Základy programování 3: C#

Martin Kauer

Palacký University
Olomouc
Czech Republic

17. října 2018

Metody třídy `System.Object`

- `bool Equals(object obj)` - virtuální metoda, rozhodující zda zadaný objekt je ekvivalentní aktuální instanci.
- `int GetHashCode()` - virtuální metoda, počítá hash kód aktuální instance. Používá se jako klíč v hashovací tabulce a také má zásadní dopad na určování ekvivalence.
  - Musí platit, že pokud jsou objekty ekvivalentní, musí mít stejný hash.
  - Například při vyhledávání prvků v kolekci, se objekty budou porovnávat jen a pouze pokud se hashe rovnají, jinak k porovnání objektů vůbec nedojde!
  - Pokud přepíšete metodu `Equals` měli byste vždy přepsat i `GetHashCode`.
  - Metoda by navíc měla záviset pouze na neměnných slotech dané třídy, proč?
- `string ToString()` - virtuální metoda, vrací textovou reprezentaci aktuální instance.
- `Type GetType()` - metoda vracející typ aktuální instance.

Operátory `is`, `as`

- Přetypování objektu na typ jehož není instancí, vyvolá výjimku.
- Na testy zda objekt je instancí nějaké třídy máme operátory `is`, `as` a funkci `GetType`.
- `obj is MyType` - vrací `true` pokud `obj` je instance třídy `MyType` (tzn. může to být i instance nějakých jejich potomků), jinak `false`.
- `obj as MyType` - pokud je `obj` instance třídy `MyType`, vrací hodnotu `obj` přetypovanou na `MyType`. Jinak vrací `null`.

Referenční vs hodnotové datové typy zjednodušeně

- C# má dva druhy typů: referenční a hodnotové.
- Proměnné referenčného typu uchovávají odkaz na svá data a můžou mít hodnotu `null`.
- Proměnné hodnotového typu uchovávají svá data přímo a nemohou mít hodnotu `null`.
- Dvě proměnné referenčního typu mohou odkazovat na stejný objekt v paměti.
- Proměnné hodnotového typu vždy obsahují vlastní kopii dat.
- Referenční typy jsou ty deklarované pomocí: `class`, `interface`, `delegate`.
- Příklady referenčních typů: `Array`, `object`, `string`, jakákoli třída / interface / delegát které vytvoříme, IEnumerable, EventHandler, ...
- Hodnotové typy jsou ty deklarované pomocí: `struct`, `enum`.
- Příklady hodnotových typů: `int`, `byte`, `double`, `DateTime`, `TimeSpan`, jakákoli struktura / enum kterou vytvoříme, ...

Boxing a unboxing

- Boxing je proces konverze hodnotového typu na `object` popř. na libovolný interface, který daný typ implementuje.
- Boxing vlastně zabalí hodnotu hodnotového typu do `System.Object` což je referenční typ.
- Unboxing je opačný proces, kdy danou hodnotu získáme zpět rozbalením.
- Boxing je implicitní, unboxing explicitní.

```
int i = 5;
object box = 5; // boxing
int j = (int)box; //unboxing
```

- Boxing a unboxing je poměrně výpočetně náročný ve srovnání s klasickým přiřazením a má tedy dopad na rychlost.
- Lze použít např. pro heterogení kolekce, ale tento přístup nedoporučuji.

Prokletí jménem `null`

- Nejběžnější výjimka v programech v C# je `NullReferenceException`.
- Tato výjimka je vyvolána, když zkusíme přistoupit ke členu proměnné referenčního typu, která má hodnotu `null`.

```
object obj = null;
Console.WriteLine(obj.ToString()); // vyvola NullReferenceException
int i = null; // nelze zkompilovat, int je hodnotovy typ
DateTime date = null; // nelze zkompilovat, DateTime je hodnotovy
    typ
```

Operátory ??, ?., ?[]

- Existují operátory pro odlehčení prokletí `null`.
- Jsou to vlastně jen zkratky pro ustálené kontroly ref. typů na nullovost.
- `nullableExpr ?? expr` - pokud nemá `nullableExpr` hodnotu `null`, vrátí jeho hodnotu, jinak vrátí hodnotu `expr`.

```
var val1 = nullableObj ?? defValue;
var tmp = nullableObj;
var val2 = (tmp != null) ? tmp : defValue;
```

- `nullableExpr?.member` - pokud nemá `nullableExpr` hodnotu `null`, vrátí člen `member`, jinak vrátí hodnotu `null`.

```
var val1 = nullableObj?.member;
var tmp = nullableObj;
var val2 = (tmp != null) ? tmp.member : null;
```

Generika

- Rozumnější / přísnější obdoba templatů z C++.
- Naproti C++ vynucuje typovou bezpečnost.
- Můžeme parametrizovat typy, např. můžeme mít generickou třídu `List`, která přijmá jeden parametr a to zase typ, který určuje jaké hodnoty bude list uchovávat.
- Zápis pomocí špičatých závorek: `List<int>`, `Dictionary<int,string>`, ...
- Generika je použitelná jak na třídy, tak i na metody apod.

```
class MyClass<Typ1,Typ2,...> where Typ1 : Base1, Typ2 : Base2, ...
{
    // muzeme pouzivat typy Typ1, Typ2, ...
    // navic muzeme predpokladat, ze typ Typ1 je dedi po Base1,
       Typ2 Base2, ...
}
```

Generika

```
class List<T>
{
    T[] storage;
    ...
    public T GetItem(int index)
    {
        // meli bychom zkontrolovat nekolik veci, jake?
        return storage[index];
    }

    public void SetItem(int index, T item)
    {
        // meli bychom zkontrolovat nekolik veci, jake?
        storage[index] = item;
    }

}
```

Často používané kolekce

- Dnes jsou nejpoužívanější generické kolekce.
- Starší klasické kolekce jsou označeny jako zastaralé.
- Všechny implementují rozhraní `IEnumerable` tzn. lze je procházet pomocí `foreach`.
- Flexibilnější kolekce než pole je `List<T>`.
  - Zřejmě nejpoužívanější kolekce v C#.
  - Interně je aktuálně reprezentovaná jako pole.
  - Podporuje indexaci jako pole.
  - Můžeme libovolně přidávat a odebírat prvky pomocí metod `Add`, `Insert`, `Remove`, `RemoveAt`.
  - Počet prvků udává veřejné vlastnost `Count`.
  - Podporuje vyhledávání, třízení apod.
- Pro mapování klíčů na hodnoty máme tzv. slovník `Dictionary<TKey,TValue>` a podporuje indexaci pomocí daného typu klíčů.
- Hashovací tabulka je realizována jako třída `HashSet<T>`.
- Můžeme inicializovat pomocí inicializační částí (tj. výčet prvků ve složených závorkách).

Často používané kolekce

```
List<int> nums = new List<int>() { 2, 4, 6};
Console.WriteLine($"nums has {nums.Count} elements");
foreach(var num in nums)
    Console.WriteLine($"{num} ");

Dictionary<string, int> codes = new Dictionary<string, int>()
{
    { "CZ", 420 },
    { "UK", 44 },
    { "RU", 7 }
};
Console.WriteLine($"codes has {codes.Count} entries");
Console.WriteLine($"code for Czech Republic is {codes["CZ"]}");
foreach (var kvp in codes)
    Console.WriteLine($"country {kvp.Key} has a code {kvp.Value}");
```

Delegáti

- `delegate` je typ na uchování odkazu na metodu s pevně určeným seznamem parametrů a návratovým typem.
- Delegáti určují „typ“ metod.
- Jedná se o referenční typ, hodnota může být `null`.
- Vhodné pro použití na callbacky a tedy na události.

```
delegate navratovyTyp NazevDeleagata(typ1 arg1, typ2 arg2, ...);

delegate int Calculate(int a, int b);
int ComplexCalculation(int a, int b) => a + b;

int ApplyCalculation(Calculate action) => action(2,3);

int Foo()
{
    Calculate action = ComplexCalculation;
    return ApplyCalculation(action); // nebo primo ApplyCalculation
        (ComplexCalculation);
}
```

Anonymní metody, lambda výrazy

- Při vytváření anonymní funkce musíme vytvořit delegáta na daný typ metody.
- Lambda výrazy byly do C# přidány později než anonymní metody.
- Je doporučené používat lambda výrazy namísto anonymních metod pokud je to možné (to je možné téměř vždy).

```
delegate(typ1 arg1, typ2 arg2, ...) { ... }

// u lambda vyrazu lze typy vetsinou vynechat
(typ1 arg1, typ2 arg2, ...) => { ... }
```

- Při vytváření anonymních metod a lambda výrazů se zachytí proměnné v prostředí vzniku.
- U lambda výrazů můžeme často vynechat typy parametrů a kompilátor je zvládne odvodit.
- Lambda výrazy lze použít nejen na vytváření anonymních metod, ale i na tzv. *expression trees*, které se využívají zejména v `LINQ`.

Anonymní metody, lambda výrazy

```
delegate void Print(string s); // musime vytvorit typ delegata
void Foo()
{
    Print printer1 = delegate(string s) { Console.WriteLine(s); }; //
        anonymni metoda
    Print printer2 = (s) => Console.WriteLine(s); // lambda vyraz
    Print printer3 = (string s) => { Console.WriteLine(s); }; //
        lambda
    printer1("Hello");
    printer2("Hello");
    printer3("Hello");
}
```

Události

- C# má vestavěnou podporu na události.
- Události se vyvolávají obdobně jako funkce pomocí operátoru `()` nebo pomocí metod `Invoke` a `BeginInvoke`.

```
pristupnost modifikatory event TypHandleru NazevUdalosti;
```

- `TypHandleru` je delegát určující, jak má vypadat metoda, kterou lze přihlásit k odběru události.
- K odběru událostí se přihlašuje pomocí operátoru += a odhlašuje pomocí −=.

```
MyEvent += MyHandler; // prihlaseni k odberu pojmenovanou metodou
MyEvent -= MyHandler; // odhlaseni z odberu
MyEvent += (...) => { ... }; // prihlaseni k odberu anonymni
    metodou, nelze odhlasit pomoci -=
```

Události

Existuje několik konvencí jak pracovat s událostmi.

- Před vyvoláním události je potřeba zkontrolovat, zda její hodnota není `null`, tzn. jestli existuje nějaký odběratel.

```
if(MyEvent != null) ...
```

- Typ handleru je často `EventHandler<TEventArgs>`, kde `TEventArgs` je potomek třídy `System.EventArgs`, který obsahuje všechny informace o dané události.
- `EventHandler<TEventArgs>` udává, že handler musí jako první prvek přijmout `object` a jako druhý `TEventArgs`.
- Pro vyvolání události vytvoříme metodu `OnMyEvent`.
- Pokud odebíratele neplánujeme odhlašovat od události, použijeme anonymní metodu.

Události

```
class MyClass
{
    public class MyEventArgs : EventArgs
    {
        public int MyArg { get; private set; }
        public MyEventArgs(int myArg) => MyArg = myArg;
    }
    public event EventHandler<MyEventArgs> MyEvent;

    void OnMyEvent(int val)
    {
        var ev = MyEvent;
        if(ev != null) // kontrola, zda existuje nejaky odberatel
            ev(this, new MyEventArgs(val));
    }

    // pokud chceme pouze vyvolat udalost a nic jineho
    void OnMyEvent2(int arg) => MyEvent?.Invoke(this, new MyEventArgs(
        arg));
}
```

Cvičení 1

- Prozkoumejte vlastnosti a metody kolekcí `List<T>`, `HashSet<T>`, `Dictionary<TKey, TValue>`.
- Na co je každá z kolekcí vhodná?
- Jaké máme možnosti získání prvků ze slovníku?
- Upravte třídu Person tak, aby přepisovala metodu `ToString`, která bude vracet to co původně metoda `GetInfoString`.
- Vytvořte program se pro vyhledávání osoby dle rodného čísla (použijte `Dictionary` a třídu Person z předchozích seminářů, slovník naplňte několika instancemi). Od uživatele načtěte rodné číslo a vypište detail vyhledané osoby.

Cvičení 2

- Z třídy `ListInt` z minulého cvičení udělejte generickou třídu `MyList<T>`, kde `T` určuje typ prvků v kolekci.
- Dále do třídy přidejte veřejné události `ItemAdded` a `ItemRemoved`, které se vyvolají při přidání a odebrání prvku.
- Prozkoumejte interface `INotifyCollectionChanged` a třídu `ObservableCollection<T>`, popř. zajistěte, aby `MyList<T>` implementoval toto rozhraní.