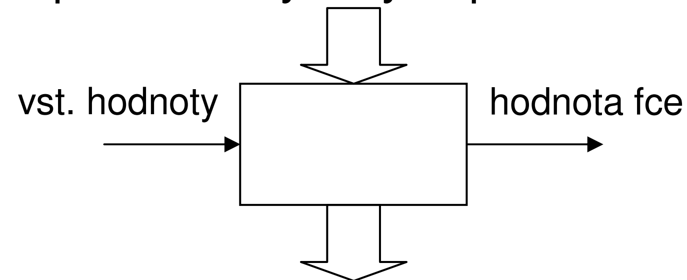
4. přednáška - funkce a procedury, složitost

- Obsah přednášky:
  - rozdělení problému na podproblémy, funkce, procedury, intuitivní chápání složitosti, vliv kvality algoritmu na celkovou dobu běhu programu.
  - Podpůrné prostředky z Javy:

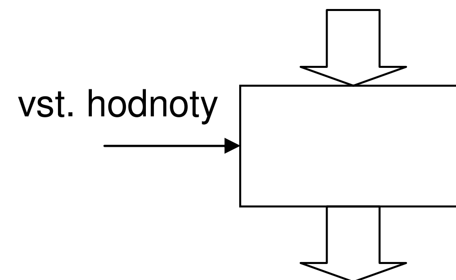
    metody - procedury a funkce

Funkce a procedury

- **Příklad**
  - výpočet sin, cos, výpočet faktoriálu, …
  - vstupy, výstupy - Systém.out.print (“vysledek”); Math.random();
- **Funkce**
  - definovány vstupní hodnoty a výstupní hodnota funkce



- **Procedura**
  - definovány vstupní hodnoty a činnost procedury



Deklarace funkce

- **Deklaraci funkce** tvoří

  hlavička_funkce   tělo_funkce

- Hlavička funkce v jazyku Java má tvar

  ```
  static typ jméno(specifikace parametrů)
  ```

  kde

  - ***typ*** je typ výsledku funkce (funkční hodnoty)
  - ***jméno*** je identifikátor funkce
  - ***specifikací parametrů*** se deklarují parametry funkce,
    každá deklarace má tvar

    ```
    typ_parametru jméno_parametru
    ```

    a oddělují se čárkou
  - specifikace parametrů je prázdná, jde-li o funkci bez parametrů

- Tělo funkce je složený příkaz nebo blok, který se provede při volání funkce.
- Tělo funkce musí končit příkazem

  ```
  return x;
  ```

  kde *x* je výraz, jehož hodnota je výsledkem volání funkce.

Parametry funkce

- **Parametry funkce jsou lokální proměnné funkce**, kterým se při volání funkce přiřadí hodnoty skutečných parametrů.
- Jestliže parametr funkce je typu *T*, pak přípustným skutečným parametrem je výraz, jehož hodnotu lze přiřadit proměnné typu *T* (stejná podmínka, jako u přiřazení).
- Příklad:

formální parametry

```
public class Max1 {
   static int max( int x, int y ) {
      if (x>y) return x;
      else return y;
   }
   public static void main(String[] args) {
      int a = 10, b = 20;
      System.out.println( max(a+20, b) );     // O.K.
      System.out.println(max(1.1, b));  // Chyba
   }
}
```

skutečné parametry

Parametry funkce

- **Parametry funkce slouží pro předání vstupních dat algoritmu**, který je funkcí realizován.
- Častá chyba začátečníka: funkce, která čte hodnoty parametrů pomocí operace vstupu dat.

```
static int max(int x, int y) {
        x = sc.nextInt();       // nesmyslný příkaz
        y = sc.nextInt();       // nesmyslný příkaz
        if (x>y)
                return x;
        else
                return y;
}

int z= max(7,99);
```

Faktoriál pomocí funkcí (1)

- Připomeňme si program pro výpočet faktoriálu:

```
public class Faktorial {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("zadejte přirozené číslo");
      int n = sc.nextInt();
      if(n<1) {
         System.out.println(n + " není přirozené číslo");
         System.exit(0);
      }
      int i = 1;
      int f = 1;
      while(i<n) {
         i = i + 1;
         f = f * i;
      }
      System.out.println (n + "! = " + f);
   }
}
```

čtení přirozeného čísla (první blok)

algoritmus výpočtu faktoriálu (druhý blok)

Čtení přirozeného čísla a **výpočet** faktoriálu jsou dva **dílčí podproblémy**,

Faktoriál pomocí funkcí (2)

- **Funkce pro čtení přirozeného čísla**

```
static int ctiPrirozene() {
    Scanner sc = new Scanner(System.in);
    System.out.println ("zadejte přirozené číslo");
    int n = sc.nextInt();
    if (n<1) {
          System.out.println (n + " není přirozené číslo");
          System.exit(0);
    }
    return n;
}
```

- **Hlavička funkce**

```
    static int ctiPrirozene()
```

vyjadřuje, že funkce nemá vstupní parametry a že výsledkem volání funkce je hodnota typu *int*.

- **Příkaz**

```
    return n;
```

předepisuje návrat z funkce, výsledkem volání je hodnota *n*.

- **Příklad volání funkce**: `int cislo = ctiPrirozene();`

Faktoriál pomocí funkcí (3)

- **Funkce pro výpočet faktoriálu**

```
static int faktorial(int n) {
    int i = 1;
    int f = 1;
    while (i<n) {
        i = i+1;
        f = f * i;
    }
    return f;
}
```

Hlavička funkce vyjadřuje, že funkce má jeden **vstupní parametr *n* typu *int*** a že výsledkem je hodnota typu *int*.

- Příklad volání funkce

```
int ff = faktorial(4);
```

Faktoriál pomocí funkcí (4)

- **Výsledné řešení:**

```
package alg4;
import java.util.*;
public class Faktorial {
    static int ctiPrirozene() {
        ...
    }
    static int faktorial(int n) {
        ...
    }
    public static void main(String[] args) {
        int n = ctiPrirozene();
        System.out.println (n + "! = " + faktorial(n));
    }
}
```

Přetěžování jmen funkcí

- **Přetěžování jmen** (overloading of names) – jedna třída může obsahovat více metod stejného jména, které se liší v počtu nebo typu parametrů.

  POZOR: Ikdyž se stejně jmenují, jsou to jiné metody!!!

- Příklad:

```
public class Max2 {
   static int max(int x, int y) {
      if (x>y) return x; else return y;
   }
   static int max(int x, int y, int z) {
      return max(x, max(y, z));
   }
   static double max(double x, double y) {
      if (x>y) return x; else return y;
   }
   public static void main(String[] args) {
      System.out.println(max(3,4));
      System.out.println(max(1,2,3));
      System.out.println(max(1.0,2.4));
   }
}
```

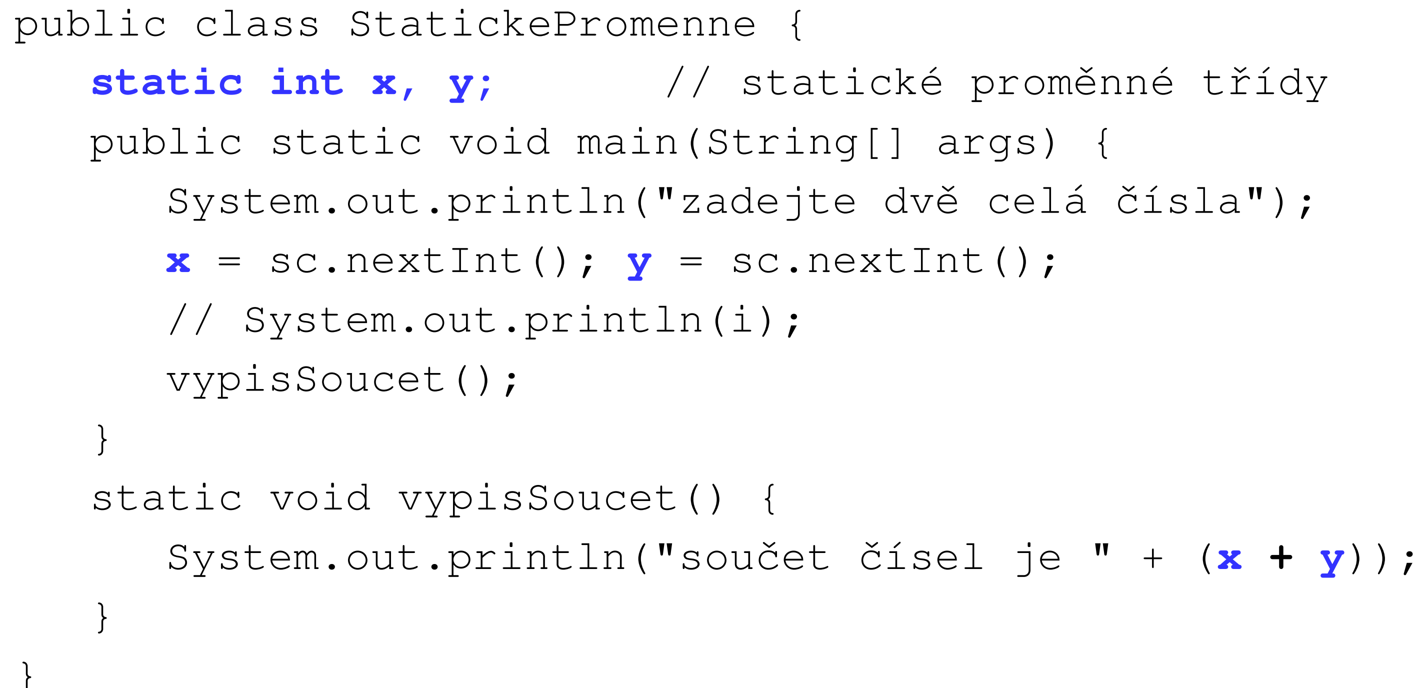
Procedury

- Metoda, jejíž typ výsledku je ***void***, nevrací žádnou hodnotu.
- Příklady procedur:
  - hlavní funkce *main*
  - výstupní operace *print* a *println*
- Příklad uživatelské procedury: výpis znaku *z* doplněného zleva mezerami na celkový počet *n* znaků

```
static void vypisZnak(char z, int n) {
   for (int i=1; i<n; i++) System.out.print(' ');
      System.out.print(z);
}
```

Statické proměnné

- Třída může obsahovat deklarace statických proměnných.
- Statické proměnné (proměnné třídy) jsou použitelné ve všech funkcích dané třídy – jsou to pro ně nelokální proměnné.
- Příklad:



```
public class StatickePromenne {
   static int x, y;        // statické proměnné třídy
   public static void main(String[] args) {
      System.out.println("zadejte dvě celá čísla");
      x = sc.nextInt(); y = sc.nextInt();
      // System.out.println(i);
      vypisSoucet();
   }
   static void vypisSoucet() {
      System.out.println("součet čísel je " + (x + y));
   }
}
```

- Poznámka: statické proměnné jsou inicializovány hodnotou nula (0, 0.0, ‘\u0000’, false)

Zastínění nelokální proměnné

- Deklarace lokální proměnné *p* zastíní deklaraci nelokální proměnné *p*.
- Příklad:

```
public class Zastineni {
   static int a = 10;
   public static void main(String[] args) {
      f();
      System.out.println(a);
   }
   static void f() {
      int a = 20;
      System.out.println(a);
   }
}
```

Zastineni.a

f.a

Co vypíše program?

Zastínění nelokální proměnné

- Deklarace lokální proměnné *p* zastíní deklaraci nelokální proměnné *p*.
- Příklad:

```
public class Zastineni {
   static int a = 10;
   public static void main(String[] args) {
      f();
      System.out.println(a);
   }
   static void f() {
      int a = 20;
      System.out.println(a);
   }
}
```

Zastineni.a

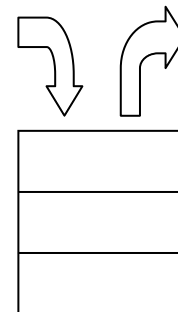
f.a

Program vypíše:

```
20
10
```

Přidělování paměti proměnným

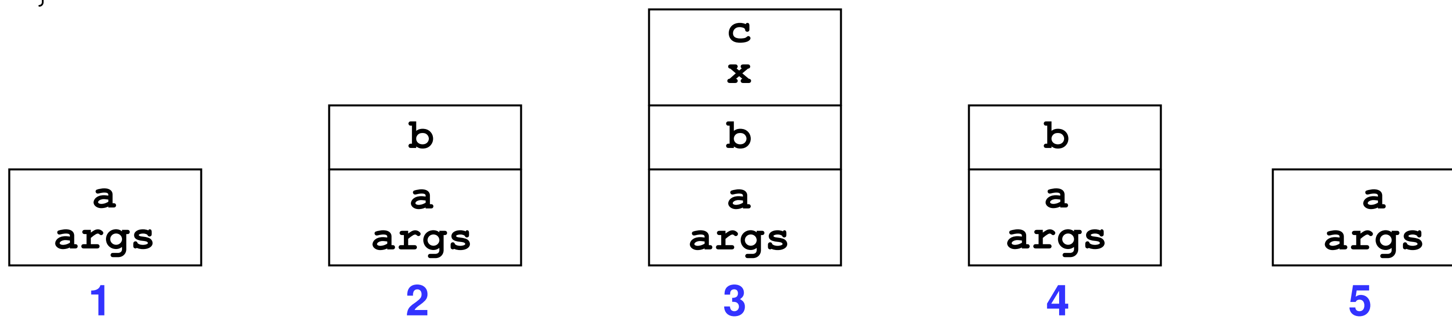
- Poznali jsme doposud dva druhy proměnných:
  - statické proměnné třídy
  - lokální proměnné funkcí
- **Přidělení paměti proměnné** - určení adresy umístění proměnné v paměti počítače.
- Statickým proměnným třídy se přidělí paměť v okamžiku, kdy se do paměti zavádí kód funkcí třídy, a zůstane jim přidělena až do ukončení běhu programu.
- Lokálním proměnným a parametrům funkce se paměť přidělí při volání funkce a zůstane jim přidělena jen do návratu z funkce (při návratu z funkce se přidělené adresy uvolní pro další použití).
- Úseky paměti přidělované lokálním proměnným a parametrům tvoří tzv. **zásobník** (stack): úseky se přidávají a odebírají, přičemž se vždy odebere naposledy přidaný úsek (**L**ast **I**n **F**irst **O**ut).



Přidělování paměti proměnným

- Příklad

```
public static void main(String[] args) {
   int a; ... //1
   f();       //5
   ...
}
static void f() {
   int b; ... //2
   g(10);     //4
   ...
}
static void g(int x) {
   int c; ... //3
   ...
}
```



Problém: prvočísla

- **Problém**: Zjistěte, zda zadané číslo *n* je prvočíslo.

  Prvočíslo - přirozené číslo větší než 1, které je beze zbytku dělitelné pouze jedničkou a samo sebou.

  Zkusíme se zaměřit na **efektivitu výpočtu**.

Příklad: prvočísla I, dle definice

- Přirozené číslo ***n*** je **prvočíslo**, právě tehdy když jej beze zbytku dělí pouze číslo ***n*** a číslo **1.**

```
public static boolean isPrvocislo1(long n){
   if (n < 2 ) return false;  // ostatní celá čísla nejsou
                              // prvočísla
   int pocetDelitelu = 0;
   for (int i=1; i<=n; i++) { // jsou potřeba ty závorky?
      if (n%i == 0)
         pocetDelitelu++;
   }
   if (pocetDelitelu == 2) return true;
      else return false;
}
```

Jak dlouho poběží tento algoritmus pro číslo 100 000 000 000 003?

A jak dlouho pro 100 000 000 000 000?

Příklad: prvočísla II

- Vylepšení `isPrvocislo1`: Najdeme-li prvního dělitele jiného než čísla **1** a **n,** již to není prvočíslo.

```
public static boolean isPrvocislo2(long n){
   if (n < 2 ) return false;
   for (int i=2; i<n; i++) {// vynecháme číslo 1 a číslo n
      if (n%i==0)
         return false;
   }
   return true;
}
```

Jak dlouho poběží tento algoritmus pro číslo 100 000 000 000 003?

A jak dlouho pro 100 000 000 000 000?

Příklad: prvočísla III

- Vylepšení `isPrvocislo2`: Číslo složené lze zapsat jako součin dělitelů, např. 35=7*5; jedno je „menší“ a druhé „větší“.

```
public static boolean isPrvocislo3(long n){
   if (n < 2 ) return false;
   for (int i=2; i<=(n/2) ; i++) {// procházíme jen do půlky
      if (n%i==0)
         return false;
   }
   return true;
}
```

Jak dlouho poběží tento algoritmus pro číslo 100 000 000 000 003?

A jak dlouho pro 100 000 000 000 000?

Příklad: prvočísla IV

- Vylepšení `isPrvocislo3`: Číslo složené lze napsat jako součin dělitelů, např. 35=7*5; jedno je „menší“ a druhé „větší“, pokud nejsou stejné jako např. 49 = 7*7.

```
public static boolean isPrvocislo(long n){
   if (n < 2 ) return false;
      for (long i=2; i<=Math.sqrt(n); i++) {// procházíme
                                            // jen do odmocniny n
         if (n%i==0)
            return false;
   }
   return true;
}
```

Jak dlouho poběží tento algoritmus pro číslo 100 000 000 000 003?

A jak dlouho pro 100 000 000 000 000?

Další příklady funkcí

- Funkce pro zjištění, zda daný rok je přestupný

```
public class Rok {
   static boolean prestupny(int rok) {
      if (rok%4==0 && (rok%100!=0 || rok%400==0))
         return true;
      else
         return false;
   }
   public static void main(String[] args) {
      int rok;
      Scanner sc = new Scanner(System.in);
      System.out.println ("zadejte rok");
      rok = sc.nextInt();
      System.out.print("rok "+rok);
      if (prestupny(rok))
         System.out.println (" je přestupný");
      else
         System.out.println (" není přestupný");
   }
}
```

Funkce pro výpočet NSD

- Jednoduchý algoritmus výpočtu největšího společného dělitele jsme již uvedli
  Efektivnější algoritmus lze sestavit na základě těchto vztahů:

  je-li $x = y$, pak $nsd(x, y) = x$

  je-li $x > y$, pak $nsd(x, y) = nsd(x-y, y)$

  je-li $x < y$, pak $nsd(x, y) = nsd(x, y-x)$

- Řešení 2:

```
static int nsd(int x, int y) {
   while (x!=y)
      if (x>y)
         x = x-y;
      else
         y = y-x;
      return x;
}
public static void main(String[] args) {
   int a=sc.nextInt(); int b=sc.nextInt();
   System.out.println("Nejvetsi spolecny delitel" +
                                 a + ", " + b + "je " + nsd(a,b));
}
```

Funkce pro výpočet NSD

- Do těla cyklu vnoříme místo podmíněného příkazu pro jediné zmenšení hodnoty *x* nebo *y* dva cykly pro opakované zmenšení hodnot *x* a *y*
- Řešení 3:

```
static int nsd(int x, int y) {
   while (x!=y) {
      while (x>y) x = x-y;
      while (y>x) y = y-x;
   }
   return x;
}
public static void main(String[] args) {
   int a=sc.nextInt();int b=sc.nextInt();
   System.out.println("Nejvetsi spolecny delitel" +
                                 a + ", " + b "je " + nsd(a,b));
}
```

Euklidův algoritmus pro výpočet NSD

- Vnitřní cykly řešení 3 počítají nenulový zbytek po dělení většího čísla menším.
- Pro výpočet zbytku po dělení máme operaci %
- Jejím využitím dostaneme Euklidův algoritmus, který lze slovně formulovat takto: určíme zbytek po dělení daných čísel, zbytkem dělíme dělitele a určíme nový zbytek, až dosáhneme nulového zbytku; poslední nenulový zbytek je nsd
- Řešení 4:

```
static int nsd(int x, int y) {
   int zbytek = x%y;
   while (zbytek!=0) {
      x = y; y = zbytek; zbytek = x%y;
   }
   return y;
}
public static void main(String[] args) {
   int a = sc.nextInt(); int b = sc.nextInt();
   System.out.println("Nejvetsi spolecny delitel" +
                             a + ", " + b "je " + nsd(a,b));
}
```