3
Klassen, Attribute, Methoden

Jörn Loviscach

Versionsstand: 20. März 2012, 15:24

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen in der Vorlesung.
Videos dazu: http://www.j3L7h.de/videos.html

1 Klassen, Attribute, Methoden

Eine Klasse ist der Bauplan für einen eigenen Typ von Objekten. Dieser Bauplan beschreibt insbesondere, welche Attribute (gleichbedeutend: Datenelememente, Instanzvariablen, member variables, fields) und welche Methoden (gleichbedeutend: Elementfunktionen, member functions) jedes Objekt dieses Typs haben soll:

[1]

Wie in Java und anders als in C++ steht am Ende kein Semikolon. Wie in C++ und anders als in Java ist in C# der Name der Datei egal. In C++ würde man fast immer eine Header-Datei für die Klasse schreiben und jede der Funktionen einzeln in einer .cpp-Datei definieren.

In Java und C# gibt es keine anderen Funktionen als die Methoden von Klassen!

Ohne weitere Maßnahmen ist nichts in der Klasse von außen sichtbar. Man könnte ohne Änderung auch überall `private` davor schreiben. Die Methode `isNow` aus dem Beispiel soll aber von anderswo aufgerufen werden können:

[2]

Nach Microsofts Wunsch sollen alle öffentlichen Elemente einen Namen haben, der mit einem Großbuchstaben anfängt (Link).

Innerhalb einer Klasse hat man direkten Zugriff auf alle Bestandteile deren oberster Ebene. Die Methode `isNow` kann also so geschrieben werden:

[3]

2 Überladen

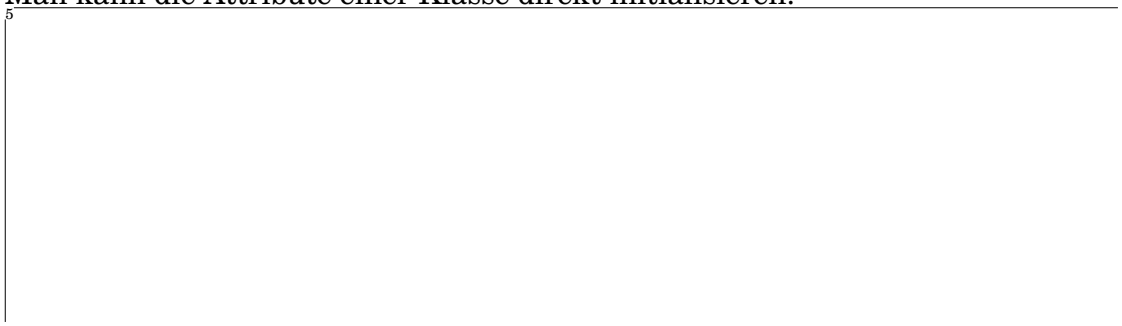
In C durften keine zwei Funktionen den gleichen Namen haben. Anders in den neueren Sprachen wie C++, Java und C#: Sogar in ein und derselben Klasse sind mehrere Methoden gleichen Namens erlaubt – wenn sich diese denn durch die Typen ihrer Parameter unterscheiden lassen. Das heißt „Überladen“ [overloading] von Funktionen. Die Syntaxhilfe in Visual Studio zeigt die verschiedenen überladenen Versionen an.

Unsere Klasse `Meeting` können wir zum Beispiel mit zwei verschiedenen Methoden zum Verschieben ausstatten:

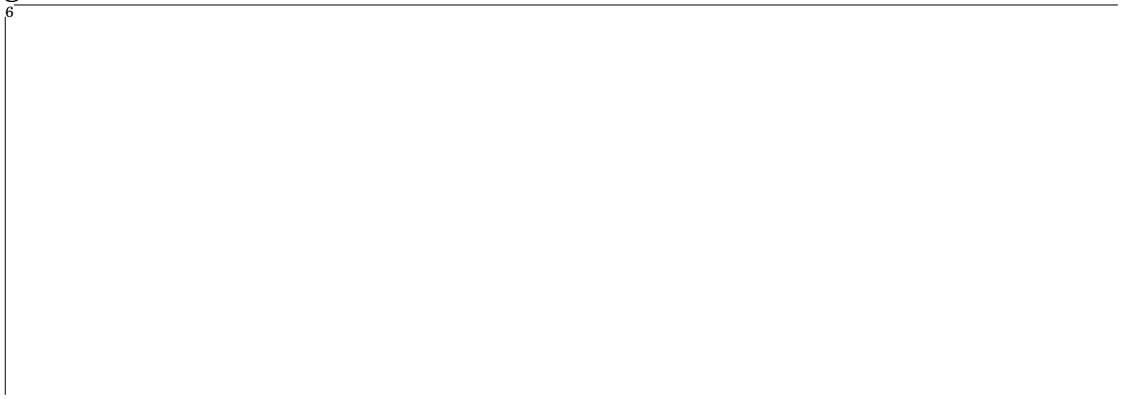
[4]

3 Initialisierung, Konstruktor

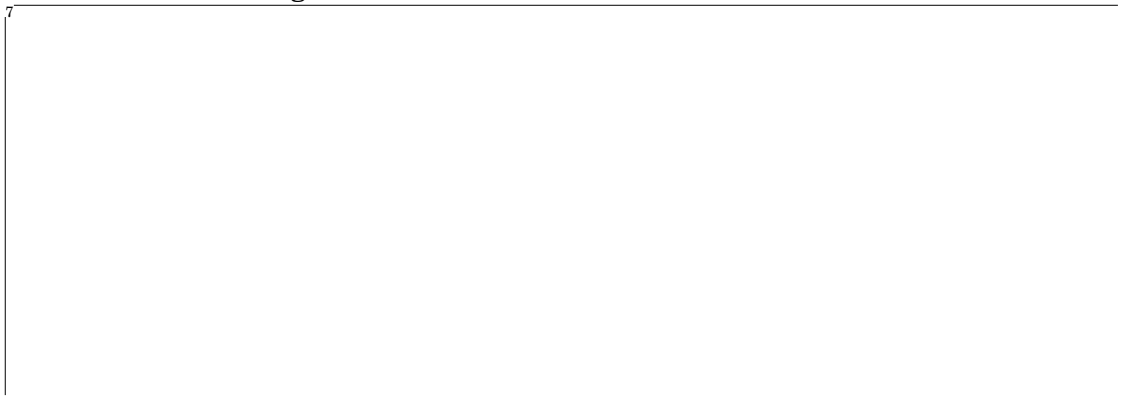
Man kann die Attribute einer Klasse direkt initialisieren:

5


Aber sinnvoller ist meist ein Konstruktor [constructor]: eine besondere Funktion, die eine Instanz der Klasse baut. Der Konstruktor wird bei `new` aufgerufen – und auch nur dann. In den gängigen Sprachen heißt der Konstruktor wie die Klasse; der Rückgabetyp wird nicht angegeben; ein `return` mit dem konstruierten Objekt gibt es auch nicht.

6


Ein solcher Konstruktor ohne Parameter heißt Standard-Konstruktor [default constructor]. Er erlaubt, ein Objekt ohne weitere Angaben zu bauen. Konstruktoren dürfen aber auch Parameter haben. Es darf sie wie die anderen Methoden in mehreren Versionen geben (Überladen von Funktionen!):

7


Diese Klasse lässt sich dann so verwenden:

```
8
```

Typischerweise baut man auch einen Konstruktor, der alle Daten entgegen nimmt:

```
9
```

Schön ist, wenn die Namen der Parameter dieselben sind wie die Namen der Datenelemente. Das führt aber erstmal zu nichts, weil die Parameter die Datenelemente gleichen Namens verdecken. Der Trick in C++, Java und C # ist, `this` zu benutzen. Das bezeichnet die aktuelle Instanz der Klasse:

```
10
```

Das `this` ist hier nett; an anderen Stellen wird es später unentbehrlich sein.

Hat man in der Klasse `Abc` *keinen* Konstruktor definiert, darf man trotzdem `Abc x = new Abc();` aufrufen – aber natürlich nicht `Abc x = new Abc(42, "test");`. Sobald man aber in der Klasse *irgendeinen* Konstruktor selbst definiert hat, ist `Abc x = new Abc();` nicht mehr erlaubt – es sei denn, man hat einen solchen Standardkonstruktor selbst definiert. Das ist eine Sicherheitsmaßnahme in C++, Java und C#, weil die selbst gebauten Konstruktoren typischerweise bestimmte Bedingungen sichern, die die Attributen erfüllen müssen. Zum Beispiel könnte es verboten sein, ein `Meeting` in

der Vergangenheit anzulegen. Bei einer Raumbuchung könnten Doppelbelegungen verboten sein.

4 Getter, Setter, Properties

Aus demselben Grund ist es heikel, Attribute öffentlich zu machen: Wenn andere Klassen Schreibzugriff auf diese Daten haben, können sie Abhängigkeiten dazwischen zerstören. Außerdem kann man die Innereien einer Klasse nur noch schlecht ändern, wenn andere Klassen tief mit diesen Innereien verstrickt sind.

Die übliche Lösung ist, Attribute *nicht* öffentlich zu machen und stattdessen öffentliche Methoden anzubieten, mit denen sie gesetzt und gelesen werden können:

[11]

Diese Methoden können dann die übergebenen Werte überprüfen, gegebenenfalls andere Attribute anpassen und/oder verbergen, dass die Klasse im Inneren ganz anders funktioniert. Gibt es nur einen Getter, ist das entsprechende Attribut vor Änderungen von außen geschützt.

Mit Gettern und Settern zu arbeiten, führt allerdings zu viel Tipparbeit – zum Beispiel, wenn man den Wert eines Attributs verändern will:

[12]

In C# gibt es deshalb „Properties“, die auf den ersten Blick verwendet werden wie Attribute, eigentlich aber Methoden sind:

[13]

Lässt man einen einen der beiden Teile weg oder machen ihn nicht öffentlich, ist

die Property schreib- oder lesegeschützt. Tipp: In Visual Studio `propfull` tippen und zweimal die Tabulatortaste drücken.

Die Klassenbibliothek macht massiv Gebrauch von Properties. Insbesondere sind die im Eigenschaftenfenster von Visual Studio aufgelisteten Daten alles Properties. Das heißt, obwohl man scheinbar direkt Werte liest und schreibt, wird hinter den Kulissen jeweils eine Methode aufgerufen. Einige wenige Properties wie die `ActualHeight` eines Fensters sind dabei schreibgeschützt.

5 Strukturen, Referenzen

Wie in C steht in Java und C# bei Typen wie `bool` und `int` die Variable für den Wert. Nach `int a = 42; int b = a;` stehen in `a` und `b` voneinander unabhängige Werte: `a += 13;` hat keinen Effekt auf `b`. Ebenso kann ein Funktionsaufruf wie `f(a)` nicht den Wert von `a` ändern.

In C# gilt dieses Verhalten auch für Strukturen. Die sind für leichtgewichtige Objekte gedacht. Sie werden mit `struct` statt `class` angelegt und sehen deshalb aus wie Strukturen in C und C++. (Der Unterschied zwischen `struct` und `class` ist in C++ allerdings ein ganz anderer als in C#.) Dies hier sind einige Strukturen aus der Klassenbibliothek:

[14]

Klassen in Java und C# werden dagegen anders behandelt: Die Variablen stehen nicht für Werte, sondern für Referenzen, ganz wie die Zeiger in C und C++. Nach

[15]

scheint auch `b` verstellt. In Wirklichkeit verweisen `a` und `b` auf dasselbe Objekt. Demo im Debugger. Ebenso kann ein Funktionsaufruf wie `f(a)` nun Attribute von `a` ändern.

Dieses Verhalten bei Instanzen von Klassen ist dramatisch anders als bei den Werttypen! Man muss sich immer wieder klar machen, ob man es gerade mit einer Referenz oder einem Wert zu tun hat. Entsprechend zu C und C++ steht in Java

und C# `null` für eine nicht vorhandene Referenz.[16]

Bei Werttypen arbeiten die Vergleiche == und `!=`, wie man das erwartet. Für `struct` sind diese beiden Vergleiche nicht automatisch definiert. Bei Referenzen sind sie dagegen automatisch definiert, prüfen aber, ob die Referenzen auf dasselbe Objekt verweisen:[17]

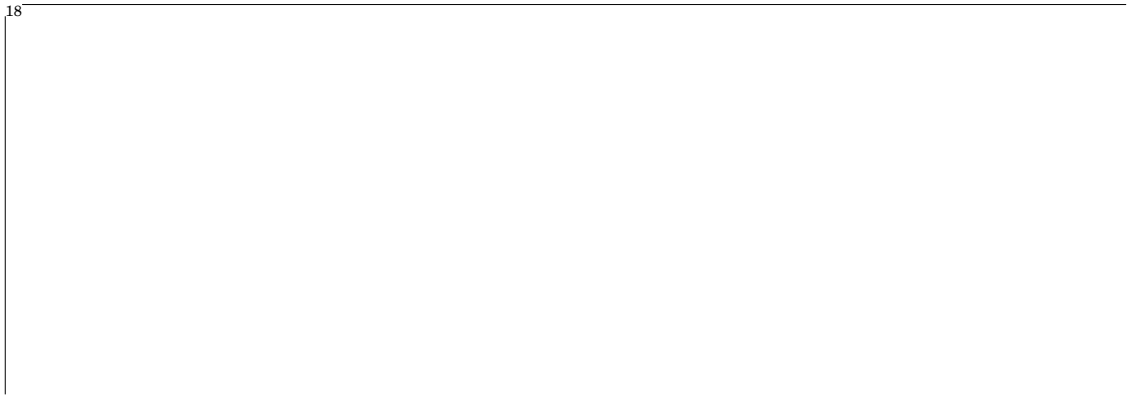
Um auch den üblichen Vergleich zu haben, sollte man seine Klassen um eine entsprechende Vergleichsmethode erweitern (Details später).

Zeichenketten verhalten sich noch etwas anders. Insbesondere sind bei ihnen die Vergleich == und `!=` wirklich Vergleiche der Inhalte.

6 Statische Attribute und Methoden

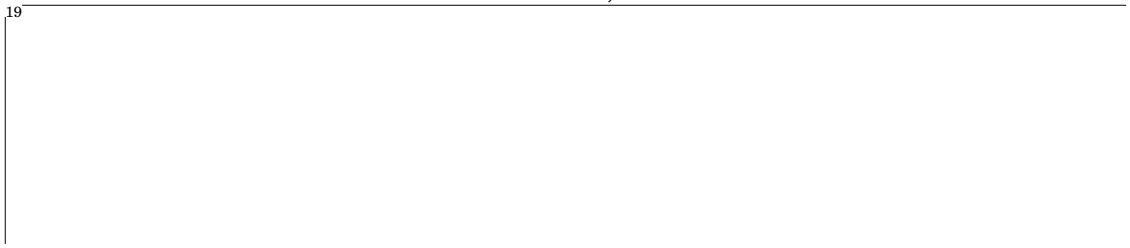
In C und C++ kann man `static` verwenden, um in Funktionen Werte vom vorigen Aufruf zu bewahren oder um die Sichtbarkeit von Variablen oder Funktionen auf eine Datei zu beschränken. In Java und C# gibt es nicht diese Bedeutungen von `static`, sondern nur die dritte Bedeutung von `static` in C++: Statische Attribute und Methoden beziehen sich auf die gesamte Klasse, nicht auf einzelne Instanzen davon. Sie heißen gerne *Klassen*variablen und *Klassen*methoden, im Unterschied zu den üblichen *Instanz*variablen und *Instanz*methoden.

Statische Attribute könnten beispielsweise zählen, wie viele Instanzen von `Meeting` erzeugt worden sind, und summieren, wie viel Zeit insgesamt eingeplant ist. Statische Methoden – oder besser noch statische Properties – könnten diese Werte zurückgeben.

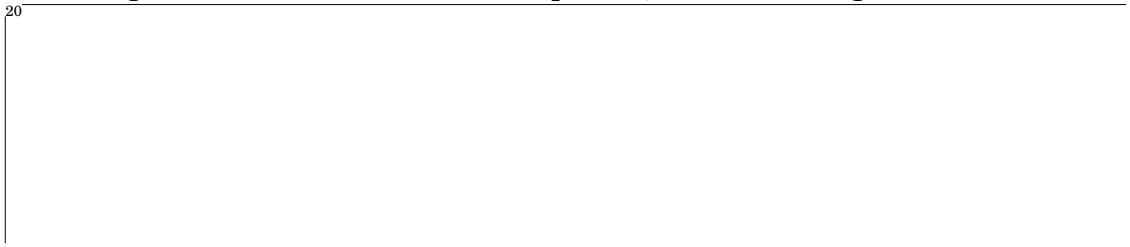
18


Achtung: In statischen Methoden und Properties kann man natürlich nur statische Elemente der Klasse benutzen, keine Elemente von Instanzen.

Statische Attribute, Methoden und Properties lassen sich in C# außerhalb der Klasse nur über den Klassennamen aufrufen, nicht über den Namen einer Instanz:

19


Hier einige statische Methoden und Properties, die wir schon gesehen haben:

20


7 Destruktor, Garbage Collection

In C++ gibt es neben dem Konstruktor noch einen Destruktor, um Objekte geregelt wieder abzubauen (Speicher freigeben, Dateien schließen usw.). Entsprechend sollte dort jedem `new` irgendwann ein `delete` folgen – wenn man es nicht vergisst, ein übliches Problem.

Das ist in Java und C# drastisch anders. Mit dem Freigeben von Speicher hat man dort nichts zu tun. Vielmehr wird nicht mehr benötigter Speicher automatisch gefunden (Garbage Collection).

Randnotizen für Fortgeschrittene

Das Beispiel mit `+=` ist ein bisschen geflunkert, weil `+=` in C# immer als `= ... + ...` ausgeführt wird, also mit einer Zuweisung. Es entspricht al-

so nicht einem reinen Methodenaufruf `a.foo(b)`, sondern einer Zuweisung `a = foo(a,b);`

`DateTime`, `TimeSpan` und `Complex` sind „immutable“ (unveränderlich). Das heißt: Deren Properties sind nur lesbar, aber nicht schreibbar und deren Methoden ändern nie etwas an den Daten, sondern geben im Zweifelsfall eine neue Instanz zurück. Insofern ist der gefühlte Unterschied zwischen `struct` und `class` hier nicht so groß: Wenn man sowieso nichts ändern kann, ist es recht egal, ob man mit einer Kopie arbeitet oder nicht. Deutlicher wird der Unterschied zum Beispiel mit `System.Windows.Point`.