Jazyk C# a .NET

prednáška

4

ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta riadenia
a informatiky

# Základy jazyka C# (3)

Ing. **Štefan Toth**, PhD.

13.03.2025

KST Katedra softvérových technológií
Fakulta riadenia a informatiky, Žilinská univerzita v Žiline

# Join at slido.com
# #3237110

# Obsah

- **Štruktúry** (structs)
- **Rekordy** (records)
- **Anonymné typy** (anonymous types)
- **Vymenované typy** (enumeration / enum types)
- **N-tice** (tuple types)
- **Rozhrania** (interfaces)
- **Generiká** (generics)

# Štruktúry (structures / struct types) (1)

- Štruktúry sú **hodnotové typy vytvárané na zásobníku** (stack)
- Ak sú ale vložené do parametra objektu, potom nastáva **boxing** – sú **kopírované do haldy** (heap)
  - To neplatí ale pre štruktúry deklarované s modifikátorom **ref** (ref struct) – štruktúra bude mať niekoľko obmedzení, ale boxing nenastane
- **Nepodporujú dedičnosť**, vždy majú **bezparametrický konštruktor**
- Ak členy štruktúry nemenia svoj stav (sú „immutable“), potom môže byť štruktúra deklarovaná aj s modifikátorom **readonly**
  - Kompilátor zaručí, že nebude možné pridať akýchkoľvek členov, ktorí by modifikovali jej stav

# Štruktúry (2)

- Dobrá prax je vytvárať štruktúry nemenné – „immutable“:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() =>
        $"({X}, {Y})";
}
```

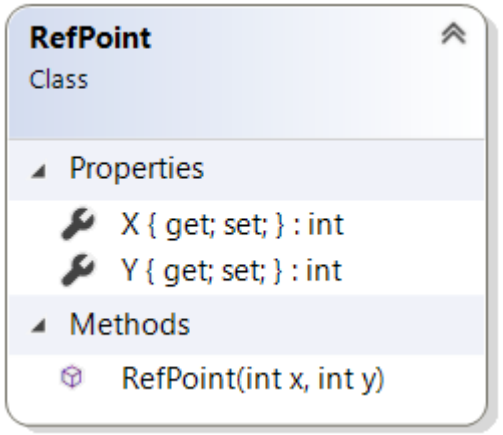
```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() =>
        $"({X}, {Y})";
}
```

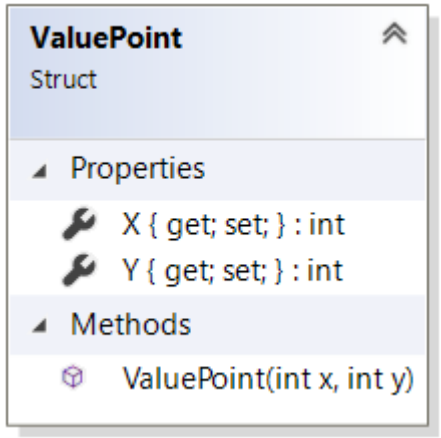
# Triedy verzus štruktúry – porovnanie

- **Trieda** (class) – „zložitejší“ dátový typ
  - **Referenčný typ**, objekty sa alokujú v spravovanej halde (heap), pri priradení do inej premennej alebo použití v metódach sa **kopíruje iba referencia** na inštanciu
  - **Jednoduchá dedičnosť** – môže priamo dediť od jednej jedinej triedy
  - Implementovať môže viacero rozhraní (interface)
  - Implicitná (default) hodnota: **null**
  - Používa sa pre zložitejšie objekty a pre objekty, ktoré vyžadujú dedičnosť a polymorfizmus

- **Štruktúra** (struct) – je ako „odľahčená“ trieda
  - **Hodnotový typ**, objekty sa alokujú v zásobníku (stack), pri priradení do inej premennej alebo použití v metódach sa **kopíruje celá štruktúra** so všetkými hodnotami
  - **Nepodporuje dedičnosť** – nemôže dediť od žiadnej štruktúry, ani triedy
  - Implementovať môže viacero rozhraní (interface)
  - Implicitná (default) hodnota: závisí od typu, **nemôže byť null**
  - Je rýchlejšia, používa sa na malé objekty

# Trieda vs. štruktúra




```
var point1 = new RefPoint(1, 1);
var point2 = point1;

point1.X *= 10;

// Čo sa vypíše na obrazovku?
Console.WriteLine(point1.X);
Console.WriteLine(point2.X);

class RefPoint
{
    public int X { get; set; }

    public int Y { get; set; }

    public RefPoint(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

```
10
10
```

```
var point1 = new ValuePoint(1, 1);
var point2 = point1;

point1.X *= 10;

// Čo sa vypíše na obrazovku?
Console.WriteLine(point1.X);
Console.WriteLine(point2.X);

struct ValuePoint
{
    public int X { get; set; }

    public int Y { get; set; }

    public ValuePoint(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

```
10
1
```

# Audience Q&A

# Rekordy (1)

- Od C# 9 sa definovalo kľúčové slovo **record** pre referenčné typy, ktoré **poskytujú vstavané funkcionality** pre zapuzdrené dáta
- Od C# 10 je možné vytvoriť rekordy aj pre hodnotové typy, preto sa zaviedla nová syntax:
  - **record class** (pre referenčné typy, to isté ako pôvodný record)
  - **record struct** (pre hodnotové typy)
- Ich použitie zmenšuje veľkosť kódu, pretože **kompilátor automaticky generuje** vlastnosti, implementuje porovnávanie objektov IEquatable<T>, Equals(), GetHashCode(), ToString(), Clone(), operátory, kopírovacie konštruktory, dekonštruktor, ...

# Rekordy (2)

- Príklad definovania a použitia rekordu referenčného typu

```
// Vytvorený rekord pozičnou syntaxou, ktorá definuje 2 vlastnosti FirstName a LastName:
public record Person(string FirstName, string LastName);
```

```
// Alebo od C# 10 sa dá to isté definovať aj ako record class:
public record class Person(string FirstName, string LastName);
```

```
// Alebo rovnaký rekord, avšak vytvorený nominálnou syntaxou ako klasická trieda:
public record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }

    public Person(string firstName, string lastName) =>
        (FirstName, LastName) = (firstName, lastName);
}
```

```
Person person = new("Ján", "Mrkvička");
Console.WriteLine(person);
```

```
Person { FirstName = Ján, LastName = Mrkvička }
```

# Rekordy (3)

- Ďalšie príklady definovania rekordov

```
public record Person(string FirstName, string LastName)
{
    // Môžeme pridať akékoľvek členy – vlastnosti, metódy, konštruktory, ...
}
```

```
public record Person // „Immutable“ record, vyžaduje pri inicializácii nastaviť vlastnosti
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
}
```

```
public record Person // „Mutable“ record, vyžaduje pri inicializácii nastaviť vlastnosti
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}
```

# Rekordy (4)

- Kompilátor generuje veľa užitočného kódu
  - https://sharplab.io/#v2:CYLg1APgAgTAjAWAFDKgZgAQCcCmBjAey2AwAUcsBnAgOwAoo4AGDAMQEsqAXAOQEMAtjgA0GRiwAyfSr0E4AlAG5kQA

```
public record Person(string FirstName, string LastName);
```

```
public class Person : IEquatable<Person>
{
    private readonly string <FirstName>k__BackingField;
    private readonly string <LastName>k__BackingField;

    public string FirstName
    {
        get => <FirstName>k__BackingField;
        init => <FirstName>k__BackingField = value;
    }

    // ...
}
```

# Rekordy (5)

- **Vytvorenie nových objektov z existujúcich** pomocou **with**
  - V inicializátori objektov definujete vlastnosti, ktoré sa majú zmeniť od pôvodných
  - Používa metódu Clone()

```
public record Person(string FirstName, string LastName);
```

```
Person person = new("Jana", "Mrkvičková");
Person person2 = person with { LastName = "Vydatá" };
Console.WriteLine(person2);
```

```
Person { FirstName = Jana, LastName = Vydatá }
```

# Rekordy (6)

- Porovnávanie objektov

```
public record Person(string FirstName, string LastName);
```

```
Person person1 = new("Jana", "Mrkvičková");
Person person2 = new("Jana", "Mrkvičková");

// Vracia true, pretože operátor == je implementovaný v rekorde
if (person1 == person2)
    Console.WriteLine("Obidva rekordy majú rovnaké hodnoty.");

// ReferenceEquals vracia false, pretože sú to rôzne objekty
if (!ReferenceEquals(person1, person2))
    Console.WriteLine("Rekordy majú iné referencie.");
```

```
Obidva rekordy majú rovnaké hodnoty.
Rekordy majú iné referencie.
```

# Rekordy (7) – štruktúry

```
// Meniteľná (mutable) štruktúra
public record struct Point(double X, double Y);
```

```
// Meniteľná (mutable) štruktúra
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
}
```

```
// Nemeniteľná (immutable) štruktúra
public readonly record struct Point(double X, double Y);
```

```
// Nemeniteľné (immutable) štruktúra
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
}
```

# Audience Q&A

slido

# Aký je rozdieľ medzi štruktúrou a triedou?

ⓘ Start presenting to display the poll results on this slide.

# Anonymné typy (anonymous types)

- **Anonymný typ** je **trieda bez mena**, ktorá dedí od triedy object
- Definované implicitne pomocou kľúčového slova **var**
  - Nie je možné zapísať typ explicitne, lebo názov triedy neexistuje
- Kompilátor vytvorí **triedu s vlastnosťami len na čítanie**

```
var captain = new
{
    FirstName = "James",
    MiddleName = "Tiberius",
    LastName = "Kirk"
};
```

```
var doctor = new
{
    FirstName = "Leonard",
    MiddleName = string.Empty,
    LastName = "McCoy"
};
```

```
{ FirstName = Leonard, MiddleName = , LastName = McCoy }
McCoy Leonard
```

```
Console.WriteLine(doctor);
//doctor.LastName = "Nimoy"; // Error CS0200
Console.WriteLine($"{doctor.LastName} {doctor.FirstName}");
```

# Vymenované typy (enumeration / enum types) (1)

- Enum je **hodnotový typ**, ktorý definuje **množinu pomenovaných konštánt** číselného typu

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

```
enum Color : short
{
    Red = 1,
    Green = 2,
    Blue = 3
}
```

```
Color c = Color.Red;
Console.WriteLine(c);
Console.WriteLine((short)c);
Console.WriteLine((Color)2);
```

```
Red
1
Green
```

# Vymenované typy (enum types) (2)

- Ak chceme použiť **viacero hodnôt naraz**, musíme označiť enum atribútom **[Flags]** a jednotlivé hodnoty konštánt musia mať rôzne bity, aby bolo možné pracovať s bitovými operátormi & alebo |

```
[Flags] // Toto je atribút [Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}
```

```
Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
Console.WriteLine(meetingDays);

Days workingFromHomeDays = Days.Thursday | Days.Friday;
Console.WriteLine($"{meetingDays & workingFromHomeDays}");

bool isMeetingOnTuesday =
  (meetingDays & Days.Tuesday) == Days.Tuesday;
Console.WriteLine($"Meeting on Tuesday? {isMeetingOnTuesday}");

var a = (Days)37;
Console.WriteLine(a);
```

```
Monday, Wednesday, Friday
Friday
Meeting on Tuesday? False
Monday, Wednesday, Saturday
```

# N-tice (tuples) (1)

- **N-tica (tuple)** zoskupuje **viaceré hodnoty** rôznych typov do jednej premennej bez nutnosti vytvárania štruktúry alebo triedy

```
var tuple1 = ("reťazec", 123, new Book("Professional C#", "Wrox Press")); // Bez pomenovania
(string AString, int Number, Book Book) tuple2 = ("reťazec", 123, new Book("Professional C#", "Wrox Press"));
var tuple3 = (AString: "reťazec", Number: 123, Book: new Book("Professional C#", "Wrox Press"));
Console.WriteLine($"{tuple1.Item1} == {tuple2.AString} == {tuple3.AString}");
```

```
reťazec == reťazec == reťazec
```

- Používa sa na návrat **viacerých hodnôt z metódy** alebo na prenos viacerých hodnôt v jednom parametri metódy

```
public static (int Left, int Top) GetCursorPosition()
{
    int left = Console.CursorLeft;
    int top = Console.CursorTop;

    return (left, top);
}

var position = GetCursorPosition();
Console.WriteLine($"X: {position.Left}, Y: {position.Top}");
```

```
(int result, int remainder) Divide(int dividend,
    int divisor)
{
    int result = dividend / divisor;
    int remainder = dividend % divisor;
    return (result, remainder);
}
```

# N-tice (tuples) (2) – dekonštrukcia

- **Dekonštrukcia** – rozloženie prvkov do samostatných premenných:

```
var tuple = (AString: "reťazec", Number: 123, Book: new Book("Professional C#", "Wrox Press"));

(string aString, int number, Book book) = tuple;
// Alebo to isté aj jednoduchšie:
//var (aString, number, book) = tuple
Console.WriteLine($"Reťazec: {aString}, číslo: {number}, kniha: {book}");
```

```
Reťazec: reťazec, číslo 123, kniha: Professional C# (Wrox Press)
```

- **Podčiarkovník** _ („discard“ premenná) môžeme použiť na ignorovanie (zahodenie) niektorých prvkov, ktoré nepotrebujeme:

```
(_, _, var book1) = tuple;
Console.WriteLine(book1.Title);
```

```
Professional C#
```

# N-tice (tuples) (3) – dekonštruktory (deconstructors)

- **Dekonštrukcia s vlastnými typmi**
  - Nutné vytvoriť **dekonštruktor** – metódu v tvare **void Deconstruct(out parametre)**
  - Je možné vytvoriť aj **viacero metód Deconstruct()**, ale musia sa **odlišovať iba v počte parametrov** („arity“) – nezáleží na typoch ako je to pri preťažovaní metód

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    //...

    public void Deconstruct(out string firstName,
        out string lastName, out int age)
    {
        firstName = FirstName;
        lastName = LastName;
        age = Age;
    }
}
```

```
var p = new Person("Ján", "Mrkvička", 18);
var (fName, lName, age) = p;
Console.WriteLine($"{fName} {lName}, {age}!");

var (_, lName2, _) = p;
Console.WriteLine($"{lName2}");
```

```
Ján Mrkvička, 18!
Mrkvička
```

# N-tice (tuples) (4) – „pattern matching“

- „Pattern matching“ umožňuje porovnávať hodnoty s rôznymi vzormi a vykonávať rôzne akcie na základe toho, ktorý vzor sa zhoduje s hodnotou

```
static void PrintPersonInfo(
    (string FirstName, string LastName, int Age) person)
{
    switch (person)
    {
        case var (firstName, lastName, _) when
                 firstName == "Ján" && lastName == "Mrkvička":
            Console.WriteLine("Ty si Jano Mrkvička!");
            break;
        case var (_, _, age) when age >= 18:
            Console.WriteLine("Ty si dospelý!");
            break;
        case var (_, _, age) when age < 18:
            Console.WriteLine("Ešte nemáš 18!");
            break;
        default:
            Console.WriteLine("Ty si niekto iný!");
            break;
    }
}
```

```
string Foo(int x, int y)
{
    return (x, y) switch
    {
        (> 32, not 3) => "foo",
        (> 40, not > 4) => "bar",
    };
}
```

```
string RockPaperScissors(string first, string second) =>
    (first, second) switch
    {
        ("kameň", "papier") => "Papier vyhral",
        ("kameň", "nožnice") => "Kameň vyhral",
        ("papier", "kameň") => "Papier vyhral",
        ("papier", "nožnice") => "Nožnice vyhrali",
        ("nožnice", "kameň") => "Kameň vyhral",
        ("nožnice", "papier") => "Nožnice vyhrali",
        (_, _) => "Remíza"
    };

Console.WriteLine(RockPaperScissors("kameň", "papier"));
```

# Audience Q&A

slido

# Aká je platná hlavička metódy, ktorá príjma a aj vracia n-tice (tuples):

ⓘ Start presenting to display the poll results on this slide.

slido

# Akou špeciálnou premennou môžeme ignorovať (discard) hodnoty pri dekonštrukcii?

ⓘ Start presenting to display the poll results on this slide.

slido

**Akú metódu musíme vytvoriť, aby sme mohli vytvoriť n-ticu (tuple) z nejakej vlastnej triedy?**

ⓘ Start presenting to display the poll results on this slide.

# Porovnanie anonymných typov

```
var doctor1 = new
{
    FirstName = "Leonard",
    MiddleName = string.Empty,
    LastName = "McCoy"
};


var doctor2 = new
{
    FirstName = "Leonard",
    MiddleName = string.Empty,
    LastName = "McCoy"
};

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
False
False
True
```

```
var doctor1 = new
{
    FirstName = "Leonard",
    MiddleName = string.Empty,
    LastName = "McCoy"
};


var doctor2 = doctor;

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
True
True
```

# Porovnanie n-tíc

```
var doctor1 = (FirstName: "Leonard", MiddleName: string.Empty, LastName: "McCoy");
var doctor2 = (FirstName: "Leonard", MiddleName: string.Empty, LastName: "McCoy");

Console.WriteLine(doctor1 == doctor2);
// Warning CA2013: Do not pass an argument with value
// type... to ReferenceEquals. Due to value boxing,
// this call to ReferenceEquals will always return
// false
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
False
True
```

```
var doctor1 = (FirstName: "Leonard", MiddleName: string.Empty, LastName: "McCoy");
var doctor2 = doctor1;

Console.WriteLine(doctor1 == doctor2);
// Warning CA2013: Do not pass an argument with value
// type... to ReferenceEquals. Due to value boxing,
// this call to ReferenceEquals will always return
// false
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
False
True
```

# Porovnanie class rekordov

```
record class Person(string FirstName, string MiddleName, string LastName);
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = new Person("Leonard", string.Empty, "McCoy");

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
False
True
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = doctor1;

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
True
True
```

# Porovnanie struct rekordov

```
record struct Person(string FirstName, string MiddleName, string LastName);
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = new Person("Leonard", string.Empty, "McCoy");

Console.WriteLine(doctor1 == doctor2);
// Warning CA2013: Do not pass an argument with value
// type... to ReferenceEquals. Due to value boxing,
// this call to ReferenceEquals will always return
// false
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
False
True
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = doctor1;

Console.WriteLine(doctor1 == doctor2);
// Warning CA2013: Do not pass an argument with value
// type... to ReferenceEquals. Due to value boxing,
// this call to ReferenceEquals will always return
// false
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
False
True
```

# Porovnanie class

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

    public Person(string firstName, string middleName, string lastName) =>
        (FirstName, MiddleName, LastName) = (firstName, middleName, lastName);
}
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = new Person("Leonard", string.Empty, "McCoy");

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
False
False
False
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = doctor1;

Console.WriteLine(doctor1 == doctor2);
Console.WriteLine(ReferenceEquals(doctor1, doctor2));
Console.WriteLine(doctor1.Equals(doctor2));
```

```
True
True
True
```

# Porovnanie struct

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

    public Person(string firstName, string middleName, string lastName) =>
        (FirstName, MiddleName, LastName) = (firstName, middleName, lastName);
}
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = new Person("Leonard", string.Empty, "McCoy");

//Console.WriteLine(doctor1 == doctor2); // Error CS0019
Console.WriteLine(ReferenceEquals(doctor1, doctor2)); // Warning CA2013 boxing
Console.WriteLine(doctor1.Equals(doctor2));
```

```
False
True
```

```
var doctor1 = new Person("Leonard", string.Empty, "McCoy");
var doctor2 = doctor1;

//Console.WriteLine(doctor1 == doctor2); // Error CS0019
Console.WriteLine(ReferenceEquals(doctor1, doctor2)); // Warning CA2013 boxing
Console.WriteLine(doctor1.Equals(doctor2));
```

```
False
True
```

# Audience Q&A

# Rozhrania (interfaces) (1)

- **Rozhranie** (interface) definuje kontrakt, ktorý môže byť implementovaný triedami alebo štruktúrami
- Môže obsahovať **metódy**, **vlastnosti**, **udalosti** a **indexery**
- Rozhranie **zvyčajne neposkytuje implementácie** členov, ktoré definuje
- Rozhrania môžu **využívať viacnásobnú dedičnosť** (multiple inheritance)
  - Používa sa znak **:** (dvojbodky)

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox
{

}
```

# Rozhrania (2)

- Ak chceme implementovať v nejakej triede alebo štruktúre rozhranie, použijeme **dvojbodku :**
- Triedy a štruktúry môžu implementovať aj viacero rozhraní oddelených čiarkou:

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

# Rozhrania (3) – predvolený (default) členovia

- Metódy alebo vlastnosti môžu mať **predvolené** (default) **implementácie**:

```
public interface ILogger
{
    void Log(string message);
    public void Log(Exception ex) => Log(ex.Message);
}
```

# Rozhrania (4) – explicitná a implicitná implementácia

- Rozhranie môže byť **implementované implicitne** alebo **explicitne**
  - Explicitná implementácia sa používa na vyriešenie problémov s preťažovaním metód (rôzne signatúry) alebo „skrytie“ metódy inštancie

```
public interface ILogger
{
    void Log(string message);
}
```

```
public class ConsoleLogger : ILogger
{
    // Implicitná implementácia – vytvorená inštančná Metóda s prístupovým modifikátorom public
    public void Log(string message) => Console.WriteLine(message);
}
```

```
public class ConsoleLogger : ILogger
{
    // Explicitná implementácia v tvare NázovRozhrania.Metóda a bez prístupového modifikátora public
    void ILogger.Log(string message) => Console.WriteLine(message);
}
```

# Rozhrania (5) – explicitná a implicitná implementácia

- Príklad pre implementácie rovnako nazvaných metód a vlastností s inými signatúrami:

```
public class People : IEnumerable<Person>
{
    // ...

    // Explicitná implementácia metódy GetEnumerator
    // z rozhrania IEnumerable
    IEnumerator IEnumerable.GetEnumerator()
    {
       return (IEnumerator)GetEnumerator();
    }

    // Implicitná implementácia metódy GetEnumerator
    // z generického rozhrania IEnumerable<T>
    public IEnumerable<Person> GetEnumerator()
    {
        return new PeopleEnum(_people);
    }
}
```

```
public class PeopleEnum : IEnumerator<Person>
{
    // ...

    // Explicitná implementácia vlastnosti Current
    object IEnumerator.Current => Current;

    // Implicitná implementácia vlastnosti Current
    public Person Current
    {
        get
        {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

# Rozhrania (6) – vybrané rozhrania v .NET

- **IComparable**, **IComparable<T>** – umožňujú porovnávať objekty pomocou metódy CompareTo()
- **IEnumerable**, **IEnumerable<T>** – umožňujú iterovať kolekcie objektov cez metódu GetEnumerator(), ktorá je nutná pre foreach cyklus
- **IEquatable, IEquatable<T>** – umožňujú implementovať vlastné porovnávacie metódy
- **IDisposable** – umožňuje implementovať metódu Dispose() na uvoľnenie prostriedkov (súbory, spojenia so sieťou, ...)
- **IClonable** – umožňuje implementovať metódu Clone() na kopírovanie objektu
- **IFormattable** – umožňuje implementovať metódu ToString() pre vypísanie objektu do reťazcov v rôznych formátoch

# Audience Q&A

# Generiká (1)

- Generické typy definujú **typové parametre** (type parameters), ktoré predstavujú **zástupné symboly** pre špecifický typ, ktorý sa zadáva pri vytváraní inštancie generického typu
  - Typové parametre predstavujú **zoznam parametrov** názvov **typov** v lomených zátvorkách **< >**, pričom popisný názov by mal **začínať na** písmeno **T** (napr. TKey, TValue, TInput, TOutput, ...), prípadne sa používa iba jedno písmeno (T, U, V), ak je úplne jasné, čo parameter predstavuje

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
public struct Nullable<T> where T : struct { /*...*/ }
```

```
public class Pair<TFirst, TSecond>
{
    public TFirst FirstProperty { get; }
    public TSecond SecondProperty { get; }
    public Pair(TFirst first, TSecond second) =>
        (FirstProperty, SecondProperty) = (first, second);
}
```

# Generiká (2)

- Typ, ktorý je deklarovaný s typovými parametrami, sa nazýva **generický typ** (generic type)
- Keď sa ide generický typ použiť, argumenty s typmi musia byť zadané (alebo odvodené) pre každý z typových parametrov:

```
var pair = new Pair<int, string>(1, "two");
int i = pair.FirstProperty;
string s = pair.SecondProperty;
```

- Generický typ s argumentmi typov, ako vyššie Pair<int, string>, sa označuje ako **skonštruovaný typ** (constructed type)

# Generiká (3) – obmedzenia (constraints)

- **Obmedzenia** informujú kompilátor o tom, čo musí typ spĺňať
  - Definujú sa za kľúčovým slovom **where**

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    //...
}
```

```
public class SampleClass<T, U, V> where T : V { }
```

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T
    { /*...*/ }
}
```

```
class Base { /* ... */ }

class Test<T, U>
    where U : struct
    where T : Base, new()
{
    //...
}
```

  - Ak obmedzenie nie je splnené, kompilátor neskompiluje zdrojový kód

```
var teachers = new EmployeeList<Teacher>(); // Teacher musí spĺňať všetky obmedzenia
```

# Generiká (4) – obmedzenia

| Obmedzenie (constraint) | Popis |
|---|---|
| where T : **struct** | T musí byť hodnotový typ (je vždy „non-nullable“ = nemôže byť null) |
| where T : **class** | T musí byť referenčný typ (class, interface, delegate, typ poľa, vrátane rekordov) „non-nullable“ |
| where T : **class?** | T musí byť referenčný typ buď „nullable“ alebo „non-nullable“ |
| where T : **notnull** | T musí byť „non-nullable“ typ (argument môže byť referenčný alebo hodnotový „non-nullable“ typ) |
| where T : **unmanaged** | T musí byť nemanažovaný typ (štruktúry a n-tice s nemanažovanými prvkami, enum, ukazovatele) |
| where T : **IMyInterface** | T musí implementovať rozhranie IMyInterface, T musí byť „non-nullable“ |
| where T : **IMyInterface?** | T musí implementovať rozhranie IMyInterface, T môže byť „nullable “ alebo „non-nullable“ referenčný typ, alebo hodnotový typ |
| where T : **MyBaseClass** | T musí byť potomkom triedy MyBaseClass, T musí byť „non-nullable“ |
| where T : **MyBaseClass?** | T musí byť potomkom triedy MyBaseClass, T môže byť „nullable “ alebo „non-nullable“ referenčný typ, alebo hodnotový typ |
| where T : **new()** | T musí mať verejný (public) bezparametrický konštruktor. Ak má typ viac obmedzení, toto sa uvádza ako posledné. Nemôže sa kombinovať s obmedzením *struct* a *unmanaged* |
| where T : **U** | T musí dediť od iného definovaného generického typu U |
| where T : **default** | Rieši nejednoznačnosť v prípade prekrývania metód alebo explicitnej implementácie rozhrania |
| where T : **allows ref struct** | „Antiobmedzenie“ - typ T môže byť typ ref struct |

Zdroj: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

# Nemanažované typy (unmanaged types)

- Typ je **unmanaged** (nemanažovaný / nespravovaný), ak je jedným z nasledujúcich typov:
  - sbyte, byte, short, ushort, int, uint, long, ulong, nint, nuint, char, float, double, decimal alebo bool
  - Akýkoľvek vymenovaný typ (enum)
  - Akýkoľvek typ ukazovateľa (pointer)
  - N-tica (tuple), ktorej všetky členy sú nemanažované typy
  - Akákoľvek používateľom definovaná štruktúra (struct), ktorá obsahuje iba nemanažované typy

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

# Ukazovatele / smerníky (pointer types)

- Deklarácia ukazovateľa: `typ* identifikátor;`
- Nutné použiť kľúčové slovo **unsafe** a povoliť kompilátoru preklad pomocou voľby **AllowUnsafeBlocks**

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--");
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
10
20
30
--
10
11
12
```

```
int* p1, p2, p3;    // Ok
int *p1, *p2, *p3;  // Neplatné v C#
```

Zdroj: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code#pointer-types

slido

**Ako by ste deklarovali obmedzenie, že typ musí byť "non-nullable" referenčný typ, ktorý implementuje rozhranie IClonable a má bezparametrický konštruktor?**

ⓘ Start presenting to display the poll results on this slide.

# Audience Q&A

# Použitá literatúra

- Christian Nagel: **Professional C# and .NET** 2021 Edition, 8. vydanie, ISBN-13: 978-1119797203

- .NET a C#:
  - https://learn.microsoft.com/en-us/dotnet/
  - https://learn.microsoft.com/en-us/dotnet/csharp/

ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta riadenia
a informatiky

Katedra softvérových
technológií

stefan.toth@uniza.sk

# ĎAKUJEM ZA POZORNOSŤ

# Upozornenie

- Tieto študijné materiály sú určené výhradne pre študentov predmetu **Jazyk C# a .NET** na Fakulte riadenia a informatiky Žilinskej univerzity v Žiline

- Reprodukovanie, šírenie (i častí) materiálov bez písomného súhlasu autora nie je dovolené

ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta riadenia
a informatiky

Ing. **Štefan Toth**, PhD.

stefan.toth@uniza.sk