Jazyk C# 2

11. seminář

Večeřa J., Janoštík R.

Univerzita Palackého v Olomouci

9. 5. 2022

Úvod

- C# (a celá .NET platforma) se neustále vyvíjí
- Některé změny jsou velmi zajímavé (např. zavedení LINQ)
- Je dobré vědět, kam C# směřuje (drží se trendu)
- Někdy i drobná změna „pod kapotou“ velmi zefektivní vaše programy

.NET 7 - STS

- Podpora STS vs. LTS - 18 vs. 36 měsíců
- Co je „podpora“?
  - Bug fixy a updaty
  - Dokumentace (!)
  - Technická podpora při problémech
  - Návody, vzdělávání, komunita
- Velké projekty
  - V poslední LTS (výjimky - legacy kód, kompatibilita runtime prostředí, ...)
- Malé projekty - je to jedno
  - Jednoduchý upgrade a změny
  - Čím novější, tím lepší

.NET 8

- Listopad 2023 - nová LTS verze
- Vyšlo Preview 1 a Preview 2
  - `https://devblogs.microsoft.com/dotnet/announcing-dotnet-8-preview-1/`
  - `https://devblogs.microsoft.com/dotnet/announcing-dotnet-8-preview-2/`
- Tradičně spolu s ním vyjde i nová verze C#, tedy C# 12
- Preview zatím nezajímavé, koukneme se na C# 12
  - `https://devblogs.microsoft.com/dotnet/check-out-csharp-12-preview/`

Co přinesl .NET 8?

- Nativní AOT
  - Kompilace Ahead-Of-Time do nativního kódu
  - Nevyžaduje .NET runtime na cílovém stroji
  - Rychlejší spuštění programu a menší spotřeba paměti
  - Použití na cílové systémy (win x64, linux x64, . . .)
- Kontejnery
  - Běh obrazu (image) v kontejneru
  - Podpora pro spuštění pro non-root uživatele
  - Užitečné na sdílených strojích (firmy, cloud, ...) a v testovacím prostředí pro developery

Co přinesl .NET 8?

- Random
  - Metody pro náhodný výběr prvků a míchání pořadí přímo v knihovně (cryptography)
- JSON
  - Podpora nových syntaktických konvencí (snake_case, kebab-case)
  - Další funkce pro lepší deserializaci a serializaci
- Frozen Dictionary
  - Read-only dictionary, rychlejší čtení a hledání
- Hardware akcelerace
  - Pro hardwarový design, který podporuje novou implementaci - rychlejší
  - Vector128, Vector256
  - Matrix3x2, Matrix4x4

.NET 9

- Listopad 2024 - nová STS verze
- Aktuálně Preview 1
  - `https://devblogs.microsoft.com/dotnet/our-vision-for-dotnet-9/`
- nová verze C#, tedy C# 13
  - `https://learn.microsoft.com/cs-cz/dotnet/csharp/whats-new/csharp-13/`

Co jsme v kurzu nezmínili: N-tice (tuples)

- Implicitní třída pro (uspořádané n-tice) nebyla
- „Odlehčená struktura obsahující více elementů“

```
1 (int, string, double) mojeNTice = (5, "ahoj", 3.1415);
2 Console.WriteLine(mojeNTice.Item3);
```

- Pojmenování položek:

```
1 (int cislo, string retezec, double toTreti) mojeNTice = (5,
      "ahoj", 3.1415);
2 Console.WriteLine(mojeNTice.retezec);
```

  - Pojmenování dostupné pouze v čase kompilace
  - Nedostupné přes reflexi

Co jsme v kurzu nezmínili: Dekonstrukce n-tic

- N-tici můžeme „rozbalit“ do lokálních proměnných

```
1 public static (int min, int max, double mean) getStats(int[]
      array) {...}
2
3 int[] cisla = { 1, 23, 4, 5, 6, 7 };
4 (int min, int max, double mean) = getStats(cisla);
5 Console.WriteLine($"Prumer je: {mean}");
```

- Dekonstrukce objektů - předpis, jak se z objektu má vytvořit n-tice

```
1 public class Person {
2   public int Id { get; set; }
3   public string Name { get; set; }
4   public DateTime Birthday { get; set; }
5   public double Salary { get; set; }
6 }
```

Dokonstrukce n-tic

- Dodefinujeme, jak se má objekt „rozbalit“

```
1 public void Deconstruct(out int id, out string name) =>
2   (id, name) = (Id, Name);
3 public void Deconstruct(out int id, out string name, out double
      salary) =>
4   (id, name, salary) = (Id, Name, Salary);
```

- Můžeme objekt přiřadit do dané n-tice

```
1 Person p = new Person() { Id = 1, Name = "Ja",
2         Birthday = new DateTime(2000, 2, 29), Salary = 51000.0 };
3 (int id, string name) = p;
4 (int id2, string name2, double s) = p;
```

- Více na `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples`

Co jsme v kurzu nezmínili: Pattern matching

- Rozšíření testu `if` a `switch` na strukturu
- Větev se může použít, pokud má daná struktura daný tvar
- Součástí může být i přiřazení nějaké informace

```
1 public static void ProcessObject(Object o) {
2    if (o is Person p) {
3       Console.WriteLine($"Objekt je osoba se jmenem {p.Name}");
4    } else if (o is int i) {
5      Console.WriteLine($"O je jen cislo!");
6    }
7 }
```

Pattern matching - switch

- Dříve „do switche“ mohly jít jen čísla, enumy a řetězce
- Rozšíření na všechny objekty + matching na strukturu + přiřazení

```
1  public static void ProcessWithSwitch(Object o) {
2     switch (o) {
3        case List<string> strings when strings.Count > 10:
4           Console.WriteLine($"Moc retezcu");
5           break;
6        case int i when i > 5:
7           Console.WriteLine($"Velke cislo {i}");
8           break;
9        case null:
10          Console.WriteLine($"NULLL");
11          break;
12       default:
13          Console.WriteLine("obycejny objekt");
14          break;
15    }
16 }
```

Co jsme v kurzu nezmínili: Records – záznamy

- Referenční datový typ pro uchování „čistých dat“
  - data class z Kotlinu
  - Java record
- Primárně navržen jako nemutovatelný typ, vytvářený „pozičně“, porovnání „po složkách“

```
1 public record Person(string FirstName, string LastName);
2 Person person = new("Radek", "Janostik");
3 Person person2 = new("Radek", "Janostik");
4 Console.WriteLine(person == person2) // True or False?
5 Console.WriteLine(ReferenceEquals(person, person2)); // T or F?
6
7 Person person3 = person2 with { FirstName = "Kedar" };
```

- Dědičnost – záznam může dědit ze záznamu

Záznamy – komplikace

- Mutovatelný záznam

```
1 public record Person
2 {
3     public string FirstName { get; set; } = default!;
4     public string LastName { get; set; } = default!;
5 };
```

Záznamy – komplikace

- Mutovatelný záznam

```
1 public record Person
2 {
3     public string FirstName { get; set; } = default!;
4     public string LastName { get; set; } = default!;
5 };
```

- Záznam jako hodnotový datový typ (mutovatelný i nemutovatelný):

```
1 public record struct Point
2 {
3     public double X { get; init; }
4     public double Y { get; init; }
5     public double Z { get; init; }
6 }
```

- `record class` – explicitně řečeno, že record bude referenční

Výjimky a logování

- Výjimky jsou dobré, řeknou nám, kde je v programu chyba
- Výjimky by se měli používat na **výjimečné** stavy - ne k normálnímu řízení toku programu
- Pokud očekáváte, že se daný problém bude vyskytovat, nejde o výjimku
  - Tradičně vstup od uživatele:

```
try
{   int input = int.Parse(input); }
catch (Exception e)
{ //chybny vstup }
```

  - místo

```
if (int.TryParse(input, out int num)) { //vstup v poradku }
else {  //chybny vstup }
```

- Mají vyšší náročnost než klasické prostředky řízení toku `if-else`, `switch`

Kde a jak výjimku zpracovat

- Uvažme následující kód (inspirováno vašimi úkoly):

```
1  public int? ElementAt(int index)
2  {
3      try
4      {
5          return Numbers[index];
6      }
7      catch (Exception)
8      {
9          return null;
10     }
11 }
```

- Je toto správně?

Kde a jak výjimku zpracovat

- Uvažme následující kód (inspirováno vašimi úkoly):

```
1  public int? ElementAt(int index)
2  {
3      try
4      {
5          return Numbers[index];
6      }
7      catch (Exception)
8      {
9          return null;
10     }
11 }
```

- Je toto správně?
- Metoda neví nic o tom, kdo a proč ji volal - neví jak se zachovat
- Návratová hodnota nic neříká o tom, zda se stala chyba

Kde a jak výjimku zpracovat

```
1  public int ElementAt(int index)
2  {
3      try
4      {
5          return Numbers[index];
6      }
7      catch (Exception)
8      {
9          // Zalogovat chybu
10         throw;
11     }
12 }
```

- Bez dalších informací nevíme, jak reagovat - delegujeme výš
- Chybu můžeme uložit pro pozdější zpracování
- Metoda má jasně definovanou návratovou hodnotu

Kde a jak výjimku zpracovat

- `try-catch` na nejnižší úrovni nám umožní použít `finally`
  - Pokud potřebujeme provést úkony nehledě na stav metody (Database connection, Stream, ...)
  - Pokud dáme do `finally` kód, který hodí výjimku, máme problém

- Na nejnižší úrovni je i v pořádku:

```
1 public int ElementAt(int index)
2 {
3     return Numbers[index];
4 }
```

- Ale musíme ji dobře zpracovat v další vrstvě

Kde a jak výjimku zpracovat - obecně

- Výjimku nikdy neuklidíme pryč - neodhalitelné chyby
- Výjimku předáváme výš dokud nedojdeme do vrstvy, která ví, jak zareagovat
- Ve větším programu by měl být předem definována vrstva, která se bude starat o zpracování
- Pokud výjimku zpracujeme v běhu - měli bychom chybu uložit pro pozdější vyhodnocení
- `StackTrace` a `Message` jsou kamarádi
- S uživatelem komunikujeme pouze přes UI po zpracování

Závěr

- C# se stále vyvíjí, zkuste s ním držet krok
- Spoustu změn možná nikdy nevyužijete
- Otázka: Složitost jazyka ? $=$? Lepší použitelnost?

Úkol

- Stáhněte si ExceptionProject: `https://apollo.inf.upol.cz/~janostik/slides/ExceptionProject.zip`
- Jedná se o program pro čtení svátků
- Testovací data např. `Rok: 2024`, `kód: CZ`
- Upravte program tak, aby jste ošetřili vsechny problémy a výjimky, které dle vašeho názoru mohou nastat