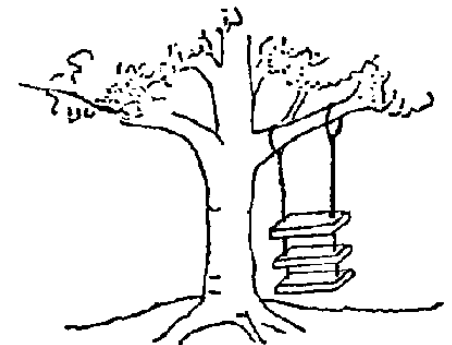
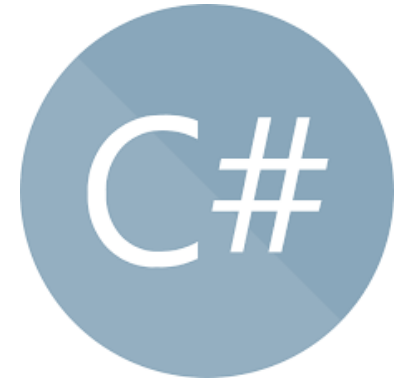


Abstract Class Interface Enum Exception

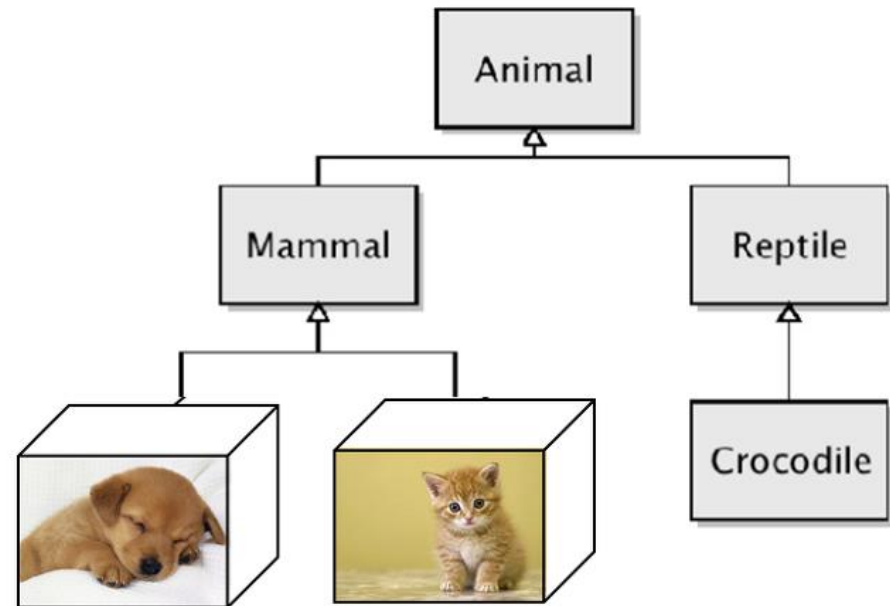
Software Entwicklung



Übersicht



- Abstrakte Klasse
 - Statische Bindung
 - Dynamische Bindung (Polymorphie)
- Interface
 - Interface `IMyPersonalInterface`
- Enum
 - Enum `EMyPersonalEnum`
- Exception
 - `throw new Exception("My personal Exception!")`
- Vererbung erzwingen oder unterbinden
 - Abstract Class & Abstract Method
 - Sealed Class & Sealed Method



Abstrakte Klassen

Dinge die es im echten Leben nicht gibt können als abstrakte Klassen implementiert werden:

Animal: Dog, Cat, Tiger, ...

Form: Circle, Triangle, Rectangle, ...

Abstrakte Klasse

- können nicht instantiiert werden
- stellen eine Basisklasse für weitere abgeleitete Klassen zur Verfügung
- stellen Attribute und Methoden für alle abgeleiteten Klassen zur Verfügung

```
public abstract class A
{
    // Class members here.
}
```

Abstrakte Klasse

- können auch abstrakte Elemente haben
- mit Hilfe des Schlüsselwort **abstract** vor dem Rückgabewert

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstrakte Klasse Tier:

- bei Abstrakten Klassen

wird ein A vor dem Klassennamen hinzugefügt:

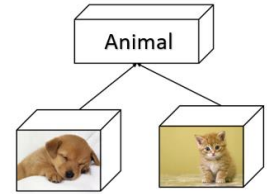
- AAnimal
- AForm
- ABuilding
- ...

```
public abstract class AAnimal
{
    private int age = 3;
    1-Verweis
    public int Age
    {
        get { return age; }
        set {
            if (value > 0)
                age = value;
        }
    }

    0 Verweise
    public void PrintAge() ...

    0 Verweise
    public virtual void Running() ...
}
```

Abstrakte Klasse Animal

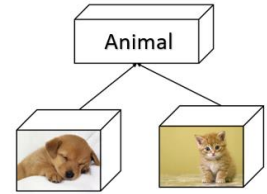


- Erstelle eine Klasse AAnimal
- Mit einem Property Name und einer Methode Sound
- Animal kann nicht instanziiert werden!

```
public abstract class AAnimal {
    public string Name { get; set; }
    public void Sound(string sound) {
        Console.WriteLine($"Das Tier {sound}");
    }
}
```

```
AAnimal a.. = new AAnimal();
```

Erstelle Elefant und Katze



- Erbe von der Klasse Animal.
- Erstelle Objekte der Klasse Elefant und Katze, setze Namen & nutze die Methode Sound

```

public abstract class AAnimal {
    public string Name { get; set; }
    public void Sound(string sound) {
        Console.WriteLine($"{Name} {sound}");
    }
}

```

```

public class Elefant : AAnimal { }
public class Cat : AAnimal { }

```

```

Dumbo tröröööt
Mia miaut

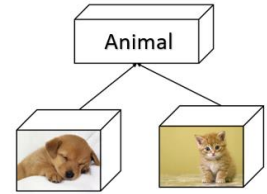
```

```

static void TestAnimal() {
    //AAnimal instanziiieren nicht möglich
    //AAnimal a = new AAnimal();
    Elefant e = new Elefant();
    e.Name = "Dumbo";
    e.Sound("tröröööt");
    Cat c = new Cat();
    c.Name = "Mia";
    c.Sound("miaut");
}

```


Array von Tieren



- Deklarierter Datentyp (Compilezeit) ist Animalarray
- Datentyp zur Laufzeit (Instanz der Klasse) ist eine davon abgeleitete Klasse
 - Methoden von Animal können aufgerufen werden
 - Abgeleitete Klasse stellt gleichnamige Methode neu implementiert zur Verfügung

Werden die Methoden
der Basisklasse (Compilezeit) oder
der abgeleiteten Klasse (Laufzeit)
verwendet?!



Methode der Basisklasse
wird auch in der abgeleiteten Klasse ausimplementiert...

Statische Bindung

Statische Bindung

Nicht Polymorphes Verhalten

Typ des Objekts zur Compilezeit

Schlüsselwort new

Überdecken

Statische Bindung

```
public abstract class AAnimal {  
    public string Name { get; set; }  
  
    public void Move() {  
        Console.WriteLine($"Das Tier bewegt sich ... ");  
    }  
}
```

- Erstelle eine Methode Move in AAnimal und in Elefant - es erscheint ein Hinweis...

```
public class Elefant : AAnimal {  
    public void Move() {  
        Console.WriteLine("Elefant bewegt sich ...");  
    }  
}
```

void Elefant.Move()

"Elefant.Move()" blendet den vererbten Member "AAnimal.Move()" aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Verwenden Sie das new-Schlüsselwort,
wenn das Ausblenden vorgesehen war.



Array von Tieren

```
public class Elefant : AAnimal {
    public new void Move() {
        Console.WriteLine($"Der Elefant {Name} spaziert");
    }
}
public class Cat : AAnimal {
    public new void Move() {
        Console.WriteLine($"Die Katze {base.Name} springt");
    }
}
public class Dog : AAnimal {
    public new void Move() {
        Console.WriteLine($"Der Hund {base.Name} rennt");
    }
}
public class Horse : AAnimal {
    public new void Move() {
        Console.WriteLine($"Das Pferd {base.Name} galoppiert");
    }
}
public class Mouse : AAnimal {
    public new void Move() {
        Console.WriteLine($"Die Maus {base.Name} eilt");
    }
}
```

```
public abstract class AAnimal {
    public string Name { get; set; }

    public void Move() {
        Console.WriteLine($"Das Tier bewegt sich ... ");
    }
}
```

```
static void TestAnimalArray() {
    AAnimal[] animals = new AAnimal[5];
    animals[0] = new Elefant() { Name = "Dumbo" };
    animals[1] = new Cat() { Name = "Minka" };
    animals[2] = new Dog() { Name = "Bello" };
    animals[3] = new Horse() { Name = "Pegasus" };
    animals[4] = new Mouse() { Name = "Cherry" };
    foreach (AAnimal a in animals) {
        a.Move();
    }
}
```

Wie lautet die Ausgabe?

Problematik erkennen



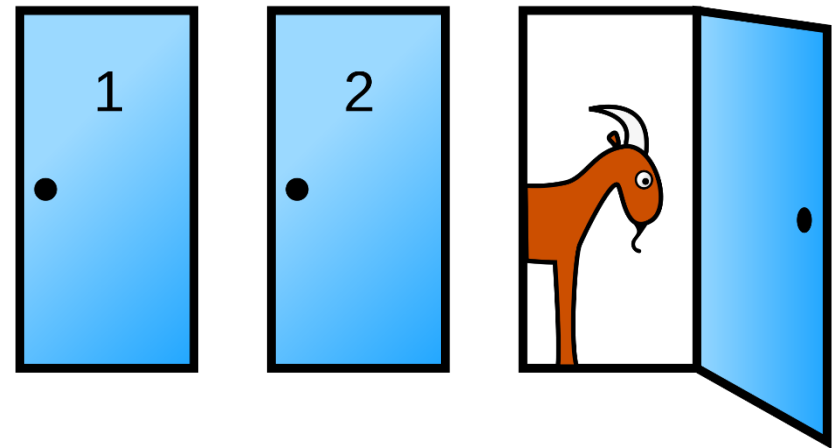
- Ist das die gewünschte Ausgabe?

```
static void TestConcreteInstance() {
    Elefant a1 = new Elefant() { Name = "Dumbo" };
    Cat a2 = new Cat() { Name = "Minka" };
    Dog a3 = new Dog() { Name = "Bello" };
    Horse a4 = new Horse() { Name = "Pegasus" };
    Mouse a5 = new Mouse() { Name = "Cherry" };
    a1.Move();
    a2.Move();
    a3.Move();
    a4.Move();
    a5.Move();
}
```

```
Der Elefant Dumbo spaziert
Die Katze Minka springt
Der Hund Bello rennt
Das Pferd Pegasus galoppiert
Die Maus Cherry eilt
```

```
static void TestAnimalArray() {
    AAnimal[] animals = new AAnimal[5];
    animals[0] = new Elefant() { Name = "Dumbo" };
    animals[1] = new Cat() { Name = "Minka" };
    animals[2] = new Dog() { Name = "Bello" };
    animals[3] = new Horse() { Name = "Pegasus" };
    animals[4] = new Mouse() { Name = "Cherry" };
    foreach (AAnimal a in animals) {
        a.Move();
    }
}
```

```
Das Tier bewegt sich ...
Das Tier bewegt sich ...
Das Tier bewegt sich ...
Das Tier bewegt sich ...
Das Tier bewegt sich ...
```



Vielgestaltigkeit

Polymorphie

Polymorphes Verhalten

Dynamische Bindung

Typ des Objekts zur Laufzeit

Überschreiben einer Methode

Schlüsselwörter virtual & override

Überschreiben

```
public class Elefant : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Der Elefant {Name} tröttet");
    }
}

public class Cat : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Die Katze {base.Name} miaut");
    }
}

public class Dog : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Der Hund {base.Name} bellt");
    }
}

public class Horse : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Das Pferd {base.Name} wiehert");
    }
}

public class Mouse : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Die Maus {base.Name} piepst");
    }
}
```

```
public abstract class AAnimal {
    public string Name { get; set; }

    public void Sound(string sound) {
        Console.WriteLine($"{{Name}} {{sound}}");
    }

    public virtual void Sound() {
        Console.WriteLine($"Das Tier macht ein Geräusch ... ");
    }

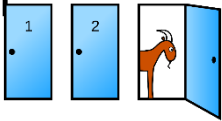
    public void Move() {
        Console.WriteLine($"Das Tier bewegt sich ... ");
    }
}

static void TestOverrideAnimalArray() {
    AAnimal[] animals = new AAnimal[5];
    animals[0] = new Elefant() { Name = "Dumbo" };
    animals[1] = new Cat() { Name = "Minka" };
    animals[2] = new Dog() { Name = "Bello" };
    animals[3] = new Horse() { Name = "Pegasus" };
    animals[4] = new Mouse() { Name = "Cherry" };
    foreach (AAnimal a in animals) {
        a.Sound();
    }
}
```


Wie lautet die Ausgabe?

Überschreiben vs Überdecken

Überschreiben

- Polymorphes Verhalten
 - Dynamische Bindung
 - Typ des Objekts zur Laufzeit
 - virtual & override
 - Überschreiben
- 
- Sinnvoll zu nutzen, man verwendet die Methoden der Instanz einer Klasse, nicht die Methoden laut Deklaration der Variable

Überdecken

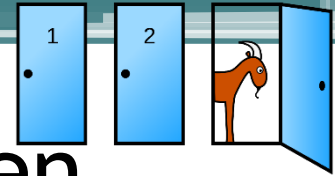
- Nicht Polymorphes Verhalten
 - Statische Bindung
 - Typ des Objekts zur Compilezeit
 - Schlüsselwort new
 - Überdecken
- 
- Einsatzgebiet fraglich

Überladen

- Methode Sound zwei mal innerhalb der Klasse
 - mit unterschiedlicher Parameterliste

```
public class Animal {  
    public string name;  
  
    public void Sound() {  
        Console.WriteLine($"Das Tier macht ein Geräusch ... ");  
    }  
  
    public void Sound(string sound) {  
        Console.WriteLine($"{name} {sound}");  
    }  
}
```

Methode überladen vs überschreiben



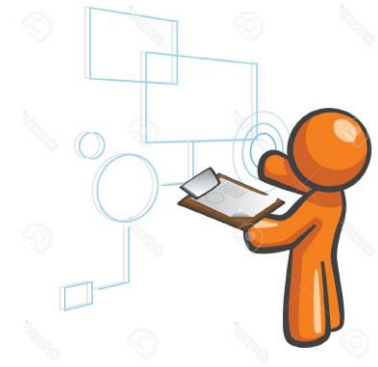
```
public abstract class AAnimal {
    public string Name { get; set; }

    public void Sound(string sound) {
        Console.WriteLine($"{Name} {sound}");
    }
    public virtual void Sound() {
        Console.WriteLine($"Das Tier macht ein Geräusch ... ");
    }
}
```

Sound wird in AAnimal
überladen
(2 Methoden gleichen Namens
unterschiedliche Parameterliste)

```
public class Elefant : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Der Elefant {Name} tröttet");
    }
}
public class Cat : AAnimal {
    public override void Sound() {
        Console.WriteLine($"Die Katze {base.Name} miaut");
    }
}
```

Sound wird in Elefant
und Cat **überschrieben**
Schlüsselwörter virtual
und override



Interface

Schnittstellen für Zusammenarbeit festlegen...

Beginnt laut Namenskonvention mit dem Buchstaben I
IMyInterface, ICountable, IMoveable, ...

Interface Erklärung & Syntax

- beinhaltet die Deklaration von Methoden, Properties und Events
- enthält keine Implementierung
- Eine Klasse, die dieses Interface implementiert, ist dafür verantwortlich die Implementierung der Methoden zur Verfügung zu stellen

```
interface InterfaceName {  
    Rückgabewert Methodename1();  
    Rückgabewert Methodename2();  
    Rückgabewert Methodename3();  
    ...  
}
```

Motivation

- Sicherstellen, dass eine gewisse Funktionalität in einer Klasse verfügbar ist
 - Eine Klasse besitzt eine `Move()` -> `IMoveabel`
- Arbeiten im Projektteam:
 - Klassendiagramm erstellen
 - Schnittstellen festlegen
 - Gleichzeitig zu programmieren beginnen

Bsp

- Programmierer 1: Implementierung der Logik
- Programmierer 2: Benutzeroberfläche, nutzt die Logik, die nur als Interface (Methodenköpfe) ohne Implementierung deren Verfügbar ist

Interface vs Abstrakte Klasse

- Bei einer Abstrakten Klasse wird Generalisiert:
 - Gemeinsamkeiten von LKW, Fahrrad, Auto ergibt eine Klasse Fahrzeug, alle 3 haben eine Marke und ein Baujahr alle 3 können “Fahren”
- Interface erfasst Funktionalitäten
 - Fahrzeuge können sich Fortbewegen -> Move
 - Person kann sich Fortbewegen -> Move

Syntax eines Interfaces

- Nutze das Schlüsselwort `interface` wie folgt:

```
interface Bezeichner {  
    Method_1 ();  
    Method_2 ();  
    ...  
    Method_n ();  
}
```

Implementiere ein Interface

```
class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

```
interface ISampleInterface
{
    void SampleMethod();
}
```


Klasse implementiert ein Interface

- Klasse Person implementiert das Interface IMoveable mit der Methode Move

```
public interface IMovable
{
    void Move();
}
```

- Die Klasse Auto und Zug kann sich auch bewegen.
- Erstelle eine Klasse Gott, welche Objekte bewegt.

```
public interface IMovable
{
    void Move();
}
public class Human : IMovable
{
    public void Move()
    {
        Console.WriteLine("Der Mensch macht einen Schritt vor.");
    }
}
public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Das Auto fährt los.");
    }
}
public class Train : IMovable
{
    public void Move()
    {
        Console.WriteLine("Der Zug fährt los.");
    }
}
```

IMoveable implementieren

Bewegliche Objekte bewegen

- Klasse Gott, bewegt Objekte:

```
public class God
{
    public void MoveObject(IMovable item)
    {
        Console.WriteLine("Ein bewegliches Objekt wird bewegt...");
        item.Move();
    }
}
```

Main Methode

```
static void Main(string[] args)
{
    Car BMW = new Car();
    Train ICE = new Train();
    Human Robert = new Human();

    God god = new God();
    god.MoveObject(BMW);
    god.MoveObject(ICE);
    god.MoveObject(Robert);

    Console.ReadLine();
}
```

Ausgabe:

Ein bewegliches Objekt wird bewegt...

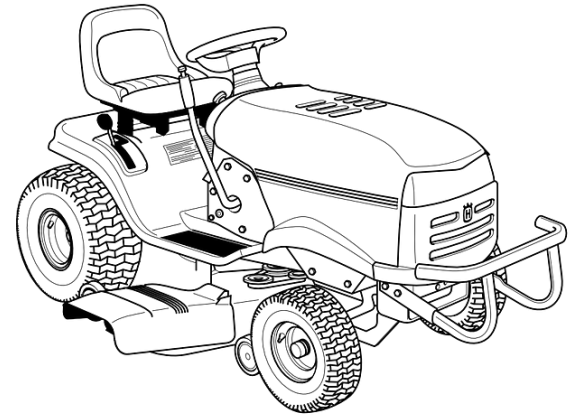
Das Auto fährt los.

Ein bewegliches Objekt wird bewegt...

Der Zug fährt los.

Ein bewegliches Objekt wird bewegt...

Der Mensch macht einen Schritt vor.



Angabe Driveable:

Erstelle eine Klasse Fahrzeug, welche ein IDriveable Interface mit einer Drive Methode implementiert.

Füge selbstständig 2 weitere Klassen hinzu, welche ebenfalls eine Drive Methode verfügbar haben.

Erstelle ein Array mit IDriveable-Objekten und lasse alle fahren!

Main & Driveable

```
IDriveable[] cars = new IDriveable[4];  
cars[0] = new SportCar();  
cars[1] = new Car();  
cars[2] = new Car();  
cars[3] = new Vehicle();
```

```
foreach (IDriveable car in cars)  
{  
    car.Drive();  
}
```

```
public interface IDriveable  
{  
    void Drive();  
}  
class Vehicle : IDriveable  
{  
    public void Drive()  
    {  
        Console.WriteLine("Das Fahrzeug fährt");  
    }  
}
```



Properties & Interfaces

- Interface ILebewesen
- Klasse Mensch

```
private interface ILebewesen
{
    string Name {get; set;}
    string Typ { get; }
    string Eigenschaft { get; }
}
```

```
private class Mensch : ILebewesen
{
}
```

Lösung

- Implementiere das Interface ILebewesen (IBeing) in der Klasse Mensch

```
private class Mensch : ILebewesen
{
    public Mensch(string name)
    {
        this.Name = name;
    }

    public string Name{ get; set; }

    public string Typ
    {
        get { return "Mensch"; }
    }

    public string Eigenschaft
    {
        get { return "redet"; }
    }
}
```


Auch Hunde und Katzen ...

```
private class Katze : ILebewesen
{
    public Katze(string name)
    {
        this.Name = name;
    }

    public string Name { get; set; }

    public string Typ
    {
        get { return "Katze"; }
    }

    public string Eigenschaft
    {
        get { return "miaut"; }
    }
}
```

```
private class Hund : ILebewesen
{
    public Hund(string name)
    {
        this.Name = name;
    }

    public string Name { get; set; }

    public string Typ
    {
        get { return "Hund"; }
    }

    public string Eigenschaft
    {
        get { return "bellt"; }
    }
}
```

Vorteil von IBeing

- Können in einem Array oder einer Liste genutzt werden:

```
ILebewesen lebewesen = ILebewesen[5];  
lebewesen[0] = new Hund("Bello");
```

Oder mit Listen:

```
List<ILebewesen> lebewesen = new List<ILebewesen>();  
lebewesen.Add(new Hund("Bello"));  
lebewesen.Add(new Mensch("Peter"));  
lebewesen.Add(new Katze("Lady"));  
lebewesen.Add(new Hund("Kira"));  
lebewesen.Add(new Hund("Bonka"));  
  
foreach (ILebewesen wesen in lebewesen)  
{  
    Console.Write( "Der/Die " + wesen.Typ + " " + wesen.Eigenschaft + " und heisst " + wesen.Name);  
}
```

Unterschied von Array und Liste

- Array besitzt eine fixe Größe
 - Zugriff mit Length auf die Größe des Arrays
 - Mit dem Indexer [] kann auf Elemente zugegriffen werden
- Liste kann eine beliebige Anzahl von Elementen beinhalten
 - Zugriff mit Count auf die Anzahl der Elemente
 - Stellt Methoden wie: Add, Find, IndexOf, Remove,... zur Verfügung

Abstract Class & Interface

- Erbt von einer Basisklasse:

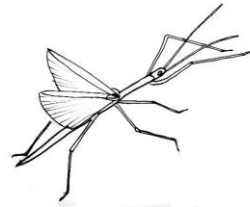
```
abstract class AFoo
{
}
```

- Implementiert ein Interface:

```
interface IFoobar
{
    void Foo();
    void Bar();
}
```

```
class Foobar : AFoo, IFoobar
{
    public void Bar()
    {
        Console.WriteLine("Bar");
    }

    public void Foo()
    {
        Console.WriteLine("Foo");
    }
}
```



Angabe Abstrakte Klassen & Interface

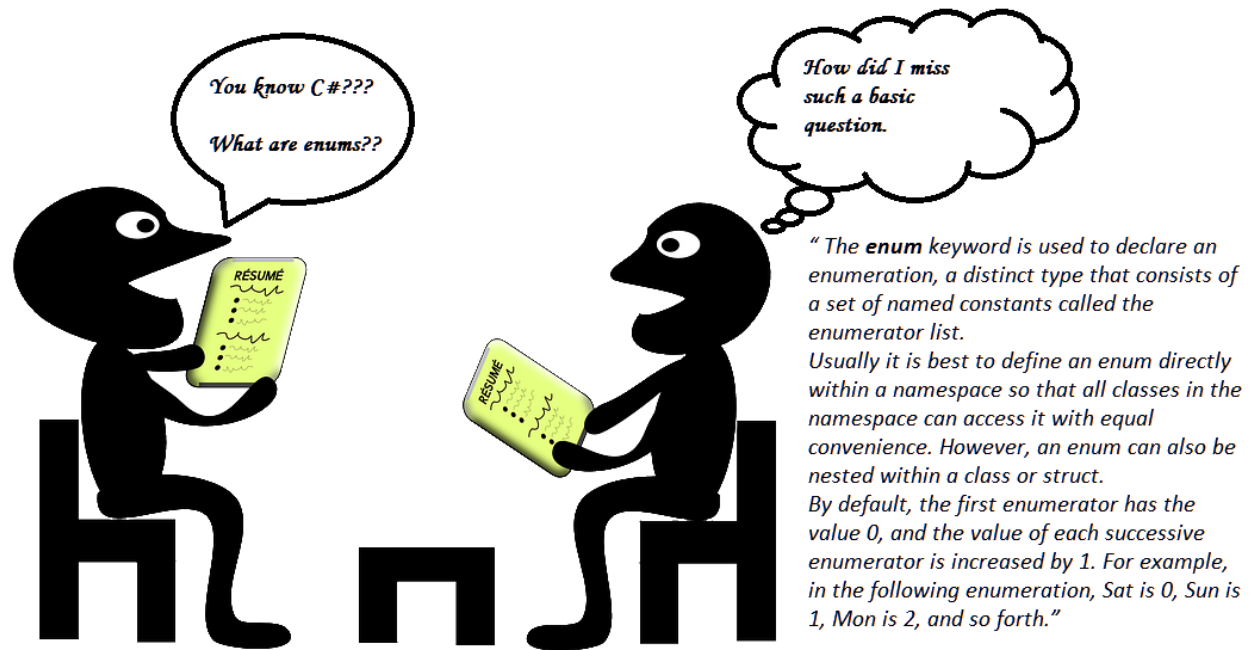
AAnimal & FlyingObject & IFlyAble

Luftfahrzeug

- Erstelle eine abstrakte Klasse FlugObjekt mit einer abstrakten Methode Start und einen Property für Marke
- Erstelle eine Klasse Hubschrauber und beerbe die Klasse FlugObjekt, überschreibe die Methode Start
- Erstelle ein Interface IFlyable mit einer Methode Fly
 - Implementiere das Interface in der Klasse FlyingObject

Tiere

- Erstelle eine Klasse AAnimal als abstrakte Basisklasse mit 2 Eigenschaften frei erfunden.
 - Erstelle eine abstrakte Methode (freier Wahl)
- Implementiere 2 abgeleitete Klassen
 - überschreibe die obige Methode
- Erstelle eine Klasse Vogel die von Tier erbt
- Implementiere das IFlyable Interface
- Teste mit Array & Listen von AAnimal, IFlyable & Flugobjekt



Enum

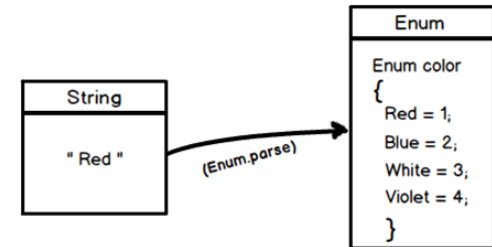
Aufzählungsdatentyp

Enumtypen beginnen mit E
Werte der Aufzählung sollten
in GROSZBUCHSTABEN geschrieben sein

Wozu Enum?

- Boolean kann 2 Werte abbilden, es sollen jedoch mehr als 2 Werte abgebildet werden.
- zB: Nord, Süd, Ost und West
 - Lösungsidee:
Verwendung von INT oder STRING?
- Aufzählungsdatentyp werden verwendet um mehrere Möglichkeiten von Werten in einem Datentyp zusammenzufassen

Enum der Aufzählungstyp



- Aufzählungen als Zahlen abzubilden:
 - NT: nicht gut lesbarer Quellcode
- Aufzählungen als Zeichenketten abzubilden:
 - NT: Fehleranfällig bei Tippfehlern
 - NT: Fehler werden erst zur Laufzeit oder überhaupt nicht gefunden -> logische Fehler
- Enum Kombiniert:
 - Intern Abbildung als Zahl
 - Genutzt als Zeichenkettenkonstante

Array von Wochentagen

```
enum EWeek { MON, TUE, WED, THU, FRI, SAT, SUN }
```

- Zähle alle Montage, welche in einem Array gespeichert sind...

```
//Lösung mit STRING-ARRAY
```

```
string[] sdays = { "Montag", "Dienstag", "Thursday",  
| "Saturday", "Mo", "MON", "samstag", "freitag", "Monday", "Dienstag", "Sunday" };
```

```
//Lösung mit INT-ARRAY
```

```
//Welcher Wochentag ist Montag? 1?!
```

```
//Welcher Wochentag ist 8?
```

```
int[] idays = { 1, 3, 4, 5, 6, 2, 1, 2, 7, 8, 2, 3, 2, 2, 4 };
```

```
//Lösung mit ENUM-ARRAY
```

```
EWeek[] eDays = { EWeek.FRI, EWeek.MON, EWeek.SAT, EWeek.THU, EWeek.FRI,  
| EWeek.THU, EWeek.MON, EWeek.SAT, EWeek.WED, EWeek.FRI, EWeek.TUE};
```

Lösung

- Ist die jeweilige Zählung korrekt?

```
int counts = 0, counti = 0, counte = 0;
foreach (var item in sdays) {
    Console.Write(item + ", ");
    if (item == "MON") counts++;
}
Console.WriteLine($"\\nMontage in Stringarray {counts}\\n");

foreach (var item in idays) {
    Console.Write(item + ", ");
    if (item == 1) counti++;
}
Console.WriteLine($"\\nMontage in Intarray {counti}\\n");

foreach (var item in eDays) {
    Console.Write(item + ", ");
    if (item == EWeek.MON) counte++;
}
Console.WriteLine($"\\nMontage in Enumarray {counte}");
```

```
Montag, Dienstag, Thursday, Saturday, Mo, MON, samstag, freitag, Monday, Dienstag, Sunday,
Montage in Stringarray 1
```

```
1, 3, 4, 5, 6, 2, 1, 2, 7, 8, 2, 3, 2, 2, 4,
Montage in Intarray 2
```

```
FRI, MON, SAT, THU, FRI, THU, MON, SAT, WED, FRI, TUE,
Montage in Enumarray 2
```

Syntax

```
enum <enum_name>
{
    enumeration list
};
```

enum EName { WERT1, WERT2, WERT3, ... }

- Ohne Wertzuweisung: (Zahlenwerte: 0,1,2,...)

```
enum ECardinalDirections { NORD, SOUTH, EAST, WEST}
enum ETemperature { LOW, MEDIUM, HIGH };
```

enum EName { WERT1[=Zahl], WERT2[=Zahl], ... }

- Mit Wertzuweisung: (Zahlenwerte: 1,2,3,...)

```
enum Day {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
enum Day {Sat=10, Sun=20, Mon=30, Tue=40, Wed=50, Thu=60, Fri=70};
```

Beispiele für Aufzählungen

- Es sollen die Wochentage gespeichert werden können:
 - Mo, Di, Mi, Do, Fr, Sa, So
- Es soll die Wassertemperatur festgelegt werden können:
 - Warm, Kalt, Lauwarm
- Es sollen die Himmelsrichtungen abgebildet werden:
 - Süd, Ost, West, Nord

Beispiele

- Wie lautet die Ausgabe?

```
public class EnumTest
{
    enum Day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Day.Sun;
        int y = (int)Day.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

Ausgabe:

Sun = 0

Fri = 5

Enum für Wochentage

- Erstelle ein Enum für Wochentage
- `enum EWorkingDays{ MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY}`

```
enum WorkingDays
```

```
{
```

```
Monday,
```

```
Tue
```

```
Wednesday,
```

```
Thursday,
```

```
Friday
```

```
}
```

WorkingDays.Monday = 0

```
enum WorkingDays
```

```
{
```

```
Monday,
```

```
Tuesday,
```

```
Wednesday,
```

```
Thursday,
```

```
Friday
```

```
}
```

WorkingDays.Friday = 4

- Montag erhält als Index 0
- Freitag erhält als Index 4

Hinweis: korrekterweise ->
EWorkingDays und MONDAY, ...

Enum und spezifische Zahlenwerte

```
enum ETemperature { LOW, MEDIUM, HIGH };
```

```
public static void Test()  
{  
    ETemperature t = ETemperature.HIGH;  
}
```

- Verändere das Enum, sodass Low auf 15, Medium auf 21 und High auf 35 als Wert im Hintergrund gesetzt wird:

```
enum ETemperature { LOW=15, MEDIUM=21, HIGH=35 };
```

Enum und spezifische Zahlenwerte

- Enum als Zahl oder Enum als Zeichenkette nutzen:

```
enum ETemperature { LOW=15, MEDIUM=21, HIGH=35 };
```

```
public static void Test()
```

```
{
```

```
    ETemperature t = ETemperature.HIGH;
```

```
    int temp = (int)t;
```

```
    Console.WriteLine("Temperatur ist {0} mit {1} Grad", t, temp);
```

```
}
```

```
Temperatur ist HIGH mit 35 Grad
```

Erstelle 3 Enum für

- Warm, Kalt, Lauwarm
 - Süd, Ost, West, Nord
 - Mo, Di, Mi, Do, Fr, Sa, So
-
- `enum EWater {LOW=10, WARM=20, HOT=30}`
 - `enum EWorkingDays{ MONDAY=1, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY}`
 - `enum ECardinalDirections { NORD, SOUTH, EAST, WEST}`

Beispiel Importance

- Erstelle ein Enum:
Elmportance mit den Werten:
- None, Trivial, Regular, Important & Critical

```
enum Importance
{
    None,
    Trivial,
    Regular,
    Important,
    Critical
};

// ... An enum local variable.
Importance value = Importance.Critical;

// ... Test against known Importance values.
if (value == Importance.Trivial)
{
    Console.WriteLine("Not true");
}
else if (value == Importance.Critical)
{
    Console.WriteLine("True");
}
```

Enum mit spezifischen Zahlenwerten

```
enum EState { DRIVING, SPEEDING=4, PARKING };
```

- Welche Werte werden für Driving und Parking gesetzt?
 - Driving -> 0
 - Speeding -> 4
 - Parking -> 5

```
try{
```



Exceptions...

Gotta catch 'em all!

C++ Exception Handling

```
}catch( Exception ){  
    //Do nothing  
}
```

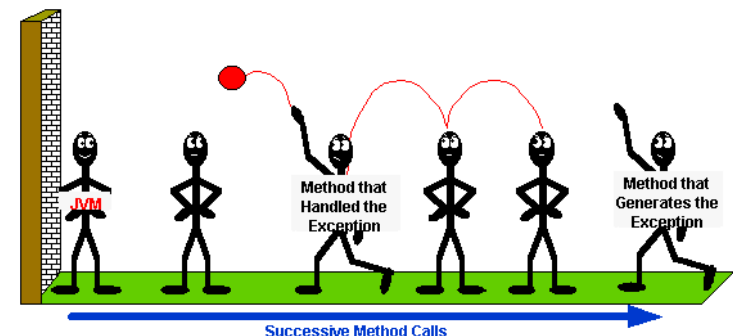
Exception

Fehler werfen statt sie in die Console schreiben...

Motivation

- Es gibt unterschiedliche Möglichkeiten mit dem Benutzer in Interaktion zu treten:
 - Console oder Windows Forms
Webanwendung oder MobilePhone
 - Je nach Oberfläche sollte im Fehlerfall ein Fehler anders dargestellt werden
- Wir werfen einen Fehler und die Oberfläche, die unsere Methode sichtlich mit falschen Werten befüllt hat, soll sich um diesen kümmern

THROW EXCEPTION!



Verwendung

- Klasse Exception kann einfach genutzt werden:
- `throw new Exception("Message");`

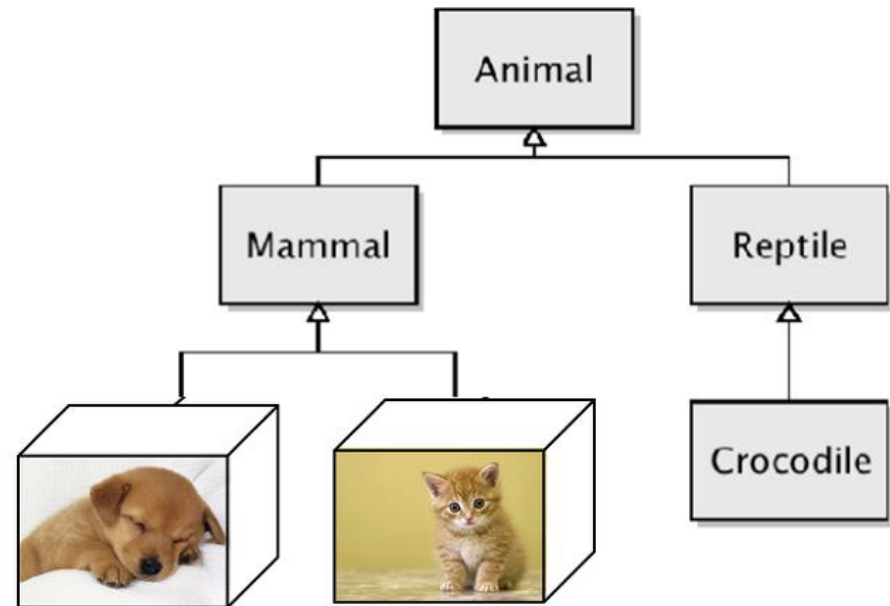
```
public static void ThrowExcpetion()  
{  
    string s = Console.ReadLine();  
    if (s != "Hallo")  
        throw new Exception("Ohne Begrüßen funktioniert das Programm nicht!");  
}
```


Exception fangen (catch)

- Try { something } catch (Exception) { ... }

```
public static void ThrowExcpetion()
{
    string s = Console.ReadLine();
    if (s != "Hallo")
        throw new Exception("Ohne Begrüßen funktioniert das Programm nicht!");
}

public static void TestException()
{
    try
    {
        //any Method which could throw an exception
        ThrowExcpetion();
    }
    catch (Exception e)
    {
        //handle the exception:
        Console.WriteLine(e.Message);
    }
}
```



Abstrakte Klassen

Dinge die es im echten Leben nicht gibt können als abstrakte Klassen implementiert werden:

Animal: Dog, Cat, Tiger, ...

Form: Circle, Triangle, Rectangle, ...

Abstrakte Klasse

- können nicht instantiiert werden
- stellen eine Basisklasse für weitere abgeleitete Klassen zur Verfügung
- stellen Attribute und Methoden für alle abgeleiteten Klassen zur Verfügung

```
public abstract class A
{
    // Class members here.
}
```

Abstrakte Klasse

- können auch abstrakte Elemente haben
- mit Hilfe des Schlüsselwort **abstract** vor dem Rückgabewert

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstrakte Klasse Tier:

- bei Abstrakten Klassen

wird ein A vor dem Klassennamen hinzugefügt:

- AAnimal
- AForm
- ABuilding
- ...

```
public abstract class AAnimal
{
    private int age = 3;
    1-Verweis
    public int Age
    {
        get { return age; }
        set {
            if (value > 0)
                age = value;
        }
    }

    0 Verweise
    public void PrintAge() ...

    0 Verweise
    public virtual void Running() ...
}
```

Abstract Class Members

- Abstrakte Methoden haben keine Implementierung
- Beinhalten Methodenkopf und Strichpunkt statt geschwungene Klammern.
- Abgeleitete Klassen müssen die abstrakten Methoden ausimplementieren
- Wenn eine abstrakte Klasse
 - eine virtuelle Methode der Basisklasse erbt
 - diese als abstrakte Methode zur Verfügung stellt
 - kann sie in einer weiteren abgeleiteten Klasse überschrieben werden
- Beispiel...

Virtual & Abstract Override

```
// compile with: /target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
    DoWork on class F cannot
    call DoWork on class D
}
```

Eine virtual-Methode der Basisklasse kann als abstracte Methode überschrieben werden. In diesem Fall muss eine weitere abgeleitete Klasse die Methode immer neu ausimplementieren!

Sealed Klassen

- Klassen die nicht mehr beerbt werden dürfen werden mit dem Schlüsselwort **sealed** versehen

```
public sealed class D
{
    // Class members here.
}
```

- Können nicht als Basisklasse genutzt werden
- Können nicht abstrakt sein
- Verhindern Vererbung

Sealed Class Members

- Abgeleitete Klassen können Sealed Eigenschaften beinhalten.

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

- **Syntax:** **sealed** Schlüsselwort vor das **override** Schlüsselwort setzen

Nur überschriebene Methoden können gesperrt werden für Vererbung

```
class SealedTestClass
{
    public override sealed String ToString()
    {
        return "Hallo";
    }

    public sealed void TestMethod()
    {
        Console.WriteLine("Hallo");
    }
}
```

Build List

Build Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense

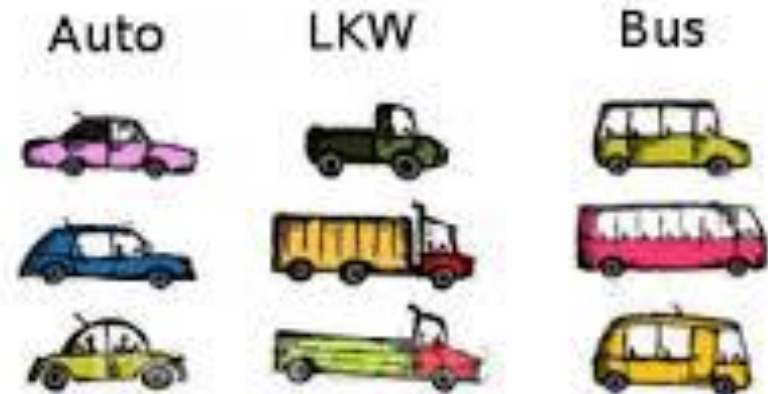
Code	Description	Project	File	Line	Suppression St...
CS0238	'SealedTestClass.TestMethod()' cannot be sealed because it is not an override	AbstractSealedTes	SealedTestClass.cs	16	Active



Angabe

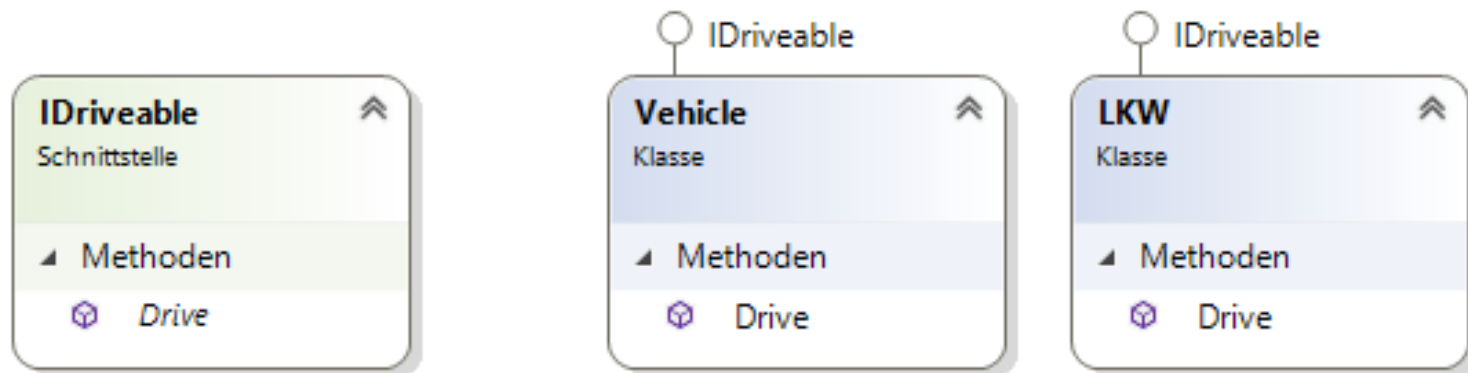
Interface & Enum &
Exception sowie

Abstrakte Klassen beliebig
kombiniert zum Üben:



Interface & Klassen

- Erstelle eine Klasse Vehicle und LKW, beide sollen die Methode Drive verfügbar haben.
- Interface IDriveable mit Methode Drive()



Interface implementieren:

```
public interface IDriveable
```

```
{  
    void Drive();  
}
```

```
class Vehicle : IDriveable
```

```
{  
    public void Drive()  
    {  
        Console.WriteLine("Das Fahrzeug fährt");  
    }  
}
```

```
class LKW : IDriveable
```

```
{  
    public void Drive()  
    {  
        Console.WriteLine("Der LKW fährt");  
    }  
}
```

Ausgabe:

Das Auto fährt schnell

Das Auto fährt

Das Auto fährt

Das Fahrzeug fährt

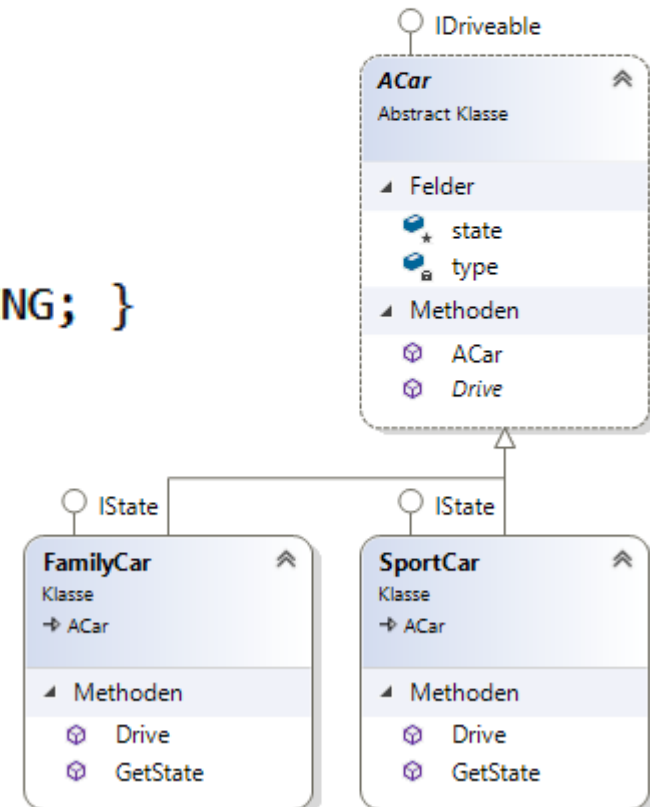
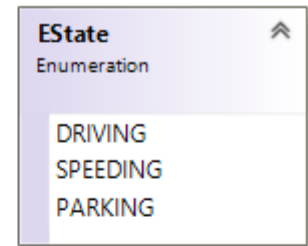
```
public static void TestDrivable()
```

```
{  
    IDriveable[] cars = new IDriveable[4];  
    cars[0] = new Vehicle();  
    cars[1] = new LKW();  
    cars[2] = new LKW();  
    cars[3] = new Vehicle();  
  
    foreach (IDriveable car in cars)  
    {  
        car.Drive();  
    }  
}
```

Abstrakte Klasse & Interface

- Erstelle eine Klasse FamilyCar, mit einer Type und einem Status: dieser kann sein Parken, Fahren, SchnellFahren.
- Erstelle eine Klasse SportCar
- Erstelle ein Interface, welches einen Status verfügbar macht - mit einer GetStatus().
- Erstelle eine abstrakte Klasse ACar, diese implementiert die Schnittstelle IDriveable mit der Methode Drive()
- Teste alle Varianten in statischen Methoden.
- Diskutiere den Unterschied zwischen Abstrakter Klasse und Interface sowie einer sinnvollen Kombination.
- Zeichne ein Klassendiagramm, implementiere die Klassen.

```
enum EState { DRIVING, SPEEDING, PARKING };  
interface IState  
{  
    EState GetState();  
}  
abstract class ACar : IDriveable  
{  
    string type;  
    protected EState state;  
  
    public ACar() { state = EState.PARKING; }  
    public abstract void Drive();  
}
```



Enum & Abstrakte Klasse

Erben und Interface implementieren

```
class FamilyCar : ACar, IState
{
    public override void Drive()...
    public EState GetState()...
}
```

```
class SportCar : ACar, IState
{
    public override void Drive()
    {
        base.state = EState.SPEEDING;
        Console.WriteLine("Das Auto fährt schnell");
    }

    public EState GetState()
    {
        return base.state;
    }
}
```

- Teste die Methoden GetState und Drive, welche Möglichkeiten gibt es diese Objekte in einem Array zu verwalten?

Testen von ACar

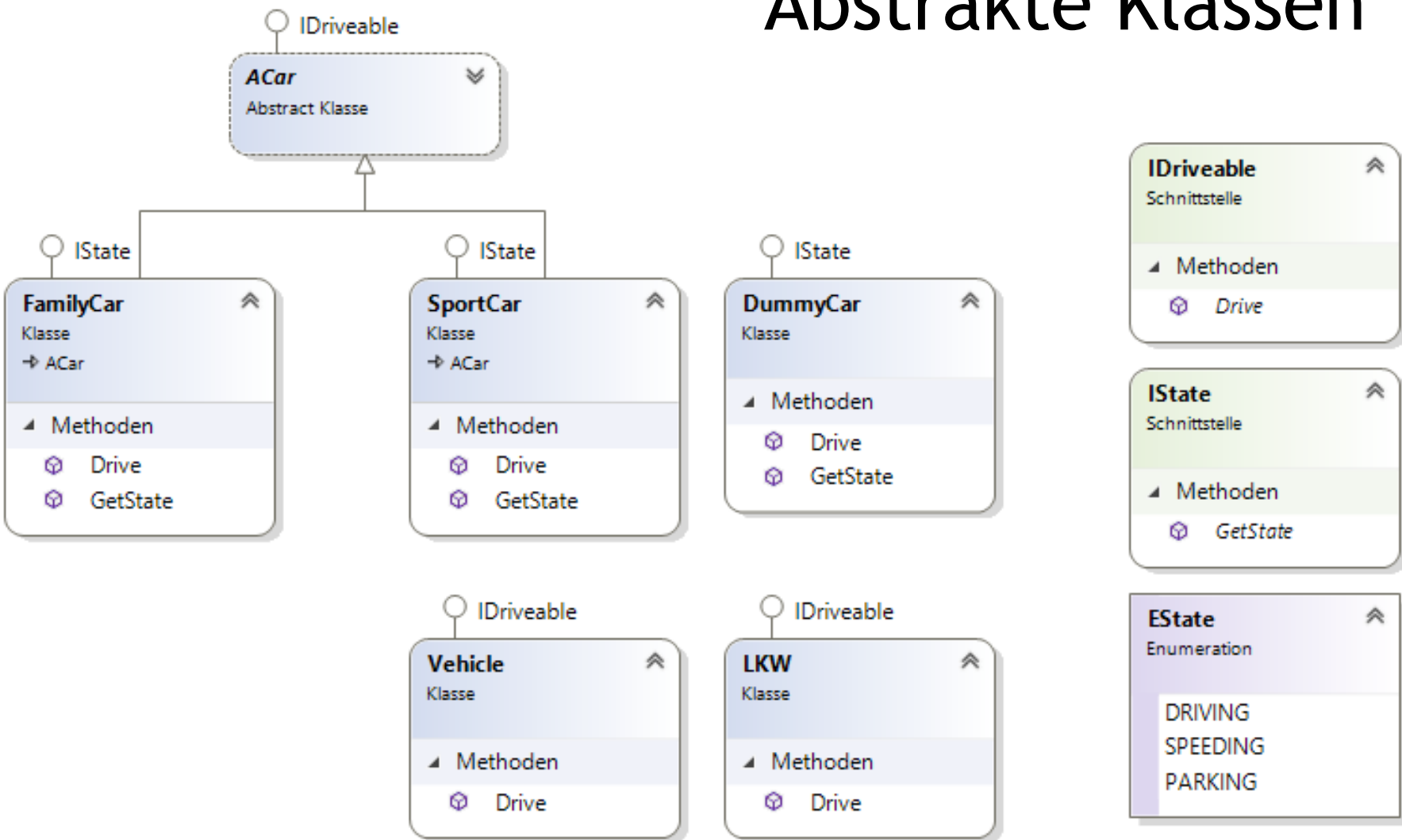
```
public static void TestACar()
{
    ACar[] cars = new ACar[4];
    ACar c = new FamilyCar();
    cars[0] = c;
    c.Drive();
    c = new SportCar();
    cars[1] = c;
    c.Drive();
    cars[2] = new SportCar();
    cars[3] = new FamilyCar();
    Console.WriteLine("-----");
    foreach (ACar car in cars)
    {
        car.Drive();
    }
}
```

Ausgabe:

Das Auto fährt
Das Auto fährt schnell

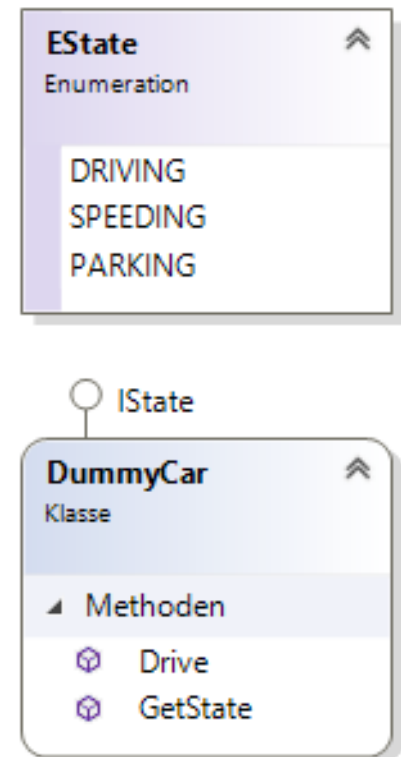
Das Auto fährt
Das Auto fährt schnell
Das Auto fährt schnell
Das Auto fährt

Interface & Abstrakte Klassen



Interface IState & Exception

- Erstelle eine Klasse DummyCar, diese soll ein Auto dass nicht fährt darstellen.
- Schreibe die Methode Drive und wirf eine Exception falls sie aufgerufen wird.



DummyCar

```
class DummyCar : IState
{
    public void Drive()
    {
        throw new Exception("Dummy Cars können nicht fahren.");
    }

    public EState GetState()
    {
        return EState.PARKING;
    }
}
```

Ausgabe:
Dummy Cars können nicht fahren.

```
enum EState { DRIVING, SPEEDING, PARKING };
interface IState
{
    EState GetState();
}
```

```
public static void TestIState()
{
    try
    {
        IState c;
        c = new DummyCar();
        DummyCar dc = c as DummyCar;
        dc.Drive();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Teste IState

```
public static void TestAllIStates()
{
    IState[] cars = new IState[5];
    cars[0] = new FamilyCar();
    cars[1] = new SportCar();
    cars[2] = new DummyCar();
    FamilyCar fc = new FamilyCar();
    fc.Drive();
    cars[3] = fc;
    SportCar sc = new SportCar();
    sc.Drive();
    cars[4] = sc;

    foreach (IState car in cars)
    {
        Console.WriteLine(car.GetState());
    }
}
```

Ausgabe:

Das Auto fährt

Das Auto fährt schnell

PARKING

PARKING

PARKING

DRIVING

SPEEDING

Parkplatz:

- Ergänze IState um einen weiteren Wert für einen freien Parkplatz, der im DummyCar gesetzt wird.
- Erstelle ein zweidimensionales Array und verwalte hier das Parken von Autos und freien Parkplätzen.
- Erstelle eine Klasse Parkplatz mit Einparken(ACar) und Ausparken(ACar).
- Freien Parkplätzen wird DummyCar zugewiesen.
- Überschreibe die ToString-Methode von Parkplatz.