BlitzMax OOP-Tutorial

Michael Zehr
(Jolinah)

1 Vorwort

Dieses Tutorial ist eher an Anfänger gerichtet, setzt jedoch einige Grundkenntnisse in BlitzMax voraus. Es sollte bekannt sein wie man:

- Variablen definiert
- Funktionen definiert und aufruft
- Listen benutzt (TList)

2 Was ist OOP?

OOP bedeutet Objekt orientiertes Programmieren. Beim OOP geht es darum in Objekten zu denken und zu programmieren, statt mit einzelnen zerstreuten Variablen zu arbeiten und diese nur virtuell (oder gar nicht) im Kopf zu einem Objekt zusammenzufassen.

Es liegt in der Natur des Menschen, in Objekten zu denken. Wenn wir die Adresse einer Person wollen, sagen wir auch nicht: „Kannst du mir den Namen, die Strasse, die PLZ, usw. von XYZ geben?“, sondern: „Kannst du mir die Adresse von XYZ geben?“. Und dabei ist jedem klar, was eine Adresse für Felder (Eigenschaften) hat.

Um das zu verdeutlichen, schauen wir uns ein mögliches Beispiel zur Adressen-Speicherung komplett ohne OOP an (BlitzBasic ohne Types):

```
Dim Name(5)
Dim Strasse(5)
Dim PLZ(5)

Name(0) = „Max Muster“
Strasse(0) = „Musterstrasse 14“
PLZ(0) = „09228 Musterstadt“
```

Natürlich gehört Name, Strasse und PLZ zu einer Adresse. Im Code-Beispiel stehen aber alle einzeln da, und man fasst diese Variablen nur im Kopf zusammen, als wären sie ein Objekt (bzw. eine Adresse).

Mit Programmiersprachen die OOP unterstützen, ist es möglich Objekte auch im Code, wirklich als solche darzustellen. Darüber hinaus kann es Beziehungen zwischen verschiedenen Objekttypen geben, was das Programmieren teilweise ziemlich erleichtern kann. Wie oben bereits erwähnt, ist es auch leichter eine Adresse zu übergeben, statt Name, Strasse und PLZ einzeln. Das heisst, durch OOP lässt sich auch viel Code einsparen.

Und wenn man OOP erstmal verstanden hat, dann macht es meistens auch einfach viel mehr Spass, und man möchte nur noch so programmieren ;-)

3 Types (auch bekannt als Klassen)

Ein Type ist eine Objektbeschreibung. Man kann Types auch als Baupläne betrachten, diese enthalten Informationen wie ein Objekt am Ende aussieht, und was es kann. In BlitzMax werden Types folgendermassen definiert:

```
Type MeinType
End Type
```

Da es nur ein Bauplan ist, existieren also noch keine Objekte, diese müssen mit dem Schlüsselwort **new** zuerst noch erstellt werden:

```
Local var:MeinType = new MeinType
```

New erstellt ein neues Objekt vom angegebenen Type und gibt das Objekt zurück. Dieses kann dann ganz gewöhnlich in einer Variable gespeichert werden. Um BlitzMax mitzuteilen, dass es sich hierbei um eine Variable handelt, die ein Objekt vom Type MeinType hält gibt man nach dem Doppelpunkt statt Int, String etc. einfach den Namen des Types an.

3.1 Field (Eigenschaft)

Der oben erstellte Type ist sozusagen ein leerer Bauplan, damit kann man nicht sehr viel anfangen. Ein Type kann aber auch Eigenschaften besitzen. Eigenschaften sind eigentlich normale Variablen, die aber später einem Objekt zugeordnet werden. Eigenschaften werden mit dem Schlüsselwort **Field** definiert. Um Beispielsweise eine Adresse zu speichern könnte man so einen Type verwenden:

```
Type Adresse
  Field Name:String
  Field Strasse:String
  Field PLZ:Int
End Type
```

Davon lässt sich jetzt wieder ein Objekt erstellen:

```
Local adresse1:Adresse = new Adresse
```

Um die Eigenschaften Name, Strasse und PLZ anzusprechen benutzt man den Punkt-Operator:

```
adresse1.Name = „Max Muster“
adresse1.Strasse = „Musterstrasse 14“
adresse1.PLZ = 23233
```

Eine Eigenschaft kann etwas beliebiges sein, nicht nur ein String oder ein Int. So lassen sich zum Beispiel zwei Types miteinander kombinieren (verschachteln):

```
Type Adresse
  Field Name:String
  Field Strasse:String
  Field PLZ:Int

  Field Zusatz:AdressZusatz
End Type

Type AdressZusatz
  Field Nickname:String
End Type
```

Dem Type Adresse wurde eine Eigenschaft **Zusatz** hinzugefügt. Diese Eigenschaft ist ein Objekt des Types AdressZusatz. Im Type AdressZusatz sieht man dann, dass dort noch ein Nickname eingetragen werden kann.

Nun wird von beiden Types ein Objekt erstellt:

```
Local adr:Adresse = new Adresse
Local zus:AdressZusatz = new AdressZusatz
```

Die Eigenschaft **Zusatz** ist jetzt aber noch nicht gesetzt. Wenn ein neues Objekt erstellt wird, werden die Eigenschaften auf einen Standardwert gesetzt. Bei Zahlenvariablen ist dies 0, bei Objekten Null. Es sei denn die Standardwerte werden im Konstruktor verändert, dazu aber später mehr.

Das heisst wir müssen die **Zusatz**-Eigenschaft erst noch setzen:

```
adr.Zusatz = zus
```

Auch bei verschachtelten Types lässt sich wieder mit dem Punkt-Operator auf die Eigenschaften zugreifen:

```
Print adr.Zusatz.Nickname
```

Ausserdem haben wir jetzt zwei Variablen, die auf das Selbe Objekt zeigen:

```
zus.Nickname = „Test“
adr.Zusatz.Nickname = „Test2“
Print zus.Nickname
```

Die Print-Anweisung gibt Test2 aus.

Da nirgendwo erneut **new** geschrieben wurde, haben wir auch kein neues Objekt erstellt. Eine Objektzuweisung kopiert ein Objekt auch nicht. Das heisst beide Variablen stellen das gleiche Objekt dar. Die Variable die nicht mehr benutzt wird könnte man jetzt auf ein Null-Objekt setzen:

```
zus = Null
```

Wenn eine Objektvariable auf Null gesetzt wird, zeigt diese Variable auf kein Objekt mehr, somit kann auch keine Eigenschaft etc. mehr angesprochen werden.

Folgendes würde jetzt nicht mehr funktionieren:

```
zus.Nickname = „Test“
```

3.2 Method (Methoden)

Eine Methode ist eine Funktion die einem Objekt zugeordnet wird. Sie kann nicht einfach von irgendwo aus aufgerufen werden, sondern nur über eine gültige Objektvariable. Eine Methode wird mit **Method** und **End Method** innerhalb eines Types definiert:

```
Type Adresse
  Field Name:String
  Field Strasse:String
  Field PLZ:Int

  Method Ausgeben()
    Print Name
    Print Strasse
    Print PLZ
  End Method
End Type
```

Wie gesagt kann die Ausgeben Methode noch nicht aufgerufen werden, folgendes ist nicht möglich:

```
Ausgeben()
Adresse.Ausgeben()
```

Dazu brauchen wir erst wieder ein Objekt. Und die Methode kann dann wieder mit dem Punkt-Operator von dem Objekt aus aufgerufen werden:

```
Local a:Adresse = new Adresse
a.Ausgeben()
```

Eine Methode kann wie jede Funktion auch Parameter haben:

```
..
Method Ausgeben(anzahl:Int)
  For Local i:Int = 1 to anzahl
    Print Name
    Print Strasse
    Print PLZ
  Next
End Method
..

a.Ausgeben(5)
```

Hat ein Parameter den Selben Namen wie eine Eigenschaft eines Objekts, so kann mit dem Schlüsselwort **Self**, welches das aktuelle Objekt anspricht, das Problem gelöst werden:

```
Method Set(Name:String)
  Name = Name   <-- so passiert nichts
  Self.Name = Name
End Method
```

3.2.1 Spezielle (reservierte) Methoden

Es gibt in BlitzMax zwei spezielle Methoden, **New()** und **Delete()**. Die Methode **New()** wird automatisch aufgerufen, bei der Erstellung eines Objekts:

```
Type Adresse
  ..
  Field Zusatz:AdressZusatz

  Method New()
    Zusatz = new AdressZusatz
  End Method
End Type

Local a:Adresse = new Adresse
```

Wenn jetzt also von Adresse ein neues Objekt erstellt wird, wird für dieses Objekt automatisch die Methode New() aufgerufen. In dieser Methode kann man nun Standardwerte setzen. Das ist sehr praktisch um zum Beispiel Objektvariablen zu setzen, so dass diese nicht Null sind, oder um Objekte gleich in eine Liste eintragen zu lassen.

Die New()-Methode ist in vielen Sprachen auch als **Konstruktor** bekannt. In BlitzMax darf New() jedoch keine Parameter besitzen, und dient nur zum initialisieren von Werten.

Die Methode **Delete()** wird, wie sich schon vermuten lässt, genau vor dem Löschen eines Objekts aufgerufen. Dort könnte man zum Beispiel manuell reservierten Speicher wieder freigeben. In der Regel ist man auf Delete() jedoch eher selten angewiesen.

In anderen Sprachen ist Delete() auch als **Destruktor** bekannt.

3.3 Global (statische Eigenschaften)

Global hat in BlitzMax zwei Bedeutungen, einerseits wird es benutzt um globale Variablen zu definieren, und innerhalb von Types um statische bzw. globale Eigenschaften zu definieren.

Statische Eigenschaften sind den normalen Eigenschaften sehr ähnlich, jedoch sind sie nicht mehr an ein Objekt gebunden, sondern an den Type bzw. an alle Objekte dieses Types gleichzeitig.

```
Type Adresse
  Global Anzahl:Int

  Field Name:String
  Field Strasse:String
  Field PLZ:Int

  Method New()
    Anzahl = Anzahl + 1
  End Method
End Type

Local a:Adresse = new Adresse
Local b:Adresse = new Adresse

Print Adresse.Anzahl
```

Die statische Eigenschaft **Anzahl** wird immer wenn ein neues Objekt vom Type Adresse erstellt wird, um eins erhöht. Von aussen kann auf diese statische bzw. globale Eigenschaft über den Namen des Types mit Hilfe des Punkt-Operators zugegriffen werden. Demzufolge gibt die Print-Anweisung am Ende **2** aus.

3.4 Function (statische Methoden, bzw. Funktionen)

Auch **Function** kommt in BlitzMax zwei Mal vor. Innerhalb von Types werden damit statische Methoden definiert. Statische Methoden sind Funktionen, die ähnlich den statischen Eigenschaften, nicht an ein Objekt gebunden sind.

```
Type Adresse
  Field Name:String
  Field Strasse:String
  Field PLZ:Int

  Function Create:Adresse(Name:String, Strasse:String, PLZ:Int)
    Local a:Adresse = new Adresse
    a.Name = Name
    a.Strasse = Strasse
    a.PLZ = PLZ
    Return a
  End Function
End Type

Local a:Adresse = Adresse.Create(„Max", „Musterstr. 14", 87387)
```

Wie bei den statischen Eigenschaften können diese von Aussen über den Namen des Types aufgerufen werden, wieder mit dem Punkt-Operator. Eine statische Methode kann auch von einer normalen Methode heraus aufgerufen werden, dort muss jedoch der Typename nicht vorangestellt werden:

```
..
Method Test()
  Create(„Max", „Musterstr. 14", 87387)
End Method
..
```

3.5 Const (Konstanten innerhalb von Types)

Das Schlüsselwort **Const** kann auch innerhalb von Types benutzt werden. In diesem Fall ist die Konstante auch nur innerhalb des Types gültig, kann aber von Aussen über den Namen des Types angesprochen werden. So wie üblich, wird dazu wieder der Punkt-Operator benutzt:

```
Type Adresse
  Const Header:String = „Adresse:"

  Field Name:String
  Field Strasse:String
  Field PLZ:Int

  Method Drucken()
    Print Header
    Print Name
    Print Strasse
    Print PLZ
  End Method
End Type

Print Adresse.Header
```

Innerhalb von Methoden muss der Typename nicht vorangestellt werden, von Aussen ist dies jedoch nötig. Wie bei Konstanten üblich kann der Wert nachträglich nicht mehr geändert werden. Die Zuweisung muss deshalb direkt bei der Definition innerhalb des Types gemacht werden, sonst wird ein Compiler-Fehler ausgelöst.

4 Extends (Ableitung bzw. Vererbung)

Mit dem Schlüsselwort **Extends** kann ein Type sozusagen kopiert und danach erweitert werden, dabei entsteht eine „Ist..ein“-Beziehung:

```
Type Fahrzeug
  Field Geschwindigkeit:Int
End Type

Type Auto Extends Fahrzeug
End Type

Local a:Auto = new Auto
a.Geschwindigkeit = 10
```

Hier wurde ein allgemeiner Bauplan (Type) für Fahrzeuge erstellt. Es wird davon ausgegangen, dass jedes Fahrzeug auch eine Geschwindigkeit hat.

Mit **Extends** wird der Bauplan vom Fahrzeug kopiert und Auto benannt. Das Auto **ist** jetzt also auch **ein** Fahrzeug. Deshalb hat das Auto auch die Eigenschaft Geschwindigkeit vom Type Fahrzeug geerbt. Wie man im Beispiel sieht, wird bei einem Auto die Geschwindigkeit auf 10 gesetzt.

Dadurch ergibt sich auch die Möglichkeit einer Fahrzeug-Variable ein Auto zuzuordnen:

```
Local a:Auto = new Auto
Local b:Fahrzeug = a
```

Die zweite Anweisung sieht für Personen die bisher nur prozedural programmiert haben natürlich falsch aus. Sie ist aber völlig korrekt, dank der Ist-Beziehung zwischen Fahrzeug und Auto, die mit **Extends** geschaffen wurde.

Jedoch klappt das nur auf einen Weg. Ein Auto ist in jedem Fall ein Fahrzeug, ein Fahrzeug muss aber nicht zwingend ein Auto sein. Folgende Anweisung funktioniert **nicht**:

```
Local b:Fahrzeug = new Fahrzeug
Local a:Auto = b
```

Wie bereits ganz am Anfang erwähnt lassen sich abgeleitete Types (Auto) unabhängig vom Basistype (Fahrzeug) erweitern:

```
Type Auto Extends Fahrzeug
  Field Reifen:Int

  Method Hupen()
    Print „HUPE!“
  End Method
End Type
```

Ein Auto hat jetzt Reifen, und kann Hupen ;-) Nach wie vor behält ein Auto die Eigenschaft Geschwindigkeit von dem Fahrzeug-Type:

```
Local a:Auto = new Auto
a.Geschwindigkeit = 10
a.Reifen = 4
a.Hupen()
```

Wenn jetzt aber die Regel von vorhin angewandt wird, die besagt dass ein Auto auch einer Fahrzeug-Variable zugeordnet werden kann, fällt etwas auf: Der Fahrzeug-Type kennt Reifen nicht, und kann auch nicht Hupen. Damit ein Fahrzeug das kann, muss es eben ein Auto sein. Es ist aber möglich ein Auto einer Fahrzeug-Variable zuzuordnen, und diese dann wieder in ein Auto zu konvertieren. Das erscheint auf den ersten Blick unlogisch, aber es ist sehr nützlich:

```
Type Bus Extends Fahrzeug
  Field Passagiere:Int
End Type

Local liste:TList = new TList
Local a:Auto = new Auto
Local b:Bus = new Bus

liste.AddLast(a)
liste.AddLast(b)

For local f:Fahrzeug = EachIn liste

  Print f.Geschwindigkeit

  If Auto(f)
    a = Auto(f)
    Print a.Reifen
    a.Hupen()

  ElseIf Bus(f)
    B = Bus(f)
    Print b.Passagiere
  EndIf

Next
```

Zuerst wird ein weiterer Type **Bus** erstellt, der ebenfalls ein Fahrzeug ist. Anschliessend werden ein Auto und ein Bus erstellt und einer Liste hinzugefügt. Diese Liste wird mit For..EachIn durchgegangen, allerdings wird der Type **Fahrzeug** dazu verwendet. Dadurch werden die Objekte in der Liste als Fahrzeug dargestellt, das heisst sie kennen nur die Eigenschaft Geschwindigkeit. Die Anweisung Typename(Objektvariable) konvertiert die Objektvariable in den Type der vor der Klammer steht. Kann das Objekt konvertiert werden, so wird natürlich das konvertierte Objekt zurückgegeben, wenn aber nicht konvertiert werden kann, so wird Null zurückgegeben. In einer If-Anweisung bedeutet Null False, deshalb kann so geprüft werden, ob das Fahrzeug ein Auto oder ein Bus ist. Je nachdem wird das Objekt dann noch einmal konvertiert und in einer Variable des richtigen Types gespeichert. Daraufhin kann ein Auto wieder wie ein Auto, und ein Bus wie ein Bus gehandhabt werden.

So kann zum Beispiel eine GUI sehr gut programmiert werden, da die meisten Elemente einer GUI eine X-, sowie eine Y-Position und Grösse besitzen. Diese müssen nun nicht in jedem Type immer wieder neu geschrieben werden, sondern es wird ein Basistype erstellt, der genau diese Eigenschaften besitzt:

```
Type Control
  Field X:Int, Y:Int
  Field Width:Int, Height:Int

  Method Move(xval:Int, yval:Int)
    X :+ xval
    Y :+ yval
  End Method
End Type

Type Button Extends Control
  Field Pushed:Int

  Method Push()
    Pushed = Not Pushed
  End Method
End Type

Local b:Button = new Button
b.X = 10
b.Y = 10
b.Push()
b.Move(30, 0)
```

Der Button ist ein Control, und jedes Control hat eine Position und eine Grösse und lässt sich mit Move() bewegen. Zusätzlich hat der Button aber noch eine Eigenschaft die festhält, ob er gedrückt ist oder nicht, und eine Push()-Methode die den Knopf umschaltet.

Ab hier sollten die Vorteile von OOP langsam zu erkennen sein. Durch die Vererbung mit **Extends** lässt sich viel Code sparen, weil nicht alles immer doppelt und dreifach geschrieben werden muss. Weiterhin lässt sich Editier-Arbeit sparen. Wenn etwas Grundlegendes an allen Elementen geändert werden muss, so reicht es nur den Basistypen zu editieren, und schon profitieren alle anderen Elemente von dieser Änderung. Ohne OOP müsste man hier für jedes Element einzeln alles umschreiben.

Das Tutorial ist aber noch nicht zu Ende, denn OOP bietet noch mehr ;-)

5 Methoden überschreiben

Methoden von Basistypen können in erweiterten Typen überschrieben werden. Dies ist ganz einfach:

```
Type Basis
  Method TuWas()
    Print „Hallo“
  End Method
End Type

Type Erweitert Extends Basis
  Method TuWas()
    Print „Was anderes“
  End Method
End Type

Local b:Basis = new Basis
Local e:Erweitert = new Erweitert

b.TuWas()
e.TuWas()
```

In dem erweiterten Type wurde die Methode TuWas() überschrieben, sie macht jetzt etwas anderes als die Basis-TuWas()-Methode. Voraussetzung zum Überschreiben einer Methode ist, dass sowohl die Basismethode, wie auch die überschriebene Methode die Selben Parameter und natürlich auch den Selben Namen haben.

Falls verschiedene erweiterte Objekte in eine Liste gepackt werden, und man sie wieder mit einem For..EachIn mit einer Basisobjekt-Variable durchgehen will, so könnte man nun natürlich vermuten, dass man die Objekte auch wieder zuerst konvertieren muss, damit dann auch die richtige Methode des richtigen Objekts aufgerufen wird. Bei Methoden ist das aber nicht so:

```
Type Basis
  Method Draw()
    Print „Basis“
  End Method
End Type

Type A Extends Basis
  Method Draw()
    Print „A“
  End Method
End Type

Type B Extends Basis
  Method Draw()
    Print „B“
  End Method
End Type

Local liste:TList = new TList
Local aobj:A = new A
Local bobj:B = new B

liste.AddLast(aobj)
liste.AddLast(bobj)

For local bas:Basis = EachIn liste
  bas.Draw()
Next
```

Mit der Anweisung bas.Draw() wird für jedes Objekt die richtige Methode aufgerufen, die Ausgabe lautet also:

```
A
B
```

Das funktioniert bei den Methoden, weil der Basistype die Methode ja auch kennt. Bei Eigenschaften jedoch, kennt der Basistype die erweiterten Eigenschaften gar nicht, daher muss das Objekt in diesem Fall erst konvertiert werden. Beim konvertieren wird nicht der Type des Objekts verändert, es wird nur eine andere Variable benutzt um das Selbe Objekt anzusprechen. Diese Variable bildet den Type ab von dem sie ist. Wenn der Basistype gewisse Eigenschaften nicht kennt, so kann mit einer Variable dieses Types auch nicht auf diese zugegriffen werden. Trotzdem hätte das Objekt im Speicher diese Eigenschaften.

5.1 Super

Neben Self gibt es ein weiteres Schlüsselwort: **Super** wird ähnlich wie Self benutzt um innerhalb von Methoden auf das aktuelle Objekt zuzugreifen, es kommt jedoch nur innerhalb von abgeleiteten Types zur Anwendung. Mit Super kann auf die Methoden des übergeordneten Types zugegriffen werden. Genau genommen zeigt **Super** auf das gleiche Objekt wie Self, nur wird das Objekt vorher in den Basistype konvertiert. Anders als bei normalen Konvertierungen, werden dann jedoch die Methoden des Basistypes aufgerufen, statt die des aktuellen Types.

Das ist sehr nützlich um Methoden zu erweitern, es lässt sich dann die alte Methode aufrufen und danach noch etwas anfügen etc.:

```
Type Basis
  Method Drucken()
    Print „Das ist ein Text“
  End Method
End Type

Type BasisEx Extends Basis
  Method Drucken()
    Print „------------------„
    Super.Drucken()
    Print „------------------„
  End Method
End Type
```

Die Ausgabe bei einem Objekt des Types Basis lautet:

```
Das ist ein Text
```

Bei einem Objekt des Types BasisEx lautet die Ausgabe jedoch:

```
------------------
Das ist ein Text
------------------
```

Mit **Super** lassen sich nur Methoden des Basistypes ansprechen, keine Eigenschaften, da diese im abgeleiteten Type ja sowieso identisch sind. Das funktioniert bei abstrakten Methoden, die im nächsten Kapitel beschrieben werden, jedoch nicht.

6 Abstrakte Types und abstrakte Methoden

Mit Abstrakten Types sind Types gemeint, die nur als Vorlage für andere Types dienen, und von denen man gar keine Objekte erstellen können soll. Als bestes Beispiel eignet sich hier die GUI von vorhin. Es ist zwar wichtig dass der Type Control existiert, niemand muss davon aber ein Objekt erstellen können. Deshalb gibt es das Schlüsselwort **Abstract**, das sowohl hinter einen Type, aber auch hinter eine Methode geschrieben werden kann:

```
Type Control Abstract
  Field X:Int, Y:Int
  ..

  Method TuWas() Abstract
End Type

Local c:Control = new Control <-- geht nicht
```

Jetzt kann von Control kein Objekt mehr erstellt werden, nur von den Types die darauf basieren, wie z.B. der Button. Trotzdem kann ein Button immer noch in ein Control konvertiert werden, und beim For..EachIn funktioniert es auch noch mit einer Control-Variable.

Abstrakte Methoden, sind Methoden ohne Implementation. Deshalb wird auch kein End Method benötigt. Das bedeutet, diese Methode muss dann in jedem Type der den Control-Type erweitert noch definiert werden:

```
Type Button Extends Control
  Method TuWas()
    Print „Hallo“
  End Method
End Type

Type RadioButton Extends Control
  Method TuWas()
    Print „Was anderes“
  End Method
End Type
```

Wenn bei einem Type kein **Abstract** steht, jedoch eine Methode in diesem Type Abstrakt ist, so wird der Type trotzdem auch Abstrakt, denn ein Objekt mit einer Methode die nicht implementiert wurde kann nicht verwendet werden.

7 Final

Das Schlüsselwort **Final** bewirkt praktisch das Gegenteil von Abstract. Während Abstract sozusagen verlangt, dass der Type noch abgeleitet wird um Objekte erzeugen zu können, so kann von einem Type der mit **Final** gekennzeichnet wurde keine Ableitung mehr erfolgen:

```
Type Fahrzeug Final
  Field Geschwindigkeit:Int
End Type

Type Auto Extends Fahrzeug <-- geht nicht mehr
  ..
End Type
```

In diesem Beispiel kann vom Type Fahrzeug nicht mehr abgeleitet werden, die Definition des Type Auto mit Extends Fahrzeug, würde einen Compiler-Fehler auslösen. Im Gegensatz zu Abstract können vom Type Fahrzeug natürlich Objekte erzeugt werden, sonst würde es ja keinen Sinn ergeben.

Das Schlüsselwort **Final** lässt sich auch auf Methoden anwenden:

```
Type Control Abstract
  Field X:Int, Y:Int

  Method Move(x:Int, y:Int) Final
    Self.X = x
    Self.Y = y
  End Method
End Type

Type Button Extends Control
  Method Move(x:Int, y:Int) <-- geht nicht
    Print "Hallo"
  End Method
End Type
```

Methoden die mit **Final** gekennzeichnet wurden, können in abgeleiteten Types nicht mehr überschrieben werden. Damit kann sichergestellt werden, dass andere Types nicht aus versehen etwas Falsches machen.

Allgemein gesagt, bedeutet Final etwa soviel wie fertig bzw. endgültige Version, aus diesem Grund können auch keine „neuen Versionen“ mehr erstellt werden, sowohl bei Types, wie auch bei Methoden.

Die beiden Schlüsselwörter Abstract und Final sind besonders für die Modulentwicklung geeignet, wo die Endbenutzer den Code evtl. nicht einsehen können. Damit lässt sich vermeiden dass andere Programmierer unvorhergesehene Sachen machen. Soll zum Beispiel von einem Type das verhalten nicht durch andere Programmierer beeinflusst werden können, so kann man Final benutzen. Stellt man einen Type zur Verfügung der eigentlich nur als Vorlage für etwas dient und die minimalen Eigenschaften und Methoden enthält die benötigt werden, so kann man diesen Type einfach als Abstrakt definieren, und die Programmierer sind gezwungen einen eigenen Type zu erstellen, der davon abgeleitet wird. Mit abstrakten Methoden wird der Programmierer auch dazu gezwungen mindestens diese Methoden selber im neuen Type zu definieren.

8 Schlusswort

Natürlich hat OOP auch einige Nachteile, es wird ein bisschen mehr Speicher verbraucht und es kann langsamer sein, als gewisse prozedurale Funktionen. Aber es ist immer noch sehr schnell, und wenn nicht gerade Systemnahe oder sehr anspruchsvolle Zeitintensive Programme geschrieben werden fallen die Nachteile nicht ins Gewicht.

Schreibfehler dürft ihr behalten ;-) Ich hoffe, dass ich damit evtl. ein paar Personen den Einstieg in BlitzMax erleichtern konnte, da es zu diesem Thema, zumindest BlitzMax spezifisch, noch nicht sehr viel Literatur gibt.