Typen

Einheitliches Typsystem

Typen

- Werttypen
  - Einfache Typen
  - Enums
  - Structs
- Referenztypen
  - Klassen
  - Interfaces
  - Arrays
  - Delegates
- Zeiger

Einfache Typen:

bool	sbyte	byte	float
char	short	ushort	double
	int	uint	decimal
	long	ulong	

Benutzerdefinierbare Typen (Enums, Structs, Klassen, Interfaces, Arrays, Delegates)

Alle Typen sind kompatibel mit object

- können *object*-Variablen zugewiesen werden
- verstehen *object*-Operationen

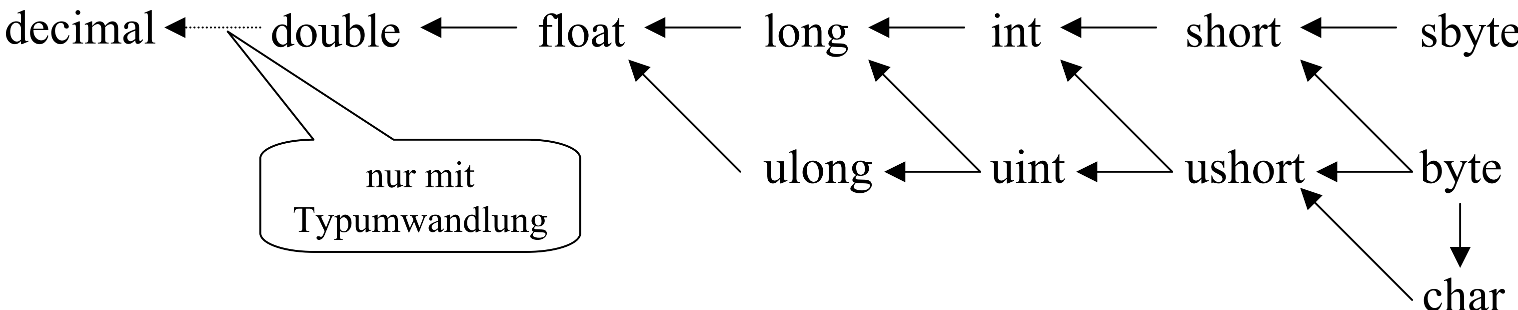
Werttypen versus Referenztypen

	Werttypen	Referenztypen
Variable enthält	Wert	Zeiger auf ein Objekt
gespeichert am	Stack	Heap
Initialisierung	0, false, '\0'	null
Zuweisung	kopiert Wert	kopiert Zeiger
Beispiel	int i = 17; int j = i;	string s = "Hello"; string s1 = s;

Einfache Typen

	Langform	in Java	Wertebereich
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	-2^{63} .. $2^{63}-1$
ulong	System.UInt64	---	0 .. $2^{64}-1$
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode-Zeichen

Kompatibilität bei einfachen Typen



Enumerationen

Aufzählungstypen aus benannten Konstanten

Deklaration (auf Namespace-Ebene)

```
enum Color {red, blue, green}    // Werte: 0, 1, 2
enum Access {personal=1, group=2, all=4}
enum Access1 : byte {personal=1, group=2, all=4}
```

Verwendung

```
Color c = Color.blue;     // Enum-Konstanten müssen qualifiziert werden

Access a = Access.personal | Access.group;
if ((Access.personal & a) != 0) Console.WriteLine("access granted");
```

Operationen mit Enumerationen

Erlaubte Operationen

<table>
<tr><td>Vergleiche</td><td>if (c == Color.red) ...
if (c > Color.red && c <= Color.green) ...</td></tr>
<tr><td>+, -</td><td>c = c + 2;</td></tr>
<tr><td>++, --</td><td>c++;</td></tr>
<tr><td>&</td><td>if ((c & Color.red) == 0) ...</td></tr>
<tr><td>|</td><td>c = c | Color.blue;</td></tr>
<tr><td>~</td><td>c = ~ Color.red;</td></tr>
</table>

Es wird <u>nicht</u> geprüft, ob der erlaubte Wertebereich über-/unterschritten wird.

Ferner

- Enumerationen sind nicht zuweisbar an *int* (außer nach Type Cast).
- Enumstypen erben alle Eigenschaften von *object* (*Equals*, *ToString*, ...)
- Klasse *System.Enum* stellt Operationen auf Enumerationen bereit (*GetName*, *Format*, *GetValues*, ...)

Arrays

Eindimensionale Arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];   // Array von Referenzen
SomeStruct[] e = new SomeStruct[10]; // Array von Werten (direkt im Array)
```

Mehrdimensionale Arrays ("ausgefranst", jagged)

```
int[][] a = new int[2][];   // Array von Referenzen auf Arrays
a[0] = {1, 2, 3};           // Können nicht direkt initialisiert werden
a[1] = {4, 5, 6};
```

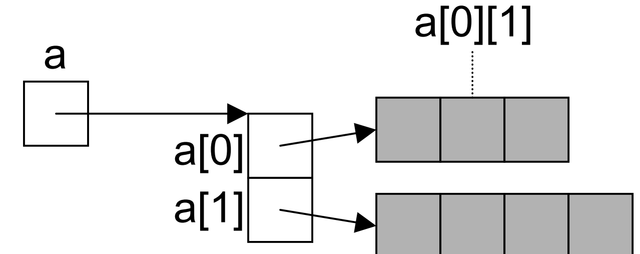
Mehrdimensionale Blockarrays (rechteckig)

```
int[,] a = new int[2, 3];        // Block-Matrix
int[,] b = {{1, 2, 3}, {4, 5, 6}}; // Können direkt initialisiert werden
int[,,] c = new int[2, 4, 2];
```

Matrix-Arten

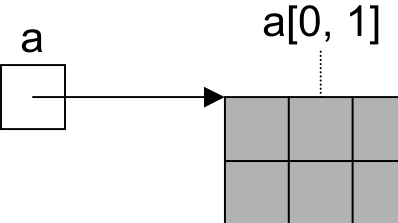
Ausgefranst (wie in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
```



Rechteckig (kompakter, effizienterer Zugriff)

```
int[,] a = new int[2, 3];

int x = a[0, 1];
```



Sonstiges über Arrays

Indizierung beginnt bei 0

Arraylänge

```
int[] a = new int[3];
Console.WriteLine(a.Length);  // 3

int[][] b = new int[3][];
b[0] = new int[4];
Console.WriteLine("{0}, {1}", b.Length, b[0].Length);   // 3, 4

int[,] c = new int[3, 4];
Console.WriteLine(c.Length);   // 12
Console.WriteLine("{0}, {1}", c.GetLength(0), c.GetLength(1));  // 3, 4
```

System.Array enthält nützliche Array-Operationen

```
int[] a = {7, 2, 5};
int[] b = new int[2];
Array.Copy(a, b, 2);      // kopiert a[0..1] nach b
Array.Sort(b);
...
```

Klasse System.String

Benutzbar als Standardtyp *string*

```
string s = "Alfonso";
```

Bemerkungen

- Strings sind nicht modifizierbar (dazu *StringBuilder*)
- Können mit + verkettet werden: "Don " + s
- Können indiziert werden: s[i]
- Längenprüfung: s.Length
- Referenztyp, daher Zeigersemantik in Zuweisungen
- aber Wertevergleich mit == und != : if (s == "Alfonso") ...
- Klasse *String* definiert viele nützliche Operationen:
  CompareTo, *CompareOrdinal*, *IndexOf*, *StartsWith*, *Substring*, ...

Variabel lange Arrays

```
using System;
using System.Collections;

class Test {

	static void Main() {
		ArrayList a = new ArrayList();
		a.Add("Caesar");
		a.Add("Dora");
		a.Add("Anton");
		a.Sort();
		for (int i = 0; i < a.Count; i++)
			Console.WriteLine(a[i]);
	}
}
```

Ausgabe:

```
Anton
Caesar
Dora
```

Assoziative Arrays

```
using System;
using System.Collections;

class Test {

	static void Main() {
		Hashtable phone = new Hashtable();
		phone["Karin"] = 7131;
		phone["Peter"] = 7130;
		phone["Wolfgang"] = 7132;
		foreach (DictionaryEntry x in phone)
			Console.WriteLine("{0} = {1}", x.Key, x.Value);
	}
}
```

Ausgabe:

```
Karin = 7131
Peter = 7130
Wolfgang = 7132
```

Structs

Deklaration

```
struct Point {
        public int x, y;                                              // Felder
        public Point (int x, int y) { this.x = x; this.y = y; }      // Konstruktor
        public void MoveTo (int a, int b) { x = a; y = b; }          // Methoden
}
```

Verwendung

```
Point p;                        // noch uninitialisiert
Point p = new Point(3, 4);      // Initialisierung mit Konstruktor-Aufruf (Objekt am Stack!!)
p.x = 1; p.y = 2;               // Feldzugriff
p.MoveTo(10, 20);               // Methodenaufruf
Point q = p;                    // Objektzuweisung (alle Felder)
```

Bemerkungen

- Structs sind Werttypen!
  Objekt durch Deklaration direkt am Stack, in Array, in Klasse oder Struct angelegt
- Konstruktor-Aufruf erzeugt neues Objekt am Stack!
- Structs dürfen keinen parameterloser Konstruktor deklarieren (haben ihn standardmäßig)
  Sie dürfen ihn aber benutzen: `p = new Point(); // initialisiert Felder mit 0, null, false, ...`

Klassen

Deklaration

```
class Rectangle {
	Point origin;
	public int width, height;
	public Rectangle() { origin = new Point(0,0); width = height = 0; }
	public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }
	public void MoveTo (Point p) { origin = p; }
}
```

Verwendung

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
Rectangle r1 = r ;  // Zeigerzuweisung
```

Bemerkungen

- Klassen sind Referenztypen
  Objekte werden am Heap angelegt
- Konstruktor-Aufruf erzeugt neues Objekt am Heap und initialisiert es
  Parameterloser Konstruktor darf deklariert werden

Unterschied zw. Klassen und Structs

Klassen	Structs
Referenztypen (Objekte am Heap angelegt)	Werttypen (Objekte am Stack angelegt)
unterstützen Vererbung (alle Klassen von *object* abgeleitet)	keine Vererbung (aber zu *object* kompatibel)
können Interfaces implementieren	können Interfaces implementieren
Parameterloser Konstruktor erlaubt	keine parameterlosen Konstruktoren
können Destruktoren haben	keine Destruktoren

Klasse System.Object

Basisklasse aller Referenztypen

```
class Object {
	public virtual bool Equals(object o) {...}
	public virtual string ToString() {...}
	public virtual int GetHashCode() {...}
	...
}
```

Wird als Standardtyp *object* benutzt

```
object obj;
```

Zuweisungskompatibilität

```
obj = new Rectangle();
obj = new int[3];
```

Erlaubt generische Methoden

```
void Push(object x) {...}
Push(new Rectangle());
Push(new int[3]);
```

Boxing und Unboxing

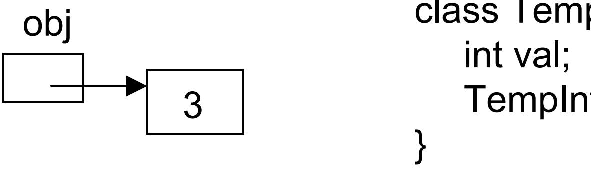
Auch Werttypen (int, struct, enum) sind zu *object* kompatibel!

Boxing

Bei der Zuweisung

```
object obj = 3;
```

wird der Wert 3 in ein Heap-Objekt eingepackt




```
class TempInt {
    int val;
    TempInt(int x) {val = x;}
}

obj = new TempInt(3);
```

Unboxing

Bei der Zuweisung

```
int x = (int) obj;
```

wird der eingepackte int-Wert wieder ausgepackt

Boxing/Unboxing

Erlaubt generische Container-Typen

```
class Queue {
	...
	public void Enqueue(object x) {...}
	public object Dequeue() {...}
	...
}
```

Diese Queue kann für Referenz- und Werttypen verwendet werden

```
Queue q = new Queue();

q.Enqueue(new Rectangle());
q.Enqueue(3);

Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```