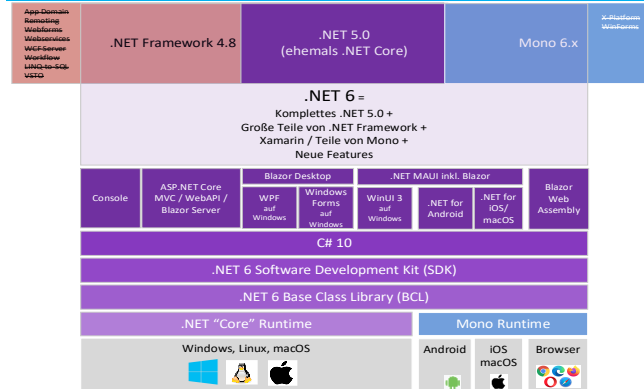


Spickzettel („Cheat Sheet“) C# 10.0 und .NET 6 – neue Sprachfeatures und neue APIs

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

V1.1.0/ 08.11.2021 / Seite 1 von 2

.NET 6 im Überblick



Neue .NET CLI-Befehle in .NET 6

Status aller installierten .NET SDK-Versionen:

```
dotnet sdk check
```

Optionale SDK-Workloads online suchen und verwalten:

```
dotnet workload search / install / list / update / repair / uninstall
```

Projektvorlagen online suchen und aktualisieren:

```
dotnet new Thema --search --language C# --author Microsoft
```

```
dotnet new --update-check
```

```
dotnet new --update-apply
```

Upgrade von .NET Framework:

```
dotnet tool install -g upgrade-assistant
```

```
upgrade-assistant "AlleProjektmappe.sln"
```

Übersetzen und Starten mit Hot Reload:

```
dotnet watch
```

C# 10: Namensräume auf Dateiebene

Namensraumdeklaration auf Dateiebene (File-Scoped Namespaces) ohne geschweifte Klammern; müssen zu Beginn der Datei vor allen Typdeklarationen erscheinen

```
namespace de.WWWings.PassagierSystem;
```

```
public class Passagier : de.WWWings.Person { ... }
```

```
public class Buchung { ... }
```

C# 10: Globaler Namensraumimport

Globaler, dateiübergreifender Namensraumimport (gilt für alle Dateien im Projekt)

```
global using System;
```

```
global using static System.Console;
```

```
global using IS = System.Runtime.InteropServices;
```

Alternativ: Globaler Namensraumimport in der Projektdatei .csproj:

```
<ItemGroup>
```

```
<Using Include="System.Runtime.InteropServices" />
```

```
<Using Include="System.Console" Static="True" />
```

```
<Using Include="System.Runtime.InteropServices" Alias="IS" />
```

```
</ItemGroup>
```

C# 10: Impliziter Namensraumimport

Auf C# 10 basierende Projekte haben zudem eine Reihe von Namensräumen, die automatisch importiert werden und nicht mehr explizit importiert werden müssen ("Implizite Namensräume").

Microsoft.NET.Sdk	System System.Collections.Generic System.IO System.Linq System.Net.Http System.Threading System.Threading.Tasks	Microsoft.NET.Sdk.Web	System.Net.Http.Json Microsoft.AspNetCore.Builder Microsoft.AspNetCore.Hosting Microsoft.AspNetCore.Http Microsoft.AspNetCore.Mvc Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging
Microsoft.NET.Sdk.Worker	Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging	Quelle: Microsoft	

Deaktivierung einzelner impliziter Namensräume in der Projektdatei:

```
<ItemGroup>
```

```
<Using Remove="System.Threading.Tasks" />
```

```
</ItemGroup>
```

Deaktivierung aller impliziten Namensräume:

```
<PropertyGroup>
```

```
<ImplicitUsings>disable</ImplicitUsings>
```

```
</PropertyGroup>
```

C# 10: Verbesserung der String Interpolation

String Interpolation in Konstanten unter der Voraussetzung:

alle Platzhalter werden mit Konstanten befüllt.

```
const string Vorname = "Holger";
```

```
const string Nachname = "Schwichtenberg";
```

```
const string GanzerName = $"Dr. {Vorname} {Nachname}";
```

Die String Interpolation ist in C# 10 deutlich schneller als C# 6.0 bis 9.0, da nun im Untergrund ein **InterpolatedStringHandler**, eine Variante eines String Builders arbeitet

[devblogs.microsoft.com/dotnet/string-interpolation-in-c-10-and-net-6].

C# 10: Lambda-Verbesserungen

Lambda mit Typherleitung (Natural Function Type)

```
var f1 = (string s, int i) => s.Substring(0, i);
```

Lambda mit expliziten Rückgabtyp (Explicit Lambda Return Type)

```
var f2 = FileSystemInfo () => new DirectoryInfo(@"c:\Windows");
```

Lambda mit Annotationen/Attributen

```
var f3 = [return: NotNull] string ([SensitiveData] string name)
```

```
=> "Hallo " + name;
```

C# 10: Record Structs

Ein **record** in C# 9 war immer eine Klasse. Seit C# 10 auch:

- record class**: Dies ist gleichbedeutend mit der Verwendung von **record** ohne Zusatz. Es entsteht wie bisher eine Klasse, also ein Referenztyp. Properties aus Primärkonstruktor sind immutable.
- record struct**: Hier entsteht eine Struktur, also ein Werttyp (implizit erbend von **System.ValueType**), aber mutable.
- readonly record struct**: Immutable-Variante der record struct

```
public record class Person_ImmutableRecordClass(int ID,  
string Vorname, string Name, string Status = "unbekannt")
```

```
{ public int Alter { get; set; } }
```

```
public record struct Person_MutableRecordStructs(int ID,  
string Vorname, string Name, string Status = "unbekannt")  
{ public int Alter { get; set; } = 0; // Initialisierung ist Pflicht! }
```

```
public readonly record struct Person_ImmutableRecordStructs(int ID,  
string Vorname, string Name, string Status = "unbekannt")  
{ public int Alter { get; init; } = 0; // Initialisierung ist Pflicht! }
```

C# 10: Sealed ToString() in Record-Typen

Bereits in C# 9.0 war es möglich, auch in einem Record-Typen Methoden zu überschreiben, auch wenn diese Methoden Teil der automatischen Codegenerierung für den Record waren, z.B. ToString(). Damit wurde die automatische Implementierung außer Kraft gesetzt. Ab C# 10.0 ist es nun nur bei ToString() erlaubt, dass dabei das Schlüsselwort "sealed" eingesetzt wird. Damit wird verhindert, dass ein davon erbender Record-Typ ToString() wieder überschreibt mit der automatischen Implementierung. Folglich gilt eine sealed ToString()-Implementierung auch für alle abgeleiteten Record-Typen.

```
public record class Person(int ID, string Vorname, string Name,  
string Status = "unbekannt") : IDisposable {  
    public sealed override string ToString()  
    { return $"Person #{ID}: {Vorname} {Name}"; } ... }
```

C# 10: Caller Argument Expressions

Per Annotation **System.Runtime.CompilerServices**.

CallerArgumentExpressionAttribute kann eine aufgerufene Methode erfahren, welche Ausdrücke (Variablennamen bzw. Formeln) hinter den vom Aufrufer übergebenen Werten stehen.

```
public void CheckRange(int value, int minValue, int maxValue,  
[CallerArgumentExpression("value")] string? valueEx = null,  
[CallerArgumentExpression("minValue")] string? minValueEx = null,  
[CallerArgumentExpression("maxValue")] string? maxValueEx = null)  
{ if (value > maxValue) throw new  
    ArgumentOutOfRangeException(nameof(value),  
    $"{value} ({valueEx}) muss zwischen {minValue}  
    ({minValueEx}) und {maxValue} ({maxValueEx}) liegen!"); }
```

Bei diesem Aufruf obiger Methode

```
var a = 5; var max = Convert.ToInt32(Math.Floor(Math.PI));  
Validation.CheckRange(a * 3, 1, max);  
entsteht folgender Text in der ggf. ausgelösten Fehlermeldung:  
'15 (a * 3) muss zwischen 1 (1) und 3 (max) liegen!'
```

C# 10: Erweiterung des Property Pattern

Über das Property Pattern kann man Unterobjekte jetzt eleganter ansprechen. Statt der Verschachtelung

```
if (p is Person { Firma: { Firmenname: "IT-Visions" } }) { ... }
```

ist ab C# 10 auch die Punktnotation erlaubt:

```
if (p is Person { Firma.Firmenname: "IT-Visions" }) { ... }
```

Spickzettel („Cheat Sheet“) C# 10.0 und .NET 6 – neue Sprachfeatures und neue APIs

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

V1.1.0/ 08.11.2021 / Seite 2 von 2

Neue Properties bei System.Environment

Environment.ProcessId und Environment.ProcessPath

Basistypen System.DateOnly und System.TimeOnly

Neue Instanzen erzeugen:

```
DateOnly d1 = new DateOnly(2021, 11, 9); // 9.11.2021  
TimeOnly t1 = new TimeOnly(11, 59, 20, 10); // 11:59:20.10
```

DateOnly und TimeOnly aus DateTime:

```
DateOnly d2 = DateOnly.FromDateTime(DateTime.Now);  
TimeOnly t2 = TimeOnly.FromDateTime(DateTime.Now);
```

DateOnly und TimeOnly zu DateTime zusammensetzen:

```
DateTime dt = d2.ToDateTime(t2);
```

Mengentyp System.Collections.Generic.PriorityQueue

```
var pq = new PriorityQueue<string, int>();  
pq.Enqueue("Christian", 3); pq.Enqueue("Annalena", 2);  
pq.Enqueue("Olaf", 1); pq.Enqueue("Robert", 2);  
pq.Enqueue("Armin", 4); pq.Enqueue("Janine", 3);  
while (pq.Count > 0) { Console.WriteLine(pq.Dequeue()); }  
Liefert: Olaf, Annalena, Robert, Christian, Janine, Armin
```

Platz in List<T>, Stack<T> & Queue<T> reservieren

```
List<double> list = new(); list.EnsureCapacity(count);
```

Neue LINQ-Operatoren

Index und Range in Take():

```
menge.Take(3..) entspricht menge.Skip(3)  
menge.Take(3..7) entspricht menge.Skip(3).Take(4)  
menge.Take(^3..) entspricht menge.TakeLast(3)  
menge.Take(3.. ^3) bedeutet: ohne erste drei und ohne letzte drei  
menge.Chunk(3) teilt Menge in 3er-Blöcke + Rest  
Neue ...By()-Operatoren: DistinctBy(), UnionBy(), IntersectBy(),  
ExceptBy(), MaxBy(), MinBy()  
Beispiel 1:
```

```
var bundestag = new List<Partei>() {  
    new Partei("CDU", 151), new Partei("SPD", 205), usw. };  
int meiste = bundestag.Max(x => x.Sitze); // 205  
int wenigste = bundestag.Min(x => x.Sitze); // 1  
Partei groeßte = bundestag.MaxBy(x => x.Sitze); // { "SPD", 205 }  
Partei kleinste = bundestag.MinBy(x => x.Sitze); // { "SSW", 1 }  
Beispiel 2:
```

```
var personen = new (string Name, string Farbe)[] { ("Annalena",  
    "grün"), ("Olaf", "rot"), ("Norbert", "rot"), ("Saskia", "rot"), ("Robert",  
    "grün"), ("Armin", "schwarz"), ("Christian", "gelb") };  
var auswahl = personen.DistinctBy(x => x.Farbe).ToList();  
liefert pro Farbe nur die erste gefundene Person
```

Direkte Speicheroperationen

Neue Klasse System.Runtime.InteropServices.NativeMemory
unsafe {

```
byte* m1 = (byte*) NativeMemory.AllocZeroed(1024 * 100);  
// oder Alloc() ohne Setzen des Speichers auf 0
```

```
IntPtr p1 = (IntPtr)m1;  
Console.WriteLine("Speicheradresse: " + p1);  
*m1 = 42;  
Console.WriteLine("Inhalt: " + *m1); // 42  
  
IntPtr p2 = new IntPtr(p1.ToInt64() + 10);  
Console.WriteLine("Speicheradresse: " + p2);  
byte* m2 = (byte*)p2.ToPointer();  
*m2 = 124;  
Console.WriteLine("Inhalt: " + *m2); // 124  
  
NativeMemory.Free(m1); }
```

Direkte Dateizugriffe ohne Stream-Objekte

Dateiinhalte schreiben mit System.IO.RandomAccess:

```
using (Microsoft.Win32.SafeHandles.SafeFileHandle handle =  
File.OpenHandle(path, FileMode.Create, FileAccess.Write)) {  
    var arr = new byte[100];  
    arr = Encoding.ASCII.GetBytes("1234567890abdef...");  
    int startpos = 0;  
    Span<byte> buffer = arr;  
    RandomAccess.Write(handle, buffer, startpos);  
    handle.Close(); }
```

Dateiinhalte lesen mit System.IO.RandomAccess:

```
using (Microsoft.Win32.SafeHandles.SafeFileHandle handle =  
File.OpenHandle(path)) {  
    // Dateilänge ermitteln  
    long length = RandomAccess.GetLength(handle);  
    // 100 Bytes einlesen, erste 10 Zeichen überspringen  
    var arr = new byte[100];  
    Span<byte> buffer = arr;  
    var bytesRead = RandomAccess.Read(handle, buffer, 10);  
    Console.WriteLine("Puffergröße: " + buffer.Length);  
    Console.WriteLine("Eingelesene Bytes: " + bytesRead);  
    Console.WriteLine("Inhalte an Position 11:" + (char)arr[10]);  
    handle.Close(); }
```

Erweiterungen für System.Text.Json

- zirkuläre Referenzen ignorieren mit ReferenceHandler.Ignore (alternativ zu ReferenceHandler.Preserve)
- Ereignisse in zu serialisierenden Objekten:
IJsonOnSerializing.OnSerializing()
IJsonOnSerialized.OnSerialized()
IJsonOnDeserializing.OnDeserializing()
IJsonOnDeserialized.OnDeserialized()
- Serialisierungsreihenfolge für Properties:
[JsonPropertyOrder(Zahl)]
- Asynchrone Streaming-Serialisierung und -Deserialisierung
IAsyncEnumerable<T> ae =
JsonSerializer.DeserializeAsyncEnumerable<T>(stream, options);
await foreach (var p in ae) { ... }

- Synchroner Serialisierung aus/in Stream:
JsonSerializer.Serialize(stream, meinObjekt, options);
JsonSerializer.Deserialize<Klasse>(stream);
- optionaler Einsatz von Source Code Generators
[JsonSerializable(typeof(Person))]
internal partial class JsonContext : JsonSerializerContext { }
...
JsonSerializer.Serialize(person, JsonContext.Default.Person);

DOM-basiertes API System.Text.Json.Nodes.JsonNode:
string jsonText = @"{"Name":"Holger Schwichtenberg",
 "Ort":"Essen", "Alter":48, "Firmen":["www.IT-Visions.de",
 "Essen"]}";
// JsonNode aus JSON-Text erstellen
JsonNode jNode = JsonNode.Parse(jsonText);
// Zugriff auf Unterknoten
int value = (int)jNode["Alter"]; // 48
// oder: value = jNode["Alter"].GetValue<int>(); // 48
// Wert für Knoten setzen und persistieren
jNode["Alter"] = 49;
// Neuen Knoten an Liste anfügen
JsonObject m = new();
m["MAXIMAGO"] = "Dortmund";
jNode["Firmen"].AsArray().Add(m);
// JSON-Dokument persistieren im Dateisystem
using var fs = new FileStream(@"t:\autor.json", FileMode.Create);
using var writer = new Utf8JsonWriter(fs);
jNode.WriteTo(writer);

Webadressen

Website des Autors

www.dotnet6.de

Neuerungen in .NET 6

docs.microsoft.com/dotnet/core/dotnet-six

Neuerungen in C# 10

docs.microsoft.com/dotnet/csharp/whats-new/csharp-10

Neuerungen in ASP.NET Core 6.0

docs.microsoft.com/aspnet/core/release-notes/aspnetcore-6.0

Neuerungen in Entity Framework Core 6.0

docs.microsoft.com/ef/core/what-is-new/ef-core-6.0/whatsnew

Breaking Changes in .NET 6

docs.microsoft.com/en-us/dotnet/core/compatibility/6.0

Über den Autor

Dr. Holger Schwichtenberg gehört zu den bekanntesten Experten für Softwareentwicklung in Deutschland. Er hat zahlreiche Bücher und Fachartikel zu .NET und Webtechniken veröffentlicht und spricht regelmäßig auf Fachkonferenzen. Sie können ihn und seine Kollegen für Schulungen, Beratungen, Coaching und Softwareentwicklung buchen.

E-Mail: anfragen@IT-Visions.de Website: www.IT-Visions.de

Weblog: www.dotnet-doktor.de Twitter: @dotnetdoktor

