

Reprezentace základních typů, pole, funkce a procedury

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 4

A0B36PR1 – Programování 1

Část 1 – Reprezentace základních typů

Základní typy a reprezentace dat v počítači

Typové konverze

Část 2 – Pole

Reprezentace pole

Pole v Javě

Příklady

Přířazení

Část 3 – Funkce a procedury

Dekompozice

Deklarace

Kódovací konvence

Předávání parametrů

Příklady

Část I

Reprezentace základních typů

Příklad typ `int` Java

Příklad – celočíselný typ `int` v Javě

- Umožňuje uložit celá čísla v intervalu $\langle -2147483648, 2147483647 \rangle$
- Můžeme použít například
 - aritmetické operace `+`, `-`, `*`, `/` s výsledkem hodnota typu `int`
 - relační operace `==`, `!=`, `>`, `<`, `>=`, `<=` jejichž výsledkem je hodnota typu `boolean`

```

1 int i; //deklarace promenne typu int
2 int decI = 120; //deklarace spolu s prirazeniem
3 int hexI = 0x78; //pocatecni hodnota v 16-kove soustave
4
5 int sum = 10 + decI + 0x13; //pocatecni hodnota je vyraz

```

lec04/DemoTypes.java

Datové typy

- Při návrhu algoritmu abstrahujeme od binární podoby paměti počítače
- S daty pracujeme jako s hodnotami různých datových typů, které jsou uloženy v paměti předepsaným způsobem
- Datový typ specifikuje:
 - Množinu hodnot, které je možné v počítači uložit
 - Množinu operací, které lze s hodnotami typu provádět
- **Jednoduchý typ** je takový typ, jehož hodnoty jsou atomické, tj. z hlediska operací dále nedělitelné

Záleží na způsobu reprezentace

Reprezentace dat v počítači

- V počítači není u datové položky určeno jaký konkrétní datový typ je v paměti uložen
- V jazyce Java musíme přidělení paměti **deklarovat** s jakými typy dat budeme pracovat
- Překladač jazyka Java pak tuto deklaraci hlídá a volí odpovídající strojové instrukce pro práci s datovými položkami například jako s odpovídajícími číselnými typy

javac

Příklad ekvivalentních reprezentací v paměti počítače

- $(0100\ 0001)_2$ – binární zápis jednoho bajtu (8-mi bitů);
- $(65)_{10}$ – odpovídající číslo v dekadické soustavě;
- $(41)_{16}$ – odpovídající číslo v šestnáctkové soustavě;
- znak `A` – tentýž obsah paměťového místa $(0100\ 0001)_2$ o velikosti 1 byte může být interpretován také jako znak `A`.

Reprezentace celých čísel

- Číselné soustavy – poziční číselné soustavy (polyadické) jsou charakterizovány bází udávající kolik číslic lze maximálně použít
 $x_d = \sum_{i=-n}^{i=m} a_i \cdot z^i$, kde a_i je číslice a z je základ soustavy

- Unární – např. počet vypitých půllitrů

- Binární soustava (bin) – 2 číslice 0 nebo 1

$$\begin{aligned} 11010,01^2 &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \\ &= 26,25 \end{aligned}$$

- Desítková soustava (dec) – 10 číslic, znaky 0 až 9

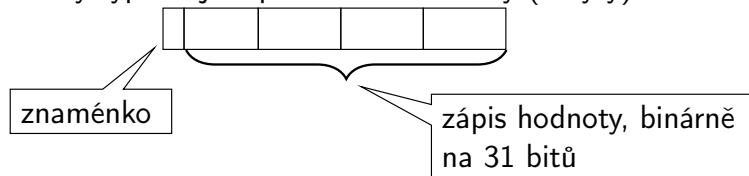
$$\begin{aligned} 138,24 &= 1 \cdot 10^2 + 3 \cdot 10^1 + 8 \cdot 10^0 + 2 \cdot 10^{-1} + 4 \cdot 10^{-2} \\ &= 1 \cdot 100 + 3 \cdot 10 + 8 \cdot 1 + 2 \cdot 0,1 + 4 \cdot 0,01 \end{aligned}$$

- Šestnáctková soustava (hex) – 16 číslic, znaky 0 až 9 a A až F

$$\begin{aligned} 0x7D_h &= 7 \cdot 16^1 + D \cdot 16^0 \\ &= 112 + 13 \\ &= 125 \end{aligned}$$

Typ `int` – celá čísla v Javě

- Celočíselný typ `int` je reprezentován 32 bity (4 byty)



- Typ `int` je znaménkový typ
- Znaménko je zakódováno v 1 bitu a vlastní číselná hodnota pak ve zbývajících 31 bitech

- Největší číslo je $0111 \dots 111 = 2^{31} - 1 = 2147483647$

Nezapomínat na 0

- Nejmenší číslo je $-2^{31} = -2147483648$

0 už je zahrnuta

- Pro zobrazení záporných čísel je použit tzv. **doplňkový kód**
 Nejmenší číslo v doplňkovém kódu $1000 \dots 000$ je -2^{31}

Více-bajtová reprezentace a pořadí bajtů

- Číselné typy s více-bajtovou reprezentací mohou mít bajty uloženy v různém pořadí

- little-endian** – **nejméně** významný bajt se ukládá na nejnižší adresu x86

- big-endian** – **nejvíce** významný bajt se ukládá na nejnižší adresu Motorola

- Pořadí je důležité při přenosu hodnot z paměti jako posloupnosti bajtů a jejich následné interpretaci

- Network byte order** – je definován pro síťový přenos a není tak nutné řešit konkrétní architekturu

- Tj. hodnoty z paměti jsou ukládány a přenášeny v tomto pořadí bajtů a na cílové stanici pak zpětně zapsány do konkrétního nativního pořadí

big-endian

- Java používá **network byte order**

Informativní

Reprezentace záporných celých čísel

- Doplňkový kód – $D(x)$
- Pro 8-mi bitovou reprezentací čísel
 - Můžeme reprezentovat $2^8 = 256$ čísel
 - Rozsah $r = 256$

$$D(x) = \begin{cases} x & \text{pro } 0 \leq x < \frac{r}{2} \\ r + x & \text{pro } -\frac{r}{2} \leq x < 0 \end{cases} \quad (1)$$

- Příklady

Desítkově	Doplňkový kód
0–127	0000 0000 – 0111 1111
128	nelze zobrazit na 8 bitů v doplňkovém kódu
-128	$D(-128) = 256 + (-128) = 128$ to je 1000 0000
-1	$D(-1) = 256 + (-1) = 255$ to je 1111 1111
-4	$D(-4) = 256 + (-4) = 252$ to je 1111 1100

Informativní

Reprezentace reálných čísel

- Pro uložení čísla vyhrajujeme omezený paměťový prostor

Příklad – zápis čísla $\frac{1}{3}$ v dekadické soustavě

- = 33333333...3333
- = $0,3\bar{3}$
- $\approx 0,33333333333333333333$
- $\approx 0,333$

V trojkové soustavě: $0 \cdot 3^1 + 0 \cdot 3^0 + 1 \cdot 3^{-1} = (0,1)_3$

- Nepřesnosti v zobrazení reálných čísel v konečné posloupnosti bitů způsobují
 - Iracionální čísla, např. e , π , $\sqrt{2}$
 - Čísla, která mají v dané soustavě periodický rozvoj, např. $\frac{1}{3}$
 - Čísla, která mají příliš dlouhý zápis

Příklad modelu reprezentace reálných čísel 1/2

Reprezentace na 7 bajtů

- Délka mantisy 3 pozice (bajtů) plus znaménko
- Délka exponentu 2 pozice plus znaménko
- Základ $z = 10$
- Nula

? ?? + 000

- Příklad $x = 77,5 = 0,775 \cdot z^{+02}$

+ 02 + 775

Model reprezentace reálných čísel

- Reálná čísla se zobrazují jako aproximace daným rozsahem paměťového místa
- Reálné číslo x se zobrazuje ve tvaru

$$x = \text{mantisa} \cdot \text{základ}^{\text{exponent}}$$

$$x = m \cdot z^{\text{exponent}}$$

- Pro jednoznačnost zobrazení musí být mantisa normalizována

$$0,1 \leq m < 1$$

- Ve vyhrazeném paměťovém prostoru je pro zvolený základ uložen exponent a mantisa jako dvě celá čísla



Příklad modelu reprezentace reálných čísel 2/2

Limitní zobrazitelná čísla

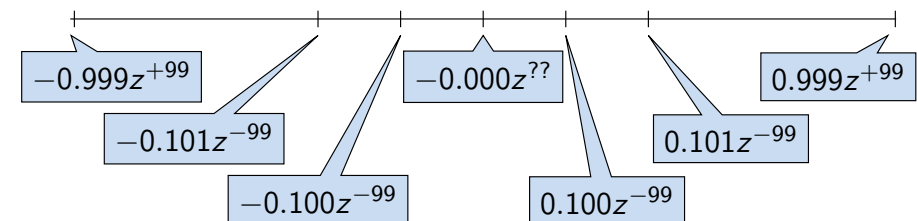
- Maximální zobrazitelné kladné číslo $0,999z^{99}$
- Maximální zobrazitelné záporné číslo $-0,100z^{-99}$
- Minimální zobrazitelné kladné číslo $0,100z^{-99}$
- Minimální zobrazitelné záporné číslo $-0,999z^{+99}$

+ 99 + 999

- 99 - 100

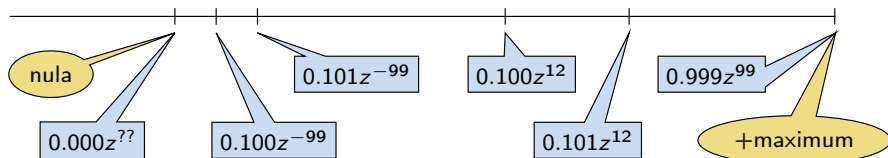
- 99 + 100

+ 99 - 999



Model reprezentace reálných čísel a vzdálenost mezi aproximacemi

- Rozsah hodnot pro konkrétní exponent je dán velikostí mantisy
- Absolutní vzdálenost dvou aproximací tak záleží na exponentu
 - Mezi hodnotou 0 a 1,0 je využit celý rozsah mantisy pro exponenty $\{-99, -98, \dots, 0\}$



- Aproximace reálných čísel nejsou na číselné ose rovnoměrně rozloženy



Přřazovací operátor a příkaz

- Slouží pro nastavení hodnoty proměnné
 - Uložení číselné hodnoty do paměti, kterou proměnná reprezentuje*
- Tvar přřazovacího operátoru

$$\langle \text{proměnná} \rangle = \langle \text{výraz} \rangle$$

Výraz je literál, proměnná, volání funkce, ...
- Zkrácený zápis

$$\langle \text{proměnná} \rangle \langle \text{operátor} \rangle = \langle \text{výraz} \rangle$$
- Přřazení je výraz
 - Asociativní zprava
- Přřazovací příkaz – výraz zakončený středníkem ;

```
int x; //deklarace promenne x
int y; //deklarace promenne y
x = 6;
y = x + 6;

int x, y; //deklarace promennych x a y
x = 10;
y = 7;
y += x + 10;
```

Typ **double** – reprezentace necelých čísel v Java

- **double** – 64 bitů (8 bajtů), norma IEEE 754

ISO/IEC/IEEE 60559:2011

- **s** – 1 bit znaménko (+ nebo –)
- **exponent** – 11 bitů, tj. 2048 možností
- **mantisa** – 52 bitů \approx 4.5 biliardy možností

4 503 599 627 370 496

- Neumožňuje přesně uložit čísla se zápisem delším než 52 bitů
- Čím větší exponent tím větší „mezery“ mezi sousedními aproximacemi čísel

- Reálné číslo x se zobrazuje ve tvaru

$$x = (-1)^s \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$$

- **bias** umožňuje reprezentovat exponent vždy jako kladné číslo

Lze zvolit, např. $\text{bias} = 2^{eb-1} - 1$, kde eb je počet bitů exponentu

<http://www.root.cz/clanky/>

[norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky](#)

Typové konverze

- Typová konverze je operace převedení hodnoty nějakého typu na hodnotu typu jiného
- Typová konverze může být
 - **implicitní** – vyvolá se automaticky
 - **explicitní** – je nutné v programu explicitně uvést
- Konverze typu **int** na **double** je v jazyku Java implicitní
 - Hodnota typu int může být použita ve výrazu, kde se očekává hodnota typu double, dojde k automatickému převodu na hodnotu typu double.*

Příklad

```
double x;
int i = 1;

x = i; //hodnota 1 typu int se automaticky
prevede na hodnotu 1.0 typu double
```

- Implicitní konverze je bezpečná

Explicitní typové konverze

- Převod hodnoty typu **double** na **int** je třeba **explicitně** předeepsat
- Dojde k „odseknutí“ necelé části hodnoty int

Příklad

```
double x = 1.2; // deklarace promenne typu
double
int i; // deklarace promenne typu int
int i = (int)x; // hodnota 1.2 typu double se
prevede na hodnotu 1 typu int
```

- Explicitní konverze je potenciálně nebezpečná

Příklady

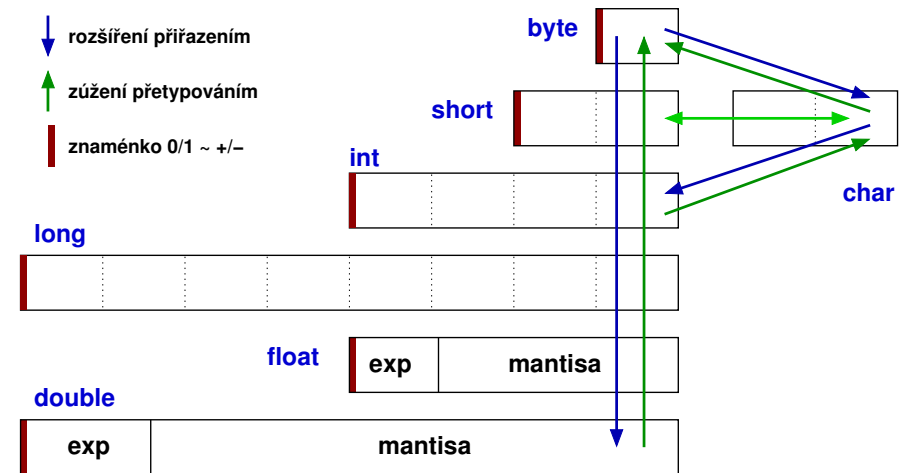
```
double d = 1e30; // i je 2147483647 // to je asi 2e9 místo 1e30
int i = (int)d;
long l = 5000000000L; // i je 705032704 // (oriznute 4 bajty)
int i = (int)l;
```

Část II

Pole

Konverze primitivních číselných typů

- Primitivní datové typy jsou vzájemně nekompatibilní, ale jejich hodnoty lze převádět



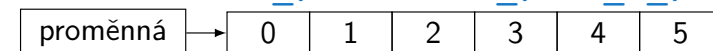
Pole (statické délky)

- Datová struktura pro uložení více hodnot **stejného typu**
- Slouží k reprezentaci posloupnosti hodnot v paměti
- Jednotlivé prvky mají identickou velikost a jejich relativní adresa vůči počátku pole je tak jednoznačně určena

- Prvky můžeme adresovat pořadím prvku v poli

Relativní adresa vůči prvnímu prvku

„adresa“ = velikost_prvku * index_prvku_v_poli



- Proměnná typu pole reprezentuje adresu vyhrazeného paměťového prostoru, kde je pole skutečně uloženo
- Deklarací proměnné dochází k alokaci paměti pro uložení odkazu na začátek pole
- **Velikost pole statické délky nelze měnit**

Garance souvislého přístupu k položkám pole

Deklarace pole v Javě

- Proměnná typu pole je referenční typ

Druhý referenční typ je objekt

- Hodnota proměnné typu pole je reference (odkaz / adresa) na místo v paměti, kde je pole uloženo
- Deklarace proměnné typu pole se skládá z typu prvků, jména proměnné a hranatých závorek []
`typ proměnná [];`

- Závorky [] slouží také k přístupu (adresaci) prvku pole
`proměnná _typu _pole [index _prvku _pole]`

Příklad deklarace proměnné typu pole hodnot typu `int` a alokace paměti pro až 10 prvků pole

```
int values[] = new int[10];
```

Deklarace a alokace pole v Javě

Příklad deklarace a alokace pole v Javě

```
int values[]; //deklarace promenne typu pole int
                //hodnot, pocatecni hodnotu je nutne
                //nastavit, pokud nezname velikost
                //tak muzeme nastavit na null

values = new int[10]; //alokace pameti pro 10 int hodnot

for (int i = 0; i < 10; i++) {
    values[i] = 3*i - 2*i*i; //naplneni hodnot prvku pole
}

values[10] = 10; //adresace mimo rozsah pole neni dovolena

int n = 5;
values = new int[n * 2]; //alokace noveho pole
                //velikost je vyraz s hodnotou
                //typu int

for (int i = 0; i < values.length; ++i) {
    //pocet prvku pole je pristupny pres prommenou typu pole
    System.out.println("values[" + i + "]: " + values[i]);
}

lec04/DemoArray.java
```

Proměnná typu pole, deklarace a přidělení paměti

- Proměnná typu pole odkazuje na místo v paměti, kde je pole konkrétní délky umístěno
- Deklarací proměnné typu pole alokujeme (a pojmenováváme) místo v paměti pro uložení odkazu, kde jsou uloženy prvky pole
Pole je referenční typ a v Javě musí být před prvním použitím pole explicitně nastavena jeho počáteční hodnota, např. `null`.
- Deklarací proměnné typu pole tak **nealokujeme** vlastní prostor pro prvky pole
- Alokaci paměti pro uložení hodnot prvků pole **konkrétní** délky provádíme operátorem `new` následovaný typem (prvků) a počtem prvků pole v hranatých závorkách []

```
proměnná _typu _pole = new typ [počet _prvků _pole]
```

Hodnoty prvků jsou nastaveny na výchozí hodnotu typu.

Specifikace JVM (Java Virtual Machine) neuvádí, že je pole uloženo v paměti jako souvislý blok paměti. Zpravidla tomu tak bývá např. v programovacím jazyku C. Java to však nespécifikuje a pouze garantuje přístup k prvkům pole prostřednictvím indexu, tj. není třeba se starat o adresaci prvků.

Příklad – Pole řetězců 1/3

- Vytvořte program, který načte 5 řádků ze souboru a následně načítá textový vstup od uživatele a vypíše čísla řádků s výskytem zadaného řetězce.
- Pro testování výskytu řetězce v řetězci použijte metodu `indexOf` třídy `String`

```
1 public String[] loadLines(int numberOfLines, String filename) {
2     String lines[] = new String[numberOfLines];
3     try {
4         Scanner scanner = new Scanner(new FileReader(filename));
5         for(int i = 0; i < lines.length; ++i) {
6             lines[i] = scanner.nextLine();
7             System.err.println("Info: line[" + i + "]: " + lines[i]);
8         }
9     } catch (FileNotFoundException e) {
10        System.err.println("Error: filename '" + filename + "' not
11        found!");
12    }
13    return lines;
14 }

lec04/DemoArrayString.java
```

Příklad – Pole řetězců 2/3

```

1 public void printWordOccurance(String[] lines, String word) {
2     String wordOccurance = "";
3     int counter = 0;
4     for(int i = 0; i < lines.length; ++i) {
5         if (lines[i] != null && lines[i].indexOf(word) != -1) {
6             counter += 1;
7             wordOccurance += " " + i;
8         }
9     }
10    if (counter > 0) {
11        System.out.println("Word '" + word + "' detected in " +
12        counter + " lines: " + wordOccurance);
13    } else {
14        System.out.println("Word '" + word + "' has zero occurrence
15        in the input lines");
16    }
17 }

```

- Explicitně testujeme, zda-li je řádek v poli nenulový, pokud se například soubor nepodaří kompletně načíst

lec04/DemoArrayString.java

Příklad – reprezentace matice 1/5

- Vytvořte program pro reprezentaci matice hodnot typu double s rozměrem $N \times M$, jako pole polí hodnot typu double
 - Napište funkci pro tisk matice na obrazovku
 - Napište funkci pro součet dvou matic
- Dekompozice programu na *Co musí být splněno pro součet dvou matic?*
 1. Alokace paměti pro matici
 2. Vyplnění matice náhodnými hodnotami pro otestování
 3. Tisk proměnné typu pole polí (matice) na obrazovku
 4. Součet dvou matic

```

1 public double[][] createMatrix(int n, int m) {
2     double[][] matrix = new double[n][m];
3     for(int row = 0; row < matrix.length; ++row) {
4         matrix[row] = new double[m];
5     }
6     return matrix;
7 }

```

lec04/DemoArrayOfArray.java

- Nebo jen

```

1 public double[][] createMatrix(int n, int m) {
2     return new double[n][m];
3 }

```

Příklad – Pole řetězců 3/3

```

1 public static void main(String[] args) {
2     final String FILENAME = "lines.txt";
3     final int NUMBER_OF_LINES = 5;
4     DemoArrayString demo = new DemoArrayString();
5     Scanner scanner = new Scanner(System.in);
6
7     String[] lines = demo.loadLines(NUMBER_OF_LINES, (args.length
8     > 0 ? args[0] : FILENAME));
9
10    String word = "";
11    do {
12        System.err.print("Enter a word: ");
13        System.err.flush();
14        word = scanner.nextLine();
15        if (word.length() > 0) {
16            demo.printWordOccurance(lines, word);
17        }
18    } while (word != null && word.length() > 0);
19 }

```

- Vyzkoušejte si program sami napsat a otestujte jeho chování pro různé vstupy!

lec04/DemoArrayString.java

Příklad – reprezentace matice 2/5

```

1 public void fillMatrix(double[][] matrix) {
2     if (matrix != null) {
3         for(int row = 0; row < matrix.length; ++row) {
4             if (matrix[row] != null) {
5                 for(
6                     int column = 0;
7                     column < matrix[row].length;
8                     ++column
9                 ) {
10                    matrix[row][column] = Math.random() * 10;
11                }
12            }
13        }
14    }
15 }

```

- Explicitně testujeme alokaci polí a jejich velikost

lec04/DemoArrayOfArray.java

Příklad – reprezentace matice 3/5

```

1 public void printMatrix(double[][] matrix) {
2     if (matrix != null) {
3         for(int row = 0; row < matrix.length; ++row) {
4             if (matrix[row] != null) {
5                 for(int column = 0; column < matrix[row].length; ++column)
6                 {
7                     final String space = column > 0 ? " " : "";
8                     System.out.printf("%s%4.1f", space, matrix[row][column]);
9                 }
10                System.out.println(""); //print new line after row
11            }
12        }
13    }
14 }

```

- Každý řádek matice vytiskneme na samostatný řádek, tj. za vytištěným řádkem odřádkujeme (tiskneme nový konec řádku)

lec04/DemoArrayOfArray.java

Příklad – reprezentace matice 5/5

```

1 public static void main(String[] args) {
2     final int N = 3;
3     final int M = 4;
4     DemoArrayOfArray demo = new DemoArrayOfArray();
5     double[][] matrix = demo.createMatrix(N, M);
6
7     System.out.println("Matrix after initialization:");
8     demo.printMatrix(matrix);
9     demo.fillMatrix(matrix);
10
11    System.out.println("\nFirst matrix:");
12    demo.printMatrix(matrix);
13
14    System.out.println("\nSecond matrix:");
15    double[][] matrix2 = demo.createMatrix(N, M);
16    demo.fillMatrix(matrix2);
17    demo.printMatrix(matrix2);
18
19    System.out.println("\nSum of matrices");
20    double[][] sum = demo.sum(matrix, matrix2);
21    demo.printMatrix(sum);
22 }

```

- Vyzkoušejte si program sami napsat a otestujte chování funkcí pro různé vstupy!

lec04/DemoArrayOfArray.java

Příklad – reprezentace matice 4/5

```

1 public double[][] sum(double[][] m1, double[][] m2) {
2     double[][] sum = null;
3     if (
4         m1 != null && m2 != null &&
5         m1.length == m2.length
6     ) {
7         sum = new double[m1.length][];
8         for(int r = 0; r < m1.length; ++r) {
9             if (
10                m1[r] != null && m2[r] != null &&
11                m1[r].length == m2[r].length
12            ) {
13                sum[r] = new double[m1[r].length];
14                for(int c = 0; c < m1[r].length; ++c) {
15                    sum[r][c] = m1[r][c] + m2[r][c];
16                }
17            } else {
18                System.err.println("Error: matrix dimensions does not
19                match!");
20                sum = null;
21                break;
22            }
23        }
24        return sum;
25    }
26 }

```

lec04/DemoArrayOfArray.java

Přirazení mezi referenčními proměnnými typu pole

```

int[] a = new int[5];
int[] b = a;

```

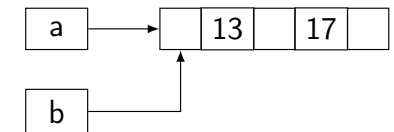
```

b[1] = 13;
a[3] = 17;

```

```
System.out.println(a[1]); //vypise 13
```

```
System.out.println(b[3]); //vypise 17
```



- Po přirazení obě proměnné odkazují na stejné pole
- Přirazení hodnot pole není v Javě definováno

```

b = new int[a.length]
for (int i = 0; i < a.length; ++i) {
    b[i] = a[i];
}

```

Pro kopírování obsahu pole lze použít systémové knihovny
System.arraycopy()

Pole v Javě – shrnutí 1/2

- Pole n prvků typu **T** lze v Javě vytvořit pouze operátorem **new T[n]**
Dynamické vytvoření
- Referenci na vytvořené pole typu **T** lze uložit do **referenční proměnné** typu **T[]**
- **Referenční proměnnou pole lze deklarovat bez vytvoření pole** deklarací např. `int[] a;`
- Před prvním použitím referenční proměnné pole je nutné přiřadit referenci na vytvořené pole, např. `a = new int[10];`
- Velikost vytvořeného pole nelze měnit
Lze vytvořit pole nové a obsah zkopírovat
- Po vytvoření pole mají prvky výchozí hodnotu
- Přístup k prvkům pole je přes operátor `[i]`, kde i je **celočíslný výraz** jehož hodnota je **nezáporná** a **menší než počet prvků pole**
 $0 \leq i < a.length$

Část III

Funkce a procedury

Pole v Javě – shrnutí 2/2

- Indexace mimo rozsah pole způsobí chybu běhu programu `java.lang.ArrayIndexOutOfBoundsException`.
- Počet prvků pole (např. `a`) je přístupný přes položku `length` referenční proměnné typu pole, např. `a.length`
Hodnota referenční proměnné nesmí být null.
- Pole lze zavést definicí hodnot prvků, například:

```
String[] months = { "jan", "feb", ..., "dec" };
String monthStr = (args.length > 0) ? args[0] : "Jun";
int month = -1;
monthStr = monthStr.substring(0, 3);
for(int i = 0; i < months.length; ++i) {
    if (monthStr.equalsIgnoreCase(months[i])) {
        month = i + 1;
        break;
    }
}
if (month >= 0 && month < months.length) {
    System.out.println("Parsed month '" + ...);
...

```


lec04/DemoArrayMonth.java vs lec03/DemoSwitchMonth.java

Funkce a procedury

- Funkce a procedury jsou označeny (**pojmenované**) části kódu (posloupnosti příkazů)
- Jsou klíčovým elementem pro **dekompozici** řešení problému na dílčí části
- „Zapouzdřují“ nějakou konkrétní činnost – dílčí řešení výpočtu
 - Pomáhají zvýšit čitelnost a udržitelnost programu
 - Zvyšují znovu použitelnost konkrétních částí programu
- Můžeme rozlišit
 - **Funkce** – definovány vstupní hodnoty a návratová hodnota funkce
 - Vstupní hodnoty jsou použity pro výpočet hodnoty funkce
Jedna návratová hodnota, ale můžeme „vyplňovat“ více hodnot / proměnných vstupních parametrů
 - **Procedura** – definovány vstupní hodnoty a činnost procedury
 - **Metody** – funkce nebo procedury v objektově orientovaném programování (OOP), někdy nazývané služby třídy / objektu

Deklarace funkce

- Deklarace funkce
 - Hlavička funkce
 - Tělo funkce
- V Javě má základní hlavička funkce (metody) tvar

typ jméno (parametry funkce)

 - **typ** – výsledku funkce (funkční hodnoty)
 - **jméno** – identifikátor funkce
 - **parametry** – seznam definic parametrů je ve tvaru **typ jméno** oddělené čárkou *prázdná specifikace – funkce bez parametrů*
- Tělo funkce je složený příkaz nebo blok, který se provede při volání funkce
- Tělo funkce musí dynamicky končit příkazem **return x;** kde x je výraz, jehož hodnota je výsledek volání funkce
 - Typ výsledku musí být shodný s typem v hlavičce
 - Pro funkce s prázdným typem **void** (procedury) není **return** nutný

Tělo funkce

- Složený příkaz nebo blok (posloupnost příkazů) vymezená složenými závorkami { a }
- Zápis těla funkce je **definice funkce**
- Deklarace proměnných v těle funkce se řídí stejnými pravidly jako v případě bloku, tj.
 - Rozsah platnosti je pouze uvnitř těla funkce
 - Proměnná stejného jména jako „globální“ proměnná zastihuje tu globální
- Vstupní parametry jsou proměnné definovaného typu a představují deklaraci lokálních proměnných, které jsou inicializovány na hodnotu předávanou voláním funkce

Příklady deklarací funkcí

```
int computeFactorial(int n) {
    // funkce pro vypocet n!
    // jeden vstupni parametr typu int
    // navratova hodnota typu int
    int factorial = 1;
    ...
    return factorial;
}

int getRandomValue() {
    // funkce vraci nahodne cislo typu int
    int random;
    ...
    return random;
}

void execute() {
    // funkce bez parametru
    // navratovy typ void definuje prazdny navratovy typ, tj
    // funkce nevraci zadnou hodnotu
    // v podstate se tak jedna o proceduru
    ...
}
```

Příklad definice funkce

Příklad výpočtu faktoriálu celého čísla

```
int computeFactorial(int n) {
    int factorial = 1; //lokální proměnná
    for(int i = 1; i <= n; ++i) {
        factorial *= i;
    }
    return factorial;
}

public static void main(String[] args) {
    DemoFactorial demo = new DemoFactorial();
    int n = 6;

    //predani funkci computeFactorial hodnoty promenne n
    int f = demo.computeFactorial(n);

    //predani hodnoty vyrazu, tj. 2*6 = 12
    int f = demo.computeFactorial(2*n);
}
```

lec04/DemoFactorial.java

Předávání parametrů

- Deklarace funkce obsahuje formální parametry funkce pro předávání dat do funkce (metody)
- Formální parametry jsou proměnné uvedené v kulaté závorce hlavičky funkce
 - Jsou to lokální proměnné funkce
 - Při volání funkce se jim **přiradí** hodnoty skutečných parametrů
- Skutečné hodnoty parametrů se přiřazují formálním parametrům **voláním hodnotou**
- Přípustný datový typ skutečného parametru vzhledem k datovému typu formálního parametru se řídí stejnými pravidly jako v případě **přiřazení**:
 - identické typy
 - automatická konverze typu
 - vynucená konverze typu
 - v případě nepovolené konverze nelze hodnotu přiřadit, např. typ `boolean` na typ `int`

Způsoby předávání parametrů

- Existují dva základní mechanismy předávání parametrů
 - „Volání hodnotou“ (Call by Value)
 - „Volání odkazem“ (Call by Reference)
- **Volání hodnotou**
 - Při volání funkce jsou předávány formálním parametrům kopie hodnot skutečných parametrů
 - Změnou formálního parametrů ve funkci tak **nelze změnit** hodnotu původního skutečného parametru

Metoda zná jen kopii hodnoty nikoliv adresu skutečného parametru
 - Metoda / funkce tak nemůže ovlivnit své okolí *Omezení*
 - Formální parametr volaný hodnotou nelze použít jako výstupní bod z funkce (metody)
- **Volání odkazem**
 - V místě předání skutečných parametrů do formálních se předává reference na skutečný parametr

V Javě je možné pouze volání hodnotou!

Základní doporučení pro zápis funkcí / procedur

- Funkce by měly být krátké
 - Funkce by měla dělat jen jednu věc

Např. do 20 řádků, ale žádné takové pravidlo není striktní
- Malý počet parametrů (argumentů)
- Jméno funkce volíme jako sloveso
 - V Javě zapisujeme malými písmeny
 - V případě víceslovného jména slova spojujeme a první písmeno dalších slov píšeme velké

např. `int computeFactorial(int n)`
- Jméno funkce a parametrů volíme tak, aby vyjadřovalo pořadí parametrů
- Snažíme se vyvarovat přepínání činnosti funkce hodnotou vstupních parametrů

Příklady volání hodnotou 1/3

- Voláním hodnotou dochází k vytvoření kopií hodnot proměnných a nastavení lokálních proměnných funkce definovaných v hlavičce

```

1  int computeAvg(int x, int y) {
2      System.out.println("computeAvg: x: " + x + " y: " + y);
3      x = x + y;
4      y = x / 2;
5      return y;
6  }
7
8  ...
9  int x = 1;
10 int y = 7;
11
12 System.out.println("x: " + x + " y: " + y);
13 //vytiskne 1 a 7
14
15 int avg = computeAvg(x + 1, y - 1);
16 // vytiskne 2 a 6
17
18 System.out.println("x: " + x + " y: " + y);
19 //vytiskne 1 a 7, volanim se hodnota x a y nezmeni
20
21 System.out.println("Avg: " + avg);
22 //vytiskne 4

```

lec04/DemoFunctionAvg.java

Příklady volání hodnotou 2/3

- Předáním referenční proměnné typu pole lze realizovat volání odkazem
Platí pro libovolnou referenční proměnnou
- Vytváří se kopie referenční proměnné což je odkaz, kde je pole uloženo, lokální proměnná values tak odkazuje na stejné místo v paměti a dochází tak k modifikaci obsahu paměti kam odkazuje hodnota „původní“ proměnné předávané funkci

```

1 void doSquare(int[] values) {
2     for (int i = 0; i < values.length; ++i) {
3         values[i] = values[i] * values[i];
4     }
5 }
6
7 ...
8 int array[] = {1, 2, 3, 4, 5}
9
10 print(array);
11 //vytiskne 1 2 3 4 5
12
13 doSquare(array);
14
15 print(array);
16 //vytiskne 1 4 9 16 25

```

lec04/DemoFunctionArray.java

Shrnutí přednášky

Příklady volání hodnotou 3/3

- Referenční proměnná typu String se také předává odkazem, ale hodnotu řetězce nelze měnit, vytváří se řetězec nový.
Tzv. „immutable object“

```

1 String addPrefix(String prefix, String str) {
2     str = prefix + " " + str;
3     return str;
4 }
5
6 ...
7 String str = "CTU in Prague";
8
9 System.out.println(str);
10 //vytiskne 'CTU in Prague'
11
12 String full = addPrefix("FEE", str);
13
14 System.out.println(str);
15 //vytiskne 'CTU in Prague'
16
17 System.out.println(full);
18 //vytiskne 'FEE CTU in Prague'

```

lec04/DemoFunctionString.java

- Další příklady viz příklad součtu matic nebo načítání řetězců
lec04/DemoArrayOfArray.java nebo lec04/DemoArrayString.java

Diskutovaná témata

- Číselné typy, jejich reprezentace a přetypování
- Pole – deklarace a alokace paměti
- Funkce a procedury – základní deklarace a volání hodnotou
- **Příště: Dekompozice problému**