

.NET: Spracherweiterungen in C# 3.0 – C# 7.0

Version 2.3

© J. Heinzlreiter

Historische Entwicklung von C#

Version	Erscheinungsjahr	Wesentliche Neuerungen
C# 1.0	2001	OOP
C# 2.0	2005	<u>Generics</u> , Generierung von Skalaren (viel)
C# 3.0	2007	LINQ, Auto-Properties
C# 4.0	2010	Ko-/Kontravarianz, dynamische Typisierung
C# 5.0	2012	await/async
C# 6.0	2015	Null-Conditional-Operator, String-Interpolation, nameof-Operator
C# 7.0	2017	Tupel, Pattern-Matching

Neuerungen in C# 3.0

- Die zentrale Erweiterung von C# 3.0 ist LINQ (Language Integrated Query).
- Die meisten Erweiterungen sind Voraussetzung für die Verwendung von LINQ-Ausdrücken.
 - Einfache Initialisierung von Objekten → Objekt- und Behälterinitialisierer.
 - Erzeugung von Objekten *anonymer Typen*.
 - Deklaration von Variablen, in denen Objekte von anonymen Typen gespeichert werden können → *automatische Typableitung*.
 - Einfache Definition von Funktionen und Prädikaten → *Lambda-Ausdrücke*.
 - Erweiterung von Interfaces um neue Methoden → *Erweiterungsmethoden*.
 - Repräsentation von Ausdrücken → *Expression Trees*.
- Andere Erweiterungen:
 - Automatische Implementierung von *Properties*.
 - Automatische Ableitung von *Feldtypen*.

Objektinitialisierer (*Object Initializer*)

- *Problem:* Steht kein passender Konstruktor zur Verfügung, müssen Objekte relativ umständlich über Properties initialisiert werden.

```
Employee empl = new Employee();  
empl.Id = 1;  
empl.Name = "Dobler";  
empl.City = "Hagenberg";
```

- *Lösung:* Die Initialisierung von Properties kann bei der Erzeugung eines Objekts in einer Initialisierungsliste erfolgen:

```
Employee empl =  
    new Employee { Id = 1, Name = "Dobler", City = "Hagenberg" };
```

Behälterinitialisierer (*Collection Initializer*)

- Auf ähnliche Weise können Behälter initialisiert werden:

```
List<Employee> empls =  
    new List<Employee> {  
        new Employee { Id = 1, Name = "Jacak", City = "Linz" },  
        new Employee { Id = 2, Name = "Dobler", City = "Hagenberg" }  
    }
```

- Für jedes Objekt in der Initialisierungsliste wird die Methode *Add()* des Behälters aufgerufen.
- Auch assoziative Behälter können so initialisiert werden:

```
Dictionary<int, Employee> emplDict =  
    new Dictionary<int, Employee> {  
        { 1, new Employee { Id = 1, Name = "Jacak", City = "Linz" } },  
        { 2, new Employee { Id = 2, Name = "Dobler",  
                            City = "Hagenberg" } }  
    };
```

Automatische Typableitung (*Local Variable Type Inference*)

- **Problem:** Sowohl bei der Deklaration als auch bei der Initialisierung einer Variablen muss der Typ angegeben werden:

```
List<Employee> empls = new List<Employee>() { ... }
```

- **Lösung:** Der Compiler bestimmt automatisch den Typ der Variablen aus dem impliziten Typ des zugewiesenen Ausdrucks:

```
var empls = new List<Employee>() { ... }
```

- Mit **var** deklarierte Variablen sind statisch typisiert und haben keinesfalls das Verhalten von *Vars* von dynamischen Sprachen: Compiler leitet automatisch Datentyp ab

```
empls = "some string"; // → Syntaxfehler
```

- Darf nur für die Deklaration von lokalen Variablen verwendet werden. zur Laufzeit lässt sich dieser Typ nicht ändern
- **var** sollte mit Bedacht verwendet werden.
- Die Hauptanwendung von **var** sind Variablen, denen Objekte **anonymer Klassen** zugewiesen werden.

Anonyme Typen

- Anonyme Typen sind Klassen, deren Schnittstelle erst bei der **Erzeugung** von **Objekten dieser Klasse festgelegt** wird.
- Da die Klassendeklaration entfällt, spricht man von **anonymen Typen**.

```
var obj = new { Id = 1, Name = "Dobler" };  
Console.WriteLine(obj.GetType());  
// → f__AnonymousType0`2[System.Int32,System.String]
```

- Die Struktur der **anonymen C#-Klasse** wird durch
 - die **verwendeten Properties** und
 - die **impliziten Typen der zugewiesenen Ausdrücke**festgelegt.
- Bei Java wird die Struktur der **anonymen Klasse** aus dem implementierten **Interface bzw. der Basisklasse** abgeleitet.
- Hauptanwendung: Projektionen in **LINQ-Ausdrücken**.

Lambda-Ausdrücke

- Methoden mit Delegateparametern sind sehr flexibel einsetzbar.

```
public delegate bool Predicate<T>(T obj);  
private static IEnumerable<T> FilterWhere(IEnumerable<T> numbers,  
                                         Predicate<T> filter) {  
    foreach (T n in numbers)  
        if (filter(n))  
            yield return n;  
}
```

- An diese Methoden können anonyme Methoden übergeben werden:

```
var oddNumbers = FilterWhere(numbers,  
                             delegate(int n) { return n % 2 != 0; });
```

- Lambda-Ausdrücke ermöglichen eine einfachere Schreibweise für anonyme Methoden:

```
var oddNumbers = FilterWhere(numbers, n => n % 2 != 0);
```

- In LINQ-Ausdrücken werden sehr häufig anonyme Methoden benötigt.

Erweiterungsmethoden (*Extension Methods*)(1)

- *Problem:* Hat man keine Kontrolle über ein Interface, kann zusätzliche Funktionalität nur mittels Klassenmethoden implementiert werden.

```
static class Enumerator {  
    public static int Sum(IEnumerable<int> numbers) {  
        int sum = 0;  
        foreach (int i in numbers) sum += i;  
        return sum;  
    }  
}
```

*IEnumerator<int> l = ...
int s = l.Sum();*

- Beim Aufruf der Methode muss das Objekt, dessen Klasse erweitert worden ist, als Parameter übergeben werden:

```
var numbers = new List<int> { 2, 3, 5, 7 };  
int s = Enumerator.Sum(numbers);
```

→ prozedural

- Man möchte die Erweiterungsmethode aber wie eine Objektmethode aufrufen.

Erweiterungsmethoden (*Extension Methods*)(2)

- Lösung:

```
namespace EnumeratorExt {  
    public static class Enumerator {  
        public static int Sum(this IEnumerable<int> numbers) {  
            int sum = 0;  
            foreach (int i in numbers) sum += i;  
            return sum;  
        }  
    }  
}
```

ausschließlich statische Methoden enthalten

das ist eine Erweiterungsmethode von IEnumerable

- Die Erweiterungsmethode kann nun wie eine Objektmethode aufgerufen werden:

```
using EnumeratorExt; → geht nicht ohne das  
var numbers = new List<int> { 2, 3, 5, 7 };  
int s = numbers.Sum();
```

- Die Erweiterungsmethode ist nur in jenem Namenraum sichtbar, in dem sie definiert wurde.
- Der Compiler generiert Code, der die Erweiterungsmethode wie eine Klassenmethode behandelt.

Automatische Implementierung von Properties

- Für Properties wird häufig nur die Standardimplementierung benötigt:

```
class Employee {  
    private string name;  
    public string Name {  
        get { return name; }  
        set { name = value; }  
    }  
}
```

- Diese Standardimplementierung kann der C#-3.0-Compiler automatisch erzeugen:

```
class Employee {  
    public string Name { get; set; }  
}
```

*Sichtbarkeit von get und set
kann ich ändern*

LINQ: Language Integrated Query

- LINQ ist eine **Abfragesprache**, die direkt in C#/VB.NET eingebettet ist.
- Der Compiler kann **syntaktische Korrektheit der Abfrage überprüfen**.
- Abfragen können für auf beliebige Datenbestände durchgeführt werden, für die ein LINQ-Provider existiert:
 - Felder und Objektbehälter (LINQ to Objects)
 - XML (LINQ to XML)
 - Datenbanktabellen (LINQ to Entities, LINQ to NHibernate, ...)

- Beispiel:

```
using System.Linq;  
IEnumerable<Employee> employees = ...;  
var query = from e in employees
```

```
    where e.City == "Linz"
```

```
    select new { Id = e.Id, Name = e.Name };  
impliziter Lambdaausdruck
```

- LINQ-Abfragen werden in **Aufrufe von Erweiterungsmethoden** übersetzt:

```
var query = employees.Where(e => e.City == "Linz")  
    .Select(e => new { Id = e.Id, Name = e.Name });  
anonyme Typen und Typableitung  
implizit verwendet
```

Neuerungen in C# 4.0

- Benannte Parameter
- Optionale Parameter
- Dynamisch typisierte Variable (Schlüsselwort `dynamic`)
- Kovarianz und Kontravarianz

Benannte Parameter

- Die Zuordnung von **Aktual- zu Formalparametern** erfolgt in den meisten Sprachen über **deren Position in der Aufrufliste**.

```
class Rational {  
    public Rational(int num, int denom) { ... }  
    ...  
}  
  
Rational r1 = new Rational(1, 2);
```

- Durch Qualifizierung mit **dem Namen des Formalparameters** können in C# die **Aktualparameter in beliebiger Reihenfolge übergeben** werden.

```
Rational r2 = new Rational(num:1, demon:2); // Rational(1,2)  
Rational r3 = new Rational(demon:2, num:1); // Rational(1,2)
```

- Hauptanwendungsgebiet: Parameterübergabe bei **Methoden mit langen Parameterlisten** und **optionalen Parametern**.

Optionale Parameter

- Im Kopf einer Methode können Standardwerte für Parameter festgelegt werden:

```
class Rational {  
    public Rational(int num = 0, int denom = 1) { ... }  
    ...  
}
```

- Optionale Parameter müssen am Ende der Parameterliste definiert werden.
- Beim Aufruf müssen für Parameter am Ende der Liste keine Werte übergeben werden.

```
Rational r1 = new Rational(); // Rational(0,1)  
Rational r2 = new Rational(5); // Rational(5,1)
```

- Mithilfe benannter Parameter können Parameter selektiv übergeben werden:

```
Rational r3 = new Rational(denom: 5); // Rational(0,5)
```

Dynamische Typprüfung

- Wird eine Variable **dynamic** deklariert, wird die Typprüfung von der Übersetzungs- in die Laufzeit verlagert.

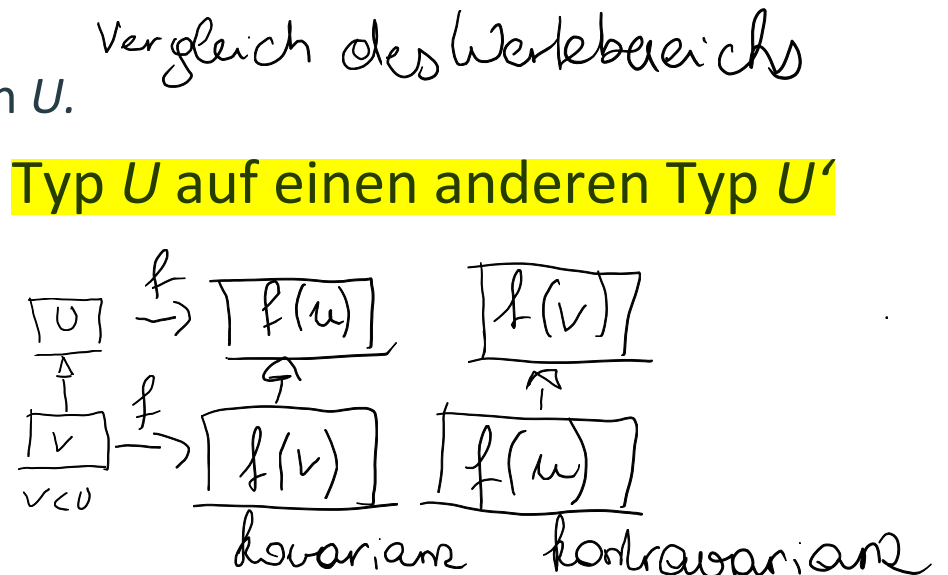
```
dynamic d = "abc"; so wie var oder int  
if (condition) d = new int[] { 5, 17, 3, 8 };  
int len = d.Length; // runtime checks if method Length is available  
object obj = d[1]; // runtime checks if indexer is defined for dynamic type d.
```

- Anwendung: **Einbindung von Skriptsprachen und COM-Komponenten**

```
string script = @"def factorial(n):  
    for i in range(1, n): n = n * i  
    return n";  
ScriptEngine engine = Python.CreateEngine();  
ScriptScope scriptScope = engine.CreateScope();  
ScriptSource scriptSource = engine.CreateScriptSourceFromString(script,  
    SourceCodeKind.Statements);  
scriptSource.Execute(scriptScope);  
dynamic mathScript = scriptScope;  
int fact = mathScript.factorial(5);
```


Kovarianz und Kontravarianz

- Gegeben seien zwei Typen U und V mit einer Relation $<$, die eine Ordnung auf Typen definiert.
 - Beispiel 1: $\text{int} < \text{float} < \text{double}$
 - Beispiel 2: $V < U$, falls V ist eine Unterklasse von U .
- Sei $f: U \rightarrow U'$ ist eine Abbildung, die einen Typ U auf einen anderen Typ U' abbildet.
 - Beispiel: $T \rightarrow \text{IEnumerable}<T>$ oder $T \rightarrow T[]$
- Sei $V < U$. Dann ist
 - f ist **kovariant**, wenn $f(V) < f(U)$
 - f ist **kontravariant**, wenn $f(U) < f(V)$
 - f ist **invariant**, wenn f weder kovariant noch kontravariant ist.



Kovarianz bei Feldern

- Die Abbildung $T \rightarrow T[]$ eines Referenztyps T ist in C# und Java kovariant (Ordnungsrelation ist Vererbungsbeziehung).
 - U is subtype of $V \Rightarrow U[]$ is subtype of $V[]$

```
object[] objArr;  
string[] strArr = new string[] { "abc", "efg" };  
objArr = strArr;
```

- Allerdings geht dadurch die Typsicherheit verloren.
 - Zur Laufzeit kann eine `ArrayTypeMismatchException` (C#) auftreten.

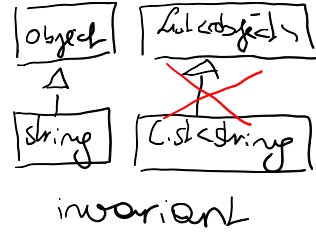
```
objArr[0] = DateTime.Now; // throws ArrayTypeMismatchException  
string s = strArr[0];
```

Lesender Zugriff ok - Schreibend nicht
Anwendung von Funktionen wird zum Problem

- Für Wertetypen (Ordnungsrelation Wertebereich) gilt dies nicht.

```
double[] fa = new float[3]; // syntax error
```

Kovarianz bei Generics (1)



- Die Abbildung $T \rightarrow \text{GenericType}\langle T \rangle$ eines Referenztyps T ist in C# 3.0 und Java generell nicht kovariant.
 - V is subtype of $U \Rightarrow \text{GenericType}\langle V \rangle$ is subtype of $\text{GenericType}\langle U \rangle$

```
List<object> objList;  
List<string> strList = new List<string>();  
objList = strList; // syntax error
```

invariant

- Kovarianz hätte im Allgemeinen auch hier einen Verlust der Typsicherheit zur Folge:

```
objList.Add(DateTime.Now); // would result in a runtime error  
string s = strList[0];     // would result in a runtime error (cast exception)
```

Kovarianz bei Generics (2)

- Falls bei einem generischen Typ `G<T>` der Typparameter `T` bei sämtlichen Methoden ausschließlich bei Ausgangsparametern verwendet wird, geht die Typsicherheit nicht verloren:

```
IEnumerable<object> objEnum;  
IEnumerable<string> strEnum = new List<string> { "abc", "efg" };  
objEnum = strEnum; // valid from C# 4.0 on
```

```
IEnumerator<object> e = objEnum.GetEnumerator();  
while (e.MoveNext()) Process(e.Current);
```

IEnumerator hat nur
eine Methode
deswegen geht es so

geht nur auf IF,
diese sind speziell
gekennzeichnet

- In C# 4.0 ist daher die Abbildung $T \rightarrow \text{GenericInterface}<\text{out } T>$ kovariant.
- Das Schlüsselwort **out** stellt sicher, dass `T` nur zur Deklaration von Ausgangsparametern verwendet wird.

```
public interface IEnumerable<out T> {  
    IEnumerator<T> GetEnumerator();  
}
```

Kontravarianz bei Generics

- Wird bei $G<T>$ der Typparameter T ausschließlich für Eingangsparameter verwendet, gilt:

$$V < U \Rightarrow G<U> < G<V>$$

```
IComparable<Person> personComparer = new Person { Name = "King" };  
IComparable<Student> studentComparer;  
studentComparer = personComparer;  
  
Student student = new Student { MatNo = "se0000", Name = "Mayr" };  
int res = studentComparer.CompareTo(student);
```

- In C# 4.0 ist daher die Abbildung $T \rightarrow \text{GenericInterface}<\mathbf{in} T>$ kontravariant.
- Das Schlüsselwort ***in*** stellt sicher, dass T nur zur Deklaration von Eingangsparametern verwendet wird.

```
public interface IComparable<in T> {  
    int CompareTo(T other);  
}
```

Ko- und Kontravarianz bei Delegates

- Die Abbildung $T \rightarrow \text{delegate } T D()$ ist kovariant (T ist der Typ eines Ausgangsparameters).

```
delegate Person PersonFactoryHandler();  
private static Student CreateStudent() { ... }  
PersonFactoryHandler pfHandler = CreateStudent;  
Person p = pfHandler();
```

- Die Abbildung $T \rightarrow \text{delegate void } D(T t)$ ist kontravariant (T ist der Typ eines Eingangsparameters).

```
delegate void StudentHandler(Student s);  
private static void PrintPerson(Person p) { ... }  
StudentHandler sHandler = PrintPerson;  
sHandler(new Student { ... });
```

Neuerungen in C# 5.0

- Vereinfachte asynchrone Programmierung (async/await)
- Attribute zur Ermittlung von Aufrufdaten (Tracing)

Vereinfachte **asynchrone Programmierung (1)**

- Problem:
 - Zur Erhaltung der Responsivität sollten länger andauernde blockierende Methodenaufrufe vermieden werden (Windows 8: $\geq 50\text{ms}$)
 - Die Synchronisation mit Threads ist sehr aufwändig und fehleranfällig.
- In C# 5.0 können Methoden **async** deklariert werden:
 - Methode kann die **Kontrolle** an den **Rufer zurückgeben**, bevor alle **Anweisungen durchgeführt** wurden.
nur in Methode wird dort nicht weitergemacht
 - Rückgabewert der Methode muss **void, Task oder Task<T>** sein.
- Eine asynchrone Methode kann dem **Schlüsselwort await** auf die **Ergebnisse länger andauernder Berechnungen warten**.
Programm wird dadurch nicht blockiert, aber wir geben Kontrolle an Rufer wenn Funktion fertig, mach ich lei
 - Methode, in der **await** verwendet wird, muss **asynchron sein**.
await weiter
 - **await** kann auf Methoden angewandt werden, die **Task oder Task<T>** zurückgeben.
 - Ist die **Ausführung der Methode**, auf deren Ergebnis **gewartet wurde abgeschlossen**, wird die Ausführung in der **rufenden Methode fortgesetzt**.
sobald Methode länger dauert muss es auch async Methode geben

Vereinfachte asynchrone Programmierung (2)

- Der **Synchronisationskontext** regelt, welcher **Thread** die Kontrolle erhält, wenn eine **await-Operation** abgeschlossen wurde.
- Der Synchronisationskontext von **Windows-Forms**-, **WPF**- und **Windows-Store-Anwendungen** sorgt dafür, dass die **gesamte asynchrone Methode im UI-Thread durchgeführt wird** → keine Synchronisation notwendig.

```
async Task<object> CallingMethod() {  
    SyncMethod1();  
    await AsyncMethod();  
    SyncMethod2();  
}
```

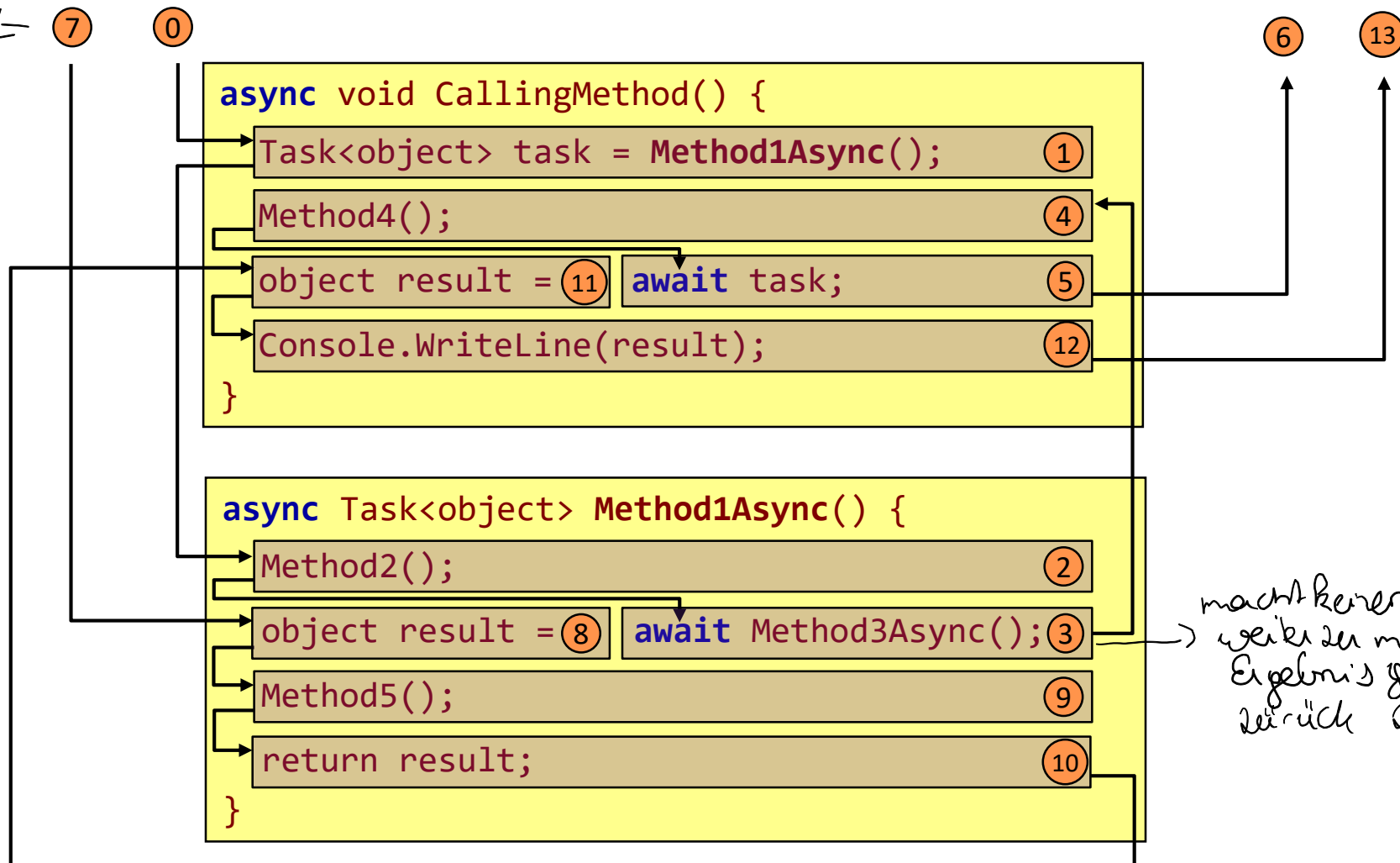
UI-Thread

Worker-Thread

- Ist **kein Synchronisationskontext** vorhanden, wird die Kontrolle an den *TaskScheduler* übergeben, der wiederum **einen Thread aus seinem Thread-Pool** mit der **Abarbeitung der restlichen Methode betraut**.

Vereinfachte asynchrone Programmierung (3)

wenn 3
beliebig ist!



Neuerungen in C# 6.0

- Null-Conditional Operator
- Initialisierer für Auto-Properties
- Verkürzte Methodendefinition
- Operator nameof
- String-Interpolation
- Statische Imports
- Filter bei Ausnahmen
- await in catch- und finally-Blöcken erlaubt

Null-Conditional Operator

- `expr.Property` bzw. `expr[index]` → `NullReferenceException`, falls `expr == null`.
- `expr?.Property` bzw. `expr?[index]` → `null`, falls `expr == null`.
- Beispiel:

```
public class Person {  
    public string Name { get; set; }  
    public int Age { get; set; }  
    public Person[] Children { get; set; }  
}
```

```
Person person = new Person { ... };  
// Person person = null;  
string name = person?.Name;  
int? age = person?.Age;  
string childName = person?.Children?[0]?.Name;
```

*hier steht entweder null oder
Name des ersten Kindes*

Initialisierer für Auto-Properties

- Auto-Properties können wie Felder initialisiert werden.
- Auch „read only“-Properties können auf diese Weise initialisiert werden.
- Beispiel:

```
public class Person {  
    public string Name { get; set; } = "John";  
    public int Age { get; } = 20;  
}
```

Verkürzte Methodendefinition

- Für Methoden- und Property-Definitionen existiert eine verkürzte Schreibweise.
- Ist nur dann möglich, wenn Methodendefinition aus einer einzigen Anweisung besteht.
- Bei Methoden mit Rückgabeparameter, muss return weggelassen werden.
- Beispiel:

```
public class Point {  
    private int x, y;  
    public Point(int x, int y) { ... }  
    public Point Translate(int dx, int dy) => new Point(x+dx, y+dy);  
    public int X => x; // read-only properties  
    public int Y => y;  
}
```

Operator nameof

- Mit nameof können Programmkonstrukte wie Variablen, Klassen- und Methodennamen in eine Zeichenkette konvertiert werden.
 - Vorteil: Schreibfehler können verhindert werden
- Beispiel:

```
public class Person {  
    private string lastName;  
    public string LastName{  
        get { return lastName; }  
        set {  
            name = value;  
            OnNotifyPropertyChanged(nameof(LastName));  
        }  
    }  
}
```

String-Interpolation

- Mit String-Interpolation können formatierte Zeichenketten einfacher erzeugt werden.
 - Vorteil: Durch den Wegfall von Platzhaltern ist die Zeichenketten-generierung weniger fehlerträchtig.

- Mit Platzhaltern:

```
logger.Log(String.Format("{0} + {1} = {2:F2}", a, b, a+b));
```

- Mit String-Interpolation:

```
logger.Log($"{a} + {b} = {a+b:F2}");
```


Neuerungen in C# 7.0

- Tupel
- Pattern-Matching
- out-Variablen
- Lokale Funktionen
- Referenzvariablen
- Verbesserungen bei Literalen

Tupel

- Werte unterschiedlichen Datentyps können zu *Tupel* zusammengefasst werden.
 - Werte werden auf Wertetyp `System.ValueTuple<T1, ..., Tn>` abgebildet.
 - NuGet-Paket `System.ValueTuple` muss hinzugefügt werden.
- Syntax:

```
(int, string) addr1 = (4232, "Hagenberg");  
Console.WriteLine($"{addr1.Item1} {addr1.Item2}");  
  
(int zip, string city) addr2 = (4020, "Linz");  
Console.WriteLine($"{addr2.zip} {addr2.city}");
```

```
(int, string) CreateAddress() { return (1010, "Wien"); }  
  
(int zip, string city) addr3 = CreateAddress();  
var (zip1, city1) = CreateAddress();  
(var zip2, var city2) = CreateAddress();  
(var zip3, _) = CreateAddress();
```

Pattern-Matching

- C# 7.0 definiert folgende Arten von Mustern (patterns):
 - Konstante Muster: `v is null`
 - Typ-Muster: `v is DateTime`
 - Variablen-Muster: `v is DateTime d`
- Beispiele:

```
dynamic v = 42;
if (v is int i)
    Console.WriteLine($"v is an integer with value {i}");
switch (person) {
    case Student s:
        Console.WriteLine($"Student with matnr {s.MatNr}"); break;
    case Person p when p.Age >= 18:
        Console.WriteLine($"Adult person with name {p.Name}"); break;
    case null:
        Console.WriteLine("<null>"); break;
}
```

Referenzvariablen

- C# unterstützte auch schon bisher Call-by-Reference.
- C# 7.0 ermöglicht das Speichern von und das Retournieren von Referenzen in Funktionen.
- Beispiel:

```
private static void TestReferences() {  
    ref int FindRef(int index, int[] a) {  
        for (int i = 0; i < a.Length; i++)  
            if (a[i] == index)  
                return ref a[i];  
        throw new IndexOutOfRangeException();  
    }  
    int[] array = { 1, 2, 3 };  
    ref int r = ref FindRef(2, array);  
    r = 9; // array = { 1, 9, 3 }  
}
```

Verbesserungen bei Literalen

- C# 7.0 unterstützt Binär-Literale:

```
var b = 0b1010;
```

- Mit dem Literaltrenner _ können Literale übersichtlich definiert werden:

```
var d = 1_000_000_000;  
var x = 0xFA_F9;  
var b = 0x1111_1010_1111_1001;
```

async Main (C# 7.1)

- Problem:

```
static async Task SomeAsyncFunc() {  
    await Task.Delay(1000);  
}  
  
static void Main(string[] args) {  
    var result = SomeAsyncFunc().GetAwaiter().GetResult();  
    ...  
}
```

hier blockiere ich
wirklich, da man
nicht async sein
darf

- Lösung:

```
static async Task Main(string[] args) {  
    var result = await SomeAsyncFunc();  
}
```

- C# 7.x ($x \geq 1$) muss in Projekteinstellungen von VS von **explizit aktiviert werden**-