada"
  >
    <cislo>
      A12345
    </cislo>
    <jmeno>
      Pavel Herout
    </jmeno>
  </identifikace>

  <jmeno>
    Praktické poznatky z využití XML v lihovarnictví
  </jmeno>
  <hodnoceni bodu="100"/>
</evidencePredmetuPRJ5>
```

- toto je prakticky využitelná možnost, protože se elementy z různých sad málokdy „míchají do sebe“
- jmenné prostory mají mnoho dalších možností – viz literatura

13.6. Literatura

Kosek, J.: *XML pro každého*, GRADA, 2000

Marchal, B.: *XML v příkladech*, Computer Press, 2000

Harold, E.R. – Means, W.S.: *XML v kostce*, Computer Press, 2002