



VYSOKÉ UČENÍ FAKULTA
TECHNICKÉ INFORMAČNÍCH
V BRNĚ TECHNOLOGIÍ

03 – OOP & C#

Zadání projektu WIKI - <http://1drv.ms/21qF2sM>



- Registrace týmů
 - FIT – viz WIS
 - FEKT - <http://1drv.ms/1PMv1II>
- Zadání viz [WIKI](#) - <http://1drv.ms/21qF2sM>



Obsah přednášky

- Objektivě orientované programování v C#
- Tři pilíře objektivě orientované programování
- Základní pojmy
- Interface
- Struktura
- Modifikátory přístupu
- Generika
- Základy Entity Frameworku



Objektově orientované programování

- Poprvé použito v jazyce **SIMULA 67**
- Abstrakce objektu reálného světa
- Reálný objekt (pes) lze popsat souhrnem určitých **vlastností** (délka, barva srsti...) a schopností vykonávat **činnosti** (štěkot, kousaní)
- Objekt ve smyslu OOP umožňuje sdružit *vlastnosti a činnosti*
- Činnosti jsou popsány procedurami a funkcemi, v OOP označovaných jako **metody**, které jsou součástí objektu

```
class Dog
{
    string name;
    int length;
    string color;
    void Bite(string who) {...};
    bool Bark(int howLong) {...};
}
```



Základní tři pilíře OOP

OOP umožňuje sdružovat logicky související data a kód.

- **zapouzdření (encapsulation)**
- **dědičnost (inheritance)**
- **polymorfismus (polymorphism).**

Zapouzdření

- skrytí implementačních detailů
- zvýšení modularity
- izolace nesouvisejících částí kódu

Dědičnost - pracuje s hierarchií pro zapouzdření, tedy nové objekty lze vytvářet jako potomky svých předků od nichž přebírají (dědí) vlastnosti a přidají vlastnosti nové

```
class Animal
{
    private string name;
    public void DrawIt();
}
```

```
class Dog : Animal
{
    private string color;
    public void Bite(string who);
}
```



Základní pojmy

- typ **class** - pouze "konstrukční plán" objektu
- **instance** - konkrétní objekt, instance dané třídy
- **field** – hodnota či objekt uvnitř objektu
- **property** – navenek zpřístupněný field objektu
- **method** - pojmenovaná procedura nebo funkce, zapouzdřená a patřící objektu
- identifikátor ***null*** - reference, která "neukazuje nikam"
- identifikátor ***this*** - referuje aktuální instanci objektu
- identifikátor ***base*** - umožňuje volání metody nadřazené v třídní dědičnosti



Modifikátory přístupu

- **private** - viditelné jen ze "vnitřku" třídy
- **protected** - jako **private** a navíc pro všechny její dědice
- **public** - přístupné **okudkoliv**
- **internal** – viditelné v dané assembly
- **protected internal** – viditelné v dané assembly, nebo ve zděděné třídě v jiné assembly

- Modifikátory přístupu je více než vhodné používat pro odstínění implementačních detailů a zvýšení bezpečnosti kodu.
- Pokud není modifikátor uvedený, implicitně se předpokládá nejrestriktivnější omezení.



Class - Třída

- *Nejčastější referenční typ*
- Konstrukce pro vytvoření vlastního typu sdružujícího
 - **Fields** – proměnné
 - **Properties** - vlastnosti
 - **Methods** - metody
 - **Events** - události
- Třída je „Blueprint“
- **Zapouzdřuje data i chování daného typu**
- **Statická** – jedna instance pro daný běh programu
- **Nestatická** – instance jsou tvořeny za běhu programu
- **Klíčová slova**
- Před:
 - Atributy a třídní modifikátory
 - *public, internal, abstract, sealed, static, unsafe, and partial*
- Za: *Generic type parameters, a base class, and interfaces*
- V závorkách: *Class members - methods, properties, indexers, events, fields, constructors, overloaded operators, nested types, and a finalizer*

```
//Nejjednodušší třída  
class Cat  
{  
}
```




Fields – proměnné

- *Proměnná, která je členem třídy nebo struktury*
- Readonly – není možné změnit po konstrukci
- Inicializace
 - volitelné
 - Neinicializovaný field má defaultní hodnotu
 - (0, \0, null, false)
 - Probíhá před voláním konstruktorů
 - Vícenásobná deklarace polí
- **Modifikátory**
- Statický – *static*
- Přístupový – *public, internal, private, protected*
- Dědičnosti – *new*
- Unsafe code – *unsafe*
- Pouze pro čtení – *readonly*
- Vlákňový – *volatile*

```
class Cat
{
    string _name;
    private int _livesLeft = 9;
    private readonly int
        _eyes = 9, _legs = 2;
}
```



Metody

- *Sekvenčně vykonávají akce deklarované v jejich těle*
- **Mají přístup k členským proměnným objektu**
- *Mohou*
 - přijímat parametry – hodnoty, referenční typy, **ref**
 - vrátet výsledek – v návratovém typu (return), **ref, out**

```
class Cat {  
    bool TryMeow(int loudness, out int duration){}}
```

- **Modifikátory**
- Statický – *static*
- Přístupový – *public, internal, private, protected*
- Dědičnosti – *new, virtual, abstract, override, sealed*
- Partial method - *partial*
- Unsafe code – *unsafe, extern*
- Asynchronní - *async*



Metody – Expression-bodied methods

- **Novinka C# 6**
- Metody skládající se z jednoho výrazu mohou být zapsány pouze výrazem

- Klasický zápis:

```
int Foo(int x) { return x * 2; }
```

- Expression-bodied metoda:

```
int Foo(int x) => x * 2;
```

- Metoda může mít i *prázdný* návratový typ:

```
void Foo(int x) => Console.WriteLine(x);
```



Metody – overload - přetížení

- Typ může přetěžovat metody – zachování stejného jména metody
- Překrytí parametrů metody – signatura metody musí být odlišná

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (ref double x) {...} // OK so far
void Foo (out double x) {...} // Compile-time error
```

- **Návratový typ neurčuje signaturu!**

```
void Foo (int x) {...}
int Foo (int x) {...} // Compile-time error
```

- **Přeřízené metody mohou mít různé návratové typy**

```
int Foo (int x) {...}
double Foo (double x) {...} // OK
```



Konstruktor

- Spustí inicializační kód třídy nebo struktury
- Definován jako metoda bez návratového typu a stejného jména jako konstruovaný typ
- Využívá se **dědičnosti**, konstruktory předka jsou v potomku přístupné
- Třída může mít **více** konstruktorů
- Bezparametrový konstruktor vytvořen automaticky, pokud není deklarován jiný
- Deklarace nealokuje paměť, ale pouze **statický ukazatel** na dynamicky alokované místo kde se nachází "vlastní objekt"
- **Modifikátory**
- Přístupový – *public, internal, private, protected*
- Unsafe code – *unsafe, extern*

```
public class Dog : Animal
{
    private string name;
    // constructor
    public Dog(string dogName)
    {
        name = dogName;
    }
}

...

// constructor call
var alik = new Dog("Alík");
```



Konstruktor – overloading - přetížení

- Typ může mít **více konstruktorů**
- Stejná pravidla jako přetížení metod
- Zabraňuje duplikaci kódu a zvyšuje přehlednost
- Klíčová slova **this**, **base**

```
public class Cat : Pet {  
    private readonly int _livesLeft = 9;  
    public Cat(string name):base() {  
        this.Name = name;  
    }  
    public Cat(string name, int livesLeft):this(name) {  
        this._livesLeft = livesLeft;  
    }  
}
```



Konstruktor – Inicializace - DEMO

```
public class Pet {
    public Pet(){ //5
        this.Name = "Jane Doe";//6
    }
    public string Name { get; protected set; } = "John Doe"; //4
}
```

```
public class Cat : Pet {
    private readonly int _livesLeft = 9; //1

    public Cat(string name):base(){ //3
        this.Name = name;//7
    }
    public Cat(string name, int livesLeft) : this(name){ //2
        this._livesLeft = livesLeft;//8
    }
}
```



Properties – vlastnosti - DEMO

- navenek se jeví jako jednoduchá proměnná
- jedná se o "bezpečnostní prvek,, unifikující zápis a čtení pomocí }
- přístupové metody k atributu
- odstínění implementačních detailů

property mohou být:

- automatické (automatic)
- počítané (calculated)
- metody get i set mohou využívat modifikátory přístupu, defaultně public

```
public class Pet : Animal, Iname {
    public string Name
        { get; protected set; }
        = "John Doe";
}

public class Cat {
    private readonly int _livesLeft = 9;
    private string _name;
    public new string Name {
        get {
            return _name ??
                (_name = $"{base.Name} Cat");
        }
        private set { _name = value; }
    }
    public int Prize => _livesLeft*10;
}
```




Abstraktní třídy - DEMO

- instance třídy deklarované jako **abstraktní** nemůže být nikdy vytvořena, lze vytvořit pouze její potomky
- potomek tuto implementaci musí poskytnout, pokud není sám opět abstraktní
- **abstraktní** členy jsou jako virtualní členy, pouze neposkytují defaultní implementaci
- abstraktní třída nemůže být „**sealed**“, viz. dále

```
public abstract class Animal
{
    public abstract void Draw();
}
```

```
public class Dog : Animal
{
    private int numberOfLegs;
    public override void Draw()
    {
        /*implementace draw dog*/
    }
}
```



Kompatibilita typů - DEMO

- umožňuje **efektivní** využívání virtuálních metod
- jedná se o **kompatibilitu ukazatelů na instance tříd**
- odpovídá do jakého ukazatele na instanci třídy lze přiřadit ukazatel na instanci jiné třídy
- kompatibilní se třídou jsou **všichni její dědicové**, jde o implicitní **upcast**, který vytvoří referenci na báзовou třídu z reference potomka, je vždy úspěšný
- **downcast** vytvoří referenci potomka z báзовé třídy, nemusí být vždy úspěšný

```
Dog dog1 = new Dog();  
Animal a1 = dog1;    // Upcast  
Dog dog2 = (Dog)a1;  // Downcast  
dog2 == a1;         // True  
dog2 == dog1;       // True
```

```
class Cat : Animal  
Cat cat1 = new Cat();
```

```
// Upcast always succeeds  
Animal a2 = cat1;
```

```
//Downcast fails:a2 is not a Dog  
Dog dog3 = (Dog)a2;
```



polymorfismus (polymorphism)

doslova "mnohotvarost,,

- schopnost přizpůsobit chování objektu (při volání jeho funkcí) konkrétní instanci tohoto objektu
- jde o mechanismus volání metod svázaný s **dědičností** umožňující pod stejným jménem volat **různé** metody stejného jména

```
public class Pet
{
    public virtual void Draw {...}
}
public class Dog : Pet
{
    public override void Draw {...}
}
public class Cat : Pet
{
    public override void Draw {...}
}
```



Polymorfismus, virtuální metody II

- jejich aktivace probíhá pomocí mechanismu **pozdní vazby**, která se vytváří až běhu
- **late binding x early binding**
- virtuální mohou být:
 - metody
 - vlastnosti
 - indexery
 - události



Operátory IS / AS - DEMO

Operátor AS

- provádí **downcast**, který porovná s hodnotou **null**
- je užitečný ve spojení s následným testem na **null** hodnotu, například namísto výjimky, pokud downcast selže

```
var a = new Animal();  
Console.WriteLine(a is Dog ?  
    "a is a Dog" :  
    "a is not a Dog");
```

Operátor IS

- zkouší zda jsou reference kompatibilní, tedy zda objekt dědí z dané třídy, či interface
- jde o obvyklý test před downcastem

```
var a = new Animal();  
var dog = a as Dog; //Dog == null  
Console.WriteLine(dog != null ?  
    "dog is a Dog" :  
    "dog is not a Dog");
```



Klíčová slova **sealed**, **base**

Klíčové slovo **sealed**

- označuje třídu ze které již nejde dědit
- lze aplikovat i na metodu, kterou nelze dále překrýt (override)

Klíčové slovo **base**

- Slouží k přístupu k překryté (override) metodě, member z potomka
- Často se používá při volání konstruktoru báze třídy

```
class Pet
{
    protected string name;
}

sealed class Cat : Pet
{
    public string CatName
    {
        get
        {
            return base.name + " Cat";
        }
    }
}

//Compile-time error
public class Kitten : Cat {}
```



System.Object

- object (**System.Object**) je společný předek všech typů
- každý objekt lze přetypovat na **System.Object**
- Obsahuje následující metody:
 - ToString()
 - Equals()
 - GetHashCode()
 - GetType()
- Pro zjištění typu objektu lze použít:
- **Object.GetType()** vyhodnocuje se za běhu programu
- operátor **typeof** se vyhodnocuje staticky v době překladu

```
Dog d = new Dog();
```

```
Console.WriteLine(d.GetType().Name);  
// Returns "Dog"
```

```
Console.WriteLine(typeof(Dog).Name);  
// Returns "Dog"
```

```
Console.WriteLine(d.GetType() ==  
typeof(Dog));  
// Returns true
```



Finalizer

- metody jenž se vykonávají na nereferecované instanci před tím než garbage collector uvolní paměť
- obdoba destrukturu z C++
- jde vlastně o přepsání metody *Finalize()* třídy *Object*, kompilér si jej přeloží jako:

```
protected override void Finalize()  
{  
    ...  
    base.Finalize();  
}
```

```
class Dog  
{  
    ~Dog()  
    {  
        // Cleanup code  
        ...  
    }  
}
```




Částečné třídy (Partial classes)

- umožňují rozdělit třídu do více souborů
- typicky jeden autogenerated, druhý ručně psaný
- klasické použití pro autogenerated design formuláře a v druhém souboru jeho kod
- existují i **partial methods**

```
// Dog1Gen.cs - auto-generated
partial class Dog1Form {
    public Dog1Form() {
        this.Bark();
    }
    partial void Bark();
}

// Dog1Form.cs - hand-authored
partial class Dog1Form {
    partial void Bark() {
        Console.WriteLine("Bark");
    }
}
```



Boxing / Unboxing

Když přetypováváte mezi **objektem** a **hodnotovým typem**, CLR musí provést operaci pro konverzi mezi hodnotovým a referenčním typem – boxing / unboxing

Tyto operace „něco stojí“, tedy v případě jejich velkého počtu snižují časovou efektivitu

```
int x = 9;
```

```
// Box the int  
object obj = x;
```

```
// Unbox the int  
int y = (int)obj;
```



Struktury (struct)

Jsou podobné třídě s následujícími rozdíly:

- struktura je **hodnotový typ**, *třída referenční*
- struktury implicitně dědí z **System.ValueType**
- *nepodporují dědičnost*
- struktury mohou mít jakékoliv členy jako třídy, s výjimkou **bezparametrického konstruktoru**, **finalizeru** a **virtuálních členů**
- konstruktor vždy musí inicializovat všechny členy struktury
- je zakázána inicializace členu struktury v její deklaraci

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

```
Point p1 = new Point (1, 1);
// p1.x and p1.y will be 1
Point p2 = new Point ();
// p2.x and p2.y will be 0
```



Enums, Flags - DEMO

- **enum** je hodnotový typ umožňující vytvořit skupinu pojmenovaných numerických hodnot (int, 0,1...)
- **underlying type** je možné změnit
- jako **flag** je označen typ enum, jeho proměné mohou následně mít víc hodnot

```
private enum HorseColor {  
    Siml, Palomino, Ryzak }
```

```
HorseColor color = HorseColor.Siml;  
int i = (int) HorseColor.Ryzak;  
Enum.TryParse(string, out value);
```

```
[Flags]
```

```
public enum HorseType { None=0,  
    Racing=1, Breeding=2, ForSosages=4 }
```

```
HorseType type= HorseType.Racing |  
                HorseType.Breeding;  
    type |= HorseType.ForSosages;  
Console.WriteLine(horse.Type);  
//Racing, Breeding, ForSosages
```



Rozhraní (Interfaces)

- rozhraní poskytuje pouze specifikaci, ne konkrétní implementaci svých členů
- členy interface jsou všechny veřejné
- třída či struktura může implementovat více rozhraní
- prvky rozhraní jsou implementovány třídami, které rozhraní implementují
- rozhraní může obsahovat metody, vlastnosti, události a indexery

IEnumerator `interface`

`// definováno v
System.Collections`

```
public interface IEnumerator  
{  
    bool MoveNext();  
    object Current { get; }  
    void Reset();  
}
```



Proč používat interface

- využívejte dědičnosti pro typy, které přirozeně sdílí svoji implementaci
- vyžijte interface pro typy, jenž mají nezávislé implementace
- je možné, aby jedna třída implementovala více rozhraní
- podle tohoto pravidla můžeme říci, že hmyz a ptáci sdílí implementaci, tedy mohou zůstat třídami
- naopak létavci a masožravci mají nezávislé způsoby příjmu potravy, proto budou rozhraními
- **interface IFlying {}**
- **interface ICarnivore {}**

```
abstract class Animal { }
abstract class Bird : Animal { }
abstract class Insect : Animal { }
abstract class Flying : Animal { }
abstract class Carnivore : Animal { }

// Concrete classes:
class Ostrich : Bird { }
class Eagle : Bird, FlyingCreature,
Carnivore
{ } // Illegal

class Bee : Insect, FlyingCreature
{ } // Illegal
class Flea : Insect, Carnivore
{ } // Illegal
```



Vytváření znovupoužitelného kódu I

C# má dva mechanismy pro vytváření znovupoužitelného kódu

- dědičnost
- generika
- kompozice

příklad: zásobník pro různé datové typy

- 1) hardcoded pro každý typ (duplikace kódu)
- 2) využít typu object (boxing, downcasting)
- 3) generika

příklad: ObjectStack pro celá čísla

```
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj)
    {
        data[position++] = obj;
    }
    public object Pop()
    {
        return data[--position];
    }
}
stack.Push ("s");
// Wrong type, but no error!
int i = (int)stack.Pop();
// Downcast - runtime error
```



Vytváření znovupoužitelného kodu II

- dědičnost vyjadřuje znovupoužitelnost
bázového typu, generika umožňují
používání šablon
- generika také zvyšují typovou
bezpečnost a snižují počet
přetypování a boxingu
- existují generické interfaces
parametry mohou být omezeny typy:
 - where T : base-class
 - where T : interface
 - where T : class
 - where T : struct
 - where T : new()
 - where U : T

```
public class Stack<T>
{
    int position;
    T[] data = new T[100];

    public void Push (T obj)
    {
        data[position++] = obj;
    }

    public T Pop()
    {
        return data[--position];
    }
}
```




Generické metody

- pomocí generických metod je možno implementovat spoustu základních algoritmu univerzální cestou
- generické metody obsahují ve své signatuře též typ parametru
- generické metody mohou obsahovat více generických parametrů

```
static void Swap<T> (ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

rozlišujeme:

- otevřený typ – Stack<T>
- uzavřený typ – Stack<int>

za běhu jsou všechna generika uzavřená

Covariance

umožňuje používat konkrétnější typ, než byl původně zadán

proměnné typu

`IEnumerable<Base>` lze přiřadit instanci `IEnumerable<Derived>`

Contravariance

umožňuje používat obecnější (méně odvozený) typ, než byl původně zadán

proměnné typu

`IEnumerable<Derived>` lze přiřadit instanci `IEnumerable<Base>`

Invariance

znamená, že můžete použít pouze původně zadaný typ, parametr invariantního obecného typu není ani kovariantní, ani kontravariantní

proměnné typu

`IEnumerable<Derived>` nelze přiřadit instanci `IEnumerable<Base>` a naopak



Poděkování

- **Autoři původních materiálů k IW5, XMW5 2015**
- Ing. Radek Gajdušek
- Ing. Zdeněk Jurka
- **Ing. Martin Minařík, Ph.D.**
- Ing. Jaroslav Nečas
- Bc. Václav Pachta
- Mgr. Martin Procháska