

Lebenszeit von Objekten - Garbage collection

Wodurch wird ein Objekt un erreichbar und damit “verwaist”?

Garbage collector erhält CPU und auf ein bestimmtes Objekt zeigt keine Referenzvariable mehr durch:

- lokale Referenzvariable werden beim Verlassen des Scopes ungültig
- Referenzvariable können explizit auf “null” gesetzt werden
- einer Referenzvariablen kann ein anderes Zielobjekt zugewiesen werden

```
using System;
```

```
class Program
```

```
{  
    CTruck c4=null;  
    static void Main ()  
    {  
        var c1 = new CTruck (2,150,2);    // Objekt a  
        var b2 = new CTruck ();           // Objekt b  
        c1 = b2;                          // Objekt a verwaist, ist nicht mehr erreichbar  
        var b3 = new CTruck ();           // Objekt c  
        c4 = new CTruck ();               // Objekt d  
        b3 = null;                        // Objekt c verwaist, nicht mehr erreichbar  
    }    // ab hier Objekt b verwaist, da c1 und b2 hier ungültig werden. Nur Objekt d bleibt erhalten.  
}
```

Aufräumarbeiten beim Objekt-Tod: Destruktor

Finalisierer (alias: **Destruktor**): spezielle Methode für das Aufräumen von Objekten
(Gegenstück zu den Konstruktoren)

Aufgaben:

- Freigabe von Ressourcen, die *nicht* von der CLR verwaltet werden (z. B. Datei-, Netzwerk- oder Datenbankverbindungen) – Destruktor nur definieren, wenn diese Aufgabe zu übernehmen ist!

Ziel:

- alle Ressourcen des Objekts vor seiner endgültigen Löschung freigeben

Aufruf:

- ausschließlich vom Garbage collector

Abarbeitungsfolge:

- jeder Destruktor (auch default-Destruktor) ruft am Ende automatisch den Destruktor der nächsthöheren Klasse im Stammbaum auf (usw.), zerstört wird von unten nach oben

Notwendigkeit:

- nur, wenn andere Ressourcen als Speicher allokiert wurden – sonst weglassen!



Destruktoren/ Finalizer

Syntax:

- Methodenname entspricht *~Klassenname*
- kein Rückgabewert, keine Parameterliste
- kein „public“
- nur 1 Destruktor pro Klasse erlaubt

```
~CVehicle()
{
    if (logfile)
    {
        logfile.Close();
        logfile=null;
    }
}
```



Sofortige Ressourcenfreigabe vor GC

Für sofortige Freigabe der Ressourcen eines verwaisten Objekts (ohne Warten auf den GC):

- Interface **IDisposable** implementieren, d.h.: die von der Schnittstelle **IDisposable** geforderte Methode **Dispose()**

Darin: die von einem Objekt belegten unmanaged Ressourcen (z. B. Datei-, LAN- oder DB-Verbindungen) freigeben.

Nutzer der Klasse rufen explizit **Dispose(true)** auf für Freigabe des Objekts, wenn dieses nicht mehr benötigt wird.



Sofortige Ressourcenfreigabe vor GC

Empfehlung für sicheres Disposing:

- Unverwaltete Ressourcen in ein Objekt aus einer Ableitung der Klasse **SafeHandle** (Namensraum **System.Runtime.InteropServices**) verpacken (SafeFileHandle, SafePipeHandle, SafeWaitHandle, ...) -> managed!
- diese Klasse verfügt über eine robuste, auch unter ungünstigen Bedingungen korrekt arbeitende Dispose-Methode, die von der Dispose()-Methode der Klasse aufgerufen werden kann
 - > Wird das versäumt, dann ruft der Garbage Collector die (besonders gegen mehrmaligen Aufruf robuste, Dispose(true/false) vermeidende) Finalisierungsmethode der **SafeHandle** -Ableitung auf.

Sofortige Ressourcenfreigabe vor GC

```
using Microsoft.Win32.SafeHandles;  
using System.IO;
```

```
public class CMyFile: IDisposable  
{  
    private SafeHandle myFile;  
    private bool disposed = false;  
  
    CMyFile( /* ... */ ) // ctor  
    {  
        myFile = File.OpenHandle  
            ("myfile", FileMode.Create, FileAccess.Write);  
    }  
    public void Dispose()  
    {  
        Dispose(disposing: true);  
        GC.SuppressFinalize(this);  
    }  
    ~CMyFile() // nicht mehr unbedingt notwendig  
    {  
        Dispose(disposing: false);  
    }  
    // ...  
}
```

```
protected virtual void Dispose(bool disposing)  
{  
    if (!disposed)  
    {  
        if (disposing)  
        {  
            myFile.Dispose(); // Dispose von SafeFileHandle,  
                               // managed type  
            myFile = null;  
        }  
        disposed = true;  
    }  
}
```



Rückgabewerte mit Referenztyp:

Soll ein methodenintern erzeugtes Objekt das Ende der Methodenausführung überleben, kann ein Rückgabewert mit Referenztyp eine Referenz zum Aufrufer der Methode geben:

```
public CTruck CreateNew()  
{  
    return new CTruck (2, 150, 2);  
}
```

Aufruf: `var c2 = c1.CreateNew();`



this enthält die Referenz auf das aktuelle Objekt:

- ein Objekt kann sich selbst ansprechen (z.B. bei Übereinstimmen von Instanz- mit Formalparameternamen):

```
public bool CVehicle (float i_pricePerDay, int seats)
{   this.pricePerDay      // Feld der Instanz
    = i_pricePerDay;      // Eingangsparameter
}
```




this enthält die Referenz auf das aktuelle Objekt:

- seine eigene Adresse als Methodenparameter an andere Objekte weitergeben („fluent interface“), zum Methodenchaining/-cascading

```
Customer c1 = new Customer(); // Object creation, all methods return „this“  
public Customer FirstName(string firstName)  
    { _context.FirstName = firstName; return this; }  
  
c1.FirstName("vinod").LastName("srivastav").Sex("male").Address("bangalore").Print();  
    // Using method chaining to assign & print data with a single line
```



Operatoren überladen

Ziel:

Verknüpfung/ Verrechnung von Objekten miteinander
(Operatoren, die sich auf die Objekte der Klasse beziehen, nicht auf deren einzelne Felder!)

Bsp.: Addition zweier Brüche:

Operator +

Verrechnung zweier komplexer Zahlen: Operator *

```
class CComplex { int real; int imag; }
```

Vergleich zweier Bücher bzgl. Erscheinungsjahr:

Operator <

```
class CBook { string autor;  
             string title;  
             int    year;  
             }
```

Preisvergleich zweier Leihwagen:

Operator >

```
class CVehicle { float pricePerDay;  
                // ...  
                }
```



Operatoren überladen

Klassenoperator:

Statische Methode ohne Objektbezug mit dem Namen des Operators, z.B. „**operator+**“.

Zu verarbeitende Objekte werden als Parameter übergeben!

Ein unärer Operator bekommt 1 Eingabeparameter. Ein binärer Operator bekommt 2 Eingabeparameter.

Statt

```
Bruch b3 = new Bruch(Zaehler=b1.Zaehler*b2.Nenner+b2.Zaehler*b1.Nenner, Nenner=b1.Nenner*b2.Nenner);
```

wäre es eleganter, denselben Zweck mit der folgenden Anweisung zu erreichen:

```
Bruch b3 = b1 + b2; // dazu „operator+“ als statische Methode der Klasse Bruch überladen
```

Addiert zwei Brüche:

```
public static Bruch operator+ (Bruch b1, Bruch b2) // Operatorüberladungen in C# immer statisch!  
{  
    Bruch temp = new Bruch(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler, b1.nenner * b2.nenner);  
    temp.Kuerze();  
    return temp;  
}
```



Operatoren überladen

Vergleicht zwei Leihwagen nach Preis:

```
public static bool operator> (CVehicle v1, CVehicle v2)
{
    bool erg=false;
    if (v1.PricePerDay > v2.PricePerDay) erg=true;
    return erg;
}
```

Nutzung:

```
CVehicle veh1 (75.40, 4);
CVehicle veh2 (137.90, 5);
// ...

if (veh1 > veh2) ... // false
```

Indexer (Operator [] überladen)

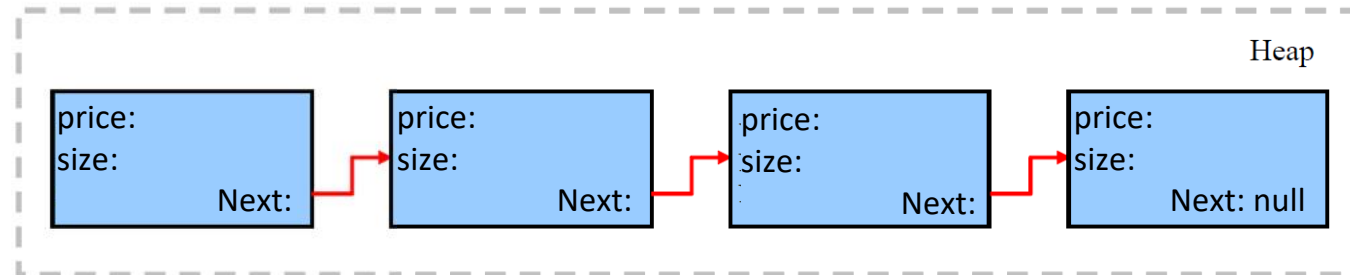
Bei Arrays, Klasse **String** und Klasse **ArrayList**: Standard-Indexzugriff per [] - Operator ist sehr nützlich

Für eine eigene Klasse, Kollektion, Struktur oder Liste mit zahlreichen Elementen soll ein wahlfreier Zugriff auf die Elemente möglich sein: -> **eigenen Indexer (Index-Operator) schreiben**

Wie bei Property: ein *Paar von Methoden* für den lesenden bzw. schreibenden Zugriff auf ein Element per Indexsyntax.

Bsp.: *CFleet* enthält verkettete Objekte der Klasse *CVehicle*, aber Zugriff mittels *CStock[]* soll möglich sein:

```
class CVehicle // als Liste organisierbar
{
    public float PricePerDay { get; set; }
    public int Seats { get; };
    public CVehicle Next { get; set; };
    // zeigt auf nächstes Objekt
    public CVehicle (float i_price, int i_seats) //C'tor
    {
        PricePerDay=i_price;
        Seats=i_seats;
    }
}
```





Regeln für Indexer

Neue Methode für Klasse *CFleet*: `public CVehicle this [int i] // gibt für Index i ein Objekt zurück`

Nach den optionalen Modifikatoren (z.B. *public*) wird der Rückgabe-Datentyp angegeben (im Beispiel: *CVehicle*).

- Der **Name der Methodendefinition** lautet stets **this [...]**.
- Hinter dem Schlüsselwort **this** wird *eckig* eingeklammert der Indextyp und -name (z.B. `int i`) angegeben.
- Der **set**-Methode wird wie bei Properties ein impliziter Parameter namens **value** übergeben.

Ein Objekt der Klasse CFleet verwaltet eine verkettete Liste von CVehicle-Objekten:

```
// in class CVehicle:
private CVehicle Next { get; set; }

class CFleet
{
    int n; // Anzahl, initial null
    CVehicle first, last; // Anfangs- und Endeanker

    public int ItemCount
    {
        get { return n; }
    }

    public void AddToFleet (CVehicle i_veh) // am Ende hinzu
    {
        if (i_veh == null) return;
        if (n == 0) first = last = i_veh; // Referenz!
        else
        { last.Next = i_veh; // letztes Element zeigt auf neues
          last = i_veh; // Endeanker verändern
        }
        n++;
    }
}
```

```
public CVehicle this[int i] // Fortsetzung CFleet, gibt für Index i i-tes CVehicle zurück
{
    get // lesender Zugriff auf Index i
    { if (i >= 0 && i < n)

        { CVehicle sel = first;
          for (int j = 0; j < i; j++) sel = sel.Next;
          return sel;
        }
        else return null;
    }
    set // schreibender Zugriff auf i-tes Element
    { if (i >= 0 && i < n && value != null) // CVehicle value; null setzen nicht erlaubt;
        // Aufruf: fleet[i]=vehicle;
        { if (i == 0) // erstes Element
            { value.Next = first.Next; // Next-Wert des Neuen zeigt auf den Nachfolger
              first = value; // Der Neue wird zum Startelement
            }
            else // nicht erstes Element
            { CVehicle current = first; // Referenz
              for (int j = 0; j < i - 1; j++) current = current.Next; // bis Index
              value.Next = current.Next.Next; // Next-Wert des Neuen zeigt auf den Nachfolger
              current.Next = value; // current.Next ersetzt
            }
        }
    }
}
```



Indexer: Unittest

```
using System;  
using NUnit.Framework;  
using RentACar;
```

```
namespace TestCases
```

```
{
```

```
    [TestFixture]
```

```
    public class Tests
```

```
    {
```

```
        [Test]
```

```
        public void Test1()
```

```
        {
```

```
            CRentACar rac=new CRentACar();
```

```
            CTruck truck1 = new(2, 140, 2);
```

```
            rac.Fleet.AddToFleet(truck1);
```

```
            CMoped moped1 = new(1, 55f, 1);
```

```
            rac.Fleet.AddToFleet(moped1);
```

```
            CTruck truck2 = new(1, 130, 2);
```

```
            rac.Fleet.AddToFleet(truck2);
```

```
            CVehicle test2 = rac.Stock[2];
```

```
            CVehicle test1 = rac.Stock[1];
```

```
            CVehicle test0 = rac.Stock[0];
```

```
            CVehicle test3 = rac.Stock[3];
```

```
            Assert.True ((test2.Seats==2) &&  
                        ((test1 as CMoped).Helmet==1) &&  
                        ((test0 as CTruck).PayLoad == 2) &&  
                        (null==test3));  
        }  
    }  
}
```


Indexer überladen

Elemente einer Kollektion können alternativ durch *mehrere* Variablen identifiziert werden (z. B. Landkreise durch eine PLZ und einen Namen):

- > Überladen des Indexers durch die Verwendung verschiedener Parametertypen
- > mehrdimensionale Indexer sind möglich

Bsp. ergänzt:

Indexer-Überladung mit zusätzlichem **int**-Parameter:
liefert das n-te Item, dessen *Seats* mit dem übereinstimmt (Index als key) (Bsp.: zweiter Viersitzer)

```
public CVehicle this[int i_seats, int n]
{
    get
    {
        int index=0;
        for (int j = 0; j < n; j++)
            if (this[j].Seats == i_seats)
            { if (index==n) return this[j];
              index++;
            }
        return null;
    }
}
```

[Test]

```
public void Test2()
{
```

```
    CRentACar rac=new CRentACar();
```

```
    CTruck truck1 = new(2, 140, 4);
    rac.Fleet.AddToFleet(truck1);
```

```
    CMoped moped1 = new(1, 55f, 1);
    rac.Fleet.AddToFleet(moped1);
```

```
    CTruck truck2 = new(1, 130, 4);
    rac.Fleet.AddToFleet(truck2);
```

```
    CVehicle test = rac.Fleet[4,2];
```

```
    Assert.True((test.GetType().Name=="CTruck") && (test.PricePerDay==130) );
```



Komposition=Aggregation von Objekten

Als Member einer Klasse sind auch Referenztypen (andere Objekte, also deren Referenz) zugelassen
(z. B. in der Anzug(*CCycleCar*)-Definition zwei Instanzvariable vom Referenztyp ***CMotorcycle*** und ***CSidecar***)

-> es ist möglich, vorhandene Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden

= Wiederverwendung vorhandener Typen bei der Definition neuer Typen

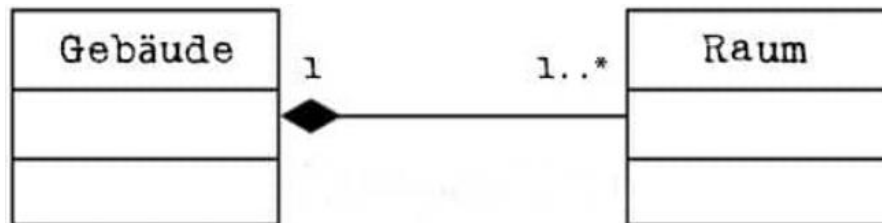
= für realitätsnahe Modellierung unverzichtbar - reale Objekte (z. B. eine Firma) enthalten

andere Objekte (z. B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z. B. ein Gehaltskonto und einen Terminkalender)

= Komposition: „...hat ein...“

(Ableitung: „...ist ein...“)

= UML:





Komposition=Aggregation von Objekten

Bsp. RentACar:

using System;

public class CCycleCar: CVehicle

{

public class CMotorCycle {}

public class CSideCar {}

public CMotorCycle MotorCycle { *get*; }

public CSideCar SideCar { *get*; }

public CCycleCar (*float* i_price, *int* seats): *base* (i_price, seats)

{

 MotorCycle=new CMotorCycle();

 SideCar=new CSideCar();

}

public override void GetNextTUEV()

{

throw new NotImplementedException();

}

}

Alternativen zu Top-Level-Klassen



Innere (eingeschachtelte) Klassen

- Eine Klasse darf neben Variablen, Methoden etc. auch Klassendefinitionen enthalten
- sinnvoll, wenn die innere Klasse nur zur Modellierung von speziellem Zubehör der umgebenden Klasse dient, das außerhalb nicht sichtbar sein muss
- bei Schutzstufe **private** ist die innere Klasse im restlichen Programm (außerhalb der umschließenden Klasse) nicht sichtbar, kann dort nicht verwendet werden und auch keine Namenskollision verursachen.
- innere Klassen haben aber Zugriff auf die privaten Attribute der umgebenden Klassen



Komposition=Aggregation von Objekten

Bsp. RentACar:

using System;

public class CCycleCar: CVehicle

{

public class CMotorCycle {}

public class CSideCar {}

public CMotorCycle MotorCycle { *get*; }

public CSideCar SideCar { *get*; }

public CCycleCar (*float* i_price, *int* seats): *base* (i_price, seats)

{

MotorCycle=*new CMotorCycle*();

SideCar=*new CSideCar*();

}

public override void GetNextTUEV()

{

throw new NotImplementedException();

}

}

Innere (eingeschachtelte) Klassen

```
public class CCycleCar: CVehicle
{
    private CMotorCycle cycle;
    private CSideCar sideCar;

    private class CMotorCycleWithSidecar
    {
    }

    private class CSideCar
    {
    }

    public CCycleCar( float i_price, int i_seats): base (i_price, i_seats)
    {
        sideCar=new CSideCar();
        cycle=new CMotorCycle();
    }
}
```

```
public int GetSeats()
{
    return Seats;
}

public override void GetNextTUEV()
{
    throw new NotImplementedException();
}
}
```



DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Übung 5 - BBB

Übung 5 - Freitextantwort

3) Im folgenden Programm soll die statische Eigenschaft Count ausgelesen werden:

```
using System;
class Program
{
    static void Main()
    {
        var c1 = new Dress(), c2 = new CBoots();
        Console.WriteLine("Jetzt sind wir " + CItem.Count);
    }
}
```

Es liegt folgende Eigenschaftsdefinition zugrunde:

```
static int count;

public static int Count
{
    get { return Count; }
}
```

Statt der erwarteten Auskunft:

Jetzt sind wir 2

erhält man jedoch

(beim Programmstart im Konsolenfenster) die Fehlermeldung:

Process is terminated due to StackOverflowException.

Offenbar hat sich ein Fehler in die Eigenschaftsdefinition eingeschlichen, den der Compiler nicht bemerkt.

Welcher?



Übung 5

- 4) Lokalisieren Sie bitte im Quellcode der Klasse CItem 12 Begriffe der OOP, und geben Sie die Positionen auf Abfrage an!



Übung 5

public abstract class CItem

```
{
    string color;
    static int count; // nur 1x pro Klasse
    public CItem Next { get; set; }
    public CItem()
    {
        count++;
    }
    public CItem(float price, int size)
    {
        Price = price;
        Size = size;
        count++;
    }
    public virtual float Price { get; set; }
    public int size;
    public virtual int Size { get; set; }

    public abstract void Demonstrate();
    public static int GetAnzahl()
    { return count; }
}
```

	Begriff	Pos.
	Definition einer Instanzmethode mit Referenzrückgabe	
B	Deklaration lokale Variable	
	Def. einer Instanzmeth. mit Wertparameter vom Typ einer Klasse	
D	Deklaration von Instanzvariablen	
E	Methodenaufruf	
F	Überschreiben einer Methode	

[Test]

public void Test3()

```
{
    CBoutique bout=new CBoutique();
    CSuit suit = new(100f, 56 );
    Assert.True(suit.GetSuitColor() != "multicolor" );
}

public abstract class CS Shoes: CItem
{
    private int heel;

    protected CS Shoes(int heel, float price, int size):base(price, size)
    {
        this.heel = heel;
    }
    public override int Size
    {
        get { return size; }
        set { if ((value>=35) && (value <=48)) size=value; }
    }
    public override void Demonstrate () {}
}
```

	Begriff	Pos.
	Konstruktordefinition	
	Deklaration einer Klassenvariablen	
	Objekterzeugung	
	Definitionskopf einer Klassenmethode	
	Definition einer Instanzeigenschaft	

G
H
I
J
K



Boxing und Unboxing im Common type system

Alle Klassen stammen direkt oder indirekt von **Object** ab.

Common Type System -> auch Werttypen (int, struct, enum) sind zu **Object** kompatibel!

Problem: Man möchte eine Bibliotheksfunktion mit folgender Signatur aufrufen:

```
void Add (in int a, in int b, out int c); // nur Referenzparameter, Variable sind auf dem Stack
```

Benutzung:

```
void foo()  
{
```

```
    int x=4, y=5, z; // Value type
```

```
    Add (x, y, z); // Gebildete Referenzen würden auf den Stack zeigen! Unmanaged pointers!  
                  // Deshalb Objektkopie „in eine Box“ auf den managed Heap, Referenz zeigt dorthin,  
                  // dann Kopie zurück aus der Box auf den Stack
```



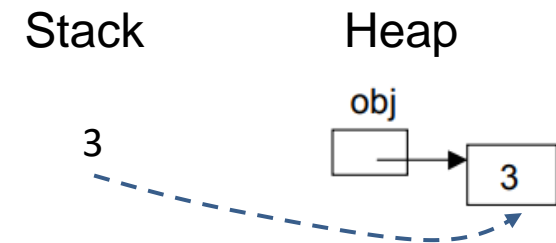
Boxing und Unboxing im Common type system

Alle Klassen stammen direkt oder indirekt von **Object** ab.

Common Type System -> auch Werttypen (int, struct, enum) sind zu **Object** kompatibel!

Boxing:

Bei der Zuweisung `Object obj = 3;`
wird der Wert 3 (value type) in ein Heap-Objekt eingepackt



Unboxing:

Bei der Zuweisung `int x = (int) obj;`
wird der eingepackte int-Wert wieder ausgepackt und auf x kopiert.
x liegt auf dem Stack



Array-Referenzvariablen

- Arrays sind als Klassen definiert: `System.Array` erbt von `System.Object`
- das Objekt enthält (im Gegensatz zu C) noch Verwaltungsdaten (z. B. die per **Length**- Property ansprechbare Anzahl seiner Elemente). Methode `Resize()` erlaubt Verlängerung mittels Umkopieren.
- Klasse `Array` bietet viele statische Methoden zur Suche nach dem 1. Auftreten eines Wertes, z.B. *`IndexOf()`* etc. ...



Array-Referenzvariablen

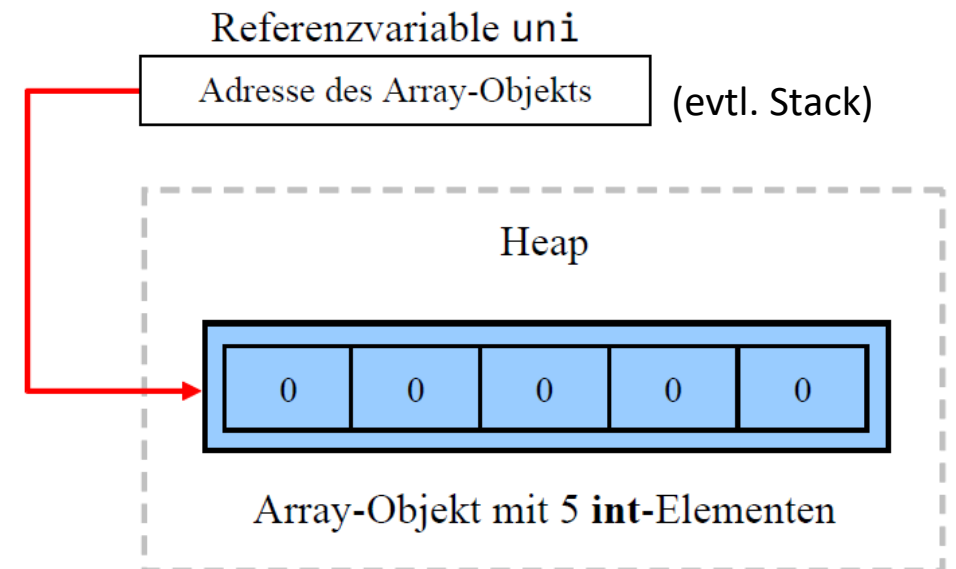
Deklaration:

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Elementtyp und einer bestimmten Größe auf dem Heap:

```
int[] uni;           // nur leere Referenz auf int-Array  
uni = new int[max+1]; // uni zeigt auf max+1 Arrayelemente  
                        // vom Typ int (Index 0... max)
```

oder

```
int[] uni = new int[5]; // uni zeigt auf 5 Arrayelemente vom  
                        // Typ int (Index 0... 4)
```





Array-Initialisierung

- mit Initialisierungslisten statt new+Dimension

```
int[] wecktor = {1, 2, 3}; // Referenztyp mit Initialisierung, Werte auf dem Heap
```

oder

```
int[] wecktor; // Referenz ohne Initialisierung, null  
...  
wecktor = new int[] {1, 2, 3}; oder wecktor = new[] {1, 2, 3};
```

nicht erlaubt:

```
int[] wecktor;  
...  
wecktor = {1, 2, 3}; // keine Elemente angelegt!
```


Quellcode	Ausgabe												
<pre>using System; class Prog { static void Main() { int[,] matrix = new int[4, 3]; // 4 Zeilen, 3 Spalten int nrow = matrix.GetLength(0); // GetLength() liefert die Anzahl der Indexwerte in der int ncol = matrix.GetLength(1); // angegebenen Dimension (0: Zeilen, 1: Spalten) int nelem = matrix.Length; // Property Length liefert die Gesamtzahl der Array-Elemente Console.WriteLine("{0} Dimensionen,\n{1} Zeilen, {2} Spalten" + "\n{3} Elemente", matrix.Rank, nrow, ncol, nelem); // Property Rank enthält die Anzahl for (int i = 0; i < nrow; i++) { // der Dimensionen (den Rang). for (int j = 0; j < ncol; j++) { matrix[i, j] = (i + 1) * (j + 1); Console.Write("{0,3}", matrix[i, j]); } Console.WriteLine(); } } }</pre>	<p>2 Dimensionen, 4 Zeilen, 3 Spalten 12 Elemente</p> <table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>2</td><td>4</td><td>6</td></tr><tr><td>3</td><td>6</td><td>9</td></tr><tr><td>4</td><td>8</td><td>12</td></tr></table>	1	2	3	2	4	6	3	6	9	4	8	12
1	2	3											
2	4	6											
3	6	9											
4	8	12											



Klassen für Strings

Zwei Klassen:

- **String** (im Namensraum **System**), alias **string** `string s1 = "abcde";`

String-Objekte können nach dem Erzeugen nicht mehr geändert werden, nur implizit neu angelegt und zugewiesen.

- **StringBuilder** (im Namensraum **System.Text**)

Für variable, d.h. im Programmablauf änderbare Zeichenketten sollte unbedingt die Klasse **StringBuilder** verwendet werden, weil deren Objekte nach dem Erzeugen noch modifiziert werden können.

Konstruktoren:

public StringBuilder() Nutzung: `StringBuilder sb = new StringBuilder();`

public StringBuilder(String str) Nutzung: `StringBuilder sb = new StringBuilder("abc");`

Klassen für Strings

```
using System;  
using System.Text;
```

```
class StrBlDrDemo
```


```
{  
    static void Main()  
    {  
        const int N = 100000;  
        String s = "*"; // ----- String -----  
        long vorher = DateTime.Now.Ticks; // DateTime.Now.Tick is intended to represent 100 nanoseconds  
        for (int j = 0; j < 10; j++) // 1 Mio Verkettungen  
        {  
            for (int i = 0; i < N; i++) s = s + "*"; // stets neuer String  
            s = "*";  
        }  
        long diff = DateTime.Now.Ticks - vorher;  
        Console.WriteLine("Zeit für String-Manipulation:\t\t" +  
                           diff/ 1.0e4 + "\t\tMillisekunden");  
  
        var t = new StringBuilder("*"); // ----- StringBuilder -----  
        vorher = DateTime.Now.Ticks;  
        for (int j=0; j<10; j++) // 1 Mio Verkettungen  
        {  
            for (int i = 0; i < N; i++) t.Append("*");  
            t.Clear();  
        }  
        diff = DateTime.Now.Ticks - vorher;  
        Console.WriteLine("Zeit für StringBuilder-Manipulation:\t\t" +  
                           diff/ 1.0e4 + "\t\tMillisekunden");  
    }  
}
```

Ausgabe:

```
Zeit für String-Manipulation:      5778,4554      Millisekunden  
Zeit für StringBuilder-Manipulation:  1,9911      Millisekunden
```

Anonyme Klassen

Objektinitialisierer *ohne* Klassenname – es entsteht ein unveränderliches Objekt aus einer anonymen Klasse:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { var a = new {Name = "Knut", Alter = 53 }; Console.WriteLine(a.GetType()); } }</pre>	<pre><>f__AnonymousType0`2[System.String,System.Int32]</pre> <p> vom Compiler vergebener Name</p>

- Equals()-Methode führt einen Inhaltsvergleich durch!
- Zwei durch anonyme Objektinitialisierer entstandene Objekte gehören dann zur selben Klasse, wenn bei den Initialisierungslisten die Namen und Typen der Member sowie die Reihenfolgen übereinstimmen
- zur Unterstützung der LINQ-Technik (*Language Integrated Query*) eingeführt

```
[Test]
public void Test4()
{
    var d1 = new { payload= 1, price = 225f, seats = 3 };

    var d2 = new { payload = 3, price = 77f, seats = 4 };
    //anonym
    Assert.True(d1.GetType() == d2.GetType());
}
```



Nachteile von Strukturen:

- Aufwand durch die Notwendigkeit einer Definition
- erlauben Daten UND Methoden

Tupel: wie Strukturen, aber:

- ohne Definition instanzierbar, indem Typen und Namen (optional!) der Member zwischen (und) deklariert werden
- Weglassen der Namen führt zu automatischen Namen: Item1, Item2,...
- ebenfalls keine Objekte auf dem Heap notwendig
- Tupel selbst und alle Elemente sind public
- besonders nützlich bei **privaten Methoden, die mehr als einen Wert zurückliefern sollen** (der Übersichtlichkeit halber nicht für public Methoden)



Tuples

```
(double, int) t1 = (4.5, 3);  
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
```

```
// Output:  
// Tuple with elements 4.5 and 3.
```

```
(double Sum, int Count) t2 = (4.5, 3);  
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
```

```
// Output:  
// Sum of 3 elements is 4.5.
```



Tuples: named

```
public class CCycleCar: CVehicle
{
    private CMotorCycle cycle;
    private CSideCar sideCar;

    private class CMotorCycleWithSidecar
    {
    }

    private class CSideCar
    {
        public int Seats { get; }
    }

    public CCycleCar( float i_price, int i_seats): base (i_price, i_seats)
    {
        sideCar=new CSideCar();
        cycle=new CMotorCycleWithSidecar();
    }

    public int GetCycleCarSeats()
    {
        return Seats;
    }

    // ...
}
```

Ohne Tuples

```
public class CCycleCar: CVehicle
{
    private CMotorCycle cycle;
    private CSideCar sideCar;

    private class CMotorCycle: CVehicle
    {
    }

    private class CSideCar
    {
        public int Seats { get; }
    }

    public CCycleCar( float i_price, int i_seats): base (i_price, i_seats)
    {
        sideCar=new CSideCar();
        cycle=new CMotorCycle();
    }

    public (int, int) GetCycleCarSeats()
    {
        return (Seats-1, 1);
    }
}
```

Mit Tuples



Tuples: named

[Test]

```
public void Test3()
{
    CRentACar rac=new CRentACar();
    CSideCar scar = new(100f, 3 );

    var seats = scar. GetCycleCarSeats();

    Assert.True(seats == 3 );
}
```

[Test]

```
public void Test3()
{
    CRentACar rac=new CRentACar();
    CSideCar scar = new(100f, 3 );

    var (bikeSeats, sideSeats) = scar. GetCycleCarSeats();

    Assert.True((bikeSeats== 2)&&(sideSeats==1) );
}
```


Im folgenden Programm wird den beiden **object**-Variablen o1 und o2 derselbe **int**-Wert zugewiesen. Wieso haben die beiden Variablen anschließend nicht denselben Inhalt?

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { object o1 = 1; object o2 = 1; Console.WriteLine(o1 == o2); } }</pre>	False



Erstellen Sie ein Programm, das 6 Lottozahlen (von 1 bis 49) zieht (bitte keine Duplikate) und sortiert auf die Konsole ausgibt. Vermutlich werden Sie für die gezogenen Lottozahlen ein eindimensionales **int**-Array verwenden. Dieses lässt sich mit der statischen Methode **Sort()** aus der Klasse **Array** im Namensraum **System** bequem sortieren.

...

```
Random zzg = new Random();
```

...

```
temp = zzg.Next(49)+1;
```

...



DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Lösung



Lottozahlen: Übung

Erstellen Sie eine Klasse Cdraw (Ziehung) mit einer Methode Run(), die die Arbeit von Main() übernimmt.

Rufen Sie die Methode Run() 10x aus Main() auf, jeweils gefolgt von *Thread.Sleep(10)* .



DHBW
Duale Hochschule

Lottozahlen: Lösung/ Übung



DHBW
Duale Hochschule

Lottozahlen: Lösung/ Übung



Erstellen Sie ein Programm zum Berechnen einer persönlichen Glückszahl (zwischen 1 und 100), indem Sie:

- den Vor- und Nachnamen als Befehlszeilenargumente einlesen,
- den ersten Buchstaben des Vornamens sowie den letzten Buchstaben des Nachnamens ermitteln (beide in Großschreibung),
- die Nummern der beiden Buchstaben im Unicode-Zeichensatz bestimmen,
- die beiden Buchstabennummern addieren und die Summe als Initialisierungswert für den Pseudozufallszahlengenerator aus der Klasse **Random** verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als zwei Befehlszeilenargumente übergeben wurden.

Tipps:

- Um die durch Leerzeichen getrennten Befehlszeilenargumente im Programm als **String**-Array verfügbar zu haben, definiert man im Kopf der **Main()** - Methode einen Parameter vom Typ **String[]**:

```
static void Main(string[] args) {...}
```

- Wie jede andere Methode kann auch **Main()** per **return**-Anweisung spontan beendet werden.



DHBW

Duale Hochschule
Baden-Württemberg
Heidenheim

Lösung