Einführung in die Programmiersprache C#

II

C# Übersicht

- Classen, Interfaces
- Ausnahmen
- Events & Lambda
- Generics
- Collections

Vererbung

- Syntax :

```
class Person{ ... }

class Student : Person { ... }
```

- Abstrakte Klassen/Methoden: abstract

```
abstract class Node{

    public abstract int CompareTo(Node e);

}
```

- Interfaces: interface

```
public interface IComparable{ int CompareTo(Object o); }
```

Vererbung

- class vs interface
- eine Klasse kann nur von einer Basisklasse erben, aber mehrere Interfaces implementieren

```
class A { ... }

interface B{...}

interface C : B {...} //interface inheritance

interface CC {...}

class SA : A, B, CC  { ... }

// class inheritance, interface
```

Virtuelle Methoden

- Schlüsselwort `virtual` zur Kennzeichnung von virtuellen Methoden
- virtuelle Methoden können überschrieben werden
- Methoden, Properties, Indexers, Events können virtual sein

```
public class Person{
        private String address;
        public virtual String getAddress(){...};
}
public class Student:Person{
        public override String getAddress(){...}
}
```

base

- man kann mit `base` den Konstruktor der Basisklasse aufrufen

```
class SubClass : BaseClass{
   SubClass(int id) : base (){
      //explicitly call the default constructor
      //...
   }
   SubClass(String n, int id) : base(n){
      //...
   }
}
```

sealed

- eine mit `sealed` markierte Klasse kann nicht geerbt werden
- für Methoden: können nicht von einer Unterklasse überschrieben werden

```
sealed class DD{ ... }

class AA { public virtual int f(){...} }

class BB: AA{ public sealed override int f(){...} }


class CC: BB{

public override int f(){...} // compile time error

}
```

static

- eine statische Klasse besitzt nur statische Elemente
- kann nicht geerbt werden

```
static class EncodingUtils{
     public static String encode(String txt){...}
     public static String decode(String txt){...}
}
```

static

- der statische Konstruktor wird einmal aufgerufen (nicht einmal pro Instanz)
- eine Klasse kann nur einen statischen Konstruktor definieren
- wird ausgeführt, bevor Instanzen des Typs erstellt werden
  - und bevor andere statische Elemente zugegriffen werden
- hat keine Parameter und denselben Namen wie die Klasse
- kann nur auf die statischen Attribute der Klasse zugreifen

partial-Klassen

- die Definition der Klasse ist auf mehreren Dateien aufgeteilt
- nützlich wenn die Klasse auto-generated wurde
  - und danach weitere Methoden manuell hinzugefügt sind
- jede Definition muss `partial` haben
- jede Definition kann keine widersprüchlichen Elemente haben

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

// PaymentForm.cs – manually added
partial class PaymentForm { ... }
```

partial-Methoden

- die Signaturen in beiden Teilen des partiellen Typs müssen übereinstimmen
- Die Methode muss `void` zurückgeben
- Partielle Methoden sind implizit `privat`

```
partial class PaymentForm {
    partial void ValidatePayment(decimal amount);}
partial class PaymentForm {
    partial void ValidatePayment(decimal amount) {
        if (amount > 100) …
    }
}
```

as - und is-Operatoren

- is-Operator: prüft, ob der Laufzeittyp eines Ausdrucks mit einem angegebenen Typ kompatibel ist
- as-Operator: konvertiert einen Ausdruck explizit in einen angegebenen Typ, wenn der Laufzeittyp mit diesem Typ kompatibel ist

```
Person pers=new Person();

Student st= pers as Student();    //st is null

Person pers1=new Student();

Student st1=pers1 as Student();   //st is not null

if ( pers1 is Student){ ... }
```

Interfaces

```
public interface ILog {
   void write(String mess);
}

public interface IFile {
   void write(String mess);
}

public class MyFile : ILog, IFile{
   public void write(String mess) {
      Console.WriteLine(mess);
   }
}

 class Program {
    static void Main(string[] args) {
       MyFile mf = new MyFile();
       IFile mff = mf;
       ILog mfl = mf;

       mf.write("ana"); //ana
       mff.write("ana"); //ana
       mfl.write("ana"); //ana
    }
 }
```

```
public interface ILog{
    void Write(String mess);
}

public interface IFile{
    void Write(String mess);
}

public class MyFile:ILog, IFile {
    void IFile.Write(String mess){
        Console.WriteLine(mess);
    }

    void ILog.Write(String mess){
        Console.WriteLine("ILog: {0}", mess);
    }
}
class Program {
   static void Main(string[] args) {
        ILog ilog=new MyFile();
        IFile ifile=new MyFile();
        ifile.Write("ana"); //ana
        ilog.Write("ana"); //ILog: ana
   }
}
```

Java


```java
interface A {
    void f();
}

interface B {
    void f();
}

class C implements A, B {
    public void f() {System.out.println("C::F()"); }
}

class Main {
    public static void main(String args[]) {
        C c = new C();
        c.f();
    }
}
```

Java


```java
interface A {
    default void f() {System.out.println("A::F()"); }
}

interface B {
    default void f() {System.out.println("B::F()"); }
}

class C implements A, B {
    public void f() {
        System.out.println("C::F()");
        A.super.f();
        B.super.f();
    }
}

class Main {
    public static void main(String args[]) {
        C c = new C();
        c.f();
    }
}
```

struct

- man kann keinen parameterlosen Konstruktor deklarieren
  - jede struct stellt bereits einen impliziten parameterlosen Konstruktor bereit
- der Konstruktor muss alle Felder initialisieren
- kann nicht von einer anderen Klasse oder einem anderen Strukturtyp erben, und er kann nicht die Basis einer Klasse sein
- kann Schnittstellen implementieren
- man kann keinen Finalizer bzw. Destruktoren deklarieren

struct

```
public struct Point{
    int x = 1;
    int y;

    public Point( ) {}  //error
    public Point(int x) {this.x = x;} //error
}
```

Implizit typisierte lokale Variablen

- `var` = der Compiler den angemessensten Typen bestimmt und zuweist

```
var x = 5;

x = "hello";    // Kompilierzeitfehler; x int
```

- kann zu niedriger Lesbarkeit führen

```
Random r = new Random(); var x = r.Next(); //int
```

- var kann nicht für Felder im Klassenbereich verwendet werden
- Variablen, die mit var deklariert wurden, können nicht im Initialisierungsausdruck verwendet werden

```
(line = reader.ReadLine()) != null

var i = (i = 20); //Kompilierzeitfehler
```

using-Anweisung

- using als Alias für einem Typ in einem Namespace
- `object` → `System.Object`
- `string` → `System.String`

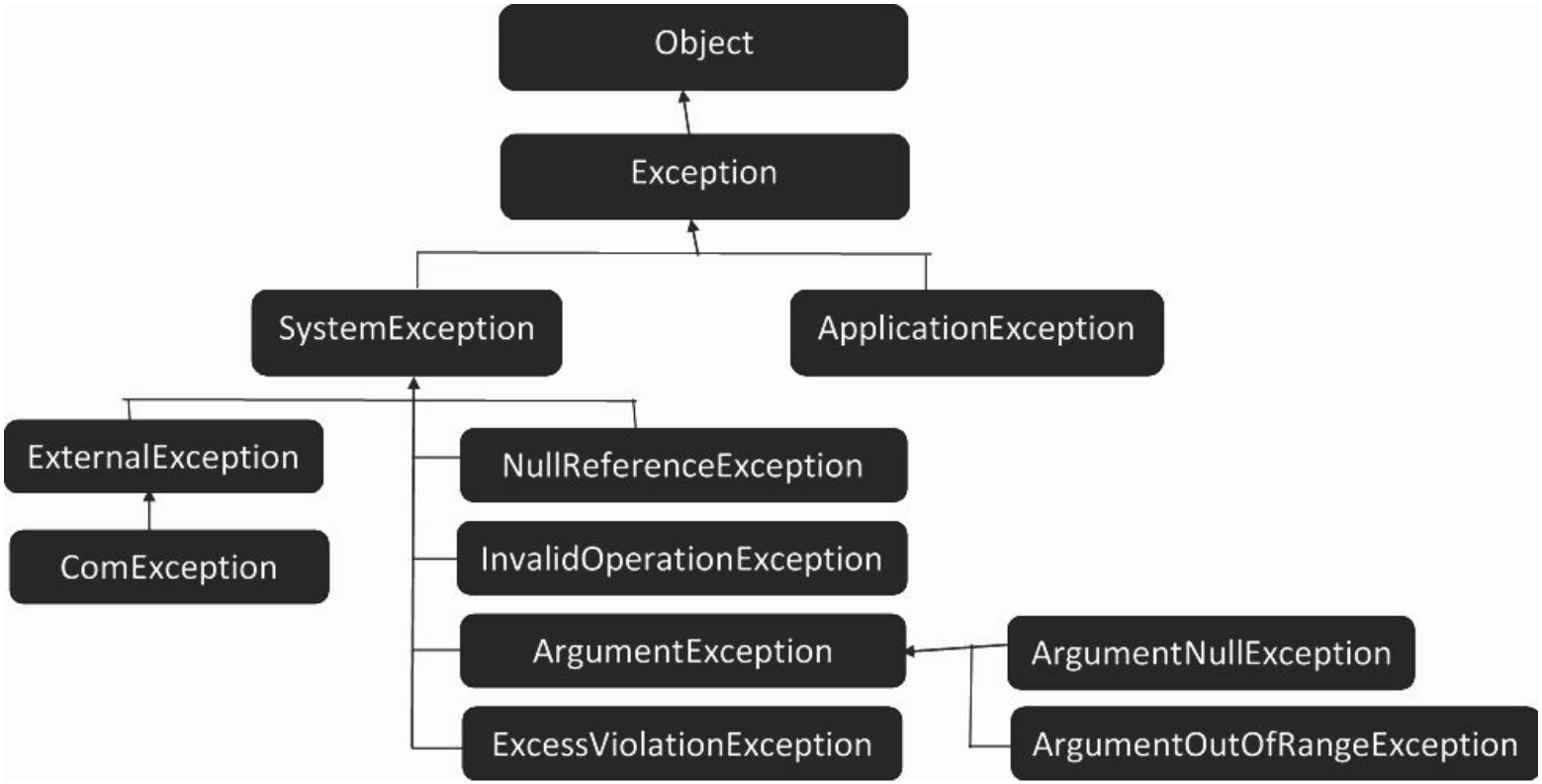
```
using person = entities.Person; //alias type
using gen=System.Collections.Generic; //alias namespace
class Test{
 void Main(){
      person p = new person();
      gen.List<int> list = new gen.List<int>(){1,2,3};
 }}
```

BCL (Base Class Library)

- System (basic language functionality, fundamental types)
- System.Collections (collections of data structures)
- System.IO (streams and files)
- System.Net (networking and sockets)
- System.Reflection (reflection)
- System.Security (cryptography and management of permissions)
- System.Threading (multithreading)
- System.Windows.Forms (GUI components, nonstandard, specific to the Windows platform)

Ausnahmen

- ungeprüfte Ausnahmen
- bei Ausnahmen handelt es sich um Typen, die alle letztlich von System.Exception abgeleitet werden



Ausnahmen

- erben von Exception

```
class StockException : ApplicationException{
  public StockException(){ }
  public StockException(String message):base(message){}
  public StockException(String msg, Exception exp)
         :base(msg,exp) { }
}
```

Ausnahmen

- ein try-Block erfordert einen oder mehrere zugeordnete catch-Blöcke oder ein finally-Block oder beide

```
try{
    int a=10, b=0;
    int d=a/b;
} catch(Exception e){
    Console.WriteLine("Exception "+ e);
}
```

Lambdaausdrücke

- (input-parameters) => expression
- (input-parameters) => { <sequence-of-statements> }
- anonyme Methode, die mit einem `Delegate` kompatibel ist
- kann in einen Delegattyp konvertiert werden
- Variablen, die in einem Lambdaausdruck eingeführt wurden, sind in der einschließenden Methode nicht sichtbar

Lambdaausdrücke

```
delegate int Transformer (int i);

Transformer sqr = x => x * x; //lambda expression

Console.WriteLine (sqr(3)); // 9
```

- ein Lambda, der beispielsweise zwei Parameter hat und keinen Wert zurückgibt, kann in einen `Action<T1,T2>`-Delegat konvertiert werden

```
Action<string> greet = name => Console.WriteLine(name);

greet("World");
```

- ein Lambda, der einen Parameter hat und einen Wert zurückgibt, in einen `Func<T,TResult>`-Delegat konvertiert werden

```
Func<int, int> square = x => x * x;

Console.WriteLine(square(5));
```

Lambdaausdrücke

- der Lambda muss dieselbe Anzahl von Parametern enthalten wie der Delegattyp
- der Rückgabewert (falls vorhanden) muss implizit in den Rückgabetyp des Delegaten konvertiert werden können
- jeder Eingabeparameter im Lambda muss implizit in den entsprechenden Delegatparameter konvertiert werden können
- eine return-Anweisung in einem Lambdaausdruck bewirkt keine Rückgabe durch die einschließende Methode

```
x => { return x * x; };
```

Lambdaausdrücke

- Lambda als Parameter

Lambdaausdrücke

```
public class IDOBject {
public int ID {get; set;}

public IDOBject(int id ) {ID = id;}
}

public class Student : IDOBject {
public int Age {get; set;}
public string Name {get;set;}

public Student (int id, string name, int age) : base(id)
{Age = age; Name=name;}

public override string ToString()
{
return $"Student(Id={ID}, Name={Name}, Age={Age})";
}
}
public delegate string Encode<T> (T obj) where T:
IDOBject;
```

```
public static void FilterStudents(List<Student> students,
Func<Student, bool> condition, Encode<Student> encode) {
students.Where(condition).ToList().ForEach(student =>
Console.WriteLine(encode(student)));
// students.Where(condition).ToList().ForEach(Console.WriteLine);
}
static string Encoder(Student st) {
return $"{st.Name} = {st.Age}";
}
```

```
List<Student> students = new List<Student> {
new Student(1, "Bob", 20),
new Student(2, "Dob", 21),
new Student(3, "Lob", 19)
};

FilterStudents(students, student => student.Age >= 20, student =>
Encoder(student));
```

Events

- Ereignisse (Events) sind eine Funktion, die das Publisher/Subscriber (Observer) -Muster formalisiert
- ein Wrapper für einen Delegaten, der nur innerhalb einer Klasse definiert und auslösbar ist
- der Hauptzweck von Ereignissen besteht darin, zu verhindern, dass sich Subscribers gegenseitig stören
- um zu definieren, wird das `Event` Keyword vor einem delegate Feld gestellt

Observer

- definiert eine 1:N-Abhängigkeit zwischen Objekten
  - damit alle "abhängigen" Objekte automatisch benachrichtigt und aktualisiert werden, wenn ein Objekt seinen Status ändert
- Publisher (Subject) - Subscriber (Observer) Beziehung
- der Publisher stellt die Daten bereit und benachrichtigt die Subscriber, die diese für diesen Publisher abonniert hat
- Beispiel: Repository (Seminar)

Events

- definiere einen Delegat

```
public delegate void DelegateEvent(Object sender, EventArgs args);
```

- definiere eine Klasse, die Events auslöst (Publisher)

```
class Publisher{
    public event DelegateEvent eventName;
        ...
        ... someMethod(...){
            //code that raises an event
            EventArgsSubClass args=new EventArgsSubClass(<some data>);
            eventName(this, args);
            //or eventName(this, null);
        }
    }
}
```

Events

- definiere Klassen, die Events behandeln (Subscriber)

```
class Observer{
    //the methods that matches the delegate signature
    public void OnEventName(Object sender, EventArgs args){
        //event handling code
    }
    ...
}
```

- alles zusammengestellt

```
class StartApp{
    ... Main(){
        //create the publisher(subject)
        Publisher pub=new Publisher(...);

        //create the observers
        Observer obs1=new Observer(...);
        Observer obs2=new Observer(...);

        //subscribe the observers to the event
        pub.eventName+=new DelegateEvent(obs1.OnEventName);
        pub.eventName+=new DelegateEvent(obs2.OnEventName);
        pub.someMethod(...);//explicit call of the method that raises the event
    }
}
```

Events

```
2 references
public delegate void TimerEvent(object sender, EventArgs args);
2 references
class ClockTimer{
    2 references
    public event TimerEvent timer;
    1 reference
    public void start(){
        for(int i=0;i<3;i++){
            timer(this, null);
            Thread.Sleep(1000);
        }
        }
}
0 references
class Test{
    0 references
    static void Main(){
        ClockTimer clockTimer=new ClockTimer();
        clockTimer.timer+=new TimerEvent(OnClockTick);
        clockTimer.start();
    }
    1 reference
    public static void OnClockTick(object sender, EventArgs args){
        Console.WriteLine("Received a clock tick event!");
    }
}
```

Generics

- man Verwendet Generics, um die Wiederverwendung von Code, Typsicherheit und Leistung zu maximieren
- generische und type-safe Containers
- Typsicherheit zur Kompilation ist durchgesetzt
- `List<String> safeBox = new List<String>();`
- Java: `public <T> T foo (T x);`
- C#: `public T foo <T> (T x);`
- Bounds
  - `public T test <T> (T x) where T : Interface/Class/Struct`

Generics

- Generics sind nativ durch .NET-Bytecode unterstützt (nicht wie Java)
- Daher sind alle Einschränkungen von Java-Generics nicht relevant
- man kann Generics mit Werttypen instanziieren
- zur Laufzeit kann man den Unterschied zwischen `List <Integer>` und `List <String>` erkennen
- Ausnahmeklassen können generisch sein

Wildcards

- der Mechanismus existiert in C# nicht

```
class Circle:Shape{...}

class Rectangle:Shape{...}
```

- Was sollte die Signatur einer Methode DrawShapes sein, die eine Liste von Shape-Objekten zeichnet?

```
DrawShapes(List<Shape> shapes)
```

- funktioniert nicht: `List<Circle>`
- `List<Circle>` keine Unterklasse von `List<Shape>`

Generics und Vererbung

- `S` ein Subtyp von `T`
- es gibt keine Beziehung zwischen

  `SomeGenericClass<S>` und `SomeGenericClass<T>`

- `S<AClass>` ist ein Subtyp von `T<AClass>`
- `AClass` ist nicht generisch

Wildcards

```
abstract class Shape {
    public abstract void draw();
}
class Circle: Shape{
    public override void draw() {
        Console.WriteLine("Drawing Circle...");
    }
}

class Rectangle: Shape{
    public override void draw() {
        Console.WriteLine("Drawing Rectangle...");
    }
}

class Program {
   static void DrawShapes <T> (List<T> shapes) where T: Shape{
       shapes.ForEach(s => s.draw());
   }
   static void Main(string[] args) {
       List<Circle> l = new List<Circle>(){new Circle(), new Circle()};
       DrawShapes(l);

   }
}
```

Generics

- `where G : new()`
- das Typargument muss einen öffentlichen, parameterlosen Konstruktor aufweisen

- `var t = default (T)`
- erzeugt den Standardwert für einen Typ
- `Console.WriteLine(default(int));  // output: 0`
- `Console.WriteLine(default(object) is null);  // output: True`

Generics

- man kann generischen Schnittstellen, Klassen, Methoden, Ereignisse und Delegaten erstellen
- Beispiele

```
public class IntStack : Stack<int> {...}

public class Customer<T> {
    private static List<T> customerList;
    private T customerInfo;
    public T CustomerInfo { get; set; }
    public int CompareCustomers( T customerInfo );
}

public interface IDrawable { public void Draw(); }

//contraint: type T implements the IDrawable interface
//no casting needed
public class SceneGraph<T> where T : IDrawable {
    public void Render() {
        ... T node; ...
        node.Draw();
    }
}
```

Generics

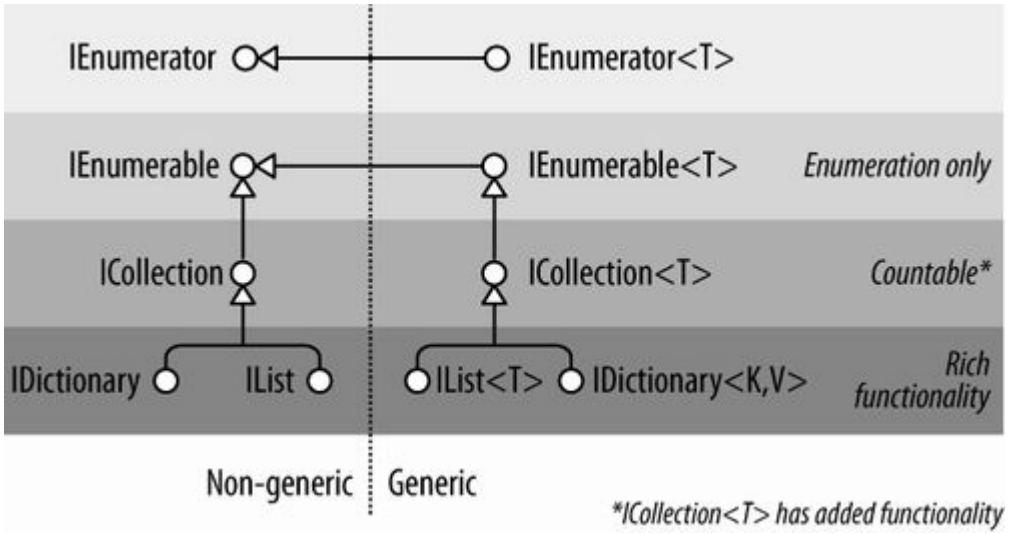
```
//The type argument must be a reference type
public class CarFactory<T> where T : class {...}

//Type T must provide a public parameter-less constructor
//No support for other constructors or other method syntaxes
//The new() constraint must be the last constraint
public class Stack<T> where T : new() {
    public T PopEmpty() {
        return new T();
    }
}

//can parameterize a method with generic types
public static void Swap<T>(ref T a, ref T b ){
    T temp = a;
    a = b;
    b = temp;
}
public class Report<T> : where T: IFormatter {...}
public class Insurance {
    public Report<T> ProduceReport<T>() where T : IFormatter {...}
}
```

Collections

- System.Collections
- System.Collections.Specialized
  - `ListDictionary`, `OrderedDictionary`, `BitVector`
- System.Collections.Generic
- System.Collections.Concurrent
  - `ConcurrentStack`



IList, IList<T>

```
public interface IList : ICollection, IEnumerable{
    object this [int index] { get; set }
    bool IsFixedSize { get; }
    bool IsReadOnly  { get; }
    int  Add      (object value);
    void Clear(  );
    bool Contains (object value);
    int  IndexOf  (object value);
    void Insert   (int index, object value);
    void Remove   (object value);
    void RemoveAt (int index);
}

public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}
```

IDictionary

```
public interface IDictionary : ICollection, IEnumerable{
    IDictionaryEnumerator GetEnumerator(  );
    bool Contains (object key);
    void Add      (object key, object value);
    void Remove   (object key);

    void Clear(  );
    object this [object key] { get; set; }

    bool IsFixedSize    { get; }
    bool IsReadOnly     { get; }
    ICollection Keys    { get; }
    ICollection Values  { get; }
}

public interface IDictionaryEnumerator : IEnumerator {
    DictionaryEntry Entry { get; }
    object Key { get; }
    object Value { get; }
}
```

IEnumerator, IEnumerable

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

//System.Collections.Generic
public interface IEnumerator<T> : IEnumerator, IDisposable{
    T Current { get; }
}

//System.Collections
public interface IEnumerable{
    IEnumerator GetEnumerator( );
}

//System.Collections.Generic
public interface IEnumerable<T> : IEnumerable{
    IEnumerator<T> GetEnumerator(  );
}
```

```
class Set : ISet, IEnumerable{
    object[] elems;
    public IEnumerator GetEnumerator() {...}
    //...
 }

 ...

 Set s=new Set();
 s.add(“ana”);
 s.add(“are”);
 s.add(“mere”);
foreach(Object o in s){
  Console.WriteLine(“{0} “,o);
 }
```

IComparable, IComparer

```
public interface IComparable {
 int CompareTo(object obj)
}

public interface IComparer {
 int Compare(object o1, object o2);
}

List.Sort()
List.Sort(IComparer cmp)
```

IComparable, IComparer

```
public class IDOBject {
public int ID {get; set;}

public IDOBject(int id ) {ID = id;}
}

public class Student : IDOBject {
public int Age {get; set;}
public string Name {get;set;}

public Student (int id, string name, int age) : base(id)
{Age = age; Name=name;}

public override string ToString()
{
return $"Student(Id={ID}, Name={Name}, Age={Age})";
}
}
```

```
class StudentComparer : IComparer<Student> {
public int Compare(Student s1, Student s2) {
return s1.Age - s2.Age;
}
}
```

```
List<Student> students = new List<Student> {
new Student(1, "Bob", 20),
new Student(2, "Dob", 21),
new Student(3, "Lob", 19)
};
```

```
students.Sort((s1,s2) => s1.Age - s2.Age);
students.ForEach(Console.WriteLine);
```

ICloneable

- Stellt einen Mechanismus für Cloning
- eine neue Instanz einer Klasse wird mit demselben Wert wie eine vorhandene Instanz erstellt
- bad idea?

```
public interface ICloneable {
    object Clone();
}
```

IDisposable

- stellt einen Mechanismus für die Freigabe nicht verwalteter Ressourcen bereit

- der Garbage Collector gibt den Arbeitsspeicher, der einem verwalteten Objekt zugeordnet ist, automatisch frei
  - wenn dieses Objekt nicht mehr verwendet wird

- ist jedoch nicht möglich, vorherzusagen, wann Garbage Collection auftreten werden

IDisposable

```
public interface IDisposable{ void Dispose(  ); }
using(Stream stream = File.Open(fileName, FileMode.Open))
    ...

Stream stream = File.Open(fileName, FileMode.Open)
try {
      ...
}
finally {
    if (stream != null) stream.Dispose();
}
```

IDisposable

```
public delegate string StringEncoder<T>(T o) where T : IDObject;

public interface Repository<T> : IObservable<T> where T : IDObject{
    void Add(T o);
    bool Remove(int id);
    T Find(int id);
    T[] GetAll();
    string ToString(StringEncoder<T> encoder);
    void Close();

    IDisposable Subscribe(IObserver<T> observer);
}


public class ConcreteRepository<T> : IDisposable, Repository<T>  where T : IDObject
{
    public void Dispose()
    {
        ....
    }
}
```