6 Objektorientiertes Programmieren

Ziel: Hier erfahren Sie die Grundlagen des Objektorientierten Programmierens, damit Sie in der Praxis saubere Programm-Architekturen entwerfen und diese dann ausprogrammieren können. Da dieses Kapitel eine ausführliche Schritt-für-Schritt Anleitung enthält, eignet es sich zum Selbststudium.

Als objektorientierte Programmiersprache erfüllt .NET folgende Kriterien:

- **Abstraktion**: Darunter verstehen wir die Möglichkeit, „black box" Code zu schreiben: ein Customer Objekt ist die Abstraktion eines Kunden, ein DataTable Objekt ist die Abstraktion einer Tabelle.
- **Kapselung**: Wir trennen Interface (die Schnittstelle nach außen) und die Implementierung (wie ist etwas intern programmiert). Solange das Interface eines Objekts gleich bleibt, kann der Code innen beliebig oft geändert werden, ohne dass irgendjemand etwas merkt. So kann z.B. ein langsames durch ein schnelles Sortierverfahren ausgetauscht werden, ohne andere Code Teile zu beeinflussen.
- **Polymorphismus**: Polymorphismus ermöglicht uns, Objekte verschiedener Klassen einheitlich zu behandeln.
- **Vererbung**: Ein Klasse kann die Zustände(Daten) und das Verhalten (Methoden) einer anderen Klasse erben, d.h. übernehmen und anschließend diese Funktionalität erweitern.
- **Mehrfache Interfaces**: Eine Klasse kann mehr als ein Interface implementieren.

Das klingt sehr trocken, und wir wollen uns die Dinge deshalb gleich an einem ausführlichen Beispiel ansehen. Es führt uns durch die Welt der Klassen, Objekte, Methoden, Eigenschaften, Events, Konstruktoren und Destruktoren.

Im Mittelpunkt steht die Klasse Person. Eine Klasse ist wie ein Bauplan, nach ihrer Vorlage können Objekte erzeugt werden. Mit Objekten kann man in einem Programm „arbeiten".

Eine Klasse besteht aus folgenden Elementen:

- Variables
- Methods (Subroutines, Functions)
- Properties
- Constructors
- Events

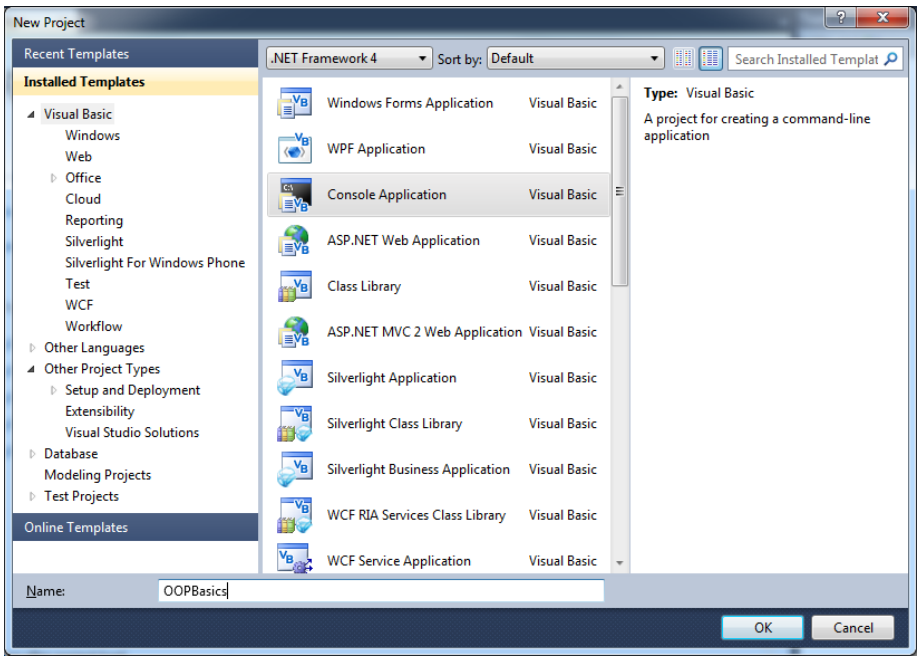
die wir in diesem Kapitel kennenlernen.

Das Beispiel mit der Personen Klasse findet sich in vielen Büchern. Hier ist es am aktuellen Stand von VS.2010 und .NET 4.0.

Wenn auf den nächsten Seiten Codezeilen gelb hinterlegt sind, so sind diese in unser Beispielprogramm einzugeben bzw. im Code zu ändern.

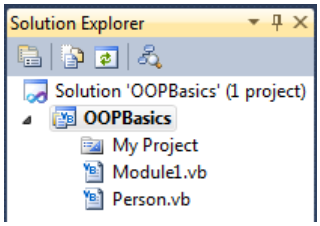
6.1 Beispiel Personen Klasse

Schritt 1: Starten Sie VS und erzeugen Sie ein neues Konsole-Projekt namens OOPBasics.



Schritt 2: Speichern Sie es in C:\Beispiele\Kapitel06.

Schritt 3: Wechseln Sie zum Solution Explorer. Klicken Sie mit der rechten Maustaste auf das Project OOPBasics und wählen Sie Add > Class… Benennen Sie die Klasse Person. Der Solution Explorer zeigt nun:



Eine Klasse besteht aus Member Variables, Methods, Properties, Events, Constructors und Destructor. Diese wollen wir nun schrittweise in unsere Personenklasse einbauen.

6.2 Member Variables

Member Variable, auch Instance Variable genannt, werden in der Klasse definiert und sind in jedem Objekt, das von der Klasse instantiiert wird, einmal vorhanden. D.h. wenn wir von der Klasse Person, die einen Namen hat, die Objekte Otto und Eva anlegen, dann hat Otto einen Namen und Eva einen anderen Namen.

Schritt 4: Fügen Sie folgende Zeile in die Klasse Person ein:

```
Public Class Person
    Private mName As String
    Private mBirthdate As Date
End Class
```

Es ist eine Konvention, dass Variablennamen mit einem Präfix beginnen, das anzeigt, dass es sich um eine lokale Variable handelt. Das Präfix ist typisch „m“ oder „_“. Manchmal enthält es auch einen Hinweis auf den Datentyp der Variablen. Es gibt im Internet Aufstellungen solcher Regeln, auch von Microsoft. Leider ändert Microsoft seine Regeln immer wieder, sodass ich Ihnen folgenden Tipp gebe: suchen Sie sich jene Naming Convention, die Ihnen zusagt.

Wir regeln den Zugriff auf Variable (Scope) durch folgende Schlüsselwörter: Private, Friend, Protected, Protected Friend und Public. Typischerweise sind Variable als Private deklariert. Ein anderes wichtiges Schlüsselwort ist ReadOnly. Solchen Variablen kann nur bei der Deklaration oder im Konstruktor ein Wert zugewiesen werden.

6.3 Methoden, Subroutines

Objekte brauchen typischerweise Methoden, die andere Objekte aufrufen können. Diese verwenden die lokalen Daten und eventuell übergebene Parameter um etwas zu tun. Der Zugriff auf Methoden wird mit den Schlüsselwörtern Private, Friend, Protected, Protected Friend und Public geregelt. VB bietet zwei Arten an Methoden: Subroutines (ohne Rückgabewert) und Functions (mit Rückgabewert).


Schritt 5: Fügen Sie den folgenden Code in der Klasse Person hinzu:

```
Public Sub Walk()
    'ToDo: implementation missing
End Sub
```

Um diesen Code zu nutzen, verwendet man typisch Code in der Form

```
Dim myPerson As New Person
myPerson.Walk()
```

6.4 Methoden mit Rückgabewert, Functions

Eine Methode, die etwas an den Aufrufer zurückgibt, heißt Function.


Schritt 6: Fügen Sie den folgenden Code in der Klasse Person hinzu:

```
Public Function Age() As Integer
    'this can be 1 to high if this year's birthdate is in the future
     Dim age1 = Date.Now.Year - mbirthDate.Year

     If mbirthDate.AddYears(age1) > Date.Now Then 'this year's birthdate
        age1 -= 1                                 'correct age
     End If
     Return age1
End Function
```

Sie können diese Funktion beispielsweise so benutzen:

```
Dim myPerson As New Person
Dim age As Integer
age = myPerson.Age
```

6.5 Methoden mit Parametern

Manchmal muss man einer Methode beim Aufruf Daten mitgeben.

Schritt 7: Erweitern und ändern Sie den Code:

```
Public Class Person
    Private mName As String
    Private mBirthdate As Date
    Private mTotalDistance As Integer

    Public Sub Walk(ByVal distance As Integer)
        mTotalDistance += distance
    End Sub
```

Wenn wir nun eine Person auffordern, zu gehen, sagen wir auch, wie weit gegangen werden soll. Und jedes Objekt der Klasse Person summiert seine gegangenen Entfernungen auf. Sie können diese Subroutine beispielsweise so benutzen:

```
Dim myPerson As New Person
myPerson.Walk(5)
```

Das Schlüsselwort ByVal bewirkt, dass eine Kopie des Wertes 5 an die Subroutine Walk übergeben wird. Daher könnten wir (da es sich um einen Value Type handelt), den Parameter auch innerhalb von Walk verändern, ohne dass dies den Aufrufer betrifft.

Per Konvention werden Parameternamen klein geschrieben.

Wenn wir einen Value Type Parameter ändern wollen und diese Änderung soll auch der Aufrufer erhalten, verwenden wir das Schlüsselwort ByRef.

Achtung: Beim Programmieren sehen Sie in der Intellisense Anzeige, ob ein Parameter ByVal oder ByRef übergeben wird! Dabei wird ByVal unterdrückt, ByRef wird angezeigt.

6.6 Properties

Eine Property ist eine spezielle Art von Methode, die aus einer Art Function zum Lesen einer Variablen und einer Art Subroutine zum Schreiben (Setter) derselben Variablen besteht. In unserem Beispiel bietet sich die private Variable mName an, mit einer Property von außen benutzt zu werden. Anstatt eine Subroutine und eine Function zu codieren, schreiben wir kurz eine Property.

Schritt 8: Fügen Sie den Code ein:

```
    Public Property Name() As String
        Get
            Return mName
        End Get
        Set(value As String)
            mName = value
        End Set
    End Property
```

Die Property wird dann so verwendet, wobei kein Unterschied zur Lösung mit Subroutine/Function oder mit einer globalen Variablen zu sehen ist:

```
        Dim myPerson As New Person
        myPerson.Name = "Feichtinger"
        Console.WriteLine(myPerson.Name)
```

Ein Vorteil der Property liegt in der einfacheren Wartung, da der Get und Set Teil immer zusammen bleiben. Die Definition der Property enthält einen Scope und einen Datentyp. Der Set Teil verlangt einen Parameter von diesem Datentyp, der im Code den Namen Value trägt.

Properties haben eine Reihe von Vorteilen: Sie dienen der Kapselung von Variablen. Statt einem Aufrufer ungehinderten Zugriff auf eine Public Variable zu geben, ist es besser die Variable Private zu deklarieren und mit einer Public Property zu versehen. Sowohl in den Get als auch in den Set Teil kann weiterer Code geschrieben werden, z.B. zum Schutz gegen falsche Daten (Validierung) oder zur Umrechnung in andere Einheiten. Auch das Synchronisieren des gleichzeitigen Zugriffs aus mehreren Threads kann in Properties erfolgen. Der Get-Teil kann einen Default Wert zurückliefern, der Set-Teil ein ValueChangedEvent auslösen.

Beispielsweise könnte der Set-Teil so aussehen:

```
Set(value As String)
   If String.IsNullOrWhiteSpace(value) Then Throw New Exception("Invalid name.")
   mName = value
End Set
```

Wenn der übergebene Name ein leerer String ist oder gar nicht existiert, wird ein Fehler gemeldet.
Beispielsweise könnte der Set-Teil so aussehen:

```
Get
    If String.IsNullOrWhiteSpace(mName) Then Return "not defined"
    Return mName
End Get
```

Properties werden in vielen Code-Generatoren verwendet, um Objekte zu verpacken. Andererseits gibt es andere Programmiersprachen oder .NET APIs wie WCF, die keine Properties kennen. Erlaubt man diesen den Zugriff auf lokale Objekte, scheitern diese an den Properties.

Für den Anfänger, der keine Netzwerk-Programme schreibt, sind Properties ideal.

In C# ist es üblich, die lokale Variable klein zu schreiben und die zugehörige Property mit einem Großbuchstaben beginnen zu lassen. Das geht in VB nicht, da VB case insensitive ist, Groß- und Kleinbuchstaben nicht unterscheidet. Daher die Regel: private Variable beginnen mit „m“ (oder _).

In WPF und Silverlight gibt es neben diesen normalen Properties auch sogenannte „Attached Properties“. Diese funktionieren ganz anders und werden in diesem Buch nicht weiter behandelt.

Properties können mit dem Schlüsselwort ReadOnly auf den Get Teil bzw. mit WriteOnly auf den Set Teil beschränkt werden.

Auto-implemented Properties erlauben die Definition einer Property und der nicht sichtbaren lokalen Variablen dahinter in kurzer Schreibweise. Die Variable ist nur über die Property erreichbar und in der Property kann keine Logik enthalten sein. Auch ein initialer Wert kann zugewiesen werden:

```
Public Property Name() As String = "Feichtinger"
```

6.6.1 Parameterized Properties

Viele Personen haben mehr als eine Telefonnummer. Wir können diese Situation mittels einer parameterisierten Property abbilden, indem wir zu jeder Telefonnummer auch eine Beschreibung speichern. Die Verwendung sieht so aus:

```
Dim myPerson As New Person
myPerson.Phone("mobile") = "(212) 555 1234"
Dim homePhone As String = myPerson.Phone("home")
```

Da wir nun mehrere Nummern pro Person speichern, reicht eine einfache Variable nicht mehr aus und wir müssen bei der Verwendung der Property jeweils die zur Nummer gehörige Beschreibung verwenden. Als Speicher für die Nummern und Beschreibungen bietet sich eine Collection, z.B. eine HashTable an. Wir haben schon gelernt, dass Collections i.d.R. besser als Arrays sind. Eine Hashtable speichert Name/Value Paare ab. d.h. ein Wert wird unter seinem Namen gespeichert und mit dem Namen wieder gefunden.

Schritt 9: Fügen Sie den Code ein:

```
Public Class Person
    Private mName As String
    Private mBirthdate As Date
    Private mTotalDistance As Integer
    Private mPhones As New Hashtable

    Public Property Phone(ByVal location As String) As String
        Get
            Return CStr(mPhones.Item(location))      'convert to string
        End Get
        Set(value As String)
            If mPhones.ContainsKey(location) Then    'if location already exists
                mPhones.Item(location) = value       'then overwrite it
            Else
                mPhones.Add(location, value)         'else insert new value
            End If
        End Set
    End Property
```

Die Property Phone besitzt den Parameter location. Unter diesem Namen wird der Wert der Telefonnummer gespeichert bzw. gesucht. Der Parameter location steht im Get und Set Teil zur Verfügung. Wenn im Get Teil keine Telefonnummer mit dem Namen location gefunden wird, kommt es zu einem Laufzeitfehler. Wie man diesen abfängt, sehen wir später noch.

Die HashTable merkt sich nicht, von welchem Datentyp die gespeicherten Telefonnummern sind. Daher wird im Get Teil ein Object von der HashTable zurückgeliefert, das in einen String umzuwandeln ist. Durch Verwendung einer Generic Collection wäre dieser Schritt nicht notwendig, wie wir später noch sehen werden, da sich Generic Collections den Type merken.

6.6.2 Default Property

Default Properties erlauben dem Aufrufer eine verkürzte Schreibweise. Sie machen Sinn, wenn es eine Property gibt, die signifikant wichtiger ist, als alle anderen Properties einer Klasse. Default Properties müssen parametrisiert sein und es kann nur eine Default Property pro Klasse geben.

Schritt 10: Erweiten Sie die Zeile:

```
Default Public Property Phone(ByVal location As String) As String
```

Die Benutzung der Default Property sieht nun so aus:

```
Dim myPerson As New Person
myPerson("mobile") = "(212) 555 1234"
Dim homePhone As String = myPerson("home")
```

Der Schreibaufwand sinkt und die Lesbarkeit steigt durch die Default Property.

Hinweis: US Telefonnummern, die mit 555 beginnen, wie (212) 555 1234 dürfen per US Gesetz nicht existieren und sind daher ideal für Beispiele geeignet. Verwenden Sie niemals echte Namen, Adressen, URLs, IP Adressen, Telefonnummern usw. in Beispielen. Die Firma Compaq hat einmal weltweit eine Telefonnummer verwendet, ohne dies zu überprüfen. In Frankreich war diese eine Notrufnummer – und die französischen Kunden, die die Beispiele nachmachten, mussten dann für jeden Anruf Strafe zahlen.

6.7 Events

Mit Methoden und Properties sagt ein Aufrufer einem Objekt, was es tun soll. Der Zeitpunkt wird damit immer vom Aufrufer bestimmt. Aber was ist, wenn ein Objekt von sich aus etwas Wichtiges zu sagen hat? Z.B. die gemessene Temperatur überschreitet einen Grenzwert oder der Eurokurs fällt unter $ 1,30. Wir können nicht warten, bis jemand mit einer Funktion vielleicht irgendwann den Wert ausliest – es muss u.U. sofort reagiert werden.

Für solche Benachrichtigungen gibt es einen Event Mechanismus. Er läuft schnell ab, belastet den Rechner wenig und ist für normale Ereignisse gedacht, also nicht für Fehlermeldungen. Eine Programmiersprache, die keine Events hat, wäre nur beschränkt brauchbar. In .NET basieren Events auf dem Mechanismus von Delegates. Events und Delegates werden in .NET intensiv eingesetzt. In VB sehen wir die Delegates hinter den Events nicht, sie werden für uns automatisch erzeugt und verwaltet. Ein C# Programmierer codiert die Delegates zu den Events selbst.

Ein Event kann von mehreren Empfängern erhalten werden. Der Auslöser eines Events weiß dabei nicht, wer auf das Event reagieren wird. Wenn es mehrere Empfänger des Events gibt, ist auch die Reihenfolge der Benachrichtigung im Standardfall nicht vorhersagbar. Die Event Handler werden einer nach dem anderen abgearbeitet.

6.7.1 Handling Events

Im vorangehenden Kapitel haben wir den Code zur Behandlung des Button Click Events gesehen:

```
Private Sub btnOK_Click(sender As System.Object, e As System.EventArgs) _
Handles btnOK.Click
End Sub
```

Die Subroutine besitzt zwei Parameter. Der erste ist das Objekt, welches den Event auslöste, der zweite enthält nähere Informationen zum Event. Am Ende der Zeile findet sich das Schlüsselwort Handles. Handles btnOK.Click sagt aus, dass diese Subroutine auf das Event Click des btnOk Objekts reagiert.

Der Name der Subroutine, btnOK_Click, und die Namen der Parameter können geändert werden, ohne dass sich dies auswirkt.

Eine Subroutine kann auch auf mehrere Events reagieren und auf dasselbe Event mehrere Objekte. Die einzige Bedingung ist, dass die Parameterliste der Subroutine und aller Events gleich sein muss. Ein typisches Beispiel ist das Reagieren auf das Click Event aller Buttons:

```
Private Sub MyButtons_Click(sender As System.Object, e As System.EventArgs) _
```

```
Handles btnB1.Click, btnB2.Click
End Sub
```

6.7.2 Raising Events

Um ein Event auslösen zu können (engl. to raise an event), muss zunächst in der Klasse das Event definiert werden. In unserem Beispiel wollen wir jedesmal, wenn die Person gegangen ist, die gegangene Distanz mit einem Event namens Walked melden.

 Schritt 11: Fügen Sie den Code ein:

```
Public Class Person
    Private mName As String
    Private mBirthdate As Date
    Private mTotalDistance As Integer
    Private mPhones As New Hashtable

    Public Event Walked()
```

 Schritt 12: Um dem Event Parameter mitzugeben, ändern wir diese Zeile auf:

```
Public Event Walked(ByVal distance As Integer)
```

Nachdem ein Event definiert ist, kann es an einer oder mehreren Stellen ausgelöst werden.

 Schritt 13: Erweitern Sie die Walk Subroutine:

```
   Public Sub Walk(ByVal distance As Integer)
        mTotalDistance += distance
        RaiseEvent Walked(distance)
    End Sub
```

Der Befehl RaiseEvent löst ein Event aus.

6.7.3 Receiving Events mit WithEvents

Um auf ein Event zu reagieren, verwenden wir das Schlüsselwort WithEvents.

 Schritt 14: Erweitern Sie die Datei Module1:

```
Module Module1
    Dim WithEvents myPerson As Person

    Sub Main()
        myPerson = New Person
        myPerson.Walk(5)

        Console.ReadLine()
    End Sub
```

Wir deklarieren eine Variable mit WithEvents, die auf ein Objekt der Klasse Person zeigen kann. In der Sub Main erzeugen wir ein neues Objekt der Klasse Person. Da die Variable myPerson mit WithEvents definiert ist, reagiert Module1 auf alle Events des Objekts auf das myPerson zeigt.

Aber wir haben noch keinen Code, der auf ein Event reagiert. Wir wollen auf das Walked Event reagieren.

 Schritt 15: Erweitern Sie Module1:

```
Module Module1
    Dim WithEvents myPerson As Person

    Sub Main()
        myPerson = New Person
        myPerson.Walk(5)

        Console.ReadLine()
    End Sub

    Private Sub myPerson_Walked(ByVal distance As Integer) Handles myPerson.Walked
        Console.WriteLine("Person walked " & distance)
    End Sub
End Module
```

Der Name der Subroutine und der Name des Parameters sind frei wählbar. Wichtig ist nur der Typ des Parameters und die Handles Deklaration, die bestimmt, das myPerson_Walked auf das Walk Event von myPerson reagiert.

Schritt 16: Testen Sie das Programm mit F5
Am Bildschirm erscheint: „Person walked 5“

Der Event Handler läuft im selben Thread wie der Code, in dem das RaiseEvent stattfand. Ein Threadwechsel ist mit RaiseEvent nicht möglich.

6.7.4 Receiving Events mit AddHandler

Mit WithEvent und AddHandler kann man nur auf Objekte reagieren, deren Variable bereits existieren, wenn der Code geschrieben wird. Wenn wir jedoch dynamisch im Programmlauf Variable erzeugen, oder wenn wir das Verarbeiten von Events bei Bedarf ein- und ausschalten wollen, brauchen wir einen anderen Mechanismus. Dieser wird durch die Befehle AddHandler und RemoveHandler geboten.

Schritt 17: Ändern und erweitern Sie Module1:

- Entfernen Sie das Schlüsselwort WithEvents
- Fügen Sie die Zeile mit AddHandler ein
- Entfernen Sie `Handles myPerson.Walked`

```
Module Module1
    Dim myPerson As Person

    Sub Main()
        myPerson = New Person
        AddHandler myPerson.Walked, AddressOf myPerson_Walked
        myPerson.Walk(5)    'this method raises the Walked event of myPerson

        Console.ReadLine()
    End Sub

    Private Sub myPerson_Walked(ByVal distance As Integer)
        Console.WriteLine("Person walked " & distance)
    End Sub
End Module
```

Schritt 18: Testen Sie das Programm mit F5

Am Bildschirm erscheint: „Person walked 5“

Das Abschalten des Event Handlers erfolgt mit

```
RemoveHandler myPerson.Walked, AddressOf myPerson_Walked
```

Achtung: vermeiden Sie rekursive Events. Wenn ein Event Handler direkt oder indirekt das Event auslöst, auf das er reagiert, kommen Sie u.U. in des Teufels Küche.

Anmerkung: Mit Covariance und Contravariance wurde die Regel aufgeweicht, dass die Parameterlisten des Events und des Event Handlers übereinstimmen müssen. Bitte halten Sie sich trotzdem an die Regel.

Anmerkung: Für die in einem späteren Kapitel behandelte Serialisierung ist die Verwendung von AddHandler notwendig, da WithEvents nicht funktioniert.

6.8 Konstruktoren

Ein Konstruktor ist eine spezielle Methode einer Klasse, die ausgeführt wird, wenn ein Objekt instantiiert wird. Sie heißt in VB immer New. Ein Konstruktor wird daher typisch für das Initialisieren verwendet, da der Konstruktor immer als erste Methode ausgeführt wird. Wenn wir keinen Konstruktor programmieren, legt VB automatisch einen an, den wir nicht sehen.

Schritt 19: Fügen Sie den folgenden Code in die Personen Klasse ein:

```
Public Sub New()
    Phone("home") = "(212) 555 1234"
    Phone("mobile") = "(888) 555 4321"
End Sub
```

Konstruktoren können Parameter übernehmen.

Schritt 20: Erweitern Sie den Konstruktor der Person Klasse:

```
Public Sub New(ByVal name As String, ByVal birthdate As Date)
    mName = name
    mBirthdate = birthdate
```

Schritt 21: Jetzt müssen Sie die Zeile in Module1 ändern, in der das Person Objekt erzeugt wird:

```
myPerson = New Person("Monroe Marilyn", #6/1/1926#)'date literals: #mm/dd/yyyy#
```

Jeder Code, der ein Personen Objekt erzeugen will, muss diese Schreibweise verwenden. Alternativen sind optionale Parameter oder die Überladung des Konstruktors (das Erzeugen mehrere Versionen mit unterschiedlicher Parameterliste), was wir später sehen werden.

6.9 Termination und Cleanup, Destruktor

Ein Konstruktor erzeugt ein Objekt. Die Aufgabe, es wieder zu zerstören, übernimmt der Destruktor. Er heißt in .Net Finalizer und wird durch den sogenannten Garbage Collector aufgerufen.

Der Garbage Collector (GC) läuft automatisch im Hintergrund eines Programms (in einem eigenen Thread) und vernichtet Objekte, die nicht mehr in Verwendung stehen, um den Speicherplatz wieder frei zu bekommen. Memory Leaks, wie sie von älteren Programmiersprachen bekannt sind und zum Absturz des Computers führen können, sind also durch .NET Code nicht mehr möglich. Der Zeitpunkt,

wann der GC ein Objekt löscht, kann nicht vorhergesagt und nur bedingt beeinflußt werden werden (durch Aufruf von GC.Collect).

Ein Objekt steht nicht mehr in Verwendung, wenn es keine Variable mehr gibt, die auf das Objekt zeigt. D.h. wenn wir einer Variablen Nothing (C#: null) zuweisen, sie auf ein anderes Objekt zeigen lassen, oder die Variable aus dem Codebereich gerät, in dem sie deklariert wurde (Scope), löschen wir die Referenz auf das Objekt. Wenn es keine Referenzen auf ein Objekt mehr gibt, kann es durch den GC gelöscht werden.

Vermeiden Sie den Begriff Destruktor, da 1) unter Destruktor in C++, C#, VB und Java etwas anderes verstanden wird und 2) es in .NET zwei Methoden für das Aufräumen von Objekten gibt: Finalize und DIspose.

6.10 Finalize und Dispose

In einem Programm, das vollständig in .NET geschrieben ist und auch keinen alten Code verwendet, brauchen wir uns um das Zerstören eines Objekts also nicht zwingend zu kümmern. Der GC ruft in dem zu zerstörenden Objekt eine Art Destruktor auf. Dieser heißt in .Net Finalize.

Finalize darf nur vom GC aufgerufen werden. Was aber, wenn unser Code ein Objekt sauber „aufräumen“ muss, z.B. Weil es wertvolle Resourcen blockiert? Dann ist der Finalize der falsche Platz, dafür gibt es die Methode Dispose(). Dispose wird von User-Code, nicht vom GC aufgerufen. Das heisst der Entwickler ist für den richtigen Zeitpunkt des Aufrufs von Dispose bzw. die richtige Implementierung von Dispose verantwortlich.

Wenn Sie nicht die übliche 08/15 Bürosoftware programmieren, sondern echte Steuerungen bauen, wie z.B. Fabriks-Automation, Modelleisenbahn-Elektronik oder eine Drohne, müssen Sie sich intensiv mit Dispose auseinandersetzen.

Nehmen wir an, Sie fliegen mittels einem .NET Objekt namens myHeli Ihren ferngesteuerten Hubschrauber. Wenn Sie nun unabsichtlich Ihr Programm beenden, dann wird natürlich irgendwann das myHeli Objekt im Speicher gelöscht. Der dabei aufgerufenen Destruktor (VB: Finalizer) sendet ein Notaus an den Helikopter, falls Sie das eingebaut haben. Aber bis dahin fliegt ihr Hubschrauber unkontrolliert weiter. Und dann kann er von alleine landen?

Nicht gut, besonders wenn Zuschauer in der Nähe stehen. Sie müssen also die Heli Klasse so bauen, dass in jedem Fall vor Zerstören eines Objekts der Hubschrauber sanft landet. Das ist nicht trivial.

Wenn Sie ein sauberes Beenden der Verwendung eines Objekts brauchen (was nicht dasselbe ist wie seine Zerstörung), dann müssen Sie das Interface IDisposable implementieren. Dieses bringt die Methode Dispose().

Merken Sie sich folgende Regeln:

- Wenn Sie nur managed .NET Code verwenden, und alle Methoden, die Ihr Code aufruft, genau dasselbe tun, brauchen Sie sich um das Beenden eines Objekts nicht mehr zu kümmern. Das ist für die Praxis aber zuwenig.
- Leider dokumentiert kein Hersteller einer Bibliothek, ob er auch unmanaged Code verwendet. Das sind z.B. Aufrufe ins Betriebssystem Windows. Alle .NET Bibliotheken, die mit Dateien, Grafik oder Betriebssystem-Funktionalität zu tun haben, nutzen unmanaged Code. Diese bieten eine Methode Dispose() oder Close(), das Dispose() entspricht.

- Wenn Sie ein Objekt einer Klasse verwenden, die die Methode Dispose() bietet, dann müssen Sie Dispose() aufrufen, wenn Sie das Objekt nicht mehr benötigen.
- Die Verwendung von Dispose() können Sie durch das Using Konstrukt erzwingen. Die Verantwortung bleibt aber beim Entwickler: entweder der direkte Aufruf von Dispose() selbst oder der indirekte über Using muss programmiert werden.
- Wenn Sie eine Klasse bauen, deren Objekte nicht einfach nur weggeworfen werden dürfen, müssen Sie in dieser Klasse IDisposable implementieren.

 Schritt 22: Fügen Sie folgende Zeile in die Person Klasse ein und drücken Sie ENTER:

```
Public Class Person
    Implements IDisposable
```

Am Ende der Person Klasse wurde damit Code eingefügt, der die Methoden Dispose() und Finalize() enthält.

Wichtig: VB und C# unterscheiden sich hier: In C# wird nur das reine Interface IDispose implementiert, in VB ein ganzes Dispose/Finalize Pattern. C# Entwitckler sollen dieses Pattern selbst implementieren.

Wichtig: In .NET gibt es also zwei Methoden am Ende der Verwendung eines Objekts: Dispose() wird durch Benutzercode aufgerufen, um zu zeigen „ich brauche dich nicht mehr“ und Finalize() wird durch den GC aufgerufen, um das Objekt endgültig zu zerstören. Ein sehr guter Artikel dazu auf MSDN ist „Implementing a Dispose Method“.

Unterbrechen Sie momentan das Thema Destruktor und arbeiten Sie unbedingt zu einem späteren Zeitpunkt das entsprechende ausführliche Beispiel im Download zum Buch durch.

6.11 Fortgeschrittene Konzepte

6.11.1 Overloading Methods

Optionale Parameter sind schwerfällig in der Benutzung. Einerseits weil sie immer am Ende der Parameterliste stehen, weil ihr Datentyp nicht geändert werden kann und weil manche Programmiersprachen solche Parameterlisten nicht erzeugen können.

Das Überladen von Methoden (Method Overloading) löst diese Probleme. Wir definieren Methoden mit demselben Namen, aber Parameterlisten, in denen sich die Parameter im Datentyp oder in der Anzahl unterscheiden. Der Name der Parameter ist dabei bedeutungslos.

 Schritt 23: Fügen Sie folgenden Code in die Personen Klasse ein, um die Walk Methode zu überschreiben:

```
    Public Sub Walk()
        RaiseEvent Walked(0)
    End Sub
```

Das Schlüsselwort Overloads ist hier nicht notwendig – sondern erst, wenn ein Overload zusammen mit Vererbung passiert.

Jetzt können wir aus Module1 zwei verschiedene Aufrufe von Walk benutzen:

```
        myPerson.Walk(10)
        myPerson.Walk()
```

6.11.2 Shared Methods, Variables und Events

Bis jetzt haben wir Instanz Methoden, Instanz Variable und Instanz Events verwendet. Das heißt, um sie zu benutzen, muss zuerst ein Objekt einer Klasse instantiiert werden. Jedes Objekt hat dann einen eigenen Satz an Variablen, die von seinen Methoden und Events genutzt werden.

Mit dem Schlüsselwort Shared definieren wir Variable, Methoden und Events, die nicht einem Objekt, sondern der Klasse gehören. Shared Elemente sind für alle Objekte der Klasse gleich und können nur durch shared Methoden geändert werden.

6.11.3 Shared Variables

Shared Variables sind also nur einmal vorhanden, egal wieviele Objekte erzeugt werden und alle Objekte nutzen diese Variablen gemeinsam.

Wir wollen in unserem Beispiel jeder Person eine eindeutige ID geben und die Anzahl der erschaffenen Personen Objekte zählen.

Schritt 24: Fügen Sie die Zeile in die Person Klasse ein:

```
Public Class Person
    Implements IDisposable

    Private mName As String
    Private mBirthdate As Date
    Private mTotalDistance As Integer
    Private mPhones As New Hashtable
    Private Shared mCounter As Integer
```

Shared bedeutet, dass die Variable mCounter von allen Objekten der Klasse Person gemeinsam benutzt wird.

Schritt 25: Fügen Sie folgende Zeilen ein:

```
    Public Sub New()
        Phone("home") = "(212) 555 1234"
        Phone("mobile") = "(888) 555 4321"
        mCounter += 1
    End Sub

    Public Sub New(ByVal name As String, ByVal birthdate As Date)
        mName = name
        mBirthdate = birthdate
        Phone("home") = "(212) 555 1234"
        Phone("mobile") = "(888) 555 4321"
        mCounter += 1
    End Sub
```

Der mCounter enthält die Anzahl der erzeugten Person Objekte.

Schritt 26: Fügen Sie den Code für die Abfrage des Zählers hinzu:

```
  Public ReadOnly Property Count() As Integer
        Get
            Return mCounter
        End Get
  End Property
```

Da dieser Code nur auf Shared Variable zugreift, kann er auch als Shared Method deklariert werden:

```
Public Shared ReadOnly Property Count() As Integer
```

Non-shared Methoden können shared Variable lesen, und greifen zum Schreiben über den Namen der Klasse auf die shared Variable zu. Auf static Variable kann man durch static Methoden oder durch Verwendung des Klassennamens zugreifen. Für das Initialisieren von shared Variablen gibt es einen shared Konstruktor.

6.11.4 Shared Methods

Shared Methods werden von allen Objekten gemeinsam benutzt. Daher können sie nicht auf Instance Variables zugreifen. Da sie der Klasse gehören, können Sie über den Klassennamen aufgerufen werden – noch bevor ein Objekt der Klasse erzeugt wurde.

In unserem Fall erlaubt also die Definition

```
Public Shared ReadOnly Property Count() As Integer
```

dass wir die Anzahl der Objekte jederzeit mit der Zeile

```
Console.WriteLine("Number of persons: " & Person.Count)
```

abfragen können.

Als weiteres Besipiel wollen wir eine Methode codieren, die das Alter zweier Personen vergleicht.

Schritt 27: Fügen Sie folgenden Code in die Klasse Person ein:

```
Public Shared Function CompareAge(ByVal person1 As Person, _
ByVal person2 As Person) As Boolean
        Return person1.Age > person2.Age
End Function
```

Diese Methode kann über die Klasse aufgerufen werden:

```
If Person.CompareAge(myPerson1, myPerson2) Then …
```

6.11.5 Shared Events

Normale Events können von Shared Methods nicht ausgelöst werden, Shared Events schon.

Wenn Obj1, Obj2, Obj3 Objekte derselben Klasse sind und eines davon zB Obj1 das Shared Event X auslöst, dann erhalten alle Consumer, die für die Events Obj1.X, Obj2.X, Obj3.X einen EventHandler definiert haben, dieses Event mit Obj1 als *sender* Parameter. Bei einem Shared Event genügt also *ein* Objekt der Klasse, um von *allen* Objekten das Shared Event zu bekommen, egal welches Objekt das Event auslöst. Daher macht ein zentraler Event Handler Sinn.

Man kann auch mit der Klasse alleine einen Event Handler für das Shared Event definieren:

```
AddHandler SharedEvents.Alarm, AddressOf OnAlarmSharedEvent
```

Damit lässt sich mit AddHandler ein EventHandler definieren, der auf ein Shared Event einer Klasse reagiert, egal von welchem Objekt dieser Klasse er ausgelöst wird!

Wichtig: Shared Methoden und Shared Events können keine Interface Members implementieren!

6.12 Delegates

Anmerkung: Delegates sind eines der schwierigsten Themen für Anfänger, aber a) .NET macht wie jede professionelle Software heftig davon Gebrauch und b) ernsthaftes Programmieren erfordert daher den fließenden Umgang damit. Investieren Sie daher Zeit, um Delegates und deren verschiedene Einsatzfälle zu verstehen.

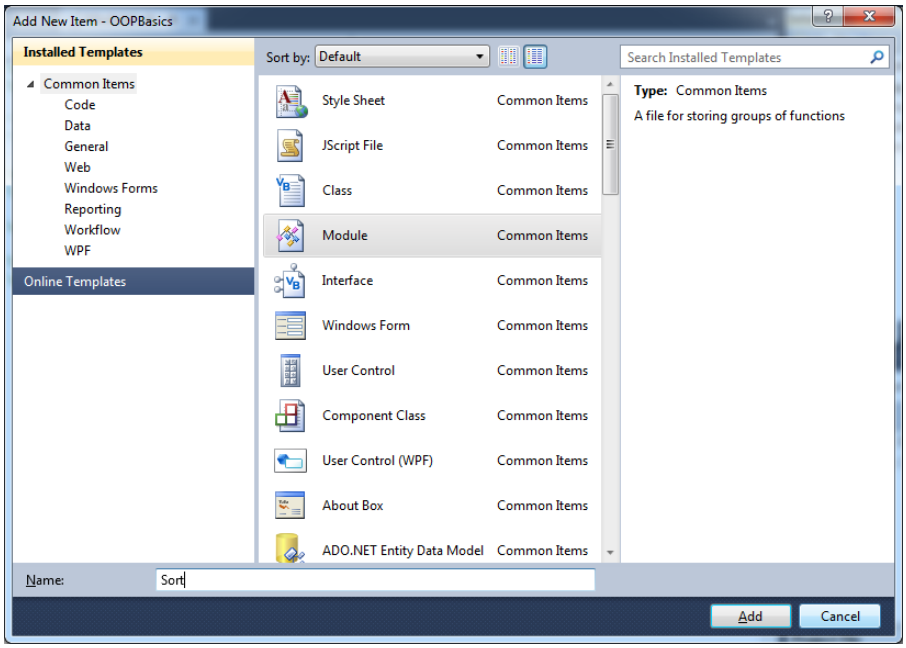
Wenn wir ein allgemein gültiges Sortierverfahren bauen, und diesem einmal Äpfel und ein anderes Mal Birnen übergeben, wird das nicht funktionieren. Woher soll die allgemein gültige Sortier Methode wissen, wie man Äpfel vergleicht, wie man Birnen vergleicht? Das heißt, wir benötigen neben der Möglichkeit Daten als Parameter zu übergeben, auch die Möglichkeit Methoden als Parameter zu übergeben: „Hier hast du Äpfel zu sortieren und das ist das Sortierkriterium dazu“.

6.12.1 Deklaration eines Delegates

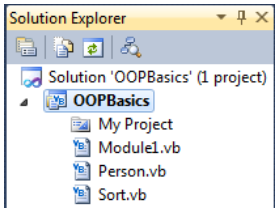
Lassen Sie uns anhand eines Beispiels lernen.

- Wir erzeugen einen Delegate der auf eine Methode zeigt, die zwei Objekte vergleicht und einen Boolschen Wert zurückliefert. True bedeutet, das erste Objekt ist größer als das zweite. Ansonsten wird False zurückgeliefert.
- Dann erzeugen wir eine Methode mit einem Sortieralgorithmus, die den Delegate für das Sortieren verwendet.
- Zuletzt bauen wir die eigentliche Methode, die den Vergleich implementiert, und übergeben ihre Adresse an die Sortier-Methode.

Schritt 28: Klicken Sie mit der rechten Maustaste im Solution Explorer auf das Projekt und wählen Sie den Menüpunkt ADD > NEW ITEM... Im Add New Item Dialog wählen Sie Module und geben ihm den Namen Sort. Klicken Sie auf OK.



Der Solution Explorer zeigt nun:



Methoden und Variable in einem Module, das kein Sub Main enthält, sind automatisch Shared.

Schritt 29: Fügen Sie folgenden Code ins Module Sort ein:

```
Public Delegate Function Compare(ByVal v1 As Object, ByVal v2 As Object) _
As Boolean
```

Der Delegate Compare definiert eine Methode, die zwei Objekte als Parameter erhält und einen Boolschen Wert zurückliefert.

Die Definition des Delegates legt die Reihenfolge der Datentypen und den Datentyp des Rückgabewerts als neuen Datentyp mit dem Namen Compare fest. d.h. eine Variable vom Typ Compare muss die Adresse einer Methode mit einer Parameterliste (object, object) und einem Rückgabewert Boolean beinhalten. Über die Variable kann die Methode aufgerufen werden.

6.12.2 Benutzung eines Delegates

Nun können wir eine Sortier-Routine schreiben, die einen Parameter vom Datentyp des Delegates besitzt. Damit kann jeder diese Sortier-Routine aufrufen, der die Adresse einer Methode mitliefert, die eine passende Parameterliste und einen passenden Rückgabewert hat. Die Adresse wird als Delegate vom Typ Compare als zweiter Parameter übergeben.

Um eine Methode auszuführen, auf die ein Delegate zeigt, ist die Invoke() Methode des Delegates auszuführen.

Schritt 30: Fügen Sie folgenden Code in Sort.vb ein:

```
Public Sub DoSort(ByVal dataToSort() As Object, ByVal GreaterThan As Compare)
    Dim outer As Integer    'index outer element
    Dim inner As Integer    'index inner element
    Dim temp As Object      'helper element

    For outer = 0 To UBound(dataToSort)
        For inner = outer + 1 To UBound(dataToSort)
            'delegate must be called using Invoke(),
            '(invoke does not need to be written)
            If GreaterThan.Invoke(dataToSort(outer), dataToSort(inner)) Then
                'swap inner and outer elements
                temp = dataToSort(outer)
                dataToSort(outer) = dataToSort(inner)
                dataToSort(inner) = temp
            End If
        Next
    Next
End Sub
```

Durch die Verwendung von Object als Datentyp der zu sortierenden Daten ist die DoSort Routine vollständig unabhängig vom wirklichen Datentyp der zu sortierenden Daten.

Frage: Warum muss man die zu sortierenden Daten im ersten Parameter nicht ByRef übergeben?

6.12.3 Implementierung einer Delegate Methode

Jetzt müssen wir noch eine Methode zum Vergleich zweier Objekte bauen, die eine passende Parameterliste und einen passenden Rückgabewert aufweist, die der Delegate Definition entspricht.

Unsere Person Klasse hat bereits eine Methode CompareAge, deren Parameter leider nicht den passenden Typ haben. Wir lösen das Problem mittels Overloading, indem wir eine zusätzliche CompareAge mit passender Parameterliste schreiben.

Schritt 31: Fügen Sie folgenden Code in die Person Klasse ein:

```
Public Shared Function CompareAge(ByVal person1 As Object, _
ByVal person2 As Object) As Boolean
        Return CType(person1, Person).Age > CType(person2, Person).Age
End Function
```

Da die Methode zwei Parameter vom Type Object erhält, müssen diese erst auf den Typ Person gewandelt werden, damit wir die Property Age aufrufen können. Die Umwandlung erfolgt mittels TypeCast CType().

Wenn wir die DoSort Methode aufrufen, müssen wir als zweiten Parameter einen Delegate vom Typ Compare übergeben, der die Adresse einer anderen Methode beinhaltet, die die von Compare verlangte Parameterliste und den Rückgabewert besitzt. Die Adresse einer Methode ermitteln wir mit dem AddressOf Operator.

Jetzt brauchen wir die Sort Methode nur mehr zu testen:

Schritt 32: Fügen Sie folgenden Code in die Datei Module1 ein:

```
'create person array and initialize it
Dim myPeople(4) As Person       'create array holding 5 persons
myPeople(0) = New Person("Herbert", #7/9/1965#)
myPeople(1) = New Person("Kurt", #1/21/1955#)
myPeople(2) = New Person("Eva", #2/1/1960#)
myPeople(3) = New Person("Paul", #5/13/1970#)
myPeople(4) = New Person("Otto", #10/1/1975#)

'sort the person array by age
DoSort(myPeople, AddressOf Person.CompareAge)

'output data to console window: name and age
For Each p As Person In myPeople
      Console.WriteLine(p.Name & " " & p.Age)
Next
```

Schritt 33: Starten Sie das Programm. Der Bildschirm zeigt
Otto 35
Paul 40
Herbert 45
Eva 50
Kurt 55

Wenn wir nun die Personen nach einem anderen Kriterium sortieren wollen, benötigen wir nur eine andere Compare Methode, die dieselbe Parameterliste und denselben Rückgabewert aufweist.

Schritt 34: Fügen Sie folgenden Code in die Person Klasse ein:

```
Public Shared Function CompareName(ByVal person1 As Object, _
ByVal person2 As Object) As Boolean
        Return CType(person1, Person).Name > CType(person2, Person).Name
End Function
```

Schritt 35: Ändern Sie folgende Zeile in der Datei Module1:

```
'sort the person array by age
   DoSort(myPeople, AddressOf Person.CompareName)
'output data to console window: name and age
```

Der Bildschirm zeigt nun:

Eva 50
Herbert 45
Kurt 55
Otto 35
Paul 40

Hinweis: In Zusammenhang mit Delegates nennt man die Reihenfolge der Datentypen der Parameter *und* den Datentypen des Rückgabewerts Signatur. In Zusammenhang mit Overload spricht man auch von Signatur, meint aber nur die Reihenfolge der Datentypen der Parameter *ohne* den Datentyp des Rückgabewerts. Da dies Anfänger verwirrt, wurde der Begriff oben nicht verwendet.

Hinweis: Wir haben bisher stets lokale Variable zusammen mit deren globalen Properties definiert. VS erlaubt dafür eine verkürzte Schreibweise, bei der der Get und Set Teil automatisch erzeugt wird und die dahinter liegende lokale Variable, deren Namen wir nicht kennen:

```
Public Property Name() As String
```

Daher können wir innerhalb der Klasse, die diese Definition erhält nur mit Me.Name über die Property auf die Variable zugreifen. Me ist eine Referenz auf die Instanz des Objekts innerhalb der dieser Code ausgeführt wird.

Bei dieser Schreibweise kann die lokale Variable auch gleich initialisiert werden:

```
Public Property FirstName() As String = "Herbert"
```

6.12.4 Definition Delegates

Ein Delegate ist ein Datentyp, der eine Methoden Signatur definiert (die Reihenfolge der Datentypen der Parameter *und* den Datentypen des Rückgabewerts). Wenn eine Variable vom Typ des Delegates angelegt wird, kann dieser eine beliebige Methode mit kompatibler Signatur zugewiesen werden. Diese wird über die Invoke Methode ausgeführt.

Mit Delegates kann man eine Methode als Parameter übergeben. Die Zuordnung kann dynamisch erfolgen. Ein Delegate ist also ein „typsicherer Pointer“. Mit einem Delegate kann man eine Methode indirekt aufrufen.

Die Arbeit mit einem Delegate erfolgt in vier Schritten:

1. Man definiert einen Delegate mit einer bestimmten Signatur und gibt ihm einen Namen. Dieser Delegate ist ein neuer Datentyp. Oder man nimmt einen der vielen in .NET vordefinierten Delegate.
2. Man legt eine Variable vom Typ dieses Delegates an.
3. Man weist der Variablen die Adresse einer Methode zu, deren Signatur mit dem Delegate übereinstimmt. Diese Variable enthält also eine Adresse.

4. Durch Aufruf der Invoke() Methode der Variable wird die Methode ausgeführt, auf die die Variable zeigt.

```
Module Module1
    'define a delegate named MyDelegate with a signature
    '     "Params: int; Returns boolean"
    'each instance of this delegate can hold a reference to a method that
    'takes one int parameter and returns a bool.
    Public Delegate Function MyDelegate(ByVal i As Integer) As Boolean

    Sub Main()
        'create a variable of type MyDelegate
        Dim myDelegateVariable As MyDelegate

        'this can hold the address of a function with matching signature
        myDelegateVariable = AddressOf DoSomething

        'execute the code pointed to by the variable using Invoke()
        Dim input As Integer = 4
        Dim b As Boolean = myDelegateVariable.Invoke(input)
        Console.WriteLine("Result: " & b.ToString)

        Console.ReadLine()
    End Sub

    Public Function DoSomething(i As Integer) As Boolean
        If i < 0 Then Return False Else Return True
    End Function
End Module
```

Delegates spielen eine große Rolle in .NET: Events, asynchrone Operationen, Multithreading u.v.a.m. Zu Delegates gibt es viele „Unterklassen“, entsprechend den unterschiedlichen Anforderungen wie z.B. MulticastDelegate, AsyncCallback, EventHandler, CrossAppDomainDelegate.

Mit Delegates können Code-Teile und Assemblies entkoppelt werden. Damit vermeidet man zirkulare Referenzen.

Hinweis (ohne Beziehung zu Delegates): Es gibt viele weitere Möglichkeiten, wie Sie Schreibaufwand sparen. Schlagen Sie in der Online Hilfe folgende Themen nach: Array Initializer, Collection Initializer.

Übung 15 Firmen-Objektmodell

Erstellen Sie ein neues Console Application Projekt namens Company, das ein Unternehmen abbildet:

- Fügen Sie folgenden Klassen ein: Company, Department, Employee, Customer.
- Überlegen und codieren Sie für jede Klasse sinnvolle Variable, Konstruktoren, Methoden, Properties und Events.

Hinweis: dieses Beispiel wird in den folgenden Kapiteln weiter ausgebaut!

Übung 16 Behälter-Überwachung

Teil 1: Erstellen Sie eine Klasse Staudamm (engl. dam):

- Ein Staudamm kennt

- seinen aktuellen, den maximal und minimal erlaubten Wasserstand in m
- Den maximalen Wert des Abflusses in m^3/min
- Den minimalen und maximalen Gesamtwert der Zuflüsse in m^3/min

○ Die Regelung des Staudamms bietet folgende „Befehle“:
- Einstellen des Zuflusses auf einen bestimmten Wert in m^3/min
- Einstellen des Abflusses auf einen bestimmten Wert in m^3/min
- Not-Ablassen des gesamtem Stausees

○ Die Regelung liefert folgende Zustandsanzeigen:
- Momentaner Wasserstand
- Der Speicher wird gefüllt bzw. geleert
- Wie lange dauert es beim momentanen Zufluss und Abfluss, bis der Stausee leer ist?
- Wie lange dauert es beim momentanen Zufluss und Abfluss, bis der Stausee voll ist?

○ Die Regelung liefert einen Alarm (Event) in folgenden Situationen:
- Der erlaubte maximale Wasserstand wird überschritten
- Der erlaubte minimale Wasserstand wird unterschritten
- Die Menge an zufließendem Wasser liegt über dem Maximalwert oder unter dem Minimalwert

Teil 2: Bauen Sie zu dieser Klasse ein Console Application Project als Testprogramm, mit dem sämtliche Funktionen getestet werden können.

6.13 Fragen

1. Wie erklären Sie die Begriffe Klasse und Objekt?
2. Wie unterscheiden sich in Subroutine und Function?
3. Was ist eine Property? Wie ist sie aufgebaut? Wozu dient sie?
4. Was ist eine parametrisierte Property?
5. Was ist eine Default Property?
6. Was ist ein Event? Wofür wird es verwendet? Wie funktioniert es?
7. Was ist ein Delegate? Wozu dient er? Wie wird er definiert und wie funktioniert er?
8. Wo verwendet .NET Delegates?
9. Wozu verwendet man Delegates?

6.14 Kompetenz-Check

	Ziel erreicht?	ja	nein
	Ich kenne den Unterschied zwischen Klasse und Struktur.	☐	☐
	Ich kann den Event Mechanismus erklären.	☐	☐
	Ich kann eine Klasse mit Methoden, Properties und Events bauen.	☐	☐
	Ich kann die Bedeutung von FInalize und Dispose erklären.	☐	☐
	Ich kann den Delegate Mechanismus erklären und ein Beispiel dazu programmieren.	☐	☐

6.15 Wo stehen wir?

In diesem Kapitel haben Sie die grundlegenden Bausteine des Objektorientierten Programmierens kennen gelernt. Das nächste Kapitel beschäftigt sich mit Vererbung (Inheritance) und multiplen Interfaces.