Programovací jazyk MiniC#

Radovan Šimek

2006

Obsah

1. Specifikace jazyka MiniC#

Jazyk MiniC# (čteno mini sí šárp) vznikl v souvislosti diplomovou prací autora tohoto dokumentu. Lze říci, že jazyk MiniC# je podmnožinou jazyky C#. Jazyk C# je moderní objektově orientovaný, typově bezpečný programovací jazyk, jež patří do rodiny jazyků vycházejících z jazyka C. Jazyk C# vznikl jako hlavní programovací jazyk pro platformu .NET společnosti Microsoft v roce 2001.

Z jazyka C# ve verzi 1.2 jsou do jazyka MiniC# vybrány všechny prvky potřebné pro objektově orientovaný jazyk a další konstrukty, které jsou nejpoužívanější a užitečné.

Zjednodušeně (a tedy pro znalé jazyka C#) by se dalo říci, že MiniC# obsahuje vše z jazyka C# kromě:

- členění typů do jmenných prostorů namespace,
- deklarací uživatelských hodnotových typů,
- deklarací vnořených typů,
- deklarací a tudíž i používání rozhranní,

- deklarací a tudíž i používání delegátů,
- používání uživatelských atributů,
- deklarací indexerů a přetížených operátorů,
- systému výjimek,
- deklarací a používání tzv. zubatých (jagged) polí, tedy polí, jejichž prvky jsou pole stejných dimenzí, ale potažmo různých rozsahů,
- některých méně obvyklých výrazů a příkazů, konkrétně `goto`, `typeof`, `checked`, `unchecked`, `lock`, `using`,
- používání identifikátorů se jmény shodnými s klíčovými slovy (tedy v C# s předponou `@`).

Dále MiniC# neobsahuje ty prvky, které v důsledku vynechání výše uvedených prvků nemají smysl, například neobsahuje modifikátory typů, nebo `extern` modifikátor pro funkční členy tříd.

1.1. Charakteristiky jazyka

Objekty

Jazyk MiniC# vykazuje čistě objektově orientované rysy. Není možné používat například globální funkce nebo proměnné. Plně podporuje dědičnost a polymorfismus. Všechny entity v paměti běžícího programu jsou *objekty*. Co se týče tříd, terminologie plně neodpovídá objektově orientovanému přístupu. Objekty jsou v MiniC# (z historických důvodů) instancemi *typů*. *Třídou* je nazýván uživatelsky definovaný tzv. odkazový typ (viz sekce 1.2.).

Typová bezpečnost

MiniC# je typově bezpečný jazyk. Znamená to, že nedovoluje uvést výraz daného typu na místě, kde je požadován výraz jiného typu. Naštěstí jazyk poskytuje řadu implicitních konverzí pro základní typy a také unifikovaný typový systém (viz sekce 1.2.). Typová bezpečnost umožňuje psát bezpečnější a robustnější programy a téměř veškeré typové kontroly probíhají jen při kompilaci programu.

Proměnné a jejich hodnoty

Proměnné představují místa v paměti označená symboly. Každá proměnná má přiřazený svůj datový typ. Díky typové bezpečnosti nelze do proměnné přiřadit hodnotu jiného typu. V jazyku MiniC# existuje několik kategorií proměnných: statické datové položky, instanční datové položky, prvky polí, hodnotové parametry, odkazové parametry, výstupní parametry a lokální proměnné.

V jazyce MiniC# je vyžadováno, aby proměnná byla definitivně přiřazena předtím, než je použita. Statické a instanční datové položky, prvky polí a hodnotové a odkazové parametry jsou vždy počátečně přiřazeny a mají svoji počáteční hodnotu. U ostatních typů kompilátor

analyzuje možné větve výpočtu a v místě použití hodnoty proměnné kontroluje, zda byla přiřazena v každé možné větvi programu vedoucího do tohoto místa.

Hodnotami proměnných hodnotových typů jsou přímo instance objektů. Hodnotami proměnných odkazových typů jsou odkazy (pointery, reference) na objekty uložené na paměťové haldě programu, nebo hodnota `null`, která představuje nepřítomnost instance.

Proměnným, které jsou považovány za počátečně přiřazené, je při jejich alokaci automaticky přiřazena počáteční hodnota. Pro odkazové typy je to hodnota `null`, pro hodnotové typy jsou jimi hodnoty `0`, `0.0`, `false`, `'\0'` (podle typu), tedy hodnoty, jež odpovídají vyplnění paměti objektu bitově nulami.

Automatická správa paměti

Jazyk MiniC# používá automatickou správu paměti, která oprošťuje autory programů od manuálního alokování a uvolňování paměti zabírané objekty. Automatickou správu paměti zajišťuje tzv. garbage collector (sběrač smetí).

Životní cyklus objektu z hlediska správce paměti je následující:

1. V okamžiku vzniku objektu mu správce paměti přidělí paměť, zavolá se konstruktor a objekt se považuje za živý.
2. Jestliže k objektu nelze přistoupit žádným možným pokračování běhu programu, objekt je považován za nepoužitelný a je tedy vhodný ke zrušení.
3. Po nějakém čase (nejčastěji v případě potřeby paměti pro nový objekt) je paměť přidělená objektu, který je určen ke zrušení, uvolněna.

1.2. Typový systém

Jazyk MiniC# používá unifikovaný typový systém. Znamená to, že typy tvoří hierarchickou strukturu s jediným kořenem tvořeným typem `object`. Vztahy mezi typy jsou dány vztahem dědičnosti. Jestliže jeden typ dědí z jiného typu, znamená to, že přebírá některé jeho tzv. členy. Některé členy mohou být potomkovi skryty. MiniC# podporuje jen jednonásobnou (jednoduchou) dědičnost. Znamená to, že každý typ (kromě typu `object`) má jen jednu přímou základní třídu. Typ `object` nemá žádnou základní třídu.

Typy se dělí do dvou kategorií: hodnotové typy a odkazové typy, podle toho, jak se s jejich instancemi zachází v paměti. Hodnotami hodnotových typů jsou přímo objekty, kdežto hodnotami odkazových typů jsou vždy reference na objekt uložený na haldě programu.

Hodnotové typy jsou dále děleny na jednoduché typy a na výčtové typy. Odkazové typy se dělí na třídy a pole. Pouze výčtové typy a třídy jsou uživatelsky definovatelné.

Jednoduché hodnotové typy

Jazyka MiniC# obsahuje několik předdefinovaných hodnotových typů, nazývaných jednoduché hodnotové typy. Tyto typy jsou identifikovány pomocí rezervovaných slov jazyka. Jednoduché hodnotové typy jazyka MiniC# jsou tyto:

`bool`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `decimal`

Jednoduché typy jsou běžnými typy, proto obsahují jednak zděděné, jednak vlastní členy. Navíc jazyk s nimi dovoluje pracovat i odlišně než s uživatelem definovanými typy:

- Hodnoty jednoduchých typů je možno zapisovat pomocí literálů.
- Výrazy obsahující pouze literály, konstantní výrazy, je možno spočítat již při překladu.

Následuje popis jednotlivých jednoduchých typů:

- Typ `byte` reprezentuje booleovské logické hodnoty `true` a `false`. Neexistují žádné konverze mezi tímto typem a číselnými typy.
- Typ `byte` reprezentuje 8-bitová celá čísla bez znaménka z rozsahu 0 až 255.
- Typ `short` reprezentuje 16-bitová celá čísla se znaménkem z rozsahu -32768 až 32767.
- Typ `int` reprezentuje 32-bitová celá čísla se znaménkem z rozsahu -2147483648 až 2147483647.
- Typ `long` reprezentuje 64-bitová celá čísla se znaménkem z rozsahu -9223372036854775808 až 9223372036854775807.
- Typ `char` reprezentuje 16-bitová celá čísla bez znaménka z rozsahu 0 až 65535. Hodnoty odpovídají hodnotám znaků Unicode.
- Typ `float` reprezentuje 32-bitová čísla v plovoucí čárce dle standartu IEEE 754. Reprezentuje reálná čísla v rozsahu 1.5×10^{-45} až 3.4×10^{38}, s přesností na 7 míst.
- Typ `double` reprezentuje 64-bitová čísla v plovoucí čárce dle standartu IEEE 754. Reprezentuje reálná čísla v rozsahu 5.0×10^{-324} až 1.7×10^{308}, s přesností na 15 míst.
- Typ `decimal` reprezentuje 128-bitová reálná čísla. Je vhodný pro finanční operace. Rozsah typu je 1.0×10^{-28} až 7.9×10^{28} s přesností na 28 míst.

Všechny výše uvedené typy dědí přímo z typu `object`.

Výčtové typy

Výčtové typy jsou samostatnými typy s pojmenovanými konstantami. Všechny výčtové typy mají svůj tzv. podložní (underlying) typ, kterým je typ `int`. Všechny výčtové typy dědí z typu `Enum`. Množina hodnot výčtového typu je stejná jako množina hodnot typu `int`. Hodnoty výčtového typu *nejsou* omezeny jen na hodnoty pojmenovaných konstant.

Odkazové typy

Hodnoty odkazových typů jsou reference na instance těchto typů, nazývaných objekty. Speciální hodnota `null` je kompatibilní se všemi odkazovými typy a reprezentuje nepřítomnost instance.

Třídy

Třídy umožňují deklarovat struktury obsahující datové členy (datové položky a konstanty) a funkční členy (konstruktory, metody, vlastnosti). Třídy plně podporují dědičnost.

Jazyk MiniC# obsahuje některé předdefinované třídy:

- Třída `object` představuje nejzákladnější typ všech ostatních typů.

- Třída `string` reprezentuje řetězce znaků Unicode. Hodnoty typu `string` mohou být zapsány pomocí literálů.
- Třída `Enum` reprezentuje základní třídu pro všechny výčtové typy, obsahuje pro ně některé společné členy.
- Třída `Array` reprezentuje základní třídu pro všechna pole, obsahuje pro ně některé společné členy.

Pole

Pole jsou datové struktury, které obsahují daní počet proměnných přístupných pomocí indexů. Jazyk MiniC# podporuje jedno i vícedimenzionální pole. Všechna pole dědí implicitně z třídy `Array`, která deklaruje některé společné vlastnosti. Pole nelze uživatelsky deklarovat, jazyk je implicitně deklaruje při použití. Pole se chovají jako jiné objekty, jazyk pro ně navíc nabízí speciální syntaxi pro přístup k jejich prvkům.

1.3. Lexikální struktura programu

Program v jazyku MiniC# se skládá z jednoho nebo více zdrojových souborů. Zdrojový soubor se skládá z posloupnosti znaků kódování Unicode. Zdrojový soubor obvykle odpovídá souboru v souborovém systému, ale tato příslušnost není vyžadována.

Řetězec znaků zdrojového souboru musí odpovídat dané lexikální struktuře. Ta je definována jako posloupnost lexikálních elementů jazyka. Lexikální elementy jazyka MiniC# jsou následující: konce řádků (line terminators), mezery (white space), komentáře (comments) a lexikální symboly (atomy, tokens).

Lexikální zpracování zdrojového souboru probíhá redukování posloupnosti znaků na posloupnost lexikálních symbolů, ostatní lexikální elementy slouží k oddělení lexikálních symbolů jazyka. Tato posloupnost lexikálních symbolů je vstupem pro syntaktickou analýzu.

Jestliže určitá posloupnost znaků zdrojového souboru odpovídá více lexikálním elementům, lexikální analyzátor vždy formuje nejdelší možný element.

Komentáře

Komentáře jsou v MiniC# dvou forem. Jednořádkový komentář začíná `//` a pokračuje až do konce řádku.

Ohraničený komentář začíná dvojicí znaků `/*` a končí dvojicí znaků `*/`. Ohraničené komentáře mohou zabírat více řádku, ale nelze je do sebe vnořovat.

Lexikální symboly – atomy

V jazyku MiniC# existuje několik druhů lexikálních symbolů, které tvoří základní atomy jazyka.

• Identifikátory

Identifikátor začíná znakem `_` (podtržítko) nebo písmenem. Na dalších místech mohou být číslice, písmena a některé další znaky, které bývají povoleny v identifikátorech, např. `_`. Identifikátor nemusí obsahovat jen znaky z anglické klávesnice, ale může obsahovat i národní znaky. Příklady identifikátorů jsou např. `i`, `j1`, `_temp` nebo `součet_12`.

• **Klíčová slova**

Klíčové slovo je sekvence znaků Unicode, které vypadá jako identifikátor, ale má speciální využití v jazyce a nemůže být použito jako identifikátor. Jazyk MiniC# obsahuje následující klíčová slova:

```
abstract  as        base      bool      break
byte      case      char      class     const
continue  decimal   default   do        double
else      enum      float     for       foreach
if        in        int       is        long
new       object    out       override  params
private   protected public    readonly  ref
return    short     static    string    switch
this      virtual   void      while     true
false     null
```

• **Booleovské literály**

Jazyk MiniC# obsahuje dva booleovské literály `true` (pravda) a `false` (nepravda), jež odpovídají klíčovým slovům. Typem (viz dále) literálu je typ `bool`.

• **Celočíselné literály**

Celá čísla lze zapsat ve zdrojovém programu pomocí dvou forem celočíselných literálů: v desítkové soustavě a v šestnáctkové soustavě. Celočíselné literály slouží pro zápis konstant celočíselných typů `byte`, `short`, `int` a `long`.

Desítkový celočíselný literál je složen alespoň z jednoho znaku `0` až `9`, po kterých může následovat přípona `l` nebo `L` pro určení rozsahu čísla. Šestnáctkový celočíselný literál je složen z předpony `0x` nebo `0X` následované alespoň jedním znakem šestnáctkových číslic, tedy `0` až `9`, `A` až `F` nebo `a` až `f`. Na konci může být opět přípona `l` nebo `L` pro určení, že typ literálu je `long`.

Hodnota literálu je určena obvyklým způsobem. Typ celočíselného literálu může být typ `int` (32 bitů) nebo `long` (64 bitů). Pokud je literál uveden na místě, kde je vyžadován např. typ `byte`, kompilátor provede automaticky konverzi.

• **Literály reálných čísel**

Literály reálných čísel slouží pro zápis konstant typů `float`, `double` a `decimal`. Tyto literály se skládají z celé části (posloupnost znaků `0` až `9`), desetinné tečky `.`, desetinné části (posloupnost znaků `0` až `9`), exponentu (znak `E` nebo `e` následovaný hodnotou exponentu) a přípony `f`, `F`, `d`, `D`, `m`, `M`. Všechny části mohou být v určitých situacích vynechány.

Hodnota literálu je určena obvyklým způsobem. Typ tohoto literálu je určen následujícím způsobem: Jestliže přípona není uvedena, typem literálu je typ `double`. Je-li přípona `f` nebo `F`, typem literálu je typ `float`. Je-li přípona `d` nebo `D`, typem literálu je typ `double`. Je-li přípona `m` nebo `M`, typem literálu je typ `decimal`. Není-li možné hodnotu literálu reprezentovat určeným typem, nastane chyba při překladu.

Příklady literálů reálných čísel jsou např.

`10.0` (`double`), `10f` (`float`), `15.465m` (`decimal`), `1E-5` (`double`), `1.45e-5m` (`decimal`).

• **Znakové literály**

Znakové literály reprezentují jednotlivé Unicode znaky. Typem tohoto literálu je typ `char`.

Znakový literál se skládá z dvojice apostrofů, mezi nimiž je uzavřen jediný znak, nebo úniková sekvence. Znakem může být libovolný znak, kromě apostrofu a znaků konce řádku. Únikové sekvence jsou následující:

Úniková sekvence	**Název znaku**	**Hexadecimálně**
`\'`	apostrof	0x0027
`\"`	uvozovka	0x0022
`\\`	obrácené lomítko	0x005C
`\0`	prázdný znak	0x0000
`\a`	zvonek	0x0007
`\b`	klávesa zpět	0x0008
`\f`	nová stránka	0x000C
`\n`	nový řádek	0x000A
`\r`	návrat vozíku	0x000D
`\t`	horizontální tabulátor	0x0009
`\v`	vertikální tabulátor	0x000B
`\x`*HHHH*	znak s hexadecimálním kódem	0x*HHHH*
`\u`*HHHH*	znak s Unicode kódem	0x*HHHH*

Příklady literálů reálných čísel jsou např.
`'a'`, `'*'`, `'\''`, `'\r'`, `'\xF'`, `'\uFC8A'`.

• Řetězcové literály

MiniC# podporuje dvě formy řetězcových literálů: obvyklý řetězcový literál a doslovný řetězcový literál.

Obvyklý řetězcový literál je složen z dvojice znaků `"` (uvozovka), mezi nimiž je řetězec znaků nebo únikových sekvencí. Význam únikových sekvencí je stejný jako u znakových literálů.

Doslovný řetězcový literál začíná znakem `@`, následovaným uvozovkou. Následuje posloupnost libovolných Unicode znaků, včetně mezer a konců řádků, jež jsou interpretovány doslovně. Jedině dvojice znaků `""`, představuje únikovou sekvenci pro uvozovku. Na konci je znak uvozovka `"`. Únikové sekvence v doslovném literálu nejsou zpracovávány. Doslovný řetězcový literál může zabírat více řádků.

Příklady řetězcových literálů jsou např.

```
"Jazyk MiniC#",
"Jazyk \r\n MiniC#",
"C:\\dokumenty\\CSharp",
@"C:\dokumenty\CSharp",

@"První řádek
Druhý řádek

Čtvrtý řádek"
```

• Operátory a znaménka

Operátory jsou používány ve výrazech, které zahrnují více operandů. Znaménka slouží jako spojovače nebo oddělovače. Jazyk MiniC# obsahuje následující operátory a znaménka:

```
{    }    [    ]    (    )    .    ,    :    ;
+    -    *    /    %    &    |    ^    !    ~
=    <    >    ?    ++   --   &&   ||   <<   >>
==   !=   <=   >=   +=   -=   *=   /=   %=   &=
|=   ^=   <<=  >>=
```

1.4. Syntaktická struktura programu

Zdrojový text programu

MiniC# dovoluje uchovávat zdrojový text programu ve více souborech. Při překladu jsou všechny zdrojové soubory zpracovávány dohromady, zdrojové soubory mohou volně referencovat ostatní soubory. Toto chování je stejné, jako by se zpracovával jediný zdrojový soubor vzniklý zřetězením všech zdrojových souborů. Předsunuté deklarace nejsou v MiniC# potřeba, neboť pořadí deklarací je, až na několik výjimek, bezvýznamné.

Základními koncepty programu v jazyce MiniC# jsou typy a jejich členy. Typy se definují pomocí deklarací. Program může obsahovat deklarace pouze pro výčtové typy a třídy. To ovšem nic nemění na tom, že je možno používat předdefinované hodnotové a odkazové typy.

Deklarace tříd mohou obsahovat tzv. funkční členy (metody, vlastnosti, konstruktory), uvnitř jejich těl se vyskytují příkazy jazyka, jež vyjadřují akce prováděné funkčním členem. Jazyk MiniC# obsahuje poměrně velké množství příkazů.

Součástí příkazů jsou výrazy, jež představují prováděné výpočty. Výrazy se skládají z operátorů a operandů. Operandy mohou být jednoduché výrazy (jména, literály) nebo další složené výrazy.

Vstupní bod programu

Běh programu začíná aktivací určené metody, tzv. vstupního bodu programu. Vstupním bodem programu může být metoda z libovolné třídy s jednou z následujících hlaviček:

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

1.5. Třídy

Třídy jsou nejpodstatnější typy v jazyku MiniC#. Obsahují datové členy a funkční členy v jedné struktuře. Navíc umožňují používat v plné míře dědičnost a polymorfismus. Třídy poskytují definici pro dynamicky vytvářené instance – objekty.

Třídy vznikají pomocí deklarace třídy. Deklarace třídy se skládá z hlavičky a z těla třídy. Hlavička třídy obsahuje klíčové slovo `class`, před nímž může být modifikátor abstraktní třídy `abstract`. Dále obsahuje identifikátor určující jméno třídy a dále volitelně dvojtečku a jméno základní třídy. Tělo třídy se skládá z deklarací členů třídy (viz dále) uzavřených mezi složenými závorkami.

[modifikátor] `class` *jméno-třídy [* `:` *základní-třída]*
`{` *seznam-deklarací-členů* `}`

Jestliže základní třída není uvedena, základní třídou deklarované třídy se stává třída `object`. Základní třídou nemohou být předdefinované třídy `Array`, `Enum`, nebo `string`. Dědičná hierarchie tříd nesmí obsahovat definici kruhem. Tedy pokud třída `A` dědí z třídy `B`, třída `B` nemůže (ani nepřímo) dědit z třídy `A`.

Třída může obsahovat následující druhy členů:

- Konstanty – konstantní hodnoty asociované s třídou.
- Datové položky – hodnoty tvořící stav objektu, nebo hodnoty asociované s třídou.
- Metody – výpočty a akce, které mohou být prováděny objekty, nebo jsou asociované s třídou.
- Vlastnosti – akce umožňující přístup k charakteristikám (vlastnostem) objektů, nebo asociovaných s třídou.
- Konstruktory – akce potřebné pro inicializaci objektů nebo třídy.

Všechny druhy členů mají přiřazenu svoji přístupnost, která určuje oblasti programu, kde je člen přístupný k použití. Existují tři možné přístupnosti:

- `public` – přístup není nijak omezen.
- `protected` – přístup je povolen jen v rámci deklarace třídy a v třídách zděděných z této třídy.
- `private` – přístup je povolen jen v rámci deklarace třídy.

V deklaracích členů se mohou vyskytovat různé modifikátory. Vždy platí, že žádný modifikátor nesmí být uveden vícekrát. Modifikátory mohou obsahovat jen jeden z modifikátorů přístupu `public`, `protected`, `private`. Pokud žádný modifikátor přístupu není uveden, člen má přístupnost `private`.

Konstanty

Konstanty jsou členy, jež reprezentují konstantní hodnoty asociované s třídou, které lze spočítat v době překladu.

Jedna deklarace může definovat více konstant se stejným typem a modifikátorem a má následující tvar:

[modifikátor] `const` *typ jméno*$_1$`=`*výraz*$_1$ *[*`,...,` *jméno*$_n$`=`*výraz*$_n$*]*`;`

Modifikátorem může být jeden z modifikátorů přístupnosti. Ačkoliv je konstanta statický člen, není povolen modifikátor `static`.

Typ konstanty není nijak omezen. Avšak výraz určující hodnotu konstanty musí být konstantní výraz, tedy určení jeho hodnoty je možné již při překladu.

Datové položky

Datové položky jsou členy, jež reprezentují stav objektu, nebo (pokud jsou statické) reprezentují hodnoty asociované s třídou.

Jedna deklarace může deklarovat více datových položek se stejným typem a modifikátory a má následující tvar:

[modifikátory] typ jméno$_1$*[*`=`*inicializátor*$_1$*]* [`,` ... `,` *jméno*$_n$*[*`=`*inicializátor*$_n$*]]*`;`

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static` a `readonly`.

Inicializátor je nepovinný, ale pokud je uveden specifikuje počáteční hodnotu datové položky. Inicializátorem může být buď libovolný výraz, nebo inicializátor pole. Typ inicializátoru musí být implicitně konvertovatelný na typ položky.

Statické a instanční datové položky

Jestliže deklarace datové položky obsahuje modifikátor `static`, jedná se o statickou datovou položku. Pokud deklarace tento modifikátor neobsahuje, jedná se instanční datovou položku. Statické položky nejsou součástí instancí třídy, reprezentují právě jedno místo v paměti běžícího programu. Naopak instanční položky jsou vždy spojeny s instancemi třídy. Každý objekt obsahuje svoji sadu instančních položek třídy.

Datové položky jen pro čtení

Pokud deklarace datové položky obsahuje modifikátor `readonly`, jedná se o tzv. položku jen pro čtení. Přiřazení hodnoty do položky jen pro čtení je povoleno pouze v deklaraci inicializátorem nebo v konstruktoru. Položky jen pro čtení je možno používat podobně jako konstanty. Hodnota se jim však přiřazuje až za běhu programu. Jejich hodnoty tedy mohou být i dynamicky alokované objekty. Navíc hodnotu je možné nastavit např. pomocí parametrů konstruktoru.

Metody

Metody implementují výpočty nebo akce, jež mohou být prováděny objekty, nebo (v případě statických metod) asociované s třídou.

Deklarace metody má následující tvar:

[modifikátory] návratový-typ jméno `(` *[seznam-formálních-parametrů]* `)`
tělo-metody

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static`, `virtual`, `override`, `abstract`. Povoleny jsou jen některé kombinace modifikátorů. Návratový typ metody určuje typ hodnoty počítané a navrácené metodou. Pokud metoda nevrací hodnotu, návratovým typem je klíčové slovo `void`. Seznam formálních parametrů může být prázdný.

Tělo metody je tvořeno u abstraktních metod pouze středníkem, u ostatních metod blokem příkazů (viz sekce 1.10.).

Jméno metody a seznam jejích formálních parametrů tvoří *signaturu* metody. Jméno metody musí být odlišné od jmen všech ostatních členů třídy, které nejsou metody. Signatura metody musí být odlišná od všech ostatních signatur metod se stejným jménem.

Formální parametry metody

Formální parametry se používají pro předání aktuálních hodnot argumentů při aktivaci metody. Existují čtyři druhy formálních parametrů – *hodnotové* parametry, *odkazové* parametry, *výstupní* parametry a *pole parametrů*. Formální parametry jsou v seznamu odděleny čárkami. Pole parametrů může být uvedeno jen na konci seznamu.

Každé vyvolání metody si vytváří svoji vlastní kopii parametrů.

- **Hodnotové parametry** odpovídají lokálním proměnným metody s počáteční hodnotu danou při aktivaci metody hodnotou příslušného argumentu. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  typ jméno

- **Odkazové parametry** nevytvářejí novou proměnnou, ale reprezentují proměnnou předanou jako příslušný argument. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  `ref` *typ jméno*

  Proměnná předávána jako odkazový parametr musí být v místě aktivace metody definitivně přiřazena (viz sekce 1.1.).

- **Výstupní parametry** také nevytvářejí novou proměnnou a opět reprezentují proměnnou předanou jako příslušný argument. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  `out` *typ jméno*

  Na rozdíl od odkazového parametru, proměnná předávána jako výstupní parametr nemusí být v místě aktivace metody definitivně přiřazena. Výstupní parametr musí být definitivně přiřazen před ukončením metody.

- **Pole parametrů** umožňuje předat metodě libovolný počet parametrů. Deklarace je následující:

  `params` *typ-pole jméno*

  Typ tohoto druhu parametru musí být jednodimenzionální pole. Uvnitř metody se tento parametr chová jako běžný parametr s daným typem pole. Při aktivaci metody je ovšem možné místo jediného parametru uvést libovolný počet argumentů, jejichž typ je konvertovatelný na typ prvků pole.

Statické a instanční metody

Jestliže deklarace metody obsahuje modifikátor `static`, potom je metoda statická. V opačném případě je metoda instanční. Samotnou instanci, nad níž je instanční metoda prováděna je možno získat uvedením klíčového slova `this`.

Virtuální metody

Instanční metody mohou být virtuální. Metoda je virtuální, pokud její deklarace obsahuje modifikátor `virtual`. Při vyvolání virtuální metody rozhoduje skutečný (za běhu programu) typ instance, nad níž je volání prováděno, která implementace metody bude provedena. U nevirtuálních metod je metoda pro provedení jednoznačně určena již v době překladu na základě typu výrazu uvedeného v programu, který představuje instanci.

Přepsané (override) metody

Virtuální metoda může být ve zděděné třídě přepsána (potlačena, overridden). Jestliže deklarace metody obsahuje modifikátor `override`, pak tato metoda přepisuje (potlačuje) virtuální metodu se stejnou signaturou v některé ze základních tříd. Pokud tato tzv. základní metoda neexistuje, nebo její hlavička plně neodpovídá hlavičce přepsané metody, nastane chyba při překladu. Zatímco virtuální metoda vytváří novou metodu, přepsaná metoda specializuje virtuální metodu poskytnutím nové implementace.

Abstraktní metody

Instanční metoda, jejíž deklarace obsahuje modifikátor `abstract` je abstraktní metoda. Abstraktní metoda je virtuální metoda, která ovšem neposkytuje počáteční implementaci. Z tohoto důvodu tělo abstraktní metody tvoří středník. Abstraktní metody se mohou nacházet pouze v abstraktních třídách. Zděděná třída, která není abstraktní, musí přepisovat všechny zděděné abstraktní metody.

Přetěžování (overloading) metod

Přetěžování metod umožňuje v deklaraci třídy uvést několik metod se stejným jménem, pokud mají navzájem jiné signatury. Při překladu aktivace metody s daným jménem kompilátor vybere nejvhodnější metodu, nebo pokud nelze nejvhodnější metoda nalézt, nastane chyba při překladu. Nejvhodnější metoda je vybírána tak, aby si nejlépe odpovídaly formální parametry metody a aktuální parametry uvedené při aktivaci.

Vlastnosti

Vlastnosti jsou členy tříd, jež poskytují přístup k charakteristikám objektů nebo tříd. Vlastnosti jsou přirozeným rozšířením datových položek. Oboje jsou pojmenovanými členy a syntaxe pro přístup k nim je stejný. Nicméně vlastnosti neoznačují místa v paměti. Namísto toho vlastnosti obsahují *akcesory*, jež určují příkazy, které se mají provést při čtení nebo zapisování vlastnosti.

Deklarace vlastnosti je následující:

[modifikátory] typ jméno
`{`
  `get` *tělo-akcesoru*$_1$
  `set` *tělo-akcesoru*$_2$
`}`

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static`, `virtual`, `override`, `abstract`. Povoleny jsou jen některé kombinace modifikátorů, a to stejně jako u metod. Pořadí akcesorů `get` a `set` může být i opačné, deklarace taktéž nemusí obsahovat oba akcesory, vždy ale alespoň jeden. Na začátku akcesorů jsou vyžadovány

identifikátory `get` nebo `set`, ačkoliv to nejsou klíčová slova. U abstraktních vlastností těla akcesorů tvoří pouze středník, v ostatních případech blok příkazů (viz sekce 1.10.).

Akcesor `get` (getter) odpovídá bezparametrické metodě, jejíž návratový typ je stejný jako typ vlastnosti. Getter je volán při získávání hodnoty vlastnosti.

Akcesor `set` (setter) odpovídá metodě s jedním hodnotovým parametrem se jménem `value`, jehož typ je stejný jako typ vlastnosti. Tento akcesor nemá návratovou hodnotu. Setter je volán s argumentem, jež představuje novou hodnotu vlastnosti.

Statické a instanční vlastnosti

Podobně jako metody a datové položky i vlastnosti mohou být statické a instanční. Vlastnost, jejíž modifikátory obsahují modifikátor `static` je statická, v opačném případě je instanční.

Virtuální, přepsané a abstraktní vlastnosti

Stejně jako metody mohou být i vlastnosti virtuální, přepsané a abstraktní, jestliže jejich deklarace obsahuje modifikátory `virtual`, `override`, nebo `abstract`. Mechanismus virtuálních metod se přenáší z vlastnosti na její akcesory se stejným chováním jako metody. Při přepisování vlastnosti ze základní třídy, která má oba akcesory je možné přepisovat pouze jeden z nich, nebo oba dva.

Instanční konstruktory

Instanční konstruktory jsou členy třídy, které implementují akce potřebné pro inicializaci instance třídy. Deklarace instančního konstruktoru je následující:

[modifikátor] jméno-třídy `(` *[seznam-formálních-parametrů]* `)` *[inicializátor]*
tělo-konstruktoru

Modifikátorem může být jeden z modifikátorů přístupnosti. Jméno konstruktoru musí být stejné jako jméno obsahující třídy. Seznam formálních parametrů může být prázdný a má stejný tvar i význam jako u metod. Tělo konstruktoru je tvořeno blokem příkazů.

Inicializátor konstruktoru nemusí být vždy uveden, ale pokud uveden je, má jeden z následujících tvarů:

`: base (`*[seznam-argumentů]* `)`

nebo

`: this (` *[seznam-argumentů]*`)`

Inicializátor konstruktoru určuje jiný instanční konstruktor a jeho argumenty, který se má vyvolat před provedením těla konstruktoru. Daným konstruktorem může být konstruktor ze stejné třídy (se slovem `this`) nebo z přímé základní třídy (se slovem `base`). Pokud inicializátor není uveden, překladač automaticky přidává volání bezparametrického konstruktoru základní třídy. Překladač hledá nejvhodnější konstruktor podobně jako u přetížených metod pomocí uvedených argumentů. Pokud žádný odpovídající konstruktor není nalezen, nastává chyba při překladu.

Seznam argumentů může být prázdný. Pokud není prázdný, obsahuje čárkami oddělené argumenty. Jejich popis je uveden u výrazu volání metody (viz sekce 1.9.).

Deklarace třídy, podobně jako u metod, nemůže obsahovat dva instanční konstruktory se stejnou signaturou. Instanční konstruktory se nedědí ze základních tříd.

Implicitní konstruktor

Pokud třída nedeklaruje žádný instanční konstruktor, tzv. implicitní konstruktor je automaticky vygenerován. Implicitní konstruktor nemá parametry a pouze volá bezparametrický konstruktor přímé základní třídy. Jestliže třída je abstraktní, pak implicitní konstruktor obsahuje modifikátor `protected` v opačném případě modifikátor `public`.

Statický konstruktor

Statický konstruktor je člen třídy, který obsahuje akce potřebné pro inicializaci třídy. Třída může deklarovat nejvýše jeden statický konstruktor a jeho deklarace má vždy tvar:

`static` *jméno-třídy* `( )` *tělo-konstruktoru*

Statický konstruktor vždy obsahuje modifikátor `static` a nemůže obsahovat žádný modifikátor přístupnosti, neboť tento konstruktor nelze explicitně volat. Statický konstruktor je volán buď při vytváření prvního objektu třídy, nebo při prvním přístupu k libovolnému statickému členu třídy. Jméno statického konstruktoru musí být opět stejné jako jméno obsahující třídy. Tělo konstruktoru tvoří blok příkazů.

1.6. Pole

Pole jsou datové struktury obsahující daný počet proměnných, ke kterým se přistupuje pomocí indexů. Všechny proměnné mají stejný typ a nazývají se *elementy* pole. Typ elementů může být jakýkoliv typ, kromě polí. Pole jsou odkazové typy, tudíž jsou vždy alokovány na hromadě programu.

Každé pole má svoji dimenzi, která určuje počet indexů pro přístup k prvkům. Dimenze je součásti typu polí. Pole tedy mohou být jednodimenzionální a vícedimenzionální.

Každá dimenze pole má asociovanou délku, která se samozřejmě nezáporná. Délky dimenzí nejsou součásti typu pole, ale jednotlivých instancí a jsou určeny při vzniku pole za běhu programu. Délky dimenzí jsou po celý život instance pole neměnné. Indexy v jednotlivých dimenzích jsou vždy v rozmezí 0 až délka dimenze-1.

Typy polí se zapisují následujícím způsobem:

typ-elementů `[ ,,, ]`

Počet čárek uvnitř hranatých závorek zvýšený o jedna určuje dimenzi pole. Např. jednodimenzionální pole s elementy typu `int` se zapíše `int[]`, třídimenzionální pole typu `string` se zapíše `string[,,]`.

Pole jsou vytvářeny pomocí výrazů s operátorem `new`, k prvkům pole se přistupuje pomocí výrazu přístupu k prvku pole. Oba výrazy viz sekce 1.9.

Pole nelze explicitně deklarovat jako třídy nebo výčty, neboť mají pevně definovanou strukturu. Pole deklaruje kompilátor automaticky při použití. Každý typ pole automaticky dědí z třídy `Array`, a tudíž přebírá její členy.

Inicializátor pole

V deklaraci datové položky, v deklaraci lokální proměnné (viz sekce Příkazy 1.10.) nebo ve výrazu vytvoření pole je možné uvést inicializátor pole, který specifikuje počáteční hodnoty všech elementů pole. Inicializátor má následující tvar.

`{` *seznam-inicializátorů-proměnných* `}`

Inicializátory proměnných jsou v seznamu odděleny čárkami. Každý inicializátor proměnné je buď libovolný výraz, nebo v případě vícedimenzionálních polí, vnořený inicializátor pole. Deklarace nebo výraz vytvoření pole, v němž se inicializátor nachází, určuje typ elementů a dimenzi pole pro inicializátor.

Pro jednodimenzionální pole musí inicializátor obsahovat seznam výrazů, které lze přiřadit do proměnné typu elementů pole. Počet výrazů v inicializátoru určuje délku dimenze pro instanci. Inicializátor pro pole typu `int[]` může vypadat takto:

```
{0, 2, 4, 6, 8}
```

Pro vícedimenzionální pole musí inicializátor obsahovat tolik úrovní vnoření inicializátorů, kolik je dimenzí pole. Délku každé dimenze určuje počet prvku v dané úrovni zanoření. Pro vnořené inicializátory na stejné úrovní vnoření musí platit, že mají stejný počet prvků. Následující inicializátor inicializuje pole s typem `int[,]` s délkami dimenzí 5 a 2.

```
{{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}}
```

1.7. Výčtové typy

Výčtové typy (výčty) jsou samostatné hodnotové typy, které deklarují množinu pojmenovaných (celočíselných) konstant, jejichž typ je typ deklarovaného výčtu. Hodnoty, jež může proměnná s výčtovým typem nabývat nejsou omezeny jen na deklarované hodnoty, ale mohou to být všechny hodnoty tzv. podložního (underlying) typu `int`.

Výčtové členy automaticky dědí z předdefinované třídy `Enum` a tudíž přebírají její členy.

Výčtové typy vznikají pomocí jejich deklarací. Deklarace výčtu se skládá z hlavičky a z těla výčtového typu. Hlavička obsahuje klíčové slovo `enum` následované identifikátorem, který určuje jméno třídy. Tělo výčtu se skládá z deklarací členů výčtu (viz dále) uzavřených mezi složenými závorkami.

`enum` *jméno-výčtu*
`{` *seznam-deklarací-členů* `}`

Seznam deklarací členů může být prázdný. Pokud není prázdný, obsahuje deklarace členů výčtu oddělené čárkami.

Členy výčtů

Členy výčtu představují pojmenované konstanty výčtového typu. Každý člen má asociovanou konstantní hodnotu. Typem této konstanty je podložní typ `int`. Deklarace členu typu je následující:

jméno [= konstantní-výraz]

Žádné dva členy nemohou mít stejné jméno. Více členů může sdílet stejnou konstantní hodnotu. Pokud v deklaraci není uveden výraz určující konstantní hodnotu, hodnotu přiřadí překladač automaticky následovně:

- Pokud je daný člen prvním deklarovaným členem ve výčtu, jeho konstantní hodnota bude 0.
- Pokud daný člen není první, jeho hodnota bude rovna hodnotě předchozího členu zvýšené o 1.

Následující kousek kódu ukazuje deklaraci výčtového typu:

```
enum Color
{
  Red,        // hodnota 0
  Green = 5,  // hodnota 5
  Blue,       // hodnota 6
  Max = Blue  // hodnota 6
}
```

1.8. Konverze

Výrazy v jazyce MiniC# mají svůj typ (z hlediska typového systém). Konverze umožňují, aby výraz jednoho typu mohl být použit tam, kde se očekává výraz jiného typu. Konverze se dělí na *implicitní* a *explicitní*. Je-li konverze implicitní, pak nemusí být v programu přímo (explicitně) uvedena a konverzi provede kompilátor. Explicitní konverze umožňuje v programu konvertovat i výrazy, jež není kompilátor schopen implicitně konvertovat.

Implicitní konverze

Následující typy konverzí jsou prováděny implicitně:

- **Identická konverze**

  Identická konverze konvertuje výraz jednoho typu do toho samého typu. Má-li výraz požadovaný typ, pak je do tohoto typu konvertovatelný.

- **Implicitní numerické konverze**

  Výrazy následujících numerických typů lze implicitně konvertovat do cílových typů:

  - Z typu `byte` do typů `short`, `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `short` do typů `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `int` do typů `long`, `float`, `double`, `decimal`.
  - Z typu `long` do typů `float`, `double`, `decimal`.
  - Z typu `char` do typů `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `float` do typu `double`.

  Konverze z typu `int` a `long` do typu `float` a z typu `long` do typu `double` mohou ztratit přesnost. Implicitní numerické konverze ovšem nikdy neztrácejí informaci.

- **Implicitní výčtová konverze**

  Výraz, který odpovídá literálu `0`, lze konvertovat do libovolného výčtového typu.

- **Implicitní konverze odkazových typů**

  Výrazy následujících odkazových typů lze implicitně konvertovat do cílových typů.

  - Z každého odkazového typu do typu `object`.
  - Z každé třídy `S` do třídy `T`, pokud `S` dědí z `T`.

  - Z každého pole do typu `Array`.
  - Výraz `null` do jakéhokoliv odkazového typu.

  Tento typ konverze může změnit typ odkazu, ale nikdy nemění ani typ, ani hodnoty referencovaného objektu.

- **Boxování (boxing)**

  Boxování umožňuje výraz jakéhokoliv hodnotového typu konvertovat na typ `object`. Konverze probíhá tak, že je na haldě programu alokován nový objekt a do něj je zkopírována hodnota výrazu.

- **Implicitní konverze konstantních výrazů**

  Konstantní výraz typu `int` je možné konvertovat na typ `byte` nebo `short`.

Explicitní konverze

Explicitní konverze jsou povoleny ve výrazu přetypování. Explicitní konverze jsou takové konverze, které nemusejí vždy uspět, nebo hodnoty výrazů při nich mohou ztrácet informaci, nebo konvertují natolik rozdílné typy, že si zaslouží explicitní notaci.
Následující typy konverzí je možné provést explicitně:

- Všechny implicitní konverze.
- **Explicitní numerické konverze** Explicitní numerické konverze jsou konverze mezi numerickými typy, pro něž neexistuje implicitní konverze:

  - Z typu `byte` do typu `char`.
  - Z typu `short` do typů `byte` nebo `char`.
  - Z typu `int` do typů `byte`, `short` nebo `char`.
  - Z typu `long` do typů `byte`, `short`, `int` nebo `char`.
  - Z typu `char` do typů `byte` nebo `short`.
  - Z typu `float` do typů `byte`, `short`, `int`, `long`, `char` nebo `decimal`.
  - Z typu `double` do typů `byte`, `short`, `int`, `long`, `char`, `float` nebo `decimal`.
  - Z typu `decimal` do typů `byte`, `short`, `int`, `long`, `char`, `float` nebo `double`.

  Explicitní numerické konverze mohou ztrácet informace čísel. Při konverzi z většího celočíselného typu vždy dochází k ořezání hodnoty na určený počet bitů. Při konverzi z typů `float` a `double` do celočíselných typů dochází k zaokrouhlení. Při konverzi z typu `double` do typu `float` dochází k zaokrouhlení na nejbližší hodnotu typu `float`. Při konverzi z typů `float` nebo `double` do typu `decimal`, je zdrojová hodnota převedena do reprezentace `decimal` a zaokrouhlena na nejbližší číslo s přesností 28 desetinných míst.

- **Explicitní výčtové konverze**

  Explicitní výčtové konverze jsou následující:

  - Z typů `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `decimal` do jakéhokoliv výčtového typu.
  - Z jakéhokoliv výčtového typu do typů `byte`, `short`, `int`, `long`, `char`, `float`, `double` nebo `decimal`.
  - Z jakéhokoliv výčtového typu do jiného výčtového typu.

  Tyto konverze probíhají tak, že hodnota výčtu je nahrazena hodnotou typu `int` a následně provedena příslušná konverze mezi typem `int` a druhým typem.

- **Explicitní konverze odkazových typů**

  Explicitní konverze odkazových typů jsou následující:

  - Z typu `object` do jakéhokoliv odkazového typu.
  - Z třídy `S` do třídy `T`, pokud `S` je základní třída `T`.
  - Z třídy `Array` do libovolného pole.

  Při této konverzi je potřeba provádět kontrolu typu objektu za běhu programu. Explicitní odkazová konverze uspěje, pokud je výraz `null`, nebo skutečný typ objektu, jež představuje operand konverze, je konvertovatelný do cílového typu pomocí implicitní odkazové konverze. Při neúspěchu konverze je vykonávání programu ukončeno chybou.

- **Odboxování (unboxing)**

  Odboxování povoluje explicitní konverzi z typu `object` na libovolný hodnotový typ. Konverze obsahuje kontrolu, zda operand je boxovaná hodnota daného hodnotového typu, v případě úspěchu je hodnota zkopírována z instance. Při neúspěchu konverze je vykonávání programu ukončeno chybou.

1.9. Výrazy

Výrazy jsou konstruovány z operandů a operátorů. Operátory určují, jakou operaci provést s operandy. Operátory jsou například `+`,`-`, `*`, `/` a `new`. Operandy mohou být například literály, datové položky, lokální proměnné, nebo další výrazy.

Pořadí vyhodnocování operátorů ve výrazech je dáno precedencí a asociativitou operátorů. Pokud se operand vyskytne mezi dvěma operátory, je nejdříve provedena operace s vyšší precedencí. Pokud je precedence operátoru stejná, rozhoduje asociativita operátoru – zleva nebo zprava. Například výraz `x + y * z` je vyhodnocen jako `x + (y * z)` neboť operátor `*` má vyšší prioritu než `+`. Výraz `x + y - z` je vyhodnocen jako `(x + y) - z` neboť `+` a `-` mají stejnou precedenci, ale jsou levě asociativní. Naopak výraz `x = y = z` je vyhodnocen jako `x = (y = z)`, neboť operátor `=` je asociativní zprava. Všechny binární operátory kromě přiřazení jsou asociativní zleva. Unární operátory a podmíněný výraz jsou asociativní zprava.

Následující části popisují druhy výrazů jazyka MiniC# v pořadí od nejvyšší priority:

Primární výrazy

Literály

Všechny literály uvedené v lexikální struktuře (1.3.) jsou primární výrazy.

Jednoduché jméno (identifikátor)

Jednoduché jméno je složeno pouze z identifikátoru. Význam jména je určen postupným hledáním deklarací se stejným jménem v aktuálním bloku příkazů, v nadřazených blocích, deklaraci funkčního členu, vyhledáním členu v obsaženém typu, nebo jména deklarovaného typu.

Uzávorkovaný výraz

Uzávorkovaný výraz tvaru `(` *výraz* `)` se používá pro potlačení pravidel precedence a asociativity operátorů. Výraz uvnitř závorek nesmí představovat typ nebo metodu.

Přístup k členu

Přístup k členu může mít jeden z následujících tvarů:

primární-výraz `.` *identifikátor*
předdefinovaný-typ `.` *identifikátor*
`base` `.` *identifikátor*

Předdefinovaným typem je jeden z typů `bool`, `byte`, `char`, `decimal`, `double`, `float`, `int`, `long`, `object`, `short`, `string`.

Klíčové slovo `base` slouží k přístupu k překrytým nebo přepsaným členům základní třídy.

Člen, jež je představován výrazem je určen mechanismem výběru členu (viz níže) v rámci typu, který přestavuje část výrazu před tečkou nebo v rámci typu výrazu před tečkou. Pokud část výrazu před tečkou určuje typ a ne hodnotu nebo proměnnou, potom vybraný člen musí být statický, v opačném případě musí být instanční.

Pokud je při běhu programu primární výraz vyhodnocen na `null`, k členu nelze přistoupit a program je ukončen chybou.

Výběr členu je proces, kterým je určován význam jména v rámci některého typu. Výběr členu nastává jako část vyhodnocení výrazů přístup k členu a jednoduché jméno. Výběr členu se jménem `N` v typu `T` probíhá následovně:

1. Získá se množina přístupných členů se jménem `N` deklarovaných v `T` a jeho základních typech. Členy s modifikátorem `override` jsou vynechány. Pokud je množina prázdná, člen se jménem `N` neexistuje.

2. Členy, které jsou skryté se taktéž vynechají.

3. Jestliže množina obsahuje jeden člen, který není metoda, pak tento člen je výsledkem výběru. Jinak, jestliže množina obsahuje pouze metody, výsledkem je skupina těchto metod. Jiný případ nastat nemůže.

Volání metody

Výraz volání metody má jeden z následujících tvarů:

přístup-k-členu `(` *[seznam-argumentů]* `)`
jednoduché-jméno `(` *[seznam-argumentů]* `)`

První část výrazů musí určovat skupinu metod, jinak nastane chyba překladu.

Pokud seznam argumentů není prázdný, pak se skládá z jednoho nebo více argumentů oddělených čárkami. Každý argument má jednu z následujících forem:

- *výraz* – argument je předáván hodnotou.
- `ref` *výraz* – argument je předáván odkazem, výraz musí určovat proměnnou.
- `out` *výraz* – argument je předáván jako výstupní, výraz musí určovat proměnnou.

Seznam argumentů poskytuje výrazy nebo proměnné pro formální parametry metody. Konkrétní metoda, jež se vyvolá, je vybrána jako nejvhodnější přetížená metoda ze skupiny metod a to podle typů a druhů argumentů. Pokud nejvhodnější metoda nelze určit, nastane chyba při překladu. Pokud návratový typ vybrané metody je `void`, tento výraz není možné použít jako součást dalšího výrazu.

Přístup k prvku pole nebo řetězce

Přístup k prvku pole nebo řetězce má následující tvar:

primární-výraz `[` *seznam-výrazů* `]`

Typ primárního výrazu musí být buď pole nebo typ `string`. Seznam výrazů představuje indexy. Odděleny jsou čárkami. Každý z těchto výrazů musí být konvertovatelný na typ `int`. Počet indexů musí odpovídat dimenzi pole, nebo musí být 1 u typu `string`.

Pokud je primární výraz při běhu programu vyhodnocen na `null`, k prvku nelze přistoupit a program končí chybou. Jestliže některý z indexů je mimo rozsah dané dimenze, program končí chybou. Výsledkem operace je hodnota uložená na daném indexu pole nebo řetězce.

Přístup k aktuální instanci `this`

Přístup k aktuální instanci se skládá z klíčového slova `this`. Typem tohoto výrazu je třída, v níž se tento výraz nachází.

Postfixová inkrementace a dekrementace `++`, `--`

Postfixová inkrementace má tvar *výraz* `++` a postfixová dekrementace má tvar *výraz* `--`. Výraz musí určovat proměnnou nebo vlastnost. Tyto operátory jsou předdefinované pro operandy typů `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `decimal` a výčtových typů.

Při inkrementaci je hodnota proměnné nebo vlastnosti zvýšena o 1, při dekrementaci snížena o 1. Výsledkem výrazu je ale původní hodnota proměnné nebo vlastnosti.

Vytvoření objektu

Vytvoření objektu má následující tvar:

`new` *typ* `(` *[seznam-argumentů]* `)`

Typem může být jen třída, tato třída nesmí být abstraktní. Seznam argumentů může být prázdný a má stejný význam jako u vyvolání metody. Konstruktor, který se následně po alokaci objektu zavolá je vybrán z přístupných konstruktorů typu vzhledem k seznamu argumentů. Vybrán je nejvhodnější konstruktor podobně jako u volání metody. Výsledkem výrazu je nově vytvořený objekt.

Vytvoření pole

Vytvoření pole má jeden z následujících tvarů:

`new` *typ* `[` *seznam-výrazů* `]` *[inicializátor-pole]*
`new` *typ-pole inicializátor-pole*

V prvním tvaru typ nemůže být pole. Typ vytvořeného pole je určen uvedeným typem (typ elementů) a dimenze je určena počtem výrazů v seznamu výrazů. Inicializátor pole je popsán v sekci 1.6. o polích, v prvním tvaru není povinný. Výrazy v seznamu výrazů nemusí být konstantní, ale musí být konvertovatelné na typ `int`. Určují délky jednotlivých dimenzí.

V druhém tvaru je inicializátor pole povinný, protože určuje délky dimenzí pole.

Pokud je inicializátor uveden, jednotlivým prvkům pole jsou přiřazeny hodnoty uvedené v inicializátoru.

Výsledkem výrazu je nově vytvořený instance pole.

Unární výrazy

Unární výrazy mají vždy prefixovou notaci, tedy mají tvar *unární-výraz op*.

Unární operátory `+`, `-`, `!`, `~`

Operátor unární plus `+` je předdefinován pro operand následujících typů: `int`, `long`, `float`, `double`, `decimal`. Výsledkem operace je jednoduše uvedený operand.

Operátor unární mínus `-` je předdefinován opět pro operand následujících typů: `int`, `long`, `float`, `double`, `decimal`. Výsledkem operace je opačná hodnota operandu.

Operátor logické negace `!` je předdefinován pouze pro operand typu `bool`. Pokud je operand `true`, výsledkem je `false` a obráceně.

Operátor bitový doplněk `~` je předdefinován pro operand typů `int`, `long` a výčtových typů. Výsledkem je bitový doplněk hodnoty operandu.

Prefixová inkrementace a dekrementace `++`, `--`

Pro tyto operátory platí to samé jako pro postfixové operátory `++` a `--` s tím rozdílem, že výsledkem těchto výrazů je nová hodnota proměnné po inkrementaci nebo dekrementaci.

Výraz změny typu

Výraz změny typu se používá pro explicitní změnu typu. Jeho tvar je následující:

`(` *typ* `)` *unární-výraz*

Unární výraz musí být explicitně konvertovatelný na uvedený typ. Pokud se jedná o explicitní odkazovou konverzi, změna typu nemusí uspět a program je ukončen chybou.

Binární výrazy

Multiplikativní binární operátory `*`, `/`,`%`

Operátory násobení `*`, dělení `/` a zbytek po dělení `%` jsou předdefinovány pro dvojice operandů se stejnými následujícími typy: `int`, `long`, `float`, `double`, `decimal`. Operátory jsou prováděny běžným způsobem s povoleným přetečením hodnot.

Aditivní binární operátory `+`, `-`

Operátory sčítání `+` a odčítání `-` jsou předdefinovány pro dvojice operandů se stejnými následujícími typy: `int`, `long`, `float`, `double`, `decimal` a výčtové typy. Operátory jsou prováděny běžným způsobem s povoleným přetečením hodnot.

Dále je operátor `+` používán jako operátor konkatenace řetězců typu `string`. V tomto případě musí být alespoň jeden z operandů typu `string`. Druhým operandem může být libovolný objekt. Operátor při provádění získá jeho textovou reprezentaci pomocí metody `ToString()`, která je definovaná ve třídě `object` a tedy obsahují ji všechny objekty.

Operátory bitového posunu `<<`, `>>`

Operátory bitového posunu vlevo `<<` a bitového posunu vpravo `>>` jsou předefinovány pro dvojice operandů, kde levý operand je typu `int`, nebo `long` a druhý operand je typu `int`. Levý operand představuje hodnotu a pravý operand počet bitových pozic k posunutí.

Relační operátory `==`, `!=`, `<`, `>`, `<=`, `>=`

Výsledkem relačních operátorů je logická hodnota `true` nebo `false`.

Relační operátory rovno `==`, nerovno `!=`, menší `<`, větší `>`, menší nebo rovno `<=` a větší nebo rovno `>=` jsou předdefinovány pro dvojice operandů stejných numerických typů. Těmito typy mohou být: `int`, `long`, `float`, `double`, `decimal` a všechny výčtové typy. Operátory jsou prováděny běžným porovnáváním numerických hodnot operandů.

Operátory `==`, `!=` jsou dále předdefinovány pro dvojice operandů stejných následujících typů: `bool`, `object` a `string`. Operátor s typy `object` porovnává reference objektů, operátor s typy `string` porovnává řetězce pomocí ordinálních hodnot znaků na jednotlivých pozicích.

Operátory testování typu `is`, `as`

Binární operátor `is` je používán pro dynamické testování typu výrazů, tedy při běhu programu. Tvar výrazů s tímto operátorem je

výraz `is` *typ*

Výsledkem operátoru `is` je logická hodnota `true` nebo `false`.

Binární operátor `as` je používán pro explicitní konverzi typu výrazů. Je povolen jen pro odkazové typy. Tvar výrazů s tímto operátorem je

výraz `as` *odkazový-typ*

Typ celého výrazu je uvedený typ. Pokud je skutečný typ operandu implicitně konvertovatelný na uvedený typ, pak operace uspěje a výsledkem je operand. V opačném případě je výsledkem hodnota `null` a nedochází tak k chybě za běhu programu.

Logické operátory `&`, `^`, `|`

Logické operátory konjunkce (AND) `&`, exkluzivní disjunkce (XOR) `^` a disjunkce (OR) `|` jsou předdefinovány pro operandy celočíselných typů `int`, `long` a pro výčtových typů. Tyto operátory provádějí bitové logické operace běžným způsobem.

Dále jsou tyto operátory předdefinovány pro logické hodnoty typu `bool` opět obvyklým způsobem.

Podmíněné logické operátory `&&`, `||`

Tyto operátory jsou definovány jen pro operandy typu `bool` a jsou obdobou operátorů `&` a `|`. Jejich vlastností ovšem je, že vyhodnocují druhý operand pouze v případě nutnosti pro určení výsledku.

Ternární podmínkový operátor `?:`

Podmínkový operátor má tvar *podmínkový-výraz* `?` $výraz_1$ `:` $výraz_2$. Typ podmínkového výrazu musí být `bool`. Pokud je podmínka pravdivá, výsledkem celého výrazu je první výraz, v opačném případě je výsledkem druhý výraz. Oba uvedené výrazy musí být stejného typu, nebo jeden musí být implicitně konvertovatelný na typ druhého.

Operátory přiřazení `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`

Operátory přiřazení nastavují novou hodnotu uvedenou jako pravý operand do proměnné nebo vlastnosti uvedené jako levý operand. Operátor `=` je jednoduchý operátor přiřazení, ostatní jsou složené operátory přiřazení.

Jednoduchý operátor přiřazení má tvar *unární-výraz* `=` *výraz* a jednoduše přiřadí hodnotu pravého operandu do proměnné nebo vlastnosti, kterou představuje levý operand. Výraz na pravé straně musí být implicitně konvertovatelný na typ operandu na levé straně. Výsledkem je hodnota přiřazená levému operandu, typ výsledku je stejný jako typ levého operandu. V případě, že levý operand je vlastnost, musí mít `set` akcesor.

Složená přiřazení mají tvar *unární-výraz op*`=` *výraz*. *op* představuje jeden z operátorů `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`. Složené přiřazení tvaru x *op*`=` y je vyhodnoceno stejně jako x `=` x *op* y s tím rozdílem, že x je vyhodnoceno pouze jednou.

1.10. Příkazy

Akce programu v jazyku MiniC# jsou vyjadřovány pomocí příkazů. MiniC# nabízí množství příkazů převážně známých z jazyků C a C++.

Následující části uvádějí všechny příkazy jazyka MiniC#:

Blok

Bloky dovolují uvést více příkazů tam, kde je požadován jediný příkaz. Bloky také tvoří těla funkčních členů. Blok je složen z několika příkazů uzavřených ve složených závorkách, nebo může být prázdný:

`{` *[seznam-příkazů]* `}`

Prázdný příkaz

Prázdný příkaz nedělá nic. Používá se tam, kde je příkaz vyžadován, ale žádná akce není potřeba provést. Prázdný příkaz se skládá pouze ze středníku:

`;`

Deklarace lokálních proměnných

V rámci jednoho příkazu deklarace lokální proměnné je možno deklarovat více proměnných:

typ jméno$_1$*[=inicializátor*$_1$*] [*`,...,` *jméno*$_n$*[=inicializátor*$_n$*]]*`;`

Uvedený typ určuje typ deklarovaných proměnných. Rozsah platnosti deklarovaných proměnných je blok příkazů, ve kterém jsou deklarovány. Inicializátor může být výraz nebo inicializátor pole konvertovatelný na typ proměnné.

Deklarace lokálních konstant

V rámci jednoho příkazu deklarace lokálních konstant je možno deklarovat více konstant:

`const` *typ jméno*$_1$`=`*konstantní-výraz*$_1$ *[*`,...,` *jméno*$_n$`=`*konstantní-výraz*$_n$*]*`;`

Uvedený typ určuje typ deklarovaných konstant. Rozsah platnosti deklarované konstanty je blok příkazů ve kterém je deklarována. Konstantní výraz musí být konvertovatelný na typ konstanty.

Příkaz – výraz

Tento typ příkazu vyhodnotí daný výraz. Příkaz – výraz se skládá z výrazu následovaného středníkem:

výraz `;`

Ne všechny výrazy jsou v tomto příkazu povoleny. Povoleny jsou jen výrazy volání metody, vytvořená objektu, přiřazení, inkrementace a dekrementace (viz sekce 1.9.).

Podmíněný příkaz – `if`

Podmíněný příkaz vybere příkaz k provedení na základě vyhodnocení logické podmínky. Podmíněný příkaz má tvar:
`if (` *podmínkový-výraz* `)` *příkaz*$_1$ *[*`else` *příkaz*$_2$*]*

Podmínkový výraz je takový výraz, jehož typ je typ `bool`. První příkaz je proveden, pokud je podmínka vyhodnocena na hodnotu `true`. Pokud je podmínka vyhodnocena `false` a je přítomna sekce `else`, je proveden druhý příkaz. Uvedenými příkazy nemohou být jen příkazy deklarace proměnných nebo konstant.

Příkaz přepínač – `switch`

Příkaz `switch` vybere k provedení sekci příkazů uvedených v jeho těle za jedním z návěštích, jež je označeno výrazem, jehož hodnota je stejná jako hodnota řídícího výrazu:

```
switch ( řídící-výraz )
{
  case konstantní-výraz11 :
  ...
  case konstantní-výraz1m1 : seznam-příkazů1
  ...
```

```
  case konstantní-výraz_n1 :
  ...
  case konstantní-výraz_nmn : seznam-příkazů_n
  [default : seznam-příkazů_n+1]
}
```

Blok příkazu `switch` obsahuje sekce, které obsahují jedno nebo více návěští následované příkazy. Typ řídícího výrazu může být jeden z následujících typů: `byte`, `short`, `int`, `long`, `char`, `string` a libovolný výčtový typ. Konstantní výrazy ve všech návěštích musí být implicitně konvertovatelné na typ řídícího výrazu. Žádné dvě návěští nemohou obsahovat výraz se stejnou hodnotu. Sekce `default` může být uvedena nejvýše jednou.

Hodnota řídícího výrazu je postupně srovnávána s konstantními výrazy v návěštích. Jestliže je hodnota shodná, vykonají se příkazy následující za návěštím. Pokud hodnota řídícího výrazu není shodná s žádnou hodnotou uvedenou v návěštích, provedou se příkazy v sekci `default`, pokud tato sekce existuje.

V jazyku MiniC# není možné, aby vykonávání příkazů „propadlo“ do jiné sekce. Sekci příkazů je potřeba ukončit příkazem skoku `break`, `return` nebo `continue`.

Příkaz cyklu s podmínkou na začátku – `while`

Příkaz cyklu s podmínkou na začátku podmíněně cyklicky provádí daný příkaz. Cyklus `while` má následující formát:

`while (` *podmínkový-výraz* `)` *příkaz*

Příkaz je cyklicky vykonáván dokud je hodnota podmínkového výrazu rovna hodnotě `true`.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu `while` se výpočet bezpodmínečně přesouvá k opětovnému vyhodnocení podmínky. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz cyklu s podmínkou na konci – `do`

Příkaz cyklu s podmínkou na konci podmíněně provádí daný příkaz, nejméně však jednou. Cyklus `do` má následující formát:

`do` *příkaz* `while (` *podmínkový-výraz* `) ;`

Příkaz je cyklicky vykonáván dokud je hodnota podmínkového výrazu rovna hodnotě `true`.

Příkazem `continue`, uvedeným v rámci složeného příkazu v příkazu `do`, se výpočet bezpodmínečně přesouvá k vyhodnocení podmínky. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz cyklu s řídící proměnnou – `for`

Formát příkazu for je následující:

`for (` *inicializátor-cyklu* `;` *podmínkový-výraz* `;` *iterátor* `)` *příkaz*

Inicializátor cyklu, podmínka i iterátor jsou nepovinné. Inicializátorem cyklu může být buď příkaz deklarace lokálních proměnných nebo seznam výrazů, jež mohou být příkazy,

oddělených čárkami. Platnost deklarovaných proměnných sahá přes celý příkaz `for`. Iterátor je tvořen seznamem výrazů, jež mohou být příkazy, oddělených čárkami.

Příkaz `for` nejdříve jedinkrát vyhodnotí inicializátor cyklu. Následně se vyhodnotí podmínka. Pokud je její hodnota `true`, provede se uvedený příkaz. Jestliže se dosáhne konce uvedeného příkazu, provedou se výrazy v iterátoru a následuje další iterace začínající vyhodnocením podmínky. Neplatí-li podmínka příkaz je ukončen.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu for se výpočet bezpodmínečně přesouvá na konec složeného příkazu. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz procházení kolekcí – `foreach`

Příkaz `foreach` prochází kolekci a pro každý element kolekce provede uvedený příkaz. Příkaz `foreach` má následující tvar:

`foreach (` *typ jméno* `in` *výraz* `)` *příkaz*

Typ a jméno představují lokální proměnnou, která je jen pro čtení a představuje aktuální element kolekce dané uvedeným výrazem. Typ výrazu kolekce musí odpovídat níže uvedenému vzoru a typ prvků kolekce musí být explicitně konvertovatelný na typ iterační proměnné.

Libovolný výraz `C` splňuje vzor pro kolekci, jestliže platí:

- Typ výrazu `C` obsahuje veřejnou instanční metodu se signaturou `GetEnumerator()`, jejíž návratový typ je třída, dále označena jako `E`.
- `E` obsahuje veřejnou instanční metodu s hlavičkou `bool MoveNext()`.
- `E` obsahuje veřejnou vlastnost se jménem `Current`, její typ je typem elementu kolekce.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu `foreach` se výpočet bezpodmínečně přesouvá na konec složeného příkazu. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz ukončení cyklu nebo přepínače – `break`

Příkaz `break` ukončuje nejbližší příkaz `switch`, `while`, `do`, `for`, nebo `foreach`. Všechny následující příkazy v cyklu jsou vynechány. Příkaz `break` má tvar

```
break ;
```

Příkaz ukončení iterace cyklu – `continue`

Příkaz `continue` nepodmíněně začíná novou iteraci nejbližšího příkazu `while`, `do`, `for`, nebo `foreach`. Všechny následující příkazy v těle cyklu pro danou iteraci jsou vynechány. Příkaz `continue` má tvar

```
continue ;
```

Příkaz návratu z funkce – `return`

Příkaz `return` ukončuje vykonávání aktuálního funkčního členu (metody, akcesoru vlastnosti nebo konstruktoru) a vrací řízení programu funkci, která aktuální funkci aktivovala. Příkaz `return` má následující tvar:

`return` *[výraz]* `;`

Forma bez výrazu je povolena jen ve funkčních členech, které nemají návratový typ. Forma s výrazem je povolena jen ve funkčních členech, které mají návratový typ. Pokud je výraz uveden, musí být implicitně konvertovatelný na návratový typ funkčního členu. Hodnota výrazu se stává návratovou hodnotou funkčního členu. Následně je řízení programu vráceno volajícímu funkčnímu členu.

2. Knihovna základních typů

Součástí projektu MiniC# je knihovna základních typů, jež se nachází v souboru `mcsbaselib.dll`. Knihovna obsahuje především definice typů, jež vyžaduje přímo programovací jazyk MiniC# a dále vybrané „systémové“ třídy např. pro práci se soubory, standardními vstupy a výstupy nebo např. se systémovým časem.

2.1. Předdefinované typy

Následuje výčet většiny typů základní knihovny spolu s jejich popisem a popisem jejich členů.

Třída `object`

```
class object
```

Třída `object` představuje základní typ všech ostatních typů a tím tvoří kořen stromu hierarchie typů v programu jazyka MiniC#. Tato třída dále poskytuje základní metody společné pro všechny objekty.

Konstruktory:

```
public object()
```

Inicializuje novou instanci třídy `object`. Tento konstruktor je volán konstruktory zděděných tříd, ale může být použit i pro přímé vytvoření objektu.

Metody:

```
public virtual bool Equals(object obj)
```

Určuje, zda specifikovaný objekt `obj` je roven aktuální instanci. Tato metoda porovnává objekty pomocí shodnosti referencí. Zděděné třídy mohou tuto metodu přepsat tak, že objekty jsou porovnávány podle svých datových položek.

Návratová hodnota je rovna `true` jestliže `obj` je rovno aktuální instanci, jinak `false`.

```
public static bool Equals(object objA, object objB)
```

Určuje, zda specifikované objekty jsou si `objA` a `objB` rovny.

Návratová hodnota je `true`, jestliže `objA` určuje stejnou instanci jako `objB`, nebo jsou oba parametry `null`, nebo jestliže `objA.Equals(objB)` vrací `true`, jinak `false`.

```
public static bool ReferencesEquals(object objA, object objB)
```

Určuje, zda specifikované instance objektů `objA` a `objB` určují stejnou instanci. Návratová hodnota je `true`, jestliže `objA` určuje stejnou instanci jako `objB`, nebo jsou oba parametry `null`, jinak `false`.

```
public virtual string ToString()
```

Vrací objekt typu `string`, který reprezentuje aktuální objekt. Implementace metody ve třídě `object` vrací jméno typu instance. Zděděné typy obvykle přepisují tuto metodu a vracejí textovou reprezentující aktuální instance. Např. numerické typy vracejí textovou reprezentaci své číselné hodnoty.

Třída `string`

```
class string : object
```

Reprezentuje textové řetězce jako sekvenci znaků Unicode.

Konstruktory:

```
public string(char[] chars)
```

Inicializuje novou instanci třídy `string` na hodnotu, danou polem znaků.

```
public string(char[] chars, int startIndex, int length)
```

Inicializuje novou instanci třídy `string` na hodnotu danou polem znaků, pozicí počátečního znaku a délkou.

```
public string(char[] char, int count)
```

Inicializuje novou instanci třídy `string` na hodnotu danou opakováním znaku daným počtem.

Datové položky:

```
public static readonly string Empty
```

Reprezentuje prázdný řetězec `""`. Tato datová položka je pouze ke čtení.

Vlastnosti:

```
public int Length { get; }
```

Určuje počet znaku v aktuální instanci.

Metody:

```
public int CompareTo(string value)
```

Porovná aktuální instanci se specifikovanou instancí typu `string`. Návratová hodnota je 32bitové číslo, které je záporné, pokud aktuální instance je menší než `value`, je rovno nule, když aktuální instance je rovna `value` nebo je větší než nula, pokud aktuální instance je větší než `value`.

`public static int Compare(string strA, string strB)`

Porovná dvě specifikované instance třídy `string`. Návratová hodnota je 32bitové číslo, které je záporné, pokud `strA` je menší než `strB`, je rovno nula, když `strA` je rovna `strB` nebo je větší než nula, pokud `strA` je větší než `strB`.

`public static int Compare(string strA, string strB, bool ignoreCase)`

Porovná dvě specifikované instance třídy `string`, parametr `ignoreCase` umožňuje určit zda při porovnání bude záležet na velikosti písmen. Návratová hodnota je 32bitové číslo, které je záporné, pokud `strA` je menší než `strB`, je rovno nula, když `strA` je rovna `strB` nebo je větší než nula, pokud `strA` je větší než `strB`.

`public static string Concat(params object[] objects)`

Vrací spojení řetězcových reprezentací prvků ve specifikovaném poli.

`public static string Concat(params string[] strings)`

Vrací spojení řetězců ve specifikovaném poli.

`public override bool Equals(object obj)`

Určuje, zda aktuální instance typu `string` a specifikovaný objekt mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `string` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

`public static bool Equals(string strA, string strB)`

Určuje, zda specifikované instance typu `string` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `strA` má stejnou hodnotu jako `strB`, jinak `false`.

`public static string Format(string format, params object[] objects)`

Vytvoří kopii řetězce `format`, v níž nahradí formátovací položky v řetězci `format` textovými reprezentacemi odpovídajících objektů ve specifikovaném poli `objects`.

Řetězec `format` může obsahovat formátovací položky. Formátovací položky mají tvar `{` *index* [`,` *zarovnání*] `}`, která specifikuje povinný index a nepovinou délku a zarovnání formátovaného textu. *index* je celé nezáporné číslo, které určuje prvek pole `objects`, který má být formátován. *zarovnání* je celočíselná hodnota, která určuje minimální šířku oblasti, která bude obsahovat formátovanou hodnotu. Jestliže délka formátované hodnoty je menší než *zarovnání*, oblast je doplněna mezerami. Pokud je *zarovnání* záporné, text je zarovnán doleva, pokud je kladné, pak je text zarovnán doprava. Pokud *zarovnání* není uvedeno, délka oblasti bude stejná jako délka formátované hodnoty. Znaky `{` a `}` je možné ve formátovacím řetězci zapsat jako `{{` a `}}`.

`public string Substring(int startIndex, int length)`

Vrací nový řetězec, který odpovídá podřetězci aktuálního řetězce, který začíná znakem na indexu `startIndex` a má délku maximálně `length`.

`public string ToUpper()`

Vrací nový řetězec, který odpovídá aktuálnímu řetězci převedenému na velká písmena.

`public string ToLower()`

Vrací nový řetězec, který odpovídá aktuálnímu řetězci převedenému na malá písmena.

`public override string ToString()`

Vrací aktuální instanci.

Typ `char`

`class char : object`

Reprezentuje 16bitový znak kódování Unicode.

Konstanty:

`public const char MaxValue`

Reprezentuje největší možnou hodnotu typu `char`. Hodnota této konstanty je hexadecimálně 0xFFFF.

`public const char MinValue`

Reprezentuje nejmenší možnou hodnotu typu `char`. Hodnota této konstanty je hexadecimálně 0x0000.

Metody:

`public int CompareTo(char value)`

Porovnává aktuální instanci se specifikovanou hodnotou typu `char`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`. Instance typu `char` se porovnávají podle svých číselných hodnot.

`public override bool Equals(object obj)`

Určuje, zda aktuální instance typu `char` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `char` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

`public static char ToUpper(char c)`

Vrací znak, jenž je ekvivalentem aktuální instance převedený na velké písmeno, nebo vrací `c`, pokud `c` je již velké písmeno, nebo znak `c` není písmeno.

`public static char ToLower(char c)`

Vrací znak, jenž je ekvivalentem aktuální instance převedený na malé písmeno, nebo vrací `c`, pokud `c` je již malé písmeno, nebo znak `c` není písmeno.

`public override string ToString()`

Vrací textovou reprezentaci hodnoty aktuální instance.

Typ `bool`

```
class bool : object
```

Reprezentuje booleovskou logickou hodnotu `true` nebo `false`.

Metody:

```
public int CompareTo(bool value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `bool`. Návratová hodnota je číslo typu `int`, které je záporné, pokud aktuální instance je `false` a `value` je `true`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je `true` a `value` je `false`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `bool` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `bool` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci hodnoty aktuální instance. Pokud je hodnota aktuální instance `true`, vrací řetězec `"True"`, jinak vrací řetězec `"False"`.

Typ `byte`

```
class byte : object
```

Reprezentuje 8bitové celé číslo v rozsahu 0 až 255.

Konstanty:

```
public const byte MaxValue
```

Reprezentuje největší možnou hodnotu typu `byte`. Hodnota této konstanty je 255.

```
public const byte MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `byte`. Hodnota této konstanty je 0.

Metody:

```
public int CompareTo(byte value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `byte`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `byte` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `byte` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `short`

```
class short : object
```

Reprezentuje 16bitové celé číslo se znaménkem v rozsahu -32768 až 32767.

Konstanty:

```
public const short MaxValue
```

Reprezentuje největší možnou hodnotu typu `short`. Hodnota této konstanty je 32767.

```
public const short MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `short`. Hodnota této konstanty je -32768.

Metody:

```
public int CompareTo(short value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `short`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `short` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `short` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `int`

```
class int : object
```

Reprezentuje 32bitové celé číslo se znaménkem v rozsahu -2147483648 až 2147483647.

Konstanty:

```
public const int MaxValue
```

Reprezentuje největší možnou hodnotu typu `int`. Hodnota této konstanty je 2147483647.

```
public const int MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `int`. Hodnota této konstanty je -2147483648.

Metody:

```
public int CompareTo(int value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `int`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule,

pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `int` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `int` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `long`

```
class long : object
```

Reprezentuje 64bitové celé číslo se znaménkem v rozsahu -9223372036854775808 až 9223372036854775807.

Konstanty:

```
public const long MaxValue
```

Reprezentuje největší možnou hodnotu typu `long`. Hodnota této konstanty je 9223372036854775807.

```
public const long MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `long`. Hodnota této konstanty je -9223372036854775808.

Metody:

```
public int CompareTo(long value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `long`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `long` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `long` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `float`

```
class float : object
```

Reprezentuje 32bitové číslo v plovoucí čárce dle standardu IEEE 754. Hodnoty reprezentované tímto typem jsou čísla z rozsahu -3.402823e38 až 3.402823e38 s přesností na 7 číslic, dále kladná i záporná nula, kladné a záporné nekonečno a hodnotu, která není číslo (NaN).

Konstanty:

```
public const float Epsilon
```

Reprezentuje nejmenší kladné číslo typu `float`. Hodnota této konstanty je 1.4e-45.

```
public const float MaxValue
```

Reprezentuje největší možné číslo typu `float`. Hodnota této konstanty je 3.402823e38.

```
public const float MinValue
```

Reprezentuje nejmenší možné číslo typu `float`. Hodnota této konstanty je -3.402823e38.

```
public const float NaN
```

Reprezentuje hodnotu typu `float`, která není číslo. Tato hodnota je např. výsledkem dělení nuly nulou. Pro testování hodnoty typu `float` na hodnotu `NaN` se používá metoda `IsNaN`.

```
public const float NegativeInfinity
```

Reprezentuje záporné nekonečno. Tato hodnota je např. výsledkem dělení záporného čísla nulou, nebo pokud výsledek některé operace je menší než `MinValue`. Pro testování hodnoty typu `float` na hodnotu `NegativeInfinity` se používá metoda `IsNegativeInfinity`.

```
public const float PositiveInfinity
```

Reprezentuje kladné nekonečno. Tato hodnota je např. výsledkem dělení kladného čísla nulou, nebo pokud výsledek některé operace je větší než `MaxValue`. Pro testování hodnoty typu `float` na hodnotu `PositiveInfinity` se používá metoda `IsPositiveInfinity`.

Metody:

```
public int CompareTo(float value)
```

Porovnává aktuální instanci se specifikovanou hodnotou typu `float`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `float` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `float` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public static bool IsNaN(float value)
```

Určuje, zda specifikovaná instance typu `float` je NaN.

`public static bool IsInfinity(float value)`

Určuje, zda specifikovaná instance typu `float` je kladné nebo záporné nekonečno.

`public static bool IsNegativeInfinity(float value)`

Určuje, zda specifikovaná instance typu `float` je záporné nekonečno.

`public static bool IsPositiveInfinity(float value)`

Určuje, zda specifikovaná instance typu `float` je kladné nekonečno.

`public override string ToString()`

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `double`

`class double : object`

Reprezentuje 64bitové číslo v plovoucí čárce dle standardu IEEE 754. Hodnoty reprezentované tímto typem jsou čísla z rozsahu -1.79769313486232e308 až 1.79769313486232e308 s přesností na 15 - 16 číslic, dále kladná i záporná nula, kladné a záporné nekonečno a hodnotu, která není číslo (NaN).

Konstanty:

`public const double Epsilon`

Reprezentuje nejmenší kladné číslo typu `double`. Hodnota této konstanty je 4.94065645841247e-324.

`public const double MaxValue`

Reprezentuje největší možné číslo typu `double`. Hodnota této konstanty je 1.79769313486232e308.

`public const double MinValue`

Reprezentuje nejmenší možné číslo typu `double`. Hodnota této konstanty je -1.79769313486232e308.

`public const double NaN`

Reprezentuje hodnotu typu `double`, která není číslo. Tato hodnota je např. výsledkem dělení nuly nulou. Pro testování hodnoty typu `double` na hodnotu `NaN` se používá metoda `IsNaN`.

`public const double NegativeInfinity`

Reprezentuje záporné nekonečno. Tato hodnota je např. výsledkem dělení záporného čísla nulou, nebo pokud výsledek některé operace je menší než `MinValue`. Pro testování hodnoty typu `double` na hodnotu `NegativeInfinity` se používá metoda `IsNegativeInfinity`.

`public const double PositiveInfinity`

Reprezentuje kladné nekonečno. Tato hodnota je např. výsledkem dělení kladného čísla nulou, nebo pokud výsledek některé operace je větší než `MaxValue`. Pro testování hodnoty typu `double` na hodnotu `PositiveInfinity` se používá metoda `IsPositiveInfinity`.

Metody:

`public int CompareTo(double value)`

Porovnává aktuální instanci se specifikovanou hodnotou typu `double`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

`public override bool Equals(object obj)`

Určuje, zda aktuální instance typu `double` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `double` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

`public static bool IsNaN(double value)`

Určuje, zda specifikovaná instance typu `double` je NaN.

`public static bool IsInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je kladné nebo záporné nekonečno.

`public static bool IsNegativeInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je záporné nekonečno.

`public static bool IsPositiveInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je kladné nekonečno.

`public override string ToString()`

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Typ `decimal`

`class decimal : object`

Reprezentuje desetinná čísla v rozsahu -79228162514264337593543950335 až 79228162514264337593543950335 s přesností na 28 - 29 číslic. Instance typu `decimal` mají 128 bitů, 1 bit zabírá znaménko, 96 bitů celočíselná hodnota a zbytek zabírá dělící faktor 0 - 28, který představuje mocniny 10.

Konstanty:

`public const decimal MaxValue`

Reprezentuje největší možnou hodnotu typu `decimal`. Hodnota této konstanty je 79228162514264337593543950335.

`public const decimal MinValue`

Reprezentuje nejmenší možnou hodnotu typu `decimal`. Hodnota této konstanty je -79228162514264337593543950335.

Metody:

`public int CompareTo(decimal value)`

Porovnává aktuální instanci se specifikovanou hodnotou typu `decimal`. Návratová hodnota je číslo typu `int`, která je záporná, pokud aktuální instance je menší než `value`, je rovna nule, pokud aktuální instance je rovna `value`, nebo je větší než nula, pokud aktuální instance je větší než `value`.

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance typu `decimal` a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `decimal` a jeho hodnota je stejná jako hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci numerické hodnoty aktuální instance.

Třída `Array`

```
abstract class Array : object
```

Představuje základní typ všech polí, poskytuje společné metody a vlastnosti.

Vlastnosti:

```
public int Length { get; }
```

Vrací počet prvků ve všech dimenzích aktuální instance pole dohromady.

```
public int Rank { get; }
```

Vrací počet dimenzí pole.

Metody:

```
public int GetLength(int dim)
```

Vrací počet prvků aktuální instance ve specifikované dimenzi `dim`. Dimenze jsou číslovány od nuly.

```
public object GetValue(int[] indices)
```

Vrací hodnotu na specifikované pozici v aktuální instance typu `Array`. Pozice je dána polem indexů `indices`. Počet indexů musí být roven dimenzi aktuální instance. Tato metoda se používá v případě, kdy dimenze pole není známa při překladu. V opačném případě je vhodné použít syntaxi přístupu k prvku pole pomocí `[]`.

```
public void SetValue(object obj, int[] indices)
```

Nastavuje hodnotu na specifikované pozici v aktuální instanci typu `Array` na hodnotu `obj`. Pozice je dána polem indexů `indices`. Počet indexů musí roven dimenzi aktuální instance. Tato metoda se používá v případě, kdy dimenze pole není známa při překladu. V opačném případě je vhodné použít syntaxi přístupu k prvku pole pomocí `[]`.

Třída `Enum`

```
abstract class Enum : object
```

Představuje základní typ všech výčtových typů a poskytuje společné metody.

Metody:

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je stejného typu jako aktuální instance a číselná hodnota `obj` je stejná jako číselná hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci hodnoty aktuální instance. Jestliže hodnota aktuální instance je rovna některé pojmenované konstantě výčtu, metoda vrací jméno této konstanty, jinak číselnou reprezentaci.

2.2. Ostatní typy

Třída `Console`

```
class Console : object
```

Reprezentuje standardní vstupní a výstupní textové streamy programu.

Metody:

```
public static void Write(object value)
```

Zapíše textovou reprezentaci objektu `value` do standardního výstupního streamu programu.

```
public static void Write(string format, params object[] objects)
```

Zapíše textovou reprezentaci objektů v poli `objects` do standardního výstupního streamu programu s použitím formátovacího řetězce `format`.

```
public static void WriteLine()
```

Zapíše na standardní výstupní stream konec řádku.

```
public static void WriteLine(object value)
```

Zapíše textovou reprezentaci objektu `value` následovanou koncem řádku do standardního výstupního streamu programu.

```
public static void Write(string format, params object[] objects)
```

Zapíše textovou reprezentaci objektů v poli `objects` následovanou koncem řádku do standardního výstupního streamu programu s použitím formátovacího řetězce `format`.

```
public static int Read()
```

Přečte a vrátí další znak ze standardního vstupního streamu programu. Znaky jsou reprezentovány jako čísla typu `int`. Jestliže na vstupu není žádný znak, vrací hodnotu -1.

```
public static string ReadLine()
```

Přečte další řádek ze standardního vstupního streamu programu a vrátí jej jako řetězec typu `string`.

Třída `DateTime`

```
class DateTime : object
```

Reprezentuje datum a čas v rozmezí let 1 až 9999 Gregoriánského kalendáře.

Konstruktory:

```
public DateTime(long ticks)
```

Inicializuje novou instanci pomocí počtu tiků. Jeden tik je dlouhý 100 nanosekund.

```
public DateTime(int year, int month, int day)
```

Inicializuje novou instanci pomocí roku, měsíce a dne. Čas je nastaven a půlnoc mezi daným dnem a předchozím dnem. Hodnoty parametrů musí udávat reprezentovatelné datum.

```
public DateTime(int year, int month, int day, int hour, int minute, int second)
```

Inicializuje novou instanci pomocí roku, měsíce a dne, hodiny, minuty a sekundy. Hodnoty parametrů musí udávat reprezentovatelný čas.

```
public DateTime(int year, int month, int day, int hour, int minute, int second, int millis
```

Inicializuje novou instanci pomocí roku, měsíce a dne, hodiny, minuty a sekundy a milisekundy. Hodnoty parametrů musí udávat reprezentovatelný čas.

Datové položky:

```
public static readonly DateTime MinValue
```

Reprezentuje nejmenší reprezentovatelnou hodnotu data a času – začátek roku 1.

```
public static readonly DateTime MaxValue
```

Reprezentuje největší reprezentovatelnou hodnotu data a času – konec roku 9999.

Vlastnosti:

```
public static DateTime Now { get; }
```

Vrací aktuální lokální datum a čas nastavený na tomto počítači.

```
public int Year { get; }
```

Vrací rok data reprezentovaného aktuální instancí.

```
public int Month { get; }
```

Vrací měsíc data reprezentovaného aktuální instancí.

```
public int Day { get; }
```

Vrací den v měsíci data reprezentovaného aktuální instancí.

```
public int DayOfYear { get; }
```

Vrací den v roce data reprezentovaného aktuální instancí.

```
public DayOfWeek DayOfWeek { get; }
```

Vrací den v týdnu data reprezentovaného aktuální instancí. Návratová hodnota je výčtového typu `DayOfWeek`.

```
public int Hour { get; }
```

Vrací hodinu data a času reprezentovaného aktuální instancí.

```
public int Minute { get; }
```

Vrací minutu data a času reprezentovaného aktuální instancí.

```
public int Second { get; }
```

Vrací sekundu data a času reprezentovaného aktuální instancí.

```
public int Millisecond { get; }
```

Vrací milisekundu data a času reprezentovaného aktuální instancí.

```
public int Ticks { get; }
```

Vrací počet tiků od začátku roku 1, jenž reprezentuje aktuální instanci data a času.

Metody:

```
public static bool IsLeapYear(int year)
```

Indikuje zda uvedený rok je přestupný.

```
public static int DaysInMonth(int year, int month)
```

Vrací počet dnů uvedeného měsíce v uvedeném roce.

```
public void Add(long ticks)
```

Přidá k aktuální instanci uvedený počet tiků.

public void Add(int year, int month, int day, int hour, int minute, int second, int millisecond)

```
public void Add(int year, int month, int day, int hour, int minute, int second, int millis
```

Přidá k aktuální instanci uvedená počet roků, měsíců dnů hodin atd.

```
public int CompareTo(DateTime dt)
```

Porovná aktuální instanci typu `DateTime` s uvedenou instancí `dt`. Výsledná hodnota je větší než nula, pokud aktuální instance reprezentuje datum a čas větší než hodnota `dt`, nebo je rovna nule, pokud reprezentují stejné datum a čas a je menší než nula, pokud aktuální instance reprezentuje menší datum a čas než `dt`.

`public override bool Equals(object obj)`

Určuje, zda aktuální instance a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je typu `DateTime` a `obj` reprezentuje stejný čas jako aktuální instance, jinak `false`.

`public override string ToString()`

Vrací textovou reprezentaci aktuální instance ve formátu krátké datum a dlouhý čas.

`public string ToShortTimeString()`

Vrací textovou reprezentaci aktuální instance ve formátu času v hodinách a minutách.

`public string ToLongTimeString()`

Vrací textovou reprezentaci aktuální instance ve formátu času v hodinách, minutách a sekundách.

`public string ToShortDateString()`

Vrací textovou reprezentaci aktuální instance ve formátu data s uvedením dne v měsíci, čísla měsíce a roku.

`public string ToLongDateString()`

Vrací textovou reprezentaci aktuální instance ve formátu data s uvedením dne v měsíci, jména měsíce a čísla roku.

Výčtový typ `DayOfWeek`

`enum DayOfWeek`

Specifikuje dny v týdnu.

Členy:

`Monday` – pondělí `Tuesday` – úterý `Wednesday` – středa `Thursday` – čtvrtek `Friday` – pátek `Saturday` – sobota `Sunday` – neděle

Třída `Parse`

`class Parse : object`

Tato třída poskytuje statické metody pro převod hodnot základních typů z textové podoby.

Metody:

`public static bool ToBool(string s, out bool val)`

Převede logickou hodnotu reprezentovanou řetězcem `s` do reprezentace typem `bool`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

`public static bool ToByte(string s, out byte val)`

Převede číslo reprezentované řetězcem `s` do reprezentace typem `byte`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToChar(string s, out char val)
```

Převede znak Unicode reprezentovaný řetězcem `s` do reprezentace typem `char`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToShort(string s, out short val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `short`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToInt(string s, out int val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `int`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToLong(string s, out long val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `long`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToFloat(string s, out float val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `float`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToDouble(string s, out double val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `double`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
public static bool ToDecimal(string s, out decimal val)
```

Převede číslo reprezentované řetězcem `s` do reprezentace typem `decimal`. Převedená hodnota je uložena do parametru `val`. Návratová hodnota indikuje, zda převod proběhl úspěšně.

```
class Math : object
```

Poskytuje konstanty a statické metody pro obvyklé matematické funkce.

Konstanty:

```
public const double E
```

Reprezentuje Eulerovo číslo přibližnou hodnotou 2.71828.

```
public const double PI
```

Reprezentuje Ludolfovo číslo ϕ přibližnou hodnotou 3.14159.

Metody:

```
public static short Abs(short val)
```

Vrací absolutní hodnotu čísla typu `short`.

Třída `Math` dále obsahuje přetížené metody `Abs` pro typy `int`, `long`, `float double` a `decimal`.

`public static short Sign(short val)`

Vrací znaménko čísla typu `short`. Znaménka jsou reprezentována hodnotami -1, 0, +1.

Třída `Math` dále obsahuje přetížené metody `Sign` pro typy `int`, `long`, `float double` a `decimal`.

`public static byte Max(byte a, byte b)`

Vrací větší z čísel `a` a `b`.

Třída `Math` dále obsahuje přetížené metody `Max` pro typy `short`, `int`, `long`, `float double` a `decimal`.

`public static byte Min(byte a, byte b)`

Vrací menší z čísel `a` a `b`.

Třída `Math` dále obsahuje přetížené metody `Min` pro typy `short`, `int`, `long`, `float double` a `decimal`.

`public static double Round(double val)`

Vrací nejbližší celé číslo k číslu `val`. Pokud je `val` přesně mezi dvěma čísli, návratovou hodnotou je to sudé z nich.

`public static double Round(double val, int digits)`

Vrací číslo s danou přesností k číslu `val`. Pokud je `val` přesně mezi dvěma celými čísli, návratovou hodnotou je to sudé z nich.

`public static decimal Round(decimal val)`

Vrací nejbližší celé číslo k číslu `val`. Pokud je `val` přesně mezi dvěma čísli, návratovou hodnotou je to sudé z nich.

`public static decimal Round(decimal val, int digits)`

Vrací číslo s danou přesností k číslu `val`. Pokud je `val` přesně mezi dvěma celými čísli, návratovou hodnotou je to sudé z nich.

`public static double Acos(double val)`

Vrací úhel v radiánech, jehož kosinus je `val`. Pokud je `val` mimo definiční obor funkce, metoda vrací `double.NaN`.

`public static double Asin(double val)`

Vrací úhel v radiánech, jehož sinus je `val`. Pokud je `val` mimo definiční obor funkce, metoda vrací `double.NaN`.

`public static double Atan(double val)`

Vrací úhel v radiánech, jehož tangens je `val`. Pokud je `val` mimo definiční obor funkce, metoda vrací `double.NaN`.

`public static double Cos(double val)`

Vrací kosinus úhlu daného v radiánech.

`public static double Cosh(double val)`

Vrací hyperbolický kosinus úhlu daného v radiánech.

`public static double Sin(double val)`

Vrací sinus úhlu daného v radiánech.

`public static double Sinh(double val)`

Vrací hyperbolický sinus úhlu daného v radiánech.

`public static double Tan(double val)`

Vrací tangens úhlu daného v radiánech.

`public static double Tanh(double val)`

Vrací hyperbolický tangens úhlu daného v radiánech.

`public static double Exp(double val)`

Vrací číslo e umocněné na `val`.

`public static double Log(double val)`

Vrací přirozený logaritmus čísla `val`.

`public static double Log10(double val)`

Vrací logaritmus o základu 10 čísla `val`.

`public static double Log(double x, double y)`

Vrací logaritmus o základu `y` čísla `x`.

`public static double Sqrt(double val)`

Vrací druhou odmocninu z čísla `val`. Pokud je číslo `val` záporné vrací `double.NaN`.

`public static double Pow(double x, double y)`

Vrací číslo `x` umocněné na `y`.

`public static double Ceiling(double val)`

Vrací nejmenší celé číslo, které je větší nebo rovno než číslo `val`.

`public static double IEEERemainder(double x,double y)`

Vrací zbytek největšího celočíselného násobku čísla `x`/`x` do čísla `x`.

`class ArrayList : object`

Implementuje seznam pomocí pole, které je v případě potřeby dynamicky zvětšováno.

Konstruktory:

`public ArrayList()`

Inicializuje novou instanci, která je prázdná.

`public ArrayList(int capacity)`

Inicializuje novou instanci, která je prázdná a nastavuje počáteční kapacitu na `capacity`.

`public ArrayList(params object[] objects)`

Inicializuje novou instanci a naplní ji objekty danými parametrem `objects`.

`public ArrayList(ArrayList a)`

Inicializuje novou instanci a naplní ji objekty z daného `ArrayListu` `a`.

Vlastnosti:

`public int Capacity { get; }`

Vrací kapacitu aktuální instance.

`public int Count { get; }`

Vrací počet prvků vložených do aktuální instance.

Metody:

`public int Add(object obj)`

Přidá daný objekt `obj` na konec aktuálního `ArrayListu`.

`public int AddRange(params object[] objects)`

Přidá objekty dané parametrem `objects` na konec aktuálního `ArrayListu`.

`public int AddRange(ArrayList a)`

Přidá objekty z daného `ArrayListu` `a` na konec aktuálního `ArrayListu`.

`public object GetItem(int i)`

Vrátí objekt na indexu `i`.

`public void SetItem(int i, object obj)`

Nastaví objekt na indexu `i` na hodnotu `obj`.

`public void Clear()`

Odstraní všechny objekty z aktuální instance.

`public bool Contains(object obj)`

Zjistí, zda aktuální instance instance obsahuje objekt `obj`.

`public int IndexOf(object obj)`

Vrátí index prvního výskytu objektu `obj` v aktuální instanci. Jestliže objekt `obj` není obsažen vrací -1.

`public void Insert(int index, object obj)`

Přidá objekt `obj` na index `index` v aktuální instanci.

`public void Remove(object obj)`

Odebere první výskyt objektu `obj` z aktuální instance.

`public void RemoveAt(int index)`

Odebere objekt na indexu `index` z aktuální instance.

`public object[] ToArray()`

Vytvoří a vrátí pole obsahující všechny objektu vložené v aktuální instanci.

`public ArrayListEnumerator GetEnumerator()`

Vrací objekt umožňující procházení aktuální instance příkazem `foreach`.