.Net 6

1 VS 2022

1.1 Projectfile

Als TargetFramework muss net6.0 bzw. net6.0-windows (für WPF) angegeben werden.

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0-windows</TargetFramework>
  <UseWPF>true</UseWPF>
</PropertyGroup>
```

Die unsägliche Einstellung für `DisableWinExeOutputInference` ist nicht mehr nötig, um Ausgaben auf die Konsole zu ermöglichen.

1.2 Global usings

Häufige using-Direktiven kann man jetzt global angeben, indem man sie mit global using notiert. Dadurch müssen diese dann nicht mehr in jedem Sourcefile wiederholt werden.
Grundsätzlich können diese using-Angaben in jedem beliebigen cs-File angegeben werden. Vorgeschlagen werden aber folgende zwei Möglichkeiten.

1.2.1 ImplicitUsings im Projectfile

Durch Angabe von `<ImplicitUsings>enable</ImplicitUsings>` werden je nach Projekttyp bestimmte usings automatisch erzeugt.

```
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

Dadurch wird vom Compiler die Datei *namespace*`.GlobalUsings.g.cs` erzeugt, in dem die globalen usings stehen:

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

Nachteil: man muss wissen, welche global usings bei welchem Projekttyp erzeugt werden.

1.2.2 globalUsings.cs

Man verzichtet auf die implizit erzeugten global usings und erstellt eine eigene Datei, die man am besten globalUsings.cs nennt, und schreibt dort ale benötigten usings hinein.

```
global using System;
global using System.Collections.Generic;
global using System.Linq;
global using System.Reflection;
global using global::System.IO;
```

Hinweis: Auch hier werden nicht benötigte usings „gedimmt“ dargestellt.

1.3 namespace

Die Angabe des Namespace kann jetzt als einzeilige Anweisung angegeben werden.

Vorher:
```
namespace GenAlgFramework
{
    public class Randomizer...
}
```
Nachher:
```
namespace GenAlgFramework;

public class Randomizer...
```

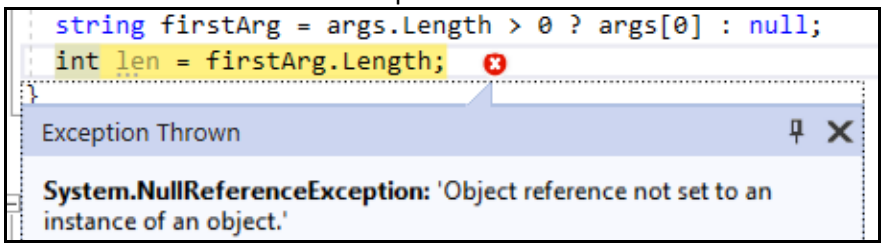
1.4 Nullable reference types

Die mit C#9 eingeführten „nullable reference types“ sind jetzt per Default aktiviert.

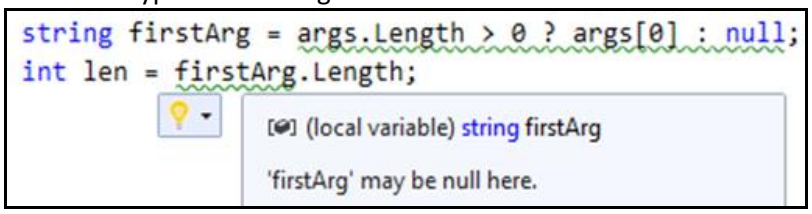
```
<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net6.0-windows</TargetFramework>
  <Nullable>enable</Nullable>
  <UseWPF>true</UseWPF>
</PropertyGroup>
```

Das führt vermutlich zu vielen entsprechenden Warnings.

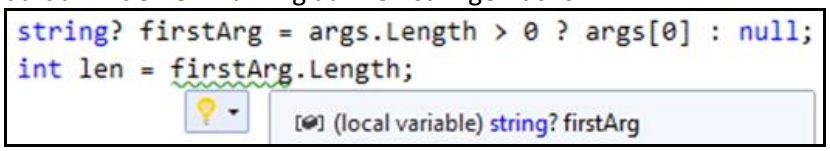
Zur Erinnerung:
Eine häufige Fehlerquelle bei Programmen ist, dass Referenztypen nicht initialisiert sind und daher null sind. Das kann zu einer NullReferenceException führt:



Hilfreich wäre, dass man das explizit zulässt (oder eben auch nicht). Seit C#9 kann man das tun, indem man auch hier beim Typ ein **?** anhängt:



Mit **`string?`** kann man also darauf hinweisen, dass eben der Wert null sein könnte und man wird vom Compiler darauf mit einer Warning aufmerksam gemacht:



1.5 dotnet-ef

Nicht vergessen, das Tool zum Erzeugen der Entity-Klassen zu aktualisieren:

```
dotnet tool update --global dotnet-ef
```

Bzw. falls vorher noch nicht installiert:

```
dotnet tool install --global dotnet-ef --version 6.0
```

Überprüfen:

```
C:\>dotnet-ef --version
Entity Framework Core .NET Command-line Tools
6.0.0
```

Beim Befehl selbst (dotnet ef dbcontext scaffold…) hat sich nichts geändert.

2 Neue Typen

2.1 Neue Typen DateOnly und TimeOnly

Es gibt neben DateTime jetzt neue separate Typen für Datum und Uhrzeit, nämlich DateOnly und TimeOnly. Sie funktionieren so, wie man es sich vorstellt.
Sie entsprechen den Type “date“ bzw. „time“ von SQLServer.

3 LINQ

3.1 Range Parameter

- Bei ElementAt kann man mit ^ den Index von hinten zählen lassen.
- Bei Take kann man range parameter angeben. Die Verwendung sollte aus den Beispielen klar sein:

```
var iValues = Enumerable.Range(1, 10);

int lastButOne = iValues.ElementAt(^2);
Console.WriteLine($"lastButOne = {lastButOne}");

iValues.Take(..3).PrintJsonLike();
iValues.Take(3..).PrintJsonLike();
iValues.Take(2..7).PrintJsonLike();
iValues.Take(^3..).PrintJsonLike();
iValues.Take(..^3).PrintJsonLike();
iValues.Take(^7..^3).PrintJsonLike();
```

```
lastButOne = 9
1,2,3
4,5,6,7,8,9,10
3,4,5,6,7
8,9,10
1,2,3,4,5,6,7
4,5,6,7
```

Hinweis. PrintJsonLike() ist eine von mir geschriebene Extensionmethode.

3.2 TryGetNonEnumeratedCount

Damit kann man die Größe eines Enumerable eruieren, ohne dass dieses zwangsweise durchiteriert wird.

```
iValues.TryGetNonEnumeratedCount(out int count);
Console.WriteLine($"count = {count}");
```

3.3 MaxBy / MinBy / DistinctBy

Mit diesen Methoden kann man Objekte aufgrund einer Property finden:

```
var persons = new (string Name, int Age)[] { ("Susi", 33), ("Hansi", 55), ("Fritzi", 33) };
var oldest = persons.MaxBy(x => x.Age);
var ages = persons.DistinctBy(x => x.Age);
Console.WriteLine(oldest);
ages.PrintJsonLike();
```

```
(Hansi, 55)
(Susi, 33),(Hansi, 55)
```

3.4 Chunk

Mit Chunk kann man ein Enumerable in einzelne Pakete mit fixer Länge aufteilen:

```
var chunks = iValues.Chunk(4);
foreach (var chunk in chunks)
{
    chunk.PrintJsonLike();
}
```

```
1,2,3,4
5,6,7,8
9,10
```

3.5 FirstOrDefault / LastOrDefault / SingleOrDefault mit Default Parameter

Bei diesen Methoden kann man den Defaultwert jetzt selbst bestimmen.

```
var emptyList = new int[] { };
int firstA = emptyList.FirstOrDefault();
int firstB = emptyList.FirstOrDefault(666);
Console.WriteLine($"firstA = {firstA}");
Console.WriteLine($"firstB = {firstB}");
```

```
firstA = 0
firstB = 666
```