Was ist Objektorientierte Programmierung?

- Software wird als Menge interagierender Objekte modelliert
- ein Objekt ist ein "Ding", das gewisse Merkmale und Verhalten besitzt, also etwas "tun" kann
- Objekte haben eine Identität
- Objekte werden anhand einer Klasse (der Bauplan) erzeugt
- Mit Hilfe einer Klasse können wir gleichartige Objekte erzeugen, d.h. sie besitzen gleiche Merkmale und gleiches Verhalten
- das Verhalten eines Objektes wird mit einer Menge von Methoden implementiert
- den Zustand eines Objektes implementieren wir mit *Feldern* (Instanzvariablen)
- im Unterschied zur prozeduralen Programmierung werden bei der OOP Daten und darauf arbeitende Funktionen in einer logischen Einheit gekapselt
- OOP fördert bzw. erleicht die Wiederverwendung von Implementierungen
- der Zugriff auf die Daten und das Verhalten eines Objektes kann kontrolliert bzw. eingeschränkt werden
- OOP fördert die Lesbarkeit und Wartbarkeit der Codebase

Properties in C#

- Properties erlauben uns, auf Daten lesend und schreibend zuzugreifen, dabei jedoch zusätzliche Logik auszuführen
- Die Syntax ist dieselbe wie beim Zugriff auf herkömmliche Felder
- Properties erlauben unterschiedliche Zugriffsmodifizierer für das Lesen und Schreiben
- der Wert eines Properties kann aus anderen Properties und Feldern berechnet werden
- verwende das Attribut `MemberNotNull` am Setter eines Properties, um dem Compiler mitzuteilen, dass nach erfolgreicher Zuweisung ein Feld nicht `null` ist

```
object.fieldName = value; // Keine Logik bei direktem Feldzugriff
möglich.
object.SetField(value); // Alternative 1: Setter und Getter Methoden
```

```
schreiben und das Feld als private deklarieren.
object.GetField();
object.Property = value; // Alternative 2: Ein Property definieren
und die Logik in set/get Blöcken definieren.
```

Konstruktor und Konstruktordelegation

- ein Konstruktor ist eine spezielle Methode, die nur beim Erzeugen eines neuen Objektes aufgerufen wird
- ein Konstruktor hat die Aufgabe, ein Objekt in einen gültigen Anfangszustand zu bringen
- lassen sich wie jede andere Methode überladen
- können vor ihrer Arbeit einen anderen Konstruktor per Delegation aufrufen
  - dadurch können "Ketten" von Konstruktoraufrufen entstehen
- tragen denselben Namen wie ihre zugehörige Klasse

```
class Point
{
        public int X { get; private set; }
        public int Y { get; private set; }

        // Konstruktor A mit zwei Parametern
        public Point(int x, int y)
        {
                X = x;
                Y = y;
        }

        // Dieser Konstruktor B ruft vor seiner Ausführung
        // den obigen Konstruktor A auf.
        public Point(int n) : this(n, n)
        {
        }

        // Dieser Konstruktor C ruft den Konstruktor B auf, der
wiederum A aufruft.
        //
        public Point() : this(0)
        {
        }
}
```