Spickzettel („Cheat Sheet“) C# 9.0

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Top-Level Statements

Einsprungmethode Main() ist nicht mehr verpflichtend. Man kann auch direkt "freien" Programmcode in eine beliebige .cs-Datei schreiben. Es darf in einem C#-Projekt nicht mehr als eine Datei geben, die solch freien Code enthält! Falls es ein Top-Level Statement und ein Main() gibt, wird void Main() ignoriert.

Init Only Properties / Init Only Setter

```
class Person {
  public int ID { get; init; }
  … }
```

Ein Init Only Property kann man nur noch an zwei Stellen setzen:

1. Konstruktor der Klasse

```
public Person(int ID)
{  this.ID = ID; }
…
Person hs1 = new Person(123);
```

2. Objekt-Initialisier direkt bei der Instanziierung

```
Person hs2 = new Person() { ID = 123 } ;
```

Beides kann kombiniert werden:

```
Person hs3 = new Person(123) { ID = 456 };
```

Record-Typen

- Untertypus von Klassen mit Codegenerierung durch Compiler
- Wertesemantik bei Vergleich mit == und !=
- Inhaltskopie mit *with* (nur oberste Ebene)
- *ToString()* liefert Inhalt aller öffentlichen Fields und Properties (nur oberste Ebene)
- Immutability möglich mit Init Only Properties
- Vererbung möglich von anderen Record-Typen

Kurzschreibweise für Record-Typen (Immutable)

```
record Person(int ID, string Vorname, string Name, string Status =
"unbekannt");
```

Erbender Record-Typ:

```
record Dozent(int ID, string Vorname, string Name, string Status =
"unbekannt", List<string> Themen = null) : Person(ID, Vorname,
Name, Status);
```

Langschreibweise für Record-Typen (Immutable)

```
record Person {
  private int ID { get; init; }
  public string Vorname { get; init; }
  public string Name { get; init; }
  public string Status { get; init; } = "unbekannt";

 public Person(int id, string vorname, string name,
 string status = "unbekannt") {
 this.ID = id;
 this.Vorname = vorname;
 this.Name = name;
 this.Status = status;
 }
}
```

Erbender Record-Typ:

```
record Dozent : Person {
  public List<string> Themen { get; init; } = new();
  public Dozent(int id, string vorname, string name, List<string>
    themen) : base(id, vorname, name) {
        this.Themen = themen;
 }
}
```

Mischung aus Kurzschreibweise und Langschreibweise

Dieser Record-Typ ist nicht immutable, da es ein beschreibbares Property und ein beschreibbares Field gibt.

```
record Person(int ID, string Vorname, string Name, string Status = "u
nbekannt") {
 public Geschlecht Geschlecht { get; set; }
 public int Alter;

 public string GetAnrede() => Geschlecht switch {
  Geschlecht.f => "Sehr geehrte Frau " + Name,
  Geschlecht.m => "Sehr geehrter Herr " + Name,
  _ => "Hallo " + Name
 };
}
```

Record-Typen verwenden

Record-Instanz erzeugen

```
Person hs = new Person(123, "Holger", "Schwichtenberg") { Status
= "verheiratet" };
```

Alle öffentlichen Properties und Fields ausgeben

```
Print(hs);
```

Kopie der Objektreferenz

```
Person hs_ref = hs;
```

Vergleich der Objektreferenzen

```
Print(hs == hs_ref); // true
```

Wertkopie ohne Veränderung

```
Person hs_Klon1 = hs with { };
```

Wertvergleich

```
Print(hs == hs_Klon1); // true
```

Wertkopie mit Veränderung

```
Person hs_Klon2 = hs with { Status = "geklont" };
```

Wertvergleich

```
Print(hs == hs_Klon2); // false
```

Record-Dekonstruktion → nur möglich für Properties, die in Kurzschreibweise erschaffen wurden; in der Reihenfolge wie im Konstruktor

```
var (_, v, n, s) = hs;
Print ("Vorname: " + v + " Nachname: " + n + " Status: " + s);
```

Target-Typed New Expression

Instanzierung Langschreibweise (seit C# 1.0)

```
Person p = new Person();
Person hs = new Person(123, "Holger", "Schwichtenberg");
List<Person> personList = new List<Person>();
```

Verwendung von var (seit C# 3.0)

```
var p = new Person();
var hs = new Person(123, "Holger", "Schwichtenberg");
var personList = new List<Person>();
```

Target-Typed New Expression (seit C# 9.0)

```
Person p = new();
Person hs = new(123, "Holger", "Schwichtenberg");
List<Person> personList = new();
```

Auch beim Setzen von Properties und Fields sowie bei der Parameterübergabe möglich; erschwert hier aber die Lesbarkeit:

```
public void Umziehen(Adresse adresse) {
   this.Adresse = adresse;
}
…
p.Umziehen(new() { Ort = "Essen", Land = "DE" });
```

Weitere Möglichkeiten für partielle Methoden

Rückgabewerte, Sichtbarkeitsangabe und *out*-Parameter sind nun erlaubt. Allerdings muss es bei Verwendung dieses Features dann auch zwingend eine Implementierung geben!

```
partial class MeineKlasse {
  // partielle Methode alten Typs --> keine Implementierung
erforderlich!
  partial void M1();

  // partielle Methode neuen Typs, da "private" --> Implementierung
erforderlich!
  private partial void M2();

  // partielle Methode neuen Typs, da "int" --> Implementierung
erforderlich!
  private partial int M3();
 }
```

Zweiter Teil der partiellen Klasse:

```
 partial class MeineKlasse {
  private partial void M2() { }
  public partial int M3() { return 42; }
 }
```

Discard-Variable _ im Lambda-Ausdrücken

Seit C# 9.0 ist es in Lambda-Ausdrücken erlaubt, mit _ anzuzeigen, dass man einen Parameter nicht verwenden möchte.

```
DruckeGruss("...", (_) => $"Hallo {vorname} {name}!");
```

Spickzettel („Cheat Sheet“) C# 9.0

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Statische anonyme Funktionen

Eine anonyme Funktion kann auf alle Variablen der Umgebung zugreifen. Seit C# 9.0 kann man dies unterbinden mit dem Zusatz static. Der Entwickler kann sich also mit dem Zusatz static davor schützen, versehentlich auf Daten der Umgebung zuzugreifen.

```
string vorname = "Holger";  string name = "Schwichtenberg";
```

normale anonyme Funktion: vorname und name sind nutzbar

```
DruckeGruss("Guten Abend", (gruss) => $"{gruss} {vorname}
{name}!");
```

static → vorname und name sind NICHT nutzbar

```
// DruckeGruss("Guten Abend", static (gruss) => $"{gruss}
{vorname} {name}!");
```

Erweiterung des Pattern Matching in Bedingungen

In Pattern ist *and* statt &&, *or* statt || und *not* statt ! zu verwenden.

Vergleiche mit null

```
if (eingabe is null) { Print("Leer"); }
if (eingabe is not null) { Print("Nicht leer"); }
```

Typprüfung

```
if (eingabe is int z) { Print("Zahl: " + z); }
if (eingabe is not int) { Print("Keine Zahl!"); }
```

Wertevergleiche mit is

```
if (eingabe is >= 0 and <= 100) { Print($"Eingabe ist zwischen 0 u
nd 100!"); }
if (eingabe is < 0 or > 100) { Print($"Eingabe ist nicht zwischen 0 u
nd 100!"); }
if (eingabe is 0 or 100) { Print($"Eingabe ist Extremwert!"); }
if (eingabe is (>= 0 and <= 10) or (>= 90 and <= 100)) { Print(
$"Eingabe ist hoher oder niedriger Wert!"); }
```

Wertevergleiche mit is not

```
if (eingabe is not (>= 0 and <= 100)) { Print("Eingabe ist nicht zwi
schen 0 und 100!"); }
if (eingabe is not >= 0 or not <= 100) { Print("Eingabe ist nicht zw
ischen 0 und 100!"); }
```

Pattern mit Klammern

```
if (d is not (>= 'a' and <= 'z' or >= 'A' and <= 'Z')) { Print("Kein B
uchstabe!"); }
```

Gleichbedeutend, da *and* eine höhere Präzedenz als *or* hat

```
if (d is not (>= 'a' and <= 'z') or (>= 'A' and <= 'Z')) { Print("Kein
Buchstabe!"); }
```

Erweiterung des Pattern Matching in Switch Expressions

Switch Expression mit Simplified Type Pattern

```
var ausgabe = eingabe switch  {
    int => "Eingabe ist eine Zahl.",
    string => "Eingabe ist eine Zeichenkette.",
    _ => "Eingabe ist etwas anderes."
};
```

Switch Expression mit Relational Pattern

```
var ausgabe = eingabe switch  {
    < 0 => "Eingabe ist negative Zahl.",
    <= 100 => "Eingabe ist zwischen 0 und 100.",
    _ => "Eingabe ist größer als 100."
};
```

Switch Expression mit Relational Pattern und Logical Pattern

```
var ausgabe = eingabe switch  {
    < 0 => "Eingabe ist negative Zahl.",
    0 or 100 => "Alles oder nichts.",
    > 0 and < 100 => "Eingabe ist zwischen 0 und 100.",
    _ => "Eingabe ist größer als 100."
};
```

Modul-Initialisierer

- wird beim Laden einer .NET-Assembly automatisch vor allen anderen Routinen aufgerufen
- Runtime .NET 5.0 oder höher
- in Klasse, die *public* oder *internal* ist
- *public* oder *internal*
- statisch (*static*)
- parameterlos und hat keinen Rückgabewert (*void*)
- nicht-generisch
- annotiert mit *[System.Runtime.CompilerServices.ModuleInitializerAttribute]*

```
public class ModuleInitializerClass {
  [ModuleInitializer]
  public static void ModuleInitializer()  {
    var ass =
System.Reflection.Assembly.GetExecutingAssembly().GetName();
    Print($"ModuleInitializerClass1: Modul wird geladen: {ass.Name}
v{ass.Version.ToString()}");
  }
}
```

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
C# 9.0-Buch
https://it-visions.de/CS9Buch

Source-Code-Generator

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Text;
namespace ITVisions.CodeGenerators {
 [Generator]
 public class CompileInfoGenerator : ISourceGenerator {
  public void Execute(GeneratorExecutionContext context)
  {
   // Source wird erzeugt
   var source = @"
using System;
namespace ITVisions {
    public static class CompileInfo {
        public static string GetInfo() {
            return ""Assembly wurde kompiliert am [NOW] von [USER]"";
        }
    }
}";
   source = source.Replace("[NOW]", DateTime.Now.ToString());
   // hier vier \ notwendig, da im generierten Code \\ stehen muss!
   source = source.Replace("[USER]",
          System.Environment.UserDomainName + "\\\\" +
          System.Environment.UserName);
   // neuer Code wird injiziert
   context.AddSource("CompileInfoGenerator",
SourceText.From(source, Encoding.UTF8));
  }
  public void Initialize(GeneratorInitializationContext context)  {  }
 }
}
```

Einbindung des Source Code Generators in .csproj:

```
<ProjectReference
Include="..\Projekt\Generators.csproj"
OutputItemType="Analyzer"
ReferenceOutputAssembly="false" />
```

Verwendung des Source Code Generators:

```
Console.WriteLine(ITVisions.CompileInfo.GetInfo());
```

Über den Autor

Dr. Holger Schwichtenberg (MVP) gehört zu den bekanntesten Experten für die Programmierung mit Microsoft-Produkten in Deutschland. Er hat zahlreiche Bücher zu .NET und Webtechniken veröffentlicht und spricht regelmäßig auf der BASTA! u.a. Fachkonferenzen. Sie können ihn und seine Kollegen für Schulungen, Beratungen und Softwareentwicklung buchen.
E-Mail: **anfragen@IT-Visions.de** Website: **www.IT-Visions.de**
Weblog: **www.dotnet-doktor.de** Twitter: **@dotnetdoktor**