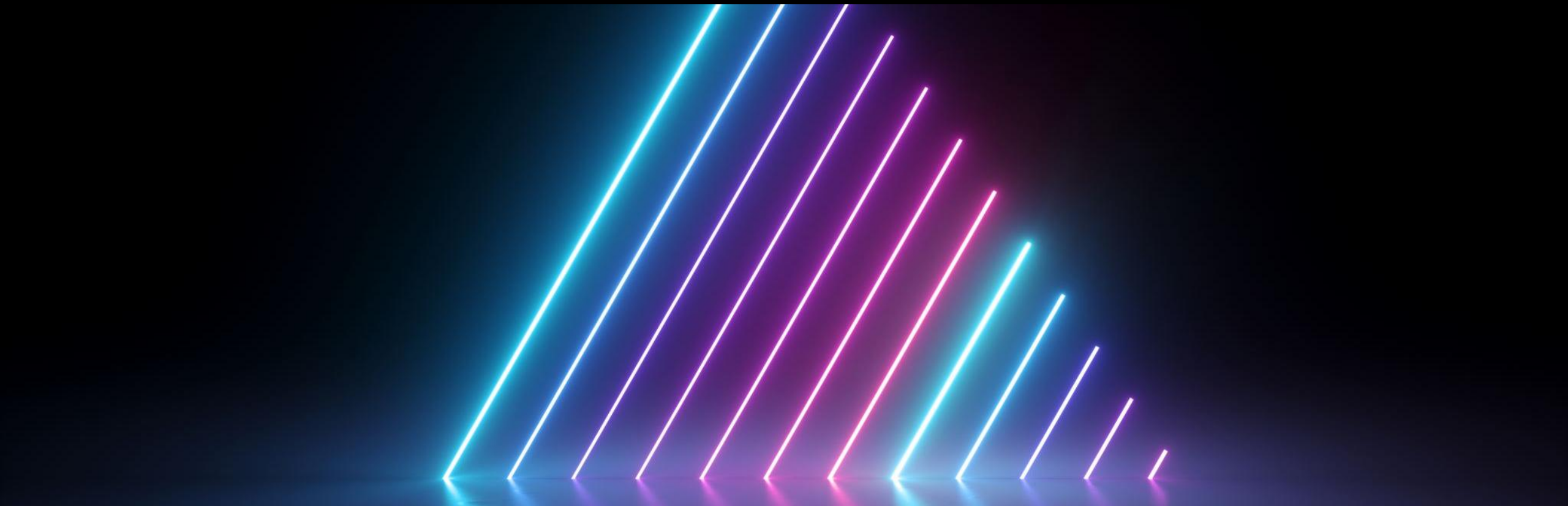
Java vs. CSharp

04.04.2025



Plan für die Woche

SMART INDUSTRY CAMPUS

Montag

- Datenbankanbindung mit C#
- Implementierung der DAO-Schicht in C#

Dienstag

- GUI-Programmierung in C# mit Hilfe der Toolbox in VS
- Implementierung der GUI um Bücher hinzuzufügen

Mittwoch

- Projekt

Donnerstag

- Projekt

Freitag

- Hauptunterschiede zwischen Java und C#
- Programmieren eines Taschenrechners

Plan für diese Woche

- ✓ Grundlegende Unterschiede zwischen Java und C#
- ✓ Herunterladen der IDE
- ✓ Taschenrechner programmieren

Grundlegende Unterschiede

Generelle Regeln

- → **Sowohl C# als auch Java sind objektorientierte Sprachen**
- → **Beide Sprachen verfügen über einen Garbage Collector (automatische Speicherverwaltung)**
- → **Beide Sprachen sind plattformunabhängig, wobei C# am Anfang primär für Windows gedacht war**

Generelle Regeln

- → Methoden und Konstanten werden in PascalCase geschrieben (z.b. GetMethod)
  - → Konstanten werden mit dem Schlüsselwort **const** gekennzeichnet
- → Variablen sollten mit einem _ anfangen
- → Datentyp String wird klein geschrieben (string)
- → Datentyp boolean wird als bool hinterlegt
- → Das C#-äquivalent zu Java final ist readonly
- → **Wenn man Visual Studio nutzt, kann pro Projekt nur eine Main-Methode existieren!**

Main-Methode

- → In C# existiert ebenfalls die Main-Methode, jedoch muss sie anders geschrieben werden
- → Kann auch keinen Parameter entgegennehmen
- → Kann auch einen int zurückgeben
- → Main-Methoden können ebenfalls überladen werden
- → **Vorsicht**: Wenn man zwei legitime Einstiegspunkte definiert, wird der Code zur Laufzeit abbrechen

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
```

Konsolenausgabe

→ Um eine Konsolenausgabe zu tätigen, schreibt man in C# anders als in Java folgendes:

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        Console.WriteLine("Hallo");
        Console.Write("Tschüß");
        Console.Write("Tschüß2");
    }
}
```

```
Hallo
TschüßTschüß2
```

Aufgabe

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        Console.WriteLine("Hallo");
        Console.Write("Tschüß");
        Console.Write("Tschüß2");
    }
}
```

→ Öffnet Visual Studio

→ Drückt „Neues Projekt erstellen“

→ Wähle „Konsolen-App“

→ Als Projektname: „ErsterCSharpCode“

→ Schreibe eine Klasse und eine Main Methode und gebe in der Konsole „Ich bin bereit C# zu lernen“ aus.

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
```

User Input

- → Um eine Nutzereingabe zu machen, wird anders als in Java vorgegangen
- → Äquivalent auch für andere Datentypen
  - → Nur muss dann der übergebene string in einen int umgewandelt werden

```
public static void Main()
{
    Console.WriteLine("Enter username: ");
    string username = Console.ReadLine();
    Console.WriteLine(username);
}
```

```
public static void Main()
{
    Console.WriteLine("Enter your age:");
    int age = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Your age is: " + age);
}
```

Formatierter String

→ In C# kann ein string direkt formatiert werden, sodass Werte eingegeben werden können

```
public static void Main()
{
    string firstName = "John";
    string lastName = "Doe";
    string name = $"My full name is: {firstName} {lastName}";
    Console.WriteLine(name);

}
```

Aufgabe

```
public static void Main()
{
    string firstName = "John";
    string lastName = "Doe";
    string name = $"My full name is: {firstName} {lastName}";
    Console.WriteLine(name);

}
```

→ Kommentiere den Code aus

→ Ändere deinen Code so ab, dass du vom Nutzer den Namen erfragst und dann in der Konsole ausgibst

```
public static void Main()
{
    Console.WriteLine("Enter username: ");
    string username = Console.ReadLine();
    Console.WriteLine(username);
}
```

Access Modifier

- → Ähnlich wie in Java gibt es:
  - → public, private und protected
  - → Internal heißt, dass es nur innerhalb des gleichen Projektes/Moduls (Assembly) sichtbar ist
  - → Private protected in abgeleiteten Klassen innerhalb desselben Assembly

Getter und Setter

- → Getter und Setter in C# können wie in Java geschrieben werden
- → Es besteht die Möglichkeit einer kürzeren Schreibweise (Properties)

```
public class Employee
{
    private int _id;                        //Datenfeld
    2 Verweise
    public int Id                           //Getter und Setter für die Methode
    {
        get { return _id; }
        set { _id = value; }
    }

    0 Verweise
    public static void Main()
    {
        Employee employee = new Employee();
        employee.Id = 1;
        Console.WriteLine(employee.Id);
    }
}
```

```
public class Employee
{
    2 Verweise
    public int Id                           //Datenfeld wird automatisch angelegt
    { get; set; }

    0 Verweise
    public static void Main()
    {
        Employee employee = new Employee();
        employee.Id = 1;
        Console.WriteLine(employee.Id);
    }
}
```

Getter und Setter

→ Getter und Setter in C# können wie in Java geschrieben werden

→ Es besteht die Möglichkeit einer kürzeren Schreibweise (Properties)

```
public class Person
{
    1 Verweis
    public string Name { get; protected set; }

    0 Verweise
    public Person(string name)
    {
        Name = name;
    }
}
```

```
public class Person
{
    1 Verweis
    protected string Name { get; set; }

    0 Verweise
    public Person(string name)
    {
        Name = name;
    }
}
```

Aufgabe

- → Kommentiere deinen bisherigen Code aus
- → Schreibe eine Klasse Animal
- → Ein Animal hat einen Namen und ein Alter
- → Schreibe Getter und Setter

```
public class Employee
{
    private int _id;                        //Datenfeld
    2 Verweise
    public int Id                           //Getter und Setter für die Methode
    {
        get { return _id; }
        set { _id = value; }
    }

    0 Verweise
    public static void Main()
    {
        Employee employee = new Employee();
        employee.Id = 1;
        Console.WriteLine(employee.Id);
    }
}
```

ODER

```
public class Employee
{
    2 Verweise
    public int Id                           //Datenfeld wird automatisch angelegt
    { get; set; }

    0 Verweise
    public static void Main()
    {
        Employee employee = new Employee();
        employee.Id = 1;
        Console.WriteLine(employee.Id);
    }
}
```

Default-Werte als Parameter

→ In C# können Methoden-Parameter mit Standardwerten definiert werden

→ Falls beim Methodenaufruf kein Argument übergeben wird, wird automatisch der definierte Standardwert verwendet

```
static void DefaultParamMethod(string defaultParam = "Default")
{
    Console.WriteLine(defaultParam);
}
0 Verweise
public static void Main()
{
    DefaultParamMethod();
}
```

```
Default
```

Named Arguments

→ In C# kann man bei der Übergabe von Parametern die Reihenfolge dieser abändern, indem man den Parameternamen nennt

```
1 Verweis
static void YoungestChild(string child1, string child2, string child3)
{
    Console.WriteLine("The youngest child is: " + child3);
}

0 Verweise
static void Main(string[] args)
{
    YoungestChild(child3: "John", child1: "Liam", child2: "Liam");
}
```

Arrays

→ Gibt verschiedene Möglichkeiten

→ Prinzipiell wie in Java, jedoch mehrere Varianten möglich

```
public static void Main()
{
    // Create an array of four elements, and add values later
    string[] cars = new string[4];

    // Create an array of four elements and add values right away
    string[] cars1 = new string[4] { "Volvo", "BMW", "Ford", "Mazda" };

    // Create an array of four elements without specifying the size
    string[] cars2 = new string[] { "Volvo", "BMW", "Ford", "Mazda" };

    // Create an array of four elements, omitting the new keyword, and without specifying the size
    string[] cars3 = { "Volvo", "BMW", "Ford", "Mazda" };


}
```

Arrays

→ Bei mehrdimensionalen Arrays sieht die Syntax anders zu Java aus

```
public static void Main()
{
    int[,] numbersA = new int[10,1];
    int[,] numbers = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

Aufgabe

```
public static void Main()
{
    // Create an array of four elements, and add values later
    string[] cars = new string[4];

    // Create an array of four elements and add values right away
    string[] cars1 = new string[4] { "Volvo", "BMW", "Ford", "Mazda" };

    // Create an array of four elements without specifying the size
    string[] cars2 = new string[] { "Volvo", "BMW", "Ford", "Mazda" };

    // Create an array of four elements, omitting the new keyword, and without specifying the size
    string[] cars3 = { "Volvo", "BMW", "Ford", "Mazda" };


}
```

→ Erstelle eine Main Methode und füge ihr einen Array mit Animal-Objekte hinzu

→ Iteriere mit Hilfe einer standard for-Schleife über diese Liste und gebe immer den Namen und das Alter aus

Listen

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        names.Add("David");
    }
}
```

- → Werte können einer Liste direkt zugewiesen werden
- → Aber auch wie in Java wieder hinzugefügt werden
- → „Primitive“ Datentypen sind legal in der <>-Annotation
  - → In C# werden primitive Datentypen als Objekte behandelt
  - → Sie können jedoch nicht null sein

Nullable Value Types

```
public class Employee
{
    0 Verweise
    public static void Main()
    {
        int? number = null;             //Java:  Integer number = null;
        Console.WriteLine(number);      //Produziert kein Nullpointer in C#
    }
}
```

→ „Primitive“ Datentypen sind legal in der <>-Annotation

→ In C# werden primitive Datentypen als Objekte behandelt

→ Sie können jedoch nicht null sein

For-each-Schleife

→ In C# wird die Schleife tatsächlich auch foreach genannt

→ Statt
int i : list
wird
int i in list
geschrieben

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        foreach (string name in names)
        {
            Console.WriteLine(name);
        }
    }
}
```

Aufgabe

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        names.Add("David");
    }
}
```

→ Kommentiere deinen bisherigen Code aus

→ Ändere deinen Code so ab,
dass das eine Liste mit Animal-Elementen existiert

→ Iteriere mit Hilfe einer foreach-Schleife über diese Liste und gebe immer den Namen und das Alter aus

```
public class Test
{
    0 Verweise
    public static void Main()
    {
        List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
        foreach (string name in names)
        {
            Console.WriteLine(name);
        }
    }
}
```

Vererbung

- → Statt des reservierten Wortes extends nutzt C# einfach ein „:“
- → Statt der @Override-Notation wird in C# override genutzt
- → Statt super() wird base() genutzt
- → Statt super. wird base. genutzt

```
class Animal
{
    1 Verweis
    public virtual void MakeSound()
    {
        Console.WriteLine("Animal sound");
    }
}

0 Verweise
class Dog : Animal
{
    1 Verweis
    public override void MakeSound()
    {
        Console.WriteLine("Bark!");
    }
}
```

```
class Person
{
    private string name;
    private int age;
    1 Verweis
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

}

1 Verweis
class Employee:Person
{
    0 Verweise
    public Employee(string name, int age) : base(name, age)
    {

    }
}
```

Aufgabe

- → Dog erbt von Animal
- → Der Hund hat jedoch noch 2 weitere Attribute wie:
  - → lengthOfFur
  - → typeOfDog
- → Implementiere das Szenario

```
class Person
{
    private string name;
    private int age;
    1 Verweis
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

}

1 Verweis
class Employee:Person
{
    0 Verweise
    public Employee(string name, int age) : base(name, age)
    {

    }
}
```

Namespaces

- → Zum Organisieren von Code
- → Zum Verhindern von Namenskonflikten
- → Strukturierung von Code
- → In Java verwendet man das Schlüsselwort package
  - → Dann import ….
- → In C# namespace
  - → Dann using….

```
namespace KonsolenAppMitDBAnbindung.service
{
    2 Verweise
    class DBCreator
    {

    }
}
```

```
using KonsolenAppMitDBAnbindung.service;
0 Verweise
class Program
{
    0 Verweise
    public static void Main()
    {
        DBCreator db = new DBCreator();
    }
}
```

Aufgabe

- → Erstelle ein Package mit einer Testmethode
  - → Rechtsklick auf Projekt – Hinzufügen – Ordner
  - → Rechtsklick auf Ordner – Hinzufügen – Element
- → Rufe dieses Package in einem anderen Package auf
- → Wie bekommt man Zugriff auf die Testmethode?

```
namespace KonsolenAppMitDBAnbindung.service
{
    2 Verweise
    class DBCreator
    {

    }
}
```

```
using KonsolenAppMitDBAnbindung.service;
0 Verweise
class Program
{
    0 Verweise
    public static void Main()
    {
        DBCreator db = new DBCreator();
    }
}
```

Try with Ressources

→ Zum Automatischen Schließen von Streams, Ressourcen, …

→ Schlüsselwort using wird gentuzt

```
try
{
    using (MySqlConnection connection = DBConnector.GetConnection())
    {
        connection.Open();
        //@ tells the compiler to interpret the string exactly as it is written without \t or \n needed
        string sql = @"DROP DATABASE IF EXISTS Library2;
                  CREATE DATABASE IF NOT EXISTS Library2;
                  USE Library2;

                  CREATE TABLE Authors (
                      author_id INT AUTO_INCREMENT PRIMARY KEY,
                      name VARCHAR(255) NOT NULL
                  );
```

Lambdas

→ Lambdas werden statt einem -> mit => dargestellt

```
public static void Main()
{
    Predicate<int> isEven = n => n % 2 == 0;
    Console.WriteLine(isEven(4));
}
```

Aufgabe

- → Programmiert einen Taschenrechner in C#
- → Erstelle hierfür ein neues Projekt oder schreibe eine Klasse
  - → Bei der Klassenvariante: Achte darauf, dass es nur eine Main-Methode geben kann