

# ***Generics und Extensions Delegates und Lambdas***

1	GENERICIS	2
1.1	Beispiel ohne Generics	2
1.1.1	Middle für int	2
1.1.2	Middle für double	2
1.1.3	Middle für string	2
1.2	Middle generisch	3
1.3	Mehrere Typen	3
1.4	Generic constraints	4
1.4.1	Beispiel	4
2	EXTENSION METHODS	6
2.1	Generics	6
3	DELEGATE	8
3.1	Motivation	8
3.2	Definition	8
3.3	Zuweisung	8
3.4	Aufruf	8
3.5	Delegate als Parameter	9
3.5.1	Objekt übergeben (falsch!)	9
3.5.2	Delegate übergeben (richtig!)	9
3.6	Multicast Delegate	10
4	LAMBDA EXPRESSIONS	11
4.1	Lambda als Parameter	11
4.2	Mehrere Parameter	12
4.3	Ohne Parameter	12
4.4	Mehrere Anweisungen	12
4.5	Returntyp void	12
4.6	Lokale Funktionen (ab C# 7)	13
5	GENERISCHE DELEGATES IN C#	14
5.1	Action<T1, T2, ..., T16>	14
5.1.1	Action	14
5.2	Func<T1, T2, ..., T16, TResult>	14
5.2.1	Predicate<T>	15
5.2.2	EventHandler<T>	15
6	ANONYME TYPEN	16

# 1 Generics

Generics spielen immer dann eine Rolle, wenn man Funktionen für unterschiedliche Typen zur Verfügung stellen möchte.

Gäbe es Generics nicht, müsste man in diesen Fällen **viel Code kopieren**, nur um ihn dann minimal zu ändern.

## 1.1 Beispiel ohne Generics

Als Beispiel soll eine Erweiterungsmethode `Middle` programmiert werden, die für eine Liste das mittlere Element retourniert.

### 1.1.1 Middle für int

Der Code für eine Integer-Liste ist sehr einfach:

```
public int Middle(List<int> list)
{
    if (list.Count == 0) return 0;
    int idx = list.Count / 2;
    int value = list[idx];
    return value;
}
```

Anwendung:

```
var integers = new List<int> { 5, 76, 3, 93, 143, 5, 11, 67 };
int middleInt = Middle(integers); //gives 143
```

### 1.1.2 Middle für double

Bräuchte man dieselbe Funktionalität für eine Double-Liste, müsste man den Code kopieren und nur den Typ ersetzen:

```
public double Middle(List<double> list)
{
    if (list.Count == 0) return 0;
    int idx = list.Count / 2;
    double value = list[idx];
    return value;
}
```

Und er Aufruf:

```
var doubles = new List<double> { 1.23, 68.25, 44.55, 96.12, 393.4567, 2.45 };
double middleDouble = Middle(doubles); //gives 96.12
```

### 1.1.3 Middle für string

Für string wieder Copy/Paste:

```
public string Middle(List<string> list)
{
    if (list.Count == 0) return null;
    int idx = list.Count / 2;
    string value = list[idx];
    return value;
}
```

```
var strings = new List<string> { "Hansi", "Pauli", "Heinzi", "Susi", "Pepi" };
string middleString = Middle(strings); //gives "Heinzi"
```

## 1.2 Middle generisch

Man möchte eine Funktion also nicht mit unterschiedlichen Werten aufrufen, sondern mit unterschiedlichen Typen! Dazu braucht man eine Möglichkeit, den **Typ einer Funktion symbolisch** angeben zu können – im Beispiel oben als int, double und int.

Alle drei obigen Funktionen kann man somit durch eine Funktion ersetzen:

```
public ANYTHING Middle(ANYTHING list)
{
    if (list.Count == 0) return default(ANYTHING);
    int idx = list.Count / 2;
    ANYTHING value = list[idx];
    return value;
}
```

Üblicherweise schreibt man aber für den Typ-Platzhalter nicht **ANYTHING** sondern **T**:

```
public T Middle<T>(List<T> list)
{
    if (list.Count == 0) return default(T);
    int idx = list.Count / 2;
    T value = list[idx];
    return value;
}
```

Man ersetzt einfach an allen Stellen, an denen man vorhin beim Kopieren den Typ geändert hat, den Typ durch eine Variable. Der Variablenname für den Typ ist wirklich frei wählbar – man könnte ihn auch Quaxi nennen, üblich ist aber eben **T**.

Um anzugeben, was der Variablenname für den Typ ist, gibt man diesen zusätzlich in spitzen Klammern beim Methodennamen an, also wird aus **Middle** → **Middle<T>**.

Man beachte: die Aufrufe für obige Listen funktionieren, ohne etwas ändern zu müssen!

Warum?

Aus dem Objekt, auf das Middle angewandt wird, ergibt sich der Typ der Liste und somit der Typ von T.

Middle(integers) → integers ist List<int> → T ist int

Middle(doubles) → doubles ist List<double> → T ist double

## 1.3 Mehrere Typen

Man ist dabei nicht auf einen Typen beschränkt:

```
public string FirstAppend<T, S>(List<T> list, S append)
{
    string s = $"{list[0]}_{append}";
    return s;
}
```

Man nimmt also aus einer Liste **beliebigen Typs (T)** das erste Element und hängt ein Objekt eines **beliebigen Typs (S)** an. Durch String-Interpolation wird vom **append**-Parameter die ToString()-Methode aufgerufen.

```
string s1 = FirstAppend(strings, "aaa");
string s2 = FirstAppend(strings, 123);
string s3 = FirstAppend(integers, "bbb");
string s4 = FirstAppend(integers, 666);
string s5 = FirstAppend(strings, new Person { Firstname = "Quaxi", Lastname = "Huber" });
```

```
s1 = Hansi_aaa
s2 = Hansi_123
s3 = 5_bbb
s4 = 5_666
s5 = Hansi_Huber Quaxi
```

## 1.4 Generic constraints

Man kann den Typ der Methode noch einschränken. Siehe <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters>:

Constraint	Description
where T : struct	The type argument must be a value type. Any value type except Nullable<T> can be specified. For more information about nullable types, see Nullable types.
where T : class	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type.
where T : unmanaged	The type argument must not be a reference type and must not contain any reference type members at any level of nesting.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

### 1.4.1 Beispiel

Angenommen, man möchte eine Liste mit Default-Objekten erzeugen. D.h. man setzt voraus, dass die zugehörige Klasse einen Default-Konstruktor enthält.

```
var persons = Generate<Person>(5);
```

Das muss man dem Compiler über einen Constraint mitteilen:

```
private List<T> Generate<T>(int nr) where T : new()
{
    var list = new List<T>();
    for (int i = 0; i < nr; i++)
    {
        list.Add(new T());
    }
    return list;
}
```

Ließe man den Constraint weg, bekäme man einen Compilerfehler:

```
private List<T> Generate<T>(int nr) //where T : new()
{
    var list = new List<T>();
    for (int i = 0; i < nr; i++)
    {
        list.Add(new T());
    }
    return list;
}
```

T in MainWindow.Generate<T>

Cannot create an instance of the variable type 'T' because it does not have the new() constraint

Aufgrund von Boxing funktioniert es auch für fundamentale Type:

```
List<int> xxx = Generate<int>(10);
```

## 2 Extension Methods

Mit Extension Methods (=Erweiterungsmethoden) kann man beliebigen Klassen Methoden hinzufügen (selbst wenn diese Klassen als „sealed“ markiert sind).

Folgendes muss beachtet werden:

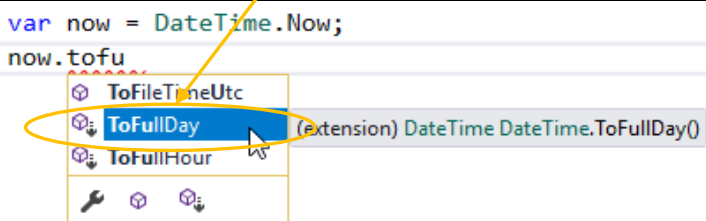
1. Die Erweiterungsmethode muss in einer **statischen Klasse** definiert werden
2. Die **Methode** selbst muss **statisch** deklariert werden
3. Die Klasse, die erweitert werden soll, muss das erste Argument der Methode darstellen und außerdem mit dem **Schlüsselwort this** markiert werden.

Beispiel:

```
public static DateTime ToFullHour(this DateTime t)
{
    return t.AddMinutes(-t.Minute)
        .AddSeconds(-t.Second)
        .AddTicks(-t.Ticks % TimeSpan.TicksPerSecond);
}
public static DateTime ToFullDay(this DateTime t)
{
    return t.AddHours(-t.Hour).ToFullHour();
}
```

```
public static class MyExtensions
{
    public static DateTime ToFullHour(this DateTime t)
    {
        return t.AddMinutes(-t.Minute)
            .AddSeconds(-t.Second)
            .AddTicks(-t.Ticks % TimeSpan.TicksPerSecond);
    }
    public static DateTime ToFullDay(this DateTime t)
    {
        return t.AddHours(-t.Hour).ToFullHour();
    }
}
```

Bei der Anwendung kann dann nicht mehr unterschieden werden, ob es sich um eine Erweiterungsmethode handelt oder um eine innerhalb der Klasse definierte Methode handelt (stimmt nicht ganz - wie man unten sieht ist das Symbol bei Intellisense leicht abgeändert).



```
var now = DateTime.Now;
now.ToFullDay();
```

### 2.1 Generics

Damit könnt man die generische Methode Middle von oben auch als Extensionmethode schreiben:

```
public static class MyExtensions
{
    public static T Middle<T>(this List<T> list)
    {
        if (list.Count == 0) return default(T);
        int idx = list.Count / 2;
        return list[idx];
    }
}
```

Die Verwendung würde sich dann so ändern:

```
int middleInt = Middle(integers); //gives 143  
middleInt = integers.Middle();
```

## 3 Delegate

Mit einem Delegate (=Stellvertreter) kann man **Funktionen in einer Variablen** speichern. Mit den eng damit verbundenen Events implementiert man in C# das Observer-Pattern.

### 3.1 Motivation

Angenommen man hat eine Klasse `Logger`, die Messages auf die Konsole oder als `MessageBox` ausgeben kann.

```
public class Logger
{
    public void LogToConsole(string msg)
    {
        Debug.WriteLine($"{DateTime.Now:HH:mm:ss.fff} --> {msg}");
    }

    public void LogAsMessageBox(string msg)
    {
        MessageBox.Show(msg, $"{DateTime.Now:HH:mm:ss.fff}");
    }
}
```

Man hat also zwei Funktionen mit derselben Signatur.

Es wäre jetzt wünschenswert, dass man im Konstruktor festlegen kann, wie man Messages ausgeben kann. Mit anderen Worten: man möchte die Funktion auf eine Variable zuweisen und beim Loggen dann diese Variable benutzen.

### 3.2 Definition

Mit einem Delegate definiert man eine bestimmte Art von Funktionen, d.h. man gibt eine bestimmte Signatur an. Die Schreibweise ist dabei wie bei einer abstrakten Funktion, wobei man aber das Schlüsselwort **delegate** angibt und damit festlegt, dass man über den Namen einen bestimmten Funktionstyp definiert.

In unserem Beispiel also:

```
public delegate void LogDelegate(string msg);
```

Man hat mit obigem Code einen neuen Typ definiert, der beliebige Funktionen repräsentiert, die **void** retournieren und einen Parameter vom Typ **string** haben.

### 3.3 Zuweisung

Jetzt kann man mit diesem neuen Typ **LogDelegate** eine Variable definieren:

```
private readonly LogDelegate log;
```

Auf diese Variable kann man jetzt die eingangs definierten Funktionen zuweisen, da sie ja beide genau der damit festgelegten Signatur entsprechen.

```
public Logger()
{
    log = LogToConsole;
}
```

Achtung: Man darf beim Funktionsnamen keine Klammern schreiben, man will ja den Namen der Funktion zuweisen.

### 3.4 Aufruf

Über diese Variable kann man dann die Funktion „stellvertretend“ für die tatsächliche Funktion aufrufen:



```
public void Log(string msg)
{
    log(msg);
}
```

Man beachte noch einmal, dass das nur geht, weil die Signaturen übereinstimmen:

```
public delegate void LogDelegate(string msg);
```

```
public void LogAsMessageBox(string msg)
{
    MessageBox.Show(msg, $"{DateTime.Now:HH:mm:ss.fff}");
}
```

### 3.5 Delegate als Parameter

Interessant wird dieser Mechanismus vor allem dann, wenn eine Klasse eine Methode einer anderen Klasse aufrufen soll, ohne diese Klasse tatsächlich zu kennen.

Aufgabenstellung also: Obige Klasse Logger soll eine Funktion in MainWindow aufrufen.

#### 3.5.1 Objekt übergeben (falsch!)

Prinzipiell könnte man auch einer Funktion einer anderen Klasse das gesamte Objekt übergeben.

```
private readonly MainWindow window;
public Logger(MainWindow window)
{
    this.window = window;
}
```

Und beim Aufruf dann das Window verwenden:

```
public void Log(string msg)
{
    //log(msg);
    window.ShowMsgInList(msg);
}
```

Im MainWindow würde es dann etwa so aussehen:

```
Logger logger;
public MainWindow() => InitializeComponent();

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    logger = new Logger(this);
}
```

```
public void ShowMsgInList(string msg) => lstMessages.Items.Add($"{DateTime.Now:HH:mm:ss.fff} --> {msg}");
```

Dadurch hätte man eine viel zu enge Kopplung zw. der Daten-Klasse und der Window-Klasse. D.h. man könnte diese Klasse nie ohne vorhandene MainWindow-Klasse benutzen;

Richtig klar wird das dann, wenn die Daten-Klasse in einer eigenen Library liegt – **zyklische Abhängigkeit** beim Kompilieren!

#### 3.5.2 Delegate übergeben (richtig!)

Da die Daten-Klasse von der MainWindow-Klasse nichts wissen muss außer der Methode, die aufgerufen werden soll, übergibt man stattdessen nur diese Funktion als Parameter.

Zur Wiederholung: Ein Delegate legt außer der Signatur absolut nichts fest!

```
public Logger(LogDelegate log)
{
    this.log = log;
}
```

```
public void Log(string msg)
{
    log(msg);
    // window.ShowMsgInList(msg);
}
```

Hier wird nicht mehr die gesamte MainWindow-Klasse übergeben, sondern nur jener Teil, der wichtig ist, also die Funktion, die den Callback behandeln soll:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    //logger = new Logger(this);
    logger = new Logger(ShowMsgInList);
}
```

Jetzt kann man die Daten-Klasse ohne Probleme in eine eigene Library geben, weil diese nicht von der Window-Klasse abhängt.

### 3.6 Multicast Delegate

Delegates sind automatisch sogenannte Multicast Delegates, d.h. man kann mehrere Funktionen auf die entsprechende Variable mit `+=` zuweisen. Folgendes würde also funktionieren:

```
public Logger(LogDelegate log)
{
    this.log += log;
    this.log += LogToConsole;
}
```

Es werden dann beide Funktionen aufgerufen, wenn man das Delegate aufruft.

## 4 Lambda Expressions

Eine Lambda Expression ist im Wesentlichen nur eine **sehr kurze Schreibweise einer anonymen Methode**, die für ein Delegate verwendet wird.

Gehen wir von folgendem Delegate aus:

```
private delegate string NameGetter(Person p);
```

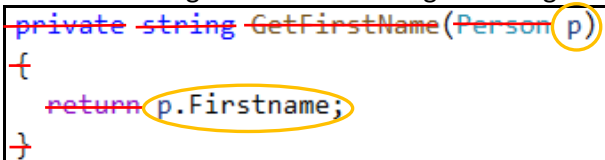
Man lässt einfach allen nicht benötigten Ballast weg und schreibt nur das absolut Notwendige hin.

```
private string GetFirstName(Person p)
{
    return p.Firstname;
}
```

Wesentlich bei obiger Methode ist nur, dass als Parameter eine Variable vom Typ Person benötigt wird und die Property FirstName dieser Person zurückgegeben wird.

Nicht wesentlich ist:

- **Name der Methode:** man will die Funktionalität benutzen – unter welchem Namen das geschieht, ist meist irrelevant
- **Name des Parameters:** es ist völlig egal, unter welchem Variablennamen die Parameter innerhalb der Methode angesprochen werden
- **Typ des Parameters:** ergibt sich aus dem Delegate
- **Rückgabewert:** ergibt sich automatisch aus dem Ausdruck, der bei return steht, bzw. aus dem Delegate
- alle möglichen runden und geschwungenen **Klammern**



```
private string GetFirstName(Person p)
{
    return p.Firstname;
}
```

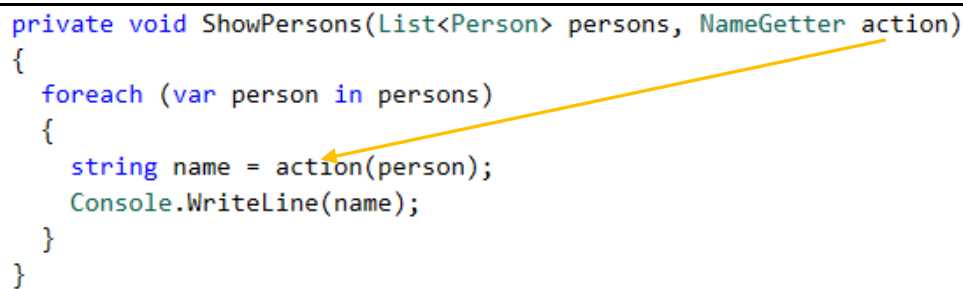
Daraus ergibt sich folgende Lambda Expression: **p => p.Firstname**

“Aus dem Argument x wird FirstName dieser Variable zurückgegeben”

Ein Doppelpfeil => markiert eine Lambda Expression. Der Typ des Ausdrucks hinter dem Doppelpfeil bestimmt den Rückgabotyp der Lambda Expression.

### 4.1 Lambda als Parameter

Interessant werden Lambda Expressions eben beim Einsatz von Delegates (die Klasse Person ist obigem Beispiel entnommen):



```
private void ShowPersons(List<Person> persons, NameGetter action)
{
    foreach (var person in persons)
    {
        string name = action(person);
        Console.WriteLine(name);
    }
}
```

Wenn jetzt persons eine List<Person> ist:

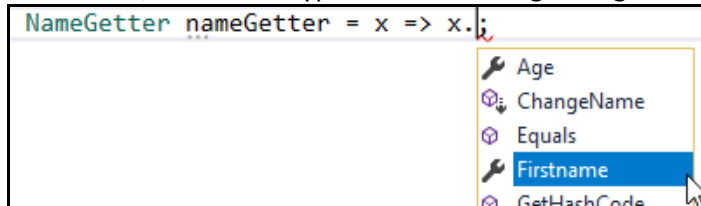
```
var persons = new List<Person>
{
    new Person { Firstname = "Susi", Lastname = "Berger", Age = 77 },
    new Person { Firstname = "Hansi", Lastname = "Huber", Age = 66 }
};
```

Sind z.B. folgende Aufrufe möglich:

```
NameGetter nameGetter = x => x.Firstname;
ShowPersons(persons, GetFirstName);
ShowPersons(persons, nameGetter);
ShowPersons(persons, abc => abc.Lastname);
ShowPersons(persons, any => $"{any.Firstname} {any.Lastname} [{any.Age}]");
```

Hinweise:

- Der Name der Variable **x** oder **abc** ist willkürlich und ohne Bedeutung. Aufgrund der Kürze des Lambdalausdrucks empfehle ich kurze Namen. Ich verwende praktisch immer **x**.
- Üblicherweise weist man den Lambda-Ausdruck nicht auf eine Delegate-Variable zu, sondern gibt ihn beim Aufruf direkt an. Dadurch entsteht extrem kurzer und lesbarer Code
- Wie erwähnt, ist der Name der Variable im Lambda-Ausdruck egal, trotzdem funktioniert auch hier Intellisense, da sich die Typen aus dem Delegate ergeben:



## 4.2 Mehrere Parameter

Verlangt ein delegate mehrere Parameter, müssen diese bei Verwendung der Lambda Expression **geklammert** werden:

```
private delegate int Calc( int x, int y );
Calc fAdd = ( a, b ) => a + b;
```

## 4.3 Ohne Parameter

Dasselbe gilt, wenn die Methode keine Parameter hat:

```
private delegate string DoIt();
DoIt f = () => "abc";
string s = f();
```

Für Javascript-Freunde ist das folgendem sehr ähnlich:

```
window.addEventListener('load', function () { alert('Hallo'); });
```

bzw. (ab Javascript 2015):

```
window.addEventListener('load', () => alert('Hallo'));
```

## 4.4 Mehrere Anweisungen

Enthält die rechte Seite mehrere Anweisungen, müssen diese in geschweiften Klammern stehen:

```
public delegate int Calc(int n);
```

```
Calc f = n =>
{
    int res = 1;
    for (int i = 1; i < n; i++)
    {
        res *= i;
    }
    return res;
};
int x = f(6);
```

Dabei stellt sich dann aber die Frage der Lesbarkeit.

## 4.5 Returntyp void

Dasselbe gilt, wenn der Returntyp void ist:

```
public delegate void Proc(int n);
```

```
Proc f2 = val => { Console.WriteLine("Val = " + val); };  
f2(666);
```

## 4.6 Lokale Funktionen (ab C# 7)

Für Funktionen, die man nur lokal innerhalb einer Funktion benutzt, kann man seit C# 7 auf die direkte Angabe des Delegate-Typs verzichten und die Funktion direkt notieren – als „lokale Funktion“:

```
NameGetter nameGetter = x => x.Firstname;
```

Als lokale Funktion:

```
string nameGetter(Person x) => x.Firstname;
```

## 5 Generische Delegates in C#

C# stellt einige vorgefertigte Delegates zur Verfügung. Damit kann man sich praktisch immer die Definition eines eigenen delegate-Typs ersparen.

### 5.1 Action<T1, T2, ..., T16>

Funktion **ohne return**-Wert und mit einem **bis zu 16 Parametern**.

Beispiel:

```
public void DoSomethingInLoop(Action<int> action)
{
    for (int i = 0; i < 10; i++)
    {
        action(i);
    }
}
```

```
DoSomethingInLoop(x => Console.WriteLine($"Hallo {x}"));
```

Oder:

```
public void DoSomethingInLoop2(int nr, Action<int> action)
{
    for (int i = 0; i < nr; i++)
    {
        action(i);
    }
}
```

```
DoSomethingInLoop2(5, x => Console.WriteLine($"Hallo {x}"));
```

#### 5.1.1 Action

Funktion **ohne return**-Wert und **ohne Parameter**.

Beispiel:

```
Action myAction = () => { Console.WriteLine(DateTime.Now.ToShortTimeString()); };
myAction();
```

### 5.2 Func<T1, T2, ..., T16, TResult>

Methode ohne bzw. **bis zu 16 Parametern**, die ein **Ergebnis des Typen TResult** zurückgibt. Der Rückgabebetyp muss dabei als letzter generischer Typ angegeben werden.

Beispiel:

```
private double BodyMassIndex(int weightKg, double heightM)
{
    //Gewicht (kg) / (Größe (m) x Größe (m))
    double bmi = weightKg / (heightM * heightM);
    return bmi;
}
```

```
Func<int, double, double> bmiCalc = BodyMassIndex;
double bmi = bmiCalc(75, 1.8);
```

oder kurz:

```
Func<int, double, double> bmiCalc = (w, h) => w / (h * h);
double bmi = bmiCalc(75, 1.8);
```

### 5.2.1 Predicate<T>

Methode, die ein Objekt vom Typ T erwartet und einen **bool-Wert zurückgibt**.

Entspricht somit `Func<T, bool>`.

Beispiel:

```
private bool IsEven(int val)
{
    return val % 2 == 0;
}
```

```
Predicate<int> fEven = IsEven;
bool isEven = fEven(12);
```

Oder kurz:

```
Predicate<int> fEven2 = x => x % 2 == 0;
bool isEven2 = fEven2(1123);
```

### 5.2.2 EventHandler<T>

Default Event Callback-Handler mit **eigener EventArgs**-Klasse.

`EventHandler<MyEventArgs>` entspricht also:

```
public delegate void EventHandler(object sender, MyEventArgs args)
```

Beispiel:

```
public class MyEventArgs : EventArgs
{
    public int Val { get; set; }
}
```

```
public event EventHandler<MyEventArgs> MyOtherEventHandler;
```

```
if ( MyOtherEventHandler!=null) MyOtherEventHandler(this, new MyEventArgs{Val=666});
```

## 6 Anonyme Typen

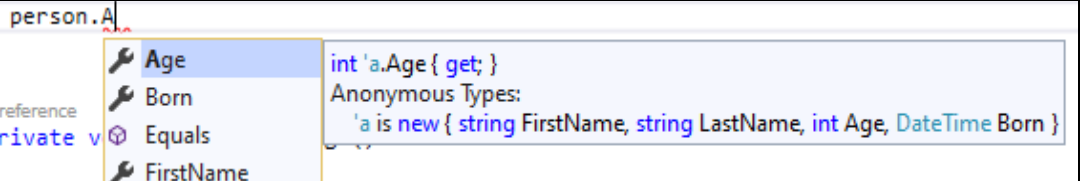
Anonyme Typen sind vom Compiler automatisch erzeugte einfache **namenlose Klassen**, die nur über get-Properties (aber keine set-Properties) verfügen.

Da Anonyme Typen erst vom Compiler erzeugt werden, kann man sie ausschließlich auf Variablen zuweisen, die mit dem Schlüsselwort **var** deklariert werden. Auch hier wird Intellisense wieder vollständig unterstützt.

Instanzen von Anonymen Typen können nur **lokal** verwendet werden – man kann sie also **nicht als Parameter** an eine Methode übergeben!

Beispiel:

```
var person = new
{
    FirstName = "Hansi",
    LastName = "Huber",
    Age = 66,
    Born = DateTime.Now
};
MessageBox.Show($"{person.FirstName} {person.LastName}: {person.Born:dd.MM.yyyy}");
```



Der tatsächliche Typ wird vom Compiler erzeugt.

```
Immediate Window
person.GetType().ToString()
"<>f__AnonymousType1'4[System.String,System.String,System.Int32,System.DateTime]"
```

Jedoch selbst bei Angabe dieses Typs ist es nicht möglich, eine Variable dieses Typs an eine Methode zu übergeben!

Obige „Klasse“ würde der folgenden Klasse entsprechen:

```
public class Person
{
    private string firstName;
    private string lastName;
    private DateTime born;
    public string FirstName
    {
        get { return firstName; }
    }
    public string LastName
    {
        get { return lastName; }
    }
    public DateTime Born
    {
        get { return born; }
    }
    public Person( string firstName, string lastName, DateTime born )
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.born = born;
    }
}
```

keine set-Property

Deklariert man eine weitere Anonyme Klasse mit denselben Typen und Namen in derselben Reihenfolge, verwendet der Compiler denselben Anonymen Typ und man kann eine Variable auf die andere zuweisen.

Folgendes funktioniert also:



```
var person = new
{
    FirstName = "Hansi",
    LastName = "Huber",
    Age = 66,
    Born = DateTime.Now
};
var person2 = new
{
    FirstName = "Heinzi",
    LastName = "Prüller",
    Age = 55,
    Born = DateTime.Now
};
person = person2;
```

Weiters gilt:

- Der Compiler erzeugt automatisch eine **ToString()**-Methode:

```
var obj = new { Id = 666, Name="Hansi" };
Console.WriteLine(obj);
```

```
{ Id = 666, Name = Hansi }
```

- Jede Anonyme Klasse ist von **Object** abgeleitet
- Die Properties einer Anonymen Klasse sind **read only** (es gibt also keine set-Property)