C Sharp

Link: PROGRAMMIERSPRACHEN

Was ist C#?

C# ist eine von Microsoft entwickelt und gepflegte Programmiersprache, die sich durch ihre Vielseitigkeit, Leistungsfähigkeit und Einfachheit auszeichnet.

1. **Typsicherheit und Typisierung**: C# ist eine stark typisierte Sprache, was bedeutet, dass Variablen und Objekte einem bestimmten Datentyp zugeordnet sind. Dies hilft, Fehler frühzeitig zu erkennen und die Codequalität zu verbessern.
2. **Objektorientierung**: C# ist eine objektorientierte Sprache, was bedeutet, dass alles in Form von Objekten modelliert wird. Sie unterstützt Konzepte wie Klassen, Vererbung, Polymorphismus und Abstraktion, was die Strukturierung und Wartung von Code erleichtert.
3. **Plattformunabhängigkeit**: C# wird oft mit der .NET-Plattform in Verbindung gebracht, die eine plattformübergreifende Entwicklung ermöglicht. Mit Werkzeugen wie dem .NET Core Framework können C#-Anwendungen auf verschiedenen Betriebssystemen wie Windows, macOS und Linux ausgeführt werden.
4. **Einfache Syntax**: Die Syntax von C# ist klar und strukturiert, was das Lesen, Schreiben und Warten von Code erleichtert. Sie ist von Sprachen wie C und C++ inspiriert, aber mit modernen Konzepten und Annehmlichkeiten erweitert.
5. **Integrierte Unterstützung für Parallelität**: C# bietet eingebaute Unterstützung für asynchrone und parallele Programmierung, was die Entwicklung von performanten und reaktiven Anwendungen erleichtert.
6. **Umfangreiche Standardbibliothek**: C# wird mit einer umfangreichen Standardbibliothek geliefert, die eine Vielzahl von Funktionen und Klassen für häufige Aufgaben wie Dateioperationen, Netzwerkkommunikation, Datenbankzugriff und vieles mehr bietet.
7. **Eventbasierte Programmierung**: C# unterstützt die eventbasierte Programmierung, bei der Ereignisse (Events) ausgelöst werden können und entsprechende Handler definiert werden können, um darauf zu reagieren.
8. **Sicherheitsfeatures**: C# verfügt über integrierte Sicherheitsfeatures wie Garbage Collection, Typsicherheit und Zugriffsmodifikation, die die Integrität und Sicherheit des Codes gewährleisten.

Das sind nur einige der Merkmale, die C# zu einer beliebten und leistungsfähigen Programmiersprache machen. Es wird häufig in der Entwicklung von Desktop-Anwendungen, Webanwendungen, mobilen Apps und Spielen verwendet.

Ein C#-Programm ausführen

1. Im Terminal in den Ordner der C#-Datei navigieren
2. Eingabe: [Pfad zum Compiler-Ordner]/csc [Dateiname]
3. Eingabe: [Dateiname].exe

Textausgabe in einem Terminalfenster

```
Console.WriteLine("Hello World");
Console.ReadLine(); // damit sich das Programm nicht direkt wieder schließt
```

Variablen: Deklaration, Initialisierung und Definition

Deklaration und Initialisierung

```
int myNumber; // Deklaration
myNumber = 10; // Initialisierung (Zuweisung eines Wertes)
```

Definition

```
int myNumber = 0; // Definition = Deklaration + Initialisierung
```

Typinferenz und das Schlüsselwort "var"

```
int explicitVar = 10; // Explicitly typed
var implicitVar = 10; // Implicitly typed ("Typinferenz")
```

Definieren mehrerer Variablen

```
int a, b, c; // Deklarierung
int a = 1, b = 2, c = 3; // Deklarierung + direkte Initialisierung
```

Zuweisungsoperator und Updaten von Variablen

```
var count = 1; // Assign initial value
count = 2;     // Update to new value
```

Klassen definieren und Objekte / Instanzen erstellen

Das wichtigste objektorientierte Konstrukt in C# ist die Klasse, welche eine Kombination aus Daten (Felder) und Verhalten (Methoden) darstellt. Die Felder und Methoden einer Klasse werden als ihre *Member* bezeichnet.

Eine Klasse wird definiert mit dem Schlüsselwort "class" und ein Objekt bzw. eine Instanz aus einer Klasse wird mit dem Schlüsselwort "new" erstellt.

```
class Car
{
        // ...
}

// Erstellen von 2 Instanzen der Klasse "Car"
var myCar = new Car();
var yourCar = new Car();
```

Felder (*fields*)

Felder haben einen Datentyp und können an jeder Stelle in einer Klasse definiert werden. Öffentliche Felder werden in der PascalCase-Schreibweise definiert, während private Felder in der camelCase-Schreibweise mit vorangestelltem Unterstrich definiert werden.

```
class Car
{
        // Accessible by anyone
        public int Weight;

        // Only accessible by code in this class
        private string _color;
}
```

Einem Feld kann ein Anfangswert zugewiesen werden. Wenn ein Feld keinen Anfangswert angibt, wird es auf den Standardwert seines Typs gesetzt. Die Feldwerte einer Instanz können über die Punkt-Notation angesprochen und aktualisiert werden.

```
class Car
{
        // Will be set to specified value
        public int Weight = 2500;

        // Will be set to default value (0)
        public int Year;
}
```

```
var newCar = Car();
newCar.Weight; // Ausgabe: 2500
newCar.Year; // Ausgabe: 0

// Update value of the field
newCar.Year = 2018;
```

Anwendungsbeispiel, wie ein privates Feld genutzt werden kann

```
class CarImporter
{
        private int _carsImported;

        public void ImportCars(int numberOfCars)
        {
                // Update private field from public method
                _carsImported = _carsImported + numberOfCars;
        }
}
```

Zugriffsmodifizierer: "public", "private" und "internal"

Das Schlüsselwort "**public**" vor einer Klasse bestimmt den Zugriffsmodifizierer der Klasse. Er legt fest, von wo aus die Klasse sichtbar und zugänglich ist. Eine "public"-Klasse ist von jedem Ort im selben Projekt sowie von allen anderen Projekten und Bibliotheken, die auf das Projekt verweisen, zugänglich.

```
// Definition einer öffentlichen Klasse
public class PublicClass
{
        public void DoSomething()
        {
                Console.WriteLine("Doing something in a public class.");
        }
}

// Verwendung der öffentlichen Klasse
class Program
{
        static void Main()
        {
                PublicClass instance = new PublicClass();
                instance.DoSomething();
```

```
        }
}
```

Das Schlüsselwort "**private**" vor einer Klasse legt fest, dass die Klasse nur innerhalb der Klasse, in der sie definiert wurde, verwendet werden kann. Im Beispiel unten ist die private Klasse "InnerPrivateClass" nur innerhalb der "OuterClass" zugänglich und kann nicht von außerhalb der "OuterClass" verwendet werden.

Vorteile einer privaten Klasse

- **Kapselung**: Eine private Klasse hilft, Implementierungsdetails zu verbergen und nur den notwendigen Zugriff zu gewähren, was die Modularität und Wartbarkeit des Codes verbessert.
- **Einschränkung des Zugriffs**: Indem der Zugriff auf die Klasse eingeschränkt wird, können unerwünschte Zugriffe von anderen Teilen des Programms verhindert werden, was die Sicherheit erhöht.

```
using System;

public class OuterClass
{
        private class InnerPrivateClass
        {
                public void DoSomething()
                {
                        Console.WriteLine("Doing something in a private inner
class.");
                }
        }

        public void UseInnerClass()
        {
                InnerPrivateClass innerInstance = new InnerPrivateClass();
                innerInstance.DoSomething();
        }
}

class Program
{
        static void Main()
        {
                OuterClass outerInstance = new OuterClass();
                outerInstance.UseInnerClass(); // Ausgabe
```

```
        }
}
```

Wenn Sie einfach nur "class" schreiben, ohne einen expliziten Zugriffsmodifizierer anzugeben, dann hat die Klasse standardmäßig den Zugriffsmodifizierer "internal".

Das Schlüsselwort "**internal**" vor einer Klasse legt fest, dass die Klasse nur innerhalb des Projekts sichtbar und zugänglich. Externe Assemblies (Projekte), die auf das aktuelle Projekt verweisen, können nicht auf "internal" Klassen zugreifen. Dies schränkt den Zugriff auf die Klasse auf den Bereich des aktuellen Projekts ein.

```
internal class HelperClass // "internal" könnte auch weggelassen werden
{
        public void UtilityMethod()
        {
                Console.WriteLine("This method is only accessible within this
project.")
        }
}

public class MainClass
{
        public void MainMethod()
        {
                HelperClass helper = new HelperClass();
                helper.UtilityMethod();
        }
}
```

using System;

Das "using System" am Anfang des Codes ist notwendig, um den Zugriff auf die Klassen und Methoden im "System"-Namespace zu ermöglichen, die im .NET-Framework definiert sind. Der "System"-Namespace enthält grundlegende Typen und Basisfunktionalitäten, die in nahezu jeder .NET-Anwendung verwendet werden. Im Kontext Ihres Beispiels sind dies die Klassen und Methoden, die zum Arbeiten mit der Konsole verwendet werden, wie z. B. "Console.WriteLine". Die Klasse "Console", die zum Schreiben von Ausgaben in die Konsole verwendet wird, befindet sich im "System"-Namespace. Ohne "using System" müssten Sie jedes Mal den vollständigen Namen ("System.Console") verwenden, um auf die "Console"-Klasse zuzugreifen.

```
// Programm ohne "using System;"
class Program
```

```
{
        static void Main()
        {
                System.Console.WriteLine("Test.")
                System.Console.ReadKey();
        }
}

// Programm mit "using System;"
using System;
class Program
{
        static void Main()
        {
                Console.WriteLine("Test.")
                Console.ReadKey();
        }
}
```

Methoden, Parameter und Rückgabewert

Eine Funktion innerhalb einer Klasse wird als "Methode" bezeichnet. Jede Methode kann 0 oder mehr Parameter haben. Alle Parameter müssen explizit typisiert werden, es gibt keine Typinferenz für Parameter. Ebenso muss der Rückgabetyp ebenfalls explizit angegeben werden. Werte werden aus Methoden mit dem Schlüsselwort "return" zurückgegeben. Um es anderen Dateien zu ermöglichen eine Methode aufzurufen, muss der Zugriffsmodifizier "public" hinzugefügt werden.

```
class Calculator
{
        public int Add(int x, int y)
        {
                return x + y;
        }
}
```

Methodenaufruf und Argumente

Methoden werden unter Verwendung der Punktnotation (.) auf einem Objekt / einer Instanz aufgerufen, wobei der Methodenname angegeben wird, der aufgerufen werden soll und Argumente für jeden der Parameter der Methode übergeben. Argumente können optional den Namen des entsprechenden Parameters angeben.

```
// Methodendefinition
class Calculator
{
        public int Add(int x, int y)
        {
                return x + y;
        }
}

// Methodenaufruf
var calculator = new Calculator();
var sum1 = calculator.Add(1, 2);
var sum2 = calculator.Add(x: 1, y: 2);
```

Ein Parameter ist eine Variable, die in der Methodendeklaration definiert ist und dazu dient, Daten an die Methode zu übergeben. Ein Argument ist ein Wert, der beim Aufruf der Methode übergeben wird und der einem bestimmten Parameter zugeordnet wird. Argumente werden verwendet, um den Wert zu übergeben, den die Methode verarbeiten soll.

Ausdrucksbasierte Methoden (*expression-bodied methods*)

Eine ausdrucksbasierte Methode in C# ist eine syntaktische Vereinfachung für die Definition von Methoden, die nur aus einer einzigen Ausdrucksanweisung besteht.

```
// Traditionelle Blockkörper-Sytax
public int GetDouble(int x)
{
        return x * 2;
}

// Ausdrucksbasierte Methode mit Lambda-Operator ("=>")
public int GetDouble(int x) => x * 2;
```

Void-Methoden

Eine **Void-Methode** ist eine Methode, die keinen Rückgabewert liefert. In C# wird eine Methode ohne Rückgabewert mit dem Schlüsselwort "void" gekennzeichnet.

```
public void PrintMessage()
{
        Console.WriteLine("This is a void method.");
}
```

Statische Methoden vs. Instanzmethoden

Statische Methoden gehören zur Klasse selbst und nicht zu einer Instanz (Objekt) dieser Klasse. Sie werden mit dem Schlüsselwort "static" deklariert. Sie werden direkt über den Klassennamen aufgerufen, ohne dass eine Instanz der Klasse erstellt werden muss.

```
public class MethUtils
{
        public static int Add(int a, int b)
        {
                return a + b;
        }
}
// Aufruf der statischen Methode
int result = MathUtils.Add(3, 5); // Ausgabe: 8
```

Instanzmethoden gehören zu einer spezifischen Instanz (Objekt) einer Klasse. Sie werden ohne ein zusätzliches Schlüsselwort deklariert. Sie operieren auf den Daten und Eigenschaften dieser Instanz. Sie werden über eine konkrete Instanz der Klasse aufgerufen, indem sie auf das Objekt angewendet werden.

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

        public void Introduce()
        {
                Console.WriteLine($"Hello, my name is {Name}.");
        }
}
// Erstellung einer Instanz und Aufruf einer Instanzmethode
Person person = new Person();
person.name = "Alice";
person.Introduce(); // Gibt aus: "Hello, my name is Alice."
```

Erweiterungsmethoden (*extension methods*)

Erweiterungsmethoden (*extension methods*) sind spezielle Methoden, die es ermöglichen, bestehende Klasse um neue Methoden zu erweitern, ohne den Quellcode dieser Klassen ändern zu müssen. Sie bieten eine elegante Möglichkeit, Funktionalitäten hinzuzufügen, insbesondere zu Klassen, die sie nicht selbst geschrieben haben oder nicht ändern können wie Klassen aus Frameworks oder Bibliotheken.

Erweiterungsmethoden sind statische Methoden, die in einer statischen Klasse definiert werden. Sie müssen einen Parameter vom Typ der Klasse enthalten, die Sie erweitern möchten. Dieser Parameter wird mit dem Schlüsselwort "**this**" vor dem Typ versehen. Nachdem Sie eine Erweiterungsmethode definiert haben, können Sie sie verwenden, als wäre sie eine normale Methode der erweiterten Klasse.

```
using System;

public static class StringExt
{
        // Erweiterungsmethode für den Datentyp "string"
        public static int WordCount(this string str)
        {
                // Method-Chaining
                return str.Split().Length;
        }

        public static int StringToInt(this string str)
        {
                return Convert.ToInt32(str);
        }
}

class Program
{
        static void Main()
        {
                string message = "This is an example.";
                int numberOfWords = message.WordCount();
                Console.WriteLine($"Wortanzahl: {numberOfWords}");

                string year = "2023";
                int newYear = year.StringToInt() + 1;
                Console.WriteLine($"Jahr: {newYear}");

                Console.ReadKey();
        }
}
```

Exercism-Beispiel

Task: Implement an extension method, that takes in some string delimiter and return the substring after the delimiter.

```
using System.Reflection;

class Program
{
    static void Main()
    {
        var log = "[INFO]: File Deleted.";
        string substring = log.SubstringAfter(": ");
        Console.WriteLine($"Substring: {substring}");
        Console.ReadKey();
    }
}

static class StringExt
{
    public static string SubstringAfter(this string str, string delimiter)
    {.
        int delimiterIndex = str.IndexOf(delimiter);
        if (delimiterIndex >= 0)
        {
            return str.Substring(delimiterIndex + delimiter.Length).Trim();
        }
        else
        {
            return str;
        }
    }
}
```

Datentyp: Bool

Booleans in C# are represented by the "bool" type, which values can be either "true" or "false". C# supports three boolean operators: "!" (NOT), "&&" (AND) and "||" (OR).

```
bool info = true;
class Beispiel
{
        public bool Swap(bool info)
        {
                if (info == true;) {
                        return false;
                }
                else {
                        return true;
                }
```

```
        }
}
```

Datentyp: String

A "string" in C# is an object that represents immutable text as a sequence of Unicode characters (letters, digits, punctuation, etc.). Double quotes are used to define a "string" instance.

```
string fruit = "Apple";
```

Multiple strings can be concatenated (added) together. The simplest way to achieve this is by using the "+" operator.

```
string name = "Jane";
"Hello " + name + "!";
// Ausgabe: "Hello Jane!"
```

For any string formatting more complex than simple concatenation, string interpolation is preferred. To use interpolation in a string, prefix it with the dollar ("$") symbol. Then you can use curly braces ("{ }") to access any variables inside your string.

```
string name = "Jane";
$"Hello {name}!";
// Ausgabe: "Hello Jane!"
```

String-Methoden

```
Replace("x", "y") // Ersetzt alle Vorkommen eines bestimmten Zeichens oder
einer Zeichenfolge in einem String durch eine andere Zeichenfolge

Contains("message") // Überprüft, ob ein bestimmtes Zeichen oder eine
bestimmte Zeichenfolge in einem String enthalten ist; gibt "true" zurück, wenn
der String die Zeichenfolge enthält, andernfalls "false"

Trim() // Entfernt führende und abschließende Leerzeichen aus einem String

Substring(int startIndex) // Methode wird an einem String ausgeführt; nimmt
als Parameter einen Integer-Wert entgegen, der den nullbasierten Index des
ersten Zeichens angibt, das in den Substring aufgenommen werden soll.

Substring(int startIndex, int Length) // Siehe oben + nimmt zusätzlich als 2.
```

```
Parameter die Länge der Unterzeichenkette entgegen, die zurückgegeben werden
soll

IndexOf(string value) // Gibt den null-basierten Index des ersten Vorkommens
der angegebenen Zeichenkette zurück; wenn die Zeichenkette nicht gefunden
wird, wird "-1" zurückgegeben

ToUpper(); // Wandelt alle Buchstaben in Großbuchstaben um
```

String-Konkatenation mit "StringBuilder"

```
using System;
using System.Text;

class Program {
        static void Main() {
                StringBuilder sb = new StringBuilder();
                int counter = 0;
                while (counter < 10) {
                        sb.Append("Text");
                        counter++;
                }
                string result = sb.ToString();
                Console.WriteLine(result);
        }
}
```

Implizite u. explizite Konvertierungen

```
int intValue = 42;
double doubleValue = intValue; // Implizite Konvertierung; kein Datenverlust
Console.WriteLine(doubleValue); // Output: 42.0
```

```
double doubleValue = 42.58;
int intValue = (int)doubleValue; // Explizite Konvertierung; verlustbehaftet
Console.WriteLine(intValue); // Output: 42
```

Kontrollstruktur: if, else if & else

```
// Exercism-Example
using System;
static class Assembly Line
```

```
{
        public static double SuccessRate(int speed)
        {
                if (speed == 0)
                {
                        return 0.00;
                }
                else if (speed >= 1 && speed <= 9)
                {
                        return 0.85;
                }
                else
                {
                        return 0.77;
                }
        }
}
```

Kontrollstruktur: Ternärer Operator

Der ternäre Operator ist ein kompakter Weg, um eine Bedingung auszuwerten und einen von zwei Werten basierend auf dem Ergebnis der Bedingung zurückzugeben. Der Operator hat folgende Syntax: (condition) ? (consequent) : (alternative)

```
// Beispiel
using System;

class Programm
{
        static void Main()
        {
                int number = 10;
                string result = (number % 2 == 0) ? "even" : "odd";
                Console.WriteLine($"The number {number} is {result}.");
        }
}
```

Beispiel: Einfaches Konsolenprogramm

```
// Verwendung des "System"-Namespace
using System;

// Deklaration einer öffentlichen Klasse namens "Test"
public class Test {
```

```
// Definition der öffentlichen Methode "Add"
        public int Add(int x, int y) {

// Rückgabewert
                return x + y;
        }
}

// Deklaration einer öffentlichen Klasse namens "Program"
public class Program {

// Definition der "Main"-Methode; sie ust der Einstiegspunkt für das Programm;
sie ist öffentlich, statisch und gibt keinen Wert zurück
        public static void Main() {

// Erstellen eines Objekts "testobj_v1" der Klasse "Test";
                var testobj_v1 = new Test();

// Aufruf der "Add"-Methode des "testobj_v1"-Objekts + Übergabe der Argumente
"3" und "5"; das Ergebnis wird in der Variablen "sum" gespeichert
                int sum = testobj_v1.Add(3, 5);

// Ausgabe des Ergebnisses
                Console.WriteLine($"Result: {sum}");

// Beliebiger Tastenanschlag beendet das Programm
                Console.ReadKey();
        }
}
```

Datenstruktur: Tupel

Ein Tupel ist eine Datenstruktur, die eine geordnete Liste von Elementen verschiedener Typen enthält. Es ist ein Container für eine feste Anzahl von Werten, die nicht notwendigerweise denselben Typ haben müssen. Zum Beispiel kann ein Tupel verwendet werden, um eine Methode mehrere Werte zurückgeben zu lassen.

```
public class Program {
        public static void Main() {

                // Einen Tupel erstellen (Möglichkeit 1)
                (string, int) person = ("Christoph, 30);

                // Einen Tupel erstellen (Möglichkeit 2)
```

```
                (string name, int age) person = ("Christoph", 30);

                // Einen Tupel erstellen (Möglichkeit 3)
                string name = "Christoph";
                int age = 30;
                (string name, int age) person = (name, age);

                // Einen Tupel erstellen (Möglichkeit 4)
                var person = (name: "Christoph", age: 30);

                // Einen Tupel erstellen (Möglichkeit 5)
                var person = Tuple.Create("Christoph", 30);

                // Ausgabe (funktioniert für alle Möglichkeiten)
                Console.WriteLine($"Name: {person.Item1}");
                Console.WriteLIne($"Alter: {person.Item2}");
                Console.ReadKey();

                // Ausgabe (funktioniert für Möglichkeit 2, 3 und 4)
                Console.WriteLine($"Name: {person.name}");
                Console.WriteLIne($"Alter: {person.age}");
                Console.ReadKey();
        }
}
```

Beispiel: Tupel als Rückgabewert einer Methode

```
public class Program {
        public static void Main() {

                // Rückgabewert der Methode "GetSameOrBigger" (= Tupel) in der
Variable
                // "result" speichern
                var result = GetSameOrBigger(10, 20);

                // Ausgabe
                Console.WriteLine($"Gleich: {result.Item1}, Größere Zahl:
{result.Item2}");
                Console.ReadKey();
        }

        // Statische Methode "GetSameOrBigger"
        static (bool, int) GetSameOrBigger (int num1, int num2) {

                // Ternärer Operator; Syntax => condition ? consequent :
```

```
alternative
                return (num1 == num2, num1 > num2 ? num1 : num2);
        }
}
```

Beispiel: Tupel als Parameter einer Methoden

```
public class Program {
        public static void() {
                // Ein Tupel mit zwei Ganzzahlen erstellen
                (int, int) mySummands = (2, 3);

                // Die Add-Methode aufrufen und das Ergebnis speichern
                int result = Add(mySsummands);

                // Das Ergebnis ausgeben
                Console.WriteLine($"Ergebnis: {result}");

        }

        static int Add((int, int) summands) {
                return summands.Item1 + summands.Item2;
        }
}
```

Kontrollstruktur: While-Loop & Do-While-Loop

```
// While Loop
int x = 23;
while (x > 10)
{
        x = x - 2;
}
```

```
// Do While Loop
int x = 23;
do
{
        x = x - 2;
} while (x > 10);
```

Datentyp: Verweistyp vs. Werttyp

Verweistypen (*reference types*)	**Werttypen (*value types*)**
Variablen speichern Verweise auf ihre Daten	Variablen enthalten ihre Daten direkt
Zwei Variablen können auf dasselbe Objekt verweisen; angewendete Operationen auf eine Variable können das Objekt beeinflussen, auf das von einer anderen Variable verwiesen wird	Jede Variable hat eine eigene Kopie der Daten; eine auf eine Variable angewendete Vorgänge können die andere Variable nicht beeinflussen
Klassen (class), Arrays, Interfaces (interface), Delegaten (delegate), String (string), Objekte (object)	Primitive Datentypen (int, double, float, char, bool, usw.), Strukturen (struct), Enumerationen (enum)

Nullbarkeit (*nullability*)

Das Konzept der Nullbarkeit bezieht sich auf die Fähigkeit eines Datentyps, den speziellen Wert "null" annehmen zu können. Verweistypen sind i. d. R. nullbar. Ein Werttyp ist i. d. R. nicht nullbar, kann aber nullbar gemacht werden, indem man dem Dateityp ein Fragezeichen (?) anhängt.

```
string nullableReferenceType = "hello";
nullableReferenceType = null; // valid

int nonNullableValueType = 5;
nonNullableValueType = null; // compile error

int? nullableValueType = 5;
nullableValueType = null; // valid
```

Um sicher mit nullbaren Werten zu arbeiten, sollte man überprüfen, ob sie "null" sind, bevor man mit ihnen arbeitet.

```
string NormalizedName(string? name) {
        if (name == null) {
                return "UNKNOWN";
        }
        else {
                return name.ToUpper();
        }
}
NormalizedName(null); // => "UNKNOWN"
NormalizedName("Elisabth"); // => "ELISABETH"
```

Der ?? Operator ermöglicht es, einen Standardwert zurückzugeben, wenn der Wert "null" ist.

```
string? name1 = "John";
name1 ?? "Anonymous"; // => "John"

string? name2 = null;
name2 ?? "Anonymous"; // => "Anonymous"
```

Der ?. Operator ermöglicht es, sicher eine Variable aufzurufen, die möglicherweise den Wert "null" hat, ohne dass eine "NullReferenceExpection" ausgeworfen wird.

```
string? fruit = "apple";
fruit?.Length; // Ausgabe: 5

string? vegetable = null;
vegetable?.Length; // Ausgabe: null
```

Datenstruktur: Array

Datenstrukturen, die null oder mehr Elemente beinhalten, werden Sammlungen (engl. *collections*) genannt. Ein Array ist eine Sammlung mit einer festen Größe bzw. Länge und deren Elemente alle den gleichen Datentyp haben.

```
// Deklarieren eines Arrays mit der feste Größe (2)
int[] twoInts = new int[2];

// Zuweisen von Elemente über den Index
twoInts[0] = 4;
twoInts[1] = 7;

// Abrufen eines Elements über den Index
twoInts[1] == 7; // Ausgabe: true
```

Weitere Methoden der Deklarierung und Initialisierung eines Arrays

```
int[] threeInts_v1 = new int[] {4, 7, 9};
int[] threeInts_v2 = new[] {4, 7, 9};
int[] threeInts_v3 = {4, 7, 9};
```

Kontrollstruktur: For-Loop

Ein For-Loop besteht aus 4 Bestandteilen: Dem Initialisierer, der Bedingung, dem Iterator und dem Schleifenkörper.

```
for (int i = 0; i < 5; i++) {
        System.Console.Write(i);
}
// Ausgabe: 01234
```

Kontrollstruktur: Foreach-Loop

Eine Foreach-Schleife wird verwendet, um über eine Sammlung oder ein Array zu iterieren, ohne explizit einen Zähler oder Index zu verwalten. Sie ermöglicht das einfache Durchlaufen von Elementen in einer Sammlung wie Listen, Arrays oder anderen vergleichbaren Objekten.

```
char[] vowels = new[] {'a', 'e', 'i', 'o', 'u'};
foreach (char vowel in vowels)
{
        System.Console.Write(vowel);
}
// Ausgabe: aeiou
```

Randomness

```
using System;
public class Player
{
        private Random random = new Random();
        public int RollDie() => random.Next(1,19);
        public double GenerateSpellStrength() => random.NextDouble();
}
// Ausgabe: Zufallszahl zw. 1 und 18
```

Datentyp: Char

Der Datentyp "**char**" ist ein Werttyp, der ein einzelnes 16-Bit-Unicode-Zeichen repräsentiert. Der Wertebereich reicht von "\u0000" (0) bis "\uffff" (65.535). Es gibt viele eingebaute Bibliotheksmethoden, um Zeichen zu inspizieren und zu manipulieren. Diese sind als statische Methoden in der Klasse "**System.Char**" zu finden.

```
// Deklaration und Initialisierung
char letterA = 'A';
char letterB = '\u0042'; // Unicode für 'B'
char newLine = '\n'; // Zeilenumbruch-Zeichen
```

Steuerzeichen (*control characters*) sind spezielle Zeichen im ASCII- oder Unicode-Zeichensatz, die keine grafische Darstellung haben, sondern Steuerfunktionen im Text oder bei der Datenübertragung ausführen.

- Null-Byte (NUL, '\0'): Kennzeichnet das Ende eines Strings
- Zeilenumbruch (LF, '\n'): Markiert das Ende einer Zeile
- Wagenrücklauf (CR, '\r'): Bewegt den Cursor an den Anfang der Zeile
- Tabulator (TAB, '\t'): Fügt eine horizontale Tabulatorstelle ein

Datenstruktur: Liste

A collection definition typically includes a placeholder in angle brackets, often "T" by convention. Such a collection is referred to as a generic type. This allows the collection user to specify what type of items to store in the collection. In the example code below we are instantiating a list of strings.

Eine Liste initialisieren + Werte hinzufügen

```
List<string> languages = new List<string>();
languages.Add("Python");
languages.Add("PHP");
languages.Add("C#");
```

Einen konkreten Wert ansprechen + Liste auf konkrete Werte überprüfen

```
bool isInList = false;
if (languages.Contains("TypeScript")) {
        isInList = true;
}
return isInList;
```

```
bool isExciting = false;
if (languages[0] == "C#") {
        isExciting = true;
}
return isExciting;
```

Die Länge einer Liste ermitteln

```
languages.Count;
```

Die Reihenfolge einer Liste invertieren

```
languages.Reverse();
```

Eine Liste sortieren

```
languages.Sort();
```

Einen konkreten Wert löschen

```
languages.Remove("Python");
```

Methode: Liste auf Einzigartigkeit prüfen

```
public static bool IsUnique (List<string> languages) {
        languages.Sort();
        for (int i = 1; i < languages.Count; i++) {
                if (languages[i].Equals[languages[i-1]]) {
                        return false;
                }
        }
        return true;
}
```