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#

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>
```

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 indepentend 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() { /**/ }
```

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;
}
```

Listing 1: type check with implicit type cast

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");
```

Listing 2: Prevent naming clash with option 2

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

Listing 3: Prevent naming clash with option 3

3 End