

C# - Objektové typy

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Objektové typy

- Struktury
- Třídy
 - Třída – Class
 - Objekty
- Složky třídy
 - Konstruktory, finalizér
 - Atributy, vlastnosti
 - Vlastnost – Property
- Metody

2 Objektové typy – dědičnost, polymorfizmus

- Dědičnost
- Polymorfizmus
 - Rozhraní – interface
 - Základní rozhraní v knihovně

Objektové typy

Struktury

Struktura – struct

struct

- **hodnotový typ** (předává se hodnota - nikoliv reference)
- může obsahovat parametrické konstruktory, atributy, vlastnosti, metody, operátory, ...
- nemůže definovat bezparametrický konstruktor, finalizér (destruktor)
- **nemůže dědit** z předka
 - ▶ vždy dědí z `System.ValueType`, dědící z `System.Object`
- může realizovat **rozhraní**
- instanci je možné vytvořit bez **new**

Struktura – struct



```
struct Student
{
    public string FirstName;
    public string LastName;

    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Struktura – struct



```
static void TestStudent()
{
    Student stPetraKratka = new Student("Petra", "Kratka");

    Student stJitkaMlada = new Student()
    {
        FirstName = "Jitka",
        LastName = "Mlada"
    };

    Student stPetrMaly = new Student();
    stPetrMaly.FirstName = "Petr";
    stPetrMaly.LastName = "Maly";

    Student stJanVelky; // pouze u struktur, nelze u tříd
    stJanVelky.FirstName = "Jan";
    stJanVelky.LastName = "Velky";
}
```

Třídy

Třídy – class

class

- **referenční typ** (předává se reference)
- **jednoduchá dědičnost**
 - ▶ může dědit z jednoho předka
 - ▶ rozlišuje se časná a pozdní vazba u metod
- může realizovat **rozhraní**
 - ▶ libovolný počet
 - ▶ implicitní/explicitní realizace rozhraní
- může obsahovat konstruktory, finalizér, atributy, vlastnosti, události, metody, operátory, vnořené typy, ...
- objekty jsou vytvářeny na haldě (heap)
- paměť spravuje **garbage collector**

Třídy – porovnání s Javou/C++

- jednoduchá dědičnost
 - ▶ logika dle javy, syntaxe dle C++
- polymorfismus
 - ▶ v C++ je nutné řešit časnou/pozdní vazbu (**virtual**), v C# také (**virtual**, **override**), Java toto zjednodušuje – vše použije pozdní vazbu automaticky
 - ▶ systém v C# je v zásadě složitější, umožňuje přerušit řetězec pozdní vazby **new virtual**
- rozhraní
 - ▶ neřeší se viditelnost složek v rozhraní
 - ▶ dva způsoby realizace – implicitní (standardní), explicitní (objekt je nutné přetypovat na typ rozhraní)

Třídy – porovnání s Javou/C++

- atributy a gettery/settery
 - ▶ C# zavádí pojem **vlastnost (property)** a speciální syntax nahrazující gettery, settery

Třída – Class

Třída – class

```
[viditelnost] [modifikátory] class NazevTridy [dědičnostARozhraní] {  
    [složkyTřidy]...  
}
```

Viditelnost třídy

- **internal** (výchozí) – viditelná v rámci assembly
- **public** – viditelná z ostatních assembly

Modifikátory

- **abstract** – abstraktní třída (nelze vytvořit instanci, obsahuje abstraktní složky)
- **static** – statická třída (nelze vytvořit instanci, pouze statické složky)
- **sealed** – zapečetěná třída (nelze z ní dědit)
- **partial** – třída rozdělená do více souborů

Složky třídy

- konstruktory
- konstanty
- atributy
- finalizéry
- metody
- vlastnosti
- indexery
- operátory
- události
- delegáty
- třídy
- rozhraní
- struktury

Objekty

Objekt

- vytvořen pomocí **new**
- předáván referencí, paměť spravuje GC



```
// nastavení reference na null
```

```
Osoba neexistujiciOsoba = null;
```

```
// volání bezparametrického konstrukturu
```

```
Osoba osobaVychazi = new Osoba();
```

```
// volání parametrického konstrukturu
```

```
Osoba osobaParametricka = new Osoba("Franta", "Maly");
```


- C# umožňuje kombinovat volání konstruktoru a inicializaci vlastností



```
Osoba osobaZvlastni = new Osoba() {  
    Jmeno = "Franta",  
    Prijmeni = "Maly"  
};
```

// uvedený kód je ekvivalentem:

```
Osoba osobaZvlastni = new Osoba();  
osobaZvlastni.Jmeno = "Franta";  
osobaZvlastni.Prijmeni = "Maly";
```

Objekt – porovnávání

- ==, Equals – porovnání referencí/obsahu (nezávisle přetížitelné)
- **object**.ReferenceEquals() – vždy porovná shodu referencí



```
Osoba mojeOsoba = new Osoba();  
Osoba ciziOsoba = mojeOsoba;
```

```
//var wl = (Action<string>)Console.WriteLine;  
wl($"==: {mojeOsoba == ciziOsoba}");  
wl($"Equals: {mojeOsoba.Equals(ciziOsoba)}");  
wl($"ReferenceEquals: {object.ReferenceEquals(mojeOsoba, ciziOsoba)}");  
// True, True, True
```

```
ciziOsoba = new Osoba();
```

```
wl($"==: {mojeOsoba == ciziOsoba}");  
wl($"Equals: {mojeOsoba.Equals(ciziOsoba)}");  
wl($"ReferenceEquals: {object.ReferenceEquals(mojeOsoba, ciziOsoba)}");  
// True/False, True/False, False
```

Ukázka přetěžování Equals a operátorů ==, !=

```
class Osoba
{
    public string Jmeno { get; set; }
    public string Prijmeni { get; set; }

    public override bool Equals(object obj)
    {
        if (obj == null || GetType() != obj.GetType())
            return false;

        Osoba osoba = (Osoba)obj;
        return Jmeno == osoba.Jmeno && Prijmeni == osoba.Prijmeni;
    }

    public override int GetHashCode()
    {
        return 11 * Jmeno.GetHashCode() + Prijmeni.GetHashCode();
    }
}
```

```
public static bool operator ==(Osoba osa, Osoba osb)
{
    return osa.Equals(osb);
}

public static bool operator !=(Osoba osa, Osoba osb)
{
    return !(osa == osb);
}
}
```

Přetěžování Equals, ==, !=

- pokud to má smysl přetižte Equals (a GetHashCode) u hodnotových i referenčních typů
- přetižte ==
 - ▶ pokud se jedná o hodnotový typ
 - ▶ pokud se jedná o referenční „základní“ typ (Point, **string**, BigInteger) nebo pokud přetěžujete operátory +, -, ...

Vnořené třídy a struktury

- třída může obsahovat vnořené třídy a struktury
- mezi nadřazenou a vnořenou třídou neexistuje žádné spojení
 - ▶ chování odpovídá v Javě vnořené třídě definované jako statické
 - ▶ stejné chování mají vnořené typy v C++
- ve vnořených typech se, ale uplatňuje genericita nadřazeného prvku
- objekt vnořeného typ se vytváří pomocí nadřazeného typu, nikoliv instance
 - ▶ `Outer.Inner instance = new Outer.Inner(...);`

×

```
class Outer { private int i; class Inner { ... } }
```

```
// Inner::method
```

```
void Method() {
```

```
    // všechny zápisy jsou neplatné — neexistuje spojení mezi objekty
```

```
    Outer.this.i = 0;
```

```
    Outer.i = 0;
```

```
    i = 0;
```

```
}
```

Složky třídy

Viditelnost složek

- **public** – veřejná (všude)
- **private** – privátní (pouze definující třída), výchozí pro složky tříd a struktur
- **protected** – chráněná (třída nebo její potomci)
- **internal** – interní (assembly)
- **protected internal** – sjednocení **protected** a **internal** (assembly a potomci (i z jiného assembly))
- **private protected** – **protected**, ale pouze u potomků ve stejném assembly

Konstruktor, finalizér

Konstruktory

[viditelnost] [modifikátory] NázevTřídy([parametry]) [voláníJiné↔
hoKonstrukturu]



```
class Student
{
    // Bezparametrický konstruktor – volá se při new Student()
    public Student() { }

    // Parametrický konstruktor – volá se při new Student(string, string)
    public Student(string jmeno, string prijmeni) { }

    // Statický konstruktor – volá se při zavedení typu do paměti
    static Student() { }
}
```

Finalizéry

```
~NázevTřídy() { ... }
```

- slouží k uvolnění prostředů
- implicitně volá metodu **object**.Finalize
- používán spíše výjimečně, není zaručeno, kdy dojde k jeho zavolání
- pro řízené uvolnění prostředků existuje rozhraní IDisposable



```
public class Auto
{
    ~Auto()
    {
        // ...
    }
}
```

Atributy, vlastnosti

- Atribut (Attribute) – Java, C++, C#

- ▶ má identifikátor, datový typ, přístupová práva
- ▶ alokuje místo v paměti pro uložení hodnoty datového typu (představuje datovou složku)
- ▶ **private string** name;

- Vlastnost (Property) – C#

- ▶ má identifikátor, datový typ, přístupová práva
- ▶ definuje getter a setter, jak přistoupit k datové složce
- ▶ není atributem, neobsahuje data!
- ▶ **public string** Name { **get** { **return** name;} **set** { name = value; } }

Atributy (datové složky)

```
[viditelnost] [modifikátory] datovýTyp názevAtributu [inicializér];
```

Modifikátor

- **static** – statická složka (neváže se na instanci)
- **const** – konstanta, musí být nastavena v inicializéru
- **readonly** – konstanta, musí být nastavena v konstruktoru

Statické složky (atributy, metody)

✓

```
class Osoba
{
    public static int PocetInstanci;

    public Osoba()
    {
        PocetInstanci++;
    }
}
```

✓

```
int pocet = Osoba.PocetInstanci;
```

Statická třída



```
static class Matematika
{
    public const double Pi = 3.141592;

    public static double Odmocnina(double hodnota)
    {
        return Math.Sqrt(hodnota);
    }
}
```



```
double pi = Matematika.Pi;
double sqrt = Matematika.Odmocnina(72);
```



```
Matematika matematika = new Matematika();
```

const/readonly složky



```
class IdGenerator
{
    const int prvniId = 123;
    readonly string prefixId;

    public IdGenerator(string prefix)
    {
        // const nelze inicializovat v konstruktoru
        // prvnild = 100;

        // readonly musí být inicializován v konstruktoru
        prefixId = prefix;
    }
}
```


Vlastnost – Property

Vlastnosti

- definují getter a setter pro přístup k datové složce
- obyčejná vlastnost neobsahuje data
- uvnitř setteru je nastavovaná hodnota předána skrze klíčové slovo `value`



```
class Student
{
    // atribut — uchovává hodnotu
    private string firstName;
    // vlastnost — definuje pravidla pro přístup k atributu
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
}
```

Vlastnosti



```
class Student
{
    private string netID;
    public string NetID
    {
        get { return netID; }
        set
        {
            if (value.StartsWith("st") && value.Length == 7)
                netID = value;
        }
    }
}
```

Vlastnosti – zkrácený zápis

C# 7

- tělo metody/getteru/setteru lze zkrátit na výraz pomocí =>



```
private string firstName;  
public string FirstName  
{  
    get => firstName;  
    set => firstName = value;  
}
```

Vlastnosti – viditelnost

- lze definovat viditelnost operací
 - ▶ pokud není uvedena u konkrétní operace, použije se viditelnost uvedená u vlastnosti



```
private string firstName;  
public string FirstName  
{  
    private get => firstName;  
    protected set => firstName = value;  
}
```

Vlastnosti – pouze setter/getter

- lze definovat pouze setter/getter

✓

```
private List<Student> studenti;  
public int PocetStudentu  
{  
    get => studenti.Count;  
}
```

Vlastnosti – automaticky implementované vlastnosti

- kompilátor na pozadí vytvoří atribut pro uložení hodnoty
- atribut (datová složka) je neviditelná a není přímo přístupná
- není možné upravit chování **get** nebo **set** operace, lze upravit viditelnost



```
public string FirstName { get; set; }  
public string MiddleName { get; set; }  
public string LastName { get; set; }
```

// od C# 6 – lze automaticky impl. vlastnosti rovnou inicializovat

```
public string NetID { get; set; } = "st12345";
```

// od C# 6 – lze definovat read-only auto prop. s inicializací v konstruktoru

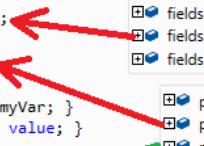
```
public uint StudentID { get; }
```

//// konstruktor

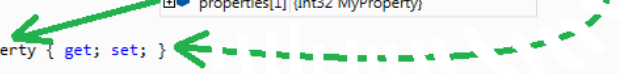
//// StudentID = 12345;

```
class TestProp
```

```
{  
    private int myVar;  
  
    public int MyVar  
    {  
        get { return myVar; }  
        set { myVar = value; }  
    }  
  
    public int MyProperty { get; set; }  
}
```



fields	{System.Reflection.FieldInfo[2]}
fields[0]	{Int32 myVar}
fields[1]	{Int32 <MyProperty>k__BackingField}



properties	{System.Reflection.PropertyInfo[2]}
properties[0]	{Int32 MyVar}
properties[1]	{Int32 MyProperty}

```
class Program
```

```
{  
  
    static void Main(string[] args)  
    {  
        var properties = typeof(TestProp).GetProperties();  
        var fields = typeof(TestProp).GetFields(BindingFlags.NonPublic | BindingFlags.Instance);  
    }  
}
```


Javism – gettery a settery

×

```
// String — klíčové slovo "string"  
// atribut — místo vlastnosti  
private String name;  
  
// metoda jako getter a setter místo vlastosti  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}
```

✓

```
// "string"  
// veřejná vlastnost — začíná velkým písmenem  
public string Name { get; set; }
```

Vytváření vlastností ve Visual Studiu

- vlastnosti lze velmi efektivně vytvářet pomocí připravených code-snippets
- `prop<TAB><TAB>` – vytvoří automaticky implementovanou vlastnost (TAB - přepíná mezi jednotlivými editovatelnými částmi)
- `propg<TAB><TAB>` – stejné jako předchozí, ale je nastaveno **private set**
- `propfull<TAB><TAB>` – vytvoří vlastnost a backing atribut pro ní
- `prodp<TAB><TAB>` – dependency property
- `propa<TAB><TAB>` – attached dependency property

Metody

Metody

```
[viditelnost] [modifikátory] typNávratovéHodnoty názevMetody([↔  
parametry]) těloMetody
```

těloMetody:

=> výraz

{ [příkazy] }

Modifikátory

- **static** – statická metoda
- **new** – zakrývání metody z předka
- **virtual** – virtuální metoda (polymorfizmus)
- **override** – přetížená metoda (polymorfizmus)
- **abstract** – abstraktní metoda (polymorfizmus)
- **async** – asynchronní metody

Metody – příklad



```
class Student
{
    public string Name { get; set; }

    public void SayHello()
    {
        Console.WriteLine($"Hello, I'm {Name}");
    }
}
```



```
Student peter = new Student()
{
    Name = "Peter"
};

peter.SayHello();
```

Metody – příklad zkráceného zápisu



```
class Student
{
    public string Name { get; set; }

    public void SayHello() => Console.WriteLine($"Hello, I'm{Name}");
}
```

Metody – výchozí hodnota parametru

- parametry mohou mít uvedenou výchozí hodnotu



```
class MyMath
{
    public const double Pi = 3.141592;

    public static double GetPowerOfPi(int power = 1)
    {
        return Math.Pow(Pi, power);
    }
}
```



```
double pi = MyMath.GetPowerOfPi();
double piSquare = MyMath.GetPowerOfPi(2);
```

Metody – ref

- parametr je možné předat odkazem (referencí)
 - u parametru i u hodnoty argumentu je nutné uvést **ref**



```
class Toolkit
{
    public static void Increment(ref int value)
    {
        value++;
    }
}
```



```
int intValue = 0;
Toolkit.Increment(ref intValue);
Console.WriteLine($"{intValue}");
```


Metody – out

- parametr může být výstupní
 - ▶ u parametru i u hodnoty argumentu je nutné uvést **out**



```
class AuthenticationService
{
    public static bool Authentize(string password, out string username)
    {
        if (password == "secr3tP4ssw0rd!")
        {
            username = "admin";
            return true;
        }

        username = null;
        return false;
    }
}
```

Metody – out



```
string username;  
if(AuthenticationService.Authenticate("password", out username))  
{  
    Console.WriteLine($"Authenticated as {username}");  
}
```



```
// C# 7 – podporuje deklarovat proměnnou v místě volání  
if(AuthenticationService.Authenticate("password", out string username))  
{  
    Console.WriteLine($"Authenticated as {username}");  
}
```

Metody – params

- je možné volat metodu s libovolným počtem parametrů (stejného typu)
 - ▶ předáváno jako parametr typu pole s mod. **params**
 - ▶ **params** parametr musí být uveden jako poslední v seznamu parametrů



```
public static int SumArguments(params int[] arguments)
{
    int sum = 0;
    foreach (var item in arguments)
    {
        sum += item;
    }

    return sum;
}
```



```
int result = Summer.SumArguments(10, 20, 30, 40, 50, 60, 70);
```

Metody – pojmenované argumenty

- při volání metody je možné specifikovat argumenty v libovolném pořadí, pokud je uveden jejich název
 - ▶ zapisuje se jako: názevParametru: hodnota



```
public static void AddProduct(string productName, int count = 1,  
    double weight = 0, string description = "")
```



```
AddProduct("Toy", 10, 1.2, "Kid's toy");  
AddProduct("Toy", count: 10, weight: 1.2);  
AddProduct("Toy", description: "Kid's toy", count: 10);  
// od C# 7.2:  
AddProduct(productName: "Toy", 10, description: "Kid's toy");
```

Metody – ref return

- z metody je možné vracet referenci (C# 7, mod. **ref**)



```
public static ref int FindGreaterThanOr(int[] array, int condition)
{
    for (int i = 0; i < array.Length; i++)
        if (array[i] > condition)
            return ref array[i];

    throw new Exception("Not found");
}
```



```
int[] array = { 1, 2, 5, 15, 32, 64 };
Console.WriteLine($"{string.Join(" ", array)}");
ref int value = ref FindGreaterThanOr(array, 10);
value += 100;
Console.WriteLine($"{string.Join(" ", array)}");
```

Metody – rozšiřující (extension) metody

- (statická) metoda se tváří jako (instanční) metoda jiné třídy
 - ▶ rozšířená třída je uvedena jako první parametr metody s mod. **this**
 - ▶ metoda se volá přímo nad objektem rozšířené třídy
- poprvé masivně použito pro realizaci LINQ



```
class Student
{
    public string Name { get; set; }
}

static class StudentExtension
{
    public static void SayHello(this Student student, string weather)
    {
        Console.WriteLine($"Hello, I'm {student.Name} and it's {weather}");
    }
}
```

Metody – rozšiřující (extension) metody

✓

```
Student student = new Student()
{
    Name = "Peter"
};

student.SayHello("sunny");
//Hello, I'm Peter and it's sunny
```

Objektové typy – dědičnost, polymorfizmus

Dědičnost

Dědičnost, polymorfismus

- jednoduchá dědičnost – je možné dědit z jednoho předka (Java)
 - ▶ syntaxe připomíná C++
 - ▶ pokud není specifikován předek, třída automaticky dědí z `System.Object`
- vícenásobná realizace rozhraní (Java)
- explicitní rozlišování polymorfních metod – časná/pozdní vazba (C++)

Konstrukce objektu

- **new** `System.Windows.Forms.Form()`
- proces postupného volání konstruktorů
 - ▶ `System.Object`
 - ▶ `System.MarshalByRefObject`
 - ▶ `System.ComponentModel.Component`
 - ▶ `System.Windows.Forms.Control`
 - ▶ `System.Windows.Forms.ScrollableControl`
 - ▶ `System.Windows.Forms.ContainerControl`
 - ▶ `System.Windows.Forms.Form`
- poté je objekt zkonstruován a lze jej používat

Dědičnost

```
[viditelnost] [modifikátory] class NazevTridy [dědičnostARozhraní] {  
    [složkyTřídy]...  
}
```

dědičnostARozhraní:
: názevPředkaNeboRozhraní, ...



```
class Person  
{  
    public string Name { get; set; }  
}  
  
class Student : Person  
{  
    public string StudentID { get; set; }  
}
```

Dědičnost – příklady

```
class Object { }  
class LivingEntity { }  
class Person : Object { }
```

```
interface IComparable { }  
interface ICloneable { }
```



```
class Student : Person { }  
class Student : Person, IComparable, ICloneable { }  
class Student : IComparable { }  
class Student : ICloneable, IComparable { }  
  
struct Student : IComparable { }
```



```
class Student : Person, Object { }  
class Student : Person, LivingEntity { }  
  
struct Student : Person { }
```

- automaticky se volá konstruktor předka
 - ▶ kompilátor umí volat pouze bezparametrický konstruktor



```
class Person
{
    public Person() => Console.WriteLine("Person()");
    public Person(string name) => Console.WriteLine("Person(string)");
}

class Student : Person
{
    public Student() => Console.WriteLine("Student()");
    public Student(string netid) => Console.WriteLine("Student(string)")
}
```



```
Student student = new Student();
```

```
// Person()
```

```
// Student()
```

```
Student studentWithParam = new Student("netid");
```

```
// Person()
```

```
// Student(string netid)
```

- pokud kompilátor nemá k dispozici konstruktor k zavolání a není možné korektně inicializovat objekt, dojde k chybě kompilace

×

```
class Person
{
    //public Person() => Console.WriteLine("Person()");
    public Person(string name) => Console.WriteLine("Person(string)");
}

// nelze zkompileovat – kompilátor neumí vytvořit objekt
class Student : Person
{
    public Student() => Console.WriteLine("Student()");
    public Student(string netid) => Console.WriteLine("Student(string)")
}
```


- konstruktor předka lze zavolat pomocí : **base**([parametry])



```
class Person
{
    //public Person() => Console.WriteLine("Person()");
    public Person(string name) => Console.WriteLine("Person(string ↵
        name)");
}

class Student : Person
{
    public Student() : base("unknown")
    {
        Console.WriteLine("Student()");
    }

    public Student(string name, string netid) : base(name)
        => Console.WriteLine("Student(string, string)");
}
```

- lze zavolat i jiný konstruktor z aktuální třídy : **this**([parametry])



```
class Student : Person
{
    public Student() : this("unknown", "unidentified")
    {
    }

    public Student(string name, string netid) : base(name)
        => Console.WriteLine("Student(string, string)");
}
```

Polymorfizmus

Polymorfizmus

Základní vlastnosti

- potomek může zastoupit předka
 - ▶ všude kde je očekáván předek je možné dosadit potomka
- potomek může metody předka **zakrýt** nebo **přepsat**
 - ▶ v Javě je automaticky uplatňováno přepsání (tzv. pozdní vazba)
 - ▶ v C++ je časná/pozdní vazba rozlišena modifikátorem u metody
 - ▶ v C# je uplatněn systém podobný C++



```
class Person
{
    public void DoWork() => Console.WriteLine("Person goes to job!");
}

class Student : Person
{
    // warning CS0108 – úmyslné zakrývání má být označeno new
    public void DoWork() => Console.WriteLine("Student goes to school!");
}
```



```
Person person = new Person();
person.DoWork(); // -> person goes to job

Student student = new Student();
student.DoWork(); // -> student goes to school

person = student;
person.DoWork(); // -> person goes to job
```

Zakrývání

- zakrývání znamená vytvoření metody se stejným názvem (resp. signaturou) jako v předkovi
 - tato metoda, ale nemá polymorfní chování
 - metodu je nutné volat přímo nad instancí potomka, jinak se její kód nepoužije
 - kompilátor vyžaduje označování těchto metod klíčovým slovem **new**



```
class Person
{
    public void DoWork() => Console.WriteLine("Person goes to job!");
}

class Student : Person
{
    public new void DoWork() => Console.WriteLine("Student goes to ↵
        school!");
}
```

Polymorfizmus

- polymorfizmus umožňuje přetížit metodu v potomkovi
 - ▶ taková metoda je použita, pokud pracujeme s typem potomka i předka
 - ▶ uplatňuje se tzv. pozdní vazba
 - ▶ v předkovi je nutné metodu označit **virtual**

×

```
class Person
{
    public virtual void DoWork() => Console.WriteLine("Person");
}
```

```
class Student : Person
{
    // warning CS0114 – zakrývání virtual metody
    public void DoWork() => Console.WriteLine("Student");
}
```

```
// uvedený kód bude mít stejné chování jako předchozí dva slidy
// uplatněno je zakrývání / časná vazba!
```

Polymorfizmus

- polymorfizmus umožňuje přetížít metodu v potomkovi
 - ▶ v předkovi je nutné metodu označit **virtual**
 - ▶ v potomkovi je nutné metodu označit **override**



```
class Person
{
    public virtual void DoWork() => Console.WriteLine("Person");
}

class Student : Person
{
    public override void DoWork() => Console.WriteLine("Student");
}
```



```
Person person = new Student();
person.DoWork(); // -> Student
```


Rozpoznání typu objektu

- operátory **is**, **as**, (přetypování)
- metoda **object.GetType()** a operátor **typeof**
- pattern matching (**switch**)

Polymorfismus

- ▶ C# umožňuje hierarchii virtuálních metod přerušit a začít novou, je tak možné vytvořit metody
 - ▶ prarodič – **virtual**
 - ▶ rodič – **override**
 - ▶ potomek – **new virtual**
 - ▶ ...

Abstraktní metody a třídy

- abstraktní metoda (**abstract**) = virtuální metoda + nemá definici (tělo)
 - ▶ musí být definována v abstraktní třídě
- z abstraktní třídy nejde vytvořit objekt
 - ▶ je nutné v potomkovi doplnit definici metody a vytvářet objekt potomka



```
abstract class Person
{
    public abstract void DoWork();
}
```



```
Person person = new Person();
```

Rozhraní – interface

Rozhraní

- značí předpis, který musí třída splnit
 - ▶ třída může realizovat i více rozhraní
 - ▶ třída může rozhraní realizovat implicitně nebo explicitně
- neobsahuje datové složky (atributy)
- rozhraní nemůže definovat viditelnost složek
- může obsahovat metody, vlastnosti, indexery, události



```
interface IExample
{
    // Vlastnost
    int Property { get; set; }
    // Metoda
    void Method(int paramAlfa, int paramBeta);
    // Indexer
    int this[int param] { get; set; }
    // Událost
    event ExampleEventHandler Event;
}
```

Implicitní a explicitní realizace rozhraní

- implicitní realizace
 - ▶ stejné jako v Javě
- explicitní realizace
 - ▶ u definice složky v realizující třídě se uvede název rozhraní
 - ▶ taková složka může být vyvolána pouze nad typem rozhraní a nikoliv nad typem realizující třídy/struktury

Implicitní realizace rozhraní

```
✓  
  
interface IStringizable  
{  
    string Stringize();  
}  
  
class Student : IStringizable  
{  
    public string Name { get; set; }  
  
    public string Stringize()  
    {  
        return Name;  
    }  
}
```

Implicitní realizace rozhraní

- obdobné chování jako v Javě
- metody rozhraní lze volat přímo nad objektem realizující třídy nebo po přetypování



```
Student student = new Student();  
var a = student.Stringize();  
  
IStringizable istringizable = student;  
var b = istringizable.Stringize();
```

Explicitní realizace rozhraní



```
interface IStringizable
```

```
{
```

```
    string Stringize();
```

```
}
```

```
class Student : IStringizable
```

```
{
```

```
    public string Name { get; set; }
```

```
    string IStringizable.Stringize() // ← uvedení názvu rozhraní před ↵  
        název metody
```

```
{
```

```
    return Name;
```

```
}
```

```
}
```


Explicitní realizace rozhraní

- nemá obdobu v Javě nebo C++
- metody rozhraní lze volat pouze nad typem rozhraní
- tento způsob realizace lze využít k rozlišení rozhraní, které mají složky se stejným identifikátorem (signaturou)



```
Student student = new Student();  
// error CS1061  
// var a = student.Stringize();  
  
IStringizable istringizable = student;  
var b = istringizable.Stringize();
```



```
interface IDrawable
{
    void Draw();
}

interface ISurface
{
    void Draw();
}

class Rectangle : IDrawable, ISurface
{
    public void Draw()
    {
        // IDrawable i ISurface
    }
}
```

- implicitní realizace
- obě rozhraní volají stejnou metodu Draw()



```
interface IDrawable { void Draw(); }  
interface ISurface { void Draw(); }  
  
class Rectangle : IDrawable, ISurface  
{  
    public void Draw()  
    {  
        // ISurface  
    }  
  
    void IDrawable.Draw()  
    {  
        // IDrawable  
    }  
}
```

- kombinace implicitní a explicitní realizace



```
interface IDrawable { void Draw(); }  
interface ISurface { void Draw(); }  
  
class Rectangle : IDrawable, ISurface  
{  
    void ISurface.Draw()  
    {  
        // ISurface  
    }  
  
    void IDrawable.Draw()  
    {  
        // IDrawable  
    }  
}
```

- obě rozhraní používají explicitní realizaci

Základní rozhraní v knihovně

- `ICloneable` – `object Clone()`
 - ▶ umožňuje provádět mělkou (`MemberwiseClone()`) nebo hlubokou kopii objektu
- `IComparable` – `int CompareTo(object obj)`
 - ▶ porovnání menší/větší/rovno mezi dvěma objekty
- `IComparer` – `int Compare(object x, object y)`
 - ▶ podobně jako `IComparable`, ale objekt funguje jako nezávislý porovnávač jiných objektů
- `IDisposable` – řízené uvolnění prostředků
- `IFormattable` – převod na řetězec v daném formátu

IEnumerable a foreach

- slouží k projití prvků kolekce
- lze využít cyklus **foreach**
- metoda `IEnumerator GetEnumerator()`
 - ▶ `IEnumerator`
 - ★ `Object Current`
 - ★ **`bool`** `MoveNext()`
 - ★ **`void`** `Reset()`



```
var enumerable = new TestEnumerable();  
foreach (var obj in enumerable)  
{  
    Console.WriteLine($"{obj}");  
}
```

IEnumerator

- enumerátor lze napsat ručně nebo lze využít automatické implementace od kompilátoru a klíčového slova **yield**



```
class TestEnumerable : IEnumerable
{
    readonly int[] values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < values.Length; i++)
        {
            if (values[i] > 5)
                yield break;

            yield return values[i];
        }
    }
}
```


IEnumerator

- **yield return** ... – přidá prvek do výstupu enumerátoru
- **yield break** – ukončí iterování



```
class TestEnumerable : IEnumerable
{
    ...

    public IEnumerable DescendingValues
    {
        get
        {
            for (int i = values.Length - 1; i >= 0; i--)
                yield return values[i];
        }
    }
}
```