

Erweiterte Grundlagen

Inhalt

- Boxing / Unboxing
- Klassen und Structs
- Indexers
- Operatoren überladen
- Konversionsoperatoren überladen
- Überlaufprüfungen

Type Inferenz -- var

```
var x = ...;
```

- *var* kann nur für lokale Variablendeklarationen verwendet werden (nicht für Parameter und Felder)
- Variable muss mit der Deklaration initialisiert werden
- Der Type der Variable wird aus dem Initialisierungsausdruck abgeleitet

```
var x = 3;
```

```
var s = "John";
```

```
var dict = new Dictionary<string, int>();
```

```
var obj = new { Width = 100, Height = 50 };
```

```
int x = 3;
```

```
string s = "John";
```

```
Dictionary<string, int> dict =  
    new Dictionary<string, int>();
```

```
??? obj = ...
```

Boxing / Unboxing

- Jeder Typ erbt von Object
- Unterscheidung Wertetypen und Referenztypen
 - Wertetypen: Stack
 - Referenztypen: Heap
- Object ist ein Referenztyp
- Ist folgendes möglich?

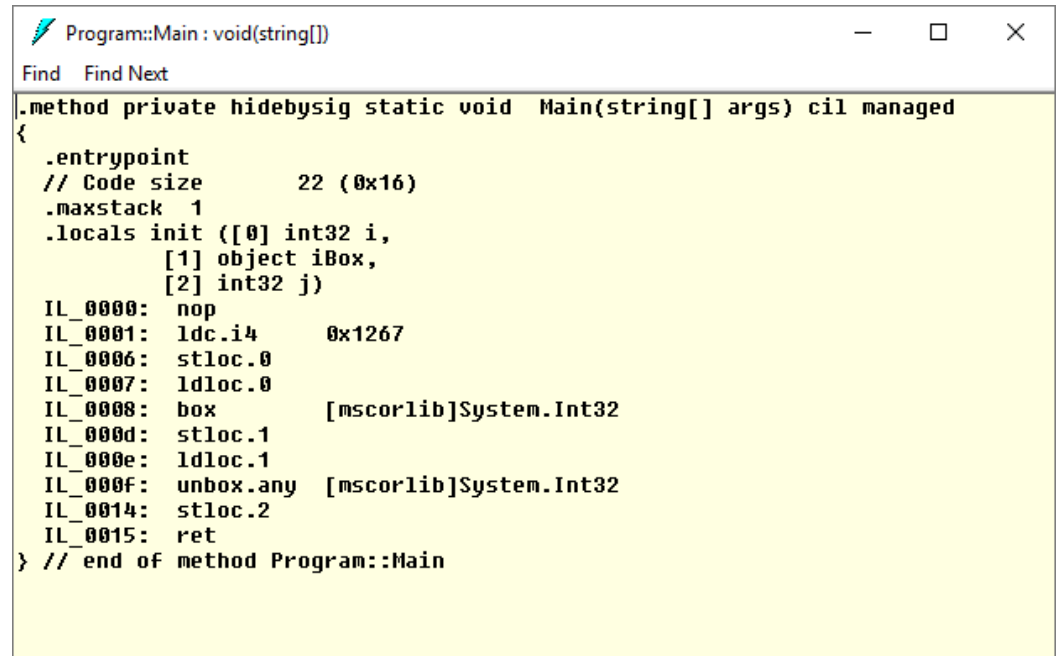
```
int i = 4711;  
object iBox = i;
```

 - ja – aber was passiert?
 - 4711 wird in ein «object verpackt» → boxing
- Unboxing

```
int j = (int) iBox;
```

Beispiel

```
class Prog {  
    static void Main() {  
        int i = 4711;  
        object iBox = i;  
        int j = (int) iBox;  
    }  
}
```



The screenshot shows a Visual Studio window titled "Program::Main : void(string[])" with a search bar containing "Find" and "Find Next". The main area displays the Intermediate Language (IL) code for the Main method, which is private, hidebysig, static, and void. The code is managed and cil. It starts with an entrypoint, followed by metadata: code size 22 (0x16), maxstack 1, and locals init for int32 i, object iBox, and int32 j. The IL instructions are: IL_0000: nop, IL_0001: ldc.i4 0x1267, IL_0006: stloc.0, IL_0007: ldloc.0, IL_0008: box [mscorlib]System.Int32, IL_000d: stloc.1, IL_000e: ldloc.1, IL_000f: unbox.any [mscorlib]System.Int32, IL_0014: stloc.2, and IL_0015: ret. The method ends with a comment "end of method Program::Main".

```
.method private hidebysig static void Main(string[] args) cil managed  
{  
    .entrypoint  
    // Code size          22 (0x16)  
    .maxstack 1  
    .locals init ([0] int32 i,  
                  [1] object iBox,  
                  [2] int32 j)  
    IL_0000: nop  
    IL_0001: ldc.i4      0x1267  
    IL_0006: stloc.0  
    IL_0007: ldloc.0  
    IL_0008: box          [mscorlib]System.Int32  
    IL_000d: stloc.1  
    IL_000e: ldloc.1  
    IL_000f: unbox.any  [mscorlib]System.Int32  
    IL_0014: stloc.2  
    IL_0015: ret  
} // end of method Program::Main
```

Klassen und Structs

- In C# gibt es zwei strukturierte Datentypen
- Klassen
 - Mittels new werden Instanzen erstellt
 - Werte werden auf dem Heap abgelegt
 - Referenztypen
 - Haben «volle» OO-Funktionalität (Vererbung usw.)
- Structs
 - Mittels new werden Werte initialisiert
 - Manchmal auf dem Stack, manchmal auf dem Heap
<https://stackoverflow.com/a/2866346/2042829>
 - Wertetypen
 - Haben «eingeschränkte» OO-Funktionalität
- Vorteile
 - Performance: der Stack kann effizienter verwaltet werden als der Heap
 - Modellierung von eigenen Wertetypen möglich, z.B. Complex, Vector

Indexers (weiteres Beispiel)

```
class MonthlySales {  
    int[] apples = new int[12];  
    int[] bananas = new int[12];  
    ...  
    public int this[int i] { // set-Methode fehlt => read-only  
        get { return apples[i-1] + bananas[i-1]; }  
    }  
  
    public int this[string month] { // überladener read-only-Indexer  
        get {  
            switch (month) {  
                case "Jan": return apples[0] + bananas[0];  
                case "Feb": return apples[1] + bananas[1];  
                ...  
            }  
        }  
    }  
}  
  
MonthlySales sales = new MonthlySales();  
Console.WriteLine(sales[1] + sales["Feb"]);
```

Operatoren implementieren/überladen

- Statische Methode, die wie ein Operator verwendet werden kann

```
struct Fraction {  
    int x, y;  
    public Fraction (int x, int y) {this.x = x; this.y = y; }  
    public static Fraction operator + (Fraction a, Fraction b) {  
        return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);  
    }  
}
```

- Verwendung

```
Fraction a = new Fraction(1, 2);  
Fraction b = new Fraction(3, 4);  
Fraction c = a + b;    // c.x == 10, c.y == 8
```

- Überladbare Operatoren:

- arithmetische: +, - (unär und binär), *, /, %, ++, --
- Vergleichsoperatoren: ==, !=, <, >, <=, >=
- Bitoperatoren: &, |, ^
- Sonstige: !, ~, >>, <<, true, false

- Müssen immer ein Ergebnis liefern

Überladen von Konversionsoperatoren

Implizite Konversion

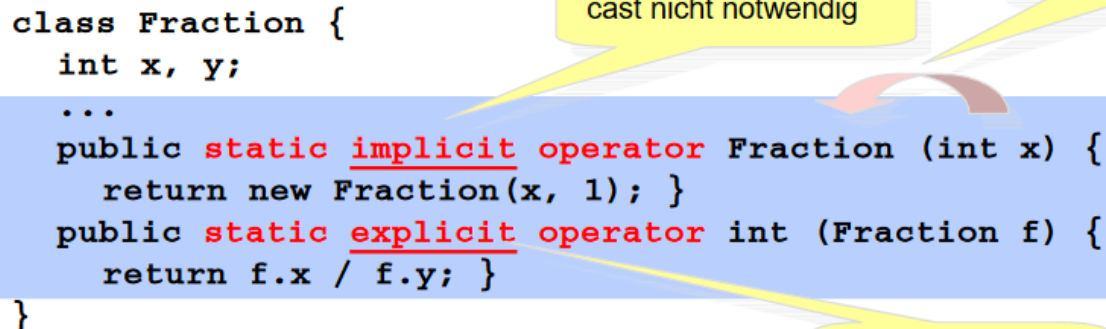
- Wenn Konversion immer möglich ist und kein Genauigkeitsverlust stattfindet
- Z.B.: long = int;

Explizite Konversion

- Wenn Laufzeittypprüfung nötig ist oder u.U. abgeschnitten wird
- Z.B. int = (int) long;

Konversions-Operatoren für eigenen Typ

```
class Fraction {  
    int x, y;  
    ...  
    public static implicit operator Fraction (int x) {  
        return new Fraction(x, 1);  
    }  
    public static explicit operator int (Fraction f) {  
        return f.x / f.y;  
    }  
}
```



Verwendung

```
Fraction f = 3;           // implizite Konversion, f.x == 3, f.y == 1  
int i = (int) f;          // explizite Konversion, i == 3
```

Überlaufprüfungen

Normalerweise wird Überlauf nicht erkannt

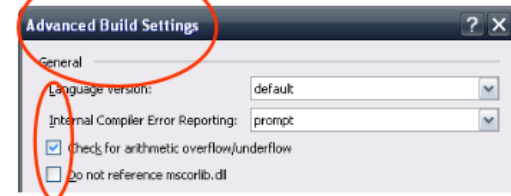
```
int x = 1000000;  
x = x * x;    // -727379968, kein Fehler
```

Überlaufprüfung

```
x = checked(x * x); // liefert System.OverflowException  
checked {  
    ...  
    x = x * x;      // liefert System.OverflowException  
}
```

Es gibt auch Compiler-Option, um Überlaufprüfung generell einzuschalten

```
csc /checked Test.cs
```



Selbststudium

- Probieren Sie die behandelten Themen selber aus
 - Erstellen Sie eine Klasse «3DPoint», in der Sie folgende Teile implementieren:
 - Einen Indexer, mit dem Sie die einzelnen Koordinatenwerte abrufen können. Z.B. soll `point[0]` den X-Wert des Punktes zurückliefern, `point[1]` entsprechend den Y-Wert und `point[2]` den Z-Wert
 - Operatoren um zwei Punkte zu addieren (+), zu subtrahieren (-) sowie zu vergleichen (`==`, `<`, `<=`, `>`, `>=`).
Stellen Sie bei der Addition und Subtraktion sicher, dass kein Überlauf auftreten kann – bzw. dass bei einem Überlauf eine Exception ausgelöst wird.