Attribute

Attribute

Benutzerdefinierbare Informationen über Programmelemente

- Kann man an Typen, Members, Assemblies, etc. anhängen.
- Erweitern vordefinierte Attribute wie *public*, *sealed* oder *abstract*.
- Werden als Klassen implementiert, die von *System.Attribute* abgeleitet sind.
- Werden in Metadaten gespeichert.
- Werden teilw. von CLR-Services benutzt (Serialisierung, Remoting, COM-Interoperabilität)
- Können zur Laufzeit abgefragt werden.

Beispiel

```
[Serializable]
class C {...}                          // macht die Klasse serialisierbar
```

Auch mehrere Attribute zuordenbar

```
[Serializable] [Obsolete]
class C {...}

[Serializable, Obsolete]
class C {...}
```

Attribute mit Parametern

Beispiel

Positionsparameter

Namensparameter
kommen nach Pos.parametern

```
[Obsolete("Use class C1 instead", IsError=true)]   // bewirkt Compilefehlermeldung, dass
public class C {...}                               // C obsolet ist
```

Positionsparameter = Parameter des Attributkonstruktors
Namensparameter = alle Attributproperties möglich

Attributdeklaration

```
public class ObsoleteAttribute : Attribute {       // Klassenname endet auf Attribute
    public string Message { get; }                 // kann aber als Obsolete verw. werden
    public bool IsError { get; set; }
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string msg) {...}
    public ObsoleteAttribute(string msg, bool error) {...}
}
```

Daher auch möglich

```
[Obsolete]                                         // Message == "", IsError == false
[Obsolete("some Message")]                         // IsError == false
[Obsolete("some Message", false)]
[Obsolete("some Message", IsError=false)]
```

Werte müssen Konstanten sein

Beispiel: Conditional-Attribut

Bedingter Aufruf von Methoden

```
#define debug                                   // Präprozessor-Anweisung

class C {

   [Conditional("debug")]                       // geht nur bei void-Methoden
   static void Assert (bool ok, string errorMsg) {
      if (!ok) {
         Console.WriteString(errorMsg);
         System.Environment.Exit(0);      // geordneter Programmabbruch
      }
   }

   static void Main (string[] arg) {
      Assert(arg.Length > 0, "no arguments specified");
      Assert(arg[0] == "...", "invalid argument");
      ...
   }
}
```

Assert wird nur aufgerufen, wenn *debug* definiert ist.
Auch gut verwendbar für Hilfsdrucke.

Beispiel: Serialisierung

```
[Serializable]
class List {
    int val;
    [NonSerialized] string name;
    List next;

    public List(int x, string s) {val = x; name = s; next = null;}
}
```

[Serializable] Anwendbar auf Klassen.
Daten von Objekten dieser Klassen werden automatisch serialisiert.

[NonSerialized] Anwendbar auf Felder.
Diese Felder werden von der Serialisierung ausgenommen.

Folgendermaßen hätte man noch mehr Einfluß darauf, was wie serialisiert wird:

```
class List : ISerializable {
    public List(SerializationInfo i, StreamingContext c) {...}
    public void GetObjectData(SerializationInfo i, StreamingContext c) {...}
    ...
}
```

Liest die serialisierten Felder von i

Schreibt die zu serialisierenden Felder nach i

Serialisierung (Forts.)

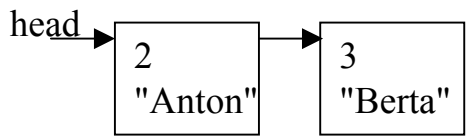
```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

class Test {

    static void Write(List head) {
        FileStream s = new FileStream("MyFile", FileMode.Create);
        IFormatter f = new BinaryFormatter();
        f.Serialize(s, head);
        s.Close();
    }

    static List Read() {
        FileStream s = new FileStream("MyFile", FileMode.Open);
        IFormatter f = new BinaryFormatter();
        List head = f.Deserialize(s) as List;
        s.Close();
        return head;
    }

    static void Main() {
        List head = new List(2, "Anton"); head.next = new List(3, "Berta");
        Write(head);
        List newList = Read();
    }
}
```



Beispiel: AttributeUsage

Verwendung neuer Attribute muß durch *AttributeUsage* festgelegt werden

```
public class AttributeUsageAttribute : Attribute {
    public AttributeUsageAttribute (AttributeTargets validOn) {...}
    public bool AllowMultiple { get; set; }          // default: false
    public bool Inherited { get; set; }              // default: false
}
```

validOn	An welche Programmelemente darf das Attribut angehängt werden?
AllowMultiple	Darf es mehrfach angehängt werden?
Inherited	Wird es von Unterklassen geerbt?

Verwendung

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface, AllowMultiple=false)]
public class MyAttribute : Attribute {...}
```

Eigene Attribute

Deklaration des Attributs

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface, Inherited=true)]
class Comment : Attribute {
    string text, author;
    public string Text { get {return text;} }
    public string Author { get {return author;} set {author = value;} }
    public Comment (string text) { this.text = text; author ="HM"; }
}
```

Anwendung des Attributs

```
[Comment("This is a demo class for Attributes", Author="XX")]
class C { ... }
```

Abfrage des Attributs zur Laufzeit

```
class Attributes {

    static void Main() {
        Type t = typeof(C);
        object[] a = t.GetCustomAttributes(typeof(Comment), true);
        Comment ca = (Comment)a[0];
        Console.WriteLine(ca.Text + ", " + ca.Author);
    }
}
```

soll auch in Unter-klassen gesucht werden?