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.

Eine C# Klasse besteht nach Microsoft Terminologie aus

- Data Members und
- Function Members

die wir in diesem Kapitel kennenlernen. Eine Klasse ist wie ein Bauplan, nach ihrer Vorlage können Objekte erzeugt werden. Mit Objekten kann man in einem Programm „arbeiten".

Zu den Data Members gehören:

- Fields (Variable)
- Constants und
- Events.

Zu den Function Members gehören

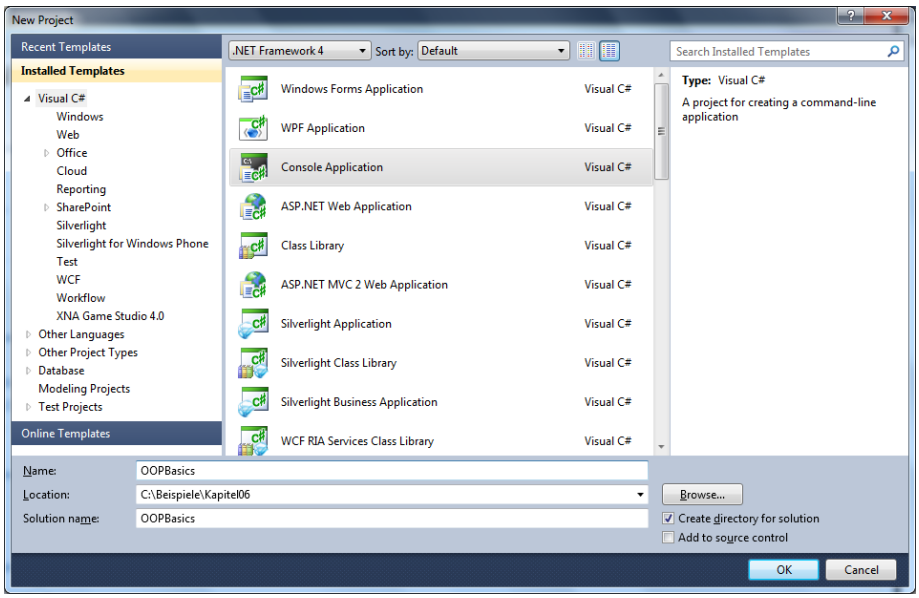
- Methods
- Properties
- Constructors
- Finalizers
- Operators und
- Indexers.

Im Mittelpunkt unseres Beispiels steht eine Klasse namens Person, wie sie in vielen Büchern zu finden ist. Der Code hier ist am Stand von VS.2010 und .NET 4.0.

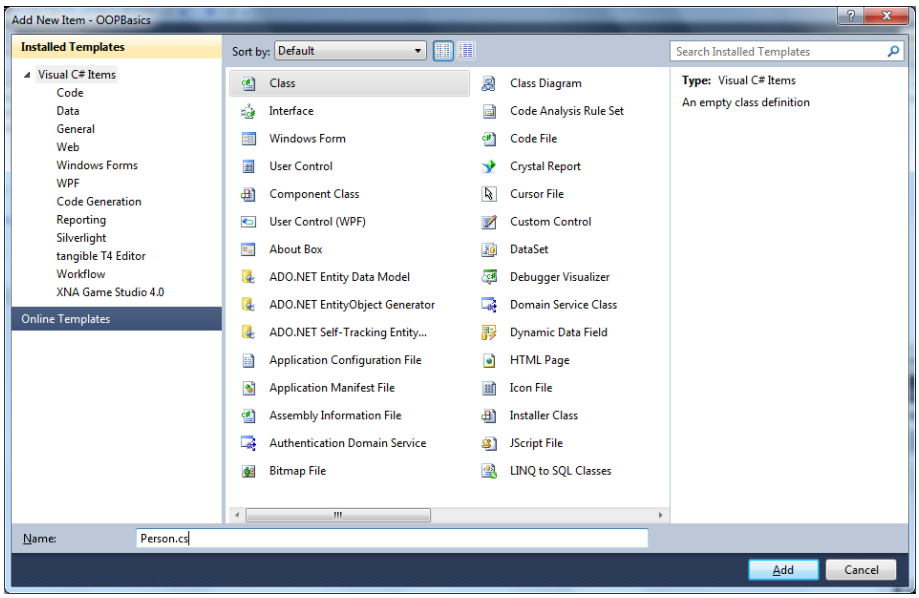
Wenn auf den nächsten Seiten Codezeilen gelb hinterlegt sind, so sind diese im Code einzugeben oder zu ändern.

6.1 Beispiel Personen Klasse

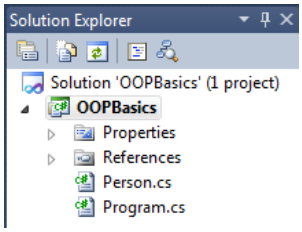
Schritt 1: Starten Sie VS und erzeugen Sie ein neues Console Project namens OOPBasics im Verzeichnis \Beispiele\Kapitel06.



Schritt 2: 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 3: Fügen Sie folgende Zeile in die Klasse Person ein:

```
namespace OOPBasics
{
    class Person
    {
        private string mName;
        private DateTime mBirthDate;
    }
}
```

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. Schimpfen Sie bitte nicht über meine: Ich versuche nur VB und C# parallel in einer Klasse zu unterrichten.

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 ohne Rückgabewert

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 4: Fügen Sie den folgenden Code in der Klasse Person hinzu:

```
namespace OOPBasics
{
```

```
    class Person
    {
        private string mName;
        private DateTime mBirthDate;

        public void Walk()
        {

        }
    }
}
```

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

```
        Person myPerson = new Person();
        myPerson.Walk();
```

6.4 Methoden mit Rückgabewert

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

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

```
        public int Age()
        {
            int age = DateTime.Now.Year - mBirthDate.Year;
            //age can be 1 to high if this year's birthdate is in the future
            if (mBirthDate.AddYears(age) > DateTime.Now)
                age -= 1; //correct age
            return age;
        }
```

Sie können diese Funktion beispielsweise so benutzen:

```
            Person myPerson = new Person();
            int age;
            age = myPerson.Age();
```

6.5 Methoden mit Parametern

Manchmal muss man einer Methode beim Aufruf Daten mitgeben.

Schritt 6: Fügen Sie die Definition von mTotalDistance in die Person Klasse ein und erweitern Sie die Walk() Methode:

```
    class Person
    {
        private string mName;
        private DateTime mBirthDate;
        private int mTotalDistance;

        public void Walk(int distance)
        {
            mTotalDistance += distance;
        }
```

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:

```
Person myPerson = new Person();
myPerson.Walk(2);
```

In C# werden alle Parameter by value übergeben, solange nicht explizit by reference verlangt wird. Damit wird hier eine Kopie des Wertes 2 an die Subroutine Walk übergeben. Und 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 ref.

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

6.6 Properties

Eine Property ist eine spezielle Art von Methode, die aus einer Art Function zum Lesen einer Variablen (Getter) 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 7: Fügen Sie den Code ein:

```
public string Name
{
    get
    {
        return mName;
    }
    set
    {
        mName = value;
    }
}
```

Die Property wird dann so wie unten verwendet, wobei kein Unterschied zur Lösung mit einer Methode oder einer globalen Variablen zu sehen ist:

```
Person myPerson = 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
{
    if (string.IsNullOrWhiteSpace(value))
        throw new Exception("Invalid name.");
    mName = value;
}
```

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

```
get
{
    if (string.IsNullOrWhiteSpace(mName)) return "not defined";
    return mName;
}
```

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.

Die Property aus Schritt 7 kann auch in weniger Zeilen geschrieben werden:

```
public string Name
{
    get {return mName;}
    set {mName = value;}
}
```

Wichtig: Beim Zugriff auf Methoden muss eine Parameterliste angegeben werden, auch wenn sie leer ist. Beim Zugriff auf Properties gibt es keine Parameterliste!

```
int age = myPerson.Age();               //method access
DateTime born = myPerson.BirthDate;     //property access
```

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.

```
public string Name {get; set;}
```

6.7 Indexer

VB kennt das Konzept der Parameterized Properties und der Default Property einer Klasse. In C# kann man dies zum Teil mithilfe eines Indexers nachbauen. Ein Indexer erlaubt es, eine Klasse, die eine Aufzählung enthält, wie ein Array anzusprechen. D.h. einzelne Daten in der Klasse werden über einen Index adressiert, ähnlich wie beim Zugriff auf ein Array.

Für Indexer gilt:

- Indexer benutzen get und set Zugriffsmethoden.
- Ein Indexer wird mit dem this Schlüsselwort definiert.
- Das Schlüsselwort value wird für die Zuweisung des Werts im set Teil verwendet.
- Indexer müssen nicht zwingend mit einem int Wert indiziert werden.
- Indexer können überladen werden.
- Indexer können mehr als einen Parameter besitzen.

Das Beispiel Indexer zeigt die Verwendung eines Indexers zum Speichern mehrerer Telefonnummern:

```
class Person
{
    public string Name;
    public string FirstName;
    private string[] phone = new string[3];

   public string this[int i]
    {
       get
        {
            return phone[i];
        }
       set
        {
            phone[i] = value;
        }
    }
}
```

Das Testprogramm dazu sieht so aus:

```
static void Main(string[] args)
{
    Person p = new Person();
    p.Name = "Mustermann";
    p.FirstName = "Max";
    p[0] = "(212) 555 1234";
    p[1] = "(800) 555 2345";
    p[2] = "(900) 555 3456";

    Console.WriteLine("{0}: Tel1: {1}; Tel2: {2}; Tel3: {3}",
                p.Name, p[0], p[1], p[2]);
    Console.ReadLine();
}
```

Das Beispiel IndexerEnum im Download löst dieselbe Aufgabe eleganter durch Verwendung einer enum für die verschiedenen Telefonnummern.

Hinweis: US Telefonnummern, die mit 555 beginnen, wie z.B. (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.

6.8 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 muss die Delegates zu den Events selbst codieren.

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.

Um eine Klasse mit einem Event auszustatten, sind folgende Schritte in C# notwendig:

1. Deklarieren Sie einen Delegate
2. Definieren Sie ein Event durch Instantiierung des Delegates
3. Lösen Sie das Event in einer Methode oder Property aus.

Um in einer Klasse auf ein Event eines Objekts zu reagieren, sind folgende Schritte in C# notwendig:

1. Deklarieren Sie eine Objektvariable, die auf das Objekt zeigt, das einen Event liefert.
2. Erstellen Sie einen Event Handler zur Behandlung des Events.
3. Melden Sie den Event Handler bei der Objektvariablen an.

Am besten sehen Sie das anhand unserer Beispiele.

6.8.1 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 8: Fügen Sie den Code in die Person Klasse ein:

```
class Person
{
    private string mName;
    private DateTime mBirthDate;
```

```
private int mTotalDistance;

public delegate void walkDelegate();
public event walkDelegate walked;
```

Schritt 9: Um dem Event Parameter mitzugeben, ändern wir diese Zeile auf:

```
public delegate void walkDelegate(int d);
```

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

Schritt 10: Erweitern Sie die Walk Methode:

```
public void Walk(int distance)
{
    mTotalDistance += distance;
    walked(distance);   //raise the walked event
}
```

6.8.2 Receiving Events

Wenn ein Objekt ein Event auslöst, weiß es nicht,

- ob überhaupt jemand auf das Event reagiert, weil sich niemand für den Empfang „registriert“ hat und
- in welcher Reihenfolge eventuelle Konsumenten des Events benachrichtigt werden.

Schritt 11: Ändern und erweitern Sie Main in Program.cs:

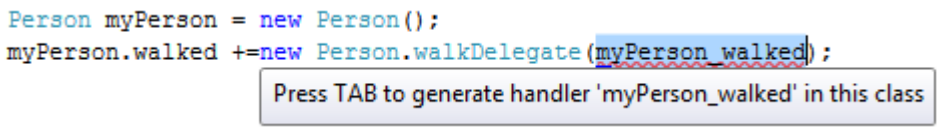
```
static void Main(string[] args)
{
    //create a person object
    Person myPerson = new Person();

    //wire up the event handler
    myPerson.walked += new Person.walkDelegate(myPerson_walked);

    myPerson.Walk(2);                    //let the person walk
    Console.ReadLine();
```

Wenn Sie `myPerson.walked +=` geschrieben haben, erscheint bereits der Intellisense Vorschlag für die Anbindung eines Event Handlers mit dem Namen <objectname>_<eventname>:

Drücken Sie TAB, um den Vorschlag anzunehmen und ein zweites Mal TAB, um sich das Skelett des Event Handlers generieren zu lassen:



Schritt 12: Ändern Sie den Event Handler in Program.cs, sodaß er am Bildschirm die gegangene Entfernung ausgibt:

```
static void myPerson_walked(int d)
{
    Console.WriteLine("Walked: {0}", d);
}
```

Da sich der Event Handler in der Klasse Program befindet, muss er static definiert sein.

Schritt 13: Starten Sie das Programm mit F5, am Bildschirm erscheint der Text des Event Handlers:

Walked: 2

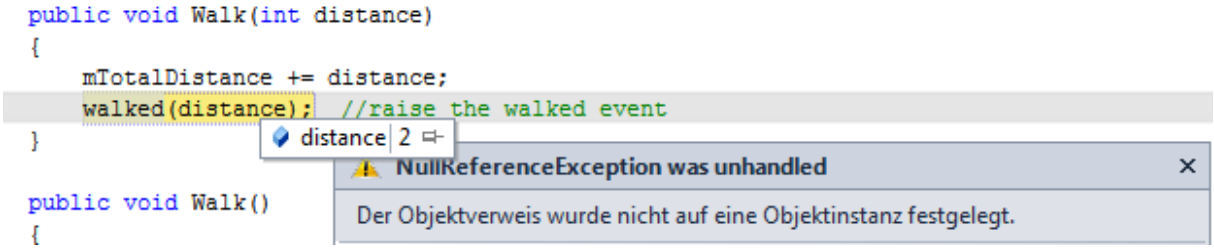
Schritt 14: Was passiert, wenn wir die Verknüpfung Event mit Event Handler erst nach dem Aufruf von Walk(2) im Main Programm durchführen? Verschieben Sie die entsprechende Zeile:

```
static void Main(string[] args)
{
    //create a person object
    Person myPerson = new Person();

    myPerson.Walk(2);                    //let the person walk

    //wire up the event handler
    myPerson.walked += new Person.walkDelegate(myPerson_walked);
```

Schritt 15: Starten Sie das Programm mit F5 – und es stürzt ab:



NullReferenceException bedeutet, dass walked nicht definiert ist. Oder besser gesagt: Es wurde kein Event Handler mit dem Event verknüpft.

Das können Sie als Entwickler der Person Klasse aber nicht beeinflussen. Also müssen Sie sich absichern.

Schritt 16: Sichern Sie durch folgenden Code in der Person Klasse das Feuern des Events ab, um die NullReferenceException zu vermeiden:

```
public void Walk(int distance)
{
    mTotalDistance += distance;
    if (walked != null)
        walked(distance);  //raise the walked event
}
```

Schritt 17: Testen Sie das Programm mit F5 – es wir nun nicht mehr abstürzen, aber natürlich auch keine Meldung über die Entfernung ausgeben.

Offensichtlich ist es mühsam, bei jedem Feuern eines Events zu prüfen, ob überhaupt ein Event Handler existiert.

Schritt 18: Besser ist es, eine Methode pro Event zu schreiben, die dann überall aufgerufen wird, wo das Event auszulösen ist. Der Name einer solchen Methode lautet standardmäßig On<Eventname>. Fügen Sie folgende Methode in die Person Klasse ein:

```
protected void OnWalked(int distance)
{
    if (walked != null)
        walked(distance);  //raise the walked event
}
```

Das Anlegen einer On<Event> Methode zum Auslösen eines Events bringt uns später einen weiteren Vorteil. Eine solche Methode in einer Basisklasse ist die einzige Möglichkeit, aus einer Subklasse ein Event der Basisklasse auszulösen.

Schritt 19: Ändern Sie die Walk(distance) Methode:

```
public void Walk(int distance)
{
    mTotalDistance += distance;
    OnWalked(distance);
}
```

Schritt 20: Verschieben Sie in Main() wieder die Verknüpfung des Event Handlers vor den Aufruf der Walk() Methode:

```
static void Main(string[] args)
{
    //create a person object
    Person myPerson = new Person();

    //wire up the event handler
    myPerson.walked += new Person.walkDelegate(myPerson_walked);

    myPerson.Walk(2);                    //let the person walk
}
```

Schritt 21: Starten Sie das Programm. Am Bildschirm erscheint wieder die Ausgabe des Event Handlers.

Das Abschalten des Event Handlers erfolgt mit

```
myPerson.walked -= new Person.walkDelegate(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.

6.9 Konstruktoren

Ein Konstruktor erzeugt ein Objekt. Er ist eine spezielle Methode einer Klasse, die ausgeführt wird, wenn ein Objekt instantiiert wird. Der Konstruktor heißt in C# immer wie die Klasse.

Ein Konstruktor wird daher typisch für das Initialisieren eines Objekts verwendet, da er immer als erste Methode des Objekts ausgeführt wird. Wenn wir keinen Konstruktor programmieren, legt C# automatisch einen an, den wir nicht sehen. Konstruktoren werden in anderen Sprachen oft auch ctor genannt.

Konstruktoren können Parameter übernehmen.

Schritt 22: Fügen Sie den Konstruktor in die Person Klasse ein:

```
public Person(string name, string birthDate)
{
    mName = name;
    mBirthDate = DateTime.Parse(birthDate);
}
```

Schritt 23: Jetzt müssen wir die Zeile in Program.cs ändern, in der das Person Objekt erzeugt wird:

```
Person myPerson = new Person("Feichtinger","18.03.1980");
```

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.

6.10 Termination und Cleanup, Destruktor

Ein Konstruktor erzeugt ein Objekt. Die Aufgabe, es wieder zu zerstören, teilen sich der Destruktor und der Garbage Collector.

Der Garbage Collector (GC) läuft im Hintergrund eines Programms 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, können so durch .NET Code nicht mehr entstehen. 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 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.

6.11 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 den Destruktor auf. Dieser heißt wie die Klasse mit dem Präfix ~ davor.

Der Destruktor darf nur vom GC aufgerufen werden. Was aber, wenn wir ein Objekt beim Beenden sauber „aufräumen“ müssen? Dann ist der Destruktor der falsche Platz, dafür gibt es ein Dispose().

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 den 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().

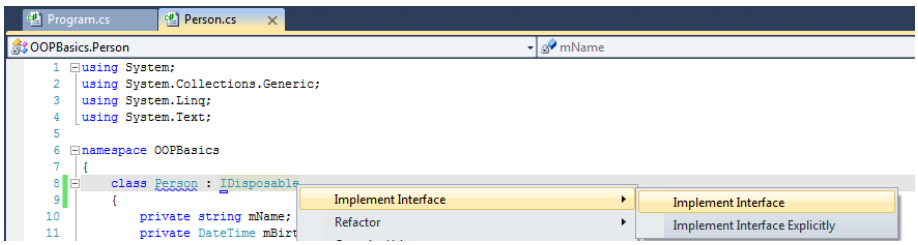
Merken Sie sich folgende Regeln:

- 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 eine Klasse bauen, deren Objekte nicht einfach nur weggeworfen werden dürfen, müssen Sie in dieser Klasse IDisposable implementieren.
- 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.

Schritt 24: Erweitern Sie die Klasse Person um das Interface IDisposable:

```
class Person : IDisposable
```

Klicken Sie mit der rechten Maustaste auf IDisposable und wählen Sie Implement Interface:



Dies fügt den Code für die Dispose Methode am Ende der Datei ein:

```
public void Dispose()
{
    throw new NotImplementedException();
}
```

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.

6.12 Fortgeschrittene Konzepte

6.12.1 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 25: Fügen Sie folgenden Code in die Personen Klasse ein, um die Walk() Methode zu überschreiben:

```
public void Walk()
{
      walked(0);  //raise the walked event
}
```

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

Jetzt können wir aus Program.cs zwei verschiedene Aufrufe von Walk() benutzen:

```
myPerson.Walk(2);
myPerson.Walk();
```

6.12.2 Static 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 static definieren wir Variable, Methoden und Events, die nicht einem Objekt, sondern der Klasse gehören. Statische Elemente sind für alle Objekte der Klasse gleich und können nur durch static Methoden geändert werden.

6.12.3 Static Variables

Eine static Variable ist also nur einmal vorhanden, egal wieviel Objekte von einer Klasse erzeugt werden, und alle Objekte dieser Klasse nutzen diese Variablen gemeinsam.

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

Schritt 26: Fügen Sie die Zeile in die Person Klasse ein:

```
class Person : IDisposable
{
    private string mName;
    private DateTime mBirthDate;
    private int mTotalDistance;
    private static int mCounter;
```

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

Schritt 27: Fügen Sie folgende Zeile ein:

```
public Person(string name, string birthDate)
{
    mName = name;
    mBirthDate = DateTime.Parse(birthDate);
    mCounter += 1;
}
```

Der mCounter enthält die Anzahl der erzeugten Person Objekte.

Schritt 28: Fügen Sie den Code für die Abfrage des Zählers hinzu:

```
public int Counter
{
    get
    {
        return mCounter;
    }
}
```

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

```
public static int Counter
```

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

6.12.4 Static Methods

Static 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 static int Counter
```

dass wir die Anzahl der Objekte jederzeit mit der Zeile

```
Console.WriteLine("Number of persons: {0}.", Person.Counter);
```

abfragen können.

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

Schritt 29: Fügen Sie folgenden Code in die Klasse Person ein:

```
public static bool CompareAge(Person person1, Person person2)
{
    return (person1.Age() > person2.Age());
}
```

Diese static Methode kann über die Klasse aufgerufen werden:

```
if (Person.CompareAge(myPeople[0], myPeople[1]))
    Console.WriteLine("{0} is older than {1}",
                myPeople[0].Name, myPeople[1].Name);
```

6.12.5 Static Events

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

Wenn Obj1, Obj2, Obj3 Objekte derselben Klasse sind und eines davon zB Obj1 das static 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 static Event genügt also *ein* Objekt der Klasse, um von *allen* Objekten das static 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 static Event definieren. Damit lässt sich mit AddHandler ein EventHandler definieren, der auf ein static Event einer Klasse reagiert, egal von welchem Objekt dieser Klasse er ausgelöst wird!

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

6.13 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“.

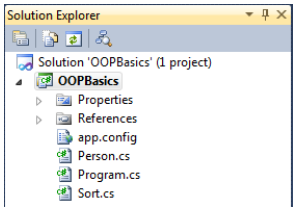
6.13.1 Deklaration eines Delegates

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 30: 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 Class und geben ihr den Namen Sort. Klicken Sie auf OK.

Der Solution Explorer zeigt nun:



Schritt 31: Ändern Sie die Definition der Klasse zu:

```
static class Sort
```

Schritt 32: Fügen Sie folgenden Code in Sort ein:

```
public delegate bool Compare(Object v1, Object v2);
```

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.

6.13.2 Benutzung eines Delegate Type

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 33: Fügen Sie folgenden Code in Sort.cs ein:

```
static public void DoSort(Object[] dataToSort, Compare greaterThan)
{
    int outer;  //index outer element
    int inner;  //index inner element
    object temp;   //help element
    for (outer = 0; outer <= dataToSort.GetUpperBound(0); outer++)
        for (inner = outer + 1; inner <= dataToSort.GetUpperBound(0); inner++)
           if (greaterThan.Invoke(dataToSort[outer], dataToSort[inner]))
            {
                temp = dataToSort[outer];
                dataToSort[outer] = dataToSort[inner];
                dataToSort[inner] = temp;
            }
}
```

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

6.13.3 Implementierung einer Delegate Methode

Jetzt müssen wir noch eine Methode zum Vergleich zweier Objekte bauen, die 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 34: Fügen Sie folgenden Code in die Person Klasse ein:

```
public static bool CompareAge(object person1, object person2)
{
    return (((Person)person1).Age() > ((Person)person2).Age());
}
```

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.

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 35: Fügen Sie folgenden Code in die Datei Program.cs ein:

```
//create person array and initialize it
Person[] myPeople = new Person[5];
myPeople[0] = new Person("Herbert","9.7.1965");
myPeople[1] = new Person("Kurt","21.1.1955");
myPeople[2] = new Person("Eva","1.2.1960");
myPeople[3] = new Person("Paul","13.5.1970");
myPeople[4] = new Person("Otto","1.10.1975");

//sort the person array by age
Sort.DoSort(myPeople,Person.CompareAge);  //short form

//output data to console window: name and age
foreach (Person p in myPeople)
    Console.WriteLine("{0}: {1}", p.Name, p.Age().ToString());
```

Schritt 36: 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 37: Fügen Sie folgenden Code in die Person Klasse ein:

```
public static bool CompareName(object person1, object person2)
{
    if ((String.Compare(((Person)person1).Name,
                ((Person)person2).Name)) != -1)
        return true;
    else
        return false;
}
```

Anmerkung: Auch hier gibt es wieder verschiedene Schreibweisen, an die Sie sich gewöhnen müssen:

```
public static bool CompareName(object person1, object person2)
{
    if ((String.Compare(((Person)person1).Name, ((Person)person2).Name)) != -1)
        return true;
    else
        return false;
}
```

oder

```
public static bool CompareName(object person1, object person2)
{
```

```
    return (String.Compare(((Person)person1).Name,((Person)person2).Name)) != -1;
}
```

oder

```
public static bool CompareName(object person1, object person2)
{
    return (String.Compare(((Person)person1).Name,((Person)person2).Name)) !=
                -1 ? true : false;
}
```

Schritt 38: Ändern Sie folgende Zeile in der Datei Programm.cs:

```
//sort the person array by name
Sort.DoSort(myPeople, Person.CompareName);
```

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.

6.13.4 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.

```
class Program
{
    // 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 bool MyDelegate(int i);

        static void Main(string[] args)
        {
            // create a variable of type MyDelegate
            MyDelegate myDelegateVariable = null;
            // this can hold the address of a function with matching signature
            // both lines are correct:
            myDelegateVariable = new MyDelegate(DoSomething);
            myDelegateVariable = DoSomething;

            // execute the code pointed to by the variable using Invoke()
            int input = 4;
            bool b = myDelegateVariable.Invoke(input);
            Console.WriteLine("Result: " + b.ToString());

            Console.ReadLine();
        }

        public static bool DoSomething(int i)
        {
            if (i < 0) { return false; } else { return true; }
        }
    }
```

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.

6.14 Fragen

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

6.15 Kompetenz-Check

	Ziel erreicht?	ja	nein
	Ich kenne die Bestandteile einer Klasse.	☐	☐
	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.	☐	☐

6.16 Wo stehen wir?

In diesem Kapitel haben Sie die Grundbausteine des Objektorientierten Programmierens kennen gelernt: Klassen, Objekte, Methoden, Properties, Events und Delegates und anhand eines Beispiels geübt. Das nächste Kapitel beschäftigt sich mit Vererbung (Inheritance) und multiplen Interfaces.