# C#

# Inhaltsverzeichnis I

# Inhaltsverzeichnis II

# Inhaltsverzeichnis III

# Allgemein

- Objektorientiert
- Garbage Collection
- Unterstützt Parallelisierung
- ; am Ende von fast jedem Befehl

# Programmeinstiegspunkt

- Main Methode

```
static void Main(string[] args)
{
    // Display the number of command line arguments.
    Console.WriteLine(args.Length);
}
```

# Namespace

- Dienen der Organisation. (Vereinigen Klassen etc. zu einer logischen Gruppe)
- Werden durch den . Operator getrennt.
- Durch „using“ besteht keine Notwendigkeit, den Namen des Namespace für jede Klasse anzugeben.

```
using System;                // Bindet Namespace System ein
using System.Diagnostics; // Bindet Unter-Namespace System.Diagnostics ein
```

```
System.Console.WriteLine("Hallo"); // ohne using System

Console.WriteLine("Hallo"); // mit using System
```

# Typen

- Klassentypen (class)
- Schnittstellentypen (interface)
- Arraytypen (array)
- Enumerationstypen (enum)
- …

# Basis Typen / Klassen

object = Basisklasse von der jede Klasse erbt

| | |
|---|---|
| bool = Wahrheitswert | true |
| int = ganze Zahl | 13 |
| float = Gleitkommazahl | 16.2f |
| double = Gleitkommazahl | 12.134d |
| decimal = exakte Kommazahl | 24.321m |
| string = Zeichenkette | "asdf" |
| char = Zeichen | ‚c‘ |

…

# Variablen / Konstanten

- Deklaration:
  - Type Name; — string Zeichenkette;
- Zuweisung:
  - Name = Wert; — Zeichenkette = „asdf“;
- Beides in einem:
  - Type Name = Wert; — char Zeichen = ‚a‘;
- Konstanten
  - Verwende Schlüsselwort const — const int Ganzzahl = 32;
- Anonyme Variable:
  - Type wird implizit angegeben — var Zahl = 32; (Type von Zahl ist implizit int)
  - var Name = Wert;
  - var Name; (inkorrekt)
- Dynamische Variable:
  - Type ist veränderbar — dynamic Irgendwas = 20.234f;
  - dynamic Name; (korrekt) — Irgendwas = „asdf“;

# Arrays

- Deklaration
  - Type[] Name; — int[] Zahlen;
- Zuweisung
  - Name = Wert; — Zahlen = new int[20];
- Beides in einem
  - Type[] Name = new Type[Größe]; — int[] Zahlen = new int[20];
- Zugriff auf ein Element in einem Array
  - Lesen: … Name[Index] … — int b = Zahlen[0];
  - Schreiben: Name[Index] = Wert; — Zahlen[0] = 15;

# Verzweigung

- Wichtigsten Vergleichsoperatoren
  - Nicht !
  - Oder ||
  - Und &&
  - Gleich ==
  - Größer >
  - Kleiner <
- if
  - if (Bedingung) { }

    if (2 > 1) { }
- if-else
  - if (Bedingung) { } else { }

    if (true || false) { } else { }

# Schleifen

- for
  - for (Am Anfang der Schleife; solange diese Bedingung erfüllt ist ; nach jeder Iteration) { }
  - for (int i = 0; i < 20; i++) { }
- foreach
  - foreach (Type Name in Collection) { }
  - foreach (string Person in Personen) { Console.WriteLine(Person); }
- while
  - while (solange diese Bedingung erfüllt ist) { }
- do-while
  - do { … } while (solange diese Bedingung erfüllt ist)
  - Bedingung wird erst nach der Iteration getestet

# Switch/Case

- Falls anhand eines Werts/Types einer Variable unterschiedlich gehandelt werden soll (insbesondere nützlich für viele verschiedene Optionen) => Switch Statment
- Syntax:

```
switch (Variablename)
{
    case Variablenwert:
        break;

    case Variablentyp Variablenname:
        break;

    case Variablenwert when Bedingung:
        break;

    case Variablenwert2:
    case Variablenwert3:
        break;

    case VariablenwertN:
        return …;

    default:
        break;
}
```

```
public static void Main()
{
   int caseSwitch = 1;

   switch (caseSwitch)
   {
       case 1:
           Console.WriteLine("Case 1");
           break;
       case 2:
           Console.WriteLine("Case 2");
           break;
       default:
           Console.WriteLine("Default case");
           break;
   }
}
```

```
private static void ShowCollectionInformation<T>(T coll)
{
    switch (coll)
    {
        case Array arr:
           Console.WriteLine($"An array with {arr.Length} elements.");
           break;
        case IEnumerable<int> ieInt:
           Console.WriteLine($"Average: {ieInt.Average(s => s)}");
           break;
        case IList list:
           Console.WriteLine($"{list.Count} items");
           break;
        case IEnumerable ie:
           string result = "";
           foreach (var e in ie)
              result += $"{e} ";
           Console.WriteLine(result);
           break;
        case object o:
           Console.WriteLine($"A instance of type {o.GetType().Name}");
           break;
        default:
            Console.WriteLine("Null passed to this method.");
            break;
    }
}
```

```
private static void ShowShapeInfo(Shape sh)
{
   switch (sh)
   {
      // Note that this code never evaluates to true.
      case Shape shape when shape == null:
         Console.WriteLine($"An uninitialized shape (shape == null)");
         break;
      case null:
         Console.WriteLine($"An uninitialized shape");
         break;
      case Shape shape when sh.Area == 0:
         Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
         break;
      case Square sq when sh.Area > 0:
         Console.WriteLine("Information about square:");
         Console.WriteLine($"   Length of a side: {sq.Side}");
         Console.WriteLine($"   Area: {sq.Area}");
         break;
      case Rectangle r when r.Length == r.Width && r.Area > 0:
         Console.WriteLine("Information about square rectangle:");
         Console.WriteLine($"   Length of a side: {r.Length}");
         Console.WriteLine($"   Area: {r.Area}");
         break;
      case Rectangle r when sh.Area > 0:
         Console.WriteLine("Information about rectangle:");
         Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
         Console.WriteLine($"   Area: {r.Area}");
         break;
      case Shape shape when sh != null:
         Console.WriteLine($"A {sh.GetType().Name} shape");
         break;
      default:
         Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
         break;
   }
}
```

# Methoden

- Rückgabetype Name (Type Name, Type Name, …) { }
- Rückgabetype == void falls nichts zurückgegeben wird
- return: Schlüsselwort zum Beenden der Methode und zurückgeben eines Werts.
- Beispiel:
  - void Testmethode() { }
  - int Addiere5 (int Zahl) { return Zahl + 5; }

# Methoden Paramter Modifizierer

- Params
  - Mithilfe des Schlüsselworts params kann ein Methodenparameter angegeben werden, der eine variable Anzahl von Argumenten akzeptiert. Der Parametertyp muss ein eindimensionales Array sein.
  - Syntax:
    - Rückgabetyp Methodenname(…, params Type[] ParamsParameterName, … )
    - Methodenname(Type a, Type b, Type c, …)
- Ref
  - Mit ref wird festgelegt, dass dieser Parameter als Verweis übergeben wird. Änderungen an dem Parameter wirken sich also auch außerhalb der Methode aus.
  - Ref kann auch als Rückgabetyp verwendet werden um eine Variable einer Instanz auch außerhalb dieser, modifizieren zu können. Will man diese Referenz speichern muss man die Variable in der man sie speichern will ebenfalls mit dem Ref Schlüsselwort kennzeichnen.
- In
  - Mit in wird festgelegt, dass dieser Parameter als Verweis übergeben wird, jedoch nicht überschrieben werden kann.
- Out
  - Mit out wird festgelegt, dass dieser Parameter als Verweis übergeben wird, dies kann auch ohne vorherige Initialisierung geschehen, was aber vor Rückgabe in der Methode nachgeholt werden muss.
- Syntax in/ref/out:
  - Methodendeklaration: Rückgabetyp Methodenname(…, ref/in/out Type ParamsParameterName, … ) { }
  - Methodenaufruf: Methodenname(…, ref/in/out ÜbergabeParameterName, …);

# Methoden Parameter Modifizierer - Beispiele

Beispiel params:

```
public static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}

public static void UseParams2(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}

static void Main()
{
    // You can send a comma-separated list of arguments of the
    // specified type.
    UseParams(1, 2, 3, 4);
    UseParams2(1, 'a', "test");
```

Beispiel out:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

Beispiel in:

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

Beispiel ref:

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

# Klassen

- Variablen und Methodenansammlung
- Definition:
  - class Name { }  class Person { }
- Objekt/Instanz einer Klasse erzeugen:
  - ClassName Name = new ClassName(..);  Person Nils = new Person(…);
- Konstruktor
  - ClassName(…) { }
- Enthält (Member):
  - Felder
  - Methoden
  - …
- Auf die Member wird mit Name.Member zugegriffen.

# Enum

- logische Auflistung/Vereinigung von verschiedenen Optionen/Zuständen…
- Deklaration:
  - enum Name { opt1, opt2, … }
- Instanziierung:
  - EnumName Name;
- Zuweisung:
  - Name = EnumName.optName;
- Beispiel:
  - enum KartenFarbe { Karo, Herz, Pik, Kreuz }
  - KartenFarbe MeineKartenFarbe = KartenFarbe.Herz;

# Interface

- Schnittstelle stellt sicher das Typ der sie implementiert gewisse Methoden / Eigenschaften (kommt später) enthält
- Beginnen per Konvention mit I
- Deklaration:
  - Interface IName { ReturnType0 MethodName0(Param0, …); … }
- Implementierung:
  - class Classname : InterfaceName { ReturnType0 InterfaceName.MethodName0(Param0, …) { … }; … }
  - Klasse kann mehrere Interfaces implementieren
- Beispiel:
  - class ComparableToString : Icomparable<string> { IComparable.CompareTo(string other) {…} }
- Ab C# 8 folgende sind weitere Member deklarierbar…

# Zugriffmodifizierer

- Public: Auf den Typ oder Member kann von jedem Code in der gleichen Assembly oder einer anderen Assembly, die darauf verweist, zugegriffen werden.
- Protected: Der Zugriff auf den Typ oder Member kann nur über Code in derselben class oder in einer class Instanz erfolgen die von dieser class abgeleitet sind
- Private: Der Zugriff auf den Typ oder Member kann nur über Code innerhalb derselben class oder struct Instanzen folgen
- Zugriffsmodifizierer werden vor das Feld / Methode / Klasse geschrieben

```
public void Hallo() { }   Methode mit Zugriffsmodifizierer

private int Alter;                Feld mit Zugriffsmodifizierer
```

| Member von | Standard-Memberzugriff | Zulässiger deklarierter Zugriffstyp des Members |
| --- | --- | --- |
| `enum` | `public` | Keiner |
| `class` | `private` | `public`<br>`protected`<br>`internal`<br>`private`<br>`protected internal`<br>`private protected` |
| `interface` | `public` | Keiner |
| `struct` | `private` | `public`<br>`internal`<br>`private` |

# Get- und Set-Methoden

- Variable x eines Types ist public => Kann von überall beliebig verändert und ausgelesen werden
- Get- und Set-Methoden verhindern das! Mache Variablen niemals public sondern immer private und verwende sie stattdessen.
  - Get Methode wird zum impliziten lesenden Zugriff auf Variable verwendet.
  - Set Methode wird zum impliziten schreibenden Zugriff auf Variable verwendet.
- Sie tragen zu Kapselung bei
  - Verhindern unbefugten Zugriff (bspw. Get-Methode alleine verhindert, das Variable von außen verändert, trotzdem aber gelesen werden kann)
  - Eine invalide Veränderung einer Variable von außen kann durch eine Set-Methode abgefangen werden.
  - Programmierfehler werden vermieden, wenn nur das veränderbar ist, was auch veränderbar sein soll.

# Eigenschaften

- Eigenschaften ermöglichen vereinfachtes und übersichtlicheres Schreiben von get und set Methoden
- Definition:
  - Lang:
    - private Type privateNam
    - public Type publicName

    {

        get { return privateName; }

        set { privateName = value; }

    }
  - Kurz:
    - public Type Name { get; set; }

# Eigenschaften II

- Zugriff:
  - Lesen:
    - … Instanzname.Eigenschaftsname …
  - Schreiben:
    - Instanzname.Eigenschaftsname = Wert;
- Achtung: Beispiel entspricht nicht dem Normalfalls. Meist ist der Grund das öffentliche Variablen nicht direkt, sondern über get und set Methoden angesprochen werden (Kapselung).

Beispiel: nameof(variable) gibt Name der Variable zurück

```
class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

# Static

- Kennzeichnet Member eines Typ als zum Typ und nicht zur Instanz dazugehörig.
- Bspw. eine Methode Addiere in einer Klasse Mathe
- static void Addiere(int a, int b) { return a + b; }
- Wird mit Mathe.Addiere(a, b) aufgerufen.

# Generik

- Generics ermöglichen das Parametrisieren von Types in einer Klasse / Methode. Innerhalb der Methode / Klasse kann dann beliebig mit diesem variablen Type gearbeitet werden.
- Definition geschieht mit <> direkt nach Methoden-/Klassennamen:
  - … Klassen-/Methodennamen <Typename0, Typename1, …> …
- Mithilfe eines nachgestellten where können für die einzelnen Typen Einschränkungen getroffen werden
  - …<Typename0, Typename1, …> where Typename0 : object
  - (Typename0 muss von object erben)
  - Liste aller Möglichkeiten online verfügbar
- Beispiele:
  - List
    - class List<T> { public void Add(T Listelement) {…}  public T Get(int Index) {…} …}
    - List<string> MeineListe = new List<string>();
    - Dient dazu das Listenelemente nur vom angegebenen Type sein dürfen.
  - Serialisierung
    - class JsonSerializer { public static T Deserialize<T>(string Json) {…} }
    - MyType test = JsonSerializer.Deserialize<MyType>(InstanceOfMyTypeAsJsonString);

# Delegates

- Verweise auf beliebige Anzahl von Methoden
- Deklaration
  - delegate Rückgabetype Name(ParamType0 NameParam0, …);
  - Legt Signatur der zuweisbaren Methoden fest.
- Instanziierung
  - DelegateName HandlerName;
  - DelagateName HandlerName = MethodenName;
  - Auch Handler gennant
- Modifikation (Methoden werden im Normalfall in Reihenfolge des Hinzufügens aufgerufen!)
  - HandlerName += new DelegateName(MethodName); Fügt neuen Verweis auf Methode Methodenname hinzu
  - Oder kurz: HandlerName += MethodName;
  - HandlerName -= MethodName; Entfernt Verweis auf Methode Methodenname
- Ausführung
  - HandlerName(Param0, Param1, …);
- Beispiel:
  - Methodenübergabe:
    - delagte void Finished();
    - Finished FinishedHandler = () => Console.WriteLine(„Hallo“);
    - FinishedHandler();

# Lambdaausdrücke

- Eine Art schnell definierte „anonyme“ Methode
- Definition:
  - (inputParam0, inputParam1, …) => { expression0;  expression1; … }
  - inputParam0 => expression0
- Beispiel:
  - … (x) => { return x * x; };
  - Oder auch … x => x * x;
  - Kombination Lambda, Delegates und Generics:
    - Es gibt vordefinierte Delegates Func<TReturnType, TParam0, TParam1, ..> die Funktionen mit Rückgabetype TReturnType und Parametern akzeptieren. Frage: Wie könnte so ein Delegate aussehen?  TReturnType Func <TReturnType, TParam0, TParam1> (TParam0 a, TParam1 b)
    - Func<int, string> strtoint = new Func<int, string>(StringValue => { return Convert.ToInt32(StringValue); });

# Exceptions und Handling

- Exceptions dienen der Fehlermeldung/Ausnahmezustandsmeldung
- Beispiel
  - int[] a = new int[3];
  - a[3] = 2; <= ArgumentOutOfRangeException wird geworfen
- Alle Exceptions erben von der Basisklasse Exception
- Handling:
  - try{…} catch (ExceptionType Name) { … } finally { … }
  - try{…} leitet mögliche Fehler die in Scope {..} passieren an catch weiter
  - catch (ExceptionType Name) fängt Fehler ab, falls Exception vom Typ ExceptionType oder davon abgeleitet.
  - finally wird immer ausgeführt, egal ob Exception geworfen wird oder nicht.
  - Auch mehrere catch Blöcke sind möglich
  - Exceptions können über das Schlüsselwort throw auch selbst geworfen werden
    - throw Excpetion();
- Beispiel:
  - try { int[] a = new int[3]; a[3] = 2; } catch(ArgumentOutOfRangeException e) { Console.WriteLine(e.Message); } finally { Console.WriteLine(„Das wird immer ausgeführt“); }
  - throw new ArgumentException(„Fehler“);

# Parallele Programmierung C# - Allgemein

- Prozess:
  - Computerprogramm
  - Kann auf einem oder mehreren Prozessorkernen / Prozessoren gleichzeitig laufen
  - Besteht aus einem oder mehreren Threads
  - Von anderen Prozessen abgeschirmt, bekommt eigenen Adressraum, …
- Thread
  - Subprogramm (Einzelner Ausführungsstrang eines Prozesses)
  - Läuft auf einem Prozessorkern / Prozessor
  - Hat eigene Umgebung, aber in geringerem Umfang als Prozess (kein vollständiger Wechsel des Prozesskontexts nötig, trotzdem Stack etc.)
  - Kann mit Threads des gleichen Prozesses kommunizieren
  - Können gestoppt, pausiert, wieder gestartet, abgebrochen werden
  - Es ist einsehbar, sobald ein Thread eine Aufgabe erledigt hat
- Threadpool
  - C# Abstraktionsebene
  - Gruppe von mehreren Threads (Ober-, Untergrenze festlegbar)
  - Arbeit/Aufgaben die erledigt werden sollen, können diesem Threadpool übermittelt werden => werden dann wenn ein Thread nichts zu tun hat abgearbeitet
  - Verhindert das zu viele Threads gestartet werden
  - Nicht einsehbar wann Aufgabe erledigt, bietet keinen Weg Ergebnis zurückzugeben!

# Parallele Programmierung C# - Allgemein II

- Task
  - Es gibt eine zentrale Instanz, den Task Scheduler. Dieser läuft auf dem Threadpool
  - Aufgaben (Tasks) werden an diesen weitergeleitet.
  - Bringt Vorteil der Threads: Es ist einsehbar wann eine Aufgabe erledigt wird und Ergebnisse können eingesehen werden. Tasks können gestoppt, angehalten, … werden.
  - Bringt Vorteil des Threadpools: Geringer Overhead, verhindert das zu viele Threads gleichzeitig gestartet werden.
  - => Praktisch fast immer die beste Wahl

# Parallele Programmierung C# - Task

- Es existiert eine Klasse Task (Namespace: System.Threading.Tasks)
- Instanziierung eines neuen Tasks:
  - Ohne Rückgabe:
    - Task Taskname = new Task (Aufgabe);
  - Mit Rückgabe:
    - Task<ReturnTyp> Taskname = new Task (TaskReturnsReturnType);
  - Dabei ist Aufgabe bzw. TaskReturnsReturnType eine Methode.
- Ausführen:
  - Taskname.Start();
- Kombination Instanziieren + Starten:
  - Task.Run(Aufgabe);
  - Beispiel:
    - Task.Run(Methode);
    - Task.Run(()=> { int a = 1; return a; });
- Aktuellen Status abfragen:
  - Taskname.Status

# Parallele Programmierung C# - Task II

- Synchron warten bis Task erledigt ist:
  - Auf einen:
    - Taskname.Wait();
  - Auf alle:
    - Task.WaitAll(Tasks);
  - Auf einen aus einer Gruppe:
    - Task.WaitAny(Tasks);
  - Tasks ist dabei ein Array das mehrere Instanzen vom Typ Task enthält
- Falls Task lange läuft verwende die „long running“ Option (https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions?view=netcore-3.1 ) so das ein extra Thread für den Task auf dem Threadpool gestartet werden kann

# Asynchrone Programmierung

- Programmierung bei der eine Aufgabe angestoßen und auf diese gewartet wird. In der Zwischenzeit können von diesem Befehl nicht abhängige Komponenten weiterarbeiten. Bspw. wird der Aufruf an die UI zurückgegeben.
- Auf einen Aufgabe(Task) kann per Schlüsselwort await asynchron gewartet werden.
  - await Task.Run(Methode);
- Methoden können über das async Schlüsselwort als asynchron deklariert werden, insofern in ihnen per await auf eine Task gewartet werden soll. Sie sollten dann ein Task<RückgabeTyp> zurückliefen, auf den per await gewartet werden kann. (Achtung: Nur der Teil der Methode auf den per await gewartet wird, wird asynchron ausgeführt)
  - async Task MethodenName(…) { }
  - await MethodenName(…);
- https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/async/