BRNO UNIVERSITY OF TECHNOLOGY

SIEMENS

2 – Základy jazyka C#

Obsah přednášky

- Syntaxe jazyka C#
- Datové typy
- Proměnné a parametry
- Výrazy a operátory
- Přikazy
- Namespace – jmené prostory

Ukázka kódu

```
using System;                   // Importing namespace

namespace FirstProgram          // Namespace declaration
{
  public class Test             // Class declaration
  {
    static void Main()          // Method declaration
    {
      int x = 12 * 30;          // Statement 1
      Console.WriteLine(x);     // Statement 2

      Console.WriteLine         // Statement 3
        (1 + 2 + 3 + 4 + 5
          + 6 + 7 + 8 + 9 + 10);
    }                           // End of method
  }
}
```

Syntaxe jazyka C#

SIEMENS

- Vychází z C a C++
- **Identifikátory**
  - Názvy tříd, metod, proměnných atd.
  - Např.: `System, FirstProgram, Test, Main`
- **Klíčová slova (Keywords)**
  - Názvy rezervované kompilátorem
  - Např.: `public, static, int`
  - Pokud potřebujeme využít název klíčového slova, je nutné použít prefix @
    - Např.: `@public, @static, @int`
    - Může být užitečné pokud používáme knihovnu napsanou v jiném .NET jazyce

Přehled klíčových slov

SIEMENS

- abstract
- as
- base
- bool
- break
- byte
- case
- catch
- char
- checked
- class
- const
- continue
- decimal
- default
- delegate
- do
- double
- else
- enum
- event
- explicit
- extern
- false
- finally
- fixed
- float
- for
- foreach
- goto
- if
- implicit
- in
- int
- interface
- internal
- is
- lock
- long
- namespace
- new
- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using
- virtual
- void
- volatile
- while

Syntaxe jazyka C#

- **Kontextová klíčová slova**
  - Lze je použít pouze v daném kontextu
  - Lze použít jako názvy i bez @, pokud nejsme v daném kontextu
  - Seznam:
    - `add`
    - `ascending`
    - `async`
    - `dynamic`
    - `equals`
    - `from`
    - `in`
    - `into`
    - `join`
    - `partial`
    - `remove`
    - `select`
    - `where`
    - `yield`

Syntaxe jazyka C#

- **Literály**
  - Data vložené do programu
  - Např.: `12, 30`
- **Oddělovače**
  - Znaky použité pro strukturování programu.
  - Složené závorky `{` a `}`
    - Slouží pro seskupení více příkazů do bloku
  - Středník `;`
    - Slouží pro oddělení jednotlivých příkazů
    - Příkaz může být na více řádků
    -
```
Console.WriteLine
        (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Syntaxe jazyka C#

- **Operátory**
  - Např.: `. () * + -`
- **Komentáře**
  - Řádkové

```
// Řádkový komentář
```

  - Blokové

```
/* Blokový komentář je možné rozdělit
na více řádků */
```

  - Dokumentační

```
/// <summary>
/// Popis metody/třídy atd.
/// </summary>
public class Test
```

SIEMENS

Datové typy

- **Datové typy definují předpis pro hodnoty.**
- Každá hodnota/výraz musí být nějakého typu.
- Viz příklad, kde hodnoty 12 a 30 jsou typu `int` a proměnná x je také typu `int`
- **Předdefinované datové typy**
  - Speciální typy podporované kompilátorem (některá klíčová slova)
  - Např.: `string, int, byte`
- **Nullable typy**
  - Struktura Nullable<T>
  - Pro hodnotové typy je možné použít sufix ?
    - Např.: `int? a = null;`
    - `Nullable<int> a = null;`

Datové typy

- **Konstanty**
  - Proměnná, která vždy obsahuje stejnou hodnotu
- **Uživatelské datové typy**
  - Z předdefinovaných typů je možné nadefinovat vlastní datové typy
  - Members
    - Instanční vs statické

SIEMENS

Předdefinované numerické typy

- **Celočíselné typy se znaménkem**

C# type	System type	Suffix	Size	Range
sbyte	SByte		8 bits	-2^7 to 2^7-1
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32		32 bits	-2^{31} to $2^{31}-1$
long	Int64	L	64 bits	-2^{63} to $2^{63}-1$

- **Celočíselné typy bez znaménka**

C# type	System type	Suffix	Size	Range
byte	Byte		8 bits	0 to 2^8-1
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$

Předdefinované numerické typy

SIEMENS

- **Desetinné číselné typy**

C# type	System type	Suffix	Size	Range
float	Single	F	32 bits	$\pm(10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm(10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm(10^{-28}$ to $10^{28})$

Numerické datové typy

- **Možnosti zápisu čísel**
  - Klasický zápis
    - Např.: 127, 42 atd.
  - Hexadecimální zápis
    - Např.: 0x7F, 0x2A atd.
  - Zápis desetinných čísel
    - Znak .(tečka) slouží pro oddělení desetinných míst
    - Znak e lze použít jako exponent
- **Datové typy numerických hodnot**
  - Pokud číslo obsahuje . nebo e pak je to typ `double`
  - Jinak je to první typ, do kterého se hodnota vejde z `int, uint, long, ulong`

Numerické datové typy

- **Možnost specifikovat datové typy**
  - Je možné pomocí písmeného suffixu definovat konkrétní datový typ čísel
  - Použití suffixů viz příklad:

```
Console.WriteLine(1f.GetType());            // Float (float)
Console.WriteLine(1d.GetType());            // Double (doulbe)
Console.WriteLine(1m.GetType());            // decimal (decimal)
Console.WriteLine(1u.GetType());            // UInt32 (uint)
Console.WriteLine(1L.GetType());            // Int64 (long)
Console.WriteLine(1ul.GetType());           // UInt64 (ulong)
```

Numerické datové typy - přetypování

- **Přetypování číselných typů**
  - Převod celočíselného typu na celočíselný typ
    - Je implicitní pokud cílový typ umožňuje pojmout celý rozsah zdrojového
    - Jinak je nutné přetypovat explicitně
  - Převod typu čísla s plovoucí čárkou na typ čísla s plovoucí čárkou
    - `float` je možné implicitně přetypovat na `double`
    - Ale double je nutné na float přetypovávat explicitně
  - Převod typu s plovoucí čárkou na celočíselný typ
    - Všechny celočíselné typy lze implicitně přetypovat na `float` nebo `double`
    - Naopak je nutné explicitní přetypování
      - Dochází zde k ořezání desetiné části
      - Může dojít ke ztrátě přesnosti

Numerické datové typy - přetypování

- **Přetypování číselných typů**
  - Převod typu decimal
    - Celočíselné typy se na typ decimal přetypovávají implicitně
    - Všecny ostatatní převodyjsou explicitní

Numerické datové typy - aritmetické operátory

- **Aritmetické operátory**

  + sčítání

  - odčítání

  * násobení

  / dělení

  ++ inkrement

  -- dekrement

Numerické datové typy - přetečení

- Přetečení celočíselných typů

```
int a = int.MinValue;
a--;
Console.WriteLine(a == int.MaxValue); // True
```

- Možnost využití klíčového slova `checked` a nebo přepínače **/checked+** při kompilaci

```
int a = int.MinValue;
var i = checked(a--); // throw OverflowException
Console.WriteLine(i == int.MaxValue);
```

SIEMENS

Numerické datové typy - bitové operace

Operator	Význam	Příklad	Výsledek
~	Not	~0xfU	0xfffffffOU
&	And	0xf0 & 0x33	0x30
|	Or	0xf0 | 0x33	0xf3
^	Xor	0xff00 ^ 0x00ff	0xf0f0
<<	Posun vlevo	0x20 << 2	0x80
>>	Posun vpravo	0x20 >> 1	0x10

Numerické datové typy

- 8 a 16-ti bitové typy nemají aritmetické operátory
- Tzn. `byte, sbyte, short, ushort`
  - A proto je kompilátor v případě potřeby převádí na větší typy a to může způsobit chybu kompilace.

```
short x = 1, y = 1;
short z = x + y; // Compile-time error
```

  - Řešením je explicitní přetypování

```
short z = (short)(x + y); // OK
```

  - Speciální hodnoty desetinných typů
    - `double.NaN`
    - `float.PositiveInfinity`
    - `double.NegativeInfinity`

SIEMENS

Zaokrouhlovací rozdíly

- Typy `float` a `double` se ukládají ve dvojkové a proto pouze násobky 2 jsou uloženy přesně

```
float tenth = 0.1f; // Not quite 0.1
float one = 1f;
Console.WriteLine (one - tenth * 10f); // -1.490116E-08
```

- Typ `decimal` je založen na desítkové soustavě

```
decimal m = 1M / 6M; // 0.1666666666666666666666666667M
double d = 1.0 / 6.0; // 0.16666666666666666
decimal notQuiteWholeM = m+m+m+m+m+m;
        // 1.0000000000000000000000000002M
double notQuiteWholeD = d+d+d+d+d+d; // 0.99999999999999989
Console.WriteLine (notQuiteWholeM == 1M); // False
Console.WriteLine (notQuiteWholeD < 1.0); // True
```

SIEMENS

Porovnání decimal hodnot

```
double x = 49.0;
double y = 1 / x;

double calculatedResult = x * y;
double expectedResult = 1.0;

bool areSame = calculatedResult == expectedResult;
```

- **Co obsahuje proměnná `areSame`?**

SIEMENS

Porovnání decimal hodnot

```
public static bool AlmostEqual(double a, double b)
{
  const double tolerance = 0.00000001;
  if (a == b)
  {
    return true;
  }
  return Math.Abs(a - b) < tolerance;
}
```

Typ Boolean

SIEMENS

- **System.Boolean / bool**
- Uložení logických hodnot
- V paměti zabírá celý byte
- Žádný z numerických typů nelze přerypovat na bool
- Operátory
  - ==, != lze použít pro porovnání jakýchkoliv typů
  - ==, !=, <, >, <=, >= lze použít pro porovnání číselných typů
- Podmíněné operátory - &&, ||

```
public bool UseUmbrela(bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

- Líné vyhodnocování (Lazy evaluation)

Řetězce a znaky

- Typ pro znak: **System.Char / char**
  - Zapisuje se do jednoduchých uvozovek např.: `'a'`
  - Převod na číselné typy
    - Pro typy mající rozsah ushort implicitně
    - Jinak explicitně
- Typ pro řetězec: **System.String / string**
  - Reprezentován posloupností znaků
  - Je to referenční datový typ
  - Zapisuje se do dvojitých uvozovek např.: `"string value"`
  - Víceřádkový řetězec je možné napsat pomocí @. Např.:

```
@"First line
second line"
```

  - Pro prázdný řetězec existuje konstanta `string.Empty`

Řetězce - Escape sekvence

Znak	Význam	Hodnota
\’	Apostrof	0x0027
\”	Uvozovky	0x0022
\\	Zpětné lomítko	0x005C
\o	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\n	Nový řádek	0x000A
\r	Carriage return	0x000D
\t	Horizontální tab	0x0009
\v	Vertikální tab	0x000B
\u \x	Unicode pomocí hexa zápisu na 4 místa \u00a9	

Řetězce

- Spojování řetězců
  - Operátor +

```
string s = "a" + "b";
```

  - Ne všechny operandy musejí být typu string a pak dochází k zavolání metody ToString()

```
string s = "a" + 5; // a5
```

  - Pro mnohonásobné používání operáoru + pro spojování řetězců se z hlediska výkonnosti vyplatí použít třídu System.Text.StringBuilder
    - **string.Format**
      - Pro formátování řetězců za pomocí řídících znaků

```
int x = 5;
string.Format("x = {0}", x);
```

Pole

- Reprezentuje fixní počet proměnných (prvky pole) stejného typu
- Ukládá se v souvislém bloku paměti
- **Deklarace pole**

```
char[] characters = new char[5];
char[] characters = new char[] {'a','b','c'};
char[] characters = {'a','b','c'};
```

- **Přístup k prvkům pole**

```
characters[0] = 'a';
characters[1] = 'b';
for (int I = 0; I < characters.Length; i++)
{
  Console.WriteLine(characters[i]);
}
```

SIEMENS

Pole

- Inicializace prvků pole
  - Hodnotové typy - defaultní hodnotu
  - Referenční typy – `null`
- Kontrola rozměrů pole
  - `IndexOutOfRangeException`

Pole - vícerozměrné

SIEMENS

- Matice
  - Delkarace pomocí [,]

```
int[,] matrix = new int[3,3];
int[,] matrix = new int[,] {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

- Přístup k prvkům

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i, j] = i * 3 + j;
```

- Pole polí
  - Deklarace pomocí [][]

```
int[][] matrix = new int[3][];
```

  - Přístup k prvkům

```
matrix[i][j] = 5;
```

SIEMENS

Proměnné a parametry

- Proměnné představují úložné místo s modifikovatelnou hodnotou.
- Proměnné se ukládají buď na zásobník (stack) nebo na haldu (heap)
- Proměnná může být
  - Lokální
  - Parametr
  - Prvek pole
- Klíčové slovo var – implicitně typované lokální proměnné
  - Lze použít při deklaraci a současném přiřazení
  - Může snižovat čitelnost kódu

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Proměnné a parametry

- **Zásobník (stack)**
  - Alokovaný blok paměti pro uložení lokálních proměnných a parametrů
  - Paměť je alokována po dobu vykonávání funkce
- **Halda (heap)**
  - Na haldu se ukládají všechny objekty (referenční datové typy)
  - O uvolňování paměti se stará Garbace collector
- **Pravidla pro přiřazení proměnných**
  - Lokálním proměnným musí být přiřazena hodnota před jejím čtením
  - Argumenty metody musí být zadány
  - Všechny ostatní hodnoty (fields, prvky polí) jsou automaticky inicializovány

Proměnné a parametry

```
static int y;
static void Main()
{
  int x;
  Console.WriteLine (x); // Compile-time error
  int[] ints = new int[2];
  Console.WriteLine (ints[0]); // 0
  Console.WriteLine (y); // 0
}
```

Typ	Výchozí hodnota
Všechny referenční	null
Všechny numerické a výčtové typy	0
Typ char	'\0'
Typ bool	false

SIEMENS

Parametry

- **Způsoby předávání parametrů**

Modifikátor	Předáno	Defínice proměnné
(none)	Hodnotou	Going in
ref	Referencí	Going in
out	Referencí	Going out

Parametry

- **Předání hodnotou**
  - Je předána kopie hodnoty
  - U referenčních typů je zkopírována reference, ale ne objekt

```
static void Foo(StringBuilder fooSB)
{
  fooSB.Append("test");
  fooSB = null;
}
...
StringBuilder sb = new StringBuilder();
Foo (sb);
Console.WriteLine (sb.ToString()); // test
```

Parametry

- **Modifikátor ref**
  - Předáno referencí

```
private static void Foo(ref int p)
{
  p = p + 1; // Increment p by 1
}
...
int x = 8;
Foo(ref x); // Ask Foo to deal directly with x
Console.WriteLine(x); // x is now 9
```

Parametry

- **Modifikátor out**
  - Obdobné jako ref s dvěmi rozdíly
    - Proměnná nemusí být inicializována před voláním funkce
    - Musí být přiřazena před koncem funkce

```
private static void Split(string name, out string firstNames,
    out string lastName)
{
  int i = name.LastIndexOf(' ');
  firstNames = name.Substring(0, i);
  lastName = name.Substring(i + 1);
}
...
string a, b;
Split("Stevie Ray Vaughan", out a, out b);
Console.WriteLine(a); // Stevie Ray
Console.WriteLine(b); // Vaughan
```

Parametry

- **Modifikátor params**
  - Může být jako použit pouze u posledního parametru metody
  - Musí být deklarován jako pole
  - Slouží pro předání proměnného počtu parametrů stejného typu

```
private int Sum(params int[] ints)
{
  int sum = 0;
  for (int i = 0; i < ints.Length; i++)
    sum += ints[i]; // Increase sum by ints[i]
  return sum;
}
...
int total = Sum(1, 2, 3, 4);
Console.WriteLine(total); // 10
```

SIEMENS

Parametry

- **Nepovinné parametry**

```
void Foo(int x = 2) { … }
...
Foo();
```

- **Pojmenované argumenty**

```
void Foo(int x = 2, int y = 3) { … }
...
Foo(y:4, x:4);
Foo(y: ++a, x: --a);
Foo(y: 1);
```

Výrazy a operátory

- **Výraz** v podstatě určuje nějakou hodnotu.
- Nejjednodušší výraz je konstatna, nebo proměnná
  - `5`
- Výrazy můžeme kombinovat pomocí **operátorů**
  - `5 * 4`
  - `(5 * 4) + 1`
- Operátory můžou být unární, binární nebo ternární
- Binární operátory používají infix notaci (operátor mezy operandy)
- **Primární výrazy**
  - Slouží k výstavbě jazyka
    - `Math.Log(1)` – obsahuje dva primární výrazy . a ()

Výrazy a operátory

- **Void výrazy**
  - Nemají hodnotu
  - Nelze je kombinovat pomocí operátorů na další výrazy
- **Výraz přiřazení**
  - `x = x + 5;`
  - Je možné použít i jako součást jiného výrazu
    - `y = 5 * (x = 2);`
  - Inicializace více proměnných
    - `a = b = c = d = e = 0;`
  - Kombinované opeátory
    - `X += 5; // ekvivalentní x = x + 5;`

SIEMENS

Výrazy a operátory

- **Priorita operátorů a přiřazení**
  - Pořadí vyhodnocování operátorů je dle priority operátorů
  - Při stejné prioritě rozhoduje pořadí
  - Zleva asociativní operátory
    - `8 / 4 / 2` odpovídá `(8 / 4) / 2`
  - Zprava asociativní operátory
    - `x = y = 3;`

Tabulka operátorů

SIEMENS

Category	Operator symbol	Operator name	Example	User overloadable
Primary	.	Member access	x.y	No
	-> (unsafe)	Pointer to struct	x->y	No
	()	Function cal	x()	No
	[]	Array/Index	a[x]	via indexer
	++	Post-increment	x++	No
	--	Post-decrement	x--	No
	new	Create instance	new x()	No
	stackalloc	Unsafe stack allocation	stackalloc(10)	No
	typeof	Get type from identifier	typeof(int)	No
	checked	Integral overflow check on	checked(x)	No
	unchecked	Integral overflow check off	unchecked(x)	No

Tabulka operátorů

Category	Operator symbol	Operator name	Example	User overloadable
Primary	default	Default value	default(int)	No
	await	Await	await myTask	No
Unary	sizeof	Get size of struct	sizeof(int)	No
	+	Positive value of	+x	Yes
	-	Negative value of	-x	Yes
	!	Not	!x	Yes
	++	Pre-increment	++x	Yes
	--	Pre-decrement	--x	Yes
	()	Cast	(int)x	No
	* (unsafe)	Value at address	*x	No
	&(unsafe)	Address of value	&x	No

Tabulka operátorů

Category	Operator symbol	Operator name	Example	User overloadable
Multiprivative	*	Multiply	x * y	Yes
	/	Divide	x / y	Yes
	%	Remainder	x % y	Yes
Additive	+	Add	x+y	Yes
	-	Subtract	x-y	Yes
Shift	<<	Shift left	x<<y	Yes
	>>	Shift right	x>>y	Yes
Relational	<	Less than	x<y	Yes
	>	Greater than	x>y	Yes
	<=	Less than or equals to	x<=y	Yes
	>=	Greater than or exuals to	x>=y	Yes

SIEMENS

Tabulka operátorů

<table>
<tr><th>Category</th><th>Operator symbol</th><th>Operator name</th><th>Example</th><th>User overloadable</th></tr>
<tr><td>Relational</td><td>is</td><td>Type is or is subclass of</td><td>x is y</td><td>No</td></tr>
<tr><td></td><td>as</td><td>Type conversion</td><td>x as y</td><td>No</td></tr>
<tr><td>Logical And</td><td>&</td><td>And</td><td>x & y</td><td>Yes</td></tr>
<tr><td>Logical Xor</td><td>^</td><td>Exclusive Or</td><td>x ^ y</td><td>Yes</td></tr>
<tr><td>Logical Or</td><td>|</td><td>Or</td><td>x | y</td><td>Yes</td></tr>
<tr><td>Conditional And</td><td>&&</td><td>Conditional And</td><td>x && y</td><td>Via &</td></tr>
<tr><td>Conditional Or</td><td>||</td><td>Conditional or</td><td>x || y</td><td>Via &</td></tr>
<tr><td>Null coalescing</td><td>??</td><td>Null coalescing</td><td>x ??</td><td>No</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Conditional</td><td>isTrue? x : y</td><td>No</td></tr>
<tr><td>Assignment</td><td>=</td><td>Assign</td><td>x = y</td><td>No</td></tr>
<tr><td></td><td>*=</td><td>Multiply self by</td><td>x*=2</td><td>Via *</td></tr>
</table>

Tabulka operátorů

Category	Operator symbol	Operator name	Example	User overloadable
Assignment	/=	Divide self by	x/=2	Via /
	+=	Add self by	x+=2	Via +
	-=	Substract from self	x-=2	Via -
	<<=	Shift self left by	x<<=2	Via <<
	>>=	Shift self right by	x>>=2	Via >>
	&=	Add self by	x&=2	Via &
	^=	Exclusive-Or self by	x^=2	Via ^
	|=	Or self by	x |=2	Via |
Lambda	=>	Lambda	x => x+1	No

Příkazy - Statements

SIEMENS

- **Blok příkazů – { }**
  - Seskupení více příkazů
- **Deklarace**

```
string someWord = "rosebud";
```

  - Lokální deklarace

```
{
  int x;
  {
    int y;
    int x; // Error - x already defined
  }
  {
    int y; // OK - y not in scope
  }
  Console.WriteLine(y); // Error - y is out of scope
}
}
```

SIEMENS

Příkazy – Výrazy

```
// Declare variables with declaration statements:
string s;
int x, y;
System.Text.StringBuilder sb;

x = 1 + 2;                 // Assignment expression
x++;                       // Increment expression
y = Math.Max(x, 5);        // Assignment expression
Console.WriteLine(y);      // Method call expression
sb = new StringBuilder(); // Assignment expression

new StringBuilder();       // Object instantiation expression
new StringBuilder();       // Legal, but useless
new string('c', 3);        // Legal, but useless
x.Equals(y);               // Legal, but useless
```

SIEMENS

Příkazy – Selection

- Pro řízení programu
  - `if`, `switch`
  - Podmíněný operátor `? :`

```
if (5 < 2 * 3)
{
  Console.WriteLine("true");
  Console.WriteLine("Let's move on!");
}
else
{
  Console.WriteLine("False"); // False
}
```

SIEMENS

Příkazy – Selection

```
switch (cardNumber)
 {
   case 10:
   case 13:
     Console.WriteLine("King");
     break;
   case 12:
     Console.WriteLine("Queen");
     break;
   case 11:
     Console.WriteLine("Jack");
     break;
   case -1 :        // Joker is -1
     goto case 12; // In this game joker counts as queen
   default:         // Executes for any other cardNumber
     Console.WriteLine(cardNumber);
     break;
 }
```

Příkazy - Cykly

- **While**

```
int i = 0;
while (i < 3)
{
  Console.WriteLine(i);
  i++;
}
```

- **Do-while** – proběhne minimálně jednou

```
i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 3);
```

Příkazy - Cykly

SIEMENS

- **For**
  - `for(<init>; <condition>; <iteration>)`
    `<statement-or-statement-block>`
    - `<init>` – spustí se před začátkem cyklu
    - `<condition>` – pokud je true tak se provede statement(block)
    - `<iteration>` – spustí se po každé iteraci

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine(prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Příkazy - Cykly

- **Foreach**
  - Provede se pro každý prvek v Enumerable objektu (např. array, string)

```
foreach (char c in "beer") // c is the iteration variable
{
  Console.WriteLine(c);
}
```

SIEMENS

Příkazy - Skokové příkazy

- **Break**
  - Slouží k ukončení cyklu

```
int x = 0;
while (true)
{
  if (x++ > 5)
  {
    break; // break from the loop
  }
}
// execution continues here after break
```

SIEMENS

Příkazy - Skokové příkazy

- **Continue**
  - Slouží k ukončení jedné iterace cyklu

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0) // If i is even,
  {
    continue; // continue with next iteration
  }
  Console.Write(i + " ");
}
```

Příkazy - Skokové příkazy

- **Goto**
  - Přesune vykonávání programu na jiné umístění

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write(i + " ");
  i++;
  goto startLoop;
}
```

Příkazy - Skokové příkazy

SIEMENS

- **Return**
  - Vystoupí z metody a musí vracet návratový typ dle dané metody

```
public decimal Return(decimal d)
{
  decimal p = d * 100m;
  return p; // Return to the calling method with value
}
```

Příkazy - Skokové příkazy

- **Throw**
  - Slouží k vyhození vyjímky

```
private static void Throw(object obj)
{
  if (obj == null)
  {
    throw new ArgumentNullException("obj");
  }
}
```

Příkazy - ostatní

- **Using**
  - Pomocný příkaz pro elegantní volání `Dispose` pro **`IDisposable`**

```
using (var file = File.Open(@"c:\Filepath.txt",
                            FileMode.OpenOrCreate))
{
  file.Write(buffer, offset, count);
} // file.Dispose() is called here
```

- **Lock**
  - Slouží jako zkratka pro volání metod `Enter` a `Exit` třídy **`Monitor`**
  - Tzn. k zamykání kritické sekce při vícevláknovém zpracování

Jmenné prostory - Namespaces

- Seskupují třídy a rozhaní do pojmenovaných skupin
- Jmený prostor `System.Security.Cryptography` obsahuje např. třídu RSA
- Použití třídy z daného namespace
  - Plné jméno třídy

```
System.Security.Cryptography.RSA rsa =
              System.Security.Cryptography.RSA.Create();
```

  - Použití direktivy using

```
using System.Security.Cryptography;
public class Namespaces
{
  public void Method()
  {
    RSA rsa = RSA.Create(); // Don't need fully qualified name
  }
}
```

SIEMENS

Jmenné prostory - Definice

- **Klíčové slovo `namespace`**

```
namespace Outer.Middle.Inner
{
  class Class1 { ... }
  class Class2 { ... }
```

- Odpovídá zápisu

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 { ... }
      class Class2 { ... }
```

Jmenné prostory - Pravidla

- **Platnost jmen**
  - Názvy deklarované ve vnějším `namespace` jsou implicitně importovány do vnitřního `namespace`

```
namespace Outer
{
  namespace Middle
  {
    internal class Class1 { ... }

    namespace Inner
    {
      internal class Class2 : Class1 { ... }
    }
  }
}
```

SIEMENS

Jmenné prostory - Pravidla

- **Skrývání názvů**
  - Pokud se název objeví ve vnitřním i vnějším namespace vnitřní název „vyhrává“.

```
namespace Outer
{
  internal class Foo { ... }
  namespace Inner
  {
    internal class Foo { ... }

    internal class Test
    {
      private Foo f1; // = Outer.Inner.Foo
      private Outer.Foo f2; // = Outer.Foo
    }
  }
}
```

SIEMENS

Jmenné prostory - Pravidla

- **Opakování jmených prostorů**
  - Název daného `namespace` je možné opakovat dokud nedojde ke shodě v názvů typů uvnitř jmenného prostoru
  - Tzn. jeden `namespace` můžeme deklarovat na více místech

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
...
namespace Outer.Middle.Inner
{
  class Class2 { }
}
```

SIEMENS

Jmenné prostory - Pravidla

- **Vnořené using direktivy**
  - Direktivu using je možné zanořit do namespace a omezit tak rozsah použití importovaných názvů pro daný namespace.

```
namespace N1
{
  class Class1 { }
}
namespace N2
{
  using N1;
  class Class2 : Class1 { }
}
namespace N2
{
  class Class3 : Class1 { } // Compile-time error
}
```

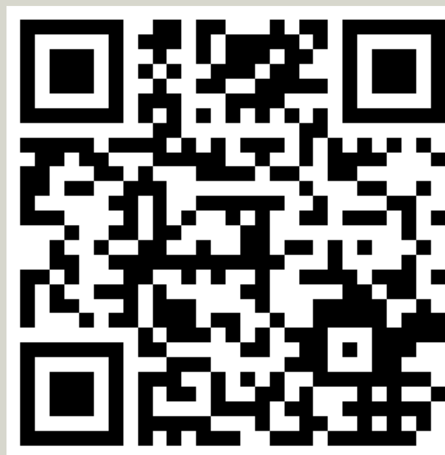
SIEMENS

Kontakt

Bc. Václav Pachta
Siemens CT DC / Sitraff Team

Olomoucká 7/9
618 00 Brno
Česká republika

E-mail: vaclav.pachta@siemens.com



SIEMENS

Reference

- http://www.amazon.com/5-0-Nutshell-The-Definitive-Reference/dp/1449320104