Klassen III – Properties

Properties

Indexer

Wiederholung: Klassen bisher

Klassen:

- bilden den Bauplan für Objekte
- bestehen aus Feldern und Methoden
- kontrollieren die direkte Nutzbarkeit mittels Zugriffsparametern
- haben Konstruktoren, die Initialwerte setzen können
- können mittels static auch eigene Felder/Methoden besitzen

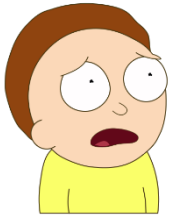
Wiederholung: Konstruktoren

```
class Bunny
{
    public string Name;
    public bool LikesCarrots;
    public bool LikesHumans;

    public Bunny(string name) => this.Name = name;
}

class App
{
    static void Main(string[] args)
    {
        // Default Constructor
        Bunny bun1 = new Bunny("Puschel");
        bun1.LikesCarrots = true;
        bun1.LikesHumans = false;
    }
}
```

Objekt-Initialisierer

```
class Bunny
{
    public string Name;
    public bool LikesCarrots;
    public bool LikesHumans;

    public Bunny(string name) => this.Name = name;
}

class App
{
    static void Main(string[] args)
    {
        // Objekt-Initialisierer (für alle erreichbaren Felder)
        Bunny bun1 = new Bunny("Puschel") {LikesCarrots = true, LikesHumans = false };
        // ist praktisch Syntax-Sugar für:
        Bunny tmpBunny = new Bunny("Puschel");
        tmpBunny.LikesCarrots = true;
        tmpBunny.LikesHumans = false;
        Bunny bun2 = tmpBunny;
    }
}
```

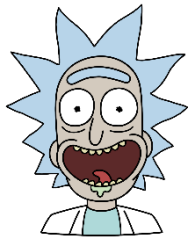
Wozu sind die eigentlich gut? Ich kann doch auch sowas machen:



```
public Bunny(string name, bool likesCarrots = true, bool likesHumans = false)
    {
        this.Name = name;
        this.LikesCarrots = likesCarrots;
        this.LikesHumans = likesHumans;
    }
```

```
Bunny bun1 = new Bunny("Puschel");
```

Das Ergebnis scheint gleich zu sein. Aber im Detail liegt der Unterschied. Mit Initialisierern ist wesentlich mehr möglich. Außerdem können wir sie auch bei anderen Konstrukten benutzen, die wir später kennen lernen.



Kopierkonstruktoren (Copy-Konstruktor)

```
public Bunny(string name) => this.Name = name;
public Bunny(Bunny otherBunny)
{
    this.Name = otherBunny.Name;
    this.LikesCarrots = otherBunny.LikesCarrots;
    this.LikesHumans = otherBunny.LikesHumans;
}
```

```
static void Main(string[] args)
{
    Bunny bun1 = new Bunny("Fluffel"){
        LikesCarrots = true,
        LikesHumans = false
    };
    Bunny bun2 = new Bunny(bun1);
}
```

- Kopierkonstruktoren werden genutzt, um Objekte zu vervielfältigen
- flache Kopie vs. tiefe Kopie
- *[Interface ICloneable gilt per Design Guideline als veraltet. Disskussion über den Status obsolet.]*

Problematik

```
class Student
{
    public string Name, Studiengang;
    public int Matrikelnummer;

    public Student(string name,
        string studiengang, int matrikelnummer)
    {
        this.Name = name;
        this.Studiengang = studiengang;
        this.Matrikelnummer = matrikelnummer;
    }
}
```

- auf Daten soll zugegriffen werden können
- grundlegend sollen Daten veränderbar sein
- konsistentes Verhalten soll gesichert sein
- Lösung: Set-/Get-Methoden

Traditionelle Get-/Set-Methoden

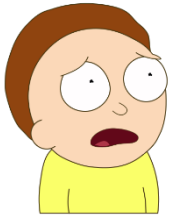
```
class Student
{
    private string Name, Studiengang;
    private int Matrikelnummer;

    // Get-Methoden
    public string GetName() => this.Name;
    public string GetStudiengang() => this.Studiengang;
    public int GetMatrikelnummer() => this.Matrikelnummer;

    // Set-Methoden
    public void SetStudiengang(string studiengang)
        => this.Studiengang = studiengang;

    // Konstruktor:
    public Student(string name,
        string studiengang, int matrikelnummer)
    {
        this.Name = name;
        this.Studiengang = studiengang;
        this.Matrikelnummer = matrikelnummer;
    }
}
```

```
static void Main(string[] args)
{
    Student stud1 = new Student("Donald Duck", "GtB", 123456);
    stud1.SetStudiengang("VT");
    // Compilerfehler:
    //stud1.Studiengang = "VT";
    Console.WriteLine(stud1.GetName() + ", " + stud1.GetStudiengang());
}
```

- Set-/Get-Methoden kapseln die private-Felder und geben beschränkten Zugriff
- Get-Methoden können Daten on-the-fly berechnen
- Set-Methoden können Plausibilität der Daten prüfen

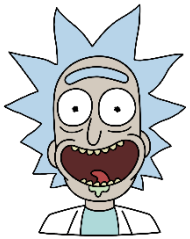
Das ist aber viel Schreibarbeit, wenn man alle Felder darüber sichern möchte und ich ja auch keine public-Felder haben möchte.

Geht das nicht einfacher?



Anders als C++ oder Java hat C# die Möglichkeit, auf dedizierte Get- und Set-Methoden zu verzichten.

Dazu gibt es Properties, die alles einfacher machen.



Properties

```
// Felder:
private string StudName, StudStudiengang;
private int StudMatrikelnummer;

// Properties:
public string Studiengang
{
    get { return StudStudiengang; }
    set { StudStudiengang = value; }
}
public string Name
{
    get { return StudName; }
}
public int Matrikelnummer
{
    get { return StudMatrikelnummer; }
}
```

- zu jedem Property gehört ein „Backing-Field“
- Set-Properties nutzen die Variable **value** als Eingabewert.
- Properties können benutzt werden wie Felder

```
static void Main(string[] args)
{
    Student stud1 = new Student("Donald Duck", "GtB", 123456);
    stud1.Studiengang = "VT";
    Console.WriteLine(stud1.Name + ", " + stud1.Studiengang);
}
```

```
>propertiesExample02.exe
Donald Duck, VT

>
```

Fat Arrow Syntax und Automatic Properties

```
// Felder:
private string StudName, StudStudiengang;
private int StudMatrikelnummer;

// Properties:
public string Studiengang
{
    get => StudStudiengang;
    set => StudStudiengang = value;
}
public string Name
{
    get => StudName;
}
public int Matrikelnummer
{
    get => StudMatrikelnummer;
}
```

```
// Properties:
public string Studiengang { get; set; }
public string Name { get; }
public int Matrikelnummer { get; }

// Konstruktor:
public Student(string name,
    string studiengang, int matrikelnummer)
{
    this.Name = name;
    this.Studiengang = studiengang;
    this.Matrikelnummer = matrikelnummer;
}
```

Die Backing-Fields werden hier automatisch mit erstellt, um die Daten zu speichern.

Automatische Properties

```
// Properties:
public string Studiengang { get; private set; }
public string Name { get; }
public int Matrikelnummer { get; } = 0;

// Methoden
public void StudiengangWechsel(string neuerStudiengang)
{
    this.Studiengang = neuerStudiengang;
}
```

- **set** und **get** können verschiedene Zugriffsparameter besitzen
- Properties können einen Initialwert haben

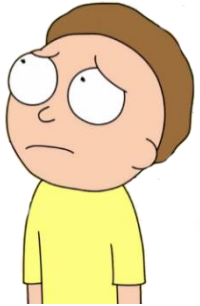
Properties mit Logik und Initialwerten

- Properties müssen keinem speziellen Feld zugeordnet sein und können auch kleine Berechnungen durchführen

```
class Produkt
{
    double NettoPreis;
    public double Mehrwertsteuer { get; } = 0.19;
    public double Preis { get => NettoPreis * (1+Mehrwertsteuer); }

    public Produkt(double nettoPreis) => NettoPreis = nettoPreis;
}

class App
{
    static void Main(string[] args)
    {
        Produkt prod1 = new Produkt(8.00);
        Console.WriteLine("Preis: " + prod1.Preis.ToString());
    }
}
```

- Properties können nicht mittels **ref/out** an Methoden übergeben werden
- Readonly Properties müssen einen Initialwert haben oder im Konstruktor definiert werden
- auf die Backing-Fields automatischer Properties kann nur über die Property zugegriffen werden
- automatische Properties sind nicht immer die beste Lösung

Static Properties



Static Properties? Gibt es sowas etwa auch?

Aber natürlich.
Static waren Klassen mit eigenen Membern. Und dort gibt es auch Properties.

Anwendungen sind dafür aber eher in komplexen Strukturen zu finden.

Indexer

- Klassen kapseln oft Listen oder Dictionaries von Werten

- **Indexer** sind spezielle Properties, die über Indexargumente laufen, statt über Namen

```
static void Main(string[] args)
{
    string s = "Hallo Welt";
    for(int i = 0; i<s.Length; i++)
    {
        Console.WriteLine(s[i]);
    }
}
```

- **Indexer** können fast wie Arrays verwendet werden, bleiben aber Properties und sind somit eher Methoden

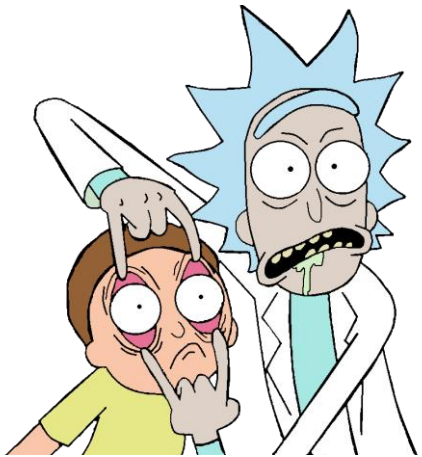
Eigene Indexer

Eigene Indexer werden durch das **this** Property implementiert.

```
class Sentence
{
    public string[] words = "Kranplätze können verdichtet sein !".Split();
    public string this [int wordNumber] {
        get => words[wordNumber];
        set => words[wordNumber] = value;
    }
    public char this [int wordNumber, int characterNumber]
    {
        get => words[wordNumber][characterNumber];
    }
}
```

```
Sentence satz = new Sentence();
satz[1] = "müssen";
foreach (string word in satz.words)
    Console.Write(word + " ");
Console.WriteLine();
```

```
>propertiesExample06.exe
Kranplätze müssen verdichtet sein !
```



Bei Indexern kann es schnell vorkommen, dass man null dereferenzieren will.

Also aufpassen mittels Exceptions.

Oder Besser: Null Operators!

? – null bedingter Operator – null conditional op

- Referenztypen sollten immer vor der Benutzung auf null geprüft werden

```
string[] words = "Hallo Welt !".Split();
words = null; // ups

if (words != null) Console.WriteLine(words[1]);
else Console.WriteLine();

try { Console.WriteLine(words[2]); }
catch (NullReferenceException) { Console.WriteLine(); }
```

- Der null bedingte Operator <?> prüft, ob eine Referenz null ist und gibt dann null zurück

```
Console.WriteLine(words?[2]);
Console.WriteLine(words?.Length);
```

- Elvis Operator wegen ?:j

?? – null Sammeloperator – null coalescing op

- wenn man auf ein null noch anders reagieren möchte, hilft <??>

```
Console.WriteLine(words?[2] ?? "ups");
Console.WriteLine(words?.Length ?? "Das Feld ist leer");
```

Compiler-Fehler

- typische Anwendung: Kommandozeilenparameter

```
static void Main(string[] args)
{
    Console.WriteLine($"Kommandozeilenparameter: {args?.Length ?? 0}");
}
```

```
>nullableExample01.exe
Kommandozeilenparameter: 0

>nullableExample01.exe Hallo Kommandozeile
Kommandozeilenparameter: 2
```

Zusammenfassung

- Objektinitialisierer sind bei der definition von **public**-Feldern nützlich
- **properties** regeln den Zugriff auf Felder
- das Konzept der **automatic properties** nimmt viel Arbeit ab
- **Indexer** ermöglichen den direkten Zugriff auf gekapselte Felder
- **?** und **??** helfen besonders bei möglichen null-Referenzen