- Premenné typu pole sú pointery (ich hodnoty sú adresy polí)
- Funkcie
  - Hlavička funkcie (Signatúra, funkčný prototyp v C)
  - Telo funkcie
  - Volanie funkcie
  - Formálne vs. aktuálne parametre
- Procedúry (Návratový typ: `void`)

Bubble Sort (druhý pokus)

```
public class BubbleSort
{ public static void sort (int[] a)
  { boolean sorted;
    do
    { sorted = true;
      for ( int i = 0; i < a.length - 1; i++ )
      { if ( a[i] > a[i+1] )
        { int h  = a[i];
          a[i]   = a[i+1];
          a[i+1] = h;
          sorted = false;
    } } }
    while ( !sorted );     }
```

Bubble Sort (druhý pokus)

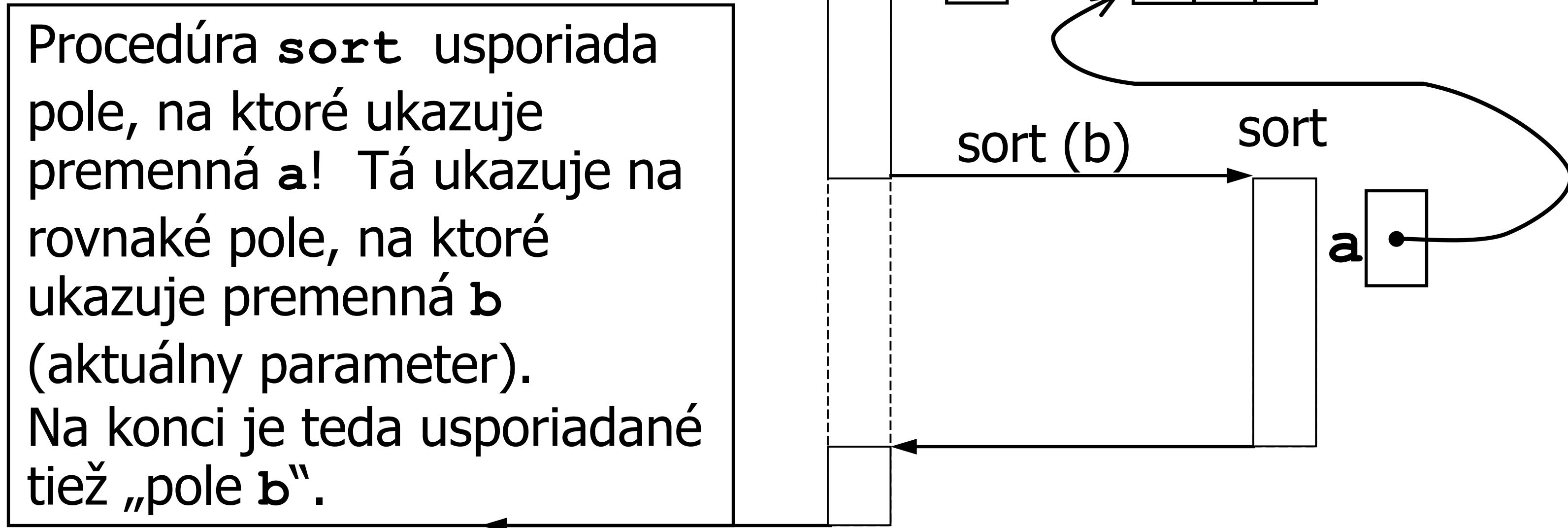
```
public static void main (String[] args)
{ int[] b = new int[args.length];
  for ( int i = 0; i < args.length; i++ )
  { b[i] = Integer.parseInt(args[i]); }

  sort(b);

  System.out.println(„Zoradené pole:");
  for ( int i = 0; i < b.length; i++ )
  { System.out.println(b[i]); }
}
}
```

Bubble Sort (druhý pokus)

```
>BubbleSort 7 1 2
```

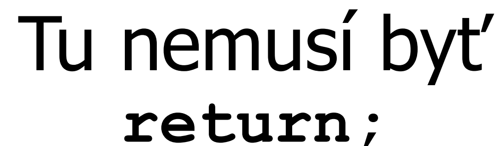


Polia ako parametre funkcií

- Parameter typu pole (prvkov nejakého typu) je vlastne adresa poľa.
- Do parametra typu pole je teda uložená adresa poľa; t.j. pole nie je skopírované (funkcia pracuje s originálnym poľom).
- Ak teda procedúra (alebo všeobecne funkcia) modifikuje prvky poľa, je modifikované pôvodné pole

Pozorovanie:

```
public static void sort (int[] a)
{ boolean sorted;
  do
  { sorted = true;
    for ( int i = 0; i < a.length - 1; i++ )
    { if ( a[i] > a[i+1] )
      { int h  = a[i];
        a[i]   = a[i+1];
        a[i+1] = h;
        sorted = false;
  } } }
  while ( !sorted );
}
```



Príkaz `return` v procedúrach

- Na konci tela procedúry nemusí byť príkaz **`return`**; procedúra je na konci automaticky ukončená.
  (V prípade funkcií s neprázdnym návratovým typom toto nie je možné, keďže takéto funkcie musia vrátiť nejakú hodnotu)

- V procedúrach tiež môžeme použiť príkaz **`return`** . V takom prípade za ním nenasleduje žiaden výraz.

8.5 Volanie hodnotou a volanie adresou

- Pri volaní funkcie je vždy kopírovaná do formálneho parametra hodnota. Ak je formálny dátový typ primitívny, tak takéto volanie nazývame volanie hodnotou (call by value);
- Pri neprimitívnych dátových typoch, napr. pri premenných typu pole, čo sú vlastne premenné, ktorých hodnotou je adresa objektu (napr. poľa) hovoríme o volaní adresou (call by name)
- Ak pri volaní hodnotou zmeníme formálny parameter, hodnota premennej použitej ako aktuálny parameter sa nezmení. Ak pri volaní adresou zmeníme hodnotu objektu, na ktorý ukazuje formálny parameter, zmeníme zároveň hodnotu objektu, na ktorý ukazuje aktuálny parameter. Je to logické, keďže sme odovzdali adresu a teda formálny aj aktuálny parameter ukazujú na to isté.

Volanie adresou a postranné efekty

- Pomocou volania adresou môže funkcia alebo procedúra „vrátiť“ nové hodnoty (t.j. zmeniť hodnoty) (pozri. Bubble Sort)!
- Pomocou volania hodnotou to nie je možné.

- V jazyku **C/C++** je možné pomocou operátora adresy & a operátora hodnoty na adrese * definovať, či má byť použité volanie adresou aj pre primitívne dátové typy: Príklad

```
#include <stdio.h>
void funkcia(int *pi) {*pi = 3;}
int main()
{       int i;
        funkcia(&i);
        printf("i = %d", i);
}
```

V **Jave** je volanie adresou možné iba pre neprimitívne dátové typy.

8.6 Globálne premenné

- Doposiaľ: Deklarácia premenných iba vo funkciách
  - **lokálne premenné** resp.
  - **parametre funkcií**

- Teraz: Deklarácia premenných mimo funkcie:
  globálne premenné

Príklad: globálne premenné

```
public class Binomialkoeficient
{ public static int pocitadlo = 0;

  public static int fak(int n)
  { int vysledok = 1;
    for (int i = 1; i <= n; i++)
    { vysledok = vysledok * i; }

    pocitadlo = pocitadlo + 1;
    return vysledok;
  }
```

Príklad: globálne premenné

```
public static void main (String[] args)
{ int n = Integer.parseInt(args[0]);
  int k = Integer.parseInt(args[1]);

  int vysledok = fak(n) / ( fak(k) * fak(n - k) );

  System.out.println("Vysledok: " + vysledok);

  System.out.print("Funkcia fak bola ");
  System.out.println(pociatadlo + " krat zavolana");
}
}
```

Globálne premenné

- Globálna premenná môže byť použitá v každej funkcii programu.
- Jej zmena v jednej funkcii je viditeľná v ostatných funkciách.
- Pomocou globálnych premenných si môžu funkcie jednoducho odovzdávať hodnoty.
- Pri deklarácii je globálnej premennej priradená automaticky preddefinovaná hodnota.

„Protipríklad“:

```
public class BubbleSort
{ public static int[] a;

  public static void sort() // Ziadny parameter
  { boolean sorted;         // Usporiada VZDY
    do { sorted = true;     // pole a!
         for ( int i = 0; i < a.length - 1; i++ )
         { if ( a[i] > a[i+1] )
           { int h  = a[i];
             a[i]   = a[i+1];
             a[i+1] = h;
             sorted = false; } }
    } while ( !sorted ); }
```

Globálne premenné: Odporúčania

- Vrátenie hodnoty pomocou globálnej premennej sa neodporúča.
- Zmysluplné použitie globálnej premennej je napr.:
  - Počítadlo
  - Debug-Modus (ďalšia Fólia)

Príklad: Debug-Modus

```
public class Binomialkoeficient    false;
{ public static boolean debug = true;

  public static int fak(int n)
  { int vysledok = 1;
    if (debug)
      System.out.println("Pred: n = " + n);
    for (int i = 1; i <= n; i++)
    { if (debug)
        System.out.println("Vnutri: i = " + i);
      vysledok = vysledok * i; }
```

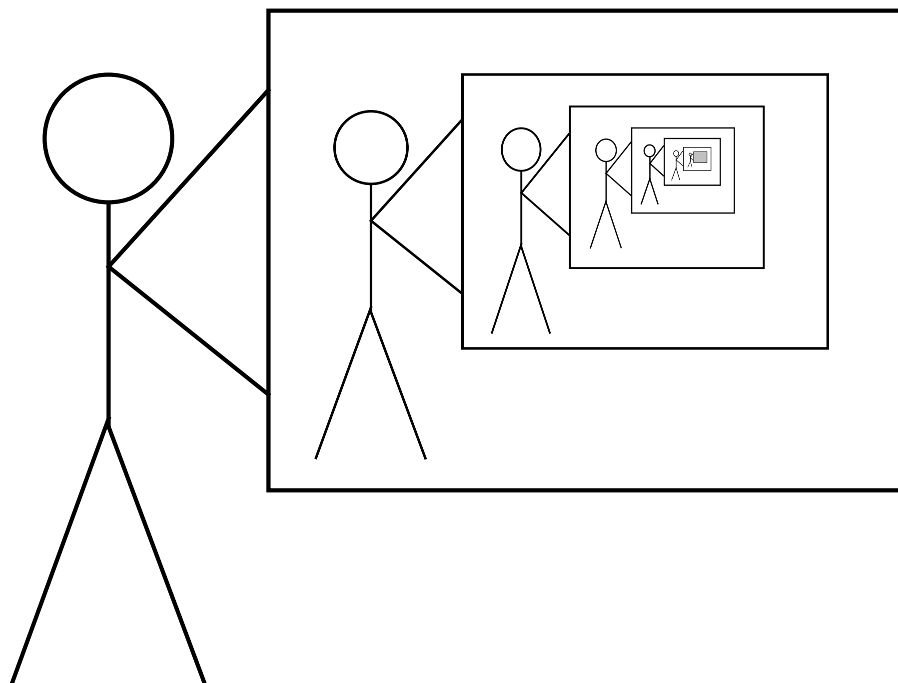
Platnosť

- Vo funkcii môže byť deklarovaná lokálna premenná s rovnakým názvom ako globálna premenná.
  V takomto prípade globálna premenná zatienená je lokálnou premennou.

8.7 Polymorphismus

- Je tiež možné definovať funkcie s rovnakým menom a s rozdielnym počtom a typom parametrov.
- Ktorá funkcia má byť zavolaná je dané typom a počtom aktuálnych parametrov.

8.8 Rekurzia



Motivácia

- Môže byť v tele funkcie použité volanie tej istej funkcie?
- Áno, je to možné!
- Áno, je to dokonca zmysluplné!

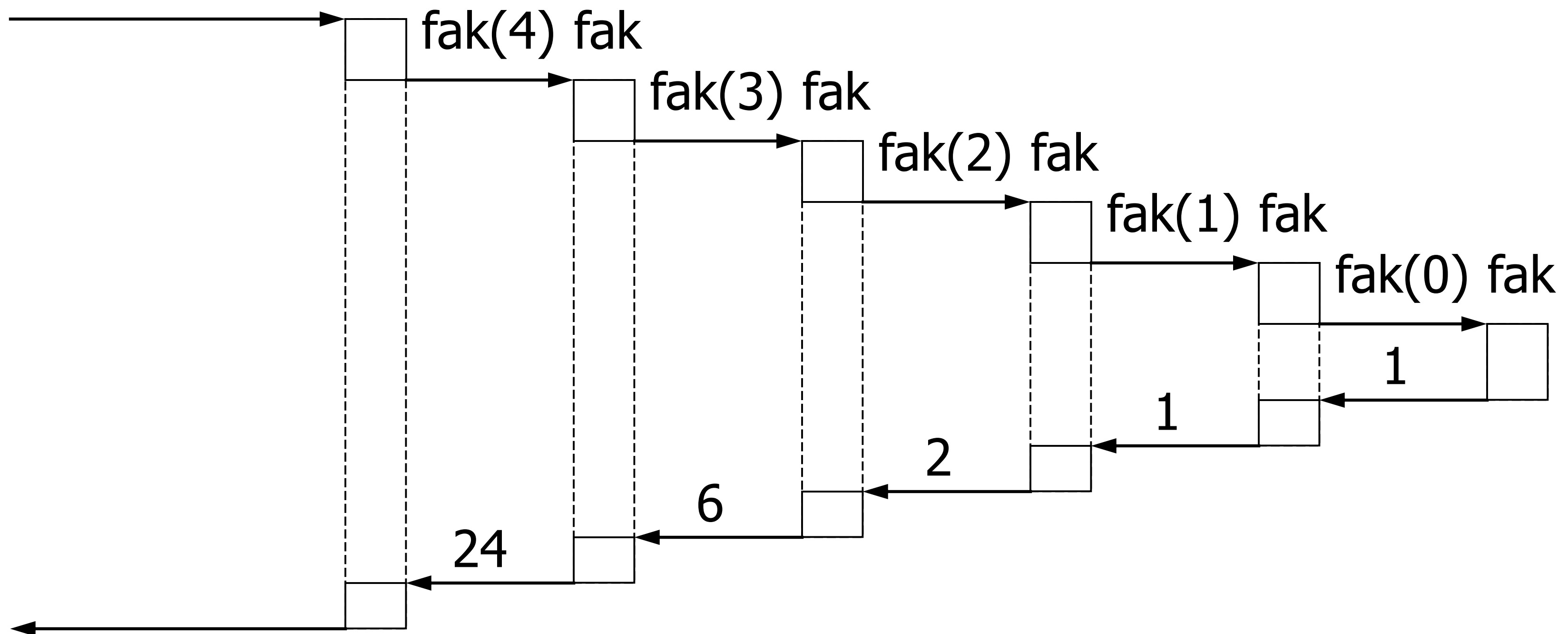
Príklad: Faktoriál

```
public class Faktorial
{
  public static int fak(int n)
  { if (n == 0)
    { return 1; }
    else
    { return n * fak(n-1); }
  }

  public static void main (String[] args)
  { int n = Integer.parseInt(args[0]);
    System.out.println(„Faktorial je: " + fak(n));
  }
}
```

Príklad: Faktoriál

`>Faktorial 4`



Princíp rekurzie

- Previesť problém na riešenie rovnakého problému s jednoduchšími dátami:

```
fak(n) -> fak(n-1).
```

- Kombinovať riešenie problémov pre jednoduchšie dáta na získanie riešenia problému s pôvodnými dátami:

```
fak(n) = n * fak(n-1).
```

- Pri volaní funkcie sa ukladajú hodnoty lokálnych premenných a miesto, z ktorého bola funkcia zavolaná.
- Po ukončení volanej funkcie pokračuje program vo vykonávaní volajúcej funkcie s uloženými hodnotami príkazom nasledujúcim za volajúcim miestom ďalej.

- Všetky hodnoty sú uložené s Stacku...

fak(0): n = 0
fak(1): n = 1, h = .
fak(2): n = 2, h = .
fak(3): n = 3, h = .
fak(4): n = 4, h = .

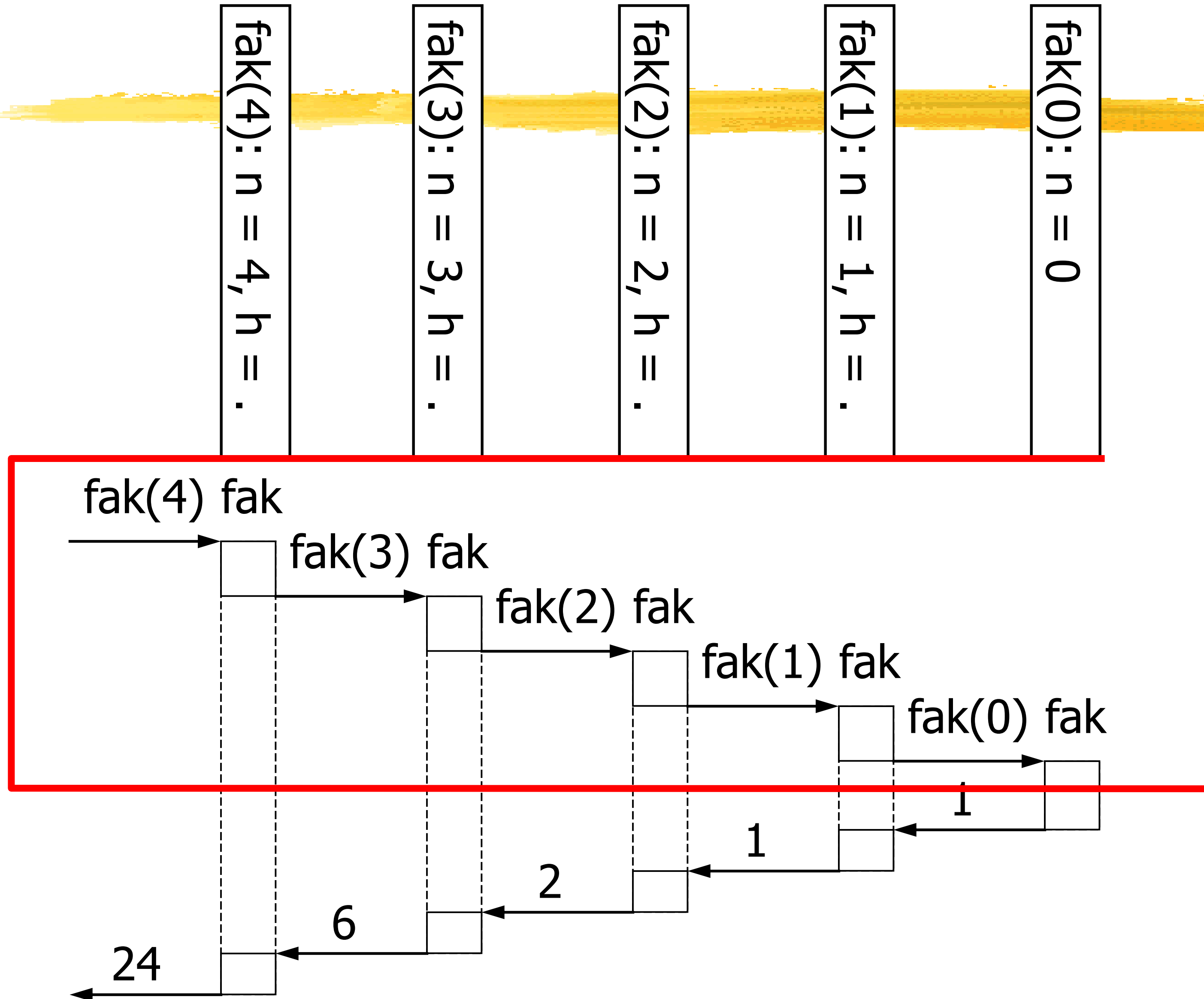
- A po ukončení zavolanej funkcie je starý stav obnovený...

```
return 1;
return n*h;
return n*h;
return n*h;
return n*h;
```

fak(4): n = 4, h = 6

24

Stack a volania



Rekurzívne funkcie

Funkcia sa nazýva rekurzívna, ak sama priamo alebo nepriamo (prostredníctvom volania iných funckií) je volaná vo svojej definícii.

- Často je možné rekurzívnymi funkciami fomulovať elegantné riešenie problému!
- Zvyčajne je rekurzívna funkcia menej efektívna ako zodpovedajúci `while`-cyklus

Nepriame volanie:

```
public static boolean even(int n)
{ // predpoklad: n >= 0
  if (n == 0)         return true;
  else if (n == 1)    return false;
  else                return odd(n-1);
}

public static boolean odd(int n)
{ // predpoklad: n >= 0
  if (n == 0)         return false;
  else if (n == 1)    return true;
  else                return even(n-1);
}
```

Terminovanie

- Musíme dávať pozor, aby rekurzívne volanie niekedy problém vyriešilo.
- V definícii rekurzívnej funkcie musí byť preto aspoň jedna alternatíva, ktorá rieši problém priamo, teda bez rekurzívneho volania.

Verifikácia rekurzívnych funkcií

Indukcia (Princíp):

1. **Počiatočný krok (PK)**:
   Dokáž správnosť pre n = 0
2. **Indukčný predpoklad (IP)**:
   Predpokladajme správnosť pre n – 1 >= 0
3. **Indukčný krok (IK)**:
   Pod podmienkou ak platí IP dokážeme, že výrok platí pre n

Tým je dokázaná správnosť pre všetky n >= 0.

Príklad: `fak(n) = n!`

1. **Počiatočný krok:**
   Zjavne platí **`fak(0)`** vracia 1 = **`0!`**
2. **Indukčný predpoklad**:
   Predpokladajme, že **`fak(n-1)` `pre`** n-1 >= 0 vracia hodnotu **`(n-1)!`**
3. **Indukčný krok**:
   Volanie **`fak(n)`** pre n > 0 zavolá **`fak(n-1)`**. Podľa (IP) platí, že volanie **`fak(n-1)`** vráti naspäť hodnotu **`(n-1)!`**, teda hodnotu **`h = (n-1)!`**
   Volanie **`fak(n)`** vracia **`return n * h`**, t.j. hodnotu **`n * (n-1)!`** ; a teda hodnotu **`n!`**

Príklad: Merge Sort

```
public static int[] merge(int[] a, int[] b)
{ int posa = 0, posb = 0, pos  = 0;
  int[] vysledok = new int[a.length+b.length];

  while ( posa < a.length && posb < b.length )
  { if (a[posa] <= b[posb])
    { vysledok[pos] = a[posa];
      posa = posa + 1;
      pos  = pos + 1; } else
    { vysledok[pos] = b[posb];
      posb = posb + 1;
      pos  = pos + 1; }
  }
```

Príklad: Merge Sort

```
while ( posa < a.length )
{ vysledok[pos] = a[posa];
  posa = posa + 1;
  pos  = pos + 1;
}

while ( posb < b.length )
{ vysledok[pos] = b[posb];
  posb = posb + 1;
  pos  = pos + 1;
}
return vysledok;                           }
```

Príklad: Merge Sort

```
public static int[] sort(int[] a)
{ if (a.length > 1)
  { int[] a1 = new int[a.length/2];
    int[] a2 = new int[a.length - a1.length];

    System.arraycopy(a, 0, a1, 0, a1.length);
    System.arraycopy(a, a1.length, a2, 0,
                                       a2.length);
    return merge(sort(a1),sort(a2));
  }
  else return a;
}
```

Poznámka

- Predchádzajúca verzia MergeSort-Algoritmu nie je najefektívnejšia (je ale ľahšie pochopiteľná).
- V praxi sa používa na triedenie QuickSort-Algoritmus; napr. v preddefinovaných funkciách v Jave

Príklad: QuickSort

```
public static void sort(int[] a, int u, int o)
{ if (u < o)
  { int o1 = o;
    int u1 = u;

    int x = a[(u + o) / 2];
    while (u1 <= o1)
    { while ( x < a[o1] )
      { o1 = o1 - 1; }

      while ( a[u1] < x )
      { u1 = u1 + 1; }
```

Príklad: QuickSort

```
      if (u1 <= o1)
      { int h = a[o1];
        a[o1] = a[u1];
        a[u1] = h;
        o1 = o1 - 1;
        u1 = u1 + 1;
    } }

    if (u <= o1)
    { sort(a,u,o1); }
    if (u1 <= o)
    { sort(a,u1,o); }
} }
```

Príklad: QuickSort

```
public static void main (String[] args)
{ int[] a = new int[args.length];
  for ( int i = 0; i < args.length; i++ )
  { a[i] = Integer.parseInt(args[i]); }

  sort(a,0,a.length-1);

  System.out.println(„Vysledok:");
  for ( int i = 0; i < a.length; i++ )
  { System.out.print(a[i]+" "); }
}
```

I. Java - Základné elementy 9. Zhrnutie

- **Procedurálne programovanie**
  - Výrazy: Syntaktický strom, Vyhodnotenie
  - Dátové typy: Premenné, Polymorphismus
  - Priradenie
  - Riadiace štruktúry: Bloky, Alternatívy, Cykly
  - Dátové štruktúry: Polia
  - Adresové premené (Pointery, Smerníky, Ukazovatele)
  - Funkcie a procedúry (Rekurzia)
- Verifikácia: Kontrolné podmienky, Invarianty cyklov, Varianty cyklov, Indukcia