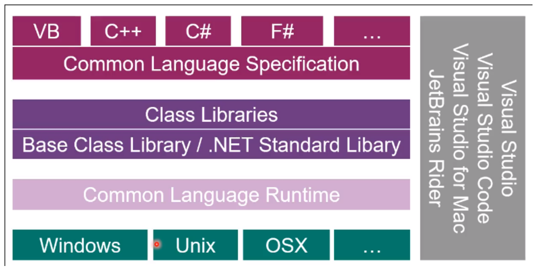
1 .NET

.NET

- .NET Framework: only for Windows, no new Updates (except security patches)
- .NET Core: new runtime, crossplatform
- .NET: successor of .NET Core. This should be used today



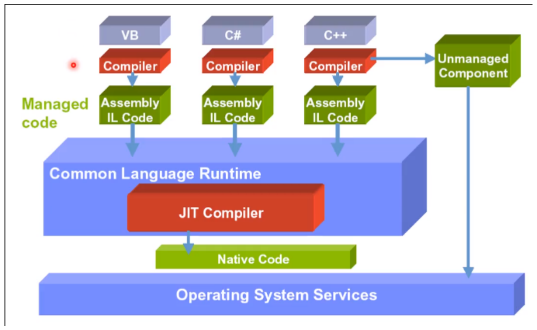
.NET Standard

The .NET Stadard is used to establish a compatibility between different implementation. The standard defines which functions, classes, etc. a implementation has to provided to be conform.

Common Language Runtime (CLR)

The Common Language Runtime (CLR) has:

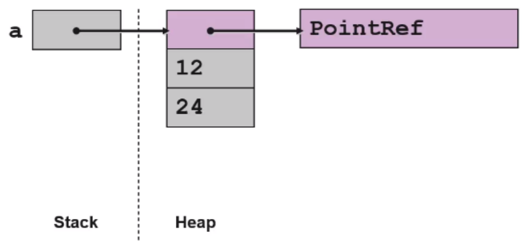
- a JIT Compiler which compiles the Microsoft Intermediate Language to native code
- Garbage Collection
- inter language debugging
- thread management



Common Type System

Common Type System (CTS) is the standard in .NET how a type definitions and specific values are stored in memory. All types in .NET inherit from `System.Object`. Also types like int, long and float inherit from `System.Object`. In CTS exists two different kind of types: Reference types (`class` keyword) and Value types (`struct` keyword). Value types are stored on the stack and are automatically boxed if it is used with something like a list, which stores its element on the heap.

Each type has always a reference to the type description



.NET Assembly

After the compilation you receive an Assembly. This could be an *.exe or *.dll in Windows. This Assembly is dynamically loadable and contains meta data. It something like the JAR file in Java.

What contains a .NET Assembly?

The Assembly contains:

- Manifest (references to other assemblies, metadata, version, author, ...)
- Module (types)
- Resources (images, translations files, ...)

Module metadata:

- public, private, ...
- describes all aspect of the code except programming logic
- used to guarantee type safety
- is normally used in IDEs to provide auto complition

Microsoft Intermediate Language (MSIL)

MSIL is similar to assembler but is platform independent. The MSIL is the same for all .NET languages. The benefits of the MSIL are:

- portability
- typesafety

The drawback is that the normal compiled project is not as efficient as native code. But you can compile a .NET project direct in native code.

MSIL in Action

- Design Time (platform independent, development, MSIL)
- Run Time (platform dependent, JIT)

JIT Compilation: compiled method calls an IL function. The runtime detects that this function is not compiled yet and calls the JIT compiler. The JIT compiler translate the IL code in native code and replace the IL code with native code in the memory.

.NET reference types

In .NET exists 4 different kinds of references:

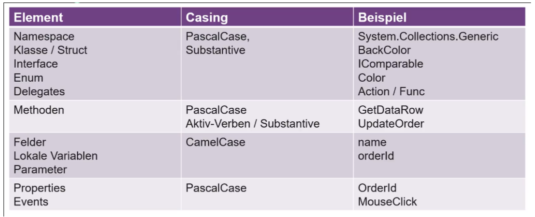
- precompiled assemblies (not possible to debug), `<Reference> ... </Reference>`
- NuGet Package (external dependency, not possible to debug), `<PackageReference> ... </PackageReference>`
- Visual Studio Project (in same solution), `<ProjectReference> ... </ProjectReference>`
- SDK (required, default classes)

C# Project Files

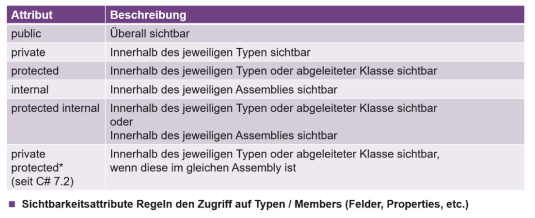
The .NET project are store in a XML file. In a C# project the file is called *.csproj. The most important part in the file are:

- PropertyGroup: Settings
- ItemGroup: item which should be compiled
- TargetGroup: a sequence of step to execute

2 C#



Element	Casing	Beispiel
Namespace Klasse / Struct Interface Enum Delegates	PascalCase, Substantive	System.Collections.Generic BackColor IComparable Color Action / Func
Methoden	PascalCase Aktiv-Verben / Substantive	GetDataRow UpdateOrder
Felder Lokale Variablen Parameter	CamelCase	name orderId
Properties Events	PascalCase	OrderId MouseClick



Attribut	Beschreibung
public	Überall sichtbar
private	Innerhalb des jeweiligen Typen sichtbar
protected	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar
internal	Innerhalb des jeweiligen Assemblies sichtbar
protected internal	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar oder Innerhalb des jeweiligen Assemblies sichtbar
private protected* (seit C# 7.2)	Innerhalb des jeweiligen Typen oder abgeleiteter Klasse sichtbar, wenn diese im gleichen Assembly ist

- **Sichtbarkeitsattribute Regeln den Zugriff auf Typen / Members (Felder, Properties, etc.)**

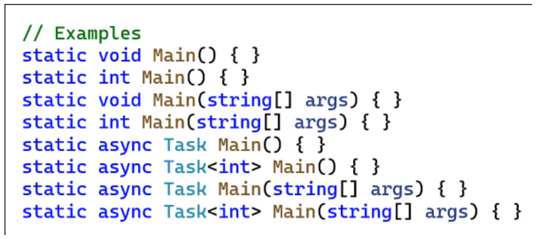
Namespaces in C#

Namespaces are similar to the packages in Java. But in C# the file systems does not have correspond the the namespace. But it is best practice to have the same hierarchy. A namespace can be renamed during the import: `using F = System.Windows.Forms`

Main Method in C#

The main method in C# can exits mulitple times. If this is the case the corret starting method has to written in the *.csproj file:

```
<StartupObject>CSharpGrundlagen_Main01.
    Program1</StartupObject>
```



```
// Examples
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task Main() { }
static async Task<int> Main() { }
static async Task Main(string[] args) { }
static async Task<int> Main(string[] args) { }
```

Arguments can be access with:

```
string[] args;
System.Environment.GetCommandLineArgs();
System.CommandLine; /*NuGet*/
```

The main method is not required. If you leave out the main method (only allowed once) the following schema is required:

- usings
- the code which is normaly in main
- functions / enums / classes / structs ...

The argument array is then always called args.

```
using System;

Volume vLow = Volume.Low;
PrintVolume(vLow);

static void PrintVolume(Volume volume) {
    /* */ }
public enum Volume { /* */ }
```

Enums in C#

Enums in C# are only a list with predefined constants (default Int32). Normally the first value is a 0 and the incremented. It is possible to define the int value explicitly (`enum Days {Sunday = 10, ... }`. After that the value is incremented again. It is possible that the enum has for different options the same int value. This could happens when you set one option explicit higher and one option explicit lower. In this case it is not recommended to convert the int in an enum (it will always take only the first option).
To save memory you can adjust the used type `enum Days : byte { Sunday, Monday ... };`. This is *not in inheritance*. Just a hint for the compiler.

String to Enum

Often you want to convert a String into an Enum (Enums in C#). For this operation you have to options:

- The Option 1 should not be used anymore, because it can throw exceptions
- Option 2 and 3 are identical except that Option 3 has same syntactical sugar.

```
// Option 1
Days day1 = (Days)Enum.Parse(typeof(Days
    ), "Monday");

// Option 2
Days day2;
bool success2 = Enum.TryParse("Monday",
    out day2);

// Option 3
bool success3 = Enum.TryParse("Monday",
    out Days day3); // C# 7.0
```

Print all Options of an Enum

Sometimes you want to iterate over all Values of a enum (Enums in C#).

```
foreach (string day in Enum.GetNames(
    typeof(Days)))
{
    Console.WriteLine(day);
}
```

String in C#

The type string is a reference type (class) but behaves like a Value type and are reused internally. Only `string.Copy()` creates a real new copy. Strings are immutable and value comparison with `=` / `!=` and `Equals` are possible (not like in Java). For escaping two methods exist:

- escaping with a backslash (\): `"C:\\"`
- or with a at @: `@"C:\"`

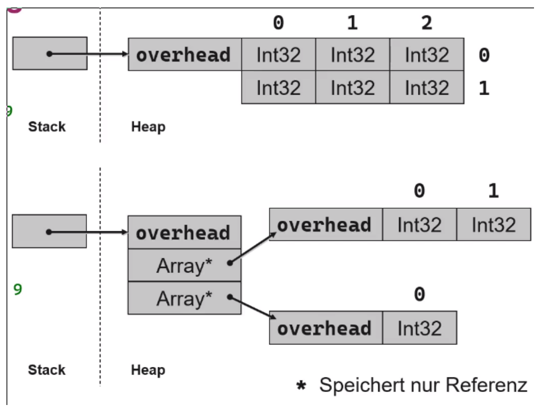
Strings should not be created with `string.Format()` or with the + operator. The better way is string interpolation: `string s3 = $"{DateTime.Now}: {"Hello"}";`

Arrays in C#

In C# the array is a reference type is therefore stored on the heap. It exists three different kind of arrays:

- Single Dimensional Arrays in C#
- Multidimension Array in C#
- Jagged Arrays in C#

If the array stores reference types then only the reference is stored in the array. If the array should store value types then these elements are automatically boxed (moved to the heap) and stored as a whole in the array.



Single Dimensional Array

An plan old array. Nothing special.

```
int[] array1 = new int[5];
int[] array2 = new int[] { 1, 4, 6};
int[] array3 = int[] {1,5,7};
int[] array4 = {1,3,5,5};
object[] array5 = new object[5];
```

Multidimensional Array

Multidimensional Arrays are also called Block Matrices because they look like a rectangle.

```
int[,] multiDim1 = new int[2,3];
int[,] multiDim2 = new { {1,2,3},
    {4,5,6}};

int[,] array = new int[3,2];
int length = array.Length; // 6
int length0 = array.GetLength(0); // 3
int length1 = array.GetLength(1); // 2
```

The benefits over Jagged Arrays in C# they are:

- more memory efficient
- faster allocation
- faster Garbage Collection

But the access to the elements are slower than in a Jagged Arrays in C#. This is because the Boundary Check is in a One Dimensional Array is optimized. This does not apply for Block Matrices.

Jagged Arrays

Jagged Arrays are Arrays of Arrays. From the first array, each index points to an independend Array. They are called jagged (de: zerklüftet) because each array from the second array can have diffrent sizes.

```
int[][] jaggedArray = new int[6][];
jaggedArray[0] = new int[] {1,2,3,4};

int[][] array1 new int[2][];
array1[0] = new int[3];
array1[1] = new int[1];
int length = array1.Length; // 2
int length0 = array1[0].Length; // 3
int length1 = array1[1].Length // 1
```

The access to the elements is in Jagged Arrays faster than in Block Matrices because the Boundary Check is for One Dimensional Array optimized.

Structs in C#

In C# `struct` are value types and therefore live on the stack. Derivation of a struct is not possible but a struct can implement interfaces. Different as by classes you can not initialize the values directly:

```
struct Point {
    int x = 0; // Compilerrror
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Structs should only used in the following cases:

- to group primitives as one data type (like a Point)
- the new type should be immutable
- is not often boxed
- short life span
- or is embedded in other objects

readonly fields

In C# besides `const` there is also a `readonly`. The value for a `readonly` fields does not have to be known at compile time. The value can be calculated during deklaration or inside the constructor.

nested types in

The outer class has access to the public functions / fields / properties of the class. But the inner class has access to everything of the outer class. Foreign classes can access only on public functions / fields / properties and only if the class itself is public.

Static Usings

In C# you can import static classes / enums with static usings:

```
using static System.Console;

WriteLine("Hello World");
```

If a naming clash occurs the normal overloading rules apply. With the class name you can ensure the correct function call.

Params

In C# exists two kind of functions calls:

- call by value
- call by reference

For call by reference exists two keywords with different purpose:

- `ref` (normal call be reference)
- `out`

For overloading `ref` / `out` are distinguished feature.

out Parameters

A function which takes `out` parameters initialized this arguments during the function call. This technique is used by the `TryParse` methods. If you are not interested in one of the parameters then use the _ to discard the "return value".

```
void Init(out int a, out int b) { a = 1;
    b = 2; }
void TestInit() { Init(out int a1, out _
    ); }
```

params array

The params array allows the caller to add any number of arguments at the end:

```
void DoSomething(string str, params
    string[] list) { /**/ }
DoSomething("{0} some string {1}", 2, 3)
    ;
```

It has to be the last parameter in the function. During compilation time the parameters are transformed in an normal array. It is not possible to use the parms array with the `out` / `ref` keyword.

Imporent: the following two functions are the same for the compiler (no valid overloading):

```
void DoSomething(string str, params
    string[] list) { /**/ }
void DoSomething(string str, string[]
    list) { /**/ }
```

Optional Parameters

In C# exist optional parameters. This is realized that some parameters have a default value:

```
void optionalParameters(int i, bool flag
    = true) { /**/ }
```

The default value has be calculated during compile time. Leaving out an optional parameter is only possible at the end. If you want to specify the last option then you have set all previous flags too.

Important: Default parameters are no distinguished feature for overloading. The following are the same function for the compiler (compiler error):

```
void optionalParameters(int i, bool flag
    = true) { /**/ }
void optionalParameters(int i, bool flag
    ) { /**/ }
```

Named Parameters

The problem with Optional Parameters in C# is that you have sometimes to specify every option even if you want only to change the default value of the last argument. With named parameters this problem does not occur:

```
void PrintOrderDetails(string
    productName, string sellerName, int
    orderNum) { /**/ }
PrintOrderDetails(orderNum: 31,
    productName: "Red␣Mug", sellerName: "
    Gift␣Shop");
```

Properties

Properties are compiler feature which implements the Getter and Setter methods. In the Set part you can access the assigned value using the `value` keyword.

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

Auto implemented properties use compiler also a compiler feature. To avoid naming conflicts the compiler creates a "unspeakable variable name". This is a variable name which the compiler not accepted from the user.
You can even initialized auto implemented properties. The set part does not even to be there.

```
public string FirstName { get; set; } =
    "Jane";
```

Another compiler feature is that you can set the values over the Setters right after the creation using the default compiler:

```
MyClass mc = new MyClass()
{
    Length = 1,
    Width = 2
};
```

```
// compiles to this
MyClass mc = new MyClass();
mc.Length = 1;
mc.Width = 2;
```

Indexers

Indexer are just a special case of Properties in C#. It is basically an overloading of the index operator (`[]`):

```
MyClass mc = new MyClass();
mc[0] = "Hello";
string value1 = mc[0];

class MyClass
{
    private string[] arr = new string
        [10];
    // this zeigt an dass es ein indexer
        ist
    public string this[int index]
    {
        get { return arr[index]; }
        set { arr[index] = value; //
            value ist ein string in
            diesem fall }
    }

}
```

Switch Expressions

The switch expression in C# works like the switch expression in Rust.

```
public static Orientation ToOrientation(
    Direction direction) => direction
    switch
{
    Direction.Up     => Orientation.North
        ,
    Direction.Right => Orientation.East,
    Direction.Down  => Orientation.South
        ,
    Direction.Left  => Orientation.West,
    _ => throw new
        ArgumentOutOfRangeException(
            nameof(direction), $"Not␣expected
            ␣direction␣value:␣{direction}"),
};
```

Default in C#

In C# for a Struct in C# the default constructor is always available. For a class when no constructor is implemented by the user or explicitly implemented. In a Struct in C# always all fields have to be initialized.
Using `default(T)` or `default` the memory location is filled with 0. So the get the default value back.

Static Constructors

A static constructor is for structs and classes identically and has never parameters and no visibility. The static constrocutr is used for initial work and is executed exactly once for the whole application during the first creation of an object of the class / struct.

Operator Overloading

The function has to be a `static` and needs the keyword `operator` with the operator afterwards:

```
public static Point operator+(Point lhs,
     Point rhs) {
    return new Point(rhs.X + lhs.X, rhs.
        Y + lhs.Y);
}
```

The return type freely selectable. But min. one parameter has to be from the same type of the class.

Partial Class

A class can be spitted in multiple files. This requires the keyword `partial`. This works with classes, structs and interfaces.

```
// File1.cs
partial class MyClass
```

```
{
    public void Test1() {}
}

// File2.cs
partial class MyClass
{
    public void Test2() {}
}
```

Usage:

- Mostly used with generators:
  - File 1: created by the generator
  - File 2: created by the developer
- Split up a big file (bad code)
  - good starting point for refactoring

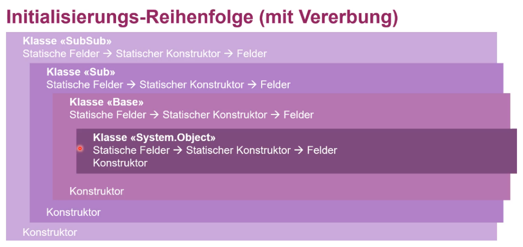
If I define in one place partially this is valid for all other files too.

Partial Method

It is also possible to implement partial methods. This is often used for user defined hooks in generated code. For this the class / struct needs to be also partial and the function has to be private and has to return void.

```
// Definition in file1.cs, e.g generated
    by an generator
partial void OnNameChanged();

// Implementation in file2.cs,
   implemented by an developer
partial void OnNameChanged() { /**/ }
```


Initialisierungs-Reihenfolge (mit Vererbung)
Klasse «SubSub»
Statische Felder → Statischer Konstruktor → Felder
Klasse «Sub»
Statische Felder → Statischer Konstruktor → Felder
Klasse «Base»
Statische Felder → Statischer Konstruktor → Felder
Klasse «System.Object»
Statische Felder → Statischer Konstruktor → Felder
Konstruktor
Konstruktor
Konstruktor
Konstruktor

Type Casting

`null` could be also casted. Except you want to cast it in a Value Type (Struct in C#). This would throw an `InvalidCastException`.

```
SubSub a = new SubSub();
if (a is SubSub) {}
```

`obj as T` short for `obj is T ? (T)obj : (T)null`

```
Base a = new Sub();
Sub b = a as Sub;

/* Same as following*/
Sub b = a is Sub ? (Sub)a : (Sub)null;


Base a = new SubSub();
if (a is SubSub casted) {
    Console.WriteLine(casted);
}

/* same as following*/
SubSub casted = default;
if (a is SubSub) {
    casted = (SubSub) a;
}
```

Override Functions

So that you can override a function it has to be marked as `virtual` in the base class. In the child class you can override it with the keyword `override`. The keyword `virtual` is not possible when:

- function is static
- function is `abstract` (implied virtual)
- private (not even possible to override)
- override (implied virtual from base class)

Dynamic Binding in C#

Rule Set in pseudo code:

```
var st = static type of obj;
var dt = dynamic type of obj;
var m = Method "M" of st; // Standard-
    Methode, existiert zwingend (evtl.
    vererbt)!
var typelist = all types between st (
    exclusive) and dt (inclusive);

foreach (var t in typelist)
{
    // Schlüsselwort "override"
    if (t has an override method "M")
        m = Method "M" of t;
    // Schlüsselwort "new"
    // Oder ohne Angabe
    else if (t has a non-override method
        "M")
        break;
}
call m;
```

Interrupt dynamic binding

If the keyword `override` is missing the original functions are hidden therefore the dynamic binding is interrupted. If you want this you should add the `new` keyword: `public new void I() {}`. This tells the compiler that you really want this and it is not a mistake.

seald keyword

With the keyword `sealed` you prevent that something inherits from this. `sealed` can be used with classes, properties, indexers and events.

`sealed` can improve the performance because the dynamic binding algorithm is not executed. sealed members are not very common and are only possible with the keyword `override`:

- `public override sealed void Add()`

But you can hide the sealed function and create a new hierarchy with the `new` keyword:

- `public new virtual void Add()`

Interfaces in C#
Because C# does not allow multiple inheritance the Interfaces are created. A class can implemented multiple interfaces at the same time. All function defined in the interface must be implemented by the class or by a base class. Implementation of Interface members must be `public` and must not be `static`. Interfaces can inherit from other Interfaces
Naming Clashes using Interfaces
If your class implements two interfaces with the same name and same signature you have 3 possible solutions:

1. implement the method regularly. The implementation counts for both interfaces
   - do this if the logic for both interfaces are the same
2. implement the methods explicit
   - do this if the logic for both interfaces are different
   - `void ISequence.Add(object x) { /* do something */ }`
   - `void IShoppingCart.Add(object x) { /* do something */ }`
3. implement one regularly and one explicit
   - this is useful if the regularly one should be the default

```
class ShoppingCart : ISequence,
    IShoppingCart {
    void ISequence.Add(object x) {}
    void IShoppingCart.Add(object x) {}
}
```

```
ISequence sc1 = new ShoppingCart();
sc1.Add("Hello");

IShoppingCart sc2 = new ShoppingCart();
sc2.Add("Hello");

class ShoppingCart : ISequence,
    IShoppingCart {
    void Add(object x) {} // will be the
        default
    void IShoppingCart.Add(object x) {}
}
```

Delegates
Delegates are type safe function pointers as a object (reference type). Internally it holds references to 0 to n methods.

```
public delegate void Notifier(string
    sender);
public static void SayHi(string sender);


Notifier greetings;
greetings = new Notifier(SayHi);
// kurzform
greetings = SayHi;
greetings("John");
```

Attention: Delegates must not be `null` if you want to invoke it.

```
if (delegateVar != null) {
    delegateVar(params);
}

// neu ab C# 6.0
delegateVar?.Invoke(params);

// Anonymous delegate
delegate bool Predicate(string s);
private static void DoSomething(
    Predicate pred) { /**/ }

DoSomething(delegate(string s){
        return s.StartsWith('S');
    });
```

Multicast Delegates
Technical all delegates are multicast delegates. Normal delegates are just a special case where the delegates hold only one reference to a function.
Delegates are taken by the compiler and generates from it a new class which inherits from the class `MulticastDelegate`. Only the compiler can use the `MulticastDelegate` class.
The compiler also generates from this `greetings("John");` the following snippet:

```
foreach(Delegate dele in greetings.
    GetInvocationList()) {
    ((Notifier)dele).Invoke("John")
}
```

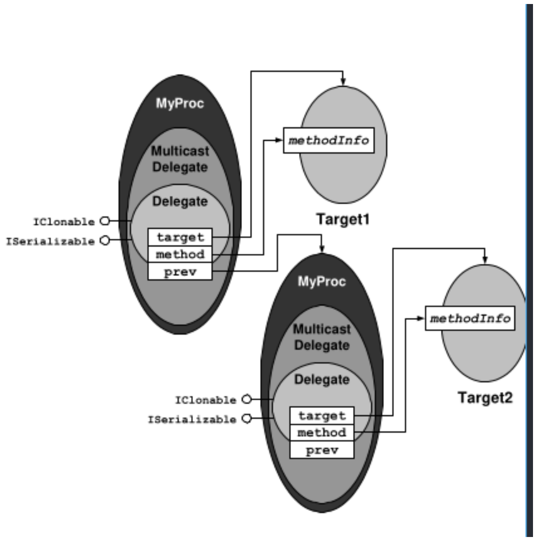
Attention: Do not use return values, out or ref parameters in Multicast Delegates. Only the values form the last function call are returned.
Manipularting Multicast Delegates
To manipulate the delegate three operations exits:

- `=` assigns a function to the delegate
- `+=` adds a function to the delegate
- `-=` removes a function from the delegate

Functions are executed in the same order as you added (`+=`) it to the Multicast Delegate. The newest function witch matches is removed when using `-=`.
Implementation Multicast Delegate
A delegate is implemented as a LinkedList and stores three things:

- target: reference to the target object

  - the instance of the class, if static `null`
- method: the method reference
- perv: the reference to the previous delegate



Predefined Generic Delegates *Action*:

- `Action()`
- `Action<in T>(T obj);`
- `Action<in T1, ..., in T16>(T obj, ..., T16 arg);`

Func:

- `TResult Func<out TResult>()`
- `TResult Func<in T, out TResult>(T arg)`
- `TResult Func<in T, ..., in T16, out TResult>(T arg, ..., T16 arg16)`
- The last type parameter is the result type

Predicate: `delegate bool Predicate<in T>(T obj);`
EventHandler:

- `EventHandler<TEventArgs>(object sender, TEventArgs e);`
- `EventHandler(object sender, EventArgs e);`

Events

Events are just syntactical sugar. The compiler takes the event and generates a private delegate and adds (un)subscribe logic as public (`+=` and `-=`).

```
public delegate void TickEventHandler(
    int ticks, int interval);

public class Clock {
    public event TickEventHandler
        OnTickEvent;

    /* event is transformed to */
    private TickEventHandler OnTickEvent
        ;
    public void add_OnTickEvent(
        TickEventHandler h) { OnTickEvent
         += h; }
    public void remove_OnTickEvent(
        TickEventHandler h) { OnTickEvent
         -= h; }
}
```

The default syntax of the events is the following:

```
public delegate void AnyHandler(object
    sender, AnyEventArgs e);
```

- `object sender`: the sender of the event, the sender hands `this` over at the call of the delegate / event
- `AnyEventArgs e`: an arbitrary sub class ob `EventArgs`. It contains information to the event.

Anonymous Methods

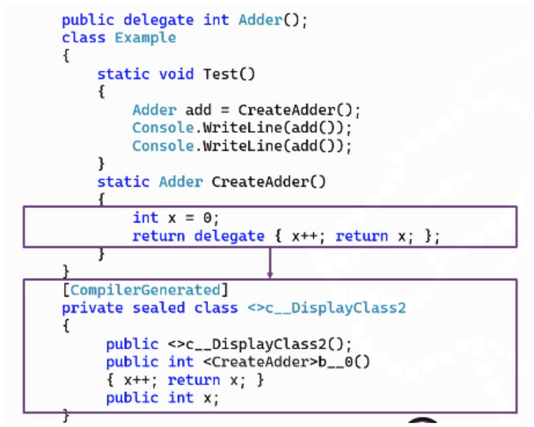
In C# you can create anonymous methods with delegates or with lambdas. Anonymous methods can not use `params[]`, `ref` and not `out` parameters.
Delegates:

```
list.ForEach(delegate(int i) { Console.
    WriteLine(i); });

int sum = 0;
list.ForEach(delegate(int i) { sum += i;
     });
list.ForEach(delegate(int i) { sum += 1;
     });
```

Lambda:

```
List<int> list = new List<int>();
int sum = 0;
list.ForEach(element => sum += element);
```



```
public delegate int Adder();
class Example
{
    static void Test()
    {
        Adder add = CreateAdder();
        Console.WriteLine(add());
        Console.WriteLine(add());
    }
    static Adder CreateAdder()
    {
        int x = 0;
        return delegate { x++; return x; };
    }
}
[CompilerGenerated]
private sealed class <>c__DisplayClass2
{
     public <>c__DisplayClass2();
     public int <CreateAdder>b__0()
     { x++; return x; }
     public int x;
}
```

3 Generic / Nullable

Benefits When you use generics in a .NET project you got the following benefits:

- reusability
- type safety
- performance improvements

The performance improvements is only available for value types. The improvement is obtained by the fact that this types does not need to be boxed / unboxed. With `object` you would always have to do a boxing / unboxing.

Types of Constraints
For the generic type T you have 6 possible constraints:

- struct - T must be a struct
- class - T must be a class
- new() - T must have a default constructor
- ClassName - T must inherit from ClassName
- InterfaceName - T must implement InterfaceName
- TBase - T must be equal to TBase or T has to inherit from TBase

```
class Something<T, K>
    where T : struct, class, new()
    where K : ISequence
{
    /* fields, methods, properties,
        ...*/
}
```

Generic Type Inference
If the type parameters is redundant it can be omitted. This is the case if `T` is formal parameter. If `T` is only the return value or not in the signature the type has to be written.

```
public void Print<T>(T t) { /**/ }
public T Get<T>() { /**/ }

Print<int>(12);
Print(12);

int i1 = Get<int>();
int i2 = Get(); // Compiler error
```

How do Generics Work
The value types are treated different than the reference types. When the runtime sees the first occurrence of a struct as a generic it generates a new concrete class where `T` is replaced with the struct. By value types during the runtime the `T=is replaced with =object` (same as in Java).

Whate are Nullable Values
Nullable value type is a normal struct which is implemented as following:

```
public struct Nullable<T>
    where T : struct {
        public Nullable(T value);
        public bool HasValue { get; }
        public T Value { get; }
    }
```

The compiler adds some syntactical sugar for easier usage:

```
int? x = 123;
Nullable<int> x = 123;
int? x = null;
```

For secure reading the following methods should be used:

- `GetValueOrDefault()`
- `GetValueOrDefault(<default>)`

Nullable Type Operators

```
int i = GetNullableInt() ?? -1;

/* same as */
int? iTemp = GetNullableInt();
int i;
if (!iTemp.HasValue) { i = -1; }
else { i = iTemp.GetValueOrDefault(); }

int? i = null;
i ??= 1234;

/* same as*/
int? i = null;
i = i ?? 1234;

int? i = null;
int iTemp = i.GetValueOrDefault();
if (!i.HasValue) { iTemp = 1234; }
i = iTemp;
```

The `?.` (Null-conditional operator) is used for secure method chaining:

```
string s = GetNullableInt()?.ToString();
```

Often also used with delegates Delegates in C#: `a?.Invoke()`

4 Exception

Exceptions
All Exceptions in C# inhert from `System.Exception` and are unchecked exceptions (diffrent in Java).
In C# you throw an exception with the `throw` keyword.

```
throw new Exception("Failure");
```

Rethrow Exception
In C# you can throw an exception in two ways:

1. Cut the Stack Trace

2. Continue the Stack Trace

Cut the Stack Trace:

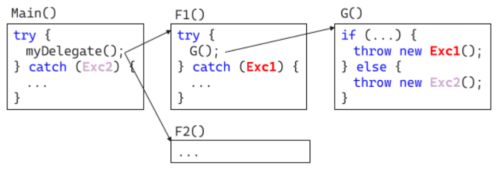
```
try {
    throw new Exception("Failure");
}
catch (Exception e) {
    throw e;
}
```

Continue the Stack Trace (also called *rethrowing*).

```
try{
    throw new Exception("Failure");
}
catch (Exception e) {
    throw;
}
```

Exception Handling with Multicast Delegates

If in a function inside of a Multicast Delegate an exception occurs, the following functions (F2 in 4) are not executed. Even if in the caller method a try-catch block exists.



```
Main()
try {
  myDelegate();
} catch (Exc2) {
  ...
}
```

```
F1()
try {
  G();
} catch (Exc1) {
  ...
}
```

```
G()
if (...) {
  throw new Exc1();
} else {
  throw new Exc2();
}
```

```
F2()
...
```

Exception Filters

In some cases you want that an exception is only catch if a specific condition is fulfilled. This can be achieved with Exception Filters.

```
try
{
    /* ... */
}
catch (Exception e) when (DateTime.Now.
    Hour < 18)
{
    /* ... */
}
catch (Exception e) when (DateTime.Now.
    Hour >= 18)
{
    /* ... */
}
```

Check Arguments

For checking the arguments in C# you have the `ArgumentNullException` and the `ArgumentOutOfRangeExceptioArgumentOutOfR` Use always the `nameof` operator in this case because it is refactoring stable (Not just a hard coded string you give as a parameter to the constructor).

```
string Replicate(string s, int nTimes)
{
    if (s == null)
    {
        throw new ArgumentNullException(
            nameof(s));
    }
    if (s.Length == 0)
    {
        throw new
            ArgumentOutOfRangeException(
            nameof(s));
    }
    if (nTimes <= 1)
    {
        throw new
            ArgumentOutOfRangeException(
            nameof(nTimes));
    }
    return new StringBuilder()
        .Insert(0, s, nTimes)
        .ToString();
}
```

5 Foreach / Iterator / Extension Methods

foreach

While an iterator is active on a collection, it is **not** possible to modify this collection.

foreach compiled output

The compiler converts the following foreach loop in a while loop.

```
foreach (int i in list)
{
    if (i == 3) continue;
    if (i == 5) break;
    Console.WriteLine(i);
}

// Compiler Output
IEnumerator enumerator = list.
    GetEnumerator();
try
{
    while (enumerator.MoveNext())
    {
        int i = (int)enumerator.Current;
        if (i == 3) continue;
        if (i == 5) break;
        Console.WriteLine(i);
    }
}
finally
{
    IDisposable disposable
        = enumerator as IDisposable;
    if (disposable != null)
        disposable.Dispose();
}
```

Make Collection foreach-able

If you want to use your collection in a foreach loop your collection must implemented the `IEnumerable` / `IEnumerable<T>` interface or functions with the same signature and name.

yield keyword
The `yield` keyword is used to create deferred evaluated functions. It exits two versions of `yield`:

- `yield return`: returns the next value
- `yield break`: terminated the iteration

The `yield` keyword tells the compiler that it should generate a state machine. After each call of the method a other (the next) element is returned until know elements are remaining in the colleciton.

Iterator Methods
Iterator methods have always the same signature and min. one `yield` keyword:

- `public IEnumerator <Name>()`
- `public IEnumerator<T> <Name>()`

A iterator method returns always an Iterator. The call of `GetEnumerator` does not execute the function. Only the call of `MoveNext()` on the iterator. This is called Deffered Evaluation.
The default iterator method is `GetEnumertor` but you can create custom functions or properties:

```
public IEnumerator<int> GetEnumerator()
    {
    for (int i = 0; i < data.Length; i
        ++)
        yield return data[i];
}
// Spezifische Iterator-Methode
public IEnumerable<int> Range(int from,
    int to) {
    for (int i = from; i < to; i++)
        yield return data[i];
}
// Spezifisches Iterator-Property
public IEnumerable<int> Reverse {
    get {
        for (int i = data.Length - 1; i
            >= 0; i--)
            yield return data[i];
    }
}
```

This methods can be used inside a foreach loop.

```
foreach (int elem in list.Range(2, 7)) {
     /**/ }
```

Extension Method
Extension Methods are a compiler feature of C# which allows you to extend a class from the user view. In reality the class is not extend. It is *not* possible to access private members of the extended class. If a conflict occurrs always the method from the class itself is used.

How to create extension Methods
Extension Methods have the following requirements:

- must be inside a static class
- the function itself must be static
- the first parameter must be `this`
    - this describes on which classes the function can be executed on

```
public static class ExtensionMethods
{
    static string ToStringSafe(this
        object obj)
    {
        return obj == null ? string.
            Empty : obj.ToString();
    }

    public static void Test()
    {
        1.ToStringSafe();
        /* turns into */
        ExtensionMethods.ToStringSafe(1)
            ;
    }
}
```

6 Lambda / Query Expression

Lambda Expression
In C# exits two kind of Lambdas:

- *Expression Lambdas*: `(<parameters>) => expression`
- *Statement Lambdas*: `(<parameters>) => { staments; }`

Lambdas should not be longer than 2 - 3 statements. If you need more statements is worthy to think about to write a complete function.

Expression-Bodied Members
Expression-Bodied Members are a special kind of Lambdas. This lambdas contain only one statement can be used instead of the `{}` block.

Object / Collection Initializers
Sometimes you have to create a object hierarchy. Normally you have to create each object first and then set the properties. Using the Object / Collection Initializers it is possible to write this in one step. Then the compiler generate from your code what you normally have to do by your self.

```
Student s1 = new Student();
s1.Name = "Hans␣Muster";
s1.Id = 1;

Student s2 = new Student();
s2.Name = "Joe␣Doe";
```

```
s2.Id = 2;

Course c = new Course();
c.Name = "MsTe";
c.Students = new List();
c.Students.Add(s1);
c.Students.Add(s2);
```

Object / Collection Initializers

```
Course c = new Course {
    Name = "MsTe",
    Students = new List {
        new Student {
            Name = "Hans␣Muster",
            Id = 1
        },
        new Student {
            Name = "Joe␣Doe",
            Id = 2
        },
    }
};
```

Anonymous Types

Anonymous Types are used to store an intermediate result. This is often used in LINQ queries. To use anonymous types you must use the `var` keyword. All properties are `readonly`. The functions `Equals()`, `GetHashCode()` and `ToString()` are overwritten.

```
var a = new { Id = 1, Name = "John" };
var b = new { a.Id, a.Name };
var studentList = new List<Student>();
var q = studentList
    .GroupBy(s => s.Subject)
    .Select(
        grp => new {
            Subject = grp.Key,
            Count = grp.Count() });
```

The compiler detects that `a` and `b` are from the same type.

7 LINQ

What is LINQ

LINQ is short for *Language Integrated Query* and is a compiler feature which uses Extensions Methods extensively. Most Query Operators are implemented als deffered functions. A few are implemented as immediate. This are normally function which do not return an `IEnumerable` like `ToList` or `Count`.

```
string[] cities = { "Bern", "Basel" };

var q1 = from c in cities select c;
/* is turn into */
var l1 = cities.Select(c => c);

var q2 = from c in cities where c.
   StartsWith("B") orderby c select c;
/* is turn into */
var l2 = cities.Where(c => c.StartsWith(
   "B")).OrderBy(c => c);
```

LINQ Query Syntax

The Syntax components from LINQ.

keyword	description
from	defines range variable and data sour
where	filter
orderby	sort
select	projection on a element type
group	group into a sequence of group elem
join	link two data sources
let	define a support variable

```
var q1 = from s in Students
    where s.Subject == "Computing"
    orderby s.Name
    select new {s.Id, s.Name };
```

Group with LINQ

```
var q = from s in Students
    group s.name by s.Subject;

foreach (var group in q)
{
    Console.WriteLine(group.Key);
    foreach (var name in group)
    {
        Console.WriteLine("␣" + name);
    }
}
```

```
Computing
  John
  Sue
Mathematics
  Ann
  Bob
```

Group Into with LINQ

```
var q = from s in Students
    group s.name by s.Subject into g
    select new
    {
        Field = g.Key(),
        N = g.Count()
    };

foreach (var x in q)
{
    Console.WriteLine(x.Field + ":" + x.
        N);
}
```

```
Computing: 2
Mathematics: 2
```

Inner Join with LINQ

Explizit:

```
var q = from s in Students
    join m in Markings
    on s.Id equals m.StudentId
    select s.Name + ",␣" + m.Course + ",
        ␣" + m.Mark;
```

Implizit: (builds Cross product and filter it with `where`) - do not use it when possible (slow)

```
var q =
    from s in Students
    from m in Markings
    where s.Id == m.StudentId
    select s.Name + ",␣" + m.Course + ",
        ␣" + m.Mark;
```

```
John Doe, Programming, 3
John Doe, Database, 2
Linda Miller: Organic Chemistry, 1
```

Group Joins with LINQ

LINQ groups by the part in the `from` part (`s` in the example). Therefore, everything which belongs `s` is put in a list (`into list`).
In this example for each student a list is created with all markings belonging to it.

```
var q =
    from s in Students
    join m in Markings
    on s.Id equals m.StudentId
    into list
    select new
    {
        Name = s.Name,
        Marks = list
    };

foreach (var group in q)
{
    Console.WriteLine(group.Name);
    foreach(var m in group.Marks)
    {
        Console.WriteLine(m.Course);
    }
}
```

Left Outer Joins with LINQ

```
var q = from s in Students
    join m in Markings
    on s.Id equals m.StudentId
    into list
    from sm in list.DefaultIfEmpty()
    select s.Name + ",␣" + (sm == null ?
        "?" : sm.Course + ",␣" + sm.Mark
        );

foreach (var x in q)
{
    Console.WriteLine(x);
}
```

```
John Doe, Programming, 3
John Doe, Database, 2
Linda Miller, Organic Chemistry, 1
Ann Forster, ?
```

Let with LINQ

`let` allows to create support variables.

```
var result =
    from s in Students
    let year = s.Id / 1000
    where year == 2009
    select s.Name + "␣" + year.ToString
        ();

foreach (string s in result)
{
    Console.WriteLine(s);
}
```

```
John Doe 2009
Linda Miller 2009
```

SelectMany with LINQ

Allows you zu summarize a nested list into one list.

```
var list = new List<List<string>>
{
    new List<string> { "a", "b", "c" },
    new List<string> { "1", "2", "3" },
    new List<string> { "ö", "ä", "ü" },
};

var q1 = list.SelectMany(s => s);
// q2 same as q1
var q2 = from segment in list
    from token in segment
    select token;


foreach (string lie in q1)
{
    Console.WriteLine("{0}.", line);
}
```

```
a.b.c.1.2.3.ö.ä.ü
```

8 Async

Task in .NET

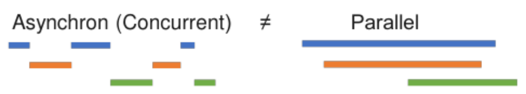
Task are a lightweight variant of a Thread and are normally used in `async` / `await` cases.

Task vs. Thread

A Task has a return value and can run multiple operations in parallel. A Thread has no return value and has only one operation. In general a Task is a more *high level* construct and easier to use. But with a Thread you have more control over the details.

async vs. parallel

Attention: Asynchron (Concurrent) ist not the same as parallel (see 8).



Async / Await operations

To avoid the problems of a blocking task you can wait on the task asynchronously. This can be achieved over callbacks or the functions returns a task object. Normally you can / should implement IO operations as async.

Blocking Task

Attention: synchronous waits on a task are most of the time not desired because they block the current Thread. If it's the UI Thread then the UI is not responding and it looks like the application is not working anymore.
Blocking API:

- `Task.Result`
- `Task.Wait()`
- `Task.WaitAll()`

async in .NET

A async function in .NET can only have the following return types:

- `Task`
- `Task<T>`
- `void` (fire and forget, should only be used in exceptional cases)

await in .NET

The `await` keyword waits on the Task to fulfill. If the task is fulfilled then the execution is continued. An alternative for await are continuations and basically the same.

```
// await syntax
int id = await GetSomeCustomerId();
string t2 = await GetOrders(id);
Console.WriteLine(t2);

// continuations
Task<int> t1 = GetSomeCustomerId();
t1.ContinueWith(id =>
{
    Task<string> t2 = GetOrders(id.
        Result);
    t2.ConinueWith(order =>
                    Console.WriteLine(
                        order.Result)
    );
});
```

9 Reflection

Usage of reflection

The most prominent application for reflection is *Intelli Sense* or the *Object Browser* in Visual Studio. Using reflection you can do the following:

- Type Discovery
- Late Binding (Method / Properties)
- Reflection Emit / Code-Emittierung

System.Type

All types in the CLR are self defined using `System.Type`. The `System.Type` is the entry point for all reflection operations. The `System.Type` class describes itself with an `System.Type` instance. The base class for `System.Type` is the `System.RuntimeType`.

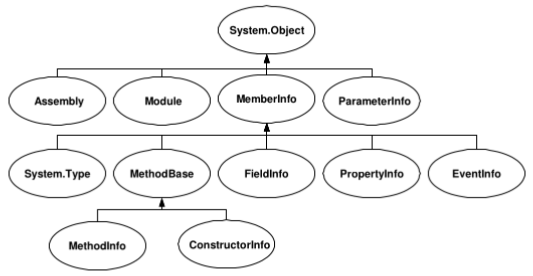
Runtime Prefix

The prefix *Runtime* in .NET means that this class is only instanced only once. For example the `System.RuntimeType`.

Get a System.Type

For every type a `System.Type` can be obtained with `typeof` or `GetType()`.

```
int someInt = 2;
var type = someInt.GetType();
var type2 = typeof(int);
```



Type discovery

```
Assembly a01 = Assembly.Load("mscorlib")
    ;
Type[] t01 = a01.GetTypes();
foreach (Type type in t01)
{
    Console.WriteLine(type);
    MemberInfo[] mInfos = type.
        GetMembers();
    foreach (var mi in mInfos)
    {
        Console.WriteLine(
            "\t{0}\t{1}",
            mi.MemberType,
            mi);
    }
}
```

```
System.Int32
  Method Int32 CompareTo(System.Object)
  Method Int32 CompareTo(Int32)
  Method Boolean Equals(System.Object)
  Method Boolean Equals(Int32)
  Method Int32 GetHashCode()
  Method System.String ToString()
```

Read members

All members

```
Type type = typeof(Counter);
MemberInfo[] miAll = type.GetMembers();
foreach (MemberInfo mi in miAll)
{
    Console.WriteLine("{0} is a {1}",
                      mi, mi.MemberType)
                          ;
}
Console.WriteLine("----------");
PropertyInfo[] piAll = type.
    GetProperties();
foreach (PropertyInfo pi in piAll)
{
    Console.WriteLine("{0} is a {1}",
                      pi, pi.
                          PropertyType);
}
```

```
Int32 get_CountValue() is a Method
Void set_CountValue(Int32) is a Method
Void Increment() is a Method
Void Decrement() is a Method
String ToString() is a Method
Boolean Equals(Object) is a Method
Int32 GetHashCode() is a Method
Type GetType() is a Method
Void .ctor(Int32) is a Constructor
Int32 CountValue is a Property
PropertyChangedEventHandler PropertyChan
  is a Event
----------
Int32 CountValue is a Property
```

Dynamically

```
Type type = typeof(Assembly);
BindingFlags bf =
    BindingFlags.Public |
    BindingFlags.Static |
    BindingFlags.NonPublic |
    BindingFlags.Instance |
    BindingFlags.DeclaredOnly;

MemberInfo[] miFound =
    type.FindMembers(
        MemberTypes.Method,
        bf,
        Type.FilterName,
        "Get*"
    );
```

Field info

The field info class describes a single field of a class. This includes:

- Name
- Type
- Visibility
- and more

It allows you to read and write regardless the visibility of the field:

- `object GetValue(object obj);`
- `void SetValue(object obj, object value)`

But try to prevent to set values over this way. Especially if the field is private / protected.

```
Type type = typeof (Counter);
Counter c = new Counter(1);
// All Fields
FieldInfo[] fiAll = type.GetFields(
    BindingFlags.Instance |
    BindingFlags.NonPublic);
// Specific Field
FieldInfo fi = type.GetField(
    "countValue",
    BindingFlags.Instance |
    BindingFlags.NonPublic);
int val01 = (int) fi.GetValue(c);
c.Increment();
int val02 = (int) fi.GetValue(c);
fi.SetValue(c, -999);
```

Property Info

The property info describes a property of a class. This includes the:

- name
- type
- visibility
- informations to get / set

The property info allows you to read / write the property regardless of the visibility:

- `object GetValue(object obj);`
- `void SetValue(object obj, object value);`

But try to prevent to set values over this way when the property is private / protected.

```
Type type = typeof(Counter);
Counter c = new Counter(1);
// All Properties
PropertyInfo[] piAll = type.
    GetProperties();
// Specific Property
PropertyInfo pi =
    type.GetProperty("CountValue");
int val01 = (int)pi.GetValue(c);
c.Increment();
int val02 = (int)pi.GetValue(c);
if (pi.CanWrite)
{
    pi.SetValue(c, -999);
}
```

Method Info

The method info inherits from `MethodBase` and describes a method of a class. This includes the:

- name

- parameters / return value
- visibility
- and more

Using the method `Invoke()` you can call the method.

```
/* Without parameters */
Type type = typeof(Counter);
Counter c = new Counter(1);
// All Methods
MethodInfo[] miAll = type.GetMethods();
// Specific Method
MethodInfo mi = type.GetMethod("
    Increment");
mi.Invoke(c, null);

/* With parameters */
Type type = typeof(System.Math);
Type[] paramTypes = { typeof(int) };
// Get method info for Cos( )
MethodInfo miAbs = type
     .GetMethod("Abs", paramTypes);
// Fill an array with the actual
    parameters
object[] @params = { -1 };
object returnVal = miAbs.Invoke(type,
                                 @params)
                                      ;
```

Constructor Info

The constructor info inherits from `MethodBase` and describes a constructor of a class. This includes:

- name
- parameters
- visibility

The constructor can be called using the `Invoke()` method.

```
Type type = typeof(Counter);
// All Constructors
ConstructorInfo[] ciAll = type.
    GetConstructors();
// Specific Constructor Overload 01
ConstructorInfo ci01 = type.
    GetConstructor(
    new[] { typeof(int) });
Counter c01 = (Counter)ci01.Invoke(
    new object[] { 12 });
// Specific Constructor Overload 02
ConstructorInfo ci02 = type.
    GetConstructor(
    BindingFlags.Instance|BindingFlags.
        NonPublic,
    null, new Type[0], null);
Counter c02 = (Counter)ci02.Invoke(null)
    ;

/* alternative using activator */
Counter c03 = (Counter)Activator
    .CreateInstance(
        typeof(Counter),
        12 // , "further params", "",
            ...
    );
// Alternative
// -> when Public Default Constructor
    exists
Counter c04 = Activator
    .CreateInstance<Counter>();
```

Attributes in .NET

Attributes are the same thing like annotations in Java. They extend the already existing attributes like public, static, abstract or sealed. All attributes inherit from `System.Attribute`.

```
[DataContract, Serializable]
[Obsolete]
// Etc.
    public class Auto
    {
        [DataMember]
        public string Marke { get; set;
            }
        [DataMember]
        public string Typ { get; set; }
    }
```

in .NET exists two kind of attributes:

- *intrinsic*: in the CLR defined and integrated
- *custom*: self defined attributes

Syntax Attribute Usage

Some attributes can be used multiple times on the same target.

```
// without parameters
[DataContract]
class a{};

// Named parameters
[DataContract(Name = "AutoClass")]
class b{};

// Positional Paramaters
[Obsolete("Alt!", true)]
class c{};

// mixed
[Obsolete("Alt!", IsError = true)]
class d{};
```

Create your own attribute

Your attribute must inherit from `System.Attribute` and must have the suffix `Attribute`. Additionaly it must use the attribute `AttributeUsage`. The `AllowMultiple` is per default set to `false`.

```
[AttributeUsage(
AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
    public class BugfixAttribute :
        Attribute
```

```
{
    public BugfixAttribute(int bugId,
                           string
                               programmer
                               , string
                                date)
    { /* ... */ }
    public int BugId { get; }
    public string Date { get; }
    public string Programmer { get; }
    public string Comment { get; set;
        }
}
```

```
[Bugfix(121, "Manuel␣Bauer", "01/03/2015
    ")]
[Bugfix(
107, "Manuel␣Bauer", "01/04/2015",
Comment = "Some␣major␣changes!␣;-)")]
    public class MyMath{};
```

Query an attribute

```
Type type = typeof(MyMath);
// All Class Attributes
object[] aiAll =
    type.GetCustomAttributes(true); //
        true = use also inherited
        attributes
IEnumerable<Attribute> aiAllTyped =
    type.GetCustomAttributes(
        typeof(BugfixAttribute) /* ,
            true */
    );
// Check Definition
bool aiDef =
    type.IsDefined(typeof(
        BugfixAttribute));
```