Funkce a procedury

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 5

A0B36PR1 – Programování 1

Část 1 – Funkce, lokální proměnné a přidělování paměti

Část 2 – Cykly a řízení jejich průchodu

Část 3 – Příklady

Část I

Funkce, lokální proměnné a přidělování paměti

Přetížení jména funkce

- Funkce lišící se v počtu nebo typu parametrů se mohou jmenovat stejně
- Přetěžování jmen (overloadings of names)

```
int max(int x, int y) {
   return x > y ? x : y;
}

int max(int x, int y, int z) {
   return max(x, max(y, z));
}

double max(double x, double y) {
   return x > y ? x : y;
}

void execute() {
   System.out.println(max(3, 5));
   System.out.println(max(4, 7, 5));
   System.out.println(max(1.4, 4.3));
}
```

lec05/DemoFunctionNameOverloading.java

Předání návratové hodnoty funkce – **return** 1/2

- Předání hodnoty volání funkce je předepsáno voláním **return**

```
int doSomeThingUsefull() {
   int ret = -1;
   ...
   return ret;
}
```

- Jak často umisťovat volání **return** ve funkci?

```
int doSomething() {
   if (
         !cond1
       && cond2
       && cond3
       ) {
      ... do some long code ...
   }
   return 0;
}
```

```
int doSomething() {
   if (cond1) {
      return 0;
   }
   if (!cond2) {
      return 0;
   }
   if (!cond3) {
      return 0;
   }
   ... some long code ....
   return 0;
}
```

http://llvm.org/docs/CodingStandards.html

Předání návratové hodnoty funkce – **return** 2/2

- Volání **return** na začátku funkce může být přehlednější

  Podle hodnoty podmínky je volání funkce ukončeno
- Kódovací konvence může také předepisovat použití nejvýše jednoho volání **return**

  Má výhodu v jednoznačné identifikací místa volání, můžeme jednoduše pak například přidat další zpracování výstupní hodnoty funkce.
- Dále není doporučováno bezprostředně používat **else** za voláním **return** (nebo jiným přerušení toku programu), např.

```
case 10:
   if (...) {
      ...
      return 1;
   } else {
      if (cond) {
         ...
         return -1;
      } else {
         break;
      }
   }
```

```
case 10:
   if (...) {
      ...
      return 1;
   } else {
      if (cond) {
         ...
         return -1;
      }
   }
   break;
```

Rozsah platnosti lokální proměnné

- Lokální proměnné mají rozsah platnosti pouze uvnitř bloku nebo funkci

```
int a = 10; //lokalni promenna

if (a == 10) {
   int a = 1;  // v Jave nelze zastinit lokalni
               // promennou jinou lokalni
               // promennou riziko chyby
               //
   int b = 20; // lokalni promena s platnosti
               // uvnitr bloku
   a = b + 10; //
}

b = 10; // b neni platnou promennou
```

- „Globální" proměnné, které mají rozsah platnosti „kdekoliv" v programu můžeme v Javě realizovat jako třídní proměnné, případně jako atributy objektu.

K tomu však potřebujeme znát základy objektově orientovaného programování

Přidělování paměti proměnným

- Přidělením paměti proměnné rozumíme určení adresy umístěné proměnné v paměti počítače
- Lokálním proměnným a parametrům funkce
  - se paměť přiděluje při volání funkce.
  - Paměť zůstane přidělena jen do návratu z funkce.

    Při návratu funkce se přidělené paměťové místo uvolní pro další použití
  - Paměť se alokuje z rezervovaného místa – **zásobník (stack)**
- Dynamické přidělování paměti
  - Alokace paměti řetězce nebo pole se provádí příkazem **new**

    Alokace objektů
  - Paměť se alokuje z rezervovaného místa – **halda (heap)**

Alokace paměti

- Java program běží ve virtuálním stroji JVM (Java Virtual Machine)
- Při spuštění JVM specifikujeme kolik fyzické paměti dedikujeme virtuálnímu stroji

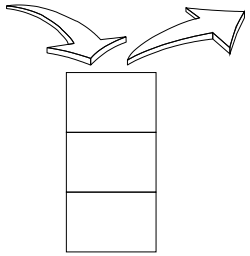
Podobně funguje fyzický limit na úrovni OS

- Paměť je rozdělena na zásobník (stack) a haldu (heap)
- Paměť přiděluje správce paměti
- Dynamicky přidělovanou paměť (voláním `new`) v Javě spravuje a automaticky uvolňuje *Garbage Collector*

Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům funkce tvoří tzv. **zásobník** (stack)
- Úseky se přidávají a odebírají
  - Vždy se odebere naposledy přidaný úsek

    LIFO – last in, first out



- Na zásobník se také ukládá „volání funkce"

Na zásobník se také ukládá návratová hodnota funkce a také hodnota „program counter" původně prováděné instrukce, před voláním funkce

- Alokují se tam proměnné parametrů funkce

Argumenty (parametry) jsou de facto lokální proměnné

Opakovaným rekurzivním voláním funkce můžeme velice rychle zaplnit velikost přiděleného zásobníku

Příklad rekurzivního volání funkce

```
void printValue(int v) {
   System.out.println("value: " + v);
   printValue(v + 1);
}

demo.printValue(1);
```

lec05/DemoStackOverflow.java

- Vyzkoušejte si program pro různé hodnoty přidělené paměti pro zásobník

```
java DemoStackOverflow 2>err | tail -n 2
value: 9918
value: 9919

java -Xss10m DemoStackOverflow 2>err | tail -n 2
value: 137803
value: 137804
```


Část II

Cykly a řízení jejich průchodu

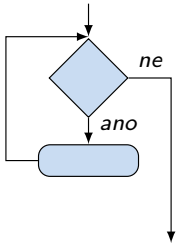
Cykly

- Cyklus **for** a **while** testuje podmínku opakování před vstupem do těla cyklu
  - **for** – inicializace, podmínka a změna řídicí proměnné součást syntaxe

```
for (int i = 0; i < 5; i++) {
   ...
}
```

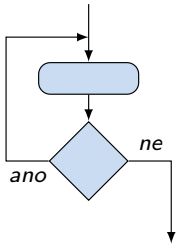
  - **while** – řídicí proměnná v režii programátora

```
int i = 0;
while (i < 5) {
   ...
   i += 1;
}
```



- Cyklus **do** testuje podmínku opakování cyklu po prvním provedení cyklu

```
int i = -1;
do {
   ...
   i += 1;
} while (i < 5);
```



Ekvivalentní provedení 5ti cyklů.

Předčasné ukončení průchodu cyklu – příkaz **continue**

- Někdy může být užitečné ukončit cyklus v nějakém místě uvnitř těla cyklu
  - Například ve vnořených **if** příkazech
- Příkaz **continue** předepisuje ukončení průchodu těla cyklu

Platnost pouze v těle cyklu!

```
for (int i = 0; i < 10; ++i) {
   System.out.print("i:" + i + " ");
   if (i % 3 != 0) {
      continue;
   }
   System.out.println(" ");
}
```

```
javac DemoContinue.java
java DemoContinue
i:0
i:1 i:2 i:3
i:4 i:5 i:6
i:7 i:8 i:9
```

lec05/DemoContinue.java

Předčasné ukončení vykonávání cyklu – příkaz **break**

- Příkaz **break** předepisuje ukončení cyklu

Program pokračuje následujícím příkazem po cyklu

```
for (int i = 0; i < 10; ++i) {
   System.out.print("i:" + i + " ");
   if (i % 3 != 0) {
      continue;
   }
   System.out.println(" ");
   if (i > 5) {
      break;
   }
}
```

```
javac DemoBreak.java
java DemoBreak
i:0
i:1 i:2 i:3
i:4 i:5 i:6
```

lec05/DemoBreak.java

Příklad – ukončení cyklu lineárního vyhledávání

```
String[] months = { "jan", "feb", "mar", "apr", "may", "
    jun", "jul", "aug", "sep", "oct", "nov", "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 '" + monthStr + "' is
    " + month + " month of the year");
} else {
   System.err.println("Month has not been matched");
}
```

lec04/DemoArrayMonth.java lec05/DemoArrayMonth.java

- Při nalezení odpovídajícího řetězce je zbytečné testovat další prvky pole.

Konečnost cyklů 1/3

- Konečnost algoritmu – pro přípustná data v konečné době skončí

2. přednáška

- Aby byl algoritmus **konečný** musí každý cyklus v něm uvedený skončit po konečném počtu kroků
- Jedním z důvodu neukončení programu je zacyklení
  - Program opakovaně vykoná cyklus, jehož podmínka ukončení není nikdy splněna.

```
while (i != 0) {
   j = i - 1;
}
```

- Cyklus se provede jednou,
- nebo neskončí.
- Záleží na hodnotě *i* před voláním cyklu

Konečnost cyklů 2/3

- Základní pravidlo pro konečnost cyklu
  - Provedením těla cyklu se musí změnit hodnota proměnné použité v podmínce ukončení cyklu

```
for (int i = 0; i < 5; i++) {
   ...
}
```

- Uvedené pravidlo konečnost cyklu nezaručuje

```
int i = -1;

while( i < 0 ) {
   i = i - 1;
}
```

Konečnost cyklu závisí na hodnotě proměnné před vstupem do cyklu.

Konečnost cyklů 3/3

```
while (i != n) {
   ... //prikazy nemenici hodnotu promenne i
   i++;
}
```

- Vstupní podmínka konečnosti uvedeného cyklu
  - $i \leq n$ pro celá čísla

Jak by vypadala podmínka pro proměnné typu `double`?

lec05/DemoLoop.java

- Splnění vstupní podmínky konečnosti cyklu musí zajistit příkazy předcházející příkazu cyklu
- Zabezpečený program testuje přípustnost vstupních dat

Příklad – test, je-li zadané číslo prvočíslem

```
public boolean isPrimeNumber(int n) {
   boolean ret = true;
   for (int i = 2; i <= (int)Math.sqrt(n); ++i) {
      if (n % i == 0) {
         ret = false;
         break;
      }
   }
   return ret;
}
```

lec05/DemoPrimeNumber.java

- Po nalezení prvního dělitele je zbytečné pokračovat ve zkoušení dalších hodnot
- Hodnota výrazu `(int)Math.sqrt(n)` se v cyklu nemění a je zbytečné výraz opakovaně vyhodnocovat

```
boolean ret = true;
final int MAX_BOUND = (int)Math.sqrt(n);
for (int i = 2; i <= MAX_BOUND ; ++i) {
   ...
}
```

V JVM bude výraz pravděpodobně vyhodnocen pouze jednou v důsledku optimalizace kódu za běhu Java HotSpot Virtual Machine

Vnořené cykly

- **break** ukončuje vnitřní cyklus

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      System.out.println("i-j: " + i + "-" + j);
      if (j == 1) {
         break;
      }
   }
}
```

```
i-j: 0-0
i-j: 0-1
i-j: 1-0
i-j: 1-1
i-j: 2-0
i-j: 2-1
```

- Vnější cyklu můžeme ukončit příkazem break se jménem

```
outer:
for (int i = 0; i < 5; i++) {
   for (int j = 0; j < 3; j++) {
      System.out.println("i-j: " + i + "-" + j);
      if (j == 2) {
         break outer;
      }
   }
}
```

```
i-j: 0-0
i-j: 0-1
i-j: 0-2
```

lec05/DemoLabeledBreak.java

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

Kódovací konvence

- Příkazy **break** a **continue** v podstatě odpovídají příkazům skoku.
- Obecně můžeme říci, že příkazy **break** a **continue** nepřidávají příliš na přehlednosti

Nemyslíme tím `break` v příkazu `switch`

- Přerušení cyklu **break** nebo **continue** můžeme využít těle dlouhých funkcí a vnořených cyklech

Ale funkce bychom měli psát krátké a přehledné

- Je-li funkce (tělo cyklu) krátké je význam **break**/**continue** čitelný
- Podobně použití na začátku bloku cyklu např. jako součást testování splnění předpokladů, je zpravidla přehledné
- Použití uprostřed bloku je však už méně přehledné a může snížit čitelnost a porozumění kódu

https://www.scribd.com/doc/38873257/Knuth-1974-Structured-Programming-With-Go-to-Statements

Část III

Příklady

Příklad – Výpočet největšího společného dělitele 1/3

- **Vstup** - celá čísla x a y
- **Výstup** - největší společný dělitel x a y, tj. celé číslo d takové, že $x\%d == 0$ a $y\%d == 0$ a zároveň d je maximální možné.
- Základní varianta

```
int getGreatestCommonDivisor(int x, int y) {
   int d = x < y ? x : y;
   while ( (x % d != 0) || (y % d != 0) ) {
      d = d - 1;
   }
   return d;
}
```

lec03/DemoGCD.java

Příklad – Výpočet největšího společného dělitele 2/3

- Místo výběru menšího z čísel x a y vnoříme do těla hlavního cyklu dva cykly zmenšující hodnoty aktuální hodnoty x a y

```
int getGreatestCommonDivisorLoops(int x, int y) {
   while (x != y) {
      while (x > y) {
         x = x - y;
      }
      while (y > x) {
         y = y - x;
      }
   }
   return x;
}
```

lec05/DemoGCD.java

- Vnitřní cykly počítají nenulový zbytek po dělení většího čísla menším

Příklad – Výpočet největšího společného dělitele 3/3

- Vnitřní cykly můžeme nahradit přímým výpočtem zbytku po dělení operátorem **%**

```
int getGreatestCommonDivisorEuclid(int x, int y) {
   int remainder = x % y;
   while (remainder != 0) {
      x = y;
      y = remainder;
      remainder = x % y;
   }
   return y;
}
```

lec05/DemoGCD.java

Euklidův algoritmus

- Určíme zbytek po dělení daných čísel
- Zbytkem dělíme dělitele a určíme nový zbytek, až dosáhneme nulového zbytku
- Poslední nenulový zbytek je největší společný dělitel

Rozklad problému na podproblémy

- Postupný návrh programu rozkladem problému na podproblémy
  - Zadaný problém rozložíme na podproblémy
  - Pro řešení podproblémů zavedeme **abstraktní příkazy**
  - S abstraktními příkazy sestavíme hrubé řešení
  - Abstraktní příkazy realizujeme jako procedury (funkce)
- Rozklad problému na podproblémy ilustrujeme na příkladu hry NIM
  - Pravděpodobně první počítačová hra (1951)
  - Hra dvou hráčů v odebírání herních kamenů (nebo např. zápalek)

Čínská hra ve sbírání kamenů „Tsyanchidzi"
"L'Année Derniere a Marienbad"

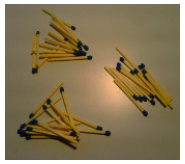
http://www.jot101.com/2013/09/nim-first-computer-game-1951_19.html

Hra NIM - Pravidla

- Pravidla
  - Hráč zadá počet herních kamenů (např. od 15 do 35)
  - Pak se střída se strojem v odebírání; odebrat lze 1, 2 nebo 3 kameny
  - Prohraje ten, kdo odebere poslední herní kámen
- Dílčí problémy
  - Zadání počtu kamenů
  - Odebrání kamenů hráčem
  - Odebrání kamenů strojem

Příklad průběhu hry

- Zadej počet kamenů (15 až 35): 18
- Stav hry
  - Počet kamenů ve hře
  - Kdo je na tahu – počítač (P) nebo hráč (H)
- Průběh:
  1. Počet kamenů 18; Kolik odeberete? **1** (H)
  2. Počet kamenů 17; Odebírám **1** (P)
  3. Počet kamenů 16; Kolik odeberete? **3** (H)
  4. Počet kamenů 13; Odebírám **1** (P)
  5. Počet kamenů 12; Kolik odeberete? **3** (H)
  6. Počet kamenů 9; Odebírám **1** (P)
  7. Počet kamenů 8; Kolik odeberete? **3** (H)
  8. Počet kamenů 5; Odebírám **1** (P)
  9. Počet kamenů 4; Kolik odeberete? **3** (H)
  10. Počet kamenů 1; Odebírám **1** (P)
- Hráč vyhrál, počítač odebral poslední kámen.



NIM pravidla pro odebírání

- Počet kamenů nevýhodných pro protihráče je 1, 5, 9, ..., obecně $4n+1$, kde n je celé nezáporné číslo
- Stroj musí z počtu kamenů K odebrat x kamenů, aby platilo

$$K - x == 4n + 1$$

  - $x == (K-1) - 4n$, tj. hledáme zbytek po dělení 4
  - $x \leftarrow (K-1) \bmod 4$
  - $x \leftarrow (K-1)\%4$
- Je-li $x == 0$, je okamžitý počet kamenů pro stroj nevýhodný a bude-li protihráč postupovat správně, stroj prohraje.

Hra NIM – Hrubý návrh řešení

```
int numberOfStones;
boolean machine = true;
numberOfStones = getNumberOfStones(limits);
do {
   if (machine) {
      machineTurn();
   } else {
      humanTurn();
   }
   machine = !machine; //change the player
} while (numberOfStones > 0);
return machine ? "Machine win!" : "Human win!";
```

- Podproblémy `getNumberOfStones`, `machineTurn` a `humanTurn` reprezentují abstraktní příkazy, které implementujeme jako funkce vracející počet kamenů k odebrání.

Hra NIM – Podrobnější návrh

```
1  final int MIN_INIT_STONES = 15;
2  final int MAX_INIT_STONES = 35;
3  boolean machine = true;
4  int numberOfStones = getNumberOfStones(limits);
5
6  if (
7        numberOfStones < MIN_INIT_STONES
8        || numberOfStones > MAX_INIT_STONES
9     ) {
10    return "Given number of stones is out of range";
11 }
12
13 do {
14    if (machine) {
15       numberOfStones -= machineTurn(numberOfStones);
16    } else {
17       numberOfStones -= humanTurn(numberOfStones);
18    }
19    machine = !machine; //change the player
20 } while (numberOfStones > 0);
21
22 return machine ? "Machine win!" : "Human win!";
```


Hra NIM – Implementace 1/4

```
Scanner scan;

int getNumberOfStones(int min, int max) {
   int n = -1;
   System.out.print("Enter number of stones in range [
     " + min + ", " + max + " ]: ");
   n = scan.nextInt();
   return (n >= min && n <= max ? n : -1);
}
```

lec05/DemoNIM.java

Hra NIM – Implementace 2/4

```
int humanTurn(int n) {
   int r = 0;
   while(r != 1 && r != 2 && r != 3) {
      System.out.print("No. of stones is " + n + ".
    How many stones you will take: " );
      r = scan.nextInt();
   }
   return r;
}
```

lec05/DemoNIM.java

Hra NIM – Implementace 3/4

```
int machineTurn(int n) {
   int r = (n - 1) % 4;
   if (r == 0) {
      r = 1;
   }
   System.out.println("No. of stones is " + n + ". I
    take " + r + ".");
   return r;
}
```

lec05/DemoNIM.java

Hra NIM – Implementace 4/5

```
1  public String play(boolean machineStart) {
2     final int MIN_INIT_STONES = 15;
3     final int MAX_INIT_STONES = 35;
4
5     boolean machine = machineStart;
6
7     int numberOfStones = getNumberOfStones(MIN_INIT_STONES,
        MAX_INIT_STONES);
8     if (numberOfStones < MIN_INIT_STONES || numberOfStones
       > MAX_INIT_STONES) {
9        return "Given number of stones is out of range";
10    }
11    do {
12       if (machine) {
13          numberOfStones -= machineTurn(numberOfStones);
14       } else {
15          numberOfStones -= humanTurn(numberOfStones);
16       }
17       machine = !machine;
18    } while(numberOfStones > 0);
19    return machine ? "Machine win!" : "Human win!";
20 }
```

lec05/DemoNIM.java

Hra NIM – Implementace 5/5

```
public static void main(String[] args) {
   DemoNIM nim = new DemoNIM();
   String result = nim.play(true); //machine start
    true/false
   System.out.println("Result of the game is: " +
    result);
}
```

lec05/DemoNIM.java

Vyzkoušejte si program sami napsat a otestujte jej pro případ kdy začíná počítač nebo člověk!

Kdo vyhrává při aplikování optimální strategie?

Výpočet faktoriálu

- Iterace

  $n! = n(n-1)(n-2)\ldots 2 \cdot 1$

- Rekurze

  $n! = 1$ pro $n \leq 1$

  $n! = n(n-1)!$ pro $n > 1$

```
int factorialI(int n) {
   int f = 1;
   for(; n > 1; --n) {
      f *= n;
   }
   return f;
}
```

```
int factorialR(int n) {
   int f = 1;
   if (n > 1) {
      f = n * factorialR(n-1);
   }
   return f;
}
```

lec05/DemoFactorial.java

Shrnutí přednášky

Diskutovaná témata

- Funkce, jejich jména a předání návratové hodnoty příkazem **return**
- Lokální proměnné a alokace paměti
- Cykly a řízení průchodu cyklu, příkazy **break** a **continue**
- Příklady dekompozice problému a jeho řešení
  - Výpočet největšího společného dělitele
  - Hra NIM
  - Výpočet faktoriálu iteračně a rekurzí
- Příště: Rekurze a iterace