Informationstechnologie I (T4SST1005)

Sebastian Schwendemann David Waidner

Willkommen zur Vorlesung

Motivation – Das „Big Picture“:

- Programmieren ist eine Schlüsselkompetenz, die immer häufiger wird
- Einsatzgebiete: Spieleentwicklung, Webanwendungen, Datenanalyse, Maschinensteuerung, Automatisierung

Details:

- Warum Programmieren wichtig ist: Automatisierung wiederkehrender Aufgaben, Problemlösung durch Algorithmen.
- Beispiele konkretisieren: Welches Fachgebiet nutzt welche Technologien (z. B. Datenanalyse → Python, Spiele → C#/Unity).
- Wichtige Lernhaltung: Fehler sind normal; Debugging ist Teil des Lernprozesses.

Organisatoriches

- 2 Dozenten in abwechselnden Vorlesungen
- Eine schriftliche Klausur am Ende
- Übungen in Vorlesung + Optionale Übungen zwischen den Vorlesungen

---

Lernziele

Am Ende der heutigen Sitzung können Sie:

- Ein erstes C#-Programm schreiben und ausführen
- Verstehen, was eine Variable ist und diese verwenden
- Mit if-else Bedingungen Entscheidungen im Programm modellieren
- Grundlegende Datentypen in C# unterscheiden und nutzen

Details:

- Konkrete Erwartungen: Sie sollen ein kleines Programm selbst starten und ändern können.
- Bewertung: Übungsaufgaben dienen dem Verständnis, nicht der Perfektion.

Was ist (prozedurales) Programmieren?

- Schritt-für-Schritt-Anweisungen an den Computer
- Programme bestehen aus Daten + Anweisungen
- **Prozedural** = Befehle werden nacheinander ausgeführt

Details:

- Kontrast zu anderen Paradigmen: kurz erwähnen objektorientiert und funktional (nur zum Einordnen).
- Typische Struktur prozeduraler Programme: Eingabe → Verarbeitung → Ausgabe.
- Einfache Beispielidee: Rezept als Analogie (Zutaten = Daten, Schritte = Anweisungen).

Was ist ein Algorithmus?

- Definition: Eine eindeutige Abfolge von Schritten zur Lösung eines Problems
- Eigenschaften: endlich, eindeutig, effektiv

Details:

- "Eindeutig": Jeder Schritt muss klar ausführbar sein — keine Mehrdeutigkeit.
- "Effektiv": Schritte müssen tatsächlich ausführbar sein (realisierbare Operationen).
- Beispiel: Sortierverfahren (kurze Erwähnung Bubble/Insertion als Einstieg).
- Hinweis: Algorithmen werden später in Pseudocode oder Code umgesetzt.

Darstellungsmethoden

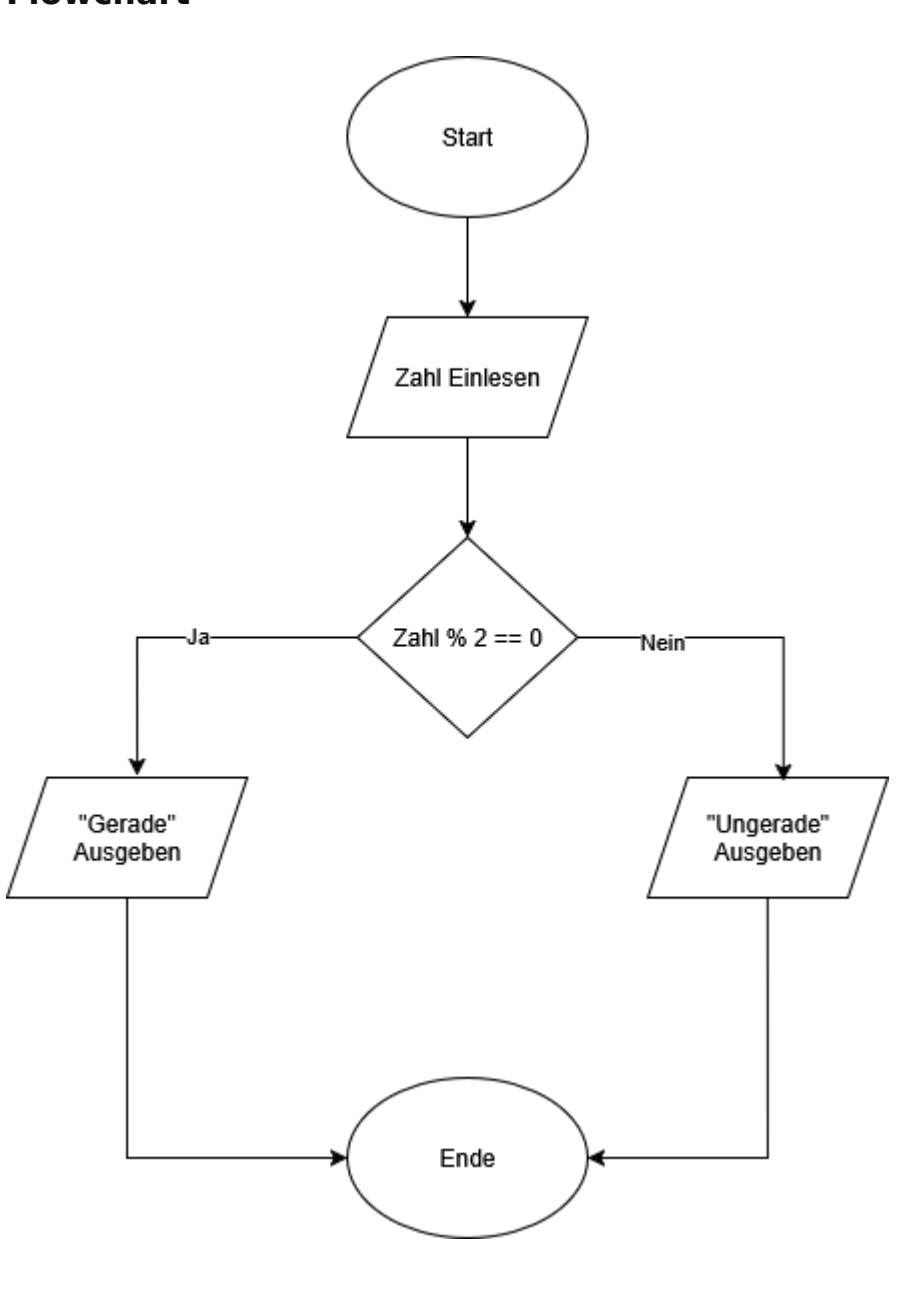
Pseudocode:

```
Lies Zahl ein
Wenn Zahl % 2 == 0 dann
    Ausgabe „Gerade“
Sonst
    Ausgabe „Ungerade“
Ende
```

Details:

- Zweck von Pseudocode: Konzepte beschreiben ohne Syntaxdetails einer Programmiersprache.
- Strukturieren mit Einrückung zeigt Kontrollfluss (wie in echtem Code).
- Beim Übertragen in Code: Bedingung (% 2 0) → in C#: if (zahl % 2 0) { … }.
- Tipp: Pseudocode sollte so präzise wie möglich sein, um Implementierungsfehler zu vermeiden.

Flowchart



Details:

- Start/Ende (runde Felder) markieren Programmbeginn und -ende.
- Parallelogramme für I/O (z. B. Eingabe lesen, Ausgabe schreiben).
- Rauten für Entscheidungen → Pfade basierend auf true/false.
- Nutzen: Komplexe Abläufe visualisieren, bevor man Code schreibt.
- Umsetzung in Code: Verzweigungen → if/else, Schleifen → Schleifenstrukturen.

Programmieren in der Praxis

- Es gibt viele Programmiersprachen (Python, Java, C, C#, …)
- Wir nutzen **C#**, weil:
  - Modern, gut geeignet für Einsteiger
  - Breites Einsatzgebiet: Desktop, Web, Spiele, Enterprise
  - Gute Integration mit Tools und Frameworks

Details:

- Warum C#: gute Balance aus einfacher Syntax und starker Typisierung; viele Lernressourcen.
- Werkzeuge: IDE (z. B. Visual Studio / Codespaces) erleichtern Debugging, Autovervollständigung.
- Tipp: Fokus auf Konzepte — sie lassen sich später auf andere Sprachen übertragen.

Erstes „Hello World“ Programm

```
Console.WriteLine("Hello, World!");
```

Details:

- Was passiert beim Ausführen: Text wird auf der Konsole ausgegeben; Programm endet.
- Wo ausführen: In Program.cs, in einem kleinen Konsolenprojekt oder mittels Top-Level-Statements.
- Fehlerquellen: Vergessene Anführungszeichen oder Semikolon führen zu Compile-Fehlern.
- Lernziel dieser Folie: Verständnis für Aufruf einer Methode und Ausgabe auf der Konsole.

Erklärung „Hello World“

- `Console.WriteLine` = Funktionsaufruf
- `("Hello, World!")` = Übergabe eines Strings
- `;` am Ende jeder Anweisung

Details:

- Console ist eine Klasse in System-Namespace; WriteLine ist eine Methode.
- Unterschied Write/WriteLine: WriteLine fügt newline hinzu.
- Methodenaufruf-Syntax: Name(Argumente); mehrere Argumente möglich.
- Compiler-Hinweise: Compiler-Fehler geben meist Zeilennummer und Fehlertyp an — lesen!

„Hello World“ – Klassische Form

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Details:

- using System; bindet Namensraum, damit Console ohne Qualifizierung nutzbar ist.
- class Program: Container für Methoden; Einstiegspunkt ist die Main-Methode.
- static void Main(string[] args): Signatur des Programms; args enthält Kommandozeilenargumente.
- Top-Level-Statements: In modernen C#-Versionen kann Main weggelassen werden — Sinn: weniger Boilerplate für Einsteiger.
- Praxishinweis: Beim Debuggen ist es hilfreich, Breakpoints in Main zu setzen.

Erste Praxis: IDE Setup mit GitHub Codespaces

Template-Link: https://github.com/Rearth/DHBW-Vorlesung-Codespace

Details:

- Codespace enthält vorkonfiguriertes Projekt, damit alle dieselbe Umgebung nutzen.
- Voraussetzungen: GitHub-Account, kann gratis erstellt werden.

Erste Praxis: IDE Setup mit GitHub Codespaces

- Codespace öffnen
- `Program.cs` auswählen
- Grünen Knopf → „Run“ klicken

Details:

- Nach Start: ggf. Build-Prozess im Terminal beobachten.
- Wenn Ausgabe nicht erscheint: sicherstellen, dass die richtige Konfiguration (Debug/Release) und das richtige Projekt ausgewählt ist.
- Änderungen live testen: Datei speichern, Wiederholen des Runs.
- Co-Pilot chat verfügbar, kann optional benutzt werden. Idealerweise bei Eigenarbeit zuhause zum erklären und helfen bei problemen. KI-Tools sollten nicht zum lösen der kompletten Aufgaben genutzt werden. In der Prüfung sind diese nicht verfügbar.

Codespaces: Erste Änderungen

- Ändern Sie die Ausgabe, z. B.:

```
Console.WriteLine("Mein erstes C#-Programm!");
```

- Programm starten und Unterschied beobachten

Details:

- Ziel: Kleine Änderungen vornehmen, kompilieren und das Ergebnis prüfen.
- Tipp: Klein anfangen — eine Zeile ändern reicht zum Vertraut werden mit Editieren/Run.

Variablen

- Speicher für Daten im Programm
- Zuweisung mit `=`
- Schlüsselwort `var` für automatische Typableitung

Details:

- Lebensdauer: Variable existiert nur im Gültigkeitsbereich (Scope), z. B. innerhalb einer Methode oder eines Blocks.
- Initialisierung: Eine Variable sollte vor Verwendung initialisiert werden.
- var ≠ dynamisch: var ist statisch typisiert; der Typ wird zur Compilezeit bestimmt.

Beispiel: Variablen nutzen

```
var zahl1 = 5;
var zahl2 = 7;
var summe = zahl1 + zahl2;
Console.WriteLine($"Summe: {summe}");
```

Details:

- Typinferenz: zahl1/zahl2 sind ints; summe ist int.
- String-Interpolation ($"..."): Variablen direkt einfügen; gute Lesbarkeit.
- Fehlerfälle: Bei gemischten Typen (z. B. int + double) wird Typkonversion angewandt — bewusst sein, wenn Genauigkeit wichtig ist.

Primitive Datentypen in C#

- `int` – Ganzzahlen
- `float` – Kommazahlen
- `bool` – Wahr/Falsch
- `string` – Zeichenketten

Details:

- Genauigkeit: float weniger genau als double;
- Größenordnung: int für übliche Zählwerte; long für sehr große Werte.
- bool nur true/false — keine numerische Interpretation wie in manchen Sprachen.

Primitive Datentypen in C# - Ergänzungen

- `short`, `byte`, `long` – Ganzzahlen
- `double` – Kommazahlen
- `char` – Zeichen

Details:

- byte in C# ist unsigned (0..255).
- char repräsentiert ein UTF-16 Codeunit; für einzelne Zeichen geeignet, für Texte string verwenden.

Datentypen im Detail

Type	Vorzeichen	Größe	Wertebereich
byte	ja	8 bit	0 to 2^8 - 1 (0…255)
short	ja	16 bit	-2^{15} to 2^{15} - 1 (-32768…32767)
int	ja	32 bit	-2^{31} to 2^{31} - 1 (-2147483648…2147483647)
long	ja	64 bit	-2^{63} to 2^{63} - 1 (-9223372036854775808…9223372036854775807)
char	nein	16 bit	16-Bit Unicode Character (0x0000…0xffff (65535_{10}))
float	ja	32 bit (V:1 bit, E:8 bit, M:23 bit)	$-3.40282347 \times 10^{38}$ to $3.40282347 \times 10^{38}$
double	ja	64 bit (V:1 bit, E:11 bit, M:52 bit)	$-1.79769313486231570 \times 10^{308}$ to $1.79769313486231570 \times 10^{308}$
boolean	-	8 bit	true/false

Details:

- Wertebereiche sind nützlich zur Entscheidung, welcher Typ passt (Speicher vs. Bedarf).
- Floating-Point: Achtung Rundungsfehler; bei Vergleich von doubles Toleranz verwenden.

Datentypen definieren

Ist ein Datentyp nicht eindeutig definiert (z.b. bei Nutzung von `var` oder bei Funktionsaufrufen), werden Daten wie folgt interpretiert:

- `"Inhalt"` -> String, durch die `"` erkannt.
- `'a'` -> Char. Das `'` erlaubt nur die Eingabe eines einzelnen Buchstabens
- `123` -> Int. Eine Zahl ohne Komma (beim Programmieren immer ein `.`) wird als int per default erkannt.
- `123.45` -> Double.
- `true` / `false` -> boolean

Details:

- Dezimaltrennzeichen in C# ist der Punkt; lokale Darstellung der Konsole kann Kommas anzeigen.
- Literaltypen können mit Suffixen explizit gemacht werden (siehe nächste Folie).
- Bei großen Ganzzahlen: Numeric separators (1_000_000) machen Zahlen lesbarer.

Spezifische Datentypen definieren

Soll ein bestimmter Variablentyp im Programmcode genutzt werden, können folgende suffixe genutzt werden:

- `123L` -> Long
- `123.45F` -> Float
- `123.45D` -> Double

Details:

- L/l für long, F/f für float, D/d optional für double.
- Groß-/Kleinschreibung bei Suffixen ist erlaubt (z. B. 123L oder 123l).

Variablen ausgeben

Strings können variablen beinhalten, wenn sie mit einem $ prefixed werden.

Beispiel:

```
var summe = 4 + 5;
Console.WriteLine($"Summe: {summe}");
```

Details:

- String-Interpolation erlaubt auch Format-Spezifizierer: $"{wert:0.00}".
- Alternative: String.Format oder concatenation (+), aber Interpolation ist lesbarer.

Beispielprogramm: Datentypen

```
int alter = 21;
double pi = 3.14159;
float temperatur = 36.6f;
char buchstabe = 'A';
bool istStudent = true;
string name = "Max";

Console.WriteLine($"Alter: {alter}");
Console.WriteLine($"Pi: {pi}");
Console.WriteLine($"Temperatur: {temperatur}");
Console.WriteLine($"Buchstabe: {buchstabe}");
Console.WriteLine($"Ist Student: {istStudent}");
Console.WriteLine($"Name: {name}");
```

Details:

- Ausgabeformat: Bei double/float Rundungsdarstellung beachten; Formatierung nach Bedarf.
- Typen bewusst wählen: float mit f-Literal markieren.
- Namen und Konvention: Variablennamen sollten selbsterklärend sein.

---

const und readonly

- `const`: Wert muss zur Compile-Zeit feststehen, ändert sich nie
- `readonly`: Kann im Konstruktor gesetzt werden, bleibt danach unveränderlich

Details:

- const ist statisch und inlined vom Compiler — bei Änderung in Bibliothek Recompile nötig.
- readonly erlaubt Laufzeitinitialisierung im Konstruktor, nützlich für abhängige Werte.
- Verwendung: const für mathematische Konstanten, readonly für konfigurierbare Werte, die nach Konstruktion fest sind.

Operatoren in C#

- **Arithmetisch**: `+`, `-`, `*`, `/`, `%`
- **Zuweisung**: `=`, `+=`, `-=`, `*=`, `/=`
- **Vergleich**: `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Logisch**: `&&`, `||`, `!`
- **Inkrement/Dekrement**: `++`, `--`

Details:

- Operatorpriorität beachten; bei Unsicherheit Klammern verwenden.
- Ganzzahlige Division rundet ab (z. B. 5/2 == 2).
- Kurzschlussverhalten: && und || führen zu bedingter Auswertung — Seiteneffekte beachten.

Operatoren Beispiele - Arithmetisch

```
int a = 10;
int b = 3;

Console.WriteLine($"a + b = {a + b}");    // = 13
Console.WriteLine($"a - b = {a - b}");    // = 7
Console.WriteLine($"a * b = {a * b}");    // = 30
Console.WriteLine($"a / b = {a / b}");    // = 3 (ganzzahl)
Console.WriteLine($"(double)a / b = {(double)a / b}");    // = 3.333
Console.WriteLine($"a % b = {a % b}");    // 1
```

Operatoren Beispiele - Zuweisungen

```
int x = 5;
x += 2; // x = x + 2
Console.WriteLine($"x nach +=2: {x}");    // = 7
x *= 3; // x = x * 3
Console.WriteLine($"x nach *=3: {x}");    // = 21
```

Operatoren Beispiele - Vergleich

```
int a = 10;
int b = 3;
Console.WriteLine($"a == b: {a == b}");    // false
Console.WriteLine($"a != b: {a != b}");    // true
Console.WriteLine($"a > b: {a > b}");      // true
Console.WriteLine($"a <= b: {a <= b}");    // false
```

Operatoren Beispiele - Logisch

```
int a = 10;
int b = 3;
bool cond1 = (a > b) && (b > 0); // true (und)
bool cond2 = (a < b) || (b > 0); // true (oder)
Console.WriteLine($"(a > b) && (b > 0): {cond1}");
Console.WriteLine($"(a < b) || (b > 0): {cond2}");
Console.WriteLine($"! (a == b): {!(a == b)}");    // true
```

Operatoren Beispiele - Inkrement / Dekrement

```
int i = 0;
Console.WriteLine($"i = {i}");      // 0
Console.WriteLine($"i++: {i++}");   // 0 (da das ++ danach evaluiert wird)
Console.WriteLine($"i: {i}");       // 1
i = 0;
Console.WriteLine($"++i: {++i}");   // 1
```

Interaktionen

- Benutzereingaben können mit `Console.ReadLine();` gelesen werden
- Wird immer als string eingelesen.
- Konvertierungen zu gängigen typen:
- Integer: `Convert.ToInt32(alterAlsText);`
- Float: `Convert.ToSingle("41.00027357");`

Details:

- Console.ReadLine() kann null zurückgeben (z. B. bei EOF) — prüfen.
- Kulturabhängigkeit: Dezimaltrennzeichen kann kulturell variieren — bei Bedarf CultureInfo verwenden.

Interaktion: Benutzereingaben lesen

```
Console.WriteLine("Bitte gib deinen Namen ein:");
string name = Console.ReadLine();
Console.WriteLine($"Hallo, {name}!");

Console.WriteLine("Bitte gib dein Alter ein:");
string alterAlsText = Console.ReadLine();
int alter = Convert.ToInt32(alterAlsText);
Console.WriteLine($"Du bist {alter} Jahre alt.");
```

Wichtig: Typumwandlung (Parsing) von `string` → `int`

Details:

- Robustheit: Nutze int.TryParse, um Abstürze bei ungültiger Eingabe zu vermeiden.
- Validierung: Negative oder unrealistische Werte abfangen.
- Benutzerfreundlichkeit: Fehlermeldungen geben und erneute Eingabe erlauben.

Kontrollstrukturen: if-else

```
int alter = 18;

if (alter >= 18) {
    Console.WriteLine("Volljährig");
} else {
    Console.WriteLine("Noch minderjährig");
}
```

Details:

- Bedingungen müssen bool zurückliefern.
- Klammern immer verwenden (auch bei einzeiligen Blöcken) für Lesbarkeit und Fehlervermeidung.
- Reihenfolge der Bedingungen kann Logik und Performance beeinflussen.

Verschachtelte Bedingungen

```
int note = 2;

if (note == 1) {
    Console.WriteLine("Sehr gut");
}
else if (note == 2) {
    Console.WriteLine("Gut");
} else {
    Console.WriteLine("Andere Note");
}
```

Details:

- else-if Ketten sind klar lesbar bis zu einer gewissen Komplexität; bei vielen Fällen switch oder Lookup-Struktur erwägen.
- Vergleich mit == bei Referenzen: bei Objekten auf Equals/GetHashCode achten (hier primitive Typen, daher unproblematisch).

`switch`-Struktur

```
int wochentag = 3;

switch (wochentag) {
    case 1:
        Console.WriteLine("Montag");
        break;
    case 2:
        Console.WriteLine("Dienstag");
        break;
    case 3:
        Console.WriteLine("Mittwoch");
        break;
    default:
        Console.WriteLine("Anderer Tag");
        break;
}
```

Details:

- Fall-through in C# ist nur mit expliziter Goto/Pattern möglich; ansonsten break erforderlich.
- Moderne C#-Versionen unterstützen Pattern Matching im switch — flexiblere Vergleiche möglich.

Guter Stil: Lesbarer Code

- Variablennamen in **camelCase** (z. B. `meinAlter`)
- Methoden- und Klassennamen in **PascalCase** (z. B. `BerechneSumme`)
- Aussagekräftige Namen verwenden
- Kommentare nutzen: `// Erklärung`
- Saubere Einrückung für bessere Lesbarkeit
- Typischerweise auf English

Details:

- Ein konsistenter Stil vereinfacht Teamarbeit und Wartung.
- Kurze, aussagekräftige Methoden bevorzugen — Single Responsibility Principle.
- Kommentare für warum, nicht was (Code sollte das "was" größtenteils ausdrücken).

Typische Stolpersteine

- Vergessenes `;`
- Groß-/Kleinschreibung beachten (`alter` ≠ `Alter`)
- Zuweisung (`=`) vs. Vergleich (`==`)
- Strings in `"..."`, chars in `'...'`

Details:

- Compiler- und Laufzeitfehler unterscheiden — beide sorgfältig lesen.
- Unit-Tests und kleine Experimente helfen, Annahmen zu überprüfen.
- Häufige Fehlerquellen: NullReferenceException — auf null prüfen oder Null-conditional operator verwenden.

Was ist C# und .NET?

- C# = Programmiersprache, entworfen von Microsoft
- .NET = Plattform/Framework, Laufzeit und Bibliotheken
- Ziel: Produktive Entwicklung, starke Typisierung, breite Bibliotheksunterstützung

Details:

- C# ist die Sprache; .NET stellt Laufzeit, Basisklassen (BCL) und Tools.
- .NET ist cross-platform (.NET Core / .NET 5+) — läuft auf Windows, Linux, macOS.
- Typische Anwendungsfälle: Desktop (WinForms/WPF), Web (ASP.NET), Spiele (Unity), Cloud/Server.
- Wichtig: Konzepte (Typen, GC, Assemblies) wiederholen sich in allen Umgebungen.

Vom C#-Code zum Bytecode (IL)

- C#-Quellcode → Compiler (Roslyn) → Intermediate Language (IL) / MSIL
- IL ist plattformunabhängiger Zwischencode mit Metadaten in Assemblies (.dll/.exe)
- Assemblies enthalten Typinformationen, Referenzen und Ressourcen

Details:

- Roslyn erzeugt IL und fügt Metadaten hinzu (Signaturen, Attributes).
- IL ist nicht direkt maschinenspezifisch — ermöglicht Portabilität zwischen Implementierungen (.NET Runtime, Mono).
- Assemblies können signiert (Strong Name) oder paketiert (NuGet) geliefert werden.
- Für Debugging: PDB-Dateien enthalten Symbol-Informationen.

Die .NET-Laufzeit (CLR / .NET Runtime)

- Runtime stellt Ausführungsumgebung bereit: Typprüfung, Sicherheitsmodelle, Garbage Collector
- JIT-Compiler wandelt IL zur Laufzeit in nativen Maschinencode um
- Verschiedene Implementierungen: CoreCLR / RyuJIT, Mono, .NET Native / AOT-Varianten

Details:

- Laufzeit sorgt für Speicherverwaltung, Typensicherheit, Exception-Handling und Interoperabilität.
- JIT erzeugt performanten nativen Code zur Laufzeit; moderne Runtimes nutzen Tiered Compilation.
- AOT-Optionen (z. B. Native AOT) übersetzen IL vorab, nützlich für Startzeit/Deploy ohne JIT.
- Unterschiede zwischen Implementierungen können Verhalten und Performance beeinflussen.

Tools und Ecosystem: Compiler, SDK, CLI

- Roslyn = moderner C#-Compiler + APIs für Analyse/Refactoring
- dotnet CLI, MSBuild, NuGet (Paketverwaltung) steuern Build & Dependencies
- IDEs/Editoren: Visual Studio, VS Code (+ C#-Extension), Rider

Details:

- dotnet SDK enthält Compiler, Runtime-Verweise, Templates und CLI-Befehle (build, run, publish).
- MSBuild orchestriert Kompilierung, Resource-Embedding und Packaging.
- NuGet erlaubt Wiederverwendung von Bibliotheken; Versionierung und Transitive Abhängigkeiten beachten.
- Roslyn erlaubt Code-Analyse/Code-Fixes — nützlich für Linting und Refactoring.

Unter der Haube: Garbage Collector & JIT/AOT

- Garbage Collector (GC): automatische Speicherbereinigung, Generationenmodell (0/1/2)
- JIT wandelt IL in nativen Code bei Bedarf; Tiered JIT und Inline-Optimierungen verbessern Performance
- AOT (Ahead-of-Time) kompiliert vorab für bessere Startzeiten / kleinere Overhead-Fälle

Details:

- GC-Familien: Workstation vs Server GC; Auswirkungen auf Latenz und Durchsatz.
- Generational GC optimiert für kurze Lebenszeit vieler Objekte — typische App-Pattern berücksichtigen.
- JIT bietet Profil-basierte Optimierungen; Tiered JIT kompiliert zuerst schnell, optimiert später.
- AOT reduziert JIT-Overhead, kann aber größere Binaries oder eingeschränkte Reflection verursachen.
- Debugging/Performance: Profiling-Tools (dotnet-trace, dotnet-counters, Visual Studio Profiler) verwenden.

Zusammenfassung

Heute gelernt:

- Erste C#-Programme ausführen
- Variablen und Datentypen verwenden
- Operatoren und Benutzereingaben
- Kontrollstrukturen und Schleifen

Details:

- Fokus auf Verständnis der Grundlagen statt perfekter Syntax.
- Empfohlene nächste Schritte: kleine Tasks zuhause wiederholen, Dokumentation (Microsoft Docs) konsultieren.
- Fehler sind Lernchance — Debugging lernen.