**Werttypen & Verweistypen / benutzerdefinierte Methoden**

**Werttypen (Value types):**

* Bei Werttypen wird **der tatsächliche Wert in der Variable gespeichert**. Sie sind direkt mit ihren Werten verknüpft.
* Beispiele: **int, float, double, char, bool, structs**
* Verhalten: Wenn eine Variable mit einem Werttyp einer anderen zugewiesen wird, wird eine **Kopie** des Wertes erstellt. **Änderungen an der Kopie wirken sich nicht auf die Originalvariable aus.**

**Verweistypen (Reference types):**

* Bei Verweistypen wird nur die Referenz (der "Zeiger") auf die Adresse im Speicher gespeichert.
* Beispiele: **string, arrays, class**
* Verhalten: Wenn eine Variable mit einem Verweistyp einer anderen zugewiesen wird, wird die Referenz auf dasselbe Objekt kopiert. **Änderungen am Objekt über eine der Variablen wirken sich auf das Original aus.**

**Speicherverwaltung: Stack vs. Heap**

**Stack (Stapelspeicher) ist für Werttypen**

* Der Stack ist ein schneller, **statisch verwalteter** Speicherbereich.
* Hier werden **lokale Variablen** und **Werttypen** gespeichert.
* Der Zugriff auf den Stack ist **extrem schnell**, da es sich um eine einfache Speicherverwaltung handelt.

**Heap (Dynamischer Speicher) ist für Referenztypen**

* Der Heap ist ein **dynamisch verwalteter** Speicherbereich.
* Hier werden **Referenztypen** (z. B. Objekte, Arrays, Strings) gespeichert.
* Die Verwaltung des Heaps ist aufwendiger, da Objekte hier **nicht automatisch** gelöscht werden, sondern erst durch den Garbage Collector (GC).
* Der Zugriff auf Heap-Objekte erfolgt über **Zeiger (Referenzen)**, was im Vergleich zum direkten Zugriff auf den Stack etwas langsamer ist.

**Probe**

static void MachEtwas(char prefix, Person person)

{

prefix = 'B';

person.Name = prefix + " " + person.Name;

}

// Werttyp (Value type)

char prefix = 'H';

// Verweisetyp (Reference type)

var person = new Person()

{

Alter = 36,

Name = "Huber"

};

string name = person.Name;

MachEtwas(prefix, person);

name = person.Name;

Console.WriteLine(prefix);

Console.WriteLine(person.Name);

Console.WriteLine(person.Alter);

Console.WriteLine(name);

**Methoden**

Methoden kapseln wiederverwendbaren Programmcode und erleichtern somit nicht nur die Arbeit des Programmierers, sondern tragen auch in erheblichem Maße zur Übersichtlichkeit des Programmcodes und dessen Strukturierung bei.

Jede Methode hat einen Namen und einen Körper.

Der Name der Methode soll „sprechend“ sein, das heißt: Das abbilden, was die Methode verrichtet bzw. „zu tun“ hat. In der modernen Programmierart verzichtet man weitgehend auf jegliche Kommentare, umso wichtiger sind gut gewählte Variablen, Klassen und Methoden-Bezeichner. Durch diese sollte sich der Code „selbst dokumentieren.

Beispiel für passenden Methodenamen:

RufeFormelAusDbUndSpeichereSieAlsJson()

Bzw.

CallFormulaFromDbAndSaveItAsJson()

Ändern sich die Aufgaben der Methode, sollte auch der Name entsprechend angepasst werden.

Eine Methode sollte nach Möglichkeit die Länge von 20 **Codezeilen** nicht überschreiten. Ihr Kompetenzbereich sollte sich möglichst **auf nur eine Hauptaufgabe** beschränken.

Grundsätzlich unterscheiden wir zwischen

Methoden **mit Rückgabewert** (Funktionen), z.B.

**decimal** GetNeuPreis()

{

return 123.90m;

}

Der Rückgabewert ist hier von Typ Integer.  
Jede Methode mit Rückgabewert muss auch solchen mittels „return“ zurückliefern.

Aufruf: var neuPreis = GetNeuPreis();

und Methoden **ohne Rückgabewert (**Prozeduren oder einfach Methoden), z.B.

**void** GenerierePasswort()

{

var passwort = "jkkf8343jhkf#hjkhdsf";

}

Hier existiert kein Rückgabewert.

Aufruf: GenerierePasswort();

Methoden können Übergabe-Parameter (Argumente) haben.

decimal GetNeuPreis(**decimal altpreis**)

{

return (altpreis + 1.0m);

}

static void MachEtwas(char prefix, Person person)

{

prefix = 'B';

person.Name = prefix + " " + person.Name;

}

**ref und out**

Ein Argument von **Wertetyp** wird in die Methode als Kopie übergeben.

Ein Argument von **Referenztyp** wird in die Methode als **Verweis** (Pointer) (mit all den damit zusammenhängenden Konsequenzen) übergeben.

Will man einen Wertetyp als Referenz in die Methode übergeben, muss man seiner Variablen das Schlüsselwort ref voranstellen.

void Verdopple(**ref** int zahl)

{

zahl \*= 2;

}

Anwendung:  
int zahl = 45;

Verdopple(ref zahl);

Will man den Wertetyp nicht überschreiben und das Ergebnis in einer **separaten** Variable erhalten, bedient man sich des Schlüsselwortes out.

void Verdopple(int zahl, out int ergebnis)

{

ergebnis = zahl \*= 2;

}

int zahl = 45;

Verdopple(zahl, out int ergebnis);

**Überladung**

Wenn mehrere gleichnamige Methoden im gleichen Gültigkeitsbereich ohne Namenskonflikt nebeneinander existieren, so spricht man von überladenen Methoden.

Gleichnamige Methoden müssen sich **mit ihren Signaturen unterscheiden**: Der Name bleibt gleich, die Parameteranzahl und / oder Parameter-datentyp müssen jeweils anders sein.

static int Addiere(int zahl1, int zahl2)

{

return zahl1 + zahl2;

}

static double Addiere(double zahl1, double zahl2)

{

return zahl1 + zahl2;

}

static string Addiere(string wort1, string wort2)

{

return wort1 + wort2;

}

static string Addiere(double zahl, string wort)

{

return zahl + wort;

}

**Optionale Parameter**

Optionale Parameter gestatten Ihnen, diese mit Standardwerten zu belegen. Bei Aufruf der Methode bleibt es Ihnen überlassen, für diese(n) Parameter ein Argument zu übergeben oder nicht.

static string GetName(string vorname = "Franz", string nachname = "Huber")

{

return vorname + " " + nachname;

}

**Benannte Parameter**

**Benannte Parameter** ermöglichen das gezielte Setzen von Werten, unabhängig von der Reihenfolge in der Parameterliste.

static string GetPerson(string vorname = "Franz", string nachname = "Huber", string stadt = "Wien")

{

return vorname + " " + nachname + "aus " + stadt;

}

var person = GetPerson(stadt: "Wien", vorname: "Andreas");

}

**Mit params**

Der Schlüsselwort **params** erlaubt eine **variable Anzahl von Argumenten**. Praktisch für Methoden, die mit einer variablen Anzahl von Eingaben arbeiten (z. B. Summen, Durchschnittsberechnungen).

static string MacheKommaseparierteZeichenkette(params string[] woerter)

{

return string.Join(',', woerter);

}

var str = MacheKommaseparierteZeichenkette("Fluss", "Meer", "Fische");

**Lokale Funktionen**

**Lokale Funktionen** sind Methoden, die innerhalb einer anderen Methode definiert werden. Sie sind nützlich, wenn eine Hilfsmethode nur innerhalb einer bestimmten Methode benötigt wird.

static string Umkehren(string eingabe)

{

char[] ZeichenUmkehren(string str)

{

char[] chars = str.ToCharArray();

Array.Reverse(chars);

return chars;

}

return new string(ZeichenUmkehren(eingabe));

}

string text = "Hallo Welt";

string umgekehrt = Umkehren(text);

**Rekursive Funktionen**

Rekursive Funktionen sind Funktionen, die sich selbst innerhalb ihres eigenen Körpers aufrufen. Sie lösen Probleme, indem sie das gleiche Problem in kleinere Teilprobleme aufteilen, wobei der rekursive Aufruf jeweils eine Vereinfachung des ursprünglichen Problems darstellt.

static int Fakultät(int n)

{

if (n < 0) throw new ArgumentException("n muss positiv sein");

if (n == 0) return 1; // Abbruchbedingung

return n \* Fakultät(n - 1); // Rekursiver Aufruf

}

**Anhang Wert- und Verweistypen**

**Wertetypen** mit ihren Standardwerten:

**Ganzzahlige Typen (Integral Types)**

|  |  |
| --- | --- |
| **Typ** | **Standardwert** |
| byte | 0 |
| sbyte | 0 |
| short | 0 |
| ushort | 0 |
| int | 0 |
| uint | 0 |
| long | 0 |
| ulong | 0 |

**2. Fließkommatypen (Floating-Point Types)**

|  |  |
| --- | --- |
| **Typ** | **Standardwert** |
| float | 0.0f |
| double | 0.0d |
| decimal | 0.0m |

**3. Zeichen-Typ (Character Type)**

|  |  |
| --- | --- |
| **Typ** | **Standardwert** |
| char | '\0' |

**4. Boolescher Typ (Boolean Type)**

|  |  |
| --- | --- |
| **Typ** | **Standardwert** |
| bool | false |