Informationstechnologie I - Skript zu Termin 3: Funktionen

1. Einführung

Methoden bilden das Rückgrat von sauberem, strukturiertem und effizientem Code. Sie erlauben es, komplexe Probleme in kleinere, handhabbare Teile zu zerlegen und Code zu schreiben, der nicht nur funktioniert, sondern auch leicht zu lesen, zu warten und wiederzuverwenden ist. Das Prinzip "Don't Repeat Yourself" (DRY) steht dabei im Fokus.

2. Was sind Methoden und warum werden sie benötigt?

Betrachten wir ein typisches Programmierproblem: Eine bestimmte Berechnung muss mehrfach mit unterschiedlichen Werten durchgeführt werden. Ein klassisches Beispiel ist die Berechnung der Fläche eines Kreises.

Ohne Methoden könnte der Code wie folgt aussehen:

```
// An einer Stelle im Programm
double radius1 = 5.0;
double area1 = 3.14159 * radius1 * radius1;
Console.WriteLine(area1);

// An einer ganz anderen Stelle, viel später im Code...
double radius2 = 10.0;
double area2 = 3.14159 * radius2 * radius2;
Console.WriteLine(area2);
```

Dieser Ansatz führt zu mehreren Problemen:

- **Code-Wiederholung (Duplizierung):** Die Berechnungslogik wird kopiert und eingefügt. Dies ist ineffizient und eine häufige Fehlerquelle. Eine Änderung der Genauigkeit von Pi (z.B. durch die Verwendung von `Math.PI`) müsste an jeder einzelnen Stelle im Code manuell durchgeführt werden.
- **Schlechte Lesbarkeit:** Die Absicht hinter der Formel ist nicht sofort ersichtlich.
- **Schwierige Wartbarkeit:** Fehlerbehebungen oder Anpassungen müssen an vielen Stellen wiederholt werden, was zeitaufwendig und fehleranfällig ist.

Die Lösung: Methoden!

Eine **Methode** ist ein benannter Codeblock, der eine spezifische Aufgabe erledigt. Dieser Block kann jederzeit über seinen Namen "aufgerufen" und ihm können die notwendigen Daten zur Verarbeitung mitgegeben werden.

Das Kreisflächen-Beispiel **mit einer Methode**:

```
// 1. Die Methode definieren (die "Anleitung" schreiben)
double CalculateCircleArea(double radius)
{
  return Math.PI * radius * radius;
}

// 2. Die Methode aufrufen (die "Anleitung" benutzen)
double area1 = CalculateCircleArea(5.0);
Console.WriteLine(area1);

double area2 = CalculateCircleArea(10.0);
Console.WriteLine(area2);
```

Eine Methode lässt sich mit einem Rezept vergleichen: Es wird einmal definiert, wie eine Aufgabe erledigt wird (die Methode). Immer wenn das Ergebnis benötigt wird, wird die Methode aufgerufen und die "Zutaten" (die **Parameter**) werden mitgegeben. Das Resultat ist der **Rückgabewert**.

Die Vorteile sind wie folgt:

- **Wiederverwendbarkeit:** Einmal geschrieben, kann die Methode überall im Programm verwendet werden.
- **Lesbarkeit:** Ein Aufruf wie `CalculateCircleArea(5.0)` ist selbsterklärend.
- **Wartbarkeit:** Änderungen an der Logik müssen nur an einer einzigen, zentralen Stelle vorgenommen werden.
- **Struktur:** Programme werden in logische, voneinander unabhängige Einheiten gegliedert, was die Übersichtlichkeit erhöht.

3. Aufbau einer Methode

Eine Methode in C# folgt einer klaren Struktur. Die allgemeine Syntax lautet:

```
[Rückgabetyp] [Methodenname]([Parameterliste])
{
    // Methoden-Körper: Der Code, der ausgeführt wird.
    return [Wert]; // Nur notwendig, wenn der Rückgabetyp nicht 'void' ist.
}
```

Die Bestandteile sind:

- `Rückgabetyp`: Definiert den Datentyp des Ergebnisses (z.B. `int`, `string`, `bool`). Führt eine Methode nur eine Aktion aus, ohne Daten zurückzugeben, wird der Rückgabetyp `void` (englisch für "leer") verwendet.
- `Methodenname`: Ein aussagekräftiger Name, der die Funktion der Methode beschreibt. Nach Konvention werden Methodennamen in C# in **PascalCase** geschrieben (z.B. `CalculateSum`).
- `Parameterliste`: Eine kommaseparierte Liste von Variablen (Parametern), die die Methode als Eingabewerte benötigt. Jeder Parameter besteht aus einem Datentyp und einem Namen. Benötigt eine Methode keine Eingabewerte, bleiben die Klammern leer: `()`.
- `Methoden-Körper`: Der in geschweiften Klammern `{}` eingeschlossene Code, der bei Aufruf der Methode ausgeführt wird.
- `return`**-Anweisung**: Beendet die Ausführung der Methode und gibt einen Wert an den Aufrufer zurück. Der Datentyp des Wertes muss mit dem deklarierten `Rückgabetyp` übereinstimmen. Beim Rückgabetyp `Void` wird einfach nur `return;` ohne variable genutzt.

In den bisher betrachteten Konsolenanwendungen (Top-Level-Statements) werden Methoden als **lokale Funktionen** definiert. Diese benötigen keine zusätzlichen Schlüsselwörter wie `public` oder `static`.

Gültigkeitsbereiche (Scopes) und Variablennamen

Ein wichtiges Konzept im Zusammenhang mit Methoden ist der **Gültigkeitsbereich (Scope)**. Variablen, die innerhalb einer Methode deklariert werden – einschließlich ihrer Parameter – sind **lokal**. Sie existieren nur innerhalb dieser Methode und sind von außen nicht zugänglich.

Der Name einer Variable, die an eine Methode übergeben wird, muss **nicht** mit dem Namen des Parameters in der Methodendefinition übereinstimmen.

```
void PrintDouble(int value) // Der Parameter hier heißt 'value'.
{
    // 'value' ist eine lokale Variable, die nur innerhalb von PrintDouble existiert.
    Console.WriteLine(value * 2);
}

int myNumber = 10; // Diese Variable heißt 'myNumber'.
PrintDouble(myNumber); // Hier wird der *Wert* von myNumber (also 10) an die Methode übergeben.
```

Im Beispiel existiert `myNumber` nur im Hauptprogramm, während `value` nur in der Methode `PrintDouble` existiert. Beim Aufruf der Methode wird der Wert von `myNumber` in den Parameter `value` kopiert. Jede Methode verfügt somit über einen eigenen, abgeschlossenen Arbeitsbereich.

Beispiel 1: Eine einfache `void`-Methode

Eine `void`-Methode führt eine Aktion aus, ohne ein Ergebnis zurückzugeben.

```
// Definition der Methode
void SayHello()
{
    Console.WriteLine("Hallo Welt!");
}

// Aufruf der Methode
SayHello(); // Gibt "Hallo Welt!" aus.
SayHello(); // Gibt erneut "Hallo Welt!" aus.
```

Hier zeigt sich die Trennung von Definition (was die Methode tun soll) und Aufruf (die tatsächliche Ausführung).

Beispiel 2: Eine Methode mit Rückgabewert

Diese Methode nimmt zwei Zahlen entgegen, berechnet deren Summe und gibt das Ergebnis zurück.

```
// Definition: Die Methode deklariert, einen 'int' zurückzugeben.
int Add(int a, int b)
{
    int sum = a + b;
    return sum; // Die 'return'-Anweisung beendet die Methode und liefert den Wert.
}

// Aufruf und Speicherung des Ergebnisses in einer Variable
int result = Add(5, 3); // 'result' hat jetzt den Wert 8.
Console.WriteLine(result);

// Direkte Verwendung des Rückgabewertes in einem anderen Aufruf
Console.WriteLine(Add(10, 20)); // Gibt direkt 30 aus.
```

Sobald eine `return`-Anweisung erreicht wird, wird die Methode sofort beendet und die Kontrolle an den aufrufenden Code zurückgegeben.

4. Parameterübergabe im Detail

Die Art der Datenübergabe an eine Methode ist ein entscheidendes Detail. In C# gibt es hierfür verschiedene Mechanismen.

Call by Value (Standard)

Standardmäßig werden Parameter in C# **"by value"** (als Wert oder Kopie) übergeben. Das bedeutet:

- Die Methode erhält eine **Kopie** des Wertes der übergebenen Variable.
- Änderungen, die am Parameter **innerhalb** der Methode vorgenommen werden, haben **keine Auswirkung** auf die ursprüngliche Variable außerhalb der Methode.

Dies ist vergleichbar mit der Übergabe einer Fotokopie eines Dokuments. Änderungen auf der Kopie beeinflussen das Original nicht.

```
void TryToChange(int number)
{
    number = 100; // Diese Änderung betrifft nur die lokale Kopie 'number'.
    Console.WriteLine($"Innerhalb der Methode: {number}"); // Ausgabe: 100
}

int myValue = 5;
TryToChange(myValue);
Console.WriteLine($"Außerhalb der Methode: {myValue}"); // Ausgabe: Immer noch 5!
```

Call by Reference mit `ref`

Um einer Methode zu erlauben, die ursprüngliche Variable zu verändern, wird das Schlüsselwort `ref` verwendet.

- Die Methode erhält eine **Referenz** (also die Speicheradresse) auf die ursprüngliche Variable.
- Änderungen am Parameter **innerhalb** der Methode wirken sich direkt auf die Variable **außerhalb** aus.
- Die Variable **muss** vor dem Aufruf einen Wert zugewiesen bekommen haben (initialisiert sein).

Dies ist vergleichbar mit der Übergabe des Originaldokuments.

```
void Swap(ref int a, ref int b)
{
    Console.WriteLine($"Vor dem Tausch in der Methode: a={a}, b={b}");
    int temp = a; // Wert von a zwischenspeichern
    a = b;
    b = temp;
    Console.WriteLine($"Nach dem Tausch in der Methode: a={a}, b={b}");
}

int x = 5;
int y = 10;
Console.WriteLine($"Vor dem Aufruf: x={x}, y={y}");
Swap(ref x, ref y); // Das 'ref' Schlüsselwort ist auch beim Aufruf erforderlich!
Console.WriteLine($"Nach dem Aufruf: x={x}, y={y}"); // x ist jetzt 10, y ist 5!
```

Parameterübergabe mit `out`

Das `out`-Schlüsselwort ist `ref` sehr ähnlich, hat aber eine andere Absicht und leicht abweichende Regeln:

- Es wird ebenfalls eine **Referenz** auf die Variable übergeben.
- Die Methode **verpflichtet** sich, diesem `out`-Parameter einen Wert zuzuweisen, bevor sie endet.
- Im Gegensatz zu `ref` **muss** die Variable vor dem Aufruf **nicht** initialisiert sein.

`out`-Parameter sind nützlich, wenn eine Methode mehrere Werte zurückgeben soll. Diese können als leere Behälter betrachtet werden, die an die Methode übergeben werden, um von dieser gefüllt zu werden.

```
// Diese Methode "gibt" zwei Werte über 'out'-Parameter zurück.
void GetMinMax(int a, int b, out int min, out int max)
{
    if (a < b)
    {
        min = a;
        max = b;
    }
    else
    {
        min = b;
        max = a;
    }
    // Die Methode hat ihre Pflicht erfüllt und beiden out-Parametern einen Wert zugewiesen.
}

int number1 = 10;
int number2 = 20;
int minimum; // Muss nicht initialisiert werden
int maximum;

GetMinMax(number1, number2, out minimum, out maximum);

Console.WriteLine($"Die kleinere Zahl ist {minimum}, die größere ist {maximum}.");
```

Ein bekannter Anwendungsfall ist die Methode `int.TryParse()`, die versucht, einen String in eine Zahl umzuwandeln und sowohl einen `bool` (Erfolg?) als auch den umgewandelten `int` (über `out`) zurückgibt.

`ref` vs. `out` vs. `in` – Eine Zusammenfassung

Schlüsselwort	Zweck	Initialisierung nötig?	Wertänderung in Methode?
(keins)	Wert übergeben (Kopie)	Ja	Nur die Kopie ändert sich
`ref`	Wert lesen **und** ändern	**Ja**	Ja, ändert Original
`out`	Wert zurückgeben (initialisieren)	**Nein**	**Muss** Wert zuweisen
`in`	Wert nur lesen (schreibgeschützt)	Ja	**Nein**, nicht erlaubt

Das `in`-Schlüsselwort ist eine Optimierung für fortgeschrittene Szenarien. Es verhält sich wie `ref` (vermeidet das Kopieren großer Datenstrukturen), garantiert aber, dass die Methode den übergebenen Wert nicht verändern kann.

5. Methodenüberladung

Angenommen, eine Methode soll Werte addieren. Diese Werte können `int`, `double` oder es können drei statt zwei Zahlen sein. Hierfür müssen keine separaten Methoden wie `AddInts`, `AddDoubles` und `AddThreeInts` geschrieben werden.

Methodenüberladung (Overloading) erlaubt es, mehrere Methoden mit dem **gleichen Namen** zu definieren, solange sich ihre **Parameterlisten** unterscheiden. Der Compiler wählt anhand der beim Aufruf übergebenen Argumente die korrekte Version der Methode aus.

Die Methoden müssen sich in einem der folgenden Punkte unterscheiden:

- Anzahl der Parameter
- Datentyp der Parameter
- Reihenfolge der Datentypen der Parameter

Achtung: Der Rückgabetyp allein ist **kein** gültiges Unterscheidungsmerkmal für eine Überladung.

```
// Version 1: Addiert zwei Integer
int Add(int a, int b)
{
    Console.WriteLine("Integer-Version wurde aufgerufen");
    return a + b;
}

// Version 2: Addiert zwei Doubles (gleicher Name, andere Parametertypen)
double Add(double a, double b)
{
    Console.WriteLine("Double-Version wurde aufgerufen");
    return a + b;
}

// Version 3: Addiert drei Integer (gleicher Name, andere Parameteranzahl)
int Add(int a, int b, int c)
{
    Console.WriteLine("Drei-Integer-Version wurde aufgerufen");
    return a + b + c;
}

// Aufrufe:
Add(2, 3);       // Ruft Version 1 auf
Add(2.5, 3.5);   // Ruft Version 2 auf
Add(1, 2, 3);    // Ruft Version 3 auf
```

Methodenüberladung vereinfacht die Nutzung einer API, da nur ein Methodenname für eine Familie von ähnlichen Operationen erforderlich ist.

6. Rekursion

Eine Methode, die sich selbst aufruft, wird als **rekursive Methode** bezeichnet. Rekursion ist ein elegantes Konzept zur Lösung von Problemen, die sich in kleinere, gleichartige Teilprobleme zerlegen lassen.

Jede funktionierende rekursive Methode benötigt zwingend zwei Bestandteile:

1. Einen **Basisfall (oder Anker):** Eine Bedingung, die die Kette der Selbstaufrufe beendet. Ohne einen Basisfall würde sich die Methode unendlich oft selbst aufrufen, was zu einem `StackOverflowException` führt, da der für Methodenaufrufe reservierte Speicher (der "Stack") überläuft.
2. Einen **rekursiven Schritt:** Der Aufruf der Methode selbst, jedoch mit einem modifizierten Argument, das das Problem dem Basisfall einen Schritt näher bringt.

Beispiel: Die Fakultät berechnen

Die Fakultät einer Zahl `n` (geschrieben als `n!`) ist das Produkt aller ganzen Zahlen von 1 bis `n`. Beispiel: `5! = 5 * 4 * 3 * 2 * 1 = 120`. Die rekursive Definition lautet: `n! = n * (n-1)!`.

```
int Factorial(int n)
{
    // 1. Basisfall: Die Fakultät von 1 (oder 0) ist 1. Hier endet die Rekursion.
    if (n <= 1)
    {
        return 1;
    }
    // 2. Rekursiver Schritt: Das Problem wird auf eine kleinere Version reduziert.
    else
    {
        return n * Factorial(n - 1);
    }
}

// Was passiert bei Factorial(4)?
// 1. Aufruf: Factorial(4) -> gibt 4 * Factorial(3) zurück
// 2. Aufruf: Factorial(3) -> gibt 3 * Factorial(2) zurück
// 3. Aufruf: Factorial(2) -> gibt 2 * Factorial(1) zurück
// 4. Aufruf: Factorial(1) -> trifft den Basisfall und gibt 1 zurück.

// Die Ergebnisse werden nun von innen nach außen aufgelöst:
// -> 2 * 1 = 2
// -> 3 * 2 = 6
// -> 4 * 6 = 24
```

Rekursion kann für bestimmte Probleme (z.B. das Durchlaufen von Baumstrukturen) die natürlichste und lesbarste Lösung sein.

Rekursion vs. Schleifen: Wann was verwenden?

Sowohl Rekursion als auch iterative Schleifen (wie `for` oder `while`) können zur Wiederholung von Operationen verwendet werden. Jedes rekursive Problem lässt sich auch iterativ lösen und umgekehrt. Die Wahl hängt oft von der Problemstellung, der Lesbarkeit und der Effizienz ab.

Wann ist Rekursion eine gute Wahl?

1. **Bei von Natur aus rekursiven Problemen:** Viele Algorithmen, insbesondere in der Datenstrukturverarbeitung, sind rekursiv definiert. Beispiele sind:
   - Das Durchlaufen von Baumstrukturen (z.B. Dateisysteme, HTML-DOM).
   - Sortieralgorithmen wie Quicksort oder Mergesort.
   - Mathematische Probleme mit rekursiver Definition (Fakultät, Fibonacci-Folge).
2. **Für mehr Lesbarkeit und Eleganz:** Bei komplexen Problemen kann eine rekursive Lösung kürzer und verständlicher sein als eine komplizierte iterative Lösung, die einen eigenen Stack verwalten muss.

Wann sind Schleifen (Iteration) zu bevorzugen?

1. **Performance:** Iteration ist in der Regel schneller. Jeder rekursive Aufruf erzeugt zusätzlichen Verwaltungsaufwand (Speichern des aktuellen Zustands auf dem Call Stack). Bei sehr vielen Wiederholungen kann dies die Ausführung verlangsamen.
2. **Speicherverbrauch:** Rekursion verbraucht mehr Speicher. Für jeden Aufruf wird ein neuer Eintrag auf dem Stack angelegt. Bei sehr tiefer Rekursion (z.B. `Factorial(100000)`) droht ein `StackOverflowException`, da der verfügbare Stack-Speicher erschöpft ist. Eine Schleife benötigt hingegen nur konstanten Speicher.
3. **Einfache, lineare Probleme:** Für einfache Wiederholungen, wie das Durchlaufen eines Arrays von Anfang bis Ende, ist eine Schleife fast immer die einfachere und direktere Lösung.

Vergleichstabelle:

Aspekt	Rekursion	Iteration (Schleife)
Definition	Eine Funktion ruft sich selbst auf.	Ein Codeblock wird wiederholt, solange eine Bedingung gilt.
Zustandsende	Erreicht einen **Basisfall**.	Die **Schleifenbedingung** wird `false`.
Performance	Langsamer aufgrund von Funktionsaufruf-Overhead.	Schneller, da kein Overhead entsteht.
Speicher	Verbraucht mehr Speicher (Call Stack).	Verbraucht wenig, konstanten Speicher.
Code-Länge	Oft kürzer und eleganter.	Kann länger und umständlicher sein.
Anwendungsfälle	Baumtraversierung, Divide-and-Conquer, Graphen.	Lineare Datenstrukturen, einfache Wiederholungen.

Fazit: Wählen Sie Rekursion, wenn sie die Komplexität eines Problems reduziert und zu saubererem Code führt. Seien Sie sich aber der potenziellen Performance- und Speichernachteile bewusst. Für die meisten alltäglichen, linearen Probleme ist eine Schleife die sicherere und effizientere Wahl.

7. Zusammenfassung

Die wichtigsten Erkenntnisse lassen sich wie folgt zusammenfassen:

- **Methoden** sind essenziell für strukturierten Code und bündeln Anweisungen zu wiederverwendbaren Einheiten.
- Sie verbessern die **Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit** von Code.
- Jede Methode wird durch ihren **Namen, Rückgabetyp und ihre Parameterliste** definiert.
- Ein Rückgabetyp von `void` signalisiert, dass eine Methode eine Aktion ausführt, aber kein Ergebnis zurückliefert.
- Bei der **Parameterübergabe** ist der Standard **Call by Value** (eine Kopie). Mit `ref` und `out` können Referenzen übergeben werden, um Originalvariablen zu modifizieren (`ref`) oder zu initialisieren (`out`).
- **Methodenüberladung** ermöglicht es, denselben Methodennamen für unterschiedliche Parameterkonstellationen zu verwenden.
- **Rekursion** ist eine Technik, bei der sich eine Methode selbst aufruft. Ein Basisfall zum Beenden der Rekursion ist dabei unerlässlich.

Diese Konzepte sind fundamental für die Softwareentwicklung.