

NPRG031 Programování 2

RNDr. Tomáš Holan, Ph.D.

Malá Strana, 4.patro, dveře 404

<http://ksvi.mff.cuni.cz/~holan/>

Tomas.Holan@mff.cuni.cz

NPRG031 Programování 2

- zkouška, písemná a ústní část
- podmínky zápočtu určuje cvičící, ale obecně
 - aktivní účast
 - domácí úkoly
 - zápočtový test
 - zápočtový program
- zvláštní cvičení – Martin Mareš

O čem to bude

- nový jazyk
- ale programování není jen jazyk a algoritmy

Nové problémy:

- jak zvládat větší program
- jak spolupracovat na tvorbě programu
- **SWING** <https://www.zentao.pm/agile-knowledge-share/tree-swing-project-management-cartoon-97.mhtml>
- na většinu otázek neexistuje JEDNA SPRÁVNÁ odpověď

S tím souvisí

dekompozice = rozklad na funkce / objekty / moduly

proč: abychom mohli program měnit

(protože změna požadavků, protože jsme rozhodli chybně, protože...)

s co nejmenším dopadem na celý program

nástroje některé nástroje jsou software, některé jen pravidla/konvence/rady

pro

- správu požadavků
- správu úkolů
- správu chyb
- správu verzí
- testování
- dokumentaci (wiki, generovaná dokumentace)
- přemýšlení (obrázky, myšlenkové mapy, UML)

S tím souvisí

omezení/zvyklosti

- coding conventions
 - PY: mezery nebo tabulátory, kolik mezer, kde
 - malá/velká písmena
 - „{“ na začátek nebo na konec řádky
- linting
- code smells
- metriky kódu

používání knihoven

- výběr KTERÉ knihovny / výběr KTERÁ VERZE
- a i na to jsou nástroje (známe z PY - PIP)

metodiky vývoje programu

K těm knihovnám...

SBOM (Software Bill of Materials)

<https://bidenwhitehouse.archives.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/>

(j) the term “Software Bill of Materials” or “SBOM” means a formal record containing the details and supply chain relationships of various components used in building software. Software developers and vendors often create products by assembling existing open source and commercial software components. The SBOM enumerates these components in a product. It is analogous to a list of ingredients on food packaging. An SBOM is useful to those who develop or manufacture software, those who select or purchase software, and those who operate software. Developers often use available open source and third-party software components to create a product; an SBOM allows the builder to make sure those components are up to date and to respond quickly to new vulnerabilities.

EU CRA (EU Cyber Resilience Act)

<https://anchore.com/sbom/eu-cra/>

The European Union's Cyber Resilience Act (CRA) mandates that manufacturers of products with software components create and maintain a software bill of materials (SBOM). This SBOM must be in a commonly used, machine-readable format and include, at a minimum, the top-level dependencies of the product.

Jazyk C#

. C#

- Java 1995, bytecode
- C# 2002, .NET
- standardy ECMA a ISO/IEC
<https://learn.microsoft.com/en-us/dotnet/csharp/specification/overview>
- Anders Hejlsberg
-

* Visual Studio

* Visual Studio Code

Přechod od Pythonu k jazyku C#

Používání mezer a řádkování

- mezery, tabelátory, konce řádek **nemají žádný význam**
- doporučená a prostředím podporovaná indentace

Proměnná

- představuje **místo v paměti**, má **svůj typ**
(a musíme ho **dodržovat**)
- musí se před použitím **deklarovat**



...takže překladač může některé naše chyby najít za nás.

Identifikátory

- case-sensitivní
- možnost používat diakritiku a národní abecedy
(ale nemusíme dělat všechno, co není zakázáno:
給我發郵件 () ;)
- klíčová slova malými písmeny
- konvence (zvyklosti):
 - proměnné malými písmeny, konstanty velkými písmeny
 - jména prostorů*), tříd, metod a vlastností, veřejné členy
 - > „PascalskáNotace“
např. `Math`, `DivideByZeroException`, `Main`, `WriteLine`
 - soukromé metody začínají malým písmenem
 - > „velbloudíNotace“

Struktura programu

Celý program se skládá ze tříd,
vše se deklaruje a používá uvnitř tříd
(proměnné, konstanty, funkce, ...).

Položky deklarované ve třídě:

- datové složky třídy = **členské proměnné**
- metody = **členské funkce**

Prozatím: celý program je tvořen jedinou statickou*)
metodou (její obsah tedy odpovídá celému programu)

Někdy příště: jak jinak to může vypadat se třídami

Proměnné

- zápis deklarace proměnné

- **syntaxe:** `int alfa;`

- umístění deklarace:

BUĎ členská proměnná třídy (tzn. datová složka objektu)

NEBO lokální kdekoliv ve funkci, ale nesmí zakrýt jinou stejnojmennou deklaraci uvedenou v téže funkci

(pozor na kolize!)

- lokální platnost deklarace v bloku, kde je uvedena

- možnost inicializace v rámci deklarace: `int alfa = 15;`

- v programu nelze použít nedefinovanou hodnotu proměnné
(kontrola při překladu) 

- hodnotové a referenční typy

- všechno*) je objekt (instance nějaké třídy)

Konstanty

- syntaxe stejná jako inicializované proměnné,
specifikátor `const`:

```
const int ALFA = 15;
```

- číselné konstanty podobné jako v Pythonu (různé typy)
- konstanty typu `char` (znak) v apostrofech: `'a'`,
typu `string` v uvozovkách: `"aaa"`



Typy

Hodnotové

celé číslo

`int` `System.Int32` 32 bitů

další typy: `byte`, `sbyte`, `short`, `ushort`, `uint`, `long`, `ulong`

desetinné číslo

`double` `System.Double` 64 bitů

další typy: `float`, `decimal`

logická hodnota

`bool`

znak

`char` `System.Char` 16 bitů Unicode

výčtový typ

`enum`

struktura

`struct`

Referenční

pole

`[ ]      System.Array`

znakový řetězec

`string      System.String`

objekt určité třídy

`class`

`(standardní třídy, např. ArrayList,
StringBuilder, List<>)`

Hlavní rozdíl mezi kategoriemi typů:

Dosazuje se **hodnota** nebo **reference**.

Aritmetické výrazy

- obvyklé symboly operací i priority stejné jako v Pythonu

+ - * /

- **POZOR:** symbol / představuje desetinné i celočíselné dělení

(zvolí se podle typu argumentů) = zdroj chyb!



- znak % pro modulo (zbytek po celočíselném dělení)
- klíčová slova checked, unchecked - určení, zda se má kontrolovat aritmetické přetečení v celočíselné aritmetice
- použití jako checked(výraz) nebo checked{blok}
- standardní matematické funkce
= statické*) metody třídy Math

Středník

- ukončuje každý příkaz
(musí být i za posledním příkazem bloku!)
- nesmí být za blokem ani za hlavičkou funkce
- odděluje sekce v hlavičce for-cyklu

Čárka

- odděluje deklarace více proměnných téhož typu
- odděluje parametry v deklaraci funkce i při volání funkce
- odděluje indexy u vícerozměrného pole

Komentáře

- jednořádkové `// xxx` do konce řádku
- víceřádkové `/* xxx */`
- dokumentační `///`

Blok (složený příkaz)

- skupina příkazů, kterou chceme spojit,
třeba vnitřek cyklu nebo větev podmíněného příkazu
- závorky `{ }` namísto odsazení v Pythonu

Dosazovací příkaz

- syntaxe: `proměnná = výraz` např. `i = 2*i + 10;`

Příkaz modifikace hodnoty

```
i++; ++i;  
i--; --i;
```

```
i += 10;  
i -= 10;  
i *= 10;  
i /= 10;  
i %= 10;
```

Podmíněný příkaz

- podmínka = výraz typu bool v závorkách

```
if (a == 5)    b = 17;
```

```
if (a == 5)    b = 17;
```

```
    else      b = 18;
```

- relační operátory: == != < > <= >=

- logické spojky

&& and (zkrácené vyhodnocování)

|| or (zkrácené vyhodnocování)

& and (úplné vyhodnocování)

| or (úplné vyhodnocování)

! not

^ xor

For-cyklus

- **syntaxe:**

for (inicializace; podmínka pokračování; příkaz iterace)
příkaz těla // **jeden příkaz!**

```
for (int i=0; i<N; i++)    a[i] = 3*i+1;
```

- některá sekce může být prázdná (třeba i všechny)

(pokud je v některé sekci víc příkazů, oddělují se čárkou)

Cykly while a do-while

- cyklus **while** stejný jako while-cyklus v Pythonu (podmínka je celá v závorce)

```
while (podmínka) příkaz;
```

- cyklus **do-while** má podmínku na konci, tzn. dokud podmínka platí, cyklus se provádí

```
do příkaz while (podmínka);
```

- více příkazů v těle cyklu musí být uzavřeno v bloku { }

Ukončení cyklu

- příkazy

`break;`

`continue;`

- stejný význam jako v Pythonu

Příkaz switch

- vícenásobné rozvětvení
- varianta se může rozhodovat podle výrazu celočíselného, znaku nebo také stringu
- sekce **case**, za každým case jediná konstanta, ale pro více case může být společný blok příkazů
- poslední sekce může být default:
- je povinnost **ukončit** každou sekci case...
(i sekci default,
neboť ta nemusí být uvedena poslední)
...příkazem
break, příp. return nebo goto



```
int j, i = ...;  
switch (i)  
{  
    case 1:  
        i++; break;  
    case 2:  
    case 3:  
        i--; break;  
    default:  
        i=20; j=7; break;  
}
```

(pozor: v C, C++, Java, PHP... se může propadat mezi
sekcemi = zdroj chyb, v C# opravený)



Funkce

- musí patřit nějaké třídě nebo objektu (později)
- když nevrací výsledek
 - > funkce typu **void**
- v deklaraci i při volání vždy píšeme **()**,
i když nemá žádné parametry
- ve funkci nelze lokálně definovat*) jinou funkci,
strukturu nebo třídu, *)
Ize tam ale deklarovat lokální proměnné
(ve třídě lze deklarovat jinou třídu
ta může mít své metody)

*) V C#7 už lze...



Funkce...

- mohou vracet i složitější*) typy
- `return <hodnota>;`
 - definování návratové hodnoty a ukončení funkce
- v případě funkcí typu `void` pouze `return;`

- předávání parametrů:

hodnotou (= výchozí způsob)

odkazem - specifikátor `ref` v hlavičce i při volání *)



výstupní parametr



- specifikátor `out` v hlavičce i při volání

(`out` je také odkazem, nemá ale vstupní hodnotu)

*) V C#7 už lze vracet "tuple"



Výchozí metoda Main()

- plní funkci hlavního programu
(určuje začátek a konec výpočtu)
- je to statická*) metoda nějaké třídy
(nic „mimo třídy“ neexistuje),
často se pro ni vytváří samostatná třída
- obvykle jediná v aplikaci
→ je tak jednoznačně*) určeno, kde má začít výpočet
- *) může jich být i více, pak se ale musí určit,
která funkce Main() (ze které třídy)
se má použít při spuštění programu
- **syntaxe:** `static void Main(string[] args)`

Standardní vstup a výstup

`Console.Read();`

vrací `int` = jeden znak ze vstupu (jeho kód)

`Console.ReadLine();`

vrací `string` = jeden řádek ze vstupu

`Console.Write(výraz);`

vypíše hodnotu zadaného výrazu

`Console.WriteLine(výraz);`

vypíše hodnotu zadaného výrazu a odřádkuje

Formátovaný výstup

```
Console.WriteLine(string);
```

```
Console.WriteLine(  
    "x0={0} x1={1} x2={2} ...a to je vše",  
    x0, x1, x2  
);
```

do stringu se dosadí hodnoty výrazů po řadě na místa
vyznačená pomocí {0}, {1}, {2}, atd., případně i s
požadovaným formátováním {0:N}

```
Console.WriteLine($"a={a} b={b} a*b={a*b}");
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            XXXXXXXXXXXXXXXXXXXXXXXX
        }
    }
}
```


Příklad: Eukleidův algoritmus

```
static void Main(string[] args)
{
    Console.WriteLine(
        "Zadej dvě kladná celá čísla:");
    int a = int.Parse(Console.ReadLine());
    int b = int.Parse(Console.ReadLine());
    while (a != b)
    {
        if (a > b) a -= b;
        else b -= a;
    }
    Console.WriteLine(
        "Nejv. spol. dělitel: {0}", a);
    Console.ReadLine();
}
```

Příklady:

Prvočíselný rozklad

Hornerovo schéma - vstup čísla po znacích

```

static void Main(string[] args)
{
    int v;
    int c;
    c = Console.Read();
    // preskocit ne-cislice:
    while ((c < '0') || (c > '9'))
    {
        c = Console.Read();
    }
    // nacitat cislice:
    v = 0;
    while ((c >= '0') && (c <= '9'))
    {
        v = 10 * v + (c - '0');
        c = Console.Read();
    }
    Console.WriteLine(v);
    Console.ReadLine();
}

```

Dynamicky alokované proměnné

- vytvářejí se pomocí zápisu
`new + konstruktor*`) vytvářeného objektu
- `new` je funkce, vrací vytvořenou instanci
(ve skutečnosti ukazatel na ni)
- v odkazech se nepíšou $\wedge$ jako se píšou v Pascalu nebo v C
- string, pole, třídy = referenční typy
- konstanta `null` (jako `None` v Pythonu)
- automatická správa paměti
nedostupné objekty jsou automaticky uvolněny z paměti
(ne nutně úplně okamžitě, ale až to bude potřeba)

Příklad

```
prvni = null;
    // to je korektní zrušení celého spojového seznamu

class Uzel
{
    public int info;
    public Uzel dalsi;
}

class Program
{
    static void Main(string[] args)
    {
        Uzel prvni = new Uzel();
        prvni.info = 123;
        prvni.dalsi = null;
        // ...
    }
}
```

Pole

- deklarace: `int[] aaa;`
- referenční typ, je nutné vytvořit pomocí `new`:
`int[] aaa = new int[10];`
- každé pole je instancí třídy odvozené*) z abstraktní statické třídy `System.Array`
- indexování vždy od 0
- možnost inicializace:
`int[] aaa = new int[3] { 2, 6, 8 };`
`int[] aaa = { 2, 6, 8 };`
- počet prvků: `aaa.Length`
- vždy se provádějí kontroly přetečení mezí 
při indexování `aaa[i]`

POZOR !

```
static void Main(string[] args)
{
    int[] aaa = { 2, 6, 8 };
    int[] bbb;
    bbb = aaa; // POZOR!!
    aaa[0] = 27;

    Console.WriteLine(bbb[0]);
}
```

dosazuje se ukazatel !!

Pole...

- připravené metody, např. `CopyTo`, `Sort`, `Reverse`, `BinarySearch`, `Array.Reverse(aaa)` ;
- vícerozměrné pole
obdélníkové `[,]` a nepravidelné `[][]`

Nepravidelné dvourozměrné pole je ve skutečnosti pole polí
(tzn. pole ukazatelů na řádky,

což jsou pole jednorozměrná),

- každý řádek je třeba zvlášť vytvořit pomocí `new`
- řádky mohou mít různou délku

```
int[][] aaa = new int[3][];  
aaa[0] = new int[4];  
aaa[1] = new int[6];  
aaa[2] = new int[2];
```


Příklad: Třídění čísel v poli - přímý výběr

```
static void Main(string[] args)
{
    Console.Write("Počet čísel: ");
    int pocet = int.Parse(Console.ReadLine());
    int[] a;
    a = new int[pocet];
    int i = 0;
    while (i < a.Length)
        a[i++] = int.Parse(Console.ReadLine());
    i = 0;

    while (i < a.Length)
    {
        int k = i;
        int j = i+1;
```

```

        while (j < a.Length)
        {
            if (a[j] < a[k]) k = j;
            j++;
        }
        if (k != i)
        {
            int x = a[i];
            a[i] = a[k];
            a[k] = x;
        }
        i++;
    }

    i = 0;
    while (i < a.Length)
        Console.Write(" {0}", a[i++]);
    Console.WriteLine();
}

```

Znakový řetězec

- deklarace: `string sss;`
- `typ string` - referenční typ, alias pro třídu `System.String`
- vytvoření instance: `string sss = "abcdefg";`
- ~~nulou ukončené řetězce~~, nemají omezenou délku
- indexování znaků od 0
- délka = `sss.Length`
- obsah nelze měnit (na to je třída `StringBuilder`)
- všechny objekty mají konverzní metodu `ToString()`,
pro struktury a objekty je vhodné předefinovat ji
(jinak se vypisuje jenom jejich jméno)

Struktura - struct

- „zjednodušená třída“
- je to hodnotový typ
(na rozdíl od instance třídy se nemusí alokovat)
- může mít i konstruktor
(má i implicitní bezparametrický konstruktor)
- některá omezení oproti třídám (např. nemůže dědit)

```
struct Bod
{
    public int x, y;
    public Bod(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Tuple - n-tice

. System.Tuple

- **class**
- **1..7 prvků**
- `new Tuple<string, int, int, int, int, int, int>(`
 `"New York", 7891957, 7781984,`
 `7894862, 7071639, 7322564, 8008278);`

NEBO

`Tuple.Create("New York", 7891957, 7781984, 7894862, 7071639,`
`7322564, 8008278);`

- **složky Item1, Item2... (od jedničky!)**

. System.ValueTuple

- **struct**
- `(double, int) t = (4.5, 3);`
 `(double a, int b) t = (a: 4.5, b: 3);`
- **složky Item1, Item2... nebo podle pojmenování**