

Spracherweiterungen seit C#6

1 ERWEITERUNGEN C# 6	2
1.1 var	2
1.2 nameof	2
1.3 *** Null-Conditional Operator	3
1.4 Initializers	4
1.4.1 *** Property Initializers	4
1.4.2 Index Initializers	4
1.5 *** Exception-catch mit Bedingung	4
2 ERWEITERUNGEN C# 7	5
2.1 Zahlen-Literale	5
2.2 *** Out-Variable	5
2.3 Patterns	5
2.3.1 is-Ausdrücke mit Patterns	5
2.3.2 switch mit Patterns	5
2.4 *** Tupel	6
2.5 *** Expression bodied members	7
2.6 Weitere Features C#7	7
3 ERWEITERUNGEN C#9	8
3.1 Environment information	8
3.2 *** Global code / Top-level statements	8
3.2.1 Mehrere Main	8
3.2.2 args	9
3.3 *** Erweiterung für new	9
3.3.1 *** Instanzvariable	9
3.3.2 *** Methodenparameter	9
3.3.3 *** return	10
3.4 *** Init only setter	10
3.5 Records	10
3.5.1 Deklaration	10
3.5.2 Erzeugung	11
3.5.3 Unveränderlich	11
3.5.4 Generierte Methoden	11
3.5.5 Neue Properties/Methoden	11
3.5.6 Copy	12
3.5.7 Deconstruct	12
3.6 Nullable reference types	12
3.7 Pattern matching enhancements	14

1 Erweiterungen C# 6

1.1 var

Das Schlüsselwort „var“ ermöglicht es, lokale Variable zu deklarieren, ohne den Typ explizit angeben zu müssen. Aus dem zugewiesenen Wert erkennt der Compiler automatisch den Typ der Variablen. Das funktioniert daher nur, wenn die Variable **bei der Deklaration initialisiert** wird.

Beispiele:

```
int x = 666;
var y = 666;
var z = DateTime.Now;
var complex = new Dictionary<string, List<Person>>();
```

Die ersten beiden Deklarationen sind semantisch absolut identisch.

```
Immediate Window
y.GetType().FullName
"System.Int32"
z.GetType().FullName
"System.DateTime"
```

Auch Intellisense wird wie bei einer herkömmlichen Deklaration unterstützt:

```
var z = DateTime.Now;
z.
va
  Add
  AddDays
  AddHours
```

Der Vorteil ist, dass man sich bei der Deklaration die redundante Angabe des Typs sparen kann.

Eine weitere Anwendung ist die Verwendung anonymer Typen (siehe später) – dabei kommt man um das Schlüsselwort **var** nicht mehr herum.

Für **fundamentale Datentypen** (int, double, string,...) sollte man **var** wenn möglich nicht verwenden (Lesbarkeit des Codes leidet).

Merkregel: kann man die Variable nicht wie den (kleingeschriebenen) Typ nennen, dann kein var verwenden!

1.2 nameof

Manchmal möchte man den **Namen einer Funktion oder Variable** ausgeben. Dabei hat man das Problem, dass der Name als String angegeben werden muss und bei einem eventuellen **Refactor** nicht auch umbenannt wird.

Durch die Funktion **nameof()** kann man das jetzt umgehen.

```
public void SomeDummyFunction()
{
    int myTempVariable = 666;
    Console.WriteLine($"{nameof(SomeDummyFunction)}: {nameof(myTempVariable)}={myTempVariable}");
}
```

```
SomeDummyFunction: myTempVariable=666
```

Durch Intellisense wird auch hier die eigentliche Schreibarbeit minimiert:

```
public void SomeDummyFunction()
{
    int myTempVariable = 666;
    Console.WriteLine($"{nameof(som)}");
}
```

Intellisense dropdown:

- SomeDummyFunction
- SomeEvent
- SomeEventArgs
- SomeExtensions

1.3 *** Null-Conditional Operator

Man muss oft Variablen prüfen, ob sie nicht null sind. So bekommt man bei folgendem Code eine Nullpointer-Exception:

```
int[] values = { 1, 2, 3, 4 };
int nr = values.Length;
values = null;
nr = values.Length;
```

Normalerweise würde man das auf eine der beiden Arten lösen:

```
nr = values != null ? values.Length : 0;
int? nr2 = values != null ? values.Length : (int?)null;
```

Mit dem Null-Conditional Operator `?` kann man das eleganter und kürzer notieren:

```
int? nr2 = values?.Length;
int nr = values?.Length ?? 0;
```

Dabei wird beim Method-Chaining abgebrochen, sobald ein Zeiger null ist:

```
List<Person> persons = null;
List<Person> persons2 = new List<Person> { new Person { Firstname="Hansi", Lastname=null} };

string lastname = persons?[0].Lastname;
string lastname2 = persons2?[0].Lastname;
```

Noch deutlicher wird die Schreibersparnis bei folgendem Beispiel:

```
public class Outer
{
    public Middle Middle { get; set; }
}
public class Middle
{
    public Inner Inner { get; set; }
}
public class Inner
{
    public string Name { get; set; }
}
```

Sieht man nichts vor, wird eine Exception geworfen:

```
string name = "";
var obj = new Outer();
try
{
    name = obj.Middle.Inner.Name; Console.WriteLine($"0: {name}");
}
catch (Exception exc)
{
    //something went wrong
    exc {"Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt."}
```

Man müsste also ziemlich umständlich so schreiben:

```
if (obj != null && obj.Middle != null && obj.Middle.Inner != null)
{
    name = obj.Middle.Inner.Name;
    Console.WriteLine($"5: {name}");
}
```

Kürzer geht es so:

```
name = obj?.Middle?.Inner?.Name;
Console.WriteLine($"4: {name}");
```

Also: Der „?“-Operator bricht ab, sobald eine Property null ist und das Gesamtergebnis ist dann null.

Mit dem Null Conditional Operator werden also alle Null-Pointer, die beim Objekt oder einer Zeiger-Property auftreten, überlebt:

```
string name = "";
var obj = new Outer();
name = obj?.Middle?.Inner?.Name; Console.WriteLine($"1: {name}");
obj.Middle = new Middle();
name = obj?.Middle?.Inner?.Name; Console.WriteLine($"2: {name}");
obj.Middle.Inner = new Inner();
name = obj?.Middle?.Inner?.Name; Console.WriteLine($"3: {name}");
obj.Middle.Inner.Name = "xxx";
name = obj?.Middle?.Inner?.Name; Console.WriteLine($"4: {name}");
```

```
1:
2:
3:
4: xxx
```

1.4 Initializers

1.4.1 *** Property Initializers

Man kann jetzt auch Auto-Properties initialisieren, und zwar so, wie man es vermuten würde:

```
class Car
{
    public string Brand { get; set; } = "VW";
    public int Year { get; set; } = 2006;
    public int NrTires { get; } = 4;
}
```

```
var car = new Car();
```

car	{CSharp_Extensions.Car}
car.Brand	"VW"
car.NrTires	4
car.Year	2006

Fehlt der Setter, ist die Property unveränderlich.

1.4.2 Index Initializers

Dictionaries können ab jetzt mit der Indexer-Syntax initialisiert werden:

```
var numberNames = new Dictionary<int, string>
{
    [5] = "fuenf",
    [11] = "elf"
};
```

```
var en2de = new Dictionary<string, string>
{
    ["five"] = "fuenf",
    ["eleven"] = "elf"
};
```

1.5 *** Exception-catch mit Bedingung

Bei einem catch-Block kann man eine Bedingung, d.h. einen boolschen Ausdruck, angeben. Der catch-Block wird nur aufgerufen, wenn die Bedingung true ist.

```
try
{
    //do something weird
}
catch (Exception exc) when (exc.InnerException != null)
{
    Console.WriteLine(exc.InnerException.Message);
}
catch (Exception exc) /*when (exc.InnerException == null)*/
{
    Console.WriteLine(exc.Message);
}
```

2 Erweiterungen C# 7

2.1 Zahlen-Literale

Für lange Zahlen kann man an beliebiger Stelle „_“-Zeichen zur besseren Lesbarkeit einfügen. Diese werden einfach ignoriert.

```
int billion = 1_000_000_000;
```

In Verbindung mit dem „0b“-Präfix für Binärzahlen werden auch diese leichter lesbar:

```
int dual = 0b1010_1100_1111_0110;
```

2.2 *** Out-Variable

Man muss out-Variable nicht mehr vorher definieren.

<pre>private void OutVariables() { GetMinMax(out int x, out var y); Console.WriteLine(\$"min={x} / max={y}"); //old style int a, b; GetMinMax(out a, out b); }</pre>	<pre>public void GetMinMax(out int min, out int max) { //... min = 42; max = 666; }</pre>
---	---

Dabei kann man auch als Typ **var** einsetzen.

Damit ist jetzt auch folgende sehr kurze **TryParse**-Variante möglich:

```
const string sVal = "123";
Console.WriteLine(int.TryParse("123", out var iVal) ? $"Int {iVal}" : $"<{sVal}> is not valid integer");
```

2.3 Patterns

2.3.1 is-Ausdrücke mit Patterns

Der is-Operator funktioniert nicht nur mit Typen, sondern auch mit Ausdrücken.

```
private void IsPattern(object obj)
{
    if (obj is null) return;
    if (obj is int i) Console.WriteLine($"obj is int {i}");
}
```

Dabei wird der Ausdruck auf die angegebene Variable zugewiesen (im Beispiel enthält also **i** den int-Wert).

Das ist etwa für folgende Art von Überprüfung sehr praktisch:

```
private void IsInt(object obj)
{
    if ((obj is int i) || (obj is string s && int.TryParse(s, out i))) Console.WriteLine($"obj is int {i}");
}
```

2.3.2 switch mit Patterns

Bei **switch**-statements kann man neben den üblichen Werten auch mit Type-Patterns prüfen:

```
private void SwitchWithPatterns(Shape shape)
{
    switch (shape)
    {
        case Circle circle:
            Console.WriteLine($"Circle with r={circle.Radius}");
            break;
        case Rectangle square when square.Width == square.Height:
            Console.WriteLine($"Square s={square.Height}");
            break;
        case Rectangle rect:
            Console.WriteLine($"Rectangle {rect.Width}/{rect.Height}");
            break;
        case null:
            Console.WriteLine("Shape is null");
            break;
        default:
            Console.WriteLine("Unknown shape");
            break;
    }
}
```

Bei einem „Treffer“ wird nicht nur das entsprechende **case** abgearbeitet, sondern es steht auch hier wieder die **gecastete Variable** zur Verfügung.

Wichtig: die Reihenfolge ist dabei entscheidend, d.h. der erste Treffer gewinnt!

2.4 *** Tupel

Ähnlich wie die out-Variablen, zielen Tupel darauf ab, mehrere Werte aus einer Funktion zurückgeben zu können, ohne eine eigene Klasse für diese Werte schreiben zu müssen.

Man gibt die Typen der Rückgabewerte mit Beistrich getrennt innerhalb runder Klammern an, und zwar sowohl beim Typ als auch bei der Zuweisung:

```
private (string, string, string) SplitName(string name)
{
    var items = name.Split(' ');
    return (items[0], items[1], items[2]);
}
```

Die Verwendung ist dann in Punkto Notation sehr ähnlich:

```
(string first, string middle, string last) = SplitName("James Tiberius Kirk");
Console.WriteLine($"{first} {middle[0]}. {last}");
```

Die Variablen können dann wie gewohnt benutzt und somit auch verändert werden:

```
(string first, string middle, string last) = SplitName("James Tiberius Kirk");
Console.WriteLine($"{first} {middle[0]}. {last}");
last = "Spock";
Console.WriteLine($"{first} {middle[0]}. {last}");
```

Hinweis: diese Funktionalität ist in Javascript unter dem Namen „Destructuring“ bekannt.

Eine Einsatzmöglichkeit wäre z.B. bei Initialisierung eines Objekts mit einer CSV-Zeile:

```
class ClassWithTupel
{
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Gender { get; set; }
    public string Email { get; set; }
    public ClassWithTupel(string line)
    {
        var items = line.Split(';');
        (Firstname, Lastname, Gender, Email) = (items[0], items[1], items[2], items[3]);
    }
}
```

2.5 *** Expression bodied members

Expression bodied members (also Zuweisung mit „=>“) sind jetzt auch hier möglich:

- Konstruktor
- getter
- setter

Mit folgendem Beispiel sollte die Verwendung klar sein:

ohne:

```
public class EmbodiedClass
{
    private int id;

    public EmbodiedClass(int id)
    {
        this.id = id;
    }
    public int Id
    {
        get { return id; }
        set { id = value; }
    }
}
```

mit:

```
public class EmbodiedClass
{
    private int id;

    public EmbodiedClass(int id) => this.id = id;

    public int Id
    {
        get => id;
        set => id = value;
    }
}
```

Diese Variante erscheint mir vor allem beim Konstruktor sinnvoll, um damit anzuzeigen, dass darin nichts mehr programmiert werden sollte.

2.6 Weitere Features C#7

Folgende Features gibt es auch noch, sie werden hier aber nicht weiter besprochen:

- Lokale Funktionen
- ref bei return

3 Erweiterungen C#9

3.1 Environment information

Mit dem Tool `dotnet-runtimeinfo` kann man sich Informationen zur aktuellen Umgebung ausgeben lassen. Dieses muss vorher mit `dotnet tool install -g dotnet-runtimeinfo` installiert werden:

```
C:\>dotnet tool install -g dotnet-runtimeinfo
Sie können das Tool über den folgenden Befehl aufrufen: dotnet-runtimeinfo
Das Tool "dotnet-runtimeinfo" (Version 1.0.4) wurde erfolgreich installiert.
```

Der neue Befehl kann überall (also auch außerhalb eines Projekts) aufgerufen werden:

```
C:\>dotnet-runtimeinfo
**.NET information
Version: 5.0.10
FrameworkDescription: .NET 5.0.10
Libraries version: 5.0.10
Libraries hash: e1825b4928afa9455cc51e1de2b2e66c8be3018d

**Environment information
OSDescription: Microsoft Windows 10.0.19041
OSVersion: Microsoft Windows NT 10.0.19041.0
OSArchitecture: X64
ProcessorCount: 8
```

3.2 *** Global code / Top-level statements

Man kann beim Hauptprogramm auch ohne `Main()`-Methode programmieren. Folgender Programmcode ist also völlig gleichwertig:

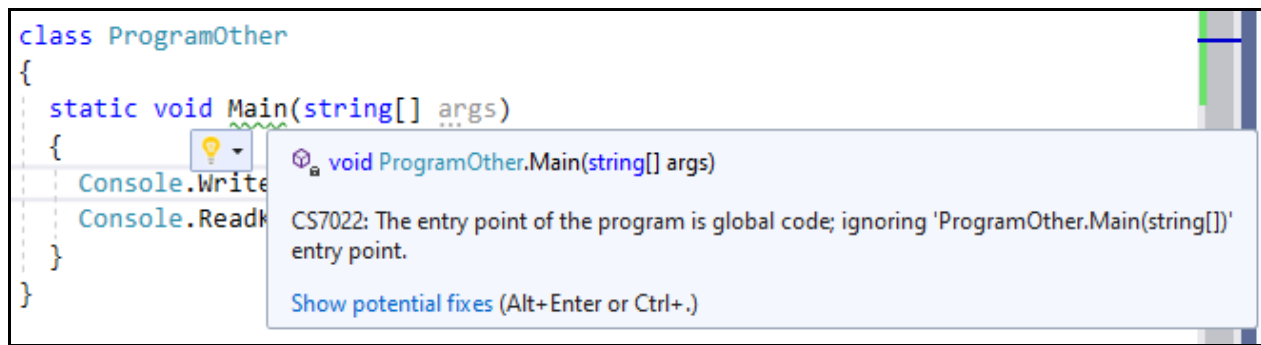
<pre>using System; namespace Net5Demo { class Program { static void Main(string[] args) { Console.WriteLine("Hello World!"); Console.ReadKey(); } } }</pre>	<pre>using System; Console.WriteLine("Hello World!"); Console.ReadKey();</pre>
--	---

Man darf auch nach wie vor Methoden in derartigen Files deklarieren, dabei gilt aber

- Schlüsselwort `static` kann entfallen (sie sind automatisch statisch)
- Dürfen keinen Modifizierer wie `private`, `public`, ... haben

3.2.1 Mehrere Main

Gibt es eine Datei mit Top-level statements, werden etwaige andere `static void Main()` in anderen Dateien ignoriert (App ist aber weiterhin kompilierbar).



3.2.2 args

Man hat Zugriff auf den Parameter **args** (incl. Intellisense), auch wenn er aufgrund der fehlenden Methodensignatur nicht explizit aufscheint.

```
using System;

Console.WriteLine("Hello World!");
int nrArgs = args.Length;
Console.WriteLine($"nrArgs = {nrArgs}");
Console.ReadKey();
```

3.3 *** Erweiterung für new

So, wie man bei der Deklaration **var** anstelle des tatsächlichen Typs angeben kann, wenn aufgrund der Zuweisung der Typ vom Compiler eruiert werden kann, kann man jetzt auch bei **new** den Typ weglassen, wenn dieser deklariert wurde.

Folgende Codezeilen sind gleichwertig:

```
var person = new Person { Firstname = "Hansi", Lastname = "Huber", Age = 66 };
Person person2 = new() { Firstname = "Hansi", Lastname = "Huber", Age = 66 };
```

Auch hier gilt wieder die Regel: wo immer der Compiler den Typ erkennen kann, muss man diesen nicht mehr explizit angeben.

3.3.1 *** Instanzvariable

Das ist vor allem beim Initialisieren von Instanzvariablen von Klassen interessant, weil ja da die Angabe von **var** nicht erlaubt ist.

```
public class Person
{
    private List<string> emails = new List<string>();
}
```

Das kann man jetzt abkürzen:

```
public class Person
{
    private List<string> emails = new();
}
```

3.3.2 *** Methodenparameter

Auch beim Aufruf von Methoden kann man **new()** angeben, wenn man damit ein Objekt mit dem Defaultkonstruktor erzeugen will.

```
static void AddRelatives(Person person, List<Person> relatives)
{
    Console.WriteLine($"Add {relatives.Count} relatives for {person}");
    //do something here
}
```

Will man jetzt eine leere Liste übergeben, kann man das folgendermaßen abkürzen:

```
AddRelatives(person, new());
```

3.3.3 *** return

Analoges gilt für retournierende Werte:

```
static List<Person> ReadPersons()
{
    //implement later
    return new();
}
```

3.4 *** Init only setter

Mit dem neuen Schlüsselwort **init** kann man bei Auto-Properties anstelle von **set** angeben, dass diese beim Initialisieren belegt werden müssen, danach aber nicht mehr verändert werden dürfen.

```
public class Wage
{
    public string Name { get; init; }
    public int Salary { get; set; }
}
```

Der Name darf somit nur beim Erzeugen des Objekts zugewiesen werden und ist ab dann read only: „Init only setters provide a window to change state. That window closes when the construction phase ends“.

```
var wage = new Wage { Name = "Susi", Salary = 3000 };
wage.Salary = 3000;
wage.Name = "Greti";
```

Console string Wage.Name { get; init; }

CS8852: Init-only property or indexer 'Wage.Name' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

Das ist vergleichbar mit **private set**, jedoch mit dem Unterschied, dass nach dem Erzeugen die Property auch innerhalb der Klasse unveränderlich ist.

3.5 Records

Records sind **Referenztypen**, die **unveränderlich** sind und bei Vergleich nicht die Adresse vergleichen, sondern dazu die Werte der Properties heranziehen. Das neue Schlüsselwort heißt **record** und ist so wie **class** oder **struct** zu verwenden.

3.5.1 Deklaration

Die Definition sieht aus wie ein Konstruktor ohne Implementierung. Die Parameter sind als Properties verfügbar und werden daher groß geschrieben.

```
namespace Net5Demo.Models
{
    public record User(string Username, string Password, bool RememberMe);
}
```

Diese Art der Schreibweise nennt man **positional records**.

Die ausführlichere (aber sonst gleichwertige) Schreibweise würde so aussehen:

```
public record User
{
    public string Username { get; }
    public string Password { get; }
    public bool RememberMe { get; }
    public User(string username, string password, bool rememberMe) =>
        (Username, Password, RememberMe) = (username, password, rememberMe);
}
```

Man kann derartige Records wie bei Klassen gewohnt abstract deklarieren und davon auch ableiten. Auch Default-Parameter sind erlaubt:

```
public record User(string Username, string Password, bool RememberMe = false);
```

3.5.2 Erzeugung

Bei der Verwendung kann man nicht erkennen, dass es sich um einen Record und nicht um eine Klasse handelt:

```
var userA = new User("Hansi", "abc", true);
User userB = new(Username: "Hansi", Password: "abc", false);
User userC = new("Hansi", "abc", true);
```

3.5.3 Unveränderlich

Die erzeugten Objekte sind unveränderlich, die Properties sind aber lesbar:

```
string name = userA.Username;
userA.Username = "Pepi";
```

🔑 `string User.Username { get; init; }`

CS8852: Init-only property or indexer 'User.Username' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

Es werden vom Compiler also **init only setters** erzeugt (wie weiter oben besprochen).

3.5.4 Generierte Methoden

Einige Methoden werden vom Compiler automatisch erzeugt, dazu gehören ToString, GetHashCode und Vergleichsoperator.

```
Console.WriteLine(userA);
Console.WriteLine($"hash of userA = {userA.GetHashCode()}");

Console.WriteLine($"isSame A/B = {userA == userB}");
Console.WriteLine($"isSame A/C = {userA == userC}");
```

```
User { Username = Hansi, Password = abc, RememberMe = True }
hash of userA = -1204940242
isSame A/B = False
isSame A/C = True
```

Man erkennt:

- Die ToString()-Methode erzeugt einen JSON-ähnlichen String incl. dem Namen der Klasse
- Zwei Objekte sind gleich, wenn alle Properties gleich sind

3.5.5 Neue Properties/Methoden

Auch neue Methoden und Properties kann man wie gewohnt einfügen, die Angabe von **override** für überschriebene Methoden ist auch hier verpflichtend.

```
public record User(string Username, string Password, bool RememberMe)
{
    public override string ToString() => $"{Username} / {RememberMe}";
    public string ProtectedName => $"{Username} / {new string('*', Password.Length)}";
}
```

Bei ToString soll/kann man die Methode **PrintMembers** verwenden:

```
public override string ToString()
{
    var builder = new StringBuilder();
    PrintMembers(builder);
    return builder.ToString();
}
```

Dabei werden aber wirklich alle Properties ausgegeben, also auch die zusätzlich notierten:

```
Username = Hansi, Password = abc, RememberMe = True, ProtectedName = Hansi / ***
```

Die Methode **PrintMembers** kann man überschreiben. Dabei sollten alle Properties mit **Name = Wert** dem StringBuilder-Parameter hinzugefügt werden. Man könnte es aber auch z.B. so schreiben:

```
protected virtual bool PrintMembers(StringBuilder builder)
{
    builder.Append("Username").Append(": ").Append(Username).Append(", ");
    builder.Append("Password").Append(": ").Append(new string('*', Password.Length)).Append(", ");
    builder.Append("RememberMe").Append(": ").Append(RememberMe);
    return true; //did I add anything to the StringBuilder?
}
```

3.5.6 Copy

Man kann aus einem Objekt eine Kopie erstellen und dabei mit dem Schlüsselwort **with** einzelne Properties verändern. Die zugrundeliegende clone-Methode kopiert alle Properties.

```
var userCopy = userA with { RememberMe = false };
Console.WriteLine($"is copy equal as B: {userB == userCopy}");
Console.WriteLine($"is copy same as B: {userB.GetHashCode() == userCopy.GetHashCode()}");
```

```
is copy equal as B: True
is copy same as B: True
```

3.5.7 Deconstruct

Da vom Compiler ebenfalls eine Deconstruct-Methode erzeugt wird, funktioniert auch folgende Notation:

```
var (first, _, remember) = userA;
Console.WriteLine($"{first} / {remember}");
```

```
Hansi / True
before change: Pepi -->
```

3.6 Nullable reference types

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

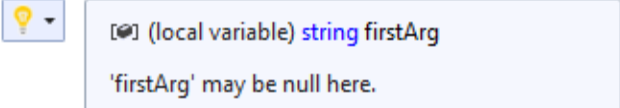
```
string firstArg = args.Length > 0 ? args[0] : null;
int len = firstArg.Length;
```

Exception Thrown

System.NullReferenceException: 'Object reference not set to an instance of an object.'

Hilfreich wäre, dass man das explizit zulässt (oder eben auch nicht). Seit C#9 kann man das jetzt tun, indem man auch hier beim Typ ein **?** anhängt. Dieses Feature muss man momentan noch extra mit **#nullable enable** einschalten (üblicherweise zu Beginn des Files, der Übersicht halber hier aber direkt davor):

```
#nullable enable
string firstArg = args.Length > 0 ? args[0] : null;
int len = firstArg.Length;
```



Oder


```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

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:


```
#nullable enable
string? firstArg = args.Length > 0 ? args[0] : null;
int len = firstArg.Length;
```



Diese Überprüfung findet auch beim Aufruf von Methoden statt:

```
static void HandleArgument(string arg)
{
    int len = arg.Length;
}
```

```
#nullable enable
string? firstArg = args.Length > 0 ? args[0] : null;
HandleArgument(firstArg);
int len = firstArg.Length;
```



CS8604: Possible null reference argument for parameter 'arg' in 'void HandleArgument(string arg)'.

Bei Änderung der Methodensignatur wird die Warning jetzt an anderer Stelle angezeigt (bis man auch das behebt):

```
#nullable enable
string? firstArg = args.Length > 0 ? args[0] : null;
HandleArgument(firstArg);
int len = firstArg.Length;
}

void HandleArgument(string? arg)
{
    int len = arg.Length;
}
```

Für weitere Infos siehe z.B. <https://devblogs.microsoft.com/dotnet/embracing-nullable-reference-types/>

3.7 Pattern matching enhancements

Wird hier nicht besprochen. Das kann man sich z.B. unter folgendem Link anschauen:

<https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#pattern-matching-enhancements>