**Objektorientierte Programmierung (OOP) 1**

Die objektorientierte Programmierung (OOP) ist eine der wichtigsten Programmierparadigmen, weil sie:

1. **Struktur und Ordnung** in den Code bringt
2. **Wiederverwendbarkeit** von Code ermöglicht
3. **Realitätsnahe Modellierung** erlaubt

**Grundbegriffe**

**Klasse** → Bauplan für Objekte

**Objekt** → Konkrete **Instanz** (Konstruktion) einer Klasse

**Eigenschaften (Attribute, Felder)** → Beschreiben den Zustand eines Objekts

**Methoden** → Definieren das Verhalten eines Objekts

**Konstruktor** → Initialisiert ein Objekt beim Erstellen

**Destruktor** → wird aufgerufen, wenn das Objekt durch den Garbage Collector „zerstört“ (aus dem Speicher entfernt) wird.

**Beispiel aus der realen Welt:**

Ein **Auto** hat Eigenschaften (Farbe, PS, Geschwindigkeit) und Methoden (Fahren(), Bremsen()).

Ein **Mitarbeiter** hat Name, Geburtsdatum, Gehalt und kann Arbeiten().

**Erstellung einer Klasse**

class Auto

{

// Eigenschaften

public string Farbe;

public int Geschwindigkeit;

// Verhalten

public void Fahren()

{

Console.WriteLine("Das Auto fährt.");

}

}

Hier definieren wir eine **Klasse Auto** mit zwei **Eigenschaften** und einer **Methode**.

**Erstellung eines Objekts**

Auto meinAuto = new Auto();

meinAuto.Farbe = "Rot";

meinAuto.Geschwindigkeit = 50;

Console.WriteLine($"Mein Auto ist {meinAuto.Farbe} und fährt {meinAuto.Geschwindigkeit} km/h.");

meinAuto.Fahren();

**Erklärung:**

* new Auto(); erstellt ein Objekt.
* Mit meinAuto.Farbe = "Rot"; setzen wir **Eigenschaften**.
* meinAuto.Fahren(); ruft eine **Methode** auf.

**Übung: Eigenes Objekt erstellen**

**Aufgabe:**

1. Erstelle eine Klasse Person mit den Eigenschaften Name und Alter.
2. Füge eine Methode Sprechen() hinzu, die den Namen der Person ausgibt.
3. Erstelle ein Objekt und rufe Sprechen() auf.

**Konstruktoren – Objekte mit Werten initialisieren**

**Warum ein Konstruktor?**

Ohne Konstruktor müssten wir jedes Objekt erst nach der Erstellung initialisieren:

Auto meinAuto = new Auto();

meinAuto.Farbe = "Blau";

meinAuto.Geschwindigkeit = 50;

Das ist fehleranfällig. Mit einem **Konstruktor** können wir **essenzielle Werte** direkt setzen.

**Konstruktor in einer Klasse**

class Auto

{

public string Farbe;

public int Geschwindigkeit;

// Konstruktor

public Auto(string farbe, int geschwindigkeit)

{

Farbe = farbe;

Geschwindigkeit = geschwindigkeit;

}

}

Jetzt können wir ein Objekt **direkt mit Werten erstellen**:

Auto meinAuto = new Auto("Blau", 60);

Console.WriteLine($"Mein Auto ist {meinAuto.Farbe} und fährt {meinAuto.Geschwindigkeit} km/h.");

**Übung: Konstruktor erstellen**

**Aufgabe:**  
Erweitere die Klasse Person um einen Konstruktor, der Name und Alter direkt setzt.

**Problem: Unkontrollierter Zugriff**

Aktuell kann jeder direkt auf Variablen zugreifen und sie ändern:

meinAuto.Farbe = "Lila"; // Jeder kann das tun!

Das kann zu falschen oder unerwarteten Werten führen.

**Lösung: Zugriffsschutz mit private und public**

Wir setzen die Eigenschaften auf private und nutzen **Getter und Setter (Zugriffsmodifikatoren)**:

public class Auto

{

private string \_farbe;

public Auto()

{

\_farbe = "blau";

}

public string Farbe

{

get

{

return \_farbe;

}

private set

{

\_farbe = value;

}

}

}

Auto meinAuto = new Auto();

meinAuto.Farbe = "gelb"; // geht nicht!

Console.WriteLine(meinAuto.Farbe); // Gibt "Blau" aus

**Übung: Zugriff auf Daten einschränken**

**Aufgabe:**

1. Ändere die Klasse Person, sodass Alter nicht direkt (von außen) gesetzt werden kann.
2. Erstelle Methoden SetAlter(int alter) um das Alter zu validieren (nur Werte zwischen 0 und 120 erlaubt).

**Abschlussprojekt: Ein kleines Programm schreiben**

**Aufgabe:**

1. Erstelle eine Klasse Buch mit den Eigenschaften Titel, Autor und Seitenzahl.
2. Nutze Kapselung: Seitenzahl darf nicht kleiner als 1 sein.
3. Erstelle einen Konstruktor, der Titel und Autor direkt setzt.
4. Erstelle ein Objekt meinBuch und gib die Details aus.

**Zusammenfassung – Das Wichtigste auf einen Blick**

**Klassen** sind Baupläne für Objekte.  
**Objekte** sind Instanzen einer Klasse.  
**Methoden** definieren das Verhalten eines Objekts.  
**Konstruktoren** initialisieren ein Objekt (mit Werten).  
**Zugriffsmodifikatoren** schützen Daten vor unkontrollierter Änderung.

**Vertiefung Konstruktor**

Der Gegensatz von Objekt-Konstruktor ist der **Destruktor**.

In C# gibt es das aut. „Müllabfuhr“-System, das nicht mehr verwendete Objekte (auf null gesetzte), aus dem RAM (Computerspeicher) entfernt. Dies passiert meistens nicht sofort, sondern wird von .NET-Framewort eingetaktet.

Bei Speicherkritischen Anwendungen (z.B. Bildbearbeitung) sollte man im Sinne besserer Performance den Speicher explizit freigeben lassen. GC.Collect() lässt die „Müllabfuhr“ sofort kommen.

**Kapselung**

In der objektorientierten Programmierung (OOP) bezeichnet Kapselung (englisch: *encapsulation*) das Prinzip, Daten (Felder/Attribute) und Methoden, die auf diese Daten zugreifen, in einer Klasse zu bündeln und den direkten Zugriff von außen einzuschränken**.**

**Ziel der Kapselung:**

* **Schutz der inneren Datenstruktur eines Objekts.**
* **Vermeidung von unkontrollierten Änderungen durch außenstehenden Code.**
* **Kontrollierter Zugriff über Methoden (z. B. Get-/Set-Methoden oder Eigenschaften).**

public class Konto

{

private decimal kontostand; // privat: von außen nicht direkt zugreifbar

public decimal Kontostand // öffentliche Eigenschaft mit Zugriffskontrolle

{

get { return kontostand; }

private set { kontostand = value; } // Setzen nur innerhalb der Klasse erlaubt

}

public void Einzahlen(decimal betrag)

{

if (betrag > 0)

Kontostand += betrag;

}

}

**Wichtige Punkte zur Kapselung:**

|  |  |
| --- | --- |
| Merkmal | Beschreibung |
| Zugriffsmodifizierer | **private, protected, public zur Steuerung des Zugriffs** |
| Getter/Setter | **Methoden oder Eigenschaften zum kontrollierten Zugriff** |
| Trennung von intern/extern | **Innere Details bleiben versteckt; nur definierte Schnittstellen sichtbar** |

Warum ist Kapselung sinnvoll?

* Fehlerminimierung: Andere Teile des Programms können interne Daten nicht versehentlich verändern.
* Änderbarkeit: Interne Implementierungen können geändert werden, ohne dass externer Code angepasst werden muss.
* Wartbarkeit & Übersichtlichkeit: Komplexität wird reduziert**.**