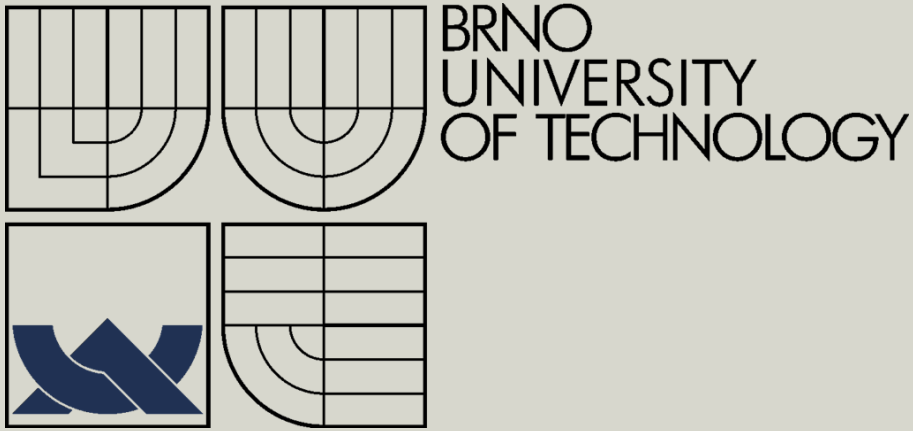


SIEMENS

03 – OOP & C#

SIEMENS

Obsah přednášky

- objektově orientované programování v C#
- tři pilíře objektově orientované programování
- interface
- struktura
- modifikátory přístupu
- generika

SIEMENS

Objektově orientované programování

- poprvé použito v jazyce **SIMULA 67**
- objekt je 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 vyjádřit oba uvedené typy charakteristik společně.
- vlastnosti – běžné datové položky, Činnost jsou popsány procedurami a funkcemi, v OOP označovaných jako **metody**, které jsou součástí objektu

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

SIEMENS

Základní pojmy

- typ **třída** (class) - pouze "konstrukční plán" objektu
- **instance** - konkrétní objekt
- **field** – hodnota či objekt uvnitř objektu
- **property** – navenek zpřístupněný field objektu
- **metoda** - pojmenovaná procedura nebo funkce, zapouzdřená a patřící objektu
- identifikátor ***null*** - adresa, která "neukazuje nikam"
- identifikátor ***this*** - implicitní parametr, který v rámci metody referuje na objekt, ze kterého byla metoda vyvolána

přetěžování metod - překrytí parametrů metody

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

SIEMENS

Základní tři pilíře OOP

OOP umožnuje 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);
}
```

SIEMENS

Konstruktor

- při deklaraci objektu se nealokuje přímo paměť pro objekt, ale pouze **statický ukazatel** ukazující na dynamicky alokované místo v paměti kde se nachází "vlastní objekt"
- inicializuje **atributy**
- využívá se **dědičnosti,** konstruktory předka jsou v potomku přístupné
- třída může mít **více** konstruktorů
- pokud není deklarován žádný konstruktor, C# automaticky vytváří bezparametrický

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

...

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

SIEMENS

Polymorfismus, virtuální metody I

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 Animal
{
  private string name;
  public virtual void Draw {…}
}
public class Dog : Animal
{
  private int numberOfLegs;
  public override void Draw {…}
}
public class Cat : Animal
{
   private int numberOfTails;
   public override void Draw {…}
}
```

SIEMENS

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

SIEMENS

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

Poznámka: modifikátory přístupu je více než vhodné používat pro odstínění implementačních detailů a zvýšení bezpečnosti kodu.

SIEMENS

Abstraktní třídy

- instance třídy deklarované jako **abstraktní** nemůže být nikdy vytvořena, lze vytvořit pouze její potomky
- potomek tuto implemetaci 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*/
  }
}
```

SIEMENS

Kompatibilita typů

- 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 bázovou třídu z reference potomka, je vždy úspěšný
- **downcast** vytvoří referenci potomka z bazové 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;
Animal a2 = new Dog();

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

SIEMENS

Operátory IS / AS

Operátor AS

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

```
Animal a = new Animal();
Dog dog = a as Dog;//Dog is null

if (a is Dog)
{
  Console.WriteLine(((Dog)a).Color);
}
```

Operátor IS

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

```
// Dog is null;
if (a == null)
{
  Console.WriteLine("a je dog");
}
```

SIEMENS

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 bázové třídy

```
class Animal
{
  protected string name;
}

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

SIEMENS

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
```

SIEMENS

Finalizer

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

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

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

SIEMENS

Částečné třídy (Partial classes)

- umožnují rozdělit třídu do více souborů
- typicky jeden autogenerovaný, druhý ručně psaný
- klasické použití pro autogenerovaný design formuláře a v druhém souboru jeho kod

```
// Dog1Gen.cs - auto-generated
partial class Dog1Form
{
  ...
}
// Dog1Form.cs - hand-authored
partial class Dog1Form
{
  ...
}
partial metoda
```

SIEMENS

Jednoduché vlastnosti (Properties)

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

property mohou být:

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

```
public class Dog
{
  private string name;
  public string Name
  {
   get { return name; }
   private set { name = value; }
  }

  public int Lenght { get; set; }

  private int price, numberOfLegs;
  public int Worth
  {
    get {return price * numberOfLegs;}
  }
}
```

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;
```

SIEMENS

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 vyjímkou bezparametrického konstruktoru, finalizeru a vitruálních členů
- konstruktor musí inicializovat všechny členy třídy
- je zakázana 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 ();
// p1.x and p1.y will be 0

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

SIEMENS

Enums, Flags

- **enum** je hodnotový typ umožnující 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);
```

SIEMENS

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();
}
```

SIEMENS

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
```

SIEMENS

Vytváření znovupoužitelného kodu I

C# má dva mechanismy pro vytváření znovupoužitelného kódu

- dědičnost
- generika
- kompozice

příklad: zásobník pro různé datové typy

1) hardcoded pro každy typ (duplikace kodu)
2) využit 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];
  }
}
```

SIEMENS

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

SIEMENS

Variance

Covariance
umožňuje používat konkrétnější typ, než byl původně zadán

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

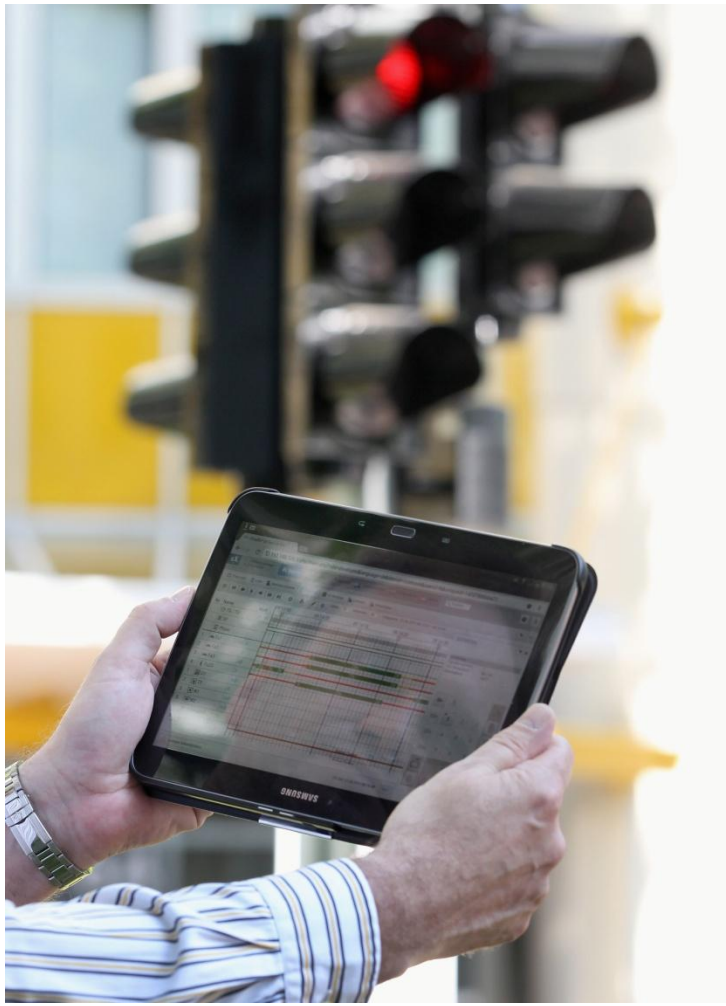
Contravariance
umožňuje používat obecnější (méně odvozený) typ, než byl původně zadán

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>` lze přiřadit instanci `IEnumerable<Base>` a naopak

Contact page



Ing. Martin Minařík, Ph.D.

Siemens CT DC / Sitraff Team

Olomoucká 7/9
618 00 Brno
Česká republika

E-mail: martin.minarik@siemens.com