BASTA!

# NEUE SPRACHFEATURES
# C#-8.0-SPICKZETTEL

Konzept und Text: Dr. Holger Schwichtenberg

## Einschränkung

Alle mit (*) gekennzeichneten Sprachfeatures sind nicht für das klassische .NET Framework verfügbar, sondern nur für .NET Core ab Version 3.0 u. a. .NET-Varianten, die .NET Standard 2.1 implementieren.

## Null-Referenz-Prüfungen/Nullable Reference Types

Nullable Reference Types: Klasse? Aber nicht erlaubt: Nullable<Klasse>

| | Nullable Annotation Context | Nullable Warning Context | Nullable Context (= Annotation + Warning Context) |
|---|---|---|---|
| Bedeutung der Deklaration Klasse k; | Non-Nullable | Nullable | Non-Nullable |
| Bedeutung der Deklaration Klasse? k; | Nullable | Nicht erlaubt (führt zur Warnung) | Nullable |
| Warnung vor NullReferenceExceptions | Nein | Ja | Ja |
| Aktivierung auf Projektebene in der .csproj-Datei | <Nullable> annotations </Nullable> | <Nullable> warnings </Nullable> | <Nullable> enable </Nullable> |
| Aktivierung im Code | #nullable enable annotations | #nullable enable warnings | #nullable enable |
| Deaktivierung im Code | #nullable disable annotations | #nullable disable warnings | #nullable disable |
| Zurücksetzung auf Projekteinstellung | #nullable restore annotations | #nullable restore warnings | #nullable restore |

```
// Normaler Kontext
string name1 = null;
Experte e1 = null;
int id1 = 1;
int? plz1 = null;

// Nullable Context einschalten
#nullable enable
string name2 = null; // Non-Nullable Reference
// Type -> Warnung!
string? name3 = null; // Nullable Reference Type
Experte e2 = null; // Non-Nullable Reference
//Type -> Warnung!
Experte? e3 = null; // Nullable Reference Type
int id2 = 1; // keine Auswirkung auf Value Types!
int? plz2 = null; // keine Auswirkung auf Value
// Types!
s = name2.Trim(); // Warnung: Dereference of a
// possibly null reference
s = name3.Trim(); // Warnung: Dereference of a
// possibly null reference
s = plz2.ToString(); // keine Warnung
// Nullable Context wieder ausschalten
#nullable disable
name2 = null; // keine Warnung
string? name4 = null; // Warnung bei ?

// nur Nullable Annotations Context einschalten
#nullable enable annotations
string name5 = null; // Nullable Reference Type,
// keine Warnung!
string? name6 = null; // Nullable Reference Type
s = name5.Trim(); // keine Warnung
s = name6.Trim(); // keine Warnung
#nullable disable annotations

// nur Nullable Warning Context einschalten
#nullable enable warnings
string name7 = null; // Nullable Reference Type,
// keine Warnung!
string? name8 = null; // Warnung bei ?, Nullable
// Reference Type nicht erlaubt
s = name7.Trim(); // Warnung: Dereference of a
// possibly null reference
s = name8.Trim(); // Warnung: Dereference of a
// possibly null reference
#nullable disable warnings
```

## Null Coalescing Assignment ??=

Mit ??= erfolgt eine Zuweisung nur dann, wenn Variable null enthält

```
p ??= new Person() { ID = 1, Name = "Holger Schwichtenberg" };
```

## Switch Expressions

Methode, die nur aus switch-Fällen besteht. Kein break notwendig!

Beispiel 1: mit Basistyp

```
static string GetKundenTypString(string name, string abc) =>
                                                abc switch {
 "A" => $"{name} ist ein guter Kunde",
 "B" => $"{name} ist ein durchschnittlicher Kunde",
 _ => $"{name} ist ein sonstiger Kunde" };
```

Beispiel 2: Property Pattern über zwei Properties einer Klasse

```
static string GetKontaktTypString(Kontakt k) => k switch {
 { Status: 'A', Art: KontaktArt.Kunde } => $"{k.Name} ist ein
                                                guter Kunde",
 { Status: 'A', Art: KontaktArt.Lieferant } => $"{k.Name} ist ein
                                                guter Lieferant",
 { Status: 'B', Art: KontaktArt.Kunde } => $"{k.Name} ist ein
                                                durchschnittlicher Kunde",
 { Status: 'B', Art: KontaktArt.Lieferant } => $"{k.Name} ist ein
                                                durchschnittlicher Lieferant",
 _ => $"{k.Name} ist ein sonstiger Kontakt" };
```

## Standardimplementierungen in Schnittstellen (*)

Schnittstellen dürfen nun auch Methoden mit Implementierungen und statische Mitglieder enthalten. Einsatzgebiet: Nachträgliche Erweiterung von bestehenden Klassen ohne Neukompilierung der Klassen.

```
interface ILogger {
 string Prefix { get; set; }
 // Methode ohne Implementierung
 void Log(LogLevel level, string message);
 // NEU: Methode mit Implementierung
 // mit Block Body
 public void Log(Exception ex) {
  Count++; // Zugriff auf statisches Mitglied!
  Log(LogLevel.Error, $"{Count:000} {ex.Message}"); }
 // NEU: Methode mit Implementierung
 // mit Expression Body
 public void LogDetails(Exception ex)
                    => Log(LogLevel.Error, ex.ToString());
 // NEU: statisches Mitglied

 public static int Count { get; set; } = 0; }

class ConsoleLogger : ILogger {
 public string Prefix { get; set; } = "LOG:";
 public void Log(LogLevel level, string message) =>
      Console.WriteLine($"{Prefix} {level}: {message}"); }
```

Nutzung der Schnittstelle: Zu beachten ist, dass die in der Schnittstelle implementierten Methoden Log(Exception) und LogDetails(Exception) auf der Variable l nur zugänglich sind, weil die Variable auf ILogger und nicht auf ConsoleLogger typisiert wurde.

```
ILogger l = new ConsoleLogger();
// Nutzung des Klassenmitglieds
l.Log(LogLevel.Info, "C# 8.0 läuft!");
// Nutzung der Schnittstellenimplementierung
var ex = new ApplicationException("Ein Fehler!");
l.Log(ex);
l.LogDetails(ex);
```

Präsentiert von: **basta.net**

Grafische Umsetzung: **meat*** – concept and design © Software & Support Media GmbH

windows .developer

Präsentiert von **entwickler.de**

NEUE SPRACHFEATURES

# C#-8.0-SPICKZETTEL

Konzept und Text: Dr. Holger Schwichtenberg

## Index ^ (*)

```
string[] Namen = { "Leon", "Hannah", "Lukas", "Anna",
"Leonie", "Marie", "Niklas", "Sarah", "Jan", "Laura",
"Julia", "Lisa", "Kevin" };
string n1 = Namen[^2]; // Index Operator: zweiter
// von hinten = "Lisa"
string n2 = Namen[Namen.Length - 2]; // alte
// Schreibweise!
Index i3 = ^2; // neue Klasse System.Index: zweiter von
// hinten
string n3 = Namen[i3]; // zweiter von hinten = "Lisa"
Index i4 = Index.FromEnd(2); // andere Schreibweise:
// zweiter von hinten
string n4 = Namen[i4]; // zweiter von hinten = "Lisa"
```

## Range .. (*)

x..y ist Ausschnitt von x bis vor y, d. h. x ist inklusiv, y ist exklusiv!

```
string[] m1 = Namen[1..3]; // zweiter und dritter:
// "Hannah", "Lukas"
string[] m2 = Namen[6..^4]; // sechs von vorne und
// vier hinten abschneiden (!): "Niklas", "Sarah", "Jan"
string[] m3 = Namen[11..]; // vom 12. Element bis
// Ende: "Lisa", "Kevin"
string[] m4 = Namen[0..^0]; // alle
string[] m5 = Namen[..]; // alle
Range r1 = 1..3; // neue Klasse System.Range
string[] m6 = Namen[r1]; // zweiter und dritter:
//"Hannah", "Lukas"
```

## Explizite readonly-Mitglieder in einer Struktur

Methoden und Properties mit dem Zusatz readonly dürfen den Zustand des Objekts nicht ändern. Automatische Properties mit readonly dürfen kein set; haben und können nur im Konstruktor gesetzt werden.

```
public struct AppInfo {
 public AppInfo(string name, Version version,
                                DateTime? datum) {
  this.Name = name; // readonly --> Zuweisung nur
  // im Konstruktor
  this.Version = version;
  this._Datum = datum; }
 // Property mit Getter und Setter
 public Version Version { get; set; }
 // NEU: readonly Auto Property
 public readonly string Name { get; }
 DateTime? _Datum;
 // NEU: Property nur mit Getter und explizitem
 // readonly
 public readonly DateTime? Datum {
  get {
   // Zuweisungen grundsätzlich nicht erlaubt, da
   // Property readonly
   // Version =?? new Version(0, 0, 0, 0);
   return _Datum; } }
 // NEU: Methode mit readonly
 public readonly int GetVersion() {
  // nicht erlaubt: Version =?? new Version(0, 0, 0, 0);
  return this.Version.Minor; } }
```

## Alternative für verbatim-interpolated Strings

@$ ist nun gleichbedeutend erlaubt zu $@

```
@$"{ID}: {Name} UNC-Pfad: \\Server123\User{ID:000}";
```

## using Declarations ohne explizite Codeblöcke

```
{
using var sw = new StreamWriter(filename);
sw.WriteLine(DateTime.Now.ToLongTimeString());
... }
```

sw ist noch gültig bis zum Blockende, dort erfolgt Aufruf von Dispose().

## Musterbasierte Dispose()-Methode für ref struct

Da Strukturen auf dem Stack (ref struct) keine Schnittstellen implementieren können, wird jetzt Dispose() ohne IDisposable erkannt.

```
public ref struct Ressource  // immer am Stack, nie
// am Heap  {
 public string Name { get; set; }
 public Ressource(string name) {
  this.Name = name;
  Console.WriteLine($"Ressource {Name} erzeugt!"); }
 // C# 8.0: Dispose() wird auch ohne IDisposable
 // erkannt
 public void Dispose() {
  Console.WriteLine($"Ressource {Name} vernichtet"); } }
```

## Asynchrone Streams/await foreach

Notwendige Schnittstelle: IAsyncEnumerable<T>

Dieses Sprachfeature lässt sich in .NET Framework nachrüsten über die NuGet-Pakete Microsoft.Bcl.AsyncInterfaces und Microsoft.Bcl.HashCode.

```
CancellationTokenSource cts = new
                              CancellationTokenSource();
PrintData(cts);
...
cts.Cancel();

// Empfang der Daten von Stream und Ausgabe
public async void PrintData(CancellationTokenSource
                                                  cts) {
  // NEU in C# 8.0: await foreach
  await foreach (var nextValue in GetDataStream()) {
   Console.WriteLine($"{nextValue:000000}");
   if (cts.IsCancellationRequested) {
    Console.WriteLine("!!!Abruch der
                                  Messdatenausgabe!!!");
    return; }

// simuliert eine datensendende Messstelle (alle 250 ms)
static async IAsyncEnumerable<int>GetDataStream() {
  try {
   for (; ; ) {
    await Task.Delay(250);
    yield return new System.Random().Next(1000000);}
  }
  finally {
   Console.WriteLine("GetDataStream: Finally"); } }
```

## Statische lokale Funktionen

Statische lokale Funktionen können im Gegensatz zu den in C# 7.0 eingeführten nichtstatischen lokalen Funktionen NICHT auf Variablen der äußeren Ebenen (der umgebenden Klasse und Methode) zugreifen.

```
class LocalFunctionsDemo {
 public int prop { get; set; } = 42;
 public void Run() {
  int x = 42;
  NonStaticLocalFunc(x);
  StaticLocalFunc(x);

// seit C# 7.0: Nichtstatische lokale Funktion
// kann umgebende Variablen nutzen!
  int NonStaticLocalFunc(int p) {
   int y = 42;
   int x = 43;  // verdeckt x aus Run()
   Console.WriteLine(x);     // OK, lokales x
   Console.WriteLine(y);     // OK, lokales y
   Console.WriteLine(prop);  // OK, prop aus
  // Klasse
   prop++; // kann sogar Werte aus
  // Umgebung verändern
   return p; }

// NEU seit C# 8.0: Kann umgebende Variablen
// NICHT sehen!
  static int StaticLocalFunc(int p) {
   int y = 42;
   int x = 43;  // verdeckt x aus Run()
   Console.WriteLine(x);    // OK, lokales x
   Console.WriteLine(y);    // OK, weil lokales y
   // Console.WriteLine(prop); // nicht erlaubt,
   // weil static local function
   return p;
  } // Ende der statischen lokalen Funktion
 } // Ende der Methode Run()
} // Ende der Klasse
```

## Über den Autor

**Dr. Holger Schwichtenberg**, alias der „DOTNET-DOKTOR", gehört zu den bekanntesten Experten für .NET und Webtechniken in Deutschland. Er hat zahlreiche Fachbücher veröffentlicht und spricht regelmäßig auf Fachkonferenzen wie der BASTA!. Sie können ihn und seine ebenso kompetenten Kollegen für Entwicklungsarbeiten, Schulungen, Beratungen und Coaching buchen.

buero@IT-Visions.de  www.IT-Visions.de
www.dotnet-doktor.de  @dotnetdoktor

## Webadressen

Projekt für das Design der Programmiersprache C#:
https://github.com/dotnet/csharplang

Projekt für die Implementierung des C#-Compilers:
https://github.com/dotnet/roslyn

Versionsgeschichte der C#-Sprachsyntax:
https://github.com/dotnet/csharplang/blob/master/Language-Version-History.md

Versionsgeschichte des neuen C#-Compilers:
https://github.com/dotnet/roslyn/wiki/NuGet-packages

Language Feature Status:
https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md

NuGet-Paket des C#-Compilers:
www.nuget.org/packages/Microsoft.Net.Compilers