Zusammenfassung: Programmieren 2 (C#)

Arrays

Eindimensional

```
int[] feld;
feld = new int[2];
int[] feld2 = new int[3];
int[] feld3 = new int[] { 1, 2, 3 };
//oder
int[] feld4 = { 1, 2, 3 };
int laenge = feld3.Length;
```

Mehrdimensional

```
int[][] matrix = new int[2][];
//Ausgefranst
matrix[0] = new int[2];
matrix[1] = new int[4];
matrix[0][1] = 33;
Console.WriteLine(matrix[0].Length); //Ausgabe: 2
int[,] tabelle = new int[2, 3]; //Rechteckig, 2 Zeilen und 3 Spalten
tabelle[1, 2] = 23;
Console.WriteLine(tabelle.GetLength(1)); //Ausgabe: 3 (Anzahl Spalten)
Console.WriteLine(tabelle.Length); //Ausgabe: 6 (2*3)
```

File IO

```
var inputFile = new FileStream(
    "/etc/pacman.conf", FileMode.Open, FileAccess.Read);
var outputFile = new FileStream (
    "./pacman.conf.bak", FileMode.Create);
var reader = new StreamReader(inputFile);
var writer = new StreamWriter(outputFile);
string line;
while((line = reader.ReadLine()) != null) {
    writer.WriteLine (line);
}
inputFile.Close ();
outputFile.Close ();
```

Template-Syntax

Für Klassen	`class Foo<T> {}`
Für einzelne Methoden	`T[] Foo<T>(T[] array,T val) {}`
Methoden voraussetzen (Klasse)	`class Foo<T> : Base, IForClass where T : IForT {}`
Methoden voraussetzen (einzelne Methode)	`Foo<T>(T[] array, T val) where T : IForT {}`
Konstruktor voraussetzen	`class Bar<T> where T : new() {}`
Arrays als Typ	`static public void SwapElements<T> (T[] array, int i1, int i2) {...}`

Delegates, Lambdaausdrücke und Events

Deklaration von Delegatentyp	`delegate bool Foo(int bar);`
Notation eines konkreten Delegates	`Foo del = delegate(int bar) { return bar == 42; };`
Notation mit Lambda-Ausdruck	`Foo del = (int bar) => { return bar == 42; };`
Notation mit Lambda-Ausdruck (einzelner Ausdruck)	`Foo del = (int bar) => bar == 42;`
Deklaration von Event	`event Foo MyEvent;`

Überschreiben vs. Überdecken

Methode überschreibbar machen	`virtual void Foo() {}`
Methode überschreibbar machen (komplett abstrakt)	`virtual void Foo();`
Methode überschreiben	`override void Foo();`
explizites Verdecken	`new void Foo();`
explizites Verdecken und neue Methode überschreibbar machen	`new virtual void Foo();`
abstrakte Methode deklarieren	`public abstract void F();`
abstrakte Methode implementieren (braucht kein override!)	`public void F() { ... }`

Erweiterungsmethoden

```
static class MainClass { public static void Foo(this ToBeExtended t) {}}
```

- Die Priorität von normalen Methoden ist höher als die von Erweiterungsmethoden.
- Die einschließende Klasse muss **static** sein.

Operatoren überladen

```
public static Vektor operator + (Vektor v_a, Vektor v_b) {
    return new Vektor(v_a.x + v_b.x, v_a.y + v_b.y);
}
public static implicit operator double(Vektor v) {
    return Math.Sqrt(v.x*v.x+v.y*v.y);
}
public static bool operator true (Vektor v) {
    return !v.IsNull();
}
```

- Überladen werden können:
  +, -, !, ~, ++, --, true, false, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=
- Operatoren für implizite und explizite Konvertierungen können definiert werden (zweites Beispiel).
- Vergleichsoperatoren == und != müssen paarweise überladen werden.
- Mindestens einer der Operanden muss vom eigenen Typen sein.
- Wenn Operator wie + überladen wird, wird auch entsprechender Operator **automatisch** += überladen.

Indexer

```
public int this[int i] {
    get { return zahlen[i]; }
    set { zahlen[i] = value; }
}
```

Enumeratoren

Interfacedefinitionen

```
namespace System.Collections {
    public interface IEnumerator
    {
        object Current {
            get;
        }
        bool MoveNext ();
        void Reset ();
    }
    public interface IEnumerable
    {
        IEnumerator GetEnumerator ();
    }
}
```

Beispiel für einfach verkettete Liste und Enumeratoren

```
class ForwardList<T>: System.Collections.Generic.IEnumerable<T>
{
    // Inneres Listenelement
    class ListElement<T>
    {
    public ListElement<T> Next;
        public T Value;

        public ListElement (T value, ListElement<T> next)
        {
            Next = next;
            Value = value;
        }
    }

    ListElement<T> m_first;
    ListElement<T> m_last;

    //Beispiele für Einfügen
    public void PushFront (T value)
    {
        var newElement = new ListElement<T> (value, m_first);
        m_first = newElement;
        if (m_last == null) {
            m_last = newElement;
        }
    }

    public void PushBack (T value)
    {
        var newElement = new ListElement<T> (value, null);
        if (m_last != null) {
            m_last.Next = newElement;
        }
        m_last = newElement;
        if (m_first == null) {
            m_first = newElement;
        }
    }
    //Beispiel für Traversierung.
    public void PrintAll ()
    {
        for (var current = m_first; current != null; current = current.Next) {
            Console.WriteLine (current.Value);
        }
    }

    // Element an bestimmter Position einfügen
    public void Insert (int pos, T value)
    {
        //Sonderfall index 0
        if (pos == 0) {
            m_first = new ListElement<T> (value, m_first);
            return;
        }
```

```
        int i = 1;
        var current = m_first;
        while (current != null) {
            if (pos == i) {
                var newElement = new ListElement<T> (value, current.Next);
                if (current.Next == null)
                    m_last = newElement;
                current.Next = newElement;
                return;
            }
            current = current.Next;
            ++i;
        }
        throw new IndexOutOfRangeException ();
    }

    // Element an bestimmter Position löschen
    public void Remove (int pos)
    {
        //Sonderfall index 0
        if (pos == 0) {
            m_first = m_first.Next;
            return;
        }
        int i = 1;
        ListElement<T> current = m_first;
        // Achtung: Andere Abbruchbedingung als bei InsertAt
        while (current.Next != null) {
            if (pos == i) {
                if (current.Next.Next == null) {
                    current.Next = null;
                    m_last = current;
                } else {
                    current.Next = current.Next.Next;
                }
                return;
            }
            current = current.Next;
            ++i;
        }
        throw new IndexOutOfRangeException ();
    }

    // Implementierung von IEnumerator mit yield return.
    public System.Collections.Generic.IEnumerator<T> GetEnumerator ()
    {
        for (var current = m_first; current != null; current = current.Next) {
            yield return current.Value;
        }
    }
    // Implementierung von nicht-generischem IEnumerator
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator ()
    {
        return this.GetEnumerator ();
    }
}
```